



RobMoSys

H2020-ICT-732410

RobMoSys

**Composable Models and Software
for Robotics Systems**

Deliverable D3.5:

**Full documentation of motion, perception and
world model stacks**



This project has received funding from the *European Union's Horizon 2020* research and innovation programme under grant agreement N732410.



RobMoSys



Project acronym:	RobMoSys
Project full title:	Composable Models and Software for Robotics Systems
Work Package:	WP 3
Document number:	D3.5
Document title:	Full documentation of motion, perception and world model stacks
Version:	1.0
Delivery date:	December 31st, 2020
Nature:	Report (R)
Dissemination level:	Public (PU)
Editor:	Herman Bruyninckx (KUL)
Authors:	Herman Bruyninckx (KUL), Marco Frigerio (KUL)
Reviewer:	

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under grant agreement N°732410 RobMoSys.

Contents

1	Selected contributions	4
1.1	Coordination	4
1.2	Situational-aware monitoring and configuration	5
1.3	“Realtime” component architectures	5
1.4	Association hierarchies for perception and control	5
1.5	Best Practices from outside the RobMoSys project	5
A	Full online version as <i>living document</i> of this Deliverable	7

1. Selected contributions

This Deliverable is a continuation and (very large) extension of the earlier Deliverables in Work Package 3. As for earlier Deliverables, the KU Leuven team has kept the first Deliverable as a *living document*, which was continuously improved and extended. About two years ago, it was also made publicly available here:

<https://robmosys.pages.gitlab.kuleuven.be/>

using a title (and subtitles) that present it as a standalone book, with a lifetime and focus that go beyond the mere objectives of a Deliverable in a research project of finite duration. At the time of writing this Deliverable D3.5, the online document has already over 350 pages, and has received inputs from outside the RobMoSys consortium, from various robot software developers that have used it as a guide to help them with making composable software components.

This version D3.5 of the RobMoSys Deliverable *includes* that public document as an inherent part (in Appendix) but complements it with a summary of the “highlights” of the last reporting period. The RobMoSys project has three complementary aims: (i) to use formal **models** as the basis for software, (ii) to use the structure provided by the models to improve the **composability** of software components, and (iii) to develop the **tools** to help developers write such software components and compose them together in systems. The aim is *not* to provide a *software framework* for robotics; but the solid foundations behind *any* such software framework that practitioners want to build. The software that is provided (see Deliverable D3.4) comes in the form of **reference implementations**, that illustrate the RobMoSys designs and architectures, but that are not stand-alone, reusable software components in themselves. However, their design is such that they are ready to be integrated in whatever component framework wrapper one chooses. Because of this ambition to be highly efficient and embeddable, the implementations use the C language ecosystem.

The following sub-sections are **just a selection** of the much richer material in the [online document](#), with a focus on the most mature parts, as well as the ones that have been added since the previous version of this Deliverable. Work Package 2 has made a significant evolution towards more explicit modelling of **inter-dependencies** between software components, and, not coincidentally of course, such dependency models (and software to work with the dependencies) are also the major focus of this WP3 Deliverable.

1.1 Coordination

We have brought much more structure and explanations about the coordination of the execution of multiple algorithms, activities, processes and components. The following **three mechanisms** represent complementary purposes and best practices:

- **bitfield flags**: to encode **protocols** via which two or more components organise their execution dependencies.
- **Petri Nets**: to coordinate the **execution of multiple algorithms**.
- **Finite State Machines**: to coordinate the **behaviour of one single component**.

1.2 Situational-aware monitoring and configuration

RobMoSys targets robotics applications that must survive in unstructured environments and contexts, and hence the concepts of *situational awareness* and *behaviour/execution monitoring* are extremely important. Both design targets are being dealt with systematically in all best practices and architectural patterns where they are relevant.

For (monitoring-based) configuration, the most important newly introduced concept and meta model is that of the *Document Object Model*, see Sec. 1.5. It is an “imported” best practice meta model for the **dependencies** between the (re)configurations of multiple components.

1.3 “Realtime” component architectures

We now have a complete set of architectural patterns to make highly efficient component implementations, that can even be deployed in hard realtime contexts, taking into account all **dependencies** between executions that influence realtime performance.

1.4 Association hierarchies for perception and control

This is probably the **major result** of the last reporting period that is 100% robotics-centric *and* innovative. These meta models allow **explainable compositions** of all aspects of perception and motion control. Even to the extent that “brute force” and “black box” approaches in perception and control, such as deep learning or reinforcement learning, can be integrated seamlessly. (But not without efforts!)

1.5 Best Practices from outside the RobMoSys project

It is an **explicit objective** of the RobMoSys project to avoid reinventing good practices and meta models that have been created and matured in other domains than robotics. This Section gives a short summary of the major *too good to miss* external developments (full *eco systems* that is, not just small individual software projects) that we have identified and that can have an extremely large impact in robotics. We refer to the [online document](#) the details about why these developments fit so well to the RobMoSys approach.

- [Apache Arrow](#): formal meta model of complex data structures, with software bindings in a large number of programming languages. Arrow can be used in zero-copying, in-memory contexts, but also as a message format for communication, and for storage in files.
- [WebAssembly](#): there is finally hope for extreme portability of source code, even written in different programming languages, by this project that is driven by all big international “platform” players. Legacy code can be recompiled in WebAssembly compatible binaries. Awareness for digital security is a major design driver behind the WebAssembly ecosystem.
- [W3C Document Object Model](#): introduces tree-structured hierarchies for configuration and event handling. While originating in text and web documents, the most recent standardization targets any type of software context. Surprisingly, the same standardization effort also integrates models that are highly relevant for our “world modelling” ambitions, namely

via the extension of the SVG (Scalable Vector Graphics) format towards (i) 3D, and (ii) symbolic representations of points and their geometric compositions.

- [Lockfree circular buffers](#): this was the source of inspiration for a lot of our new best practices in highly efficient data flow architectures.
- [JSON-LD](#), [ShEx](#) and [ShaCL](#): the ecosystem around standardisation of formal models of property graphs and constraint languages is maturing quickly, and all of its developments also bring huge opportunities to robotics, for meta meta modelling, and the tooling that is required to give streamlined access to “reasoning” with such models.
- [realtime](#) architectures, in the [Linux](#) ecosystem.

A. Full online version as *living document* of this Deliverable

The following 350+ pages are a copy of the current version of the [online document](#), which contains (and will continue to contain) the consolidated and integrated description of all best practices, models, architectural patterns, . . . that RobMoSys has created, in its specific focus on building blocks for “realtime” software components.

Design of **Complicated** Systems with **Situational Awareness**

Composable components for compositional,
adaptive, and explainable systems-of-systems

— a.k.a. —

Knowledge-based modelling and best practices to build resilient holonic
data, information, task and software architectures
for robotics and other cyber-physical systems

“It’s simple, though not easy. . . ”

Herman Bruyninckx

(KU Leuven, Belgium; TU Eindhoven, the Netherlands)

with contributions and ideas¹ by

Enea Scioni, Nico Hübel, Johan Philips, Filip Reniers, Davide Monari, Marco Frigerio,
Sergio Portoles Diez, Sander Van Driessche, Nikoloas Tsiogkas (KU Leuven, Belgium)

Christian Schlegel,² Dennis Stampfer, Alex Lotz (HSU Ulm, Germany)

René van de Molengraft, Cesar Lopez Martinez, Wouter Houtman,
Hao Liang Chen, Bob Hendrikx, Jordy Senden (TU Eindhoven, the Netherlands)

This document is a proud part of the **RobMoSys** ecosystem,³
stimulated by the European Commission under its *Horizon 2020* programme
(2017–2020, grant agreement No 732410)

2020-11-21

¹HB remains responsible for erroneous information and claims in this document.

²Christian is to be credited for the creation and maturation of many, many of the concepts, models, patterns and best practices in this book. He is also the “founding father” of, and driving force behind, the RobMoSys ecosystem.

³This document is copyrighted by HB, and released under the **Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)** license.

Executive summary

This document introduces a **paradigm** to design **components** (that reflect reality and control that reality via computers) so that they are **composable** (their behaviour is predictable in any system they are part of), such that they can be **composed** (structurally and behaviourally) into **systems** that are **compositional** (their behaviour is predictable when their components are).

It is difficult to be more vague and unprecise than in the buzzword-loaden sentence above. The concepts of a *component* or a *system* are so general and broad, that they can (and need) not be defined strictly or exhaustively, [1, 94]. Anyway, both have very few identifiable properties that are present all the time. For a *component*, these meta properties are: *computation*, *communication*, *configuration* and *coordination*. For a *system* there is just one: *composition*. A major quality measurement of such system composition is its *compositionality*: the extent to which a system's behaviour is *predictable* when (i) the behaviour of its sub-systems is known, and (ii) the interactions between the sub-systems are known. In addition, that predictability must be preserved even when (i) the number and complexity of the interactions between sub-systems increases, (ii) each interaction introduces its own type of uncertainty, ambiguity, or **inconsistency**, and (iii) any interaction can make one sub-system disturb the behaviour of the other sub-systems involved in the interaction.

Most modern systems are built in a **modular** way, that is, they are the *composition* of several modules. *Module developers* are trained to make components with **composability** in mind: a component that is ready to become part of a system is not worth investing in. However, the training of module developers seldom goes that far that their components behave in such predictable ways that the composite system in which they live, can *itself* be integrated into a larger “whole”, while *preserving* the composability property.

System architects must work closely with module developers, but they are not just responsible for creating “systems that work”, but systems that have the **compositional** property: the quality and behaviour of the system is **predictable** from the quality and behaviour of its components. This is a very simple and qualitative requirement to state, but one with far-reaching consequences.

This document focuses on reaching compositionality for **engineered systems**, more

in particular, for **cyber-physical systems**,¹ and, even more specifically, on **robotic systems**. The purpose of cyber-physical and robotics systems is to **interact** with the **physical world**, and **to control** the behaviour of that interaction.

Systems and **systems theory** apply to social, economical, cultural contexts, and many more. In the societal context, composability and compositionality of systems have, over the centuries, become a primary design property,² *because* society needs solutions that are **robust** and even **resilient** against a large number of disturbances. In the “engineering” contexts however, most focus has been on **modularity**, that is, to make components that can be reused in different system compositions. However, modularity has become too much of a goal in itself, to the detriment of (i) composability, and even worse, of (ii) compositionality.

¹The major difference with “purely digital” ICT platforms (e.g., distributed financial databases, social media applications, e-commerce platforms) is that CPS directly impact (that is, “control”) the real physical world. This brings in a lot of extra constraints on its ICT components, more in particular the need for real-time feedback loops, which include physical components that come with a dynamical behaviour that has not been designed by the system engineers.

²Unfortunately, most politicians have forgotten *why* some systems are composable or compositional, and create new systems that fail miserably, run far over budget, are not predictable, hence also not manageable, etc.

Overview of this document

Robotic systems are³ cyber-physical systems that (are expected to) **decide for themselves**, about *how*, *when*, and *how well*, to realise their own behaviour, as well as their interactions. Robots, like all *cyber-physical systems*, move, manipulate, transform and exchange **matter**, **energy** and **data**, but, like all *systems*, also more and more **information**, and, somewhere in the future, **knowledge** too.

Information is data, with part of that data being tagged as **metadata**, Fig. 8. That is, data whose semantics is formally described in **meta models** that represent, in a formal, computer-interpretable way, the relations and constraints that have to be satisfied by the data.

Knowledge is a set of relations that **explain** how and why identified pieces of information are connected. So, the semantic tag often given to knowledge representation is **meta meta models**.⁴

The *purpose* of this document is to present the **least amount** of **models**, (that is, **formally encoded knowledge**)⁵ needed **to represent** the **structure and behaviour** of **all** possible **systems**. The *ambition* of this document is to become the basis for **knowledge-based systems**: the (not yet existing) ones that use the knowledge representations **to explain** their own behaviour, nominal or faulty, **to adapt** their behaviour on their own, and, hence, to become **resilient** to disturbances from the real, **open world**.

This document's **paradigm** is that one needs only **few meta meta models**, to link **all** pieces of knowledge together, for **all** kinds of systems, by means of simple **structural and behavioural relations**. The paragraphs above already introduced a first set of structural relations:

- system, cyber-physical system, robotic system.
- wisdom, knowledge, information, data.

This document will introduce more in each of the three system “levels”, to allow system designers **to implement** their systems in such a way that they can use the

³Just by *definition* of the term, because there is no need to define the strict boundary where a cyber-physical system becomes a robotic system, and why that would be the right border to consider.

⁴A *meta model* is not a model of something in the real world itself, but it contains the constraints that any such concrete model must satisfy. A *meta meta model* is even one level of abstraction higher: it represents the relations that express the commonalities between two or more meta models. See Sec. 1.2.16 for more details.

⁵This is the “*simple*” part in the motto of this document: each knowledge model should be clear enough to understand and comprehend fully in half an hour.

formally represented knowledge **to control themselves, autonomously**, and in **predictable, resilient** and **explainable** ways,⁶ and still be **re-composable** in any type of **system architecture**.

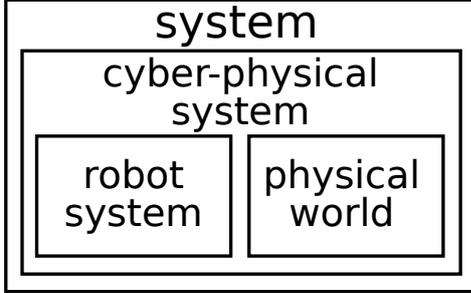


Figure 1: The **Systems hierarchy paradigm**: many aspects of computer-controlled (“cyber”) systems design do not depend on any robotics-specific properties, but hold for all designs in which the **physical** world influences the system behaviour. Similarly, many aspects of computer-controlled **systems** design do not depend on whether or not sub-systems are controlled by a computer.

The most generic **level of abstraction** is the **mereo-topological** one. *Mereology* represents *parts-whole* relations between entities. *Topology* represents *interconnections* between entities. Together, they are the *minimum* representations via which one can discuss *systems*.

This document’s “zeroth” paradigmatic foundation for modelling is:

0. **Property graphs** and the **knowledge model primitives** built on top of them, Fig. 2. The property graph is the **mechanism** to represent symbolic knowledge as an **abstract data type** suitable for computer processing. Each knowledge model is a **policy** of how to use that mechanism for representations of “higher-order relations”; the most generic such higher-order relation introduces (i) a *constraint* relation between the properties of a relation and some of its arguments, and (ii) a *tolerance* relation as an expression of how strict a constraint is.

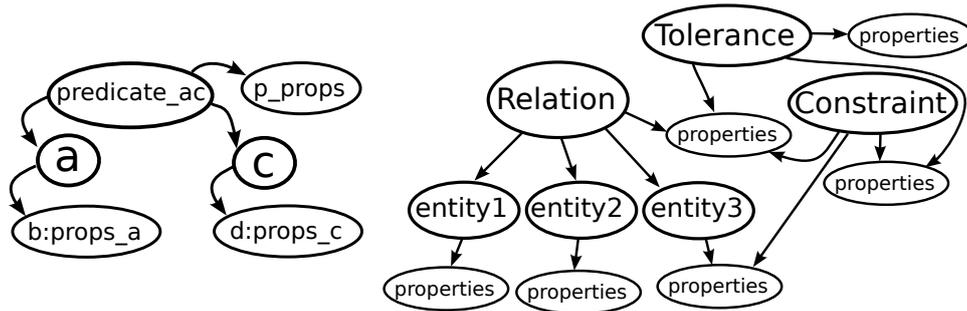


Figure 2: The property graph (left; fully domain-independent *mechanism*) and primitive knowledge relations (right; a domain-specific *policy* of using the property graph mechanism), are *directed acyclic graph* representations of knowledge.

Three other paradigmatic modelling foundations hold for all cyber-physical systems:⁷

⁶This is the “not easy” part in the motto of this document: it takes a lot of effort and iterations to get the envisaged integration of many simple things right, that is, correct, effective and efficient.

⁷The corresponding figures represent the models at only their mereo-topological level.

1. **Systems hierarchy**, Fig. 1. It provides structure that system architects can follow to make their designs as independent of the application context as possible, by assigning system properties and design decisions to the highest relevant level in the hierarchy.
2. **“5C” composition hierarchy**, Fig. 3. It provides constraints on all *computer-controlled* system architectures, irrespective of the specific properties of the specific sub-systems involved.

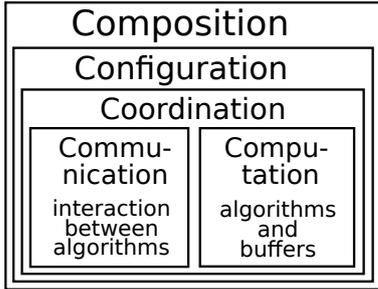


Figure 3: The **“5C” composition paradigm** for **software** system architectures: the functional core of every system consists of lots of “primitive” components of *computations* that must *communicate* with each other; what computations have to execute when, and when and what they have to communicate, must all be *coordinated*; these computations and communications must be *configured*, with the right “*magic numbers*” in the right context; this context is provided by the right *composition* architecture.

3. **Hybrid constraint-driven specification and control**, Fig. 5. Every desired behaviour of a system can be computed as the solution of a **hybrid** series of constraint-driven computation problems. The “hybrid” adjective reflects the necessity for a control system to switch between different hypotheses about how to control the system. *When* (or, rather, *why*) to switch between “*behaviours*”, is decided by *monitoring* several *constraints*. A popular special case of constraint-driven is **constrained optimization**: every control action is the result of finding the *optimal* correspondence (or rather, “*adequate*”, or “*good enough*”), between, on the one hand, the hypotheses that underly the optimization problem, and, on the other hand, the sensor and actuator data.

task state & domain	$X \in \mathcal{D}$
desired task state	X_d
robot/actuator state & domain	$q \in \mathcal{Q}$
perception/sensor state & domain	$s \in \mathcal{S}$
driver relation (\rightarrow computes how to “move”)	$f(X, X_d, q, s) = 0$
equality constraints	$g(X, q, s) = 0$
inequality constraints	$h(X, q, s) \leq 0$
tolerances	$d(X, X_d) \leq A$
monitors	compute state of constraints to react to: $m(X, q, s) \geq B$
reactive solver	algorithm to switch driver function and constraints to react to

Figure 4: Generic formulation of a *hybrid reactive problem* (HRP).

The last two paradigmatic modelling foundations make sense only for robotic systems:

task state & domain	$X \in \mathcal{D}$
desired task state	X_d
robot/actuator state & domain	$q \in \mathcal{Q}$
perception/sensor state & domain	$s \in \mathcal{S}$
objective function	$\min_q f(X, X_d, q, s)$
equality constraints	$g(X, q, s) = 0$
inequality constraints	$h(X, q, s) \leq 0$
tolerances	$d(X, X_d) \leq A$
solver	algorithm computes q and/or s
monitors	compute state of
	switching constraints: $m(X, q, s) \geq B$

Figure 5: Generic formulation of a *hybrid constrained optimization problem* (HCOP), which is a special case of *hybrid reactive problem* (Fig. 4).

4. **Association hierarchy**, Fig. 6. It describes the “*hypotheses-to-data*” associations that every robot system has to make, at all levels of decision making.

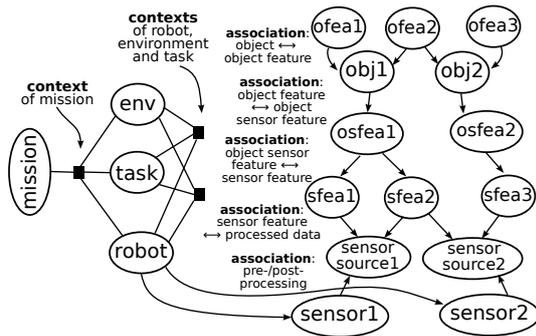


Figure 6: The components underlying the **association hierarchy paradigm**: the natural hierarchy of dependencies between the models of all components that inevitably appear in robot systems, and that force the system designers to make various *association* choices, that is, which *hypotheses* to use, at which moment, to explain or to generate *data*.

Figure 6 shows the second, complementary robotics meta meta model, that of how the above-mentioned behavioural interactions take place at various **levels of modelling the world**:

- **actuators** and **sensors** link a robot to the physical reality. They form the “smallest world” in which one can still talk about a “robot”.
- the next larger view on the world involves models of the relevant **objects** in the real world, containing the information about those models that must be linked to the sensors and actuators,
- the next larger view of the world brings these objects in relation to (i) the **robot** itself, (ii) other objects in the **environment**, and (iii) the **task** requirements.
- an individual robot’s individual task is always just a part of the largest view on the world, that of the **mission** that a human user wants to realise, using “flocks”⁸ and “fleets”⁹ of robots.

5. **Task-Skill-Resource relations**,¹⁰ Fig. 7. They connect the “*what*”, the “*why*” and the “*how*”, in that order, of a robot that must execute a task.

⁸The set of robots that are involved in the same task execution.

⁹The set of robots that are available to form fleets when needed by a task execution scheduling system.

¹⁰This paradigm has been developed by René van de Molengraft and Herman Bruyninckx.

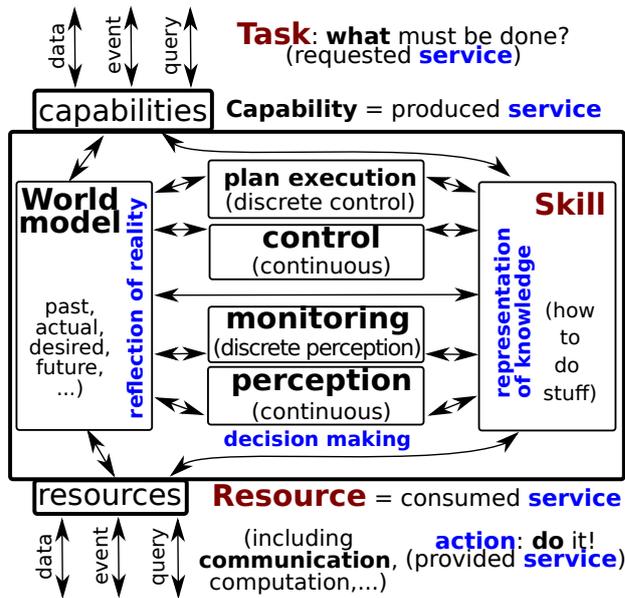


Figure 7: The components underlying the **Task-Skill-Resource paradigm** for robot control: the arrows and the rectangles represent the composition primitives of *interconnection* and *containment*. The figure *does not* represent a software component, but the *structural constraints* via which the parts are to be composed into a system.

Figure 7 depicts the mereo-topological model of a robotic system, that links the robot’s **task** (the *symbolic input* to the system of *what* the robot must do) to its **motion** (the *continuous output* to the motors, doing what is asked for), via the knowledge of *how* to realise the task with the available resources, as encoded in a **skill**. This document gives the name **skill** to the knowledge about the robot application that is used to make the right decisions at the right time, turning the **symbolic task** requirements into **physical motions** of the robot. A skill materialises as the set of **interactions** between **activities** (the hardware and software that computes, reasons, decides, communicates,...) that must realise the following responsibilities:

- **discrete and continuous control**: the decision making activities, to determine which behaviour the robot must realise, at every moment.
- **discrete and continuous perception**: the activities that process the raw data from sensors, and interpret them in the context provided by the task description.
- **state of the world** (or “**world model**”): the only activity that *couple*s the other activities, by storing and exchanging all types of **state** information, that the other activities read and/or write.

Three of the meta meta models provide **hierarchical** structure, that help with the “**vertical**” integration of relations: the vertical relations provide (the potential for) orders of magnitude speed-up in the **reasoning** to connect two or more entities in a **knowledge graph**, if the **reasoner** can find the level with the “best” **meta relations** to follow. **Flat** structure is provided in the *Task-Skill-Resource* meta meta model, and in the relations between *computations* and *their interactions* in the *Systems* model: they help the reasoners with their “**horizontal**” integration challenges.

Figures 8–11 show more specific natural hierarchies in the abstraction levels of, respectively, system, cyber-physical systems, and robotic systems.

Other **natural association hierarchies** (or “**stacks**”) have become commonplace in several **domains** already. For example: the **OSI stack** for communication networks; the **QUDT** stack of quantities, units, dimensions and types; or the **hyponymy and hypernymy, taxonomy** (that is, based on **is-a** relations) and **ontology** (that is, based on **conforms-to** relations)

Knowledge	Graph	Mereotopology (part-of, connects-to)	Conforms-to	Is-a	model
wisdom: goal-directed application of knowledge	knowledge graph (domain-specific entities + relations)	graph	meta meta model	super class	tolerance
knowledge: models with context	property graph (entity+property nodes, list incoming/outgoing edges)	tree	meta model	class	constraint
model: information with constraints	relation (subject, predicate, object)	list	model	sub class	relation
information: data & meta data	mathematics (vertex, edge)	entity	realisation	instance	entity

Figure 8: Some natural hierarchies in **systems** meta meta models.

relations on the meaning of words.

Physical system	Control
mechanical, chemical, electrical, thermal,...	preview
storage, dissipation	adaptation
transformation, gyration	control, estimation (prediction, correction)
source, sink, transport	error (desired, actual)
energy	data, signal

Figure 9: Some natural hierarchies in **physical systems** meta meta models.

Model	Language	Software	Hardware	Open standard
mathematics	story	process	cloud (network)	platforms
abstract data type	semantics	thread	fog/edge (IPC bus)	tools
data structure	syntax	event loop	computer (on-chip bus)	software
digital	vocabulary	activity	CPU, GPU, FPGA	protocols
electronic	alphabet	algorithm	core	models
		data, function		data

Figure 10: Some natural hierarchies in **cyber systems** meta meta models.

The rest of this document serves to explain the meta meta models introduced above, and to connect their **knowledge** relations to systems that can **act**. The latter is realised via **composable architectures** on **data, information, tasks, hardware, and software**. These are the *technical* aspects of systems engineering, but realising **sustainable applications in domain ecosystems** involves several *non-technical* “architectures” too: **stakeholder** interactions, **license** compatibilities, **intellectual property** agreements, and **authentication & authorisation protocols**.

This document’s approach to *architectures* is inspired by the **best practices** that can be found in successful and resilient societies and organisations, The ones created by humanity differentiate themselves from the ones created by nature, by adding to the latter’s **continuous** system components also **symbolic** and **discrete** components, to formalise, store and

Application	Robot	Motion
mission	fleet, factory	move-constrained
robot, task, environment	flock, cell, line	move-to move-guarded move
objects	platform	position velocity
sensors actuators	device component	acceleration torque current

Geometry	Shape	Environment	Building
Euclidean (metric)	R-tree simplex	jungle, crowd cattle, traffic	building aile, wing
affine (parallel)	multi-polygon polygon	pet, co-worker obstacle	floor ward, corridor
projective (incidence ratio)	polyline point	disturbance failure	room, area wall, floor, ceiling

Figure 11: Some natural hierarchies in robotic systems meta meta models.

distribute vast amounts of knowledge and decision making in **physics**, **mathematics**, **computer science**, and **systems** and **control** theory.

Because of the extra complexity caused by their being **autonomous decision makers**, **robotic systems** are the primary application target. In that context, **the simplest robotic system** that this document considers **worth studying** is one:

- with *multiple* robots,
- each executing *multiple* tasks at the same time,
- in *cooperation* with the other robots,
- with whom they share resources,
- and *interact physically* with an environment,
- that is itself in a *continuous state of change*.

Obviously, such behaviour can only be realised by means of control architectures with

- multiple asynchronous software activities,
- executed on multiple, distributed and communicating computational platforms.

This document advocates, strongly, the core **methodology** (or **paradigm**) to link models to specify actions of the system as a **hybrid reactive problem** (HRP), Fig. 4. A particular specialisation is the **hybrid constrained optimization problem** formulation (HCOP, Fig. 5. The first step in the hybrid optimization methodology is to construct a *description* of the to-be-executed activities by means of (i) **objective functions** to optimize and (ii) **constraints** to satisfy. Objective functions and constraints are of three complementary types:

- **symbolic constraint satisfaction**, that is, **reasoning** on the “knowledge relations” that populate the application’s **context**.
- **continuous** constrained optimization, that work in the “metric” domains representing the physical world, such as time, space, force, energy,...

- **discrete** optimization: the “scheduling” of which combination of specifications to solve under which conditions.

The second step of the methodology is *to solve* the problem, with as outcome (i) the “best” actuation setpoints to apply to the system, and (ii) the model that “best” explains the sensor data. Few application contexts *require* that the executed action is indeed the most optimal that exists; most are happy with a **satisfactory** solution [91, 98], not in the least because that is the best one can achieve under **bounded rationality**: any effort to reach a solution requires resources, and using resources implies cost. In addition, a solution need also not be completely computed before the system is allowed to start acting: any *feasible* solution that is available can already be used to get the system started. Not in the least because only *actions* in the real world (and not optimizations on a symbolic model of that world) can help the system controller to assess how well it is realising its objectives. In addition, there is typically enough time *during the system’s operation* to run further iterations of the solver towards more optimal/satisficing outcomes.

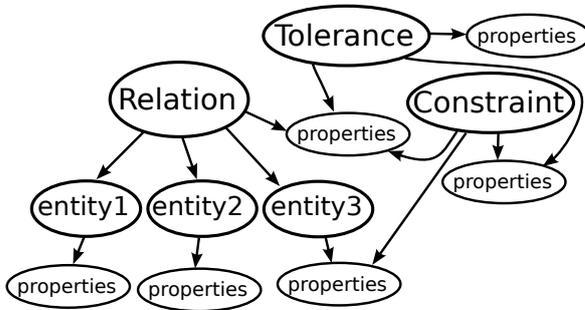


Figure 12: A property graph representation of the most fundamental higher-order model in this document: the *constraint* between the properties of a relation and one of its arguments is a higher-order relation, with its own properties. Similarly, the *tolerance* relation is an even higher-order relation, too.

The previous Sections speak a lot about *knowledge representation*. This document adopts **property graphs** as the **mechanism** to represent such complex **data structure**. Figure 12 depicts a property graph data structure, *with* the (**paradigmatic**) semantics of the core **knowledge modelling primitive** in this document: the extension to the **entity-relation** model with (i) **constraints** on the properties of entities and/or relations, and (ii) **tolerances** on how strictly these constraints must be realised in a concrete system instantiation.

Every particular integration of activities and behaviour into a robotic system requires an **information architecture**: a model of the interactions between activities, via which to realise the components’ interactions, in such a way that they *conform to*¹¹ those in both *meta meta models*. This document explains how to use information architecture models to build complex **holonic software architectures**, with a set of **design patterns** and **best practices**, around the fundamental primitives of **activity**, **event loop**, and **stream buffer**, and with **explainability**, **resilience** and **runtime configurability** as main system design drivers. With the above-mentioned core models and solver methodology, the core of **system design** is then to combine them and create **stable subsystems** (sometimes referred to as “**holons**”) [60]: a subsystem is called “stable” if it provides a “good enough” trade-off between:

- the **quality of the services** it delivers to the application.

¹¹A model *conforms to* a meta model, if all the relations in the model that are represented in the meta model, satisfy the relational constraints expressed in that meta model. One model can conform to more than one meta model.

- the **use of resources** it requires to provide those services.
- its **robustness** against the disturbances that the application’s context will bring to the system.

The paradigmatic core primitives of this document’s **software architectures** are:

- ring buffer:
- event loop:
- activity:

This document makes concrete suggestions about how to turn the state-of-the-art insights into a concrete set of (meta) models, on which to base any concrete implementation for any concrete application in robotics. The focus is on building a **digital platform**, so, a lot of attention goes to creating the “right” modularity, the “right” levels of detail, the “right” separation of concerns, and the “right” approach towards *composability*. With those “right” choices, the development of models, tools, implementations and applications becomes methodological, transparent and scalable. Generic vendor-neutral platforms stimulate the pre-competitive cooperation on the generic parts of the digital platform, as well as the competitive exploitation of that platform for innovative robotic applications.

The efforts required to create a *model-driven engineering* development work flow pay off only *after* those developments have reached a state in which the models contain not only the information about *what* the system does and about *how* it should do it, but also about *why* it should do these things. Indeed, only when the latter information is available, at runtime and in formal representation, one can expect robots **to reason** about their actions: **to explain** what they are doing, to interpret whether what they are doing corresponds to what they are supposed to do in the **task**, and to adapt their action plan accordingly.

Contents

Executive summary	2
Overview of this document	4
1 Foundations of knowledge-based systems: ((meta) meta) modelling	24
1.1 Models to represent knowledge in science and engineering	25
1.1.1 Science reflects reality — Engineering reflects decision making	25
1.1.2 Levels of abstractions: scope of relevant entities and relations	26
1.1.3 Levels of modelling formalization	27
1.1.4 Levels of decision making: context of intentions	27
1.1.5 The meaning of “meta”	28
1.2 Representation of knowledge: graph models	28
1.2.1 Paradigm, pattern, best practice	29
1.2.2 Simple, though not easy	29
1.2.3 Mechanism: graphs (RDF, property, entity-relation, factor)	29
1.2.4 Policy: first-order relation as an entity-relation graph	31
1.2.5 Policy: higher-order relation as an entity-relation graph	32
1.2.6 Observation: “higher-order” is not necessarily “more abstract”	33
1.2.7 Facts	33
1.2.8 Properties and attributes	33
1.2.9 Magic number: attribute in a higher-order causality relation	34
1.2.10 Reification — Best practice of <i>attributes are properties of relations</i>	34
1.2.11 Pattern: entities–relation–constraints–tolerances	35
1.2.12 Pattern: collection composes entities in bag, set, list, tree, graph	35
1.2.13 Pattern: structural constraints on collections as array, dictionary	35
1.2.14 Pattern: behavioural constraints on collections as queue, stack	35
1.2.15 Pattern: composition of relations as fact, model, instantiation	36
1.2.16 Best practice: meta modelling with M0–M3 structure	36
1.2.17 Policy: metadata of an entity inside a model — <code>Semantic_ID</code>	38
1.2.18 Examples and types of higher-order relations	39
1.2.19 Mechanism: querying & reasoning via graph matching & traversal	41
1.2.20 Storage and reasoning in property graph databases	42
1.2.21 Host languages to store, exchange and query models	43
1.3 Core relations: <code>conforms-to</code> and <code>is-a</code>	45
1.3.1 <code>conforms-to</code> hierarchy on relations	45
1.3.2 <code>is-a</code> hierarchy on properties	46

1.3.3	Composition and inheritance	46
1.3.4	Best practice: four levels in is-a and conforms-to hierarchies	47
1.4	Mereology: most abstract representation level	47
1.4.1	Mereology: has-a	47
1.4.2	Topology: contains, connects	48
1.4.3	Mechanism: block-port-connector	49
1.4.4	Policy on block-port-connector mechanism: data sheets	50
1.4.5	Role of mereo-topological models	51
1.5	Design patterns	51
1.5.1	Policy: constraint, dependency and causality graphs	52
1.5.2	Mechanism: Directed Acyclic Graphs for partial ordering	52
1.5.3	Policy: temporal order, hierarchy, causality	52
1.6	Declarative and imperative models of behaviour	53
1.7	Hierarchical and serial ordering: scope	53
2	Meta models for behaviour: activities, their interaction and coordination	55
2.1	Cyber-physical systems: interaction of matter, energy, information & data	56
2.1.1	Physics: coupling of space, time, spectrum, matter and energy	57
2.1.2	Cyber: decoupling of continuous, discrete and symbolic models	57
2.1.3	State of a system	58
2.1.4	State representation: ordinal, categorical, continuous, discrete, event	58
2.2	Data: structuring information for computations	59
2.2.1	Mathematical representation	59
2.2.2	Abstract data type — Semantic representation	59
2.2.3	Data structure — Symbolic representation	60
2.2.4	Digital structure — Data representation	60
2.2.5	Electronic realisation — Physical representation	61
2.2.6	Symbolically linked data: metadata, database, query, index	61
2.3	Algorithm: synchronous composition of functions	62
2.3.1	Mereology: mereo-topological model	63
2.3.2	Mechanism: data, function, control_flow, and algorithm	63
2.3.3	Block-Port-Connector model for blocks	64
2.3.4	Data access constraints	64
2.3.5	Directed Acyclic Graph: declarative model of control flow dependencies	65
2.3.6	Policy: closure in a context	66
2.3.7	Policy: application programming interface	66
2.4	Synchronous, asynchronous, sequential, concurrent and parallel execution	67
2.5	Event loop: synchronous composition of computations with asynchronous data	68
2.5.1	Mechanism: 4Cs-based programme template	69
2.5.2	Composition of event loops	70
2.6	Activity: asynchronous composition of algorithms and their interactions	70
2.6.1	Asynchronicity and interfaces	71
2.6.2	Mereology: mereo-topological model	72
2.6.3	Port in algorithm (service) versus Port in library (API)	72
2.6.4	Continuous, discrete, logic, symbolic information: data, event, flag, query	73
2.6.5	Best practice: resource ownership	74

2.6.6	Pattern: mediator for peer-to-peer interactions	75
2.6.7	Pattern: task queue and task worker	77
2.7	Flag: peer-to-peer algorithm coordination protocols	78
2.7.1	Abstract data type: flags	78
2.7.2	Mechanism: flag arrays (“bitfields”) with atomic iterator	80
2.7.3	Policy: big endian versus little endian flag arrays	80
2.7.4	Stop-and-go coordination — Barrier	81
2.7.5	Two-phase commit coordination	81
2.7.6	Data borrowing coordination	82
2.7.7	Higher-order flags: active, timeout, highwater-lowwater	82
2.8	Petri Net: algorithm coordination via dedicated mediator algorithm	83
2.8.1	Examples of multi-algorithm coordination	83
2.8.2	Role of algorithm coordination in systems-of-systems	84
2.8.3	Mechanism: place, token, transition	84
2.8.4	Representation: marking reaction table	86
2.8.5	Pattern: Petri Net with one flag array per coordinated algorithm	86
2.8.6	Flag arrays as linear Petri Nets	88
2.8.7	Stop-and-go coordination — Barrier	88
2.8.8	Any-of and All-of coordination	88
2.8.9	All-to-go, one-to-stop coordination	89
2.8.10	Fork-and-join coordination	89
2.8.11	Policies: hierarchy, cardinality, colours	90
2.8.12	Best practices in Petri Nets	92
2.8.13	Declarative and imperative Petri Net behaviour	93
2.9	Stream: peer-to-peer algorithm interactions via asynchronous data exchange	93
2.9.1	Appearance in the real world	93
2.9.2	Mechanism: producer–consumer array	95
2.9.3	Best practices	96
2.9.4	Policy: status flags for a stream’s activity and backpressure	97
2.9.5	Mechanism: composition of streams	97
2.9.6	Pattern: composition of data and metadata streams	97
2.9.7	Pattern: composition into Submission–Completion streams	99
2.9.8	Policy: submission-completion stream with partial dialogues	99
2.9.9	Pattern: function exchange via task queues	100
2.9.10	Publish-Subscribe & Request-Reply as stream architectures	100
2.10	Finite State Machine: behaviour coordination of one single algorithm or activity	101
2.10.1	Mechanism Part 1: state to coordinate one activity’s behaviour	102
2.10.2	Mechanism Part 2: events to communicate coordination changes	102
2.10.3	Mechanism Part 3: transition and event reaction table	103
2.10.4	Canonical form of a Finite State Machine	104
2.10.5	Policy: event composition	104
2.10.6	Policy: event distribution and conversion	105
2.10.7	Policy: hierarchical states	105
2.10.8	Policy: Life Cycle State Machine	107
2.10.9	Policy: don’t care, history, timeout	108
2.10.10	Policy: selection, priority, deletion	108

2.10.11	Implicit constraints on event handling meta model	109
2.10.12	Runtime creation of FSMs via transition systems	109
2.11	Flags, Petri Nets and FSMs: role in system design	109
2.11.1	Complementary roles of events, and flags and states	110
2.11.2	FSM as boundary case of single-token Petri Net	110
2.11.3	Policy: higher-order model for pre/per/post conditions	110
2.11.4	Application contexts with policy choices	111
2.11.5	Design similarities and differences	111
2.12	Dynamically linked data: discovery, session initialization	112
3	Meta models for geometry	113
3.1	Overview of meta meta models	114
3.1.1	Metadata for coordinates	115
3.1.2	<code>semantic_IDs</code> for geometry in 1D, 2D and 3D	116
3.1.3	Geometric relations in projective, affine and Euclidean spaces	116
3.1.4	Non-Euclidean space for rigid bodies	117
3.1.5	Projections in geographical coordinate system	118
3.1.6	Differential geometry	118
3.1.7	Qualitative spatial relations	118
3.1.8	Taxonomy of geometric meta models	119
3.1.9	Levels of representation in geometry	122
3.2	Meta models of <code>Point-Polyline-Polygon</code> geometry	123
3.2.1	<code>Point</code> entity and its composition relations	124
3.2.2	Extra composition relations in 3D	127
3.2.3	<code>Position</code> and <code>Motion</code> relations of a <code>Point</code>	128
3.2.4	<code>Position</code> and <code>Motion</code> relations of a <code>Rigid_body</code>	130
3.2.5	Constraint relations on mereo-topological entities	132
3.2.6	Abstract data types for <code>Coordinates</code> of <code>Position</code> and <code>Motion</code>	134
3.2.7	Operators on <code>Coordinates</code>	137
3.2.8	Data structures for <code>Coordinates</code>	138
3.2.9	Constraint relations on <code>Coordinates</code> — <code>Geometric_chain</code>	139
3.3	Composition relations: Map as set of geometric entities	141
3.4	Composition relations: <code>Semantic_map</code> , <code>World_model</code>	142
3.5	Uncertainty in geometric entities and relations	143
3.5.1	Sources of uncertainty	144
3.5.2	Covariance of a <code>Frame</code> has no meaning	145
3.6	De facto meta model standards for <code>Coordinates</code>	145
3.6.1	ROS Messages	146
3.6.2	RTT/Orocos typekits	147
3.6.3	SmartSoft Communication Object DSL	147
3.7	Differential geometry as meta meta model	148
4	Meta models for a kinematic chain and its instantaneous dynamics	149
4.1	Meta models	149
4.1.1	Mereo-topology	149
4.1.2	Types	150
4.1.3	<code>Coordinates</code> and behavioural state for <code>Motion</code>	152

4.1.4	Geometrical operations — Forward, inverse and hybrid kinematics . . .	153
4.1.5	Inconsistency, redundancy, singularity	155
4.2	Meta models of mechanical dynamics: composing force and motion	156
4.2.1	Abstract data types and data structures for dynamics	156
4.2.2	Motion as result of constrained optimization — Gauss’ Principle . . .	156
4.2.3	Energy relations — Bond Graphs	156
4.2.4	Differential geometry: manifold, (co)tangent space, linear forms . . .	157
4.3	Meta model for the mechanical dynamics of a kinematic chain	159
4.3.1	<code>Coordinates</code> for dynamics state	160
4.3.2	<code>Coordinates</code> and behavioural state for impedance	160
4.3.3	Geometrical operations revisited — The role of “virtual dynamics” .	161
4.3.4	Dynamic relations under a 5D motion constraint	161
4.4	Meta model for instantaneous motion of kinematic chains	162
4.4.1	Gauss’ Principle of Least Constraint	162
4.4.2	Specification of instantaneous motion	163
4.4.3	Operations: forward, inverse and hybrid dynamics transformations . .	164
4.5	Composition and decomposition of <code>Kinematic_chains</code>	165
4.5.1	Composition — Serial, branch, loop	165
4.5.2	Decomposition — Spanning tree	167
4.5.3	Policy: iteration via sweeps	168
4.5.4	Policy: input-output causality assignment	168
4.6	Taxonomy of kinematic chain families	168
4.6.1	Serial chains	169
4.6.2	Mobile platform chains	169
4.6.3	Parallel chains	170
4.6.4	<code>Multicopter</code> chains	170
4.6.5	Hybrid chains	170
4.6.6	<code>Cable-driven</code> chains	170
4.7	Taxonomy of motion capabilities	170
4.8	Solver meta model for hybrid kinematics and dynamics	171
4.8.1	Mechanism–I: motion drivers	171
4.8.2	Mechanism–II: procedural sweeps	172
4.8.3	Policy: scheduling options in the third sweep	173
4.8.4	Policy: free-floating base	174
4.8.5	Policy: tasks in the mechanical domain	174
4.8.6	Policy: serial kinematic chain	175
4.8.7	Policy: branched kinematic chain	175
4.8.8	Policy: kinematic chain with a loop	175
5	Meta models for robot tasks	176
5.1	Continuous, discrete and symbolic task models	177
5.2	Use case: indoor navigation with a two-wheel driven robot	178
5.2.1	Resources	178
5.2.2	Proprio-ceptive task specification	179
5.2.3	Extero-ceptive task specification	179
5.2.4	Cartho-ceptive task specification	179
5.3	Task-Skill-Resource: activities needed for task execution	180

5.3.1	Reality reflection: capability, resource, world model, perception, monitoring	181
5.3.2	Decision making: plan, control, knowledge-based reasoning	181
5.3.3	Interactions: world model, skill, perception and control activities	182
5.4	Perception and control composition: proprio-, extero-, cartho-ceptive tasks	183
5.4.1	Proprio-ceptive tasks	184
5.4.2	Extero-ceptive tasks	184
5.4.3	Cartho-ceptive tasks	184
5.5	Mechanism: task specification as optimization, satisfaction and reasoning under constraints	184
5.5.1	Specification in the continuous domain: constrained optimisation	185
5.5.2	Specification in the discrete domain: constraint satisfaction	186
5.5.3	Specification in the symbolic domain: reasoning	186
5.5.4	Policy: task requirement as objective function or as constraint	187
5.5.5	Policy: dimensionality reduction through task constraints	187
5.6	Policy: reactive task models — Guarded actions	188
5.7	Task composition	189
5.7.1	Horizontal composition: sharing world models	190
5.7.2	Vertical composition: resources become capabilities, and vice versa	190
5.7.3	Strategic, tactical and operational task levels	191
5.7.4	Multi-tasking: coordinated, orchestrated, choreographed	191
5.7.5	Shared control of task execution	192
5.8	Task specification, data sheet, contract, responsibility and commitment	193
5.9	Taxonomy of action, actor, actant, activity, agent	194
5.10	Bad practices in task specification	194
5.11	Use case: semantic indoor navigation (revisited)	194
5.11.1	Geometric world models with semantic tags for control & perception	196
5.11.2	Specification of an activity to realise a task specification	199
5.11.3	Example task plan: dock to cart in next junction left	200
5.11.4	Example task plan: escape from a room	202
5.12	Domain-specific task specification languages	206
5.12.1	Task ontology and its standardization	206
5.12.2	Semantic mobile robot motion primitives	206
5.12.3	Semantic robot arm motion primitives	207
6	Meta models for dynamic semantic maps and situational awareness	209
7	Meta models for continuous control	211
7.1	Mereology: feedback, feedforward, predictive, adaptive, preview	212
7.2	Information associations in control	214
7.3	Mechanism: state and system dynamics relations	214
7.4	Mechanism: optimal control	217
7.4.1	Policy: PID and pole placement for linear systems	218
7.4.2	From PID and pole placement to optimal control	219
7.4.3	The role of PID and linear control in robotics	220
7.4.4	Policy: model-reference adaptive control (MRAC)	220
7.4.5	Policy: model-predictive control (MPC)	221

7.5	Mechanism: setpoint, trajectory, path and tube inputs	221
7.5.1	Policy: composition of optimal control and tube inputs	221
7.5.2	Policy: control progress objective or constraint	222
7.5.3	PID alternatives: sliding mode, gain scheduling, ABAG	222
7.5.4	PID versus ABAG	224
7.6	Mechanism: cascaded control loops	225
7.6.1	Composition of energy and cascaded control	225
7.7	Mechanism: asynchronous distributed control	227
7.8	Mechanism: behaviour tree for semi-optimal control	228
7.9	Event loops revisited: control behaviour composition	228
7.9.1	Example: one-dimensional position control	229
7.9.2	Policy: hybrid event control	231
7.9.3	Policy: throughput and latency	231
7.9.4	Policy: real-time activities via the “multi-thread” software pattern	232
7.9.5	Policy: event loops for task control	233
8	Meta models for perception and its integration in tasks	234
8.1	Information associations in perception	235
8.1.1	Pre-processing	235
8.1.2	Pre-processed sensor data association to sensor feature	235
8.1.3	Sensor feature association to object sensor features	236
8.1.4	Object sensor feature association to object feature	236
8.1.5	Object feature association to object association	236
8.1.6	Association to task, environment and robot context	236
8.1.7	Association of a mission with its tasks, environments and robots	236
8.2	Mereotopological meta model: the natural hierarchy in robotic perception	237
8.3	Policy: (data) association	237
8.4	Perception example: robots driving in traffic	238
8.4.1	Escape from a room	238
8.4.2	Ego-motion estimation with accelerometer, gyro and encoder	239
8.4.3	Ego-motion estimation with visual point and region features	239
8.5	Probability: composition of data and uncertainty	239
8.5.1	Mechanism: Bayesian probability axioms	239
8.5.2	Mechanism: Bayes’ rule for optimal transformation of data into information	243
8.5.3	Policy: belief propagation	243
8.5.4	Policy: hypothesis tree for semi-optimal information processing	243
8.5.5	Policy: mutual entropy to measure change in information	243
8.6	Geometrical semantics in perception	243
8.7	Dynamical semantics in perception	243
8.8	Policy: tracking, localisation, map building	243
8.9	Mechanism of information update: Bayes’ rule	244
8.10	Mechanism of perception solver: message passing over junction trees	245
8.11	Policy: dynamic Bayesian network	245
8.12	Policy: Factor Graphs	247
8.13	Mechanism: composition of point and region features	247
8.14	Policy: feature pre-processing	247

8.15	Policy: deployment in event loop	247
9	Meta meta models for holonic and explainable system architectures	248
9.1	Hierarchy in knowledge versus hierarchy in architecture	250
9.2	Data sheets: digital resources (holons, twins, platforms) with metadata . . .	251
9.3	Components: heterarchy in activities and interfaces	251
9.3.1	Components, composability, compositionality	251
9.3.2	Vertical and horizontal composition	252
9.3.3	Lifecycle of compositions	252
9.3.4	The 5 <i>C</i> 's: Composition of 4 <i>C</i> behavioural roles in a component . . .	252
9.3.5	Types of Computation	254
9.3.6	Types of state: state, status (flag), and their changes (event)	255
9.3.7	Modes of coordination: coordinated, orchestrated, choreographed . .	255
9.3.8	Mechanism and policy in system composition	256
9.3.9	Policy: vendors add value in Configuration, Coordination, Composition	256
9.3.10	Bad practices: interpreting attributes as properties	256
9.3.11	Block-Port-Connector for components, activities and functions	257
9.4	Document Object Model: hierarchy in coordination & configuration	258
9.4.1	Good and bad practices for hierarchical models	258
9.4.2	Mechanism: elements, tags, attributes, and containment	258
9.4.3	Policy: paths, diffs and multi-tree	260
9.4.4	DOM Cascading Style Sheets for top-down configuration	260
9.4.5	DOM events for top-down and bottom up coordination	261
9.4.6	DOM model for relation-constraint-tolerance models	261
9.4.7	Graph connections between DOM model trees	262
9.4.8	Platform Independent/Specific Model, Domain-Specific Language . .	263
9.5	Block-Port-Connector for connectivity	263
9.5.1	DOM model	264
9.5.2	The mediator pattern in human society	264
9.5.3	The mediator pattern in cyber-physical systems	265
9.5.4	Two-ways symbolic indirection via third-party broker	265
9.5.5	Policy: extend a mediator into a broker	265
9.5.6	Reification of a Connector and its Ports	266
9.6	Mediator: hierarchy in decision making for inter-dependent components . . .	266
9.7	Holons for heterarchical decision making in task execution systems	267
9.7.1	Resilient, stable, robust sub-systems	267
9.7.2	Dependencies in architectures: hierarchy, heterarchy and holarchy . .	268
9.7.3	Explainability: causal driver for robustness, resilience, anti-fragility .	268
9.7.4	Key Performance Indicators	268
9.8	Autonomy as explainable decision making	268
9.8.1	Endsley's five levels of system autonomy	269
9.8.2	Sheridan's ten levels of system autonomy	269
9.8.3	Explanation levels for autonomous decision making	269
9.8.4	Role of meta models in horizontal and vertical task composition . . .	270
9.9	System safety as explainable decision making	271
9.9.1	Best practice: safety PLC	271
9.9.2	Best practice: active safety behaviour	271

9.10	System security as explainable decision making	271
10	Meta models for information architectures	272
10.1	DOM models in robotics geometry	273
10.1.1	DOM model for polygonal shapes	273
10.1.2	DOM model for coordinates	275
10.1.3	DOM model for geometric and kinematic chains	275
10.1.4	DOM model for shape and inertia kinematic chain links	278
10.1.5	DOM model for actuator and transmission in kinematic chain	279
10.2	DOM model for numeric, symbolic and semantic maps of world, building, and floors	279
10.2.1	DOM model for semantic localisation and navigation	280
10.2.2	DOM model for motion tasks	280
10.2.3	DOM model for instantaneous motion specification in kinematic chains	280
10.3	DOM model for cascaded control loops	281
10.4	DOM models for activities	281
10.4.1	Data, functions and algorithms	281
10.4.2	Activities, streams and event loops	282
10.5	Architectures for data and information exchange	284
10.5.1	Activity = Block + Port, Interaction = Connector	284
10.5.2	Symbolic indirection for composability <i>and</i> performance	284
10.5.3	Policy: producer-consumer channel specialisations	284
10.5.4	Pattern: submission-completion for journaled CRUD interfaces	284
10.5.5	Pattern: world model as blackboard	287
10.5.6	Pattern: world model as semantic database	287
10.5.7	Best practices in activity and channel design	288
10.6	Mediator architectures for interacting activities	289
10.6.1	Mechanism: execution of event handling in an event loop	289
10.6.2	Mechanism: ownership, loose coupling, semantics	291
10.6.3	Mechanism: activities exchange data via channels	291
10.6.4	Policy: discovering and connecting peers	293
10.6.5	Mediator architecture for Task-Skill-Resource	293
10.6.6	Task queue mediator activity	294
10.6.7	Task queue with worker pool	294
10.6.8	Coordination and configuration of architectures	294
10.7	Architectures for concurrent algorithms with configuration and coordination	294
10.7.1	Pattern: splitting a solver into scheduling and dispatching	295
10.7.2	Collaborative pre-emption in iterative solvers	295
10.7.3	Caching and memoization in solvers	295
10.7.4	Cascaded control loops	296
10.7.5	Adaptive control loops	296
10.7.6	Tasks as hybrid constrained optimization algorithms	296
10.7.7	Control of kinematic chains	298
10.7.8	Policy: deployment in an event loop	298
10.7.9	Composition with dynamics of resources	298
10.7.10	Composition with dynamics of perception and world model	298
10.7.11	Composition with motion trajectories	298

10.8	Architectures for proprio-, extero- and cartho-ceptive control	299
10.8.1	Proprio-ceptive architectures	299
10.8.2	Extero-ceptive architectures	299
10.8.3	Cartho-ceptive architecture: robot, edge/fog, and cloud	299
10.8.4	Situational aware architecture in the “edge”	301
10.9	Common architectures for robotic task execution	303
10.9.1	Task-Skill-Resource	303
10.9.2	Sense-Plan-Act, Three-Layered, Behaviour, Subsumption	303
10.9.3	Task-Skill-Resource versus Sense-Plan-Act	303
11	Meta models for software architectures	305
11.1	Mechanism: mapping between symbolic IDs and memory addresses	307
11.2	Mechanism: bitfields for flags, FSMs and Petri Nets	307
11.2.1	Bit fields	307
11.2.2	Dictionary of linked lists	307
11.2.3	Events or flags?	308
11.3	Mechanism: buffer, queue, socket	308
11.4	Mechanism: stream with wait-free ring buffer	310
11.4.1	Design overview	310
11.4.2	Block-Port-Connector conforming meta model of a Stream buffer	311
11.4.3	Implementation design	312
11.4.4	Policy: late binding of data chunk type	315
11.4.5	Policy: sharing a stream between processes	315
11.4.6	Policy: composition of data and metadata	315
11.4.7	Policy: time series	316
11.4.8	Policy: composition of stream and memory pool	316
11.4.9	Policy: heartBeat/watchDog mediation for “lazy” stream	316
11.4.10	Policy: contiguous data for producer and for consumer	317
11.4.11	Mechanism (software): stream with wait-free maximal freshness buffer	317
11.4.12	Standards and software projects supporting stream channels	319
11.5	Mechanism: event loop	319
11.5.1	Event loop design trade-offs	320
11.5.2	Event loop with ring buffer and scatter-gather I/O	321
11.5.3	Runtime configurable instrumentation	321
11.5.4	Multi-rate time series streams	321
11.6	Architecture to compose task queues, workers, event loops, and solvers	322
11.6.1	Pattern: composition of “task queue” and “library API” patterns	323
11.6.2	Policy: frameworks, middleware, solvers	324
11.6.3	Composition of stream with object pool	324
11.7	Architecture for hierarchical activity deployment	324
11.7.1	Hardware hierarchy: core, system-on-a-chip, computer, fog, cloud	324
11.7.2	Software hierarchy: runtime, thread, process, shell, container, cloud	325
11.7.3	DOM model for activity deployment hierarchy	327
11.7.4	DOM model for interaction streams	327
11.7.5	Pattern: composition of process, thread, activity and algorithm	327
11.7.6	Mechanism: property graph implementation	328
11.7.7	Pattern: process composing multiple threads	328

11.8	Best and bad practices	331
11.8.1	Best practices	331
11.8.2	Framework plug-ins (bad) versus library composition (good)	332
11.8.3	Bad practices in robustness	333
11.8.4	Bad practices in resource locking	334
11.8.5	Bad practices in communication	334
11.9	Mechanism: programming language operators	335
11.9.1	Async/await	335
11.9.2	Iterator	335
11.9.3	Maybe	335
11.9.4	Memory barrier	335
11.9.5	Atomic and lockfree operators	335
11.9.6	Mechanism: Conflict-Free Replicated Data Type (CRDT)	335
11.9.7	Mechanism: immutable data type	336
11.10	Modelling languages: mature implementations & tools	336
11.10.1	QUDT and UCUM	336
11.10.2	JSON and JSON-Schema	336
11.10.3	JSON-LD	337
11.10.4	RDF1.1	337
11.10.5	Abstract Syntax Notation One (ASN.1)	338
11.10.6	Hierarchical Data Format — HDF5	338
11.10.7	FlatBuffers, Protocol Buffers, Apache Arrow	338
11.10.8	Common Trace Format — CTF	338
11.10.9	BLAS, LAPACK	338
11.10.10	DFDL	338
11.10.11	KML Schemas	339
11.10.12	PROV-O provenance ontology	339
11.10.13	Functional Mockup Interface (FMI)	339
12	Skill architectures for eight-wheel mobile robots	340
12.1	The eight-wheel drive mobile platform	341
12.2	Topological navigation and metric motion tasks	342
12.3	Platform navigation and motion modes	344
12.4	2WD force transmission	345
12.5	2WD efficiency of force transmission	346
12.6	Platform pushing is (locally) stable, pulling is unstable	348
12.7	Platform force distribution over all 2WD units	349
12.7.1	Mechanism: forward force composition from wheels to platform	350
12.7.2	Mechanism: inverse force distribution from platform to wheels	351
12.7.3	Mechanism: force distribution at the wheels	352
12.7.4	Policy: numerical force distribution solvers	353
12.7.5	Mechanism: force trajectory objectives	354
12.8	Mechanism: declarative platform motion specification via tubes	354
12.9	Policy: 2WD control modes, task progress, energy, slippage	355
12.10	Policy: hybrid platform control	356
12.11	Navigation: topological motion specification and control	357
12.12	Information architecture	357

Chapter 1

Foundations of knowledge-based systems: ((meta) meta) modelling

This document’s **formal representation of knowledge** (also know as “a **model**”) is built on the **axiomatic foundation of entity and relation**: an entity represents “stuff”, “things”, “primitives”, “atoms”, . . . , and a relation represents a dependency between one or more properties of the entities [22].

Formally representing the **meaning** of a model requires relating the model to one or more **meta models**. The latter represent the relations that constructs in the model must satisfy (“**conform to**”) in order to be “**well formed**” (i.e., having an appropriate **syntax**) and “**meaningful**” (i.e., having an appropriate **semantics**).

The model that **associates** entities and relations in two or more meta models is called a **meta meta model**. It is needed to realise **model-to-model transformations**.

The relation between a model and its meta models is a relative concept: every meta model is a model in itself and hence has its own set of meta models.¹ Common practice limits the terminology to the triple **model–meta model–meta meta model**. A **knowledge system** for a particular domain is a set of models and their meta models that describe “meaning” in that domain. **Reasoning** in such a knowledge system is done via **queries**, that are **solved** via **graph matching** or **graph traversal**.

A robotic or cyber-physical system has an **explainable** control system, if the latter supports the **higher-order reasoning** required to identify the **causality chains** in the graph relations that result from a query.

This Chapter describes the modelling concepts in the core of any **knowledge-based** engineering, often also called **model-driven**² engineering. The challenges to do the modelling “right”, are:

- to find the right **levels of abstraction**: which entities and relations to include in the modelling of each level, and which ones to neglect.

¹As long as a human has made the effort to create those meta models.

²For all practical engineering purposes, this document makes no distinction between both terms: *knowledge* is formally represented in *models*, and *models* only have meaning for people with the *knowledge* to interpret them.

- **to design structures that connect levels.** with explicit models of which behavioural aspect of a system they relate to each other at different levels.
- to bring the **right levels in scope** of an offline design, or of an online decision making, at the right moment.
- how **to create** knowledge bases for computers **to reason** on the models contained in those servers.
- to find the right “**host**” **languages**, to store and communicate represented knowledge.
- how to reach the **standardization** required for realistic multi-vendor interoperability.

1.1 Models to represent knowledge in science and engineering

“Modelling” is the mental activity of the human scientist or engineer to make an artificial language (the “**modelling language**”), with which **to represent** in a formal way (the “**model**”), the **knowledge** about the behaviour and the properties of:

- **entities**, in the real world, as well as in the abstract knowledge “world”.
- **relations**, physical as well as information-theoretic, between two or more entities.

A few example of relations are:

- **scope**: what is important? What is the **domain of discourse**?
- **interaction**: which influences exist between entities?.
- **behaviour**: how do the properties of entities and relations determine their behaviour?
- **abstraction**: “*to act*” is a more abstract verb than “*to move*”, which is itself more abstract than “*to grasp*”.

1.1.1 Science reflects reality — Engineering reflects decision making

Scientists strive for models that allow **to analyse reality**. Engineers strive for models that allow **to design artefacts**, that **make decisions of how to change reality** in a **controllable** and **predictable way**. And the engineering should be done in such a way that the design models can feed machines **to implement** the artefacts in the real world. Engineering models typically make use of scientific models; the inverse is less commonplace.

Both scientists and engineers know that a model *is not* the reality, but just a *representation* of their artificial and subjective selection of those parts of reality which they consider relevant. They both also know that such relevance is not an absolute property, but one that depends on the **context**. That context determines that their modelling stops with a particular selection of axioms or facts, that are not modelled in further detail but are assumed to be **grounded**. In science, such grounding consists of (references to) other (possibly not yet formalised) models and axioms, and in engineering it consists of software, common knowledge and facts. The last resort of that grounding is the human mind: eventually, it will be humans who give the validation stamp to the quality of a **model**, or of the **software** that implements models and the **tooling** that transforms models. A core ambition of modelling in this document is to represent **context** and **composition** explicitly, and in such a structured way that one can *add* new models to already existing ones *without changing* the structure and the meaning of the latter. When developing computer-controlled machines, there is an obvious extra core ambition: to create **software** libraries of **digital twins**, that implement (or “ground”) the models of the entities and their interactions.

1.1.2 Levels of abstractions: scope of relevant entities and relations

Every model is an **abstraction** of the real and engineered world: only the *relevant part* of how the real world behaves is to be covered by a model's entities and relations. System designers must choose the “right” **scope** of these relevant parts, or, in other words, the right **levels of abstraction** that fit to the system's purposes, [2, 38, 39, 99]. This implies two complementary choices:

- **abstraction**: to choose the “right” set of entities, together with the relations that connect properties in two or more entities. A pragmatic, minimal expectation of a “good model” is that:
 - the relations do not contradict each other.
 - the model does not have sub-models that are not connected to other sub-models by any relation.
- **level**: to choose a *hierarchy* in the relations between the models, which facilitates a methodological decision about whether or not to include particular information about the real world. *Higher* or *lower* level of abstraction are not absolute properties of a model, because that label only make sense *between* two models: model A is of a higher abstraction than model B if the latter contains (a subset of) all entities and relations of A, *plus*:
 - *new entities* that are connected to entities already present in A, by the already present relations of A.
 - *new relations* that connect the already present entities of A to the new entities of B.

For example: “*to act*” is a more abstract verb than “*to move*”, because motion is a specific form of action that requires entities and relations of **physical objects** and their location in **space and time**. And “*to move*” is more abstract than “*to grasp*”, because grasping is motion with hands and fingers, and with the relations of **form and force closure** of the fingers around a physical object.

Humanity has not yet found any scientific arguments to identify *the* set of levels of abstraction. in this document, several complementary **structures** of *level of abstraction* are introduced:

- the **hierarchies** of **knowledge** and **information**.
- the **mereological** and **topological** relations, to represent “parts” and their “interconnections”. For example, the **containment** hierarchy.
- the relation between **hyponyms** and **hypernyms**, as the essence of *is-a-(sub/super)-type-of* hierarchy, or **subsumption**.
- **modelling hierarchy**.
- **mathematical** representation [92].
- physical detail (Fig. 2.1).
- coverage and extension in time and **geometrical space**.
- **tasks**: *what* has to be done is more abstract than *how* it is being done; several levels are common.
- software: an **interface** is a higher **abstraction layer** than any (of its possibly many) **implementations**.
- ...

1.1.3 Levels of modelling formalization

This document has models in all **levels of modelling** formalization identified in the **RobMoSys** ecosystem context:

1. **abstraction**: these models are not formal at all, but contain abstractions in a natural language that experts in the domain are familiar with. These models help to guide discussions and to transfer information between humans. The better they are, the more—and more harmonized—interpretations can be shared by more practitioners in a domain (including less experienced juniors) about the relevant entities and relations in that domain.

In other words, this class of modelling is commonly called **textbooks**.

2. **reuse & flexibility**: these models are still in natural language, but they do describe software engineering artefacts. Only when the “abstraction level” harmonization has matured in a community, one can expect software to be developed whose meaning and behaviour are well understood by all developers and users in that community.

This class of modelling is commonly called **documentation**.

3. **predictability**: these are models that are already formalized to the extent that they form the basis for software tools and standards. The harmonization has reached a level of maturity where the explanation in a *standards* document, and the availability of a *reference implementation* that conforms to the standard document, suffice to let everyone use software artefacts based on the standard with unambiguous interpretations, and with predictable quality of the outcome. Hence, support from software tools becomes realistic, or even mandatory as the major pragmatic way of software development in a broader community.

This class of modelling supports tools that can predictably generate artefacts *correct by construction*.

4. **automation**: the models also have *meta models* so the tooling can not only generate *correctly constructed* artefacts, but the standardisation has been formalized so far that the also the *meaning* and *behaviour* of software artefacts can be checked automatically.

This class of modelling supports automatic **verification and validation** of the results.

5. **autonomy**: these models can serve as an “artificial language” for run-time dialogues between machines. The formalization and its automatic checking has become (i) efficient enough to be used at runtime by the robots *themselves*, and (ii) rich enough so that the machines can set up added-value cooperations *themselves* via dialogues, or can configure, adapt and explain their own behaviour, with only minor human interaction.

This class of modelling supports autonomous *self*-configuration, -adaptation, and -explanation, at run-time.

1.1.4 Levels of decision making: context of intentions

An engineered system must respect reality, but still has a lot of choices **to influence** that reality. Structuring the decision making of these many choices, in a predictable and comprehensible way, reflecting the **intentions** behind the decisions, is the major challenge of the system designer.

There is a hierarchy in this decision making, where a higher level has more **context** than its lower levels, to decide which decision making configurations the lower levels will be responsible for. With this *decision making* responsibility also comes a *reporting* responsibility: a lower level must *monitor the progress* of the decision making that a higher level has giving it responsibility for, and *inform* that higher level about it.

1.1.5 The meaning of “meta”

The **natural language meaning** of the adjective “meta” denotes something that is *more comprehensive, of higher order, or reflective on itself or on others*. This document has the following meanings:

- *meta model*: model of the constraints a model must satisfy. Synonym of **meta language**.
- *meta meta model*: a model that **associates** entities and relations in two or more meta models. It is needed, for example, to realise **model-to-model transformations**.
- *metadata*: information about the interpretation and **provenance** of the fields in a data structure.
- of a *higher abstraction* level.

1.2 Representation of knowledge: graph models

Knowledge is the **interconnection** between **data, information and meaning**, [2] and formal representations of knowledge have been given names like **knowledge base, knowledge graph, semantic database, or ontology**.



Figure 1.1: Directed graphs. The graph on the left is an (un-labelled) tree, i.e., there are no *cycles* in the edge connections between nodes. The one on the right is a **directed acyclic graph** (DAG), with *labelled* (or *named*) edges.

The **topological** (or “**structural**”) basis is a **graphical model**, more in particular, a **labeled directed graph**, Fig. 1.1:

- two **nodes** can be connected by an **edge**.
- an edge can connect a node to itself.
- nodes and edges have a **label** (or “**name**”).
- a edge has a **direction**, because in many relations it is meaningful to identify the “start” and the “end” of the relation.

A knowledge representation adds **meaning** to the structural model, by connecting **named properties** to the graph’s nodes and edges.

1.2.1 Paradigm, pattern, best practice

A **paradigm** is a **loosely coupled and informally identified** set of all models, thought patterns, techniques, practices, beliefs, mathematical representations, systematic procedures, terminology, notations, symbols, implicit assumptions and contexts, values, performance criteria, . . . , shared by a community of scientists, engineers and users in their modelling and analysis of the world, and their design and application of systems. So, a paradigm is a *subjective, collective, cognitive but often unconscious* view shared by a group of humans, about how (they think) the world works. Examples of such scientific paradigms are: the dynamics of Newton and Einstein, the astronomical theory of Copernicus, Darwin’s evolution theory, meteorological and climatological theories, quantum mechanics, etc. Some of those are universally accepted, others are less so. And all of them have, to a certain extent, a subjective basis. But all of them are also *scientific* paradigms in the sense that all interpretations and conclusions based on experimental data, and made on top of the paradigm’s subjective basis, are derived in a systematically documented, refutable, and reproducible way.

The good news of having paradigms is that practitioners within the *same* paradigm need very few words to communicate or discuss their ideas and findings, because they share the paradigm’s large amount of (implicit) background knowledge and terminology. The bad news is that practitioners *from different* paradigms often find it difficult to understand each other’s reasoning and to appreciate each other’s procedures and results. And to realise that their difficulties are caused by their thinking inside different paradigms in the first place.

A (design) **pattern** is **knowledge about how to solve** a recurring design problem, by means of a parameterized solution approach. The **name originated** in **architecture**, under the influence of **Christopher Alexander**, who formalised and documented a series of solutions to problems in the **built environment**, [6]. The solution is **documented** following a structure that helps to discuss and select the **design trade-offs**: the pattern identifies explicitly what the influence is of each of the “forces” that drive the design in particular directions. In other words, a pattern has a knowledge graph behind it, so that reasoning about the design becomes possible.

A **best practice** is a **factual** observation that a particular approach has been recognized to work very well in particular contexts. The best practice is rather *monolithic*: it is to be applied as-is, without much leeway to optimize any trade-offs in the solution, for the simple reason that there is no identified relation between a set of design decisions and the resulting system performance.

1.2.2 Simple, though not easy

Simple refers to the effort *to understand*; *easy* refers to the effort *to implement*. Two concepts derived from “simple” are worth introducing too: *simplicity* refers to the (positive) ambition to remove everything except what *matters*; *simplistic* refers to the (negative) outcome where so much has been removed that what remains is inefficient,

1.2.3 Mechanism: graphs (RDF, property, entity-relation, factor)

A **graph** (Fig. 1.1) is the structure that models that some **nodes** are related (“connected”) by **edges**. This Section introduces some specific *types* of directed graphs, that each add a bit of extra **semantics** to the structure of the graph:

- **RDF** graph, or linked data graph.
- property graph.
- factor graph, or entity-relation graph.

RDF³ graph (Fig. 1.2), or **linked data graph**. This is a labeled (or, “named”) directed graph with one semantic addition: the directed edge between two nodes has the meaning of a **subject-predicate-object** relation:

- the first node is the *subject*.
- the edge is the *predicate*.
- the second node is the *object*.

This basic graph composition of two nodes and an edge is called an (RDF) **triple**. A database with RDF triples is called a **triplestore**; if the database adds the name of the graph in which the triple is stored, one calls it a **quadstore**, or a *named graph*. That name is the essential attribute to allow **linked data**.

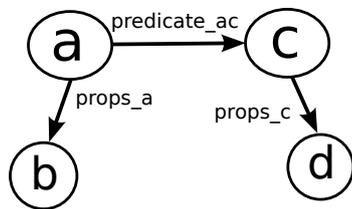


Figure 1.2: An **RDF** graph is a property graph with *triples*, that is, two nodes and one relation edge: one *subject* node (e.g., **a**), one *predicate* edge (e.g., **predicate_ac**), and one *object* node (e.g., **c**).

The figure shows the most common use case of RDF triples in knowledge representation:

- each node has one or more *property* relations connecting the node to nodes that contain a set of *parameters*, often in the form of **key-value pairs**. A property node is a *leaf node* of the graph.
- each node can be the subject or the object of one or more *predicate* relations, whose symbolic name represents some meaning in the application context. And, hence, the node is most often not a leaf node in the graph.

Of course, the property relation is a specific *type* of predicate, that appears everywhere in knowledge modelling.

Property graph (Fig. 1.3). While **RDF graphs** have their origin in the domain of **linked data** and want to serve first and foremost the *human* modeller, the property graph originates from the domain of **graph databases**, focusing on representations that can be realised completely on top of normal, plain graphs. So, a property graph is a labelled and directed graph, with a simple semantic addition: every node has an edge to a node containing its **properties**. These properties come as **key-value pairs**, or any other **abstract data type**, that can represent the properties of the node.

The “semantic richness” of a property graph is a bit less than that of an RDF graph: a relation is not expressible as a *first-class citizen* of a property graph, but must be represented by adding an extra node, whose properties represent the usage of that node as a relation between the connected nodes. Figure 1.3 shows a property graph, whose meaning *can* be exactly the same as the RDF graph in Fig. 1.2. (Whether it *is* the same depends on the interpretation of the parameters in the property nodes.)

Factor graph, or entity-relation graph (Fig. 1.4). This type of graph adds a small extra semantics to the property graph, namely that there are *three types* of nodes:

³“RDF” stands for **Resource Description Framework**.

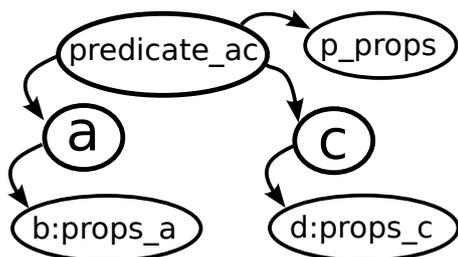


Figure 1.3: Property graph, representing an equivalent model to the RDF graph in Fig. 1.2.

- entity nodes.
- property nodes.
- relation nodes.

A relation node, or **factor**, is connected to **two or more** entity nodes, and the properties of that relation node represent *couplings* between the properties all connected entity nodes.

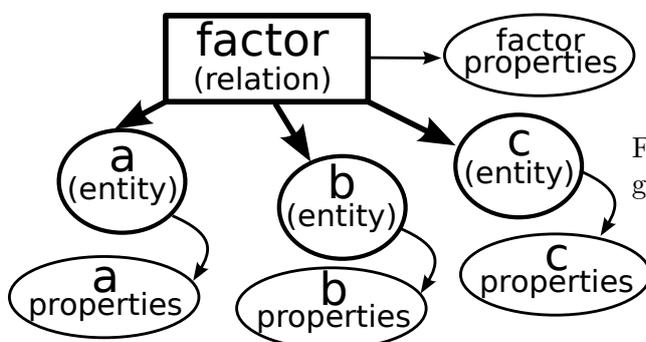


Figure 1.4: Factor graph, or entity-relation graph.

Normal, property and RDF graphs can only have an edge (and hence, represent a relation) between *two* nodes, while an entity-relation graph can represent **n-ary** relations between *more than two* nodes. (In graph theory, the latter property is sometimes given the name of a **hypergraph**.)

1.2.4 Policy: first-order relation as an entity-relation graph

This document adopts the **axiomatic** basis to model **first-order knowledge models** (that is, **relations**) with **entity-relations graphs**: **nodes** represent **entities**, **edges** represent **relations**, and both have **properties**. (When **storing** or **communicating** models, the more basic **property graph** model is a more pragmatic option. Anyway, the *semantic mapping* between both is straightforward.) Properties represent information like name, identity, type, the data structures that store the parameters that define the entity’s “behaviour”, **provenance**, **Dublin Core** metadata, etc.

A major example of a first-order relation that can be represented by an entity-relation graph is the **S-expression**. It is the basis of **context-free languages**, with programming languages as major representatives, but also **reasoning** capable languages like **Prolog**, **Lisp**, or a **query** language like **SPARQL**. A specific major type of S-expression is an **algebraic** relation; for example, **Newton’s law** expresses the linear relation $f = m a$ between force f , inertia m and acceleration a .

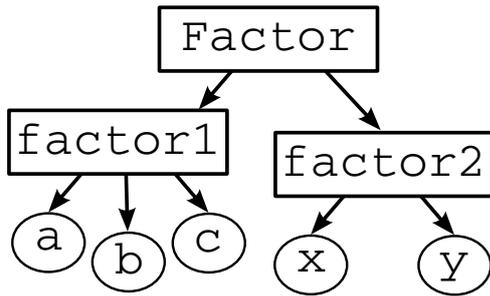


Figure 1.5: Higher-order relation: The **Factor** relation has two other relations, **factor1** and **factor2**, as its entities.

1.2.5 Policy: higher-order relation as an entity-relation graph

A second **axiomatic** basis of this document is that any application of somewhat realistic complexity needs to represent not only first-order but also **higher-order relations** (Fig. 1.5), [49, 78, 87]. That is, a relation that has other relations as its entities, or “arguments”. The obvious need for higher-order relations is the fact that each application comes with its own particular **context**, that is, the knowledge needed **to interpret** entities and relations used in that application, and (hence) **to configure** many of its “**magic numbers**” to fit that context. The above-mentioned relation of Newton’s law illustrates this situation, because a full understanding of that law requires a large set of knowledge relations that are not present in the the law’s symbolic representation itself:

- the entity *acceleration* only get its full meaning in the domains of *geometry* and *analysis*, that is, as the (second) *time-derivative* of a *function* of *position*.
- the entities *force* and *inertia* and *acceleration* have meaning in the domain of *mechanical dynamics*, in that they link the change of geometrical position over time to the use of energy.
- all entities can be given *numerical coordinates*, linking the *quantitative* values of each *type* of entity to unique *physical dimensions* and to various possible *measurement units*.
- in the context of moving the mass with a robot, the force must be generated by *actuators* and measured by *sensors*, both of which are *constrained* in their performance and resolution.
- etc.

Even this very simple example makes it clear that there is no end towards “the top” of such contextual relations, because every new higher-order relation introduces new entities and properties whose meaning has to be modelled too, before one can *reason* with them. Indeed, the relevance of higher-order relations from a **semantic** point of view is that they encode the knowledge about:

- **why** the connected properties are connected.
- **how** their connection must be used.
- **what variations** in property values are allowed.
- **what role** does each of its argument play in the relation.
- the **dependency** of the properties on the specific context in which the higher-order relation is defined.
- etc.

The “higher-orderness” is not *directly* visible in the structure of the graph model, but only in the properties of the relation. It is visible *indirectly*: a higher-order relation can not have entities in its relation that are *leaf nodes* in the graph, because its entities must be relations

with entities themselves.

1.2.6 Observation: “higher-order” is not necessarily “more abstract”

The concept of *higher-order relation* means “relation on a set of relations”, and it is often considered as a synonym for “more abstract”. This is not correct, though, as shown for example by the *entities–relation–constraints–tolerances* below: the *tolerances* are the highest-order relation in that pattern, but most often consists of a couple of numbers, and numbers are not at all concepts of high abstraction.

1.2.7 Facts

A **fact** is a *leaf* relationship in a knowledge graph: it is not *explaining* any other relation, or *deducing* one relation from some other relations.

One can always start a knowledge graph with facts, because *higher-order composition* is a very composable operation. That is, any fact can later be extended with higher-order relations that represent the reasons why the particular fact is true; or conditionally true, given some contextual constraints.

1.2.8 Properties and attributes

In most *entity-relation* models, the entities as well as the relations have a set of *key-value pairs* that contain the data that quantifies the *meaning* of the entity or the relation. And a higher-order relation is a *connection* between two or more of such keys or values in two or more entities or relations. Such a *key-value pair* is often called a *property* or an *attribute*. This document makes the following semantic difference between both terms:⁴

- **properties** of an entity are the data that that entity **possesses**, or “owns”, or that are “proper” to its existence, so that without that data the entity loses its meaning, *irrespective of the context* in which it is used.

For example, every physical object has mass and electrical conductivity.

- **attributes** are the data that are **given** to an entity, *depending on a particular context* of relations that involve that entity as an argument. More specifically, the attributes are the properties of the *role* of the entity in such a relation.

For example:

- the colour of a physical object in a camera image depends on the interplay between its surface texture, the properties of its surface paint, the lighting conditions in the environment, and the properties of the camera.
- the current that flows through the physical object depends on the properties of the electrical circuit it is part of.
- the position of a rigid body in space is always relative to other bodies or references.
- the name of an entity, in different languages.
- the state of a system is a set of data values that describe what to remember of the system in order to predict the future. This meaning depends on the *purpose* of the state in the *application* that uses the system.

Properties are typically unique; for example, one particular object has only one specific mass, not several. Attributes can be given multiple times to an entity; for example, a car can have

⁴With the choice motivated by the *etymology* of the terms.

a *serial number* given to it by the manufacturer, another one (“license plate”) given to it by a government, and a third one given to it by a *car leasing* company. Similarly, one person can have multiple names and *nicknames*, and identity numbers (passport, social security, company ID, ...).

1.2.9 Magic number: attribute in a higher-order causality relation

The concept of a *magic number* is commonly used in the implementation of complicated systems: the **cause** of the value of some *parameters* in the system can not be identified, and the human developers choose “a value that works”. This is, obviously, an undesirable situation in this document’s context of **explainable** systems. The advice to deal with magic numbers is:

- as a minimum: to identify the different entities and relations that *influence* the value of a magic number. That magic number is then an *attribute* of that higher order relation.
- as a best practice: to introduce that influence relation as an explicit new higher-order relation in the system model, to make system developers aware of the situation.
- as a permanent solution: to introduce *causality* in the influence relation, so that the *reason why* the magic number gets its value is explicit. The magic number then has become a *property* of that causality relation.

Without such causality insight, magic numbers are **extreme showstoppers** for the integration of components into a system.

1.2.10 Reification — Best practice of *attributes are properties of relations*

It is simple to express a higher-order relation as a composition of several first-order relations: make every *relation* an *entity* in itself. In this way, a relation can be given properties too, and can serve itself as an argument in other relations. This approach is sometimes called **reification** or **name binding**.

So, this document has little need to use the term “attribute”, because (i) it uses relations as first-class citizens so their properties already serve the role of attributes, and (ii) any relation is always considered to be an entity in itself. So, it is obvious that the attributes that an entity would get in a relation are the entity-specific properties of that relation.

Nevertheless, it is **best practice** for knowledge modellers to always ask themselves the question whether the key-value pairs they are adding to a knowledge graph are “properties” or “attributes”: classifying them as the latter *implies* they have to make a hitherto “hidden” relation explicit.

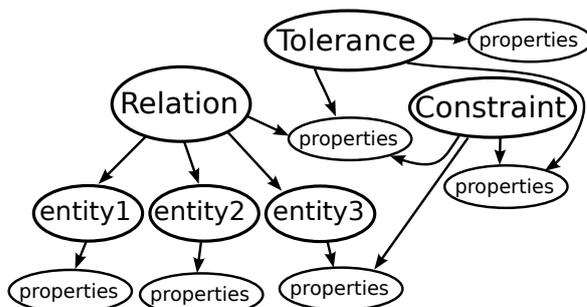


Figure 1.6: A *directed acyclic graph* representation of a higher-order model: the *constraint* between the properties of a relation and one of its arguments is a higher-order relation, with its own properties. Similarly, the *tolerance* relation is an even higher-order relation, too.

1.2.11 Pattern: entities–relation–constraints–tolerances

Figure 1.6 gives one particular **composition** of higher-order relations, which comes back in so many use cases that it merits to be called a **pattern**:

- **one** *relation* connects **several** *entities*.
- each entity has its own *property* set.
- the relation has its own *property* set, too.
- **several** *constraint* relations exist between the values of some of these properties.
- the constraint relation has its own *property* set, too.
- **several** *tolerance* relations exist on the values of these properties.

Some examples:

- the relative positions between a robot and objects in its environment.
- the quality of a robot’s motion, as an interplay between its control and perception capabilities.
- the desired quality of a system’s behaviour over a particular interval in time, or in space.

1.2.12 Pattern: collection composes entities in bag, set, list, tree, graph

The following list represent the major ways to compose entities into “something larger”, often called **collections** or **containers**:

- **bag** (or “multiset”): collection of entities.
- **set**: collection of unique entities.
- **list**: set of (unique or not unique) entities with a **serial order**.
- **tree**: every entity in a list can be a list in itself.
- **graph**: collection of trees, where entities can be present in more than one tree.

1.2.13 Pattern: structural constraints on collections as array, dictionary

The following compositions of collections are of universal importance:

- **array**: an ordered list with entities that are all of the same type.
- **dictionary** (or “**associative array**”, or “**map**”) is a composition of two lists with the following *constraints*:
 - both have the *same length*.
 - both are *ordered*.
 - the first list contains the *keys*, the second list contains the *values*.
 - all keys must be unique.
 - *meaning* is associated *only* to the composition of a key and a value at the *same index* in both lists.

1.2.14 Pattern: behavioural constraints on collections as queue, stack

The following compositions of behaviour on collections are of universal importance:

- **queue** (or “**FIFO**”): entities are **added** always on one side of the list, and **removed** at the other end.
- **stack** (or “**LIFO**”): entities are **added** on one side, and **removed** from that same side.

1.2.15 Pattern: composition of relations as fact, model, instantiation

Each of the above-mentioned relations (*relation*, *constraint*, or *tolerance*) can be a **fact** (“factual relation”, “grounded relation”) or an **abstraction**. Facts are *leafs* of the knowledge graph, because none of the entities in a factual relation is in itself a relation. In an abstraction, one or more of its entities *are* relations in themselves, and, hence, are not leafs of the graph. Hence, the abstraction relations represents many possible **instantiations** of the represented knowledge, each relevant in a *more concrete* (or, *less abstract*) view on the world.

For example, the **Pythagorean theorem**

$$a^2 + b^2 = c^2$$

is an *abstraction* of all **right-angled triangles**, and

$$a = 3, b = 4, c = 5$$

is the *factual relation* representing one specific right-angled triangle.

Here is another example: **President of the European Council**⁵ is an *abstraction* of a particular **office holder** in the **European Union**. It is a *fact* that **Herman Van Rompuy** was the first holder of that office.

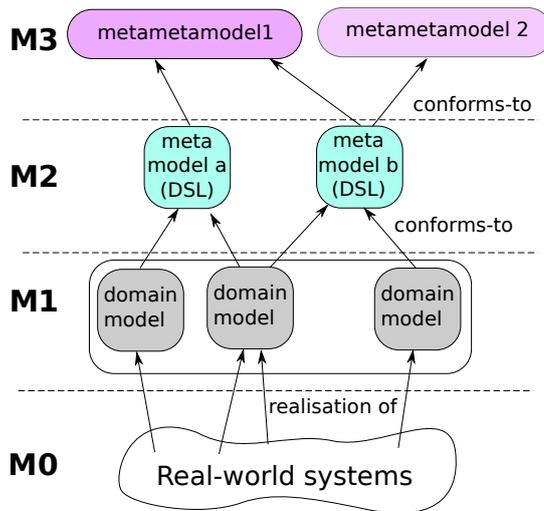


Figure 1.7: The particular (partial) order between types of models that appears in all knowledge representation projects, in one way or another. Note the essential difference of the **realisation_of** relation between the M0 and M1 levels, and the **conforms_to** relation between levels M1 and M2, and levels M2 and M3.

1.2.16 Best practice: meta modelling with M0–M3 structure

Figure 1.7 gives another particular composition of higher-order relations that has many use cases: **OMG’s M0–M3 meta modelling**,⁶ with a more generic version introduced in [12, Fig. 7, p. 178]. The choice of representing the composition between *three* modelling levels (and *one* “level of reality”), is not arbitrary, but motivated by the following insights:

⁵The Wikipedia article **just referred to** presents a nice example of a **higher-order relation**, in the form of the **infobox**: that is a filled-out version of a (more abstract) meta model, which is represented as a list of **key-value pairs** that are necessary to identify every *instantiation* of the represented relation.

⁶The **mnemonics** of the M0–M3 abbreviations in the modelling order relation is that (i) the M stands for “model”, and (ii) the number represents the number of M’s in the symbolic label of a level.

- a **model** (M1 level) is a **formal** representation of a set of **entities** and **relations**, with their **properties**, that allow software:
 - to process the **digital twin** of part of the real world, before or after any real interaction with that real world.
 - “to reason” about the real world, in order to make informed decisions (“plan”) about how “to act”.

In this document, a model takes the form of a **property graph**.

Newton’s second law, $f = ma$, is a model of the relation between the entities “mass” m of a “rigid body”, “force” f exerted on it, and its “acceleration” a . These entities have properties which are numerical values that live in a **manifold**.

- the **real world** is not a model, hence the acronym M0, where the “zero” represents the fact that (by axiomatic definition) the real world needs zero models to work.
- a **meta model** (M2 level) is a formal representation of the set of **constraints** that must be satisfied between the expressions in the M1 model, before that model is accepted as **syntactically well-formed**. That constraint satisfaction is the meaning of the expression that “*the model m conforms to the meta model M* ”.

A stronger form of meta model goes beyond mere grammatical well-formedness, and aims for **semantic** well-formedness: all parts in the model “make sense”, have “meaning” in the context of the domain for which the meta model was designed.

A particular model can have more than one meta model that it conforms to. For example, **Newton’s second law** conforms to the meta models of:

- **mechanics**. For example, the numerical value of a mass is *constrained* to be always a positive real number.
 - **linear algebra**. For example, the *constraint* that a doubling of the force corresponds to a doubling of the acceleration, when the mass remains constant.
 - **mathematical analysis**. For example, the *meaning* of the **time derivative** of a continuous function, as a *constraint* between the function values at different moments in time.
 - **physical units**. For example, the numerical values of force, mass and acceleration are given units that must be dimensionally **consistent**.
- a **meta meta model** (M3 level) is a formal representation of the correspondences between two or more meta models, that is, the entities, relations and constraints that exist between (parts of) the meta models. Commonly used correspondence relations are:
 - **thematic relation** and **theta roles**.
 - **equivalence**.
 - **equality**.
 - **identity**, in **various forms** and **instantiations**.
 - **similarity**.
 - **logical** and **functional** inverses.

The pragmatic relevances of meta meta models are:

- that **translations** between models created in two modelling languages are only possible *if* and *where* there is such a **formally represented overlap** in the semantics of the modelling languages. That overlap is sometimes called the **canonical model**.

- as a loose synonym for **paradigm**: the essential concepts (entities as well as relations) in a domain are given a name. This **nomenclature** then a constraint on designers of more concrete domain meta model: the *minimum* they have to do is to represent all concepts in the meta meta model.

The labels M2 or M3 is *not* a property of a particular model or modelling language. Rather, it is an *attribute* in a “higher-order relation” that puts the model in a broader **context** of a set of models. In other words, the labels M1, M2 and M3 are always *relative within one particular context*, and never make sense as standalone tags. For example, the **Special Euclidean group in three dimensions** is a meta meta model for rigid body motions, but it has itself several other meta models, such as that of the **Euclidean space** or of **groups**. As mentioned before, there is no end towards the “top” of meta modelling levels; but the M0-M3 pattern comes back, over and over again, as a *best practice* in model-driven engineering.

This M0–M3 structure is a meta meta model, or paradigm, in itself: it represents the *belief* of designers that a particular composition of models can be considered together as a “connected”, “complete”, “consistent”, “relevant”, “essential”, . . . representation of a domain. In addition, the discussion in this Section is a proof that the M0–M3 structure can also be applied to any set of abstract concepts that humans use to attach meaning to their observations of that real world, or to invent meaning in imaginary worlds: models do not exist in the real world, only in the minds of people.

A *meta model* is also called a *modelling language* (i.e., a language to write models in), or a **Domain-Specific Language** (DSL) when it is designed specifically to make the modelling job easier for practitioners in a particular application domain.

1.2.17 Policy: metadata of an entity inside a model — `Semantic_ID`

The motivations to add semantic metadata to all entities and relations in models in a systematic way are:

- the topological structure

$$\text{model} \leftrightarrow \text{meta model} \leftrightarrow \text{meta meta model(s)}$$

is domain knowledge. It is worth representing that domain knowledge explicitly, because then software tools can use it for semantic “graph search” purposes.

- sooner or later, any model will have to be connected to new information, and so any system will become part of an even larger system. Again, explicit metadata helps to automate such model compositions.

The just-mentioned model compositions often represent “knowledge about the knowledge”. For example: the **reasons why** relations between entities exist, or in what ways an abstraction can be turned into a concrete instantiation, and used to configure software components.

A major consequence of (i) the *entity-relation* meta model for knowledge representation, and (ii) the practical usefulness of the M1, M2 and M3 modelling levels, is that this document suggests to adopt the *policy* to give every entity in a model the following set of two complementary *metadata*:

1. semantic_ID: a set of the following three *unique identifiers*:

- **ID** (“entity ID”): a unique identifier with which the entity can be referred to, unambiguously, in another part of this model, or in other models.
- **MID** (“model ID”): a unique identifier that refers to the model in which this entity plays a role. An alternative name is the “*type*” of the entity.
- **{MMID}** (“meta model UIDs”): a **set** of unique identifiers that each point to one of the meta models of this model, needed to interpret the meaning of the model.

The purpose of the `semantic_ID` is to provide each piece of software that works with the graphical model, with the (**symbolic**) information about where to navigate to and find the semantic information that is available, at the indicated “place”, about an entity in a model. Such a **semantic search** can even be supported *at runtime*, because it is efficient to store the extra symbolic metadata information in binary versions of compiled software code.

2. edge_IDs: two sets of identifier pairs for every entity in the graphical model:

- **{outE}** (“outgoing edge UIDs”): a **set** of *pairs of identifiers*, where each pair identifies (i) one outgoing edge connected to the entity, and (ii) the entity with which that particular edge makes a connection.
- **{inE}** (“incoming edge UIDs”): the same, but for the incoming edges.

The purpose of these two identifier sets in the `edge_ID` is twofold:

- to provide the semantic information about whether an entity is a *relation* that has other entities as *arguments* (the case of the outgoing edges), or whether the entity is an *argument* in itself, in the *relation* pointed to by the incoming edges.
- to serve as **indices** to speed up the above-mentioned semantic search in a graphical model. Again, irrespective of whether the metadata is stored in a text-based “source code” form, or in compiled “binary” form.

The pragmatic cost of the two types of metadata is low: a unique identifier can just be an integer. (Its uniqueness need only hold in the context of its own meta models, because the identifiers of the latter provide extra disambiguation information.) There are some standards that can be used to represent the unique identifiers: **Universally Unique Identifiers**, **Universal Resource Identifiers**, and **Internationalized Resource Identifiers**; the latter one is the last in evolution towards making the web more agnostic of its Western world origins, and is the preferred choice for new modelling developments.

1.2.18 Examples and types of higher-order relations

This Section lists some examples of higher-order knowledge relations, and how graph traversals could use these relations to answer queries:

- *cybernetics and robotic systems*: they not only *observe* the physical world, but provide *control* around those observations to steer the physical world towards a desired state, [44].

- *intent and configuration of tasks at various levels of control*: every robotic system has motor controllers to create motion, and sensor processors to perceive the state of the world. Both activities are eventually driven by the “intent” of the task. The *Action-to-Mission Association Hierarchy* (Fig. 6) provides a set of relations that link task intent to sensor and motor data.
- the concept of a *singularity* in the mechanical configuration of the *kinematic chain* of a robot, relates the values of the chain’s joint positions with the chain’s geometrical model, and with its energy transmission capabilities. The knowledge *that* something like singularities exist is already a higher-order knowledge. The knowledge *how to assess/compute* singularities is yet a level higher in the modelling, because it relates the singularity knowledge to particular numerical solvers. Another higher level of modelling is to connect the singularity concept to loss of motion capabilities of the robot’s kinematic chain.

Hence, in case the robot’s actual motion deviates “too much” from the sensed motion, the hypothesis of being in a singular configuration is relevant to explore. But that hypothesis comes only in view of the robot controller if the latter could do the above-mentioned *cause-effect chain* reasoning, by traversing the mentioned relations from “effect” to (possible) “cause”. Finding solutions to the singularity problem requires yet another higher relation, that connects the singularity concept to skills that provide alternative ways to satisfy the same task constraints by reconfiguring the kinematic chain.

- the context of the *requirements of the task* the robot is executing: not every *geometric singularity* is also a *task singularity*. For example, pushing a load with fully stretched arms can be a good approach to reduce the amount of force needed in the muscles/motors, while it *is* a geometrically singular configuration.
- *semantic maps*: collections of geometric primitives, with relations (“semantic tags”) on top that link some geometric primitives in the map to meaning in an application context.
- *semantic localisation and tracking*: using knowledge relations to support the decision making about what sensor processing algorithms to use on which parts of the sensor data, and to find out where a robot is on which part of a map.
- *configuration* of constraint solver algorithms: which constraint and objective functions to use, how to initialise the solver, which monitors to add for deciding whether the solver has reached a desired result or not,...
- *configuration* of software architectures: which communication streams to use, with which mediators, which data models, which communication patterns, etc.
- ...
- *transitive, converse, identity, associative, commutative, symmetric, asymmetric, reflexive or equivalent* relations.
- *constraint* (Fig. 1.6) which are relations that put limits on the values of some properties in some entities connected by another relation.

- **tolerance**: the intervals within which the values of a constraint relation must fall.
- the *operator* that is naturally connected to any relation is in itself a higher-order relation for that relation: while the latter models the connection between several properties, the related operator has the extra knowledge about *how to configure* the values of these properties in a particular instance of the relation.
- the *semantics of natural languages* have a hierarchical structure of interconnected higher-order relations, the so-called **hypernyms and hyponyms**.⁷ For example, *action* is more abstract than *motion*, which is more abstract than *grasping*, which is more abstract than *pinch grasping*. Or, *seeing* is more abstract than *recognizing*, which is more abstract than *localising*, which is more abstract than *tracking*.
- **context**: the “background” knowledge that influences the property values of entities in relations. For example, that the gains in a motion control feedback loop depend on the safety requirements of the application in which the motion control is being used.
- **S-expression** gives structure to mathematical relations.
- **level of measurement**: nominal (types), ordinal, interval, and ratio.
- **taxonomy**: a tree-structured relation on entities.
- **directed acyclic graph** (DAG): a graph-structured relation on entities with particular ordering constraints.
- **causal** relation, **causal chain**.
- **dependency graph**: any relation modelling the knowledge that one particular relation can only be applied in a particular context *if* other relations are also applied. A dependency relation often comes with an *order* (partial or strict) of those applications.

(TODO: other relevant types: *theory of mind* [107]: what do agents know about what other agents know? **action languages**, **transition system**, **abstract rewriting systems**, **Kripke structure** and **semantics**, **universal algebra**, **fluents**, **graph rewriting**, **situation calculus**, **event calculus**, **constraint satisfaction problems**. Most of these are mathematical models of little practical use, except for fluents and constraint satisfaction problem solvers.)

1.2.19 Mechanism: querying & reasoning via graph matching & traversal

A graphical model is often a **static** representation of knowledge. But **behaviour** can be added to such models by means of various **graph operators**. The mainstream ones change the *structure* of the graph, or return properties of that structures (e.g., the **diameter** of a graph). Because of its context of knowledge-driven engineering, this document emphasizes *semantic* graph operations, that is, **reasoning** and **querying**. These are the two major mechanisms with which to realise reasoning and querying:

- **graph matching**. One gives a *template graph* (also called a **frame**) as input, and the output is (the set of) matching sub-graphs in the queried graph.

⁷The terminology for these relations is another name for what this document has called the **is-a** relation, and its inverse.

- **graph traversal.** The input is a data structure (“declarative programme”, “query model”) that encodes the following three mandatory parts:
 1. at which node to start the query answering;
 2. in which order to follow edges and nodes further in the graph to find the answer;
 3. what **side effect** computations to do for each visited node and traversed edge.

Obviously, to achieve *efficient query solvers*, one must find a way to exploit the above-mentioned **higher-order knowledge** about the graph:

- the knowledge that the answer must be searched by following particular sets of identified edges (“relations”) in the graph.
- the knowledge in the **Semantic_ID**, that connects “instances” of information to their higher levels of abstraction.

In summary, these three key components in a knowledge-driven system are closely coupled:

- **knowledge graph:** this one has, obviously, knowledge relations inside.
- **query:** posing a query to the graph requires a complementary type of knowledge itself, namely about how relations are encoded in the knowledge graph, and connected to each other.
- **solver:** the solving of a query adds a third type of knowledge, namely about how to exploit the two other types of knowledge in getting towards an answer to the query.

Graph traversal is an approach towards realising the **holy grail** of **higher-order reasoning**. The simplest form of graph traversal is the one that works on tree structures. And **tree serialization formats** such as **XML** and **JSON**⁸ have tree traversal query languages; e.g., **XPath** (and its superset **XQuery**), or **GraphQL**. Mainstream reasoning frameworks, like the ones built around Prolog, **OWL**, or **SWRL** cannot provide such higher-order operations, because their modelling remains at *first order*: it does not have the semantics to represent *relations about relations*.⁹

The concept of traversal can be illustrated by means of the simple example in Fig. 1.6: in order to find out which constraints should be put on the values of the **properties** node of the **entity3** node in the **Relation**, one can follow the arrows from the **Relation** node to the **Constraint** node and its **properties** node. The direction of the arrows reflect *meaning* in a relation in a model, but these arrow directions do not constrain the traversal through the graph: the **edge_ID metadata** allows to “navigate against the arrows” efficiently.

1.2.20 Storage and reasoning in property graph databases

The **storage** of *directed graph* representations as in Fig. 1.3 is realised by **graph databases**, that provide **implementations** of property graphs [8]. More in particular, they support the mereological and topological aspects of *entity-relation models*: they keep track of which entities are connected to which other ones (via the **edge_ID** meta model, or a variant thereof), and of the properties of each and every node (via a **property graph** mechanism). And they support the specification and execution of queries via graph matching (always) and/or graph traversal (in still rare cases).

⁸An insightful and concise comparison between XML and JSON can be found [here](#).

⁹This is also a main shortcoming of one of the most prominent modelling languages in engineering, namely the **Unified Modelling Language** (UML). SWRL *does* allow *relations on relations*, but is semantically limited to *logical* rules, being based itself on the **RuleML** ecosystem.

Of course, this interlinking of models via property graphs must stop somewhere. In the case of (robotics and cyber-physical) software systems, this “grounding” takes place at two “ends” of the graphical model:

- at the “*bottom*”: (a model of) a piece of concrete software is composed with the set of models for which the software reflects the behaviour.
- at the “*top*”: (a model of) the human operator inputs, in the form of manually formed queries, in a *domain specific language*.

In both cases, it is the responsibility of human experts to validate that the software and the queries are correctly and consistently realising what is represented in the models. (The software artefacts themselves are *not* stored in the graph database, but only their [metadata](#).) It is a responsibility of the community in a particular domain to decide what grounding that domain will expect, for what kind of purposes. For example, formal verification expectations are a lot lower for educational robotic systems than for planetary rovers and manipulators; hence, also the accepted level of grounding will be more stringent in the latter case.

Various types of **reasoning** are needed on the models of (software) systems, to serve various complementary purposes. For example:

- code configuration and generation,
- model validation: “*does the system specifications conform to the application’s requirements?*”.
- model verification: “*does the system implementation conform to its specifications?*”.
- model certification: “*is there an official organisation that confirms that your system implementation is validated and verified?*”.
- dialogues with human users and between different computer systems.

If all models are stored in a graph database, reasoning is realised by means of the matching/traversal query language of the graph database. Some examples of tools that support such graph traversals are [Gremlin](#), [SPARQL](#), or [Cypher](#). An early standard, [Topic maps](#), has apparently been forgotten by the community.

1.2.21 Host languages to store, exchange and query models

Only very few formal languages are designed to support the exchange of models between graph databases and other software agents. [EXPRESS](#) is a modelling language with already a mature history and industry-backing, to represent product data, institutionalized in the [ISO Standard STEP](#). More recently, modelling languages were born in the context of the “[semantic web](#)”, namely [RDF](#) and, especially, [JSON-LD](#). These modelling languages have **built-in support** to represent *named directed graphs*, and (in the case of JSON-LD) keywords standardized in the language to represent **unique identifiers** for **context**, **identity** and **type**. (The latter corresponds, one-on-one, to the contents of the `Semantic_ID` of Sec. 1.2.17.) The `@context` allows a model to point to an “external” model, in a purely symbolical way; the `@id` allows models (external as well as internal) to symbolically point towards any entity in a model. Together, these simple semantic additions facilitate *composition* of models with very low coupling, the testing of *conformance to meta models*, and the representation of *higher-order relations*.

[XML](#) is probably the most popular “host language”, with an ecosystem of tools, developers and users that is, still, an order of magnitude larger than those of JSON-LD and RDF, but it

is designed to represent only *trees* (implicitly, via the containment constraints on XML tags) and not graphs. There are XML-based extensions such as [Xlink](#) that provide the *mechanism* for cross-linking, but not the *semantics* of “context” and of “entity IDs”.

Therefore, this document adopts JSON-LD as its preferred host language, but it does not introduce any dependency on that specific choice. Here is a possible encoding in JSON-LD of the simple model in Eq. (1.1):

```
{
  "@context": {
    "generatedAt": {
      "@id": "http://www.w3.org/ns/prov#generatedAtTime",
      "@type": "http://www.w3.org/2001/XMLSchema#date"
    },
    "Entity": "IRI-of-Metamodel-for-EntityRelation/Entity",
    "Relation": "IRI-of-Metamodel-for-EntityRelation/Relation",
    "EntityPropertyStructure": "IRI-of-Metamodel-for-EntityRelation/Properties",

    "RelationName": "IRI-of-Metamodel-for-Relation/Name",
    "RelationType": "IRI-of-Metamodel-for-Relation/Type",
    "RelationRole": "IRI-of-Metamodel-for-Relation/Role",
    "RelationNoA": "IRI-of-Metamodel-for-Relation/NumberOfArguments",

    "MyTernaryRelation": "IRI-of-Metamodel-for-MyTernaryRelations/Relation",
    "MyTernaryRelationType": "IRI-of-Metamodel-for-MyTernaryRelations/Type",
    "MyTernaryRelationRole1": "IRI-of-Metamodel-for-MyTernaryRelations/Role1",
    "MyTernaryRelationRole2": "IRI-of-Metamodel-for-MyTernaryRelations/Role2",
    "MyTernaryRelationRole3": "IRI-of-Metamodel-for-MyTernaryRelations/Role3",

    "TypeArgument1": "IRI-of-MetaModel-for-Argument1-Entities",
    "TypeArgument2": "IRI-of-MetaModel-for-Argument2-Entities",
    "TypeArgument3": "IRI-of-MetaModel-for-Argument3-Entities",
  },
  "@id": "ID-Relation-abcxyz",
  "@type": ["Relation", "Entity", "MyTernaryRelation"],
  "RelationName": "MyRelation",
  "RelationType": "MyTernaryRelationType",
  "RelationNoA": "3",
  "generatedAt": "2017-06-22T10:30"
  "@graph":
  [
    {
      "@id": "ID-XYZ-Argument1",
      "@type": "TypeArgument1",
      "RelationRole": "MyTernaryRelationRole1",
      "EntityPropertyStructure": [{key, value},... ]
    },
    {
      "@id": "ID-XYZ-Argument2",
      "@type": "TypeArgument2",
      "RelationRole": "MyTernaryRelationRole2",
      "EntityPropertyStructure": [{key, value},... ]
    },
  ]
}
```

```

    "@id": "ID-XYZ-Argument3",
    "@type": "TypeArgument3",
    "RelationRole": "MyTernaryRelationRole3",
    "EntityPropertyStructure": [{key, value},... ]
  }
]
}

```

The following model represents a **constraint** on the previous model (using the constraint language **ShEx**), namely the equality between the numeric value of the `RelationNoA` property and the actual number of arguments in the `Relation`:

```

{
  "@context": {
    "RelationNoA": "IRI-of-Metamodel-for-MyTernaryRelations/RelationNoa",
    "MyTernaryRelation": "IRI-of-Metamodel-for-MyTernaryRelations/Relation"
    "length": "IRI-of-Metamodel-for-the-lenght-function/length"
  },
  "@id": "ID-RelationConstraint-u3u4d8e",
  { "@context": "http://www.w3.org/ns/shex.jsonld",
    "type": "Schema",
    "shapes": [
      { "id": "MyTernaryRelation",
        "type": "Shape",
        "expression": {
          { "type": "TripleConstraint",
            "predicate": "RelationNoA",
            "value": { "type": "NodeConstraint",
              "datatype": "http://www.w3.org/2001/XMLSchema#int",
              "value" : "{length(MyTernaryRelation)}"}
          }
        ] } }
    ] }
}

```

1.3 Core relations: conforms-to and is-a

This document considers two relations as the core of knowledge modelling, [12]:

- the **is-a** relation, to represent levels of abstraction on **properties**.
- the **conforms-to** relation, to represent levels of abstraction on **relations**.

1.3.1 conforms-to hierarchy on relations

A **meta model** (or *schema*) provides the entities, relations, and constraints with which to decide whether a particular model is (**syntactically**) **well-formed** and (**semantically**) **meaningful**. In other words, a meta model is a model of a language in which to write models; each such model must satisfy the **conforms-to** relation (rather, *constraint*) with respect to each of its meta models, that is, all constructs that are being used in the model satisfy the constraints on the relations that are made explicit in a meta model, but only *for as far as* the constructs in the model indeed use entities that are defined in a meta models. This **multiple conformance** property does *not* imply **multiple inheritance**, that is, that *all* constructs

in a model must conform to the constraints of the meta models. (This is a property of the **is-a** relation.) This “underconstraining redundancy” property allows composition with any new model and meta models, *if* these do not contradict constraints introduced by earlier meta models. This property of composition is sometimes referred to as the **open-world assumption**.

For example, Equation (1.1) was introduced as a *mereological* model, because one can identify the “parts” and the “whole” entities, and the **has-a** relations between them. Sentences in a natural language must satisfy the **syntax** rules (that is, *form*) of that language, but also its **semantics** (that is, *meaning*), and none of these have already been constrained by the mereological relations.

Another example is that any property graph **conforms-to** the mathematical meta meta model of **graphs**, that is, nodes connected with edges, independently of the nodes’ interpretation as “entity”, “relation” or “properties”.

The difficult but important responsibility in making a meta model is to identify and formalize all the **constraints that have to be satisfied** in a model before that model really carries the “meaning” that is intended, and nothing more or less. For example, a kinematic chain is constructed by joining links, joints and tool frames, but even obvious constraints such as “a kinematic chain consisting of just one link connected to three joints is not valid” must be expressed in one way or another.

(TODO: no inverse relation. Hierachy is tree, with fan-out in the direction of more abstraction/less concreteness.)

1.3.2 is-a hierarchy on properties

(TODO: instance, object, class; invers relation is **type-of**. Hierachy is tree, with fan-out in the direction of less abstraction/more concreteness.)

1.3.3 Composition and inheritance

TODO: composition extends behaviour by introducing *independent* new entity, with relations to the entities being extended that contain the coupling between the new behaviour and the already existing behaviour. Inheritance extends behaviour by introducing a *dependent* new entity, with the **is-a** relation, which only adds new behaviour. The strictest constraint on inheritance is **Liskov substitution principle**: any entity (“object”) can replace any of its ancestors. Applied in game development, under the name of **Entity-Component System**.

Some not-so-good practices in this context:

- **Industry Foundation Classes**: an “ontology” to represent structures in buildings, like windows, doors, stair cases, etc. But [here](#) is an example where deep inheritance trees are making this kind of modelling very non-composable, and (hence) extremely large and complex because they *have* to model everything themselves, and cannot reuse bits and pieces from other ontology representations.
- **URDF** (Universal Robot Description Format) is a modelling language in robotics, suffering from the same inheritance explosion problem: every new addition must find its place somewhere in the inheritance tree under the “**God Object**” robot at the root of the tree, and this compromises composition.

1.3.4 Best practice: four levels in is-a and conforms-to hierarchies

No model of the real world, or of an engineered system that controls (part of) the world, can or must cover all possible aspects of that world or of its engineered instrumentation. The *selection* of the aspects of the real world that are included in a model defines the **level of abstraction**: any use of the model, in whatever context, will have “to make abstraction from” the non-modelled aspects.

A second, complementary, way to reduce the correspondence between a model and the real-world entities that it represents, works by reducing the **level of granularity** in the model. For example, one can put a building on a map just by adding a tag with its name attached to a particular map coordinate, or one can draw a polygon on the map of the outline of the building, or one can add a 3D CAD model. In none of these three approaches, the building is abstracted away, because it can be part of modelling relations. What *is* abstracted away by the just-mentioned geometrical models of the building are its real-world properties such as material usage, function, energy consumption, etc.

The third, also complementary, way is to introduce **level of resolution** in the model: what is the accuracy with which to represent space, time, force, etc.

The **OMG M0–M3** meta modelling paradigm identifies four levels of abstraction for the **conforms-to** relation, that always make sense, *together*. In the more restricted context of **is-a** and inheritance, the **Meta-Object Facility** (MOF) is a similar four-level meta model paradigm; it has a “sister paradigm” in the **Eclipse Modeling Framework**, namely **Ecore**.

1.4 Mereology: most abstract representation level

A property graph has entity nodes with properties, and relation nodes between entities. The Sections above focused on the *graph* view; this Section describes the complementary *semantic* view, more in particular, to describe what is the *least amount of meaning* that one can give to an entity-relation graph.

1.4.1 Mereology: has-a

Humans are trained to interpret the textual representation (i.e., “model”) of a relation,

$$\text{Relation}_x (\text{Entity}_1, \text{Entity}_2, \text{Entity}_3), \quad (1.1)$$

with a lot of background knowledge. The simplest interpretation (often referred to as the “highest” **level of abstraction**), is that of its **mereology**.¹⁰ A mereological model just represents the **parts** that make up the “world”, without any additional structure or behaviour. In this case, the parts are **Relation_x**, **Entity_1**, **Entity_2** and **Entity_3**. In other words, a mereological model consists of the **set**, or **collection**, of the **Relations** and **Entities** that are relevant for the modelled system. Remark that the **context** to interpret the meaning of the relation is not specified explicitly; at the mereological level, such a context is just another, larger, mereological set, often called the **universe** or the **domain of discourse** of a model, and it represents all the entities and relations that should be considered together before one can hope to interpret the meaning of the model unambiguously.

¹⁰**Holonomy** and **meronomy** are related terms to denote the symbolic relations that humans attach to “parts” and “wholes”.

The **formalisation** of the mereological view on models comes with only one single **relation**:

- **has-a** (or **holonym**), to represent the fact that a “whole” consists of “parts”. (The inverse relationship is often called **part-of**, or **meronym**.)

and one single **entity**:

- **collection**: the entity that “owns” the **has-a** relations with all the entities “inside”. Note that it does not own the entities themselves!

The **collection** entity can get an **attribute** that represents how an application using the **collection** interprets the order in which the elements in the **collection** are provided in the model: **ordered** or **unordered**. In the **JSON-LD** modelling language, this attribute is **represented by** the “**@list**” and “**@set**” keywords, respectively.

For example, the “model” in Eq. (1.1) has eight instances of the **has-a** relation:

- the “whole” of the context is a **collection** with **has-a** relations with all its primitive “parts”, **Relation_x**, **Entity_1**, **Entity_2** and **Entity_3**.
- the “whole” of the **Relation_x** is a **collection** with **has-a** relations with its three argument parts, **Entity_i**.
- similarly, all entities have **has-a** relations with a **properties** data structure entity.

Figure 1.3 is one (of the many possible) graphical representations of the mereology of Eq. (1.1). Figure 1.6 extends the model with a *constraint* on the relation, and a *tolerance* on this constraint; both bring in “loops” in the representation. The suggested approach of model **composition** has the advantage that the directed graphical models have only *acyclic* loops; this **pattern of cycle-free** composition is a **best practice** that is sometimes called the **dependency inversion principle**, and that this document tries to follow as often as possible.

1.4.2 Topology: contains, connects

The **topological** version of Eq. (1.1) is as follows:

$$\text{Relation}_x \text{ (Argument}_1 = \text{Entity}_1, \text{ Argument}_2 = \text{Entity}_2, \text{ Argument}_3 = \text{Entity}_3 \text{)}. \quad (1.2)$$

The extra information in this “model” is that each argument **Entity_i**, $i \in \{1, 2, 3\}$ is connected to a specific **role** in the **Relation_x**. That means that extra **structural** knowledge is added to the mereological model, namely that of:

- **containment**: all arguments are contained in the relation, and that container provides the *inward-looking* context that determines the interpretation of the properties of the entities that are used in the relation.
- **connection**: **Entity_i** is connected to the *i*th argument of the **Relation_x**, and this explicit association allows to reason about how to interpret the meaning of that specific entity in that specific role, again within, both, the inward and outward contexts.

It is clear that the **formalisation** of the topological view of the “model” in Eq. (1.1) comes with two **relations**:

- **contains**: this represents a **partial order** structural relation between the entities involved.
- **connects**: this represents a symmetric structural relation between the entities involved.

and with two **entities**:

- **container**: the entity that “owns” the **contains** relations with all the entities “inside”.
- **connector**: the entity that “owns” the **connects** relations with all the connected entities.

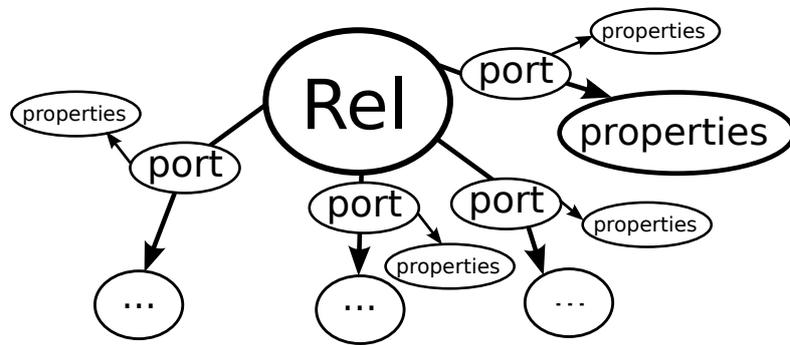


Figure 1.8: A directed graph representation of the **topological** model of the relation in Eq. (1.1). The arrows represent **connects** relations, which add extra structural information compared to the mereological model, namely the connection with a specific “port” entity that describes the *role* of an argument “part” in the “whole” relation. Since the port is an entity, it also has properties, not in the least the information about the type of the role.

1.4.3 Mechanism: block-port-connector

This Section refers to the generic **Block-Port-Connector** (*BPC*) meta meta model, [89], which has been used, in ne form or another, in software and systems engineering since many decades already. The BPC model is a *composition* of the generic (“meta meta”) relations (**is-a**, **has-a**, **represents**,...) with the following semantics:

- **Block**: every Relation **is-a** Block, so it **has-a** number of Ports.
- **Port**: each Port **represents** an argument in the Relation, and the **properties** of the Port **represent** the **type** of the argument, and the **role** the argument plays in the relation.
- **Connector**: this **connects**, through a Port of its own, a concrete **instance-of** an argument with a concrete **instance-of** a Port. The **type** of the Connector Port must, of course, match with that of the Block Port.

What is described above is the *outside* view on the Relation. The internals of the Block can be again a composition of Blocks and Ports and Connectors, then representing the “algorithm” that realises the **behaviour** of the Relation. The traditional graphical way to represent a graph (as in Fig. 1.8) is with nodes and arrows between nodes. This may be good enough for *human* consumption, because we *see* that is “*inside*” and “*outside*” of a node. To link the

outside and *inside*, the ports much get an extra modelling primitive, the **docks**: each port must have exactly one *inside* dock and one *outside* dock, and both have a *connector* between them (Fig. 1.9). The constraints on both ends of this connector are type compatibilities, as are the constraints between the inside and outside docks of a **port**.

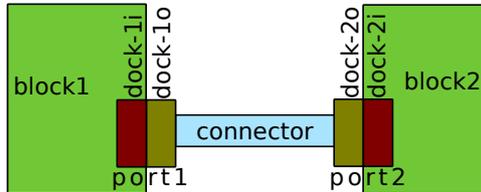


Figure 1.9: A computer-interpretable representation needs an explicit modelling primitive to represent the “*inside*” and “*outside*”: the dock. Each **port** has two **docks**, one to connect to the **port** from the inside, and one to connect to the **port** from the outside.

In the context of cyber-physical and robotics systems, BPC is *the* meta model for the composition between **components**, where “component” is any sub-system that has **behaviour**:

- Block: a component with *behaviour* inside.
- Port: a *view* (or “*interface*”) to (a part of) the behaviour in the component.
- Connector: a component whose behaviour is dedicated to the coupling of two or more Ports.

Blocks *always* have behaviour; Ports and Connectors *can* be behaviourless. For example:

- computer hardware has several ports to connect wires to, each interfacing with other computer devices, or to batteries or other power source devices. The devices are the *Blocks*, the ports are the *Ports* and the wires are the *Connectors*. Depending on the level of physical detail that is relevant for the application, the *Ports* and *Connectors* can be given behavioural models or not, in the form of their **electro-magnetic** and **thermal** properties.
- in **inter-process communication**, the communicating processes are the *Blocks*, each of them has a **socket** as *Port*, and the **session connection** (or, “channel”) between them is a *Connector*.
- when two robots execute a **task** together, each of them has a process that interfaces with a process on the other robot (via an IPC mechanism described above), and those processes have software representations of the behaviour of the task and subtasks (the *Blocks*), the information that the subtasks make available to each other (the *Ports*), and the **buffers** through which that behavioural information is exchanged (the *Connectors*).

“System”, “subsystem” and “behaviour” are **axiomatic** terms: they mean what the system designers want them to mean, and, to the best of the authors’ knowledge, any possible “definition” will always be **circular**.

1.4.4 Policy on block-port-connector mechanism: data sheets

(TODO: more details. Data sheet contains all information to allow usage of a port and a connection. Structural as well as behavioural information. Including constraints on the usage ranges, resource usage, dependency on other blocks or ports, etc. Hence, there will be a large amount of different data sheets, also on top of exactly the same implementation of the BPC mechanism. For example, safety setting for a mobile robot should depend on the context in

which they are deployed: working in an industrial setting with trained adult operators is very different from working in a hospital where curious children can be sharing the same corridor as the robot.)

1.4.5 Role of mereo-topological models

Mereological and topological models might seem overly simplified and obvious, but they have already a very important role to play in large-scale modelling efforts of digital robotic platforms: to determine what the models and reasoning tools can “talk about”, or, more importantly, can *not* talk about because of a lack of formally represented entities. So, a first agreement between the model developers in a particular domain is to get agreement about what terms are “in scope” of the effort,¹¹ and which are not, and what kind of dependencies between these terms will be covered by the models. That effort is exactly what this document is kickstarting, for the robotics sub-domains of *motion*, *perception*, *world models* and *task specifications/plans*.

For example, a kinematic chain is a relationship representing motion constraints between rigid body links and (typically) one-dimensional revolute or prismatic joints. The role of the links is to transmit mechanical energy (motion and force), while the role of the joints is to constrain or alter that transmission. Obviously, the order of the joints in the chain has an influence on the chain’s overall behaviour, and vice versa. Note that these sentences already contain a form of reasoning, such as: a robot has to have at least six joints to move its end-effector in all spatial directions; or, if a joint is not connected to a link, directly or indirectly via other links and joints, it cannot influence that link’s motion.

1.5 Design patterns

A (software) **design pattern** is a general **reusable solution** to a commonly occurring problem within a given **context**.¹² Major reasons why a design can be called a “pattern” are:

- it has been used in multiple real-world applications. In other words, it has proven “*to work*”.
- the design description explicitly refers to the various “**forces**”, that can pull the design into several (foreseen) directions. In other words,
- the design description explicitly discusses the **trade-offs** between choosing which forces to apply to a specific context, and to what extent.

The major top-level categories of patterns (in software, system development, modelling,...) are (i) the **structural** patterns, and (ii) the **behavioural** patterns. These are, not coincidentally, also two major categories of design aspects that appear often in this document. Since the latter’s focus is on *model-driven engineering*, the above-mentioned key mereological

¹¹This scope is sometimes called the “world”, or the “**universe of discourse**”.

¹²The concept comes from architecture (the bricks-and-mortar form of architecture, that is), via the **seminal work** of [6]. That book described patterns as follows: “*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*”

aspects of patterns (solution, force, context) will be modelled explicitly, as well as the relations and constraints that connect them. A good pattern model balances the **declarative** and **imperative** (or “procedural”) aspects of the description:

- the fact that the literature uses the semantic term “forces” to represent design choices indicates the preference for declarative pattern descriptions.

1.5.1 Policy: constraint, dependency and causality graphs

Many **higher-order** relations have the meaning of **constraints**: the relation represents a particular configuration (of properties in various entities or relations) that is not allowed, or that, on the contrary, is intended to be realised, etc. Even for simple systems, the designers must be able to model **dependencies** between sets of constraints (that is, even higher higher-order relations), that is, some constraints are only valid after some other constraints have been satisfied. For example, it only makes sense to take a *actuator saturation constraint* into account after the *motion control loop* that steers the actuator has been brought into operation.

Taxonomy of increasingly more constraining dependencies: connection \rightarrow relation \rightarrow constraint \rightarrow dependency \rightarrow **causality**.

Some important instances of the property graph meta model are the **constraint graph**, the **dependency graph** (Sec. 1.5.1), and the **causality graph**. They all formalize constraints that exist between the properties of nodes or relations. A dependency graph is a special case of a constraint graph, because “dependency” is a semantically richer type of “constraint”. Similarly, a causality graph is a specific type of dependency, in that it represents *cause-and-effect* dependencies.

Order: constraint (on parameters), then dependency (on relations), then causality (order of dependency)

1.5.2 Mechanism: Directed Acyclic Graphs for partial ordering

The **directed acyclic graph** (DAG) is a common abstract data type to represent **dependencies** in **order**, more in particular, to represent **partial ordering**. Two very common special cases of DAG structures are the **tree** and the **polytree**.

1.5.3 Policy: temporal order, hierarchy, causality

- **cause-effect chain**: **execution causality**, for triggering of component ports, scheduling of function executions in event loops, reasoning in graph traversals,...
- **temporal ordering**: (non)overlapping start and end events;
- **hierarchical ordering**;
- **model dependency**;
- **junction tree**: define a *spanning tree* for the graph, via domain-dependent choices of how to reduce a graph-connected sub-graph into one single node.
- ...

1.6 Declarative and imperative models of behaviour

Engineered systems are built “to do something” in the real world. The description of the desired behaviour of the system can come in two major forms: imperative and declarative. As any other type of models, behavioural models consist of *relations between entities*, and most often some *constraints* must hold between some of the *properties* in the entities and/or the relations. For example, *dependencies* between:

- the *order* in which *actions* must be *executed*.
- the *composition* of behaviour, for example, *hierarchy*, or *priorities*.

The composition of such dependencies results in a graph of relations in itself, a so-called *dependency graph* (Sec. 1.5.1). There are two major complementary ways to turn the information represented by an action dependency model into “actions”:

- **imperative**: the dependency constraints are “solved” explicitly, at **design time**. So, at **run time**, the system executes the resulting “**recipe**”. Which is sometimes also called the **control flow** of the system’s behaviour.
- **declarative**: the dependency graph is available at run time, together with:
 - a **query** coming from a “user”, or “software agent”, who wants some task to be executed. The query is a model in itself, representing *what* behaviour is desired, and *which extra constraints* have to be taken into account
 - a **solver** program that *computes* the “optimal” ordering, by solving **constraint satisfaction** or **constrained optimization** problems, on the composition of the dependency graph, the query, and the “background” knowledge space.
 - an **implicit invocation** mechanism that allows a running application to indicate and check which dependencies are met.
 - a **dispatcher** program that executes the “actions” in the right **context**.

In computer-driven systems, “context” has two complementary meanings:

- **run time**: the minimal set of **data** that must be saved to computer memory to allow the action’s **execution** to be interrupted, and later continued from the same point.
- **design time**: the minimal set of **relations** that are needed to determine the **meaning** of the action.

A declarative approach allows (but not necessarily guarantees!) to have *both* contexts available at runtime, which improves **composability**, **reactivity**, and (hence) **adaptability**, at the cost of more execution time and memory usage.

1.7 Hierarchical and serial ordering: scope

A key motivator for *model-driven engineering* (MDE) is the observation that the amount of software in modern (robotic) systems has grown so large that no human developers can keep an overview in their minds about everything, and more importantly, about the implications of interconnecting components into systems. However, a simple-minded introduction of MDE can lead to a similar mental overload of models instead of of code, which would mean that no significant progress has been made.

However, modelling has one large advantage over coding, and that is that there exist many ways to add structure between models that allow viewing a component or a system at various levels of abstraction. The mereology of the two major abstraction structures is as follows:

- **hierarchical order:** or, *taxonomies*. These are **trees** of models, where each depth level models an explicitly identified set of properties, relations and constraints.

For example, topology *implies* mereology (one can not talk about two entities being connected if the two entities have not been identified), etc. Kinematic families of serial and parallel robots, with further specialisations of 6R serial chains or Stewart-Gough parallel platforms, etc.

- **serial order:** or, *dependencies*. The most useful dependency ordering has the structure of a **Directed Acyclic Graph**, because this is a *declarative* way to model serial dependencies.

For example, execution dependencies between tasks determine whether they can be deployed at the same time or not. Data access dependencies between functions determine their concurrency scheduling order. Relative priorities between robotic systems determine the order in which they are given access to physical resources, such as space or energy.

The major usefulness of the hierarchical and serial ordering is that they allow to introduce **scoping relations** to the development process (but also to the runtime system analysis!): interpretation of information can be limited to that part of the presented orderings that has an impact on the current design or analysis, and “reasoning” can be done in a scope that is limited to, respectively, the highest level of hierarchical abstraction or the smallest set of serial dependencies, that make sense. Examples of such “scoped reasoning” are:

- every topological relation *implies* a mereological one: it does not make sense to reason about interactions between entities if these entities have not been created and identified.
- every coordinate representation *implies* a geometric relation: one does not need to look at the exact numbers or data structure in the coordinate representation of frames to detect whether their type or their physical units are compatible or not.

Chapter 2

Meta models for behaviour: activities, their interaction and coordination

This Chapter applies the [knowledge representation](#) primitives to create **component**-based models of **behaviour**, introducing the following entities and relations:

- **activities** are the entities that realise the behaviour embedded in a system **component**. Every activity is the composition of:
 - multiple *data*, *functions* and *algorithms*, as the entities of **computable** behaviour.
 - one single *event loop*, as the relation that **orders the execution** of these computations.
- **interactions** are relations *between* activities, via which one activity can **influence** the behaviour of the **other** activities.
- **decision making** are relations *inside* one activity, via which it can **adapt** its **own** behaviour to whatever changes occur in the system.
- **composition** and **coordination** of all of the above, to enable **configuration** of an application's behaviour, **system-wide**.

This document's [paradigm](#) identifies composition, coordination and configuration of behaviours as just three special cases of decision making relations, but so generic and important ones that they are **first-class citizens** in the behaviour meta model.

Application developers design the composition of all of the above into **components**, optimizing the trade-offs in the following three design aspects:

- **separation of concerns** into the five major categories (the “5C's”, Sec. 9.3.4): *Composition* of a system via *Coordination of Computation* (“activity”) and *Communication* (“interaction”), with all of the above allowing *Configuration* of their data structures and functionalities.
- **composability**: the extent to which a **component** makes all of its “5Cs” **separately configurable**, to increase the opportunities **to reuse** that component in any kind of system.
- **compositionality**: the extent to which the behaviour of a **system** can be **predicted** based on the knowledge of the individual behaviours of the com-

posing components, and of their interconnections; hence making the system more easy **to build** as a composition of components.

Extra **design drivers** are **self-reflection**, **reactivity**, and **explainability**. None of these “**non-functional**” aspects can be measured objectively and directly, or even be modelled unambiguously. Hence, this Chapter introduces a collection of **best practices** and design patterns, to help human developers **to bring structure** in the complex system development process.

The **science and engineering disciplines** share knowledge models¹ that represent **interactions between matter, energy, data and information**, Fig. 2.1. That knowledge has been consolidated in the scientific domains of **physics**, **mathematics**, **computer science**, and **systems and control theory**. This Chapter adds **computational (meta) meta models** to this common knowledge, so that it can be used by computers in the control of cyber-physical and robotic systems. The **activity** meta model plays the central role in this modelling, because it integrates (“**composes**”) all other modelling primitives introduced in this Chapter.

A major **challenge** is the inherent **asynchrony** in, both, the physical world and the cyber world. Indeed, also the latter is full of **asynchronous systems**, in hardware (**multi-core CPUs**, **distributed computer systems**) and in software (**multithreading** and **multi-tasking**). This document’s **computational meta model** has the following primitives:

- **abstract data types and functions** are composed into a **synchronous algorithm**.
- algorithms are composed into an **event loop** for mutually **concurrent** execution inside one **activity**.
- the event loops of several activities are composed into the event loop of a **thread**, that embeds the **asynchronous** execution of behaviours to the **execution** on a CPU core.
- the **stream** is the data exchange mechanism between activities.
- event loops in different activities need **coordination** by means of three complementary mechanisms:
 - **protocol flags** for **peer-to-peer** coordination.
 - a **Petri Net** for the coordination of multiple activities via one **dedicated mediator** activity.
 - a **FSM** to coordinate the behaviour inside **one single activity**.
- the **process** composes the **execution** of several threads with a collection of **resources in the operating system**: CPU cores, files, I/O, . . .

This Chapter presents only the **mereo-topological** parts² of **domain-specific components**. The **behavioural levels of abstraction** are added in later Chapters, where the *robotics* application domain starts to appear as this document’s focus within cyber-physical systems.

2.1 Cyber-physical systems: interaction of matter, energy, information & data

A **cyber-physical system** is an interconnected set of man-made (or “**engineered**”) “**machines**” that operate on the physical world, and:

¹The formalization of the generic sciences of mathematics and physics is beyond the scope of this document. For now, it is assumed that these knowledge formalisations will become available sooner or later, for computer-driven engineering systems, with the appropriate level of detail.

²That is, the composition of the **mereological** and **topological** parts.

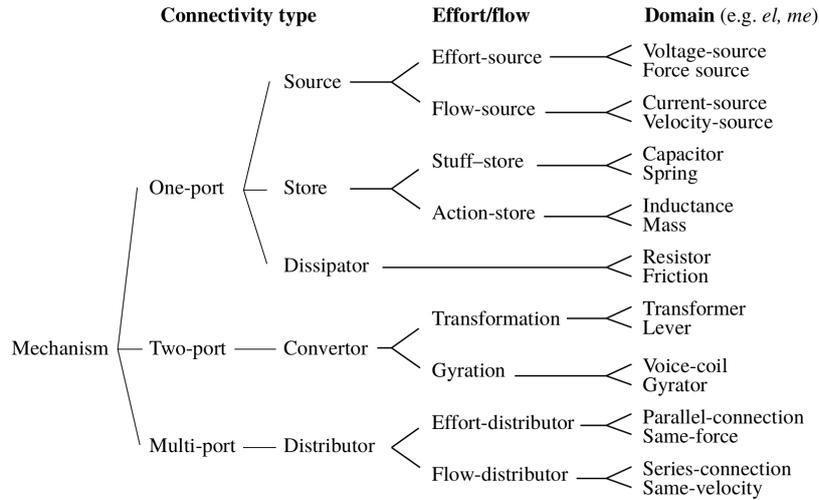


FIGURE 8. The taxonomy of physical mechanisms. The properties discriminating between the classes after branching are printed above the branch points. The classes on the right give some examples of mechanisms in the electrical and mechanical domain.

Figure 2.1: The topological relations between the various mereological top-level entities, according to [16].

- consist of physical components, such as mechanisms, chemical processes, belts, pipes, valves, etc.
- are **instrumented** with sensors to measure position, temperature, pressure, etc., to transform physical quantities into digital data,
- and with **actuators** (electrical or hydraulic motors, burners, etc.) that transform digital data into physical energy,
- are controlled via (so-called “**embedded**”) software, which computes the actuator outputs from (i) the sensor inputs, (ii) a **model** of the system, and (iii) a description of the system’s desired behaviour.

Human civilizations have spent tremendous efforts on the scientific (“mathematical”) modelling of the **physical** (or “**continuous**”, or “**hardware**”) parts. Powerful **scientific paradigms** have been created, which support most of the technological innovations of the human race. Figure 2.1 gives a summary of one of the most successful of such paradigms, that of *engineering ontologies* within a **bond graph** context. The engineering of the **cyber** [105] (or, “**discrete**”, or “**software**”) parts has a much shorter history, and a lot less completeness, consensus and harmony has been achieved in the domain of **software engineering**.

2.1.1 Physics: coupling of space, time, spectrum, matter and energy

(TODO:)

2.1.2 Cyber: decoupling of continuous, discrete and symbolic models

(TODO:)

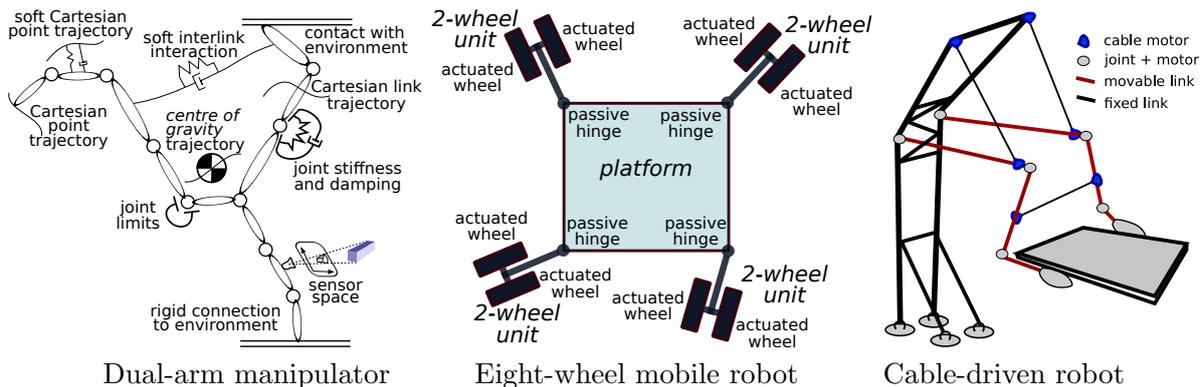


Figure 2.2: Sketches of various “advanced” robotic systems whose modelling is covered in this document.

2.1.3 State of a system

This document provides a large amount of *models* to represent cyber-physical systems, and applications engineered around them. That means that such an application will have dozens to hundreds of models, and hence often thousands of *parameters* in those models. Some of the models represent the **behaviour** of the system over time; the **state** of a system is the subset of all the system’s model parameters that (i) change over time, and (ii) are needed to describe the system’s dynamic behaviour. Several different *types* of state can be identified:

- *energy*: the amounts of energy that are stored in different parts of the system.
- *computational* state: the information needed to pause algorithmic computations for a while, and resume then at a later moment in time.
- *control flow* state: the information (the “*conditions*”) that determine which direction an algorithm is going to take at a particular moment in time.
- *behaviour coordination* state: the information (the “*plan*”) about what activities the system should run at a particular moment in time.
- *activity coordination* state: the information (the “*schedule*”) about what functions an activity should execute at a particular moment in time.
- *interaction* state: the information (the “*protocol*” activities that interact often follow a protocol of interaction that has various phases.

2.1.4 State representation: ordinal, categorical, continuous, discrete, event

(TODO: event represents that “something” has happened in one *activity*, is communicated to other activities, so that they can react to the event and change their behaviour.) **ordinal**: each data instance has its place in an order; **categorical**: each data instance has a type from a discrete set of **nominal categories**; **continuous**: in value (current, distance, temperature, power,...); with **intervals and ratios** as important sub-types.)

2.2 Data: structuring information for computations

One of the foundations of modelling is the representation of “**data**”: the symbols and numerical values that encode the *properties and attributes* of the entities and relations that make up the **information** part of a cyber-physical system. One (but not the only) **hierarchical structure** that relates formal representations of information is that of the **levels of abstraction**.³ The hierarchy comes from the observation that each of these levels is *a* (but not *the*) meta model of the level below, and has the level above as one of its own meta models. The following Sections describe this document’s five levels of abstraction; together with a “cross-sectional” data representation requirement, such as *metadata and queries*.

2.2.1 Mathematical representation

A **mathematical representation** is the (often axiomatic) definition of relations between entities that respect particular **invariants** under **transformations** of **formal mathematical models**. Traditionally, these models are written down in a symbolic form to be produced and consumed by humans only. This document will *not* suggest formalisations that are useable by computers in robots, but refer to them as *meta meta models* that are grounded “somewhere”. At least, that is the case with the basic mathematics of **algebra** (solving equations and polynomials, groups), **number theory** (natural, integer, rational, real and complex numbers), and **analysis** (differentiation and integration, series developments of functions).

The exceptions are **geometry** and **Bayesian information theory** (including **statistics**), because robot-processable geometric and uncertainty models are essential components in robotic systems. The document will make formal models of “polygonal worlds” and their “motions” over time. It relies on readers *understanding* the more theoretical aspects of geometry such as: the mathematical properties of *rigid body motion* as represented by the **SE(3) group**; the differential-geometric properties, such as **pull-backs** of **tangent spaces** and **multi-linear forms**, **exponentiation** or **logarithm**; and the lack of a **bi-invariant metric** to measure the “magnitude” of a motion.

2.2.2 Abstract data type — Semantic representation

An **abstract data type** is a **symbolic** form of a **knowledge graph**, that provides:

- entities and relations to store the **symbolic and/or numerical values** of the **coordinates** of (some of) the symbolic parts.
- the **operator** relations that are **allowed** to be applied to instances of the abstract data type.
- the **constraint** relations that the properties and their numerical values (if relevant) **must** satisfy in order to be meaningful.

One needs computers **to reason** with the symbolic representations of entities and the relations between them, but also **to compute** with the numerical values of properties. Such reasoning

³At least, for *computer-controlled engineering* systems, because there the needs to compute and to communicate between machines are essential. Sciences and humanities often stop with only the two top-most levels of abstraction, the first one for inter-human communication, the second one for computerised tools like **computer algebra systems**.

and computation can take many forms: to formalise algorithms into computational models; to check formally the validity of a model; to transform models into code; to decide which are the right “magic numbers” in algorithms, etc.

At this level of abstraction of model *representation*, one can represent various levels of abstraction of the modelled *domain*. For example, one can consider a robot as a machine that moves stuff around in space, or as a particular kinematic chain of links and joints, or with the explicit addition of motors and sensors. Whatever abstraction one works in, semantic properties that are always relevant are the **types** of entities and relations, and their physical **dimensions**, such as length, energy per time, force, or angle.

The numerical aspects require to link with the meta meta model of the **array** (or **matrix**, or **tensor**) as the ordered **list**, and even **list-of-lists** and **list-of-list-of-lists**, etc.

2.2.3 Data structure — Symbolic representation

A **data structure** model is the next step in bringing the models closer to executable software: it gives the abstract data types an explicit *software representation*, but still independent of any particular programming language.

At this level of abstraction, also the **quantitative value** of each property is added, and its a corresponding physical **unit**; for example, meter, inch, or millimeter for *length*, Newton for *force*; second or hour for *time*.

The purpose of the data structure is to represent abstract data types to a level of concreteness with which **to compute**, **to share** and **to store** data in all its variants (in memory, *marshalling/* and *serialising*, etc.). Primary examples are the **primitive data types** built-in in all programming languages: *integer* values, *floating-points* values, *string* and their *composition*, e.g., **arrays** and **unions** in the **C** language.

2.2.4 Digital structure — Data representation

One essential part of computing with data structures is **to express** them in a concrete **programming language**, **to store** them in the memory of a computer, or **to communicate** them between computers. So, one needs an extra level of representation, namely that of how many bits are being used to implement them on a particular computer hardware, and in what structural order the bits get their meaning in the data structure.

For example: the positions of a point in space can be stored as 32-bit IEEE floats, or communicated by JSON numerals; mathematical entities and operators can store **matrices** in compatible ways with **LAPACK** or **HDF5**.

The **C** language is more versatile than **JavaScript**, since the latter uses the *JavaScript Object Notation* (JSON) that allows to distinguish only between floating-point values and integers. Another example is the **Lua** language that provides only the concept of *number*. This affects not only numerical values, but also strings. For example, the **Python** language distinguishes between *strings* and *Unicode strings*.

(TODO: difference between **logical and physical** data structures; logical: 32 bits are interpreted as unsigned int, etc.; physical: 32 bits contiguous in memory.)

2.2.5 Electronic realisation — Physical representation

With modern computers, the **performance** of computations is strongly influenced by the electronic architecture of **CPU cores**, **computer memory** (including **caches** and the **physical layer** of communication. For example, computations on the **CPU registers** are orders of magnitude faster than those on higher levels in the **memory cache hierarchy**. Or **compare-and-swap** operations can avoid the **context switches** that are an inherent part of **locks** that come with **mutual exclusion**.

2.2.6 Symbolically linked data: metadata, database, query, index

In a **programming in the small** context, **abstract data types** and **functions** are directly coupled, because the data types are *explicit arguments* in a function **signature**, and the function has access to the data in its own memory space. This situation changes when the scale of a system grows:

- the data that a function need at a certain moment can be produced or stored in a very decoupled part of the system, so *relations* must be provided to help **communicate** that data from its “producer” to its “consumer”.
- only **declarative dependencies** are available as relations that contain the knowledge about how to find the data that is needed.

Often, both types of symbolical links occur together. **Solving** the symbolic links requires major resources, and clever architectures, because of the major challenges to keep data **consistent**, and to access it **efficiently** in large, dynamic systems.

Higher-order data — Metadata

Higher-order relations also exist for **abstract data types**: **Metadata** is the common terminology for all data that contains data about how other data must be interpreted. That metadata only becomes “information” in the context of a cyber-physical system’s control, when it comes with a formal meta model that can be interpreted by that system’s controller activities. Here is a non-exhaustive list of commonly useful types of metadata:

- **physical units**: what is the interpretation of the numerical magnitude of the data?
- **timestamp**: at what time was the data generated?
- **provenance**: how was the data generated? Timestamps are often a natural part of provenance metadata.
- **ownership**: which activity is allowed to change the data?
- **observability**: which activity is allowed to read the data?

In the systems-of-systems context of this document, metadata is always an **attribute** of data, because it is given to the data that one set of activities works with, by another, **mediator**, activity. Hence, ownership and observability are indispensable design drivers in that mediator pattern.

Data base: symbolically linked data

(TODO:)

Query: constraint between linked data

(TODO:)

Index: database of common queries

(TODO:)

2.3 Algorithm: synchronous composition of functions

This Section provides a **mereological** and **topological** model of an **algorithm**, as the *composition* of the three traditional parts of **programming in the small**, [29, 108]: **data structure**, **(pure) function** and **control flow**. In most application contexts, this composition includes a set of **dependency constraints** between some of the three mentioned parts of algorithms. For example, the choices for **magic numbers** coming from the application context in which the algorithm is executed; such as the gains in a **PID control** function. Even more such constraints can be expected to come from the ways how an algorithm is implemented in a programming language, and then compiled and deployed on a computer and its operating system. Examples here are constraints in the control flows to prevent **race conditions** between functions executed in **concurrently** running **threads**.

This Section conforms to the “5Cs” meta model, to classify functions in types that have a semantic meaning in the context of **architectures**. The “Composition” is realized via components, which are a container for the other four “C’s”. The algorithms in “Coordination” are typically just **Boolean functions**, that represent all *decision making* in a system; the “Communication” algorithms are typically **protocol stacks**; “Configurators” execute **configuration scripts**, possibly after **parsing** one or more **configuration “files”**. What ends up in the “Computations” parts is the rich variety of algorithms that belong to a particular application domain. System architects make two essential decisions when designing algorithms to encode the system’s computational behaviour:

- the *control flow* of an algorithm: the order in which the algorithm must execute its functions. In other words, the **coordination** of the *computations* that take place in functions.

For example, a **Kalman Filter** has “prediction” and “correction” parts that must be executed in the right order.

- *dependency injection* of a function’s *parameters*, to give them a value that is appropriate for that function’s use in a particular application. (This **configuration** of function arguments is realised via functions, too, of course, and these functions must end up somewhere in the algorithm’s control flow, too.)

An example of dependency injection is the setting of the control gains in each of the possibly many **PID controller** functions in the system. The *function* is always the same; its gain *parameters* are configured (by the user of the control algorithm) in the beginning and maybe at sparse times during the runtime of the user; and (the values of) the function’s *arguments* change every time the function is executed. These *configuration* parameters are *properties* of the *behaviour* of the function, because they determine the relation between the function’s input parameters and its output parameters. These input/output parameters are *not* part of the function’s behaviour.

The proper choice of control flow and configuration allows composition flexibility in algorithms

with high **runtime adaptability** requirements.⁴ However, in mainstream practice, they are seldom available as **first-class citizens** in the system design toolbox, not in the least because both are almost completely determined by the *context* of the algorithm, and not by the structure or behaviour of the algorithm itself. So, algorithm developers should not assume one particular execution context in which their algorithms will be executed.

2.3.1 Mereology-topological model

A description of an **algorithm** is as follows: “Starting from an initial state and initial input, the instructions describe a computation that, when executed, proceeds through a finite number of well-defined successive states, eventually producing output and terminating at a final ending state.” The mereological and topological entities and relations, with which to formalize this description, are:

- *mereological entity*: **data structures**, to represent the (“mutable”) **computational state** of an algorithm, and the “immutable data” of its **computational configuration**
- *mereological entity*: **functions**, i.e., the instructions, or computations, that “mutate” the computational state.
- *topological relation*: **(function) closure**, or “data binding”: that is, a **higher-order** data structure that represents the **connects-to** relation of a function with the data structures that serve as the function’s input and output.
- *mereological relation*: **execution** of a function on the data in its closure.
- *topological relation*: **control flow** (or **schedule**, or **activity diagram**), that is, a **higher-order** function that computes the order of the execution of other functions. Of course, in many use cases, that computation yields always the same result, so the schedule can be reduced to a data structure, namely an ordered list of functions.
- *topological relation*: **(function) invariant**, that is, a constraint (so, a **higher-order** relation) between (a subset of) the input and output data structures that is satisfied before and after a function has been executed, independently of the values of the input data.

2.3.2 Mechanism: data, function, control_flow, and algorithm

The descriptions of the previous Section give rise to the following formalisation in a *meta model*:

- **D-block**: the entity to represent **data (structures)**, via a *data model* that describes what are valid structural compositions of data.
- **F-block**: the relation to represent a **function**, that is, the computational element that has one or more D-blocks as its arguments, with some of them having the **role** of “inputs”, others of “outputs”, and some of both.

An F-block should be a **pure function**, that is, without only *explicitly visible side-effects*: the function changes the some of its output data arguments, and nothing else.

Other commonly used names for the F-block entity are: (*composite*) *operator*, *action*, or *actor*. F-blocks can easily be mapped to a *function prototype* (“signature”) in any specific procedural or functional programming language.

⁴If an algorithm and its functions need never to be adapted during the life time of their system, there is no need to introduce the dedicated mechanisms of control flow and parameter configuration.

- **S-block**: the relation to represent the **schedule** (or, **control flow**) of a collection of F-blocks, that is, **constraints on their execution order**. The model of a schedule is a D-block in itself, that represents the order in which the F-blocks must be executed. This order model can be **declarative** or **imperative**:
 - *declarative*: representatives are (i) the **control flow graph**, and (ii) its special case, the **directed acyclic graph** (DAG). The latter is introduced in Sec. 2.3.5, as this document’s composable mechanism to represent a **flowchart**.
 - *imperative*: this type has only one representative, namely the **while loop**, that is the mechanism behind the **event loop**.
- **A-block**: an **algorithm** is a composition relation, that is, the “architecture”, of all of the above, including a **collection** of constraint relations, such as invariants and closures. The *model* of an architecture is a D-block in itself, that contains the **UIDs** of the (i) composing blocks, (ii) the “**constraint**” or “**dependency**” relations, and (iii) its own **semantic metadata**.

So, the “most primitive” primitives of the meta model are F-blocks and D-blocks; the other types are *higher-order* version of the same mechanism, with a different meaning and role in a computational model. F-blocks and D-blocks are *connected* to each other, *because* an F-block *changes* some data in (some of) its D-block arguments.

2.3.3 Block-Port-Connector model for blocks

The structural part of the meta model **conforms-to** the **Block-Port-Connector** meta model (BPC) Fig. 2.3): the domain is the description of an algorithm, and it is obtained by specialising the entity **block** to F-block, D-block and S-block, and introducing domain specific constraints (and meaning) to the **connects** relation. As an example, **ports** represent the arguments of a function, and they are typed; that is they can be connected to a D-block under the constraint that the digital data representation model of the D-block and the port are compatible.

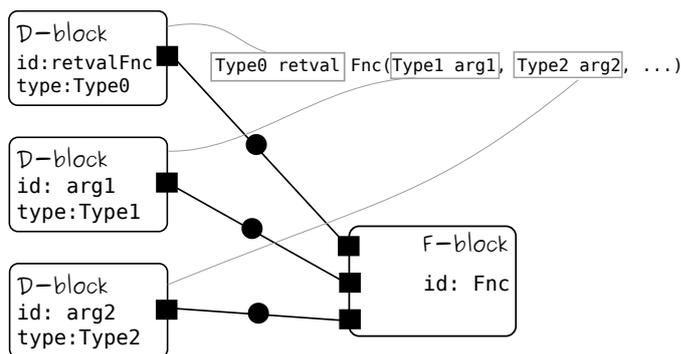


Figure 2.3: A function prototype and its graphical representation as a F-block connected to a set of D-blocks.

2.3.4 Data access constraints

An important declarative part of a schedule model are the **data access constraints** that the order of execution of two or more functions must satisfy to guarantee correctness and **consistency** of the data structures the functions operate on. These constraint relations can be of three types:

- **causality**: a function that changes data values brings in a **cause-and-effect** relation, between the data “before” and “after” the application of the function. Since many functions and data structures represent the physical world, this algorithmic causality might or might not correspond to **physical causality**; this knowledge in itself brings in another relation.
- **function invariant**: the effect of the function on the data values can be modelled *imperatively* or *declaratively* (Sec. 1.6): the imperative form is that in which the function is, in itself, a composition of functions and a schedule; the declarative form is a **relation** that holds between the data values “before” and “after” the application of the function.
- **data consistency**: different functions can change different parts of the same data structure, and their function invariants must, *together*, satisfy some **relations** that *must* hold between all parts.

Together, all the constraints in an algorithm form a **constraint graph**. For example, the data structures that represent the motion state of a robotic kinematic chain only have consistent meaning *after* the full inverse dynamics algorithm has been executed. All the above-mentioned entities and relations are **composable** (thus defining *higher-order relationships*) with some straightforward **composition constraints**:

- a D-block can contain D-blocks;
- an F-block can contain other F-blocks and D-blocks;
- an S-block can contain other S-blocks;
- an A-block can contain other A-blocks.

The **connector** that hosts a data access constraint has an extra **attribute** which indicates if the access to the data represented by the D-block is read-only, write-only or both (i.e., if the argument is input or output of a modeled function).

Since multiple F-blocks can share a data access constraint to a D-block, the latter influences the execution order of the F-blocks: the execution of an F-block that has write access to a D-block prevents other F-blocks from being executed if they are also connected to the same D-block. Therefore, the data access constraint is a *declarative* form to define concurrency properties of the modeled algorithm.

2.3.5 Directed Acyclic Graph: declarative model of control flow dependencies

Algorithms can be very large compositions of functions, and often multiple execution orders exist that all provide the same correct result. The **directed acyclic graph** (DAG) is a *best practice* data structure to represent, in a declarative way, these **partial order dependencies** in the control flow. The DAG model allows to represent the following properties of control flows:

- it is clear from the algorithm when every data structure is accessed.
- hence, data access can always be serialized, to prevent that two functions access the same data at the “**same time**”.
- the dependency data structure can be updated at runtime, without disturbing the work flows of any of the functions.
- the **serialisation** itself, of the dependencies into a linear schedule, has a partial order execution constraint with all functions involved.

Major examples of algorithms that have such DAG-structured dependencies are: computations of the forward and inverse kinematics and dynamics of kinematic chains; the message

passing solver for Bayesian networks; and the real-time control in cascaded control and estimation loops. In all of these examples, there are many ways to order the needed computations, taking into account the strict ordering constraints required by the semantics of the problem that the algorithms are solving. In robotics systems, all of the three “composite” algorithms above have to be composed themselves. For example, the estimation of the robot’s motion relies on being able to connect the data from sensors connected to the kinematic chain (e.g., IMUs or cameras) to the Bayesian “sensor fusion” computations, at the right time. These inter-dependencies make the problem larger in scale; but because of the higher number of constraints, it often also becomes easier to decide which computational ordering to choose.

Data flow model

Algorithms have only **local** data consistency constraints: a function should not be called unless the data in its arguments is available. Hence, in applications where the same functions are to be used on a continuous in-flow of data, **dataflow programming** has emerged as a pattern: the **dependency graph** is a **declarative** model of the order in which data has “to flow” through function blocks, and a *solver* computes at runtime the actual execution order of all functions from (i) this graph structure, and (ii) data availability. In practice, only *trees* or *Directed Acyclic Graphs* provide unambiguous serialisations of the dependency graph to the control flow. Many algorithms come with “feedback” or “iteration” loops, and they need a bit of modelling assistance from the human developer: the serialisation is semantically unambiguous only when *the loop is cut* through a set of data nodes, separating the *previous* from the *current* values.

The design *forces* behind the data flow pattern are:

- to maximize computational throughput.
- to maximize information hiding for the *control flow*.
- to allow runtime adaptability and resource management of the **data buffers**.

2.3.6 Policy: closure in a context

The design choices in the meta model’s mechanism are very “**low level**”, and hence flexible enough to allow various control flow policies: **multiple entry and exit points**, **multiple dispatch**, **partial application**, **callbacks**, **iterators**, **currying**, and **closures**. For example, the *A-block* mechanism extends the concept of a *closure* in a programming language, to a symbolic level:

- it **has-a collection** of one or more *functions*, and not just one.
- it **has-a collection** of one or more *control flows* that put an order on the execution of the functions.
- it **has-a collection** of one or more **dependency graphs** as declarative models for data access and function sequencing **constraints**.

Having all these parts available in symbolic form allows for online reasoning, to adapt the closure.

2.3.7 Policy: application programming interface

One of the most popular ways to make algorithms available for reuse is by means an **application programming interface** (API) of a library. APIs are popular whenever an application wants to have a grip on when *individual* functions act on *individual* data structures; for example, to guarantee data consistency.

The design *forces* are: to optimize information hiding, and minimize runtime adaptability. Only a *selection* of **F-blocks** and **D-blocks** are made accessible through the API; the **S-blocks** remain hidden, and default versions are provided that reduce the number of entry and exit points to one per **F-block** that is made visible.

(TODO: explain why libraries with APIs do not scale over addition of configurable and composable functionalities. For example, solvers for kinematic and dynamics.)

Functional programming

Functional programming has emerged as an algorithm composition policy that makes functions and their **composition first-class citizens**. The dependency graph is a special case of the data flow declarative model: the output data of one function in a composition is the input data of the next function it is composed with.

The design *forces* of the functional programming pattern are:

- to maximize information hiding for the *dataflow*.
- to allow runtime adaptability and resource management on the **callback event loop**.

2.4 Synchronous, asynchronous, sequential, concurrent and parallel execution

A composition of functions (i.e., an algorithm) has a synchronous mode of computation is:

- data is only changed by the functions in the algorithm.
- the order of execution of functions is exactly as written in the algorithm.

When one or more of these constraints are not fulfilled, the computations are called asynchronous. This is, most clearly, the case in (**multithreaded**) software applications, deployed on **multi-core** CPUs, and in **distributed computer** systems. But asynchronicity can also occur in single-threaded computational contexts, such as **non-preemptive, cooperative multitasking** making use of **coroutines** or **callbacks**. The latter is *the* computational model of the **JavaScript** language that realises the behaviour in all **web browsers**.

Algorithms compose *data structures* with *functions* that access those data structures. So, the design choices are:

- *restricted by constraints* on (i) access order between data structures, and (ii) execution order between functions.
- *relaxed by assumptions* that (i) an explicitly specified part of the data structures are **not changed by “the outside world”**, and (ii) an explicitly specified part of the functions have **no side effects**.

Activities compose algorithms, in two complementary ways:

- *serialize* them: multiple algorithms can be executed in the same program, and the program designer must make choices about the order in which the various algorithms are executed.
- *iterate* them: it is a very common use case in cyber-physical and robotic systems to execute an algorithm or program repeatedly over time, at more or less fixed time intervals, in **infinite** or finite loops.

Hence, algorithm and program developers must provide designs that are **composable** with respect to **concurrency**:

- the algorithm designer must minimize the amount of constraints on the order of data access by functions in the algorithm, while still guaranteeing the correctness of the intended behaviour.
- the program designers must take into account all constraints of data access and function execution order that come with the algorithms they have to compose, and choose serialization and iteration policies that respect them.

While algorithm and program designers take decisions about *concurrency*, the designers of the [deployment of activities](#) in threads, processes, cores, SoCs and clouds must take **composable** decisions about:

- **parallelization** of their functions over several computational cores connected by CPU buses and caches. The policy choices are determined by optimising which executions can run at the same time and still share some data.
- **distribution** over several computers connected with an local or wide area network. The policy choices are determined by optimising which executions can be offloaded to different computers, without compromising the performance of the data exchange between programs.

Of course, the algorithm designers can *artificially constrain* the amount of parallelism by providing designs with a high amount of (often implicit!) concurrency constraints. Concurrency is indeed a necessary condition for parallelization or distribution, but not a sufficient one. Concurrency is involved with the *semantics* of the functions and the data they use, and not with the *availability* of the hardware resources for computation storage and/or communication, which is the realm of distribution and parallelization.

2.5 Event loop: synchronous composition of computations with asynchronous data

The **development context** in which computations (**functions** and **algorithms**) are designed is that of **synchronous** execution: the order in which functions are *programmed* in the code of an algorithm, is also the order in which they will *execute* on the CPU. And the data needed as arguments in the functions is always available, at any moment the function needs it, and without the function developers having to worry about whether that data's consistency could be corrupted by the simultaneous access to that data by other functions, somewhere else in the system.

The **execution context** however, is (typically) **asynchronous**: just by looking at the code, one can not know when new data will arrive, and where else it will be used. So, developers need a mechanism to let their synchronous algorithms **react to** asynchronous interactions with other algorithms, also when the timing of the algorithms' execution is unknown. That mechanism is the **architectural pattern**⁵ of the **event loop**:

- **event**: the information that “something has happened, somewhere”.
- **loop**: the serialization of reactions to asynchronous events with the synchronous execution of an algorithm's functions.

Synchronicity (that is, executing functions in the order they are programmed) is a *first* necessary (but in itself not a sufficient) condition for perfect **explainability** (that is, to be able to predict what the result of an algorithm will be). The *second* necessary condition is that the

⁵It is a specialisation of the **reactor pattern** and the **proactor pattern**, in that it adds particular ordering policies to the execution of all functions that it manages.

algorithm stores it “state” in a **thread-safe** way: all the information that it needs to execute is stored in data structures of which it is the sole owner. The *third* necessary condition is that each of the functions in an activity has **no side-effects**: a function changes only the values of the data structures that are in the directly visible **scope** of its programme code, and hence no data is changed “behind the back” of the function.⁶ All of the above guarantees can not be provided by the event loop mechanism alone; the missing piece are the deterministic **interaction streams**: they let two synchronous event loops exchange data asynchronously via efficient and deterministic buffer mechanisms.

Some instantiations of the *event loop pattern* are:

- the “*Web*” with **browsers**, **servers**, and **Single Page Applications** (SPA).
- the **Programmable Logic Controller** (PLC) workhorse of the automation industry.
- **software components** in **service-oriented architectures**.
- **computer games**.

2.5.1 Mechanism: 4Cs-based programme template

The following pseudo code gives a **mereo-topological**⁷ version of the event loop, with a best practice structure of the “4Cs”, that is, *C*ommunications, *C*oordinations, *C*onfigurations and *C*omputations:

```
LCSM_state = Creating;
compute();      // execute one's own minimal initialisation
when triggered // by operating system, which deals with all
               // asynchronous side effects.
do {           // the serial control flow structure of the event loop.
  communicate() // get all "messages" with events, data & queries,
               // provided by other asynchronous activities.
  coordinate()  // handle the events in these messages, and
               // decide which ones to react to.
  configure()   // some events imply reconfiguration of event loop.

  compute()     // execute your (serialized set of) synchronous algorithms,
               // which in themselves are side effect-free computations.

  coordinate()  // the computations above can generate events that imply
               // reconfiguration, of this event loop or other activities.
  communicate() // the computations above can generate events, data & queries
               // that other asynchronous activities must know about.

  yield()      // the loop waits for its next trigger
}

```

The pseudo-code above is just a *template*:

- the suggested **schedule** (that is, the *relative order* of the “4C” functions) makes sense in general, but need not be a hard constraint for every individual application.

⁶A common source of side effects are **system calls**, that is, **interactions** with the operating system. For example: opening, closing, reading and writing of files; inter-process communication; or reading the **clock**.

⁷This model represents a serial *structure*, of 5C-conforming *types* of *behaviour*.

- its intention is to be used as a *memory aid*: starting from the template, the application developer must make a *motivated* decision *not* to use the complete version, and throw out some steps.
- the exact execution sequence of the functions, as well as their exact contents, must always be configured by the application developer.
- for *fast* running event loops (that is, with a short or non-existing `yield()`) the “second round” of `coordinate()` and `communicate()` do not add much value, because the same functions will be called in the “first round” of the next iteration of the event loop.

The schedule can be generated as a **procedural** one, at compile-time of the application, but could as well be computed itself at runtime, from a set of **declarative** dependency constraints which are “solved” to create the schedule (in the extreme case, every time the event loop is triggered).

The power of the event loop pattern, as described in the simple template above, is that it *stimulates* developers to separate the 4C concerns. That way has also proven⁸ to *facilitate* (but not to guarantee) composability and compositionality in complex distributed systems. This composition of multiple event loops into one single larger event loop expects that all algorithms inside the event loops can be created by means of **coroutines**, and that **preemptive multi-tasking** can be avoided. One necessary, but not sufficient, condition to satisfy this constraint is that all functions in the algorithms have no side effects, have short implementations, and that they access only data that is owned by the event loop activity. Often, that data is stored in the **buffers** needed for the **asynchronous I/O** in the `communicate()` parts of the event loop.

2.5.2 Composition of event loops

(TODO: partial order constraints between the 4Cs, and how this order can be optimized under composition.)

2.6 Activity: asynchronous composition of algorithms and their interactions

This Section’s topic, the *activity*, is the essential one in this Chapter when viewed in the document’s broader context of **system architecture** design: its design focus is the *application* (“system”), and not the *technology* (“components”). It is indeed the application that mandates which activities must be realised, how “big” they should be, and how “much interaction” between them is required. This Section can not give an absolute answer to these essential system-level design questions, but it does introduce the *methodology* to compose together individual *components* with which, eventually, the concrete application-driven architectural composition trade-off decisions can be made, irrespective of the size of activities and their interactions.

For the sake of simplicity, and at this early position in the overall document, the terms **activity** and **component** can be used interchangeably. Later Chapters will bring differentiation between both terms, by introducing **different types of components**, each realising the same activity in, first, an **information** architecture and, subsequently, in a **software** architecture; both of these architectural design conform to *best practices* in application-independent,

⁸For example, many servers and browsers work with an event loop architecture.

generic [system](#) architecture.

At the most abstract view on a system, the system needs **activities** to take responsibility for the realisation of the required **behaviour** of the system. Every cyber-physical system (be it of *physical* or *information processing* nature) has **behaviour**. An example of *physical* behaviour is the motion of the air that flows around a quadrotor drone and through its propellers. An example of *information-processing* (“cyber”) behaviour is the *control* of the lift force of that quadrotor by steering its rotor velocities. Cyber-physical systems are *engineered* to realise behaviour that has a particular **purpose** during a particular time, namely, to execute the tasks of the applications the systems were developed for. Of course, many systems show the same behaviour only for some time before they change to another behaviour, which is at that time more useful to fulfill the application’s requirements.

This document uses the term **activity** to denote the “cyber” form of behaviour, that is, any system component that can be made **responsible** for:

- realising the application’s **tasks** with “good enough” quality,
- while making efficient use of the **resources** that the system has available.

[Mereo-topologically](#), an activity is the **composition**⁹ of its:

- **interior**: the **functions** that run **synchronously** in **algorithms** inside the activity.
- **exterior**: the **asynchronous interactions** via which the activity exchanges data with other activities.
- **ports** (or “**interfaces**”): the connectors between the interior and exterior parts.

Hence, system developers adapt the *granularity* of their activity designs to the granularity of the tasks and the resources in their system. In other worlds, there are no *absolute* criteria to determine the optimal size of an activity, because the composition is *relative* to a system’s tasks and resources. This document help developers to design the just-mentioned *components* of activities in such a way that they can be composed with any level of granularity that might be needed in particular *systems*.

2.6.1 Asynchronicity and interfaces

The asynchronicity of the exteriorly visible part of an activity’s behaviour has multiple origins, with increasing levels of synchronization complexity:

- algorithms must **interface** each other’s, **producing** data for, and **consuming** data from, each other.
- algorithms must be **configured** to the particularities of the **computational platforms** of hardware and operating system on which they are executed, and this configuration often needs to be done by a “third-party” activity which has the knowledge about what the best configuration is.
- in addition, they must be **configured** to the **physical resources**, that do typically not respect any synchronicity constraints that the algorithms would like to see satisfied.

The term “**interface**” has the following meaning in this document: it is the formal model of the semantics and the purpose of a information exchange between activities. Indeed, while *physical activities* interact by exchanging energy, *information activities* interact by exchanging **information** (“data”, “messages”, “events”,...), about the status of the physical activities (motion, force,...) and the symbolic activities (task, performance,...) whose interactions

⁹That is, it is yet another example of a *higher-order function*.

they control. So, all **interface** choices (or interface **policies**) require two complementary **mechanisms** to be available:

- the **exchange** of information between activities.
- the **production and consumption** of information inside the interacting activities.

So, the typical interface situation involves three parties:

- the exchange mechanism.
- the producing activity in the exchange of information must be able to hand over its information to the exchange mechanism.
- the consuming activity must be able to get information from the exchange mechanism.

The generic interface model allows both interacting activities to be producer and consumer at the same time, for different types of information that are being exchanged in their interaction.

2.6.2 Mereo-topological model

An “activity” is the *pars pro toto* representation for the following *composition hierarchy* in **types** of information-processing entities:

- **function**: the data, function and control flow entities and relations of Sec. 2.3, and which form the foundation of all **computations** in software components.
- **algorithm**: the **synchronous composition** of several **functions** which share data. That is, functions are scheduled in a **serialized** way, and all data can be used exclusively by any function in such a series.
- **program**: the **asynchronous composition** of several **algorithms**, due to **concurrent execution**, that is, via **cooperative multitasking** on one single CPU core. Care is to be taken that one function does not write data *at the same time* that another one is reading or writing that same data, *but* that **coordination** is fully under the control of the **program**, because the whole **program** has the same **computing context**. Badly coordinated asynchronous execution of **functions** can indeed introduce inconsistencies in the data that is shared. But the challenge is not to control the *time* at which operations take place, but their relative *order*. The mechanisms available for the coordination are (i) **interaction primitives** between several of the **program’s algorithms** (e.g., **flags** and **streams**), and (ii) clear **ownership** of each **resource** (that is, the model that represents what an **algorithm** can do on a resource, and at what time).
- **activity**: the **asynchronous composition** of **programs** due to **parallel execution** on different CPU cores, even on different networked computers. The extra challenge compared to **programs** is that **state must be duplicated** between **programs**, so every state-based decision making must be made robust against the parallel execution.

2.6.3 Port in algorithm (service) versus Port in library (API)

The exchange mechanism behind an interface can be a passive data structure (for example, a **stream**), or an activity in itself (for example, a **broker** activity). The producing and consuming activities can be made unaware of the concrete passive/active situation, via a **Block-Port-Connector** model of an interface: Producer and consumer are *blocks*, the exchange mechanism is a *connector*, and the decoupling is realised via *ports* (that can abstract away

the active/passive implementation). *Ports* in activities are often called *services*, and *Ports* in libraries are often referred to as the *API* of the library.

2.6.4 Continuous, discrete, logic, symbolic information: data, event, flag, query

Activities **process** their “**data**” by means of “**functions**”, in many different ways. They must also be able to influence what other activities do in their processing. Hence, both “data” and “functions” must be represented in ways that allow (i) **to compose** them internally inside one activity, and (ii) **to exchange** them between activities. This document identifies three semantic types of “data”:

- **data structures** for the **continuous** world: the real world is continuous in time, space, energy, etc., and this document gives the name “continuous data” to every formal representation of that continuous world.
- **flags** and **events** for the **discrete** and **logic** world: activities must be able to influence each other’s **control flow**. This typically happens by *sending events* (or “signals”) between the activities, or by letting them *observe* the value of **flags** that they both have direct access to. In their physical and social representations, *flags* are pieces of fabric with meaning encoded in colours and structures in the fabric, and *events* is the name given to all sorts of **happenings** that people organize or that take place autonomously, in nature and in society. In their cyber representation, flags and events have very similar meanings and purposes, and they are represented, of course, also as data structures. Typically, only very small ones, because the only semantics they have to represent is the discrete (changes in the) state of the world. This document uses the following terminology:
 - *flag*: to represent that “*some condition has a particular logical value*”, or, “*a has a certain status*”.
For example, a traffic light is green; a door is open; or, an activity has a particular behavioural mode.
The **affordances** of a flag are: to be *created, set, observed, cleared, and/or deleted*
 - *event*: to represent that “*something has happened*”.
For example, a traffic light has changed colour; or, a `door_open` button has been pressed.
Its affordances are: to be *wired, fired, handled, consumed, and/or processed*.

The examples show that a common instantiation of the event concept is the one that represents that a flag has changed its status.

- **propositions** to model the **logical** world: formal statements about the world that are true or false.
For example, “*this robot has two serial-chain arms*”.
- **queries** and **dialogues** with models for the **symbolic** world: activities must be able **to represent, to discover, to configure and to update** the **functions** that they use themselves or that others use, and do that at **runtime**. In other words, to represent

current or possible behaviour (of oneself, or of the other one), and to provide new “computations” and new “data types” to each other. Such symbolic interactions take place via a **query protocol**, that composes a particularly sequence of *models* that represent interpretations of the world.

Of course, from the point of view of the technical act of creation, storage or communication, *events*, *flags*, *queries* and *models* are just special cases of *data*. So, this document uses the *pars pro toto* terms

- “data”, to refer to all three semantic variants.
- “information”, to refer to data that comes with metadata that explains how to interpret the data.

Component

This last type of “activity” is the default interpretation of the term when used elsewhere in this document. It is also this document’s “*component*” building block, conforming to the mainstream interpretation of **component-based software engineering**. In other words, it is the level of composition where *all* the component mechanisms come together:

- **to compose** functionalities into behaviour.
- **to deploy** that composition into an **event loop**, so that the behaviour can be executed.
- **to interact** with the resources, and with other activities via asynchronous interfaces.

An activity must provide a “**data sheet**” to other activities, with the model of the exteriorly visible behaviours and interactions.

2.6.5 Best practice: resource ownership

Ownership of **resources** is an important concept in a system: the “owner” of a resource should be the only one to make changes to that resource, including making decisions to make the resource accessible to others or to (re)configure its attributes. Ownership must be integrated somewhere in a system design, and the activity is the right place to host the ownership **metadata** relations. Here is a list of “best practice” constraints to be satisfied in such ownership relations:

- every resource has one and only one owner at every moment in time. Ownership can be transferred, temporarily or permanently, from the owning activity to another activity.
- the values of the properties of the resource’s model can only be changed by functions in the owning activity.
- other activities that need access to the resource must do so via **queries** to the owning activity. Access can be granted in a way that requires no explicit query for each individual access, by means of query protocols for *transfer of ownership* (for example, “**producer–consumer**”, “**broker**”), or *sharing of access* (“**borrowing**”).
- when activities receive *copies* of the state of a resource, “*the*” state of that resource exists only in its owning activity. This explicit “ownership hierarchy” allows a system to work with state data that can be inconsistent between owning and borrowing activities.
- an activity has extra “**non-functional**” algorithms, for the sole purpose of configuring and monitoring the resources that it owns.
- when the owner of a resource is stopped, the resource is not accessible anymore. *Or* the system architecture has a protocol with which the owner transfers its ownership explicitly before stopping.

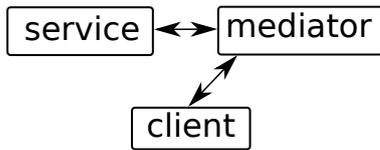


Figure 2.4: Mereo-topological model of the (peer) *mediator* pattern, to help system designers introduce **loose coupling** between “*service*” and “*client*” activities. The arrows represent *data access constraints* on the interacting activities.

2.6.6 Pattern: mediator for peer-to-peer interactions

Architectural compositions as in Figs 2.4 and 2.5 show up in many use cases in which the *client-server* behavioural composition needs to be realised between a “*service*” activity and a “*client*” activity (or multiple of them).¹⁰ System developers want to avoid *direct* behavioural coupling between both activities, that is, to let them know about each other’s:

- existence,
- name,
- software component port address,
- data access policies,
- programming language implementation,
- software version,
- etc.

But, of course, both activities *do have couplings*, in the form of:

- the *model* of the *information* they exchange,
- the relations between **data** and **metadata**,
- and the events through which they *coordinate* and *configure* their behaviours.

The important insight is that these latter couplings depend only on the *type* of their activities, and not on the concrete *instances*; the former couplings do all depend on instance properties. More and more application contexts do not have clearly distinct server and client roles anymore, so one often uses the more neutral term **peer** for both “*service*” and “*client*”. Anyway, the terminology in itself is not relevant, but the specific dependencies between the specific behaviours in all interacting peers are. The relevance of the separation between, on the one hand, *coordination and configuration* (realised by the mediator activity), and, on the other hand, the *computation and communication* (realised by the peer activities themselves), increases with:

- the *number* of interacting peers (Fig. 2.5),
- the *complexity* of the interaction protocols,
- and the *statefulness* of the peers.

The **design driver** force in the pattern is **to concentrate all knowledge and decision making** about the **behavioural coupling** of several activities in the **mediator** activity. This mediator knows:

1. the **information representation relations** of all peers involved, irrespective of whether the peer is a service provider or a consuming client. For example, a mediator:

¹⁰The exact meaning of the terminology “*client*” and “*server*” is not widely standardised, to say the least. For example, in the domain of **cloud computing**, the terms “*cloud*”, “*fog*”/“*edge*” and “*client*” are more mainstream; in databases one speaks of “*backends*” and “*frontends*” and about **microservices**. This document could also have used the term *peer mediator*, because often the service and the client are interacting bi-directionally as each other’s server and client.

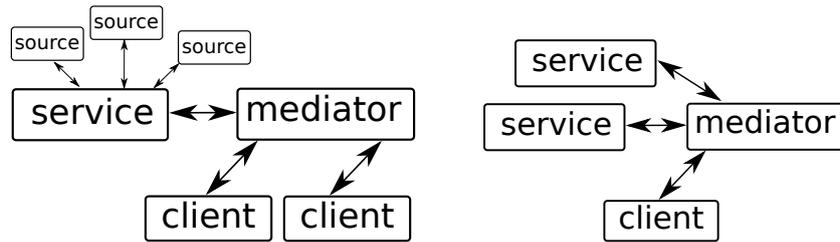


Figure 2.5: Variants of the *mediator* pattern, with multiple sources for the service, and multiple clients or services for the mediator.

- decides which interaction channels to connect to which activities.
 - initializes them properly.
 - coordinates the (synchronous or asynchronous) access of the interconnected activities to the shared channel resources.
 - checks whether all above-mentioned inter-dependency relations remain compatible and consistent.
 - decides to continue or not with its “**brokerage**” activity.
 - decides to include **glue code** where necessary.
2. which Finite State Machine **events** the various peers emit and/or react to, in order to change their behavioural states. It can then decide whether all peers have events with the same semantic meaning, and can include *event forwarding* glue code where necessary.
 3. which **flags** the various peers commit to a Petri Net coordination of some of their algorithms. The mediator is the activity that **executes the Petri Net**.
 4. which **model checking and adaptation** to realise in the *resource configuration* step of the Life Cycle State Machines of all peers involved. This can, in itself, require yet another Petri Net coordination between mediator and peer activities, to let their algorithms go through their required correct configuration steps at the same time.

This coordination is to be brought in without the latter activities being aware that their flags take part in a Petri Net. The latter often requires to copy (atomically!) the value of a flag in an algorithm into the value of its “logical twin” in the coordinating Petri Net model.
 5. about the **tasks** that the various peers have to support, together. The mediator can add extra components to realise *Service Level Agreements*, or, at least, to monitor the **quality of service** of the coupled behaviour and to fire the appropriate events to all peers when the task service falls below the configured threshold.

In summary, the mediator is the clear and unique **owner** of all decisions about how to coordinate, protect *and* optimize access to the activities it is mediating. That decision making role makes this pattern more specialised than the simpler *message broker* pattern that is used for *message-oriented middleware*, *object request brokerage*, or *enterprise service buses*. Because a mediator has all this knowledge about the workings of all the activities it mediates, it makes sense, in several use cases, to let the mediator activity be the unique representative of all other activities towards the rest of the system.

The mediator pattern for task-resource trade-offs

In this document, the interaction between tasks and resources is an important use case. This Section explains how to specialise the mediator pattern to this context (Fig. 2.6).

Tasks rely on resources (physical as well as cyber) being sufficiently available, to realise a set of capabilities. In real-world contexts, the expected *quality of service* can seldom be provided perfectly. Hence, **trade-offs** must be made between (i) the cost of resources, (ii) their availability, and (iii) the quality they provide.

(TODO: define QoS metrics of resource and of capability, **monitor** QoS, dependency model between component behaviour and QoS, trade-off solver. resource throttling; sharing same communication channel with various Tasks; sharing overlapping workspaces between two robots; how to choose specific trade-offs, and specific QoS; etc.)

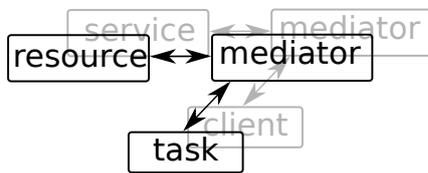


Figure 2.6: The special case of the mediator pattern applied to the interaction between *task* and *resource* activities.

2.6.7 Pattern: task queue and task worker

Imperatively specified behaviour is most often translated into a **software architecture** built upon **application programming interfaces** (“*APIs*”): the imperative approach towards the solution of a problem invites developers to fix the *control flow* of the solution already at design time. In other words, every different case of a “request” for a solution has been identified, and is met by one particular “command” encoding to the solver of the problem, and that command is then solved by a dedicated, monolithically design algorithm. In addition, an API-centric system design invites “clients” to tell their “service provider” *how* to solve their problem, instead of providing information about *what* their problem is.

Declaratively specified problems need *runtime solvers* to compute the control flow **reactively**, that is, based on the state of the data structure (e.g., a tree or a **directed acyclic graph** that encodes the declarative dependencies. The architectural pattern that supports such a reactive behaviour execution context very well, is inspired by how organisations are organised to solve many complex problems, concurrently and with multiple workers dividing the task execution. The pattern composes a **task queue** with a **worker**. That worker may be selected from a *pool* of workers that the application has available. When a worker is ready to do some work, it picks a task from the queue, and executes it. The tasks that are put on the queue are the fragments of the total solution that correspond “to traverse” the declarative data structure, one step at a time.

In order to guarantee **consistency** of all *data* involved in reactive task execution, the behavioural pattern must be complemented with an *interaction* pattern that supports the bookkeeping of:

- which tasks are waiting on the queue, to be processed.
- which tasks have already been processed, but whose results have not yet been consumed.

The **submission-completion** interaction pattern is a primary example that provides such structured bookkeeping. The obvious ways in which the task queue pattern allows performance

trade-offs to be made are:

- the granularity of the tasks.
- the level of task granularity where the declarative encoding of task solutions is stopped, and imperative encodings take over.
- the number of workers made available for task processing.
- the number of **event loops** to use to distribute the tasks over sets of tightly cooperating workers.
- the introduction of “middle management”, (i) to decide about task distribution over a hierarchical worker organisation, (ii) to monitor the quality of the task execution, and/or (iii) to monitor the overall progress of the problem context.
- the choice for a hierarchical *master-slave* coordination of workers, or a rather *peer-to-peer* coordination.

In computer-driven systems, the “middle management” role is realised by **mediator activities**.

Examples of declaratively designed behaviour abound in any engineered system of somewhat realistic complexity:

- **Finite State Machines**: the behaviour the system should have at every moment in time is represented declaratively, by enumerating the conditions under which the system should switch from its current behaviour to a new behaviour.
- **Petri Nets**: the execution order of parts in multiple algorithms is represented declaratively, by enumerating the conditions under which each algorithm is allowed to progress.
- **Bayesian networks**: the order in which new data is to be taken into account when updating the information available in the whole network is represented declaratively, by a **directed acyclic graph** of **conditional probability** relations.
- **cascaded control loops**: the order in which to execute the different parts of a control system is represented declaratively, by the **data flow architecture** of control loops.

2.7 Flag: peer-to-peer algorithm coordination protocols

This Section introduces the **flag** and flag-based (**coordination**) **protocols**, as model primitives via which to coordinate the control flows of two or more algorithms. The major reason to introduce flags as **first-class** primitives in the design of complex systems is *to increase the efficiency of making decisions*, in a system that has a large number of algorithms working **together** and **asynchronously**, and that has a various and always changing set of sub-goals. This situation is common in, for example, robotics applications of real complexity and life span: they require a lot of perception and decision making, in the “background” (trying to keep the platform operational in the “best” possible way), as well as in the “foreground” (trying to realise specific user-centric tasks).

2.7.1 Abstract data type: flags

The purpose of a coordination protocol is to help algorithms in their **logical decision making**. A *flag* has the semantics of a Boolean variable (e.g., **condA**), that is part of decision making via **logic programming** functions. A flag represents (“caches”, “remembers”) the value of a **logical condition on the computational state** of a particular function, or

sub-algorithm. A flag can have the status **true** or **false**, but in the advanced semantics of systems engineering, a *flag* should also have the status **uninitialized**, to indicate that decisions to be taken somewhere in the system should *not yet* depend on the value of that flag. The protocol is the **structural composition** of flags, to realise coordination by requiring a specific **order** in which flags become **true** or **false**.

Because control flow relations can change at runtime, and because their order influences the outcome of an algorithm, the execution of control flows must be **coordinated**, inside one single algorithm or between several algorithms.¹¹ The control flow decisions that each algorithm must make can depend on the values of a set of flags¹² whose values are changed in other algorithms. Hence, the design challenge in coordination protocols in such a multi-algorithms context is that:

- one or more of the conditions (i.e., *flags*) underlying the decision are *not* set by the decision making algorithm itself, but by another algorithm.
- and that other algorithm may or may not be executed *asynchronously*.

More in particular, many algorithms must make control flow decision of the following kind:

$$\text{if (condA and condB) then \{...\};} \tag{2.1}$$

and the Boolean truth value of condA is determined completely in the executing algorithm, but the truth value of condB is determined in another algorithm. This situation gives rise to a **race condition** as soon as that other algorithm changes the truth value of **condB** *after* this algorithm has accessed it for its own decision making. Hence, system designers need a mechanism to let both algorithms **commit** to updating truth values only according to a **protocol** that they both **explicitly** agreed upon, and that can be proven to be race-free.

Sequence diagrams are a mainstream graphical representation to model a coordination protocol. However, sequence diagram meta models most often come with other structures and behaviours rigidly attached; while it is one of the main ambitions of this document to separate those different concerns as far as possible.

The advantages of introducing flags as **first-class citizens** are:

- *efficiency*: a flag “**cached**” the outcome of the computation of the condition of a computational state, so condition computations need not be repeated.
- *concurrency*: concurrent algorithms in an activity can read/write flags to keep each other informed about their computational status.
- *separation of concerns*: the algorithm *designer* must only be concerned with logical correctness, without having to care about:
 - the computational properties of the *execution platform*’s hardware, middleware or operating system.
 - *the reason why* a flag has been given a particular truth value.
 - how *the system will react* to the logical decision that is being made.

Often, a separate **mediator** activity is introduced, as the *owner* of a specific inter-algorithm coordination. The mediator can execute, both, the *monitoring* computation to set flags, and the *coordination* computation to make a control flow decision. In this way, none of the

¹¹The case of coordination between several algorithms is conceptually not different from coordination inside one algorithm: this document uses the term “algorithm” to mean “any *composition* of functions, data and control flow that matters to be considered together.” And *coordination* is one of the aspects that determine whether a certain composition scope matters or not.

¹²On most modern computer hardware, checking those values can be done efficiently, if they are **digitally encoded** as **bit fields** or **bit arrays**. The encoding allows **atomic evaluation**, of multiple flags at the same time.

coordinated algorithms itself must know about how its execution is coordinated with other algorithms, and hence it must not be adapted when it is integrated into a larger or another context. The result is a better **composable** activity.

2.7.2 Mechanism: flag arrays (“bitfields”) with atomic iterator

Figure 2.7 sketches the mechanism: two functions coordinate their own control flows with each other as follows:

- each such coordination uses one abstract data type of an array of flags.
In most software *implementations*, one character or integer value is used, and each of the bits represents a flag. The figure hints at such an implementation: only the first four bits in the array are used for the protocol; the coloured outlines indicate which of the two coordinated algorithms are allowed to access which flags: the correct order is *green, purple, green and purple*.
- before the coordination, at time t , the flag array is still **uninitialized**.
- both functions know the *order* in which the coordination protocol expects each function to raise the next flag in the array. *By discipline* of the function behaviour, no function changes a flag in the array when it is not its turn. This discipline must guarantee the **atomic iteration** over the subsequent steps in the protocol.
- raising a flag is a **commitment** of each function to satisfy the protocol order. That is, (i) a function will not lower a flag it had raised before, and (ii) it guarantees not to change the Boolean condition that is part of the protocol.

Only the *mechanism* of coordination is modelled by the flag array. The reason *why* the coordination is needed is not represented, nor is the logic behind an algorithm’s decision to set a flag in the array. The following Sections introduce the flag arrays for some common application use cases.

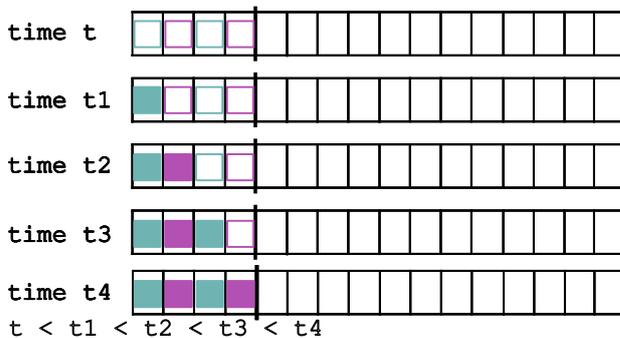


Figure 2.7: A flag array (or “bitfield”): the array provides the structural constraint of the order in which two asynchronous functions (the *green* function and the *purple* function) can set flags. The exact time of raising new flags in the array does not matter, as long as the mentioned strict order is satisfied.

In some protocols, the latter commitment can be relaxed: the participating functions guarantee not to change the Boolean condition connected to the *active* part of the protocol. Where “active” means: *set* by the function that raised the *last* changed flag, and *read* by the function that raises the *next* flag to be changed.

2.7.3 Policy: big endian versus little endian flag arrays

Figure 2.8 shows an alternative way of representing the flag array of Fig. 2.7. *Implicitly*, this latter Figure assumed the first flag to be set in the protocol to be depicted in the first place of the array when starting from the left, and subsequent flags to take the next places in the

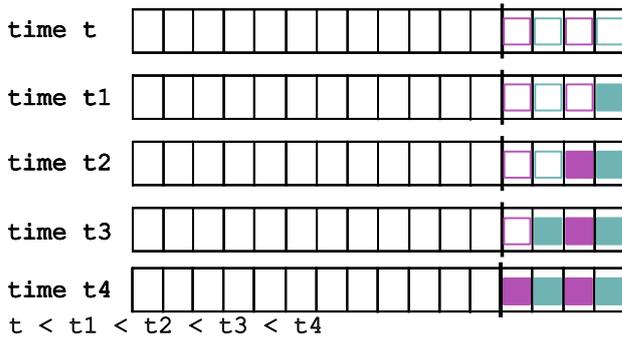


Figure 2.8: The *big endian*, or *left-to-right*, version of the flag array in Fig. 2.7 .

array. So, the difference between both figures make clear that the *meaning* of the *order* in a *protocol*, can be *represented* in various equivalent ways. This document uses the following terminology for this alternative representation:

- **big endian**: the inspiration comes from how computers store numerical values in their memory, and in *big endian* the “most significant” bit is on the lowest “address”.
- **right-to-left**: the inspiration comes from written human languages, and in *left-to-right* languages, a words and sentences start with their first “letter” on the left.

The document will mostly use the *little endian* representation of Fig. 2.7 for all conceptual, symbolic illustrations. *Software implementations* often make different choices. For example, most computer hardware is big endian, where a smaller number of filled bits on the “right hand” side of an byte array represents lower numbers, so a flag array fills up from the right, or from the lower-valued bytes.

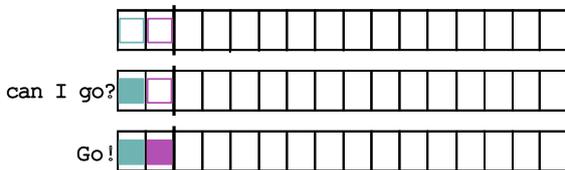


Figure 2.9: *Stop-and-go*, representing the simplest possible coordination.

2.7.4 Stop-and-go coordination — Barrier

Figure 2.9 depicts the simplest possible Petri Net, used to coordinate the behaviour of two activities:

- the coordinated activity puts a **token** in the **place**, to indicate that it has *stopped* its progress and is now waiting for a “Go!” of the coordinator.
- at a certain moment, the coordinator makes the decision to give this “Go!”, in the form of consuming the **token**.

This is a very simple for of the **barrier** synchronization method. (A **memory barrier** is a particular special case.) When the flag is extended from just a Boolean to a data structure that also carries information, the coordination becomes the **rendez-vous** synchronization method.

2.7.5 Two-phase commit coordination

Figure 2.10 shows another common coordination primitive, **two-phase commit** with only one coordinated algorithm and one coordinator. There can be many reasons for this seemingly

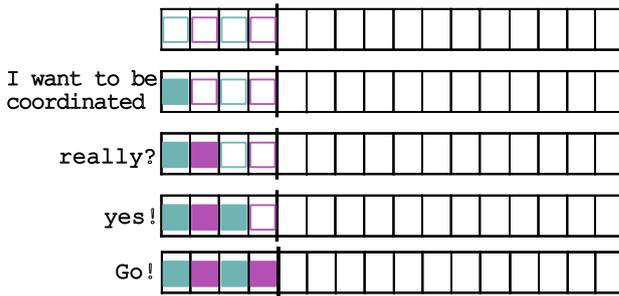


Figure 2.10: *Two-phase-commit* coordination of two algorithms.

superfluous coordination primitive. For example: the coordinator holds the coordinated algorithm until it has observed “something” to have happened elsewhere in the system, but without the algorithm that caused that “something” is itself involved in the coordination. This is an example of the *dependency injection* best practice: the coordinated algorithm does not have to know what it is waiting for, nor how to find out whether the conditions for progress are satisfied; that knowledge can remain in the coordinator, hence improving the composability of the coordinated algorithm.

2.7.6 Data borrowing coordination

This Section describes another common use case of inter-algorithm coordination, namely that where one algorithm that *owns* a particular data structure lends the ownership to another algorithm, but expects to get ownership back.

The flag array in Fig. 2.11 has the same *structure* and *behaviour* as the one in Fig. 2.10. Only the *interpretation* in the context of an application is different.

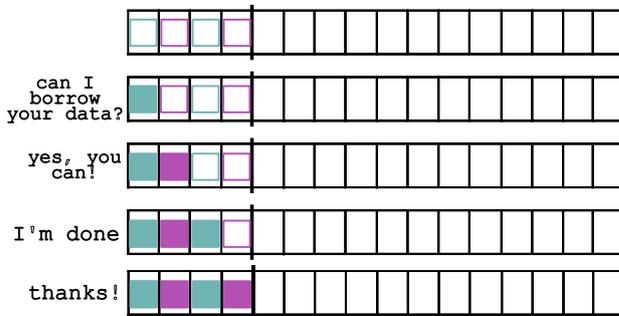


Figure 2.11: *Data borrowing* coordination of two algorithms.

2.7.7 Higher-order flags: active, timeout, highwater-lowwater

Some coordination use cases profit from *higher-order* flags, to serve the following purposes:

- **active**: a flag that indicates whether one or more other flag arrays are active, because, if not, the coordinated algorithms should not even spend time on checking the flag status.
- **timeout**: a flag that indicates that one or more other flags should not be looked at any longer, because the validity of the coordination has been preempted by a timeout.
- **highwater** and/or **lowwater**: a flag that allows one coordinated algorithm to stimulate the other coordinated activity to spend time on a shared protocol, because a lack of timely response would decrease the value of having the protocol in place.

Each of the higher-order flags can cover more than one “lower-level” protocol; or any collection of flag protocols and the two other main coordination mechanisms, [Petri Nets](#) and [Finite State Machines](#). So, a major reason to introduce higher-order flags in the first place is: *to increase the efficiency* of coordination for a system with a large number of to be coordinated algorithms. The latter situation is common in, for example, robotics applications that require a lot of perception and decision making.

2.8 Petri Net: algorithm coordination via dedicated mediator algorithm

This Section introduces the Petri Net meta model, as the mechanism to support the **coordination** between the **control flows** in multiple **algorithms**, while **decoupling** the direct coordination interaction between any two of these algorithms. In other words:

- the algorithms need not know about each others’ existence.
- each algorithm engages in a flag-based coordination with only a **mediator** algorithm.
- the mediator performs the coordination **decision making** in an internal, synchronous schedule.

These algorithms run in various [activities](#) and they require access to various **resources**, while they cooperate on realising various [tasks](#). For example, each of these tasks requires one or more perception and control algorithms, each in itself relying on multiple monitors, and maybe also on communication algorithms. For the sake of brevity, the document uses the terminology “*to coordinate multiple activities*”, because this document uses the term [activity](#) for compositions of algorithms of any kind.

The core **structure** of the Petri Net mechanism is a graph data structure (the “[Petri Net](#)”), that supports the bookkeeping of the coordination behaviour. This structure can be composed with a set of **behavioural policies**, and the resulting composition is a *coordination pattern* that designers can configure to optimize system-level *trade-offs*.

2.8.1 Examples of multi-algorithm coordination

Here are some examples of application contexts in which multi-activity coordination is needed:

- **manufacturing** applications, such as **car assembly** or **food packaging**, require the coordination of the actions of multiple machines, that are all active at the same time on the same **conveyor belt**, and each with its own control system, and (hence) with its own set of behaviours and activities. Most current deployments have a hard coded speed of the conveyor belt, and all stations along the line are designed together to follow that speed. However, the **more flexibility** one wants to introduce in conveyor belt centered applications, the more the speed of the belt should be coordinated with the progress that each of the cells can make. And preferably, that coordination can be done without any reprogramming, “just” by letting the cell controllers interact with each other.
- **humanoid** or **dual arm** robots must coordinate their different body parts with each other in order to avoid self-collisions, but also to improve the execution efficiency of the **multiple tasks** that these devices can work on at the same time.
- any shared resource can decide to require coordination between the activities that use it, for several reasons:

- to allow one activity exclusive access if that wants to execute a series of operations in an “atomic” way, that is, without being interrupted by other activities accessing the same resource.
- the resource wants to optimize its usage via access protocols. Many such **queueing protocols** have proven their value in many use cases in human society.

The warehouse in the above-mentioned manufacturing context is a primary examples of a resource that is shared by many users.

- in **multi-rate** sensor signal processing, one algorithm can be dealing with a 1 kHz signal from an **IMU**, a second one with a 10 Hz signal from a **Lidar**, and a third one with a 1 Hz signal from a camera. Yet, the robot controller wants to update its control setpoints at yet another frequency, after running a **sensor fusion** algorithm on the outcome of all three other sensor processing algorithms. This integration clearly requires coordination between all four algorithms involved. And the number of such inter-algorithm coordinations is also growing with the number of sensors, and the number of task executions that are each users of often differently configured sensor processing.

2.8.2 Role of algorithm coordination in systems-of-systems

The *purpose* of a dedicated meta model for the coordination of algorithms is to offer to systems-of-systems developers a well-documented and formalized **pattern**, that can let activities **outsource the decision making** about their mutual coordination to a third party, the **coordinating** activity, or **coordinator**, or (coordination) **mediator**. The pattern must solve the following complementary design challenges:

- *decoupling* of the four complementary decisions that have to be taken at system level:
 - the identification of *which activities* need to be coordinated.
 - the identification of the *reason why* each activity needs coordination.
 - the conditions on the activities’ state to be satisfied before, during and after the coordination.
 - the individual steps to make for each activity, in order to progress in the coordination.
- *synchronous decision making*. Typically, the coordinated activities work asynchronously with respect to each other. So, in order to guarantee consistent decision making in their mutual coordination, each activity uses one of the asynchronous **interaction mechanisms** to provide the coordinator with the information that represents its own coordination status. The coordinator can then make a consistent, **race-free** coordination decision in its own **synchronous** algorithm. It then informs all involved activities about the outcome, via the same interaction mechanisms.

2.8.3 Mechanism: place, token, transition

A **Petri Net** is an **abstract data type** with the following **entities** and **structural relations** Fig. 2.12:

- **place**: the **entity** of the Petri Net model. Graphically, it is depicted by an unfilled circle, like p1 and p2.
- **token**: the **relation** that identifies the **state** of a **place**, that is, a **place** is filled by a **token**, or it is empty. A **token** is depicted by a black dot, placed in a **place**.

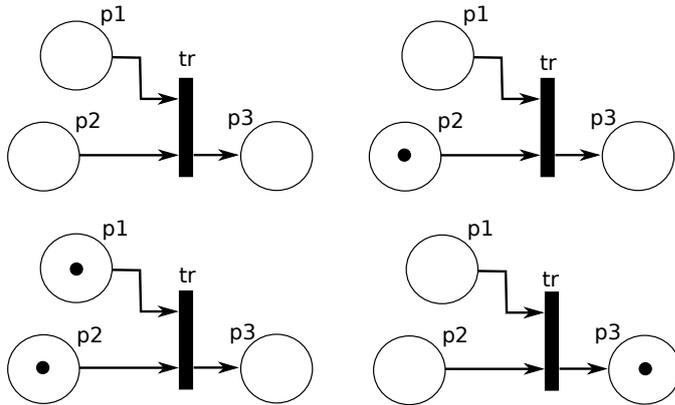


Figure 2.12: An example of the **structural** model of a Petri Net, with places p1, p2, and p3, and a transition tr. The **sequence** of the four structural models gives one possible **behavioural** model of the Petri Net.

- **transition**: the **relation** that connects its input places to its output places. It is depicted by a black rectangle, such as tr.
- **arrow**: the **relation** that connects a place to either an input or an output port of a transition. The obvious graphical depiction of an arrow is a line with a pointed arrow. The direction of that arrow represents the input or output role of the place in a transition relation.
- **marking**: the **relation** that represents one set of tokens in places. It is the **state** of the data type, after a certain number of operators have been executed.

The **behavioural relations**, or **operators**, on these entities and structural relations are:

- the **operators** on the *structure* of a Petri Net are (i) to add or remove a place, (ii) to add or remove a transition, and (iii) to add or remove an arrow.
- the **operators** on a place are (i) to put a token into the place, and (ii) to remove a token from the place.
- the **operator** on the *behaviour* of a Petri Net is to **fire** on a transition. The *structural* results is that that operator removes all tokens from its input places, and puts a token in all of its output places.
- **enabled**: the higher-order *behavioural* state relation that connects the *behavioural* fire operator to a *structural* condition on a Petri Net, namely the fact that the Petri Net has a marking in which all input places of a transition are filled.

The following **structural constraints** of “*well-formedness*” must hold:

- a place is always connected to a transition by an arrow. This constraint is an **invariant** of the **composition** of Petri Net operators: a **set** of operators that add or remove arrows, places or transitions, is valid *if and only if* this constraint is satisfied before the set of operations starts, and also after it has finished.
- a Petri Net must have at least one place. Hence, it also has at least one transition.
- a token is connected to one single place at a time.
- **enabled**: the constraint **relation** of a transition that is satisfied *if and only if* all the transition’s input places contain a token.
- the Petri Net is a **weakly connected graph**: all places are connected to all other places via arrows and transitions.

In a [property graph](#) representation, **places** are nodes; **tokens**, **arrows** and **transitions** are relations on **places**; a **marking** is a higher-order relation on **tokens**; an **enabled condition** is a higher-order relation on one **transition** and a set of **tokens**.

The following **behavioural constraints** must hold:

- a **transition** can only **fire** after it has been **enabled**.
- a Petri Net with zero **tokens** is valid.
- putting a **token** in a **place** is an **idempotent** operation: doing the operation a second time on a **place** that already contains a **token**, gives the same result as doing it once: a **place** with one single **token**.

With all of the above-mentioned entities and relations, the **behavioural model** of an **algorithm** that “solves” a Petri Net graph data structure has two major parts:

- **scheduling**: the **markings** for each **transition** determine *whether* that **transition** is tagged as **enabled**.
- **dispatching**: each **transition** that is **enabled** is **fired**, hence the current **marking** is updated.

Figure 2.12 sketches a the behaviour of a Petri Net, as the following series of states (from top-left to bottom-right):

- **empty marking**: none of the **places** at the **input** side of the **transition** **tr**, **p1** and **p2**, contains a **token**.
- **marking with one token** in the **place** **p2**.
- **marking** in which both **places**, **p1** and **p2**, are filled. The **transition** **tr** is now **enabled**.
- after the **transition** **tr** has **fired**, there is now a **token** in the **place** **p3** that is the **output** of the **transition** **tr**.

2.8.4 Representation: marking reaction table

The **marking reaction table** is the simplest form of the behavioural model of a Petri Net: it represents the *effect* of the **fire** operator on a **transition** in the form of the removal and addition of **tokens** in the **input** and **output** **places** of each **transition**. Here is the *marking reaction table*, for the one-**transition** Petri Net in Fig 2.12:

input places	transition	output places
p1, p2	tr	p3

And this is the *marking reaction table*, for the one-**transition** Petri Net in Fig. 2.13:

input places	transition	output places
p111, p121	tr1	p112
p112, p113	tr2	p114 , p122

2.8.5 Pattern: Petri Net with one flag array per coordinated algorithm

This Section explains the **semantics** of the **multi-algorithm coordination pattern**. Here is the list of **extra** entities, relations and constraints that the pattern uses, in addition to the meta models for the [Petri Net mechanism](#) and the [flag array mechanism](#):

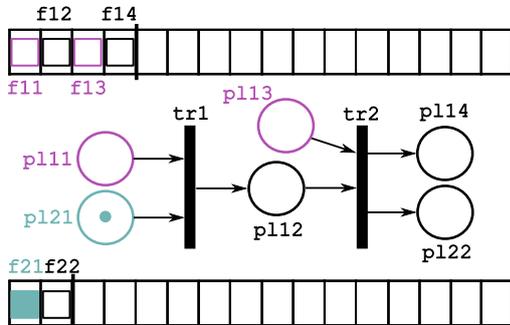


Figure 2.13: An example of the **composition** of one Petri Net (for the coordination *mediator*) with one flag array for each of the two coordinated algorithms. The similarly coloured and numbered flags in the array, and places in the Petri Net, represent how the places are *produced* and *consumed*: at “production”, the algorithm to be coordinated sets the first flag in the flag array. The Petri Net mediator notices this, and fills in the corresponding place in the Petri Net. When it has fired a transition, it fills in the flags in the arrays that correspond to the output places of that transition. In this way, each coordinated algorithm knows when it can “consume” its own coordination’s next step.

- in the coordination Petri Net data structure, a **place** is tagged as a **source**, if it is not the output of a **transition**. Similarly, a **place** is tagged as a **sink**, if it is not the input of a **transition**.
- a **source** is meant to be filled in by the coordinating algorithm *after* the coordinated algorithm has filled the *corresponding* flag in its own coordination flag array.
- as soon as the coordinating algorithm has filled in a **sink**, as the result of a decision making, it fills in its *corresponding* flag in the coordination flag array of the coordinated algorithm associated to that **sink**.

So, the coordinating activity has *exclusive access* to all **places** in the Petri Net, and to the flags in the flag array buffers that correspond to each of the Petri Net’s **sinks**. This guarantees synchronous execution of the algorithm that makes the coordination decisions.

The **coordination behaviour** of the pattern is as follows: putting a **token** in a **source place** means that the execution of the algorithm (on whose behalf the **token** is placed in the Petri Net) *waits* for a coordination with the execution of another algorithm (on whose behalf another **token** is placed in a **place** connected to the same **transition**). The waiting is over as soon as the coordinator puts a **token** in the output **place** that is associated with the waiting algorithm.

A coordinated algorithm does *not* place itself a **token** in a Petri Net. It does not even know that there is a Petri net involed in the coordination: it only sees the **flag array** via which it engages in its own part of the coordination, with *only* the coordination mediator. The latter has the same interaction with each of the coordinated algorithms. Hence, the pattern uses the following modelling parts, Fig. 2.13:

- **one Petri Net** mechanism for the **coordination mediator** algorithm. It uses the data structure for the bookkeeping of the coordination, and it is its only **owner**.
- **one flag array** protocol that coordinates how the mediator and one of the coordinated algorithms go through the coordination.

The bulk of the interaction described above takes place inside the mediator, in a synchronous way. The asynchronous parts are the atomic setting of flags in the flag arrays by each of the coordinated algorithms. The use of such a flag-based protocol implies **commitment** of each algorithm in the multi-algorithm coordination, to finish the waiting that it has started bu

filling the first flag in a protocol array. This commitment lasts until it has consumed its last flag in its protocol array.

2.8.6 Flag arrays as linear Petri Nets

Flag arrays are *linear* Petri Nets, so they *can* be given a Petri Net model. For example, the *stop-and-go* and *two-phase-commit* protocols, Figs 2.9–2.10.

The added value of a Petri Net model shows up when multiple algorithms need coordination, but they should not be aware of each other and hence should not engage in protocols together.

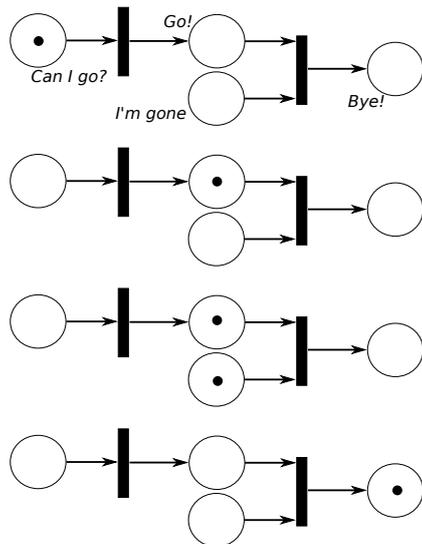


Figure 2.14: *Stop-and-go* with acknowledgement.

Figure 2.14 depicts the Petri Net version of the *stop-and-go* coordination, adding an extra confirmation step. In this coordination model, the coordinating algorithm gives the “Go!” decision, not by clearing the input place that was filled by the coordinated algorithm, as it did before, but by filling a new output place. This allows the coordinating algorithm to acknowledge the coordination explicitly, by clearing that output place.

2.8.7 Stop-and-go coordination — Barrier

Figure 2.15 depicts two possible Petri Nets, to coordinate the behaviour of multiple coordinated activities. The difference between both graphical representations is just graphical *syntactic sugar*.

In the multiple activities context, the same *barrier* use cases hold as in the simpler *two-activity* context.

2.8.8 Any-of and All-of coordination

Figure 2.16 depicts two common and complementary coordination semantics: *all-of* and *any-of*. The condition for a **transition to fire** in a Petri Net is that *all* of its input places are filled. However, some use cases want a coordination to proceed as soon as one of the inputs is filled. (A further elaboration could be to make the distinction between *exactly one* of them being filled, or *any number* of them.)

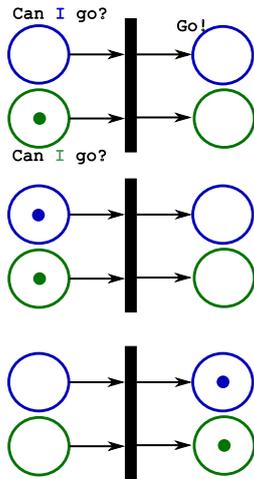


Figure 2.15: The Petri Net *structure* for a *Stop-and-go* coordination for multiple coordinated activities. The interpretation is the same as the Petri Net of one single coordinated activity, but the coordinating activity now waits for *all* of the coordinated activities before giving the “Go!” signal. The colours are added to the graphical representation to illustrate the “ownership” of particular places: The coordinating activity can read and write all places, the coordinated activities only the ones they “own”. The top figure foresees one output place per coordinated activity, in the bottom figure, all coordinated activities can observe the same output place.

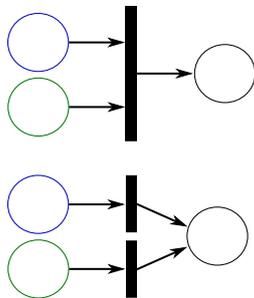


Figure 2.16: *All-of* (top) and *Any-of* (bottom) semantics of logical composition of tokens in places.

2.8.9 All-to-go, one-to-stop coordination

Figure 2.17 depicts the coordination use case to get a set of activities ready for a shared task:

- each activity that is ready to start the shared task, indicates its readiness by switching its *orange light* on.
- when **all** activities have their *orange light* on, the coordinator acknowledges their readiness, and indicates that the common task could be started by switching its own *orange light* on.
- the coordinator’s *orange light* is turned into a *green light*, as soon as the coordinated activities confirm their readiness via their own *green light*.
- the coordinator’s *green light* is turned off again, as soon as **one** of the coordinating activities signals, via its *orange light*, that it can not cooperate anymore.

The described Petri Net describes only the *nominal case*. In most use cases, the coordination complexity grows a lot, if one wants to incorporate all non-nominal situations.

Use cases: resource sharing such as conveyor belt, energy source, buffers, etc.

2.8.10 Fork-and-join coordination

Figure 2.18 depicts the Petri Nets that coordinates a *fork and join* of multiple coordinated activities:

- *fork*: the mediator fills the place `p1-1`, to indicate that it is ready to start the coordination protocol. Some time later, the first transition `tr1` fires, which gives rise to “forking” off two (or more) algorithms, 1 and 2. They inform the mediator that

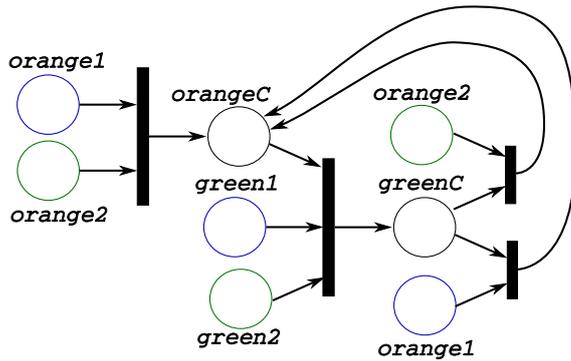


Figure 2.17: *All-to-go, one-to-stop* coordination.

they have started, by filling the places p_{11} and p_{12} . This fires transition tr_2 , which indicates the mediator to go to the *join* phase.

- *join*: after both algorithms have become active, the mediator waits for them to finish in place p_{1-2} . This waiting finishes when both algorithms have filled their places p_{12} and p_{22} . So, transition tr_3 can fire, indicating that both algorithms have “joined” their activities. This end of the coordination can be signalled explicitly via the p_{1-3} place.

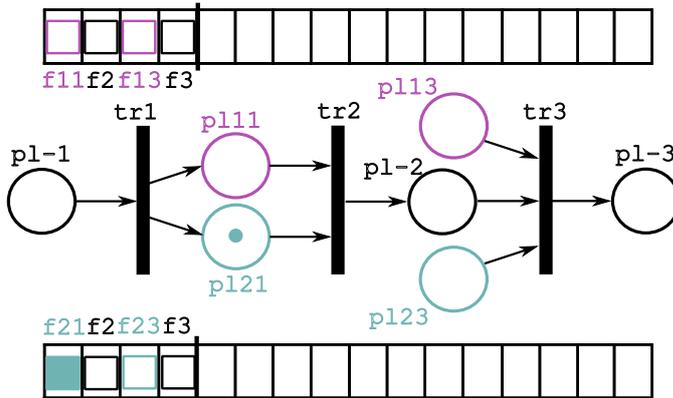


Figure 2.18: The Petri Net *structure* for a *Fork-and-join* coordination for multiple coordinated activities.

2.8.11 Policies: hierarchy, cardinality, colours

Hierarchical Petri Net

There are several reasons to introduce a hierarchical structure on Petri Nets, in the sense that transitions are first executed in the higher-order Petri Net, and only then in the lower-order Petri Net. The terminology “hierarchical Petri Net” is somewhat misleading:

- it suggests that there is one single Petri Net, but there are in fact **two Petri Nets**, referred to as “lower-order Petri Net” and the “higher-order Petri Net”.
- the hierarchy is in the **prioritization of the decision making**, not in the graphical structure: the higher-order Petri Net is observed and or executed first, and depending on that outcome, a particular part of the lower-order Petri Net is observed or executed.

Common reasons for introducing a hierarchy are:

- *inhibition* or *activation*: the lower-order Petri Net computations are only executed when there is a token in the higher-order’s “*coordination is active*” place. The logically equivalent case is that a token in the “*coordination is inhibited*” place prevents the transition computations to be executed.

The design driver behind this policy is to reduce unnecessary Petri Net computations. A configuration decision that can be introduced is *to clear* the lower-level Petri Net, every time the higher-order one enters in the **activated** place.

- *time out*: a *timer activity* is introduced in the coordination, and it influences the lower-level Petri Net computations by resetting the Petri Net if the timer goes off.

A common semantics follows a **two-phase commit** protocol:

- the timer activity fills a **place** every time it goes off.
- if the coordinated activity has filled its **place** before the timer, the Petri Net is cleared.
- otherwise, the last **place** is reached, which should trigger a corrective behaviour of the coordinated activity.

The time out model is sometimes extended with another higher level, in the form of *earliest* and *latest* allowable firing times for transitions.

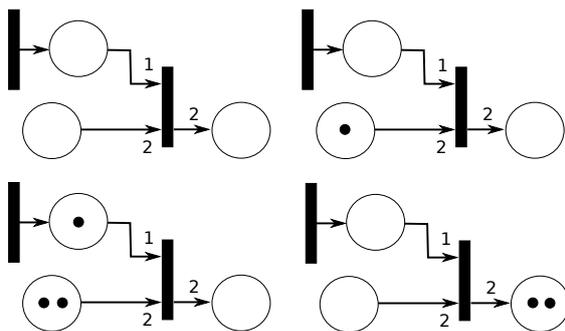


Figure 2.19: A Petri Net with *multi-token* places. The numbers on the arrows indicate the cardinality of tokens needed to enable the transition.

Multi-token places with cardinality

A **Petri Net** fires a **transition**, as soon as there is **one** token in each input **place**; the result is that (i) all **tokens** are removed from each input **place**, and (ii) **one** token is placed in every output **place**. This *single-token* behavioural constraint can be relaxed into a *multi-token* version:

- a **transition** is enabled as soon as each of its input **places** contains a **specified number** of tokens
- if the transition fires, it puts a **specified number** of tokens in each of its output **places**.

That number can be different for each **place**, it can be zero, but it should never change. In the example depicted in Fig. 2.19, the **transition** can fire (**marking** at the top) if there are two **tokens** in the **place** connected to the transition with a *cardinality condition* “2”, and one **token** in the other **place** with a *cardinality condition* “1”. The outcome of the fired transition (**marking** at the bottom) is to put “2” **tokens** in the output **place**.

It is obvious how to transform a *multi-token* Petri Net into **canonical form**: just duplicate each **place** with N tokens N times. So, the reason to introduce multi-token places is **syntactic sugar**, which makes designers gain some model creation efforts. What they loose is

mathematical expressivity: a Petri Net couples the coordination of two or more algorithms, so semantic clarity improves if one can identify, already in the *structure* of the Petri Net, which algorithms are connected to which **places**.

Coloured Petri Net for typed coordination decision making

The semantics of the adjective “**coloured**” is that a **token** comes with a *property* data structure (its “colour”), to represent any information that can be relevant for the coordination decision making. For example:

- the **identity** of the **token**.
- the **identity** of the *owner* activity of the **token**.
- the value of some parameters in the coordinated activity’s algorithm that it want to use to influence the coordination decision making.
- *history data* of the presence of the **token** in a given **place**.
- etc.

The variation in adding “colours” to the **Petri Net mechanism** is infinite; hence, a large number of mutually non-interoperable Petri Net dialects have emerged in the literature. This document considers the motivations above as **bad practice**, because it violates the **decoupling** design drivers behind this Section’s pattern. Section 2.8.12 advocates better approaches. (TODO: substantiate the claim above, with concrete examples!)

2.8.12 Best practices in Petri Nets

Here are some *best practices* to tackle the activity coordination challenges in an application:

- the *commitment status* of a coordinated activity in a Petri Net is always fully observable, for that activity and for the coordinator. When a coordinating activity decides not to commit to a coordination any longer, it is the coordinator’s decision whether and how to inform the other involved activities.
- a hierarchical Petri Net in the coordinating activity (more in particular the *inhibition/activation* semantics) allows “to glue together” Petri Nets with different sets of coordinated activities. This leads to more but smaller Petri Nets.
- each coordinated activity uses **two-phase commit** for each of the higher-order Petri Nets it is involved in, such as **time outs** and **inhibition/activity**. This (yet another higher level of coordination modelling) gives both coordinated and coordinating activity the time to react and reconsider, in use cases where the effect of the coordination decision comes with a “significant” risk. For example, the coordination was about agreeing on a “task contract” or not, and the conditions under which the “offer” was made can have changed in the meantime. Another example is the risk of **chatter**: if the Petri Net structure has loops, it is possible that the same **place** is filled or cleared via two different paths; and depending on the *timing* of the decision making logic in the coordinator, the filling state of a **place** can go through quick changes. A time-out of the “appropriate” length can provide the **hysteresis** to prevent such chatter.
- a **heartbeat** combines time-out and activity commitment: if the coordinated activity does not refresh its commitment in a coordination fast enough, the coordinating activity

can take the decisions (i) to de-activate that activity in the coordination it is involved in, and (ii) to follow the above-mentioned policy of informing the other involved activities.

- the Petri Net model identifies the coordinated activities explicitly. Keeping track of which activities are really needed in coordinations, can help to keep Petri Nets small. One observation that indicates that a Petri Net is too large, is that some activities are involved in the “beginning” of a coordination, but not any more later on.
- if a coordinated activity keeps “state” of the different phases of the protocol that it has already gone through, it can support also the coordination of possible **undo** operations (or “**undelete**”, or “**rollback**”): the coordination is aborted before it is finalized, and the state of the coordinated activity is then reset to what it was before it engaged in the coordination process.

2.8.13 Declarative and imperative Petri Net behaviour

The **behaviour** of a Petri Net is realised **computationally** by attaching to every transition a set of **Boolean operations** on its input and output places. The Petri Net is a **declarative** model of coordination, hence some extras are needed to eventually end up with the necessary imperative list of transition computations:

- *scheduling*: the declarative model requires a scheduling algorithm to decide when which set of Boolean operations will be computed.
- *non-blocking*: a proper **architecture** can prevent that the *whole* algorithm is blocked, because one of its *concurrent parts* waits for coordination.

2.9 Stream: peer-to-peer algorithm interactions via asynchronous data exchange

This Section describes the *stream* (or *channel*), as a **composable** meta model for the **asynchronous data exchange** between algorithms. **Producer and consumer**¹³ is a context-neutral terminology to differentiate the *roles* (but not the properties) of the (in general) two *peer* algorithms involved in the data exchange.

The differentiating concept, with respect to, for example, Publish-Subscribe or message passing, is that the exchange involves some form of **dialogue**: producers *and* consumers **react** to each other’s information by creating a new data exchange that **refers to explicitly** to the original message.

The coordination protocols required for such dialogues are not restricted to only a cyber or digital context: all of them are **digitizations** of **proven approaches** from **reality** and **real life**, where people and organisations apply these protocols to coordinate their interactions.

2.9.1 Appearance in the real world

This Section provides an overview of the major protocol families, with a short description of which coordination problems they solve in the real world. The families are not mutually exclusive, so a particular interaction architecture typically selects a mixture which fits its

¹³For channels, a terminology is used that fits closer to the context of “communication”: “producer” becomes “transmitter”, and “consumer” becomes “receiver”.

purposes best. The terminology is *not* adapted from the real world, but is how these information exchange patterns are known in the cyber world. A system can contain instantiations of *multiple* of the following interaction patterns, even at the same time and between the same two algorithms:

- *Message passing*, with or without a *broker*: the producer puts an envelope or a box with the address of the consumer into the latter's mailbox; or in the mailbox of the post office, that acts as mediator. The producer has to know the identity of the consumer; but not the other way around.

An envelope is a container for any kind of information, however complex, and the role of the message passing service (the public post office, or private package delivery companies) is to deliver the container, without having to worry about its contents. Or rather:

- there are different types of envelopes and boxes, for classes of contents.
- the container may have indications about how to be handled, e.g., *fragile, this side up, heavy,...*

- *Publish-Subscribe*, with *broker*: companies produce goods, that are delivered to warehouses and later to supermarkets and shops, where consumers go and buy them. Property owners publish their intent to sell or lease their property, often via **real estate agents**, to reach buyers.

The warehouse, shop and realtor act as **broker** in this exchange protocol. The producer and consumer need not interact with each other directly; they even need not be aware of each other's existence.

- *First-In, First-Out* (FIFO) **queue**: this is a special case of *Publish-Subscribe*, but with a one-on-one relation between producer and consumer. This is how a **vending machine** works.

Last-in, First-Out, or **stack**: a queue with another order of emptying.

- *Request-Reply*: the producer provides a good that is delivered to the consumer, and expects a response from the latter. This is how a **hotel reservation**, works, or one's **tax return**.

- *Stream*, with *backpressure*: the producer delivers a series of products, in several batches, and transfers ownership to the consumer batch per batch. The consumer can trigger the producer to deliver faster, or slower, depending on its own speed of processing the delivered goods. Similarly, the producer can trigger the consumer when its delivery risks not to be deliverable, because the consumer's goods receiving buffer is almost full.

This is how delivery of construction material pallets to a house building site works.

- *Submission-Completion*, with *mediator*: the producer *submits* several different pieces of "stuff" to the consumer, at irregular intervals; the consumer has a *mediator* role, so it *completes* some work on each of the submitted "stuff", and provides an answer back to the producer, also at irregular intervals and not necessarily in the same order as the producer submitted everything.

This mechanism is used for school assignments, where students hand in homeworks or project reports, and get their teachers' feedback, possibly in several iterations.

- *Blackboard*: in a classroom, both the teacher and the pupils can add and remove writings on the class' blackboard, and all changes are visible to all immediately, and in parallel.
- *Bus*, including *fieldbus*: all participants (“information”) can jump on and off the bus at any time the bus passes by. Once one gets off the bus, participants (“information”) take no part anymore in the transportation (“communication”), and hence are “lost”.
- *Interactive Connectivity Establishment*: interaction protocol to let two peer activities find each other and establish a connection. It is possible that a connection can only be created by involving a third party *relay*.
- *Offer/Answer*: one peer in a connection provides the other peer with one or more offers, each suggestion a concrete choice of how to proceed forward in their interaction. The other peer is expected to provide its answer to that offer.
- *Broadcasting*: radio and TV distribute their information to all the public in parallel. This is very effective one-way interaction, but the two-way aspects of reaction and interaction are difficult.
- *Actor model*:
(TODO: relies on message passing between actor activities, but have a mediator to decide how to react to incoming messages, and how to react to actor activities crashing.)
- *Dialogue queries*: a special case of *Submission-Completion*, but with direct responses from the consumer.

The exchange mechanisms above can be complemented by one of these two policies:

- *push*: the producer takes the initiative to deliver the exchanged data.
- *pull*: the consumer takes the initiative, and triggers the producer to produce.

In addition, the following [higher-order flags](#) can be used in the Stream context, too: *active*, *timeout*, *preempt*,...

2.9.2 Mechanism: producer–consumer array

This Section presents the mereo-topological and behavioural meta model of a stream (Fig. 2.20):

- the **stream** is an **ordered list** (or, an **array**) of **data chunks**.
- all **data chunks** are of the same **type**.
- each of the **data chunks** has a **unique identifier**.
- a **producer activity** acts as the **source** of the stream, by putting **data chunks** into the stream's array in an **order that will never change** anymore. In the same action, it gets **ownership** of these **data chunks**.
- a **producer** realises the **transfer of ownership**¹⁴ of **data chunks** that it owns, towards the **consumer**, (i) without leaving any holes in the ordered structure of the stream, and (ii) at the moment that fits best to its own activity's behaviour, **asynchronously** of the **consumer** activity. The **producer** can never revoke any ownership transfer.

¹⁴This is Rust's “move” of a buffer half from producer to consumer. The Go programming language uses this approach natively, via its policy “*Do not communicate by sharing memory; instead, share memory by communicating.*”.

- (hence) the **producer** determines where the (ownership) **barrier** in the stream lies at any moment, that is, what the identifier is that indicates the separation between the (“upstream”) part of the stream owned by the **producer**, and the (“downstream”) part owned by the **consumer**.

- a **consumer** can always *observe* that **barrier**, without interfering with the **producer**.

The **consumer**’s access to the **data chunks** it owns is **never disturbed** by a **barrier** shift, because of the strict order of **data chunks**. In the worst case, the **consumer** accesses less **data chunks** than it already owns, because the **producer** has transferred ownership after the **consumer** decided to start processing a series of **data chunks**.

- **producer** and **consumer** can **process** (“**mutate**”) all the **data chunks** in the stream part they own, in any order, but without changing that order nor their identifiers.
- the **consumer** acts as the **sink** on the stream, by taking **data chunks** out of the **stream**, forever and respecting their order, that is, without leaving any holes in the array.

2.9.3 Best practices

The stream concept has become dominant in several domains (such as the **World Wide Web**), because it composes:

- one or more uni-directional *channels*, each with a **strictly ordered** set of message data structures.
- allowing *missing data*, with identification of which data slots are missing exactly.
- a *bi-directional* exchange of *status data*.
- the **asynchronous and loosely-coupled** nature of the interactions between the *producer* algorithm and the *consumer* algorithm.
- allowing those algorithms **to adapt their behaviour** to the actual status of their interaction.
- clear **ownership semantics**: at all times there is no ambiguity about (i) which part of the stream is owned by the producer, and (ii) when it transfers ownership to the consumer, without breaking the contiguity of the owned parts of the stream.
- allowing the algorithms to access the part of the buffer that they own in a **random access** manner.
- hence, allowing **in-place** operations on the owned buffer parts.¹⁵

There is nothing in the stream meta model that prevents ownership transfer to be repeated indefinitely: the part of a stream that a consumer got from a producer can be returned to the producer, and so on. That means that

- the terms “producer” and “consumers” reflect only the *role* that an algorithm plays on a particular *part* of a stream, and
- an algorithm can switch to different roles on different stream parts at different instants in time.

This allows algorithms, for example, to engage in “dialogues” in which they process an initial information exchange data chunk incrementally and interactively.

¹⁵Hence, the name “**stream**”.

2.9.4 Policy: status flags for a stream’s activity and backpressure

A stream has a **uni-directional** transfer of **data**, from producer to consumer; but it is straightforward to add **bi-directional** exchange of **information about the status** of the stream, by means of the following `status_flags`, Fig. 2.20:

- **activity** status flags: one flag for the producer and one flag for the consumer, which they use to indicate their **own current activity** on the stream: they can be **Active**, (that is, producing, respectively, consuming), **Inactive** (that is, the “other side” should not expect any change in the part of the stream owned by the inactive peer), or **Pausing** (that is, producing/consuming activity can restart at any moment).

This status flag is an instance of the generic coordination use case of **All to go**, **One to stop**.

- **back_pressure** status flags: a producer activity could fill a stream faster than a consumer activity can process it, or the other way around, so each uses a status flag **to inform the other activity** that it wants to “speed up” or “slow down” the throughput in the stream. This reactive mechanism is called the **flow control** on the stream, or **back pressure**. Therefore, there are **two status flags** with the following meaning:
 - **HighWater**: this represents the information to the other activity of “*please, stop being active on the stream for a while*”, because the activity’s side of the buffer is getting *full*.
 - **LowWater**: this represents the information “*please, start being active on the stream*”, because the activity’s side of the buffer is getting *empty*.

“Full” and “empty” are variables whose values must be **configured** by the application.

The combination of both types of status flags allows to keep the *decision making* about **inter-activity interactions** inside of the boundaries of each activity itself.

2.9.5 Mechanism: composition of streams

The composition of streams is simple, Fig. 2.20: one of the processing actions that a consumer activity can do on a stream is to copy the data from one stream and use them as producer on another stream. Asynchronicity and ownership remain fully transparent and unambiguous under this composition. (Which would be lost by the alternative composition mechanism of *sharing* data chunks between two streams.)

This composition pattern can be repeated “endlessly”, as illustrated in Fig. 2.21. The terminology “producer” and “consumer” is replaced by the neutral term “activity”, since each activity can, in general, act as a producer for the “downstream” part of the stream, and as a consumer for the “upstream” part. The strict order, unique identifier, and ownership features are all maintained under this serial **peer-to-peer** composition.

2.9.6 Pattern: composition of data and metadata streams

In the above-mentioned context of using streams to increase the **loose coupling** in system architectures, it makes sense to let streams carry the information that allows the consumer to interpret the producer’s data chunks without any interaction with third parties. In other

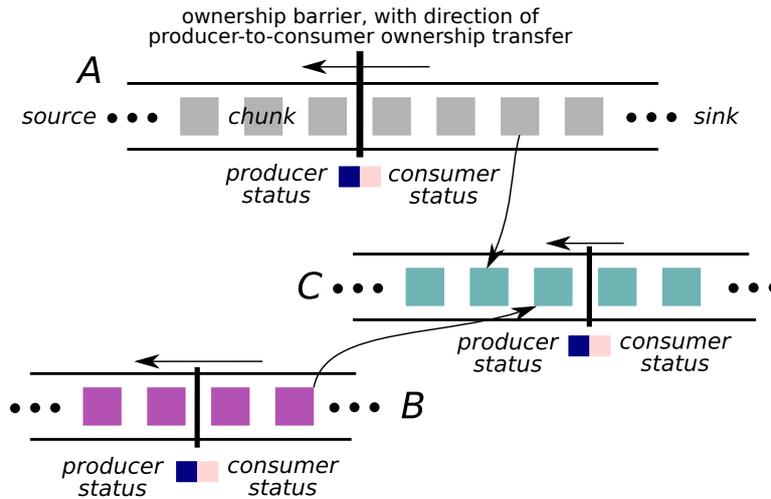


Figure 2.20: Example of *streams* and their compositions, including not only the data buffer, but also the extra semantic primitives that turn a *buffer* into a *stream*: (i) the *status flags* (reflecting, both, the activity on the stream of the producer and consumer, and whether one of them wants to inform the other one about being more or less active on the stream they share), and (ii) the *barrier* that indicates to where the ownership transfer of data chunks from the producer to the consumer has progressed. The example also shows that the producer activity of the C stream acts as a consumer on the two other streams.

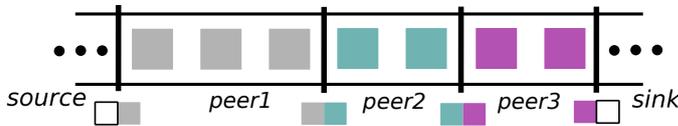


Figure 2.21: Producer-consumer stream with multiple peers.

words, a stream must have a mechanism to **piggyback metadata** on top of the data, to increase the **data lineage** (also called “**data provenance**”).

Figure 2.22 shows the simplest approach of adding a dedicated metadata stream. The metadata is typically a lot smaller than the data, and one entry in the metadata stream can represent the interpretation of many (often, even all) chunks in the data stream. Of course, the chunks in both streams are highly dependent and are hence best generated together by the same producer.

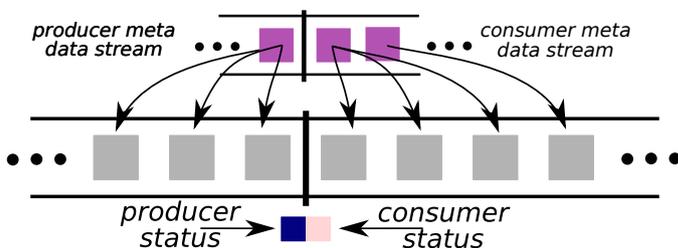


Figure 2.22: Producer-consumer stream composed with an extra metadata stream. Each chunk in the latter encodes metadata for a contiguous series of chunks in the former. Both streams share the same status flags, and ownership barriers.

Time series are a common data structure that requires metadata, namely the information about which clock and which time representation was used for the **timestamp** of each chunk in the stream. For example, indicating the **time zone**, **time epoch**, **temporal resolution**, **jitter**,

etc.

2.9.7 Pattern: composition into Submission-Completion streams

An extension of [Request-Reply](#) towards *series of related* (CRUD) requests is the pattern of **Submission-Completion streams**:¹⁶ producers of requests add them to the submission stream, and come back later to collect the result from the completion stream, Fig. 2.23.

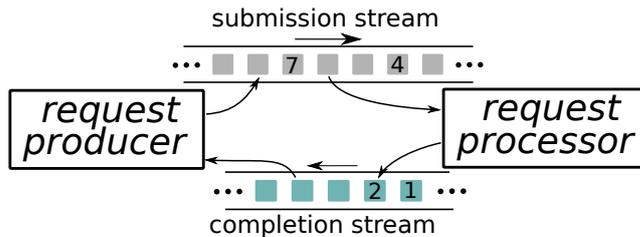


Figure 2.23: Submission-Completion streams. The processed stream goes to the original producer; the order of the items in the *completed* stream can be different from the order in the *submitted* stream.

The pattern involves an input (“submission”) stream, and an output (“completion”) stream:

- the producer submits its request to the *request submission* stream, and the request processor takes it off and processes it when it sees fit.
- the request processor submits its result to *request completion* stream, and the producer takes it off when it sees fit.
- the request processor need not return the processed requests in the same order as the producer has submitted them. That means that the index in the stream can not double as unique identifier of each request, so the latter must contain extra fields for that sole purpose of identification of a request.
- only when both streams are **coordinated by the same activity** (“*mediation*”), one can guarantee end-to-end **consistency** of request processing.
- one (and only one) of both peers, the producer or the consumer, can act itself as coordinating activity. Or, alternatively, a third-party activity is given the responsibility for the mediation.

The Submission-Completion streams pattern has been used since decades already, such as in [Channel I/O](#) for mainframe computers; [task queues](#); ¹⁷ [disk access libraries](#); or [multi-host communications](#). More recent instances appear in: [asynchronous I/O](#) via “[I/O rings](#)” that an [operating system kernel](#) provides to a [application](#).

2.9.8 Policy: submission-completion stream with partial dialogues

The [mechanism](#) of submission-completion supports the explicit bookkeeping that data chunks have been submitted and completed. Of course, an application can add [metadata](#) to the data chunks that refer to the *semantic contents* inside the data chunks. Hence, when a peer puts a data chunk into the stream, it can add a symbolic reference to a specific part of the other

¹⁶The simplest version of this pattern is the **Command stream**.

¹⁷The pattern is especially powerful in applications that want to support the **cancelling** of previously submitted but not yet executed tasks.

peers earlier data chunk to which it reacts. The result is a dialogue, about the whole or a part of an original data chunk, Fig. 2.24.

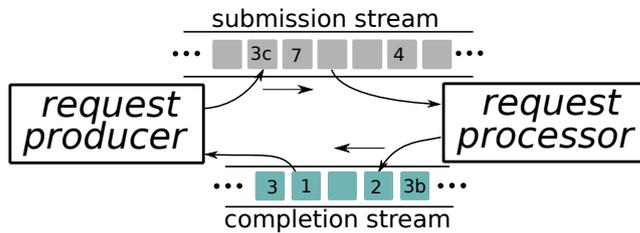


Figure 2.24: Submission-Completion streams with partial dialogues: both parties can react to parts of each other’s data chunks, with explicit reference to the reacted to parts.

2.9.9 Pattern: function exchange via task queues

The “data” that is exchanged via a stream can also be the symbolic description of a **function**. And even of a function and (part of) its data. This pattern allows one algorithm to let another algorithm execute the function, using the *submission-completion* pattern:

- there is one *submission* data chunk, with the above-mentioned symbolic model of a function and its closure.
- there can be one or more *completion* data chunks, yielding just the eventual return value of the function execution, or a stream of execution progress updates.

Several of the above-described “one-off” realisations of the pattern can straightforwardly be composed into “computational dialogues” between algorithms; that is, part of the returned data is reused in a next function submission, hence allowing to keep *state* in the exchange.

This pattern is used, for example, in **REST** interfaces on the Web. This document’s main use of the pattern is in **task queues** between activities.

2.9.10 Publish-Subscribe & Request-Reply as stream architectures

Activities must **choose** an **interaction pattern** (or “**communication protocol**”) to **coordinate** the information exchange via each mutual interface. The **design forces**, that drive the pattern’s trade-off in different directions, are:

- number and anonymity of participants and channels.
- longevity of the interface.
- necessity to support “dialogues”.
- necessity to **mediate** the traffic inside one interface.
- necessity of mediation between several interfaces.

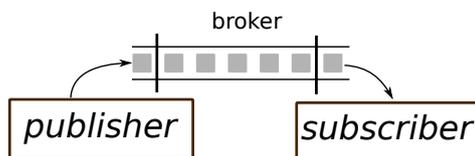


Figure 2.25: *Publish-Subscribe* communication realised by a streams composition. Both publisher and subscriber have ownership of only one single entry in the stream.

The **Publish-Subscribe** and Request-Reply communication patterns can be realised as particular cases of stream compositions:

- *Publish-Subscribe* (Fig. 2.25): the publisher activity has a stream to the broker activity, without flow control or backpressure support, and with the extra policy that the pub-

lisher can only fill the stream one entry at a time. A similar configuration holds for the stream between the broker and subscriber activities.

- *Request-Reply*: this is the Submission-Completion architecture, with the same constraints between activities as in the Publish-Subscribe case.

Differences Publish-Subscribe & Request-Reply

Two major pattern versions exist: *Publish-Subscribe* and *Producer-Consumer*, whose major differences are summarized in the table below.

Publish-Subscribe	Producer-Consumer
anonymous peers	peers know each other's identity
communication between peers outsourced to broker	peers deliver messages one-to-one themselves, via a channel (or "connection").
hence, one-way, send-and-forget	hence, two-way stream with backpressure
hence, connection concept does not even exist.	connection has its own unique identity , so that it can survive even runtime changes in where producer or consumer are deployed.
topic centred: an interaction involves <i>one single</i> data structure of a <i>fixed</i> type	message centred: an interaction is the <i>composition</i> of different data structures
one session per topic type	one session per Producer-Consumer pair.
mainstream policy: FIFO queue	mainstream policy: random access to all entries in buffer
depth of interaction memory : one message	depth of interaction memory : all messages in a conversation or dialogue .
Quality of Service owned by brokerage middleware	Quality of Service owned by application

A simple "hybrid" between both exists: **Request-Reply**, which merits to be identified as a third major pattern. It is a producer-consumer dialogue with a memory of one *request* message and one *reply* message.

Communication patterns are commonly dealt with in the general-purpose ICT literature, and are hence very mature. Detailed discussions are beyond the scope of this document, and the interested reader is referred to literature, e.g., [ZeroMQ documentation](#).

2.10 Finite State Machine: behaviour coordination of one single algorithm or activity

This Section presents the meta model of the **Finite State Machine** (FSM), as the **abstract data type**¹⁸ to represent the **behavioural state**¹⁹ of **one single activity**. The description

¹⁸Indeed, an FSM is a passive model that represents information about activity and behaviour, but it is *not* a behaviour or an activity in itself. In other words, the activities and behaviours that use an FSM data structure for their coordination must look up the state of the FSM model themselves, whenever it is an "appropriate time" to do so. There is nothing in the FSM model itself to help make that decision.

¹⁹"State" is probably the most over-used term in systems science and engineering. Hence, the meaning of the word can only be complete if the context in which the term is being used is given explicitly, too. In this

maximizes the *separation* of (i) *mechanism and policy*, and (ii) *structure and behaviour*. This decoupling is not an obvious ambition, because in the long history of state machines various FSM meta models have been “standardized”, each (more or less implicitly) introducing domain-centric couplings between all these aspects. For example, **Harel statecharts**, **Mealy machines** or **Moore machines**, [21, 103].

2.10.1 Mechanism Part 1: state to coordinate one activity’s behaviour

The FSM mechanism is introduced because an activity must be able **to realise different behaviours**: in each of its behavioural “**states**”, the activity executes a different composition of algorithms; there is the strict constraint that an activity executes **only one behaviour at a time**; in other words, it can only be in one state at each moment in time. Each of these to-be-coordinated behaviours is represented by a discrete parameter (an “**enumerated value**”), called a **state**, or a **mode**.

Take the example of the **above-mentioned** activity that does LIDAR- and IMU-based velocity control for a robot: its behaviour will be different when other control and estimation algorithms are used. It makes no sense to switch between these algorithms randomly, or instantaneously, so the switching must be *coordinated* explicitly.

To the **outside** of an activity (that is, an activity’s behaviour that can be observed by other activities), the state just *represents* a behaviour with an “identity”, and one does not have to know the details about *how* that behaviour is realised, that is, to know about the exact composition of algorithms inside the activity. The important information is about *how to interface* with the behaviour. This implies that also the protocol that is used on an activity’s interface can change when the activity changes behavioural state. That information must be in the activity’s **data sheet**, so that the activity at the other end of the interaction interface has access to the information to decide to adapt its own behaviour to state changes in the interfaced activity.

For the **inside** of an activity, each behavioural state corresponds to one particular choice of

- algorithms to realise the behaviour.
- interfaces to realise the inter-activity interactions. This interaction requires dedicated algorithms.
- **set of constraints** (or “**guards**”) on the algorithms and their interactions, which must be monitored. Again, this monitoring requires dedicated algorithms.

2.10.2 Mechanism Part 2: events to communicate coordination changes

Previous Section explained the complementary roles of **flags** and **Petri Nets** to coordinate various algorithms inside one or more activities. This Section introduces another mechanism for coordination, because it is relevant for one activity

- *to inform* other activities of changes in its own (outward-visible) behaviour, or
- *to influence* other activities to change their (outward-visible) behaviour.

Events are the proven **mechanism** to realise this ambition: an event has a very simple semantics, in that it just represents “that something has changed”. That change information can then be **communicated** via any type of **interaction channel** between activities. The

Section, *state of an activity* would be a semantically more complete term.

“other end” of such a communication is responsible to react to incoming events. That so-called **event handling** has the following operations:

- **fire**: the action *to produce* an event and *to send* it over an interaction channel.
- **handle**: the action *to receive* an event from an interaction channel, and *to consume* it by (preparing) the execution of an algorithm that must decide about possible behaviour switching.

The typical **interaction semantics** of events is:

- **send-and-forget** for the “producer”, and **handle-and-forget** for the “consumer”: they are used once in the consumer and then removed.
- **broadcast**: the events fired by a producer can be communicated to any number of consumers, that consume events independently of each other.

The interaction channel through which to communicate events, can in some cases be as simple as a flag in shared memory; in other cases, an appropriate **abstract data type** has to be sent over an inter-process/inter-computer communication channel. As soon as an event has been communicated from one activity to another, it is the latter activity’s own decision (i) whether or not to react to that event, and (ii) which behaviour it is switching to.

For *separation of concerns* reasons, it is important to avoid a common *worst practice*: the activity that fires an event gives it a name that reflects the behavioural switch that is the *intended outcome* of the event at another activity. The better approach is to give a name that reflects the *constraint violation* that took place inside the firing activity, and that was the cause for firing the event. For example: `distance_to_obstacle_below_MINVALUE` is a better name for a LIDAR processing activity to use than `switch_to_collision_avoidance_control`; this latter reaction *could* be the result of the control activity’s reaction to the event, but that reaction decision is the sole responsibility of that control activity, and not of the LIDAR processing activity.

2.10.3 Mechanism Part 3: transition and event reaction table

The **structural** part of the meta model has the following primitives (Fig. 2.26):

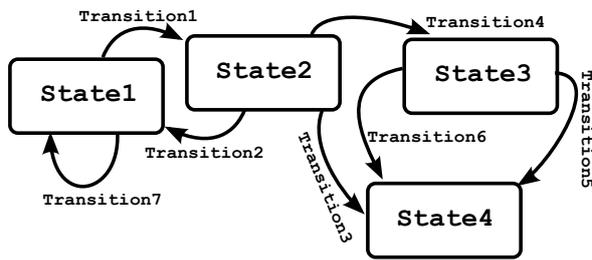
- the **state**.
- the **transition** from one state to another state.

The composition of these primitives must satisfy the ***syntactically well-formedness constraint*** that the system can be in one, and only one, state at each moment in time. There are *no constraints* on how many **transitions** can exist between two **states**. A **state** is also allowed to have a **transition** to itself.

The **behavioural** part has the following primitives (Fig. 2.27):

- the **event**: a **Boolean variable** that represents (the change in) the **truth value** of one of the **above-mentioned** pre, per and/or post conditions, or of a **logical composition** of them.
- the **event reaction table**:²⁰ the data structure that models the reaction of a FSM to incoming events. That is, it supports the decision
 - to **transition** to another **state**, and
 - to **fire** a set of **events**, as a result of the **transition**.

²⁰This term *event reaction table* is not standardized outside of the scope of this document. **Decision tables** and **decision trees** are popular instantiations of the concept. The **Karnaugh map** is one of the traditional representations of the logical computations.



start state	transition	end state
State1	Transition1	State2
State2	Transition2	State1
State2	Transition3	State4
State2	Transition4	State3
State3	Transition5	State4
State3	Transition6	State4
State1	Transition7	State1

Figure 2.26: An example of the **structural** (mereo-topological) part of a *Finite State Machine* model. Left: graphical representation; right: the semantically equivalent tabular textual form. In a **property graph** representation, **states** are nodes, and **transitions** are relations.

handled event	resulting transition	event fired
e1	Transition1	
e2	Transition2	E2
e3	Transition3	E3
e2	Transition4	E4
e1	Transition5	
e6	Transition6	E7
e4	Transition7	E4

Figure 2.27: One possible **behaviour** for the FSM in Fig. 2.26, modelled as an *event reaction table*, in **canonical** form.

2.10.4 Canonical form of a Finite State Machine

The **canonical form** of an event reaction table satisfies two **constraints**:

- each transition is triggered by **one single** event.
- **at most one** event is triggered by each transition.

It is *not* a constraint that every transition must be triggered by a *separate* event, nor that a transition *must* result in an event being fired. The canonical case reflects the *best practice* of **decoupling** the following concerns:

- the *reason why* an application requires events.
- the *decision making* of when to switch behavioural state.
- the *decision making* of whether to react to a behavioural state switch by firing an event.

This decoupling provides the modular basis to realise all behavioural FSM variants by:

- **composition** of a *policy* model as a higher-order constraint on the canonical *mechanism*.
- **configuration** of such *policy* models to satisfy the specific requirements of a specific application.

Policy models are introduced in the following Sections.

2.10.5 Policy: event composition

Table 2.28 shows an extension of the canonical finite state machine behaviour that is used in many applications: the application has *multiple events* that it wants to react to in order to trigger *one particular* transition.

The *best practice* policy is, without any loss of semantic expressivity, to introduce a **event monitoring** function that generates the **single canonical** event whenever the required

handled events	resulting transition	events fired
$e_1 \vee e_3$	Transition1	
e_2	Transition2	E_2
$e_3 \wedge e_1$	Transition3	E_1, E_3
$\neg e_4$	Transition4	E_4
$e_1 \wedge e_3$	Transition5	

Figure 2.28: An example of an *event reaction table* with *composed events*.

logical composition of application-centered events occurs. Similarly, a monitoring function can capture a transition generated via the canonical event reaction table, and react to it by firing the requested set of application-centered events. The advantage of this practice is that the same FSM can be used with different event monitoring logic, which allows to decouple three complementary but different design efforts:

- **inside behaviour:** the FSM is the mechanism to structure the internal behaviour of an activity, and this is best done by designers who are responsible for the ins and outs of the “thing” whose behaviour must be controlled.
- **outside behaviour:** an activity fires events to let the outside world know that it has “done something”. The naming of these events should conform to the terminology used in the [data sheet](#) of that activity. So, the naming of events is best done by designers who are responsible for the outward facing behaviour of the activity.
- **system behaviour:** the system designers, and *only* them, have the information about how to couple the outward facing behaviours of several activities together. This often requires renaming events, or grouping them together in logical monitors, etc.

2.10.6 Policy: event distribution and conversion

The events handled by an activity’s FSM can be generated by other activities and/or by the activity itself; similarly, the produced events can be reacted to by other activities and/or by the activity itself. The consequence for the system architecture is the need for:

- **distribution** of events between activities.
- **conversion** of event streams from several sources into the ones in an FSM’s event reaction table, and the other way around for the generated events.

It is the *system architect*’s responsibility to avoid the *event name space pollution*, that can occur under the just-mentioned policy composition; the *best practice* here is *to inject*, into the event loop schedule, the *translations* of event names between the application’s context and the canonical FSM form.

2.10.7 Policy: hierarchical states

The *structural* relation of **hierarchy**, Fig. 2.29, is a commonly used addition to the canonical Finite State Machine. The *behavioural* part of the FSM does not change: the activity **must** always be in one and only one of the **leaf states** of the hierarchical structure. What *does* change is the **view** on the outward facing behaviour: the hierarchy allows system designers to reveal different levels of detail of an activity’s behaviour to different other activities in the system. The hierarchical structure model has the following semantics:

- **containment relation:** a **state** can be **contained** in another “super” **state**.

- **containment tree constraints:** a state can only be **contained** in one single “super” state, and these containment constraints can only form a **tree**. The **leaf states** of this tree are the real behavioural states.
- **shared transitions:** a transition from a super state to another state represents the set of transitions from **all** of the internal states to the same other state.
- **non-shared event conditions:** each internal state **can** have a different event condition for the above-mentioned shared transition.
- **initial_state, final_state:** when the overall state machine transitions *into* the super state, **one** internal state **must** be selected as the **final_state** of *that* transition; that state then becomes the **initial_state** of any *subsequent* transition, so one often sees this state being tagged as “initial state” in graphical representations of an FSM. One internal state **can** be selected as **final_state**, which means that an internal transition into this **final_state** **automatically** gives rise to a transition away from the super state, *if* that super state has only one possible transition.
- there **can** be a policy **to remember** the internal state that the FSM is in when a transition takes place out of the super state, so that this so-called **history state** will be selected as the **initialstate** for the next transition into the super state.

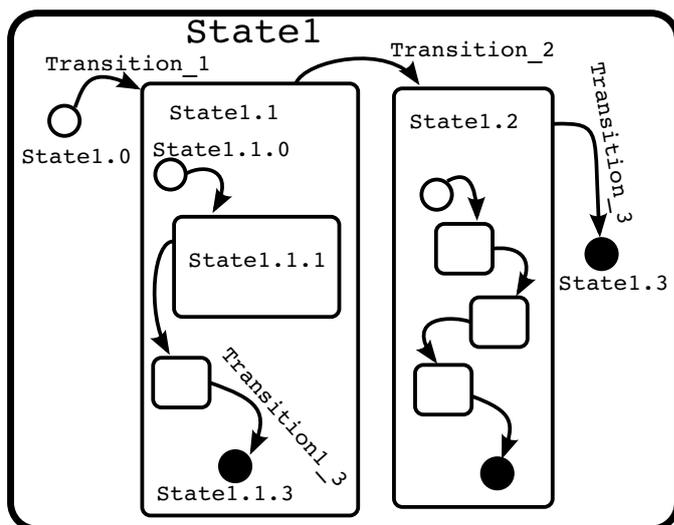


Figure 2.29: An example of a hierarchical *Finite State Machine*. The open and filled circles represent **initial_state** and **final_state**, respectively.

The major design motivations to choose for hierarchical state machines are the following:

- **reduction of externally visible state explosion:** one can “hide” all states below a certain level in the containment tree, hence reducing the (apparent) complexity of the number of states and transitions.
- **improved semantic value of communication:** it *may* be useful to provide for each the many complementary interaction purposes (logging, reporting, introspection, interfacing, information hiding,...), another view on the FSM than the full structural model.
- **just-in-time creation**, or **lazy evaluation**, of state machines from a model. The reason can be to reduce the effects of state explosion on the **runtime memory consumption**, but the policy also allows for **runtime adaptation** of the discrete behaviour of a system by the *just in time* creation of new state machines via **runtime reasoning** on the basis of (declarative) behavioural models in the application.

2.10.8 Policy: Life Cycle State Machine

Almost no component or system can provide its *services* to other components or users, without itself making use of the services of multiple *resources*. These resources can be services of other components or sub-systems, but also physical and/or infrastructural resources (energy, CPUs, memory, communication bandwidth, etc.), and behavioural resources ([interfaces](#), [algorithms](#), etc.). Hence, it is not realistic to expect the services of a component to be provided, and/or reconfigured, instantaneously.

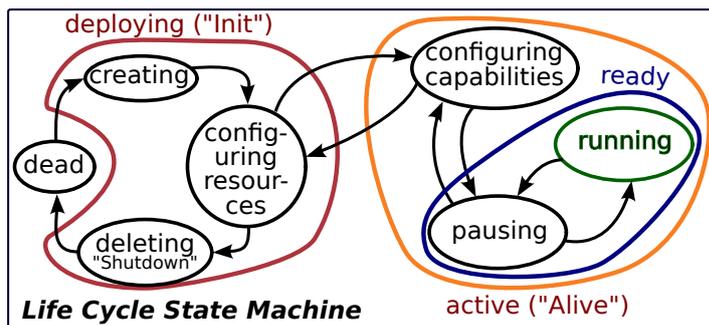


Figure 2.30: The Life Cycle State Machine meta model coordinates the configuration of the resources required for a certain activity before that activity's capabilities can be offered as a service to third parties.

The *Life Cycle State Machine* (LCSM), Fig. 2.30, has been (since decades, in one form or another), the mechanism to coordinate the availability of *internal* resources for the provision of *external* services. That is, to make sure that a component's own capabilities are configured appropriately *before* it provides services to others. The LCSM concept appears in many incarnations and using many alternative names for the states; this document uses the following terminology and semantics:

- **dead**: the activity is not able to do anything itself, and must be *brought to life* by another activity.
- **deploying**: the activity is busy with finding and configuring all the resources it needs, to be able to offer its services to others. While doing so, the activity **does not react to external coordination events**. This super state contains three semantically complementary states:
 - **creating**: the software resources are created, with which the activity does the bookkeeping of its own behaviour and of its usage of its external resources.
 - **deleting**: the above-mentioned software resources are cleanly removed.
 - **configuring resources**: the activity is configuring the service resources it requires for its own operation.

Resource *configuration* also includes *re-configuration at runtime*, so the latter state can be transitioned to and from multiple times during the life time of the activity.

- **active**: the activity is visible to other systems, to coordinate with them its engagement in mutual interactions. So, in this super state, the activity **reacts to external coordination events**, but with two semantically complementary behaviours:
 - **configuring capabilities**: the activity **is not** providing its services, because it (re)configures its own capabilities to provide a particular service configuration.
 - **ready**: the activity **is** providing its services to other systems. This super state contains two semantically complementary states:

- * **running**: the activity provides its services.
- * **pausing** (or “*idling*”): the provision of the activity’s services is put on hold, but can resume immediately. This happens when it must wait for an LCSM state change in one or more of the “peer” activities it interacts with; for example, during a **All to go**, **One to stop** coordination.

An example is the service to control the motion of a robot: in its deployment superstate, it must not only create and configure the data structures and functions that it needs for its motion control service, but it must wait till other services have become active (e.g., a kinematics computation service, and the input/output device drivers to the robot’s actuators and encoders). It can provide several types of motion control services, like force, impedance, velocity or position control; sometimes it will have to pause its own service, because the end user task system is itself not active yet.

2.10.9 Policy: don’t care, history, timeout

In some contexts, it is useful or even necessary to indicate explicitly that some events do *not* influence the FSM behaviour. One way of realising this is to introduce (runtime changeable) **don’t care** tags to some events, so that these are temporarily not considered in the event transition table.

Another relevant policy is that of **history** in the event processing: the fact a **particular sequence** of events and/or transitions has taken place can have the meaning of an event in itself. That is, the activity’s application might want to adapt the behaviour of the activity whenever such a sequence has occurred. One common example is that of a **livelock**: the system adapts its behaviour because of a particular event, but the adaptation in itself triggers a reaction in another event, whose change in state causes the first activity “to undo” its former reaction, *ad infinitum*.

Finally, it often makes sense to put a **timeout** on the duration between events and/or state transitions. For example, a robot navigating in a building can decide to take another route if it has been waiting “too long” in front of a path obstruction on its current route to disappear. An FSM with this behaviour is often referred to as a **timed automaton**.

2.10.10 Policy: selection, priority, deletion

At any moment in time, the `event_queue` of an FSM contains zero or more events that it is expected to react to. And the `event_processing` relations in an **event reaction table** are *declarative*. Hence, extra choices must be made to determine the actual execution behaviour of the event handling. For example:

- which internal events and transitions to include in the model.
- which callback functions to attach to which events, and to which activity. This is the trade-off between
 1. *communicating the event and keeping the computation local*. The **event** is not handled in the event loop of the Coordinating state machine, but communicated to the coordinated activity; the latter then computes the callback in its own computational context.
 2. *communicating the computation to a non-local context*. The **event** is handled in the event loop of the Coordinating state machine, so the computation of the callback takes place in the event loop of the coordination activity.

- how many events to take from the event queue in one single event condition evaluation step.
- priorities on the selection of events from the queue.
- the rules to decide when to remove which events from `event_queues`.

2.10.11 Implicit constraints on event handling meta model

The **constraints** below are (possible) assumptions to make in an FSM meta model. It is necessary to make these assumptions formally explicit, when one needs to subject an FSM model to **formal verification** (“*does the system implementation conform to its specifications?*”), and **validation** (“*does the system specifications conform to the application’s requirements?*”):

- a modelled behaviour is in one, and only one, state at each moment in time.
- an FSM represents the behavioural state of only one activity.
- every state and every transition has a unique ID.
- state transitions take no time.
- event processing takes no time.
- event firing takes no time.
- representation of all parts of the model takes no space in computer memory.
- event queue bookkeeping takes no time.
- the management of the event processing takes no time.

2.10.12 Runtime creation of FSMs via transition systems

Finite state machines are a good model to represent the discrete switching between behaviour, but any somewhat realistic application requires thousands of behavioural states, which compromises scalability. The obvious solution is to use **higher-order relations** that represent the links between “pre conditions” of “actions” and the models of finite state machines to realise those actions. The technical challenge is to develop solvers to do the *model-to-text* transformations that generate executable state machines.

(TODO: explain how the concept of **transition systems** applies to (i) the online generation of FSM, based on (ii) graph traversals over knowledge about the discrete behaviour capabilities of robot systems. Links with **action languages**, **fluents**, and *triple graph grammars* [32, 36].)

2.11 Flags, Petri Nets and FSMs: role in system design

The Petri net mechanism for the **coordination of multiple algorithms** is complementary to the mechanism of **Finite State Machines**, which model coordination of **behaviour of one single activity**. Both share some parts of their structural **abstract data type**, namely (i) “circles”, and (ii) “arrows” connecting “circles”. (This structural resemblance sometimes leads to confusion about which model to use for which purpose.) Like an **FSM model**, a Petri Net model is just an abstract data type, that contains the bookkeeping information needed to coordinate the execution dependencies of algorithms. But *neither* of them is an algorithm or an activity in itself. In other words, the algorithms and activities that use FSM and Petri Net models for their coordination must look up the *state* in the FSM model, and the **markings** of the Petri Net model, themselves, whenever it is an “appropriate time” to do so.

2.11.1 Complementary roles of events, and flags and states

Events have complementary roles and properties with respect to flags and states:

- events are *produced* (that is, created out of nothing), *consumed* (that is, deleted for ever), and *duplicated* for being sent over communication channels.
- flags and states are non-duplicatable, are created once, and are then read and written but not consumed, and remain in shared memory.
- a side-effect of the *change of the value* of a flag or state *can* be the *firing* of an event that encodes that change.
- vice versa, a side-effect of the *handling* of an event can be the change of a flag or a state.

2.11.2 FSM as boundary case of single-token Petri Net

The following observations are obvious but nevertheless important:²¹

- modelling a coordination with a Petri Net that is constrained to have only one input place per transition, gives it the equivalent semantics as a Finite State Machine.
- a coordination model of a Petri Net with multiple places per transition, and (hence) multiple tokens, can be transformed into an equivalent Finite State Machine as follows: (i) every composition of markings becomes one state in the FSM, and (ii) the FSM's event transition table contains the composition of the transition logic of all Petri Net transitions connected to the filled places.

The conclusion is that it is, technically speaking, not *necessary* to introduce the two complementary modelling concepts, because a model in one meta model can be transformed into to a model in the other meta model. Nevertheless, both meta models have their place to support *human developers* in coping with *coordination complexity*.

2.11.3 Policy: higher-order model for pre/per/post conditions

The transition rules of a the **FSM** or **Petri Net** coordination mechanisms are often **hard coded**, in order to speed up reaction **latency**. So, an event reaction table represents the *first-order* (“factual”) behaviour the coordination mechanism's transitions. But *to explain why* a transition takes place requires *higher-order* (“semantic”) relations. Indeed, designers typically have knowledge about the system that they can formalize in the form of the following **three types of constraint conditions**.²²

- **pre conditions**: to be satisfied before a transition should be *entered*, that is, to allow the activity to select the corresponding behaviour.
- **per conditions**: to be satisfied *during* the whole duration of the a *state's* behaviour.
- **post conditions**: to be satisfied before a state should be *left*.

These conditions are not events in themselves, but the violation of one of them is an indication, to the activity coordination, that the activity may have to switch to another behavioural state, or to set or clear algorithmic flags. In other words, these constraints encode the *when* and the *why* behind a switch.

²¹References to the literature are still missing!

²²In many systems, these conditions are not made explicit, which makes it difficult to assess semantic correctness of an activity's behaviour. Only the first condition is relevant to a transition in the Petri Net context; all three are relevant for Finite State Machines.

When the conditions are available to the system software at runtime, they serve as **assertions** to be monitored. That is, one can (i) *check* whether a computed transition is justified, and (ii) *explain why* it is, or it is not.

2.11.4 Application contexts with policy choices

The amount of computation and communication²³ involved in all the above-mentioned monitoring algorithms is often orders of magnitude higher than the computations required for the transitions in the FSMs and Petri Nets that they feed. So, the decision about where to deploy which monitors has a high impact on the overall system design:

- *Bad*: to mix the Computations for (i) the *Boolean logic* computations to decide about making transitions, and (ii) the *continuous* computations needed to *monitor* activities and to generate their events. The former are much easier to make deterministic in time than the latter.

Good: to put such a monitoring computation into another activity than that of the FSM's or the Petri Net, and interconnect them with an event communication. While at the same time designing in extra states and logic rules in the FSM/Petri Net that make their behaviour robust against the undeterministic computation and communication timing of the monitoring events.

For example, if another activity is needed to make a decision, based on the inputs of several other activities, the Petri Net should model *when* the other activities have provided their inputs to the decision making, and *when* the decision making activity has made a decision. The data about the decision itself (inputs as well as outputs) are to be found elsewhere in the architecture, in *data channels*.

- *Bad*: to add the “guard” (monitoring) computation to a transition, since that makes the transition take a non-deterministic time to be realised.

Good: to add no computations whatsoever to transitions, but only to states. In this way, the only time taken by a transition is the time needed to adapt the FSM data structure by changing the pointer to the `current_state`.

- *Bad*: use names for the states that reflect the *reason why* a state has been reached. For example, names appear like “configured”, “error”, “started”, etc. These name choices restrict composability by introducing implicit constraints and assumptions on the *transitions* that should be present in the system.

Good: use names that reflect *what behaviour* the activity is providing while being in a state.

- *Bad*: to use Petri Nets to solve *scheduling* problems; for example, making the decisions about the *order* in which to start actions.

2.11.5 Design similarities and differences

An activity can *be* in only one behavioural state, but can *have* multiple status flags at any given moment in time. For example, the readiness of the activity to be involved in various

²³For example, before deciding to accept a “contract offer” during a task coordination, the accepting activity may need to check whether, and under which conditions, it should commit itself. That checking can involve a lot of interactions with resources or other activities, and computations to estimate costs and returns.

inter-activity coordinations, such as communication or [data ownership transfer](#) protocols. The consequences for an activity with respect to its FSM and Petri Net models are:

- it is involved in only one FSM all the time, but can participate in zero or more Petri Nets part of the time.
 - an FSM can be defined within the scope of one activity; a Petri Net’s design scope is typically at system level, requiring “someone” to introduce the coordination model and the coordination activity.
 - it owns the FSM completely, and the transition decisions can be done completely synchronously with itself.
 - it commits itself to one or more Petri Nets, and the transition decisions are done by another activity.
 - in an FSM, the scope of the decision making is defined very simply, by the current state.
- So, the asynchronicity complexity of a Petri Net coordination can be much higher than that of an FSM coordination. In each of the Petri Nets, the scope of the decision making is, canonically, all of the transitions. However, the second-order coordination via the [inhibition/activation](#) place, and the third-order coordination via [time outs](#) or [heartbeats](#) reduce the scope of transitions to consider.²⁴ That scope influences the complexity (and hence the timing) of computing the transition tables. Often, a larger influence comes from the [monitoring algorithms](#) in all activities that have to compute the local conditions under which their activity must produce or consume a token or an event.

2.12 Dynamically linked data: discovery, session initialization

(TODO: signalling, session description, session initiation.)

²⁴TODO: more examples and motivations.

Chapter 3

Meta models for geometry

The focus of this Chapter is on **polygonal** geometrical models, of *points*, *lines*, *polygons*, and *shapes* and *bodies*, in one-, two- and three-dimensional spaces,¹ because they are the basis for

- any other “smoother” representation of geometry,² and
- any composition of the three top-level robotic “services”: *manipulation* by fingers in grippers, *reaching* by arms on torsos, and *navigation* by legs, wheels, wings or propellers on platforms.

Indeed, the simplest geometric models of all the mentioned entities are polygonal: “**points**”, “**polylines**”, “**polygons**”, “**stick figures**”, “**boxes**”, or polygons with “**spline control points**”. Also base maps of world representations are polygonal relations: “**areas**” with *semantic tag points* attached to them.

It is straightforward to extend the geometric model to a **geometric chain** model: the latter then adds **relative motion constraints** between geometrical entities. Such geometric chain relations form a list, a tree, or a graph; they come in static and time-dependent versions. A major example of a time-dependent geometric constraint between points is the *rigid body*: all of the points in the body remain at the same relative distance, at all times. A geometric chain relation becomes a **kinematic chain** relation, as soon as the motion constraint is due to a **joint** between rigid bodies, which is a special case of time-dependent geometric constraint in that (i) the constraint is *bidirectional* (one can “pull” and “push”) and (ii) its time-dependency can be represented with a finite number of “joint coordinates”. Revolute and prismatic joints are most common representatives, with just one joint coordinate. A kinematic chain becomes a **dynamic chain** relation, as soon as mechanical “behaviour” (inertia, damping, elasticity) is composed onto the geometrical entities and constraint relations.

Each of these extensions is modelled as a **composition** onto a polygonal base model, via “semantic tag” **attachment points**: each tag is rigidly constrained somewhere on the polygon, and it is a mandatory argument in the relation that represents the semantic properties of the extension.

¹Primarily the **Euclidean** space, of **mechanics**, but also the less constrained **affine** and **projective** spaces, whose concepts of **incidence**, **cross-ratio**, and **parallelism** are very relevant in robotics.

²For example, **NURBS**, or **solid mechanics**.

Coordinates are a primary example of a semantic relation between attachment points, being the link to **measurements** via sensors. The same set of geometric entities can have multiple coordinate relations: each of multiple measurements provides one such coordinate set; the **physical units** of two coordinate sets can be different; one set of coordinates can represent the **expected** or **desired** values; etc.

Robots move with respect to their local environment, and their sensors come with **uncertainty**, and provide only **partial information** about that environment. Hence, geometrical models must allow to represent worlds with **inconsistent**, **ambiguous** and/or **incomplete** geometrical relations.

3.1 Overview of meta meta models

This Chapter introduces meta models for the following common geometrical entities and relations:

- **point** in a **space**, as the axiomatic primitive of geometrical modelling.
- **line segment** (composition of two points), **polyline** (composition of line segments), and **polygon** (composition of a polyline with the constraint that its start and end points are identical).
- **attachment** as a **collection** relation on one or more of the above entities. One single point can in its own already serve as an attachment. The role of the attachment concept is to serve as an argument in knowledge relations, where it represents the symbolic access to (a set of) geometric entities.
- **(semantic) tag** as a **(symbolic) relation** in an application domain that has at least one attachment as an argument. Each semantic tag serves as a **type** for a relation in a knowledge graph.
- the semantic tag of **geometric chain** constraint, of a (partial) order of any of the geometric entities above.
- the semantic tag of **(polygonal) shape** as a geometric chain of polygons (or rather, a graph), with a **closure** constraint.
- the semantic tag of the **type** of the space, as one of **Euclidean**, **affine** or **projective**; the semantic tag of a second **type** of the space, its **dimensionality**, as **0D** (a “*point*”), **1D** (a “*curve*”, with a “*line*” as special case), **2D** (a “*surface*”, with a “*plane*” as special case), or **3D** (a “*volume*”).
- the semantic tags of **(relative) position**³ and **direction**, and their time derivatives of **velocity** and **acceleration**.
- the semantic tag of **motion**, as the composition of the semantic tags of position, velocity and acceleration. These three always come as a package, in an object’s *state* of motion.
- the semantic tag of a **motion constraint** on the relative position, direction, velocity or acceleration.
- **line** and **plane**, as **relative position constraints** between a possibly infinite number of points in a space. The generic term *space* will be used to denote any line, plane or space.
- the semantic tag of **rigid body**, as a **motion constraint** on points in a Euclidean space that imposes **constant relative distance**.

³Relative position is the simplest form of a **spatial relation**.

- the semantic tag of **frame**, as a particular instance of a rigid body with an **origin** and a number of **direction vectors**, equal to the dimensionality of the space. The principal use of a frame is as an argument in a **coordinate system**⁴ relation.
- the semantic tag of a **Cartesian** coordinate system, adding the constraints of **unit length** and **mutual orthogonality** to the direction vectors of a frame.
- the semantic tag of **coordinates**, as a relation that connects the relative motion of two geometric entities to numerical values, with each of these values having:
 - a **dimension** (e.g., *length*) and a **physical unit** (e.g., *meter*).
 - two geometric entities to make the interpretation of the numerical values unambiguous: (i) the “**observer**” geometric entity from whose point of view the relative motion is quantified, and (ii) the “frame” geometric entity (or **datum**) that serves as coordinate system.

So, in total four geometric primitives must be identified explicitly before the coordinate values of a geometric motion can be correctly interpreted. Often, one and the same geometric primitive plays the *role* of two or more of the required four primitives.⁵

The same relative motion of the same two entities can have multiple coordinate relations.

- the semantic tag of a **map**, as a collection of relations between geometrical entities, and entities and relations in a particular domain. A map comes with a **legend** of domain-specific semantic tags that can be found on the map, and with one or more **indexes** into the map’s “database”.
- the semantic tag of an **atlas**, as a collection of maps.
- the semantic tag of a **gazetteer**, as the composition of a map with domain-specific data about the entities in the map. The robotics context of this document uses the term **world model** for such a composition of geometric information with the perception and control **affordances** of robots.

A model *can* conform to multiple of these types at the same time; for example a *planar* (1) *polygon* (2) in the *3D* (3) *Euclidean space* (4). A model *must* conform to the composition of **one** of the types from **both** the dimensionality and the space categories. Not every combination of such types is semantically correct; for example, a rigid body constraint is a Euclidean relation, without meaning in affine or projective spaces.

The **geometric chain** is the most complex relation in this Chapter. The **composition** of rigid bodies with **joints** as the **motion constraints** between them, are the topic of the Chapter on **kinematic chains**. The complementary **composition** of rigid bodies with **electromechanical dynamics** as complementary motion constraints are already treated in this Chapter’s Sec. 4.2.

3.1.1 Metadata for coordinates

The numerical values in coordinates often come with extra semantic tags that describe how to interpret the numbers. For example:

- **provenance**: how were the values generated? That is, with which sensor, using which device and software, etc.?
- **coverage**: a specific part of provenance, **modelling** the *spatial* origin of the numbers.
- **resolution**: for example, of the **sensor**, the **time**, the **computer**, or the **map**.

⁴Also called a coordinate *reference* system.

⁵For example, as represented by the concepts of **allocentric** and **egocentric** motion, which complement that of motion with respect to an *absolute*, or **geocentric** reference.

- **accuracy and precision**: the former refers to closeness of the measurements to a specific value, the latter refers to the closeness of the measurements to each other.
- **tolerance**: a quantitative indication of acceptable, expected or requested precision.
- **version**: an indication of when changes to the coordinates were realised.
- **time series** and **trajectory**: models of the temporal order of a series of motion coordinate values.
- **encoding**: a model of the representation of the numerical values as bit patterns. For example, the **HDF5** or **Apache Arrow** formats.
- **data catalogue**: the data can be part of a larger catalogue or **data warehouse**.

3.1.2 semantic_IDs for geometry in 1D, 2D and 3D

Section 1.2.17 describes the generic foundations of semantic metadata, namely the usage of IDs as “symbolic pointers” between models and meta models. This Section adds geometry-specific specialisations, with the following suggestion for the meta meta model IDs (MMID):

$$\{ \text{"MMID"} : [\text{"Geometry"}, \text{"3D"}, \text{"Euclidean"}, \text{"PointPolylinePolygonal"}] \}, \quad (3.1)$$

with similar notations for other geometric entities and relations, that are relevant in 1D and 2D spaces. The *order* of the semantic tags in the MMID container "["...", ..., "...]" above has meaning: **Geometry** is the **top-level** meta meta model, and the 1D, 2D and 3D tags are more *specific* than **Geometry** but at the same time also more *generic* than the identification of the geometrical space of the described entities and relations as **Euclidean**, **Projective** or **Affine** (Sec. 3.1.3); the last tag identifies a the *most specific* geometric meta model, namely that of *Point-Polyline-Polygon* introduced in this Chapter.

All of these entities and relations belong to the realm of the *mathematics* meta meta model. This document sometimes uses *composite* MMID tags P(2), A(2), E(2) and P(3), A(3), and E(3) for, respectively, the projective, affine and Euclidean spaces in two and three dimensions; the composite MMID tag R(n) represents the real line in n dimensions. Further MMID tags are introduced in the Sections below.

The 1D meta model is trivial: it represents a **line**, a **circle**, or a **curve**, or any other geometric primitive that can be mapped, continuously, to (a segment of) the real line. The 2D meta model (Sec. 3.2.1) is the core meta model, representing a **manifold**⁶ of zero-dimensional (“0D”) **Points**, and with simple composition rules to build all other entities. The 3D meta model (Sec. 3.2.2) is presented separately, because:

- 2D geometry is sufficiently relevant in itself to merit its own meta model.
- *all* 2D semantics hold in 3D too, but not the other way around, so 3D geometry is a *composition* of the 2D meta model with some *extra* semantics. For example, the 2D concept of *area* keeps its meaning in 3D, but *volume* is a meaningless concept in 2D.

3.1.3 Geometric relations in projective, affine and Euclidean spaces

All **entities** and **relations** introduced in this Section have meaning in Euclidean, affine *and* projective spaces. The **point** is the fundamental entity. The composition of points into **lines**

⁶A *manifold* is **continuously mappable** on \mathbb{R}^n , with \mathbb{R} the *real line*. Hence, \mathbb{R}^n is always one of the meta meta models of any geometric meta model in this document.

brings extra topological **constraint** relations, namely **collinearity**, **are-equal**, **intersect** and **have-ratio**, which hold for *all* mathematical spaces referred to in this Section. The line relation is where the three mentioned spaces differ from each other, and there is a clear **hierarchy**: projective geometry has the smallest amount of relations; affine geometry conforms to all of them *and* extends them with its own (rather large) number of relations; and finally Euclidean geometry is on top of the hierarchy, with a couple of relations more. None of these spaces has a **natural origin**, because any point in the space serves equally well as reference. This implies that “position” can never be a *property* of the representation of a geometric *entity*, but only a property of a *relative position relation* between geometric primitives.

The **projective space** has as key relations (i) the **incidence** of points and lines, and (ii) the **cross-ratio** between sets of two points on *four* lines.

The **affine space** extends the projective space with the **relations** of (i) **parallelism** of lines, (ii) (ii) **barycentric** coordinates, and (iii) **convexity**, and (iv) **Pappus’ law** relation between two pairs of three points on *two* intersecting lines. The latter means that an affine description of points and lines keeps the relative order between them; for example, an affine geometric model that has a street between two other streets, keeps that street in the middle, no matter what **affine transformation** is applied to the model. Examples of affine transformations are **translation**, **scaling**, **similarity**, **reflection**, **rotation**, and **shear**.

The **Euclidean space** represents relative position and **translational motion** of points. Euclidean space has a **distance** relation, which is a **quadratic form** that maps two points to a real scalar. That means the terms **length** and **angle** have meaning. The relations **are-orthogonal** (of lines) and **triangle inequality** are consequences of the **distance** relation.

3.1.4 Non-Euclidean space for rigid bodies

When points and lines are being connected together to form “rigid bodies” in the Euclidean spaces $E(2)$ and $E(3)$, a new relation of **orientation** *emerges*. It is just the shorthand for the set of **constraint relations** between all points that their mutual distance is constant over time. All of the above-mentioned mathematical entities and relations remain meaningful, but extra semantics is introduced because of the dependency between *translation* and *orientation*: because of the constant-distance constraint, translating one point has an impact on the translation of the other points it is rigidly connected to. It turns out that the semantics introduced by the constant-distance constraint has nice mathematical (*Lie*) *group* properties, represented by the following two spaces:

- the **Special Orthogonal group and algebra** in two and three dimensions (**SO(2)**, **so(2)** and **SO(3)**, **so(3)**; MMID: `Geometry::SpecialOrthogonalGroup::2D/3D`) represent the semantics of “pure” orientations. This space has a well-defined “distance” metric between any two orientations (the smallest angle over which to reorient the first orientation to make it **equal-to** the second orientation. But there is no “unambiguous zero” rotation, because rotating a body over 360 degrees brings it back to its original orientation.
- the **Special Euclidean group and algebra** in two and three dimensions (**SE(2)**,

`se(2)` and `SE(3)`, `se(3)`; MMID: `Geometry::SpecialEuclideanGroup::2D/3D`) represent the semantics of the **coupling** between **translations and orientations**.

One of the major constraints on `SE(2)/SE(3)` is the **lack of a bi-invariant metric** relation [58, 67, 68]. For rigid bodies this means that:

- one must always introduce a **weight function** (that is, a new semantic relation) between translation and orientation.
- the “normal” Euclidean metric has no physical sense,
- “orthogonality” between velocities and forces has no physical sense either [67].

Examples of physically meaningful weight/metric functions in mechanical systems are **inertia** and **elasticity/stiffness**. They get that semantic status because their physical meaning leads to a metric that is “invariant” under any change of formal representation of the relations involved. More concretely, the kinetic energy of a moving body is independent of the physical units chosen to compute that energy, and independent of the reference frame in which one computes that energy.

3.1.5 Projections in geographical coordinate system

Geographic information systems work with geometries that are different than the ones in the previous sections: each of the many choices for a **geographical coordinate system** on the Earth comes with its own **map projection** mathematics. The more or less spherical shape of the Earth implies that even projective geometry is not sufficient, because the map projections transform **straight lines** in coordinate space to **curved lines** in map space, and the other way around.

3.1.6 Differential geometry

The concepts of **manifold**, **group**, **metric** are defined in the mathematical **meta meta models** of **differential geometry** and **group theory**. That higher-order knowledge provides useful (and already highly formalized) constraint information. For example, the relation that velocities satisfy all properties of the **tangent space** to the manifold of rigid body poses, accelerations live in the **second-order tangent space**, and forces live in the **co-tangent space**, [18, 41]. The latter means that forces can play the role of **linear forms** over the motion spaces, mapping position **displacements**, velocities and accelerations into real scalar numbers, with physical interpretation of energy. More in particular, **work**, **power** and “acceleration energy”⁷, respectively, [82, 85, 109].

3.1.7 Qualitative spatial relations

Geometrical information is not always numerical and/or directly observable. Many geometric relations are qualitative (and hence purely symbolical), such as:

- *the robot is in front of the table.*
- *move towards the door until it is within arm reach.*

⁷This is not a commonly used term in the English-language scientific literature. Gauss [42] introduced the concept with the German term “*Zwang*”.

- *hand over the bottle from right to left hand.*
- *there are two cars in front of this car.*
- *move to your left.*

The list of qualitative relations is long: `connects`, `contains`, `borders`, `intersects`, `on_top_of`, `left_of`, `behind`, `in_front_of`, `in_between`, `in_middle_of`, etc.

An important argument in these geometric relations is the **point of view**: does a relation hold from one own’s point of view, or from someone or something else’s point of view? For example, for one observer, an object can be `behind` another object, while it is `in_front_of` that object for another observer.

(TODO: qualitative **Spatio-temporal reasoning**, **spatial relations** between regions in space, e.g. **Region Connection Calculus**, **DE-9IM**, or *Cross Calculi* [15, 62].)

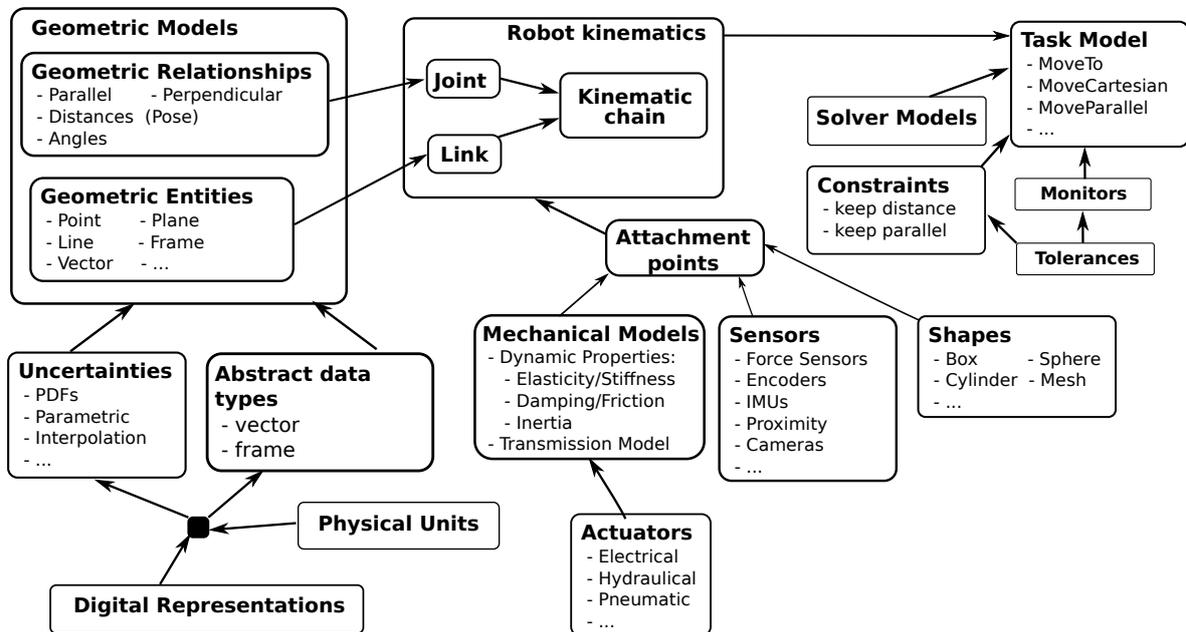


Figure 3.1: The mereo-topological overview of the “motion stack” in which a majority of the depicted blocks is a *structural* composition of geometrical entities and relations, or a “*higher-order*” composition on top of such a geometrical structure. Each arrow represents such a composition of complementary and separated *models*. The “black square” arrow represents an *n-ary* composition relation.

3.1.8 Taxonomy of geometric meta models

This document introduces the **partially ordered hierarchical structure** of Fig. 3.1 as a modelling axiom (a so-called **taxonomy**) for the (mostly geometric parts of) **motion**, **perception** and **world modelling “stacks”**. This partially ordered modelling structure serves as **structural backbone** of reasoning, querying and model transformations; each of the entries in itself is candidate to be (meta) modelled by at least one formal model, with **semantic metadata** linking between them. The lower levels of abstraction (with the lowest numbers in the descriptions below) represent the (almost) domain-independent aspects of mereology (the *set* of “things” in

the model) and topology (the *connections* between “things”). The gradually more “domain”-specific levels are motivated by specific sets of use cases in robotic world modelling, motion and perception. As far as the *geometry*⁸ parts of the taxonomy are concerned, this document introduces the following partially ordered set of levels of abstraction, using the *kinematic chain* as the running example to illustrate the kind of knowledge representation required at each level:

1. **Mereology**: this abstraction meta meta level is introduced in Sec. 1.4.1, and models the “whole” and its “parts”, with just **has-a** as relevant relation, and the resulting **collection** as relevant higher-order entity.

In the mereology of the geometrical meta model, a kinematic chain **has-a collection** of **links** and **joints**, and it **has-a workspace**, that is, a part of the physical world that it can occupy.

- 1.1 **Objects & 1.2 Manifolds**: the “wholes” and “parts” in a robotics context are further classified in (the top level of) a taxonomy with two complementary branches: “*discrete*” things (“objects”) and “*continuous*” things (the “**manifolds**” of the **configuration spaces** of the objects). One particular mereological entity or relation has typically discrete as well as continuous parts; for example, a kinematic chain has a continuous motion space, but also a discrete set of actuators and sensors; which by themselves again have continuous state spaces.

A kinematic chain **has-a** its whole-part relation with its **links** and its **joints** as **objects**, and its **workspace** as the **manifold** of all reachable positions of its parts.

2. **Topology** with its **Contains** and **Connects** relations. This abstraction level introduced in Sec. 1.4.2 models “neighbourhood”, more in particular the **contains** and **connects** relations, in, both, discrete and continuous spaces. There are also topological relations in non-geometric *taxonomies*, for example in all knowledge models where hierarchical **hypernym/hyponym** relations have been identified (not only for objects, but also for verbs).

A kinematic chain **connects** its **links** to its **joints**, forming the kinematic graph; most common topological instantiations are: *serial*, *tree* or *parallel*.

- 2.1 **Spatial topology**: in the *continuous* real-world space around us, the following relations specialise the **connects-contains** relations:

- **neighbourhood** relations like **near-to**, **left-of**, **on-top-of**, **inside-of**, etc.
- **tesselation** (or “**tiling**”) representations of spatial **coverage**.

Both are defined in the (two- as well as three-dimensional) Euclidean, affine, and projective spaces. Such spatial topological relations apply to kinematic chains relative to objects in the environment, for example, when its **end effector** comes **above** a table, or **inside-of** a box, or when a mobile platform covers contiguous areas in the world.

⁸The taxonomy includes **dynamics** too, which is covered by the mathematical theory of **differential geometry**.

2.2 **Object topology:** within the *continuous* real-world space around us, objects can be physically connected to each other, and their connects relations have **block**, **port**, **connector**, and **dock** parts.

Serial kinematic chains have “arm” and “hand” object topologies, which are more concrete (i.e., behaviour-rich) than a “manipulator”, which is more concrete than “actor”.

3. **Geometry:** this is the essential next level of modelling abstraction, for the *manifold* type of entities and relations, and a large taxonomy of geometrical meta models has been defined in the mathematics literature already, of which the **affine**, **projective** and **metric** versions are most relevant to robotics.

The context of a **Task** implies that a kinematic chain **has-a** lot of **points** attached to its **links**, and whose geometrical **position** and **velocity** in space are of interest in the task for which the kinematic chain is used.

3.1 **Affine geometry:** this introduces non-metric geometrical entities, such as **point**, **line**, and **hyperplane**, and relations such as **intersect**, **parallel**, and **ratio**.

Many serial kinematic chains have some **parallel** joint angle axes.

3.2 **Dimensionless (or “qualitative”) metric geometry:** many use cases do not need *absolute* distances or angles, but rather *relative* ones. In other words, the **absolute scale** of the geometrical entities is not used, but the *ratios* of lengths or angles (which are physically dimensionless) are the relevant information in a task specification. For example, when driving through an office corridor, the robot perception system can track the relative (changes in) areas and the directions of wall, door, ceiling or floor surfaces, without having to know their absolute sizes. Indeed, staying at the “center” of a corridor, driving towards its “end”, or moving twice as far as the nearest door, are all relative (or “qualitative”) motion specifications.

A serial kinematic chain often has the “zero configuration” of its joint angles in the **middle** of their physical motion range, which is geometrically well-defined without the need to use absolute numbers. Similarly, a specification to move a joint “away from” its mechanical limits is a meaningful and metrically dimensionless motion specification.

3.3 **Metric geometry:** as soon as one introduces a **metric** (or “distance function”), one can start talking about entities such as **rigid-body**, **shape**, **orientation**, **pose**, **angle**, and relations like **distance**, **orthogonal**, **displacement**,...

A kinematic chain transforms metric speeds at its **actuators** to metric spatial velocities of its **links**.

4. **Dynamics:** this modelling level brings in the interactions between “effort” and “flow” in the exchange and transformation of “energy”. For mechanical systems, that means force and motion; for electrical systems, that means current and voltage; etc. In general, these effort-flow relations are called **impedances**.

A kinematic chain transforms mechanical energy between (i) the motors attached to its **joints**, and (ii) its **links**. The latter can themselves transform mechanical energy with objects in the chain’s environment; for example, when pushing a box over a table.

4.0. **Differential geometry:** this is the domain-independent representation of physical systems, that is, all features that are shared between the mechanical domain, hydraulic domain, electrical domain, thermal domain, etc. The most fundamental concepts are: `Tangent_space`, `Linear_form`, `Vector_field`, and `metric`.

A kinematic chain transforms electrical energy at its `actuators` to mechanical energy at its `links`. Each of the latter's motion properties have the mathematical properties of the *Special Euclidean group* `SE(3)` and its *Lie algebra* `se(3)`.

4.1. **Mechanics:** there are just three fundamental types of mechanical interaction: `stiffness`, `damping` and `inertia` linking `force` to, respectively, `position`, `velocity` and `acceleration`. Other relevant entities and relations are: `mass`, `elasticity`, `gravity`, `momentum`, `potential-energy`, and `kinetic-energy`.

All of the above mechanical entities and relations are present in kinematic chains.

4.2. **Electro-magnetics:** the interactions between `current` and `voltage`, namely `resistor`, `inductance`, `capacitance`, `reluctance`, `back-emf`, `flux`,...

These entities and relations are present in a kinematic chain with electrical actuators.

Later Sections and Chapters will provide more detailed descriptions of many of the mathematical/physical concepts above, and (software) engineering extensions will be added, for example, to model data structures, coordinates and physical dimensions.

3.1.9 Levels of representation in geometry

Section 2.2 introduced a natural hierarchy in the composition of knowledge representations, and this Section applies this hierarchy to the context of formal meta models in geometry, *and* adds two other representation forms that are cutting across the generic hierarchy: *physical units* and *uncertainty*. These formal models apply to (a selection of) the geometric meta models described in Secs 3.1–3.1.8.

Meta models of geometric entities and their relations add semantic meaning (“information”) to the digital quantities (“data”) that they (or rather, their software instantiations) work with [14, 26], and these semantic relations have the **dependency structure** of Fig. 3.2. A summary of the relevant levels of representation is:

- **mathematical:** each geometric entity can be represented by (“*composed with*”) multiple mathematical models. For example, a `LineSegment` is the set of all `Points` that lie on the `Line` between a `start Point` and an `end Point`.

(This document does not pursue the mathematical modelling, because that is beyond its scope.)

- **abstract data type:** each of the above can be represented with multiple *abstract data type* models. For example, a `LineSegment` is an `array` with two members, with one member having a metadata tag of `start` and the other member a meta tag of `end`.
- **data structure:** each of the above must be represented in a concrete programming language with the available (instances of) *data structures*. For example, the `array` in `C` comes in the following forms of an ordered list (array), or an ordered list of ordered lists (matrix):

```

int cat[10]; // array of 10 elements, each of type int
int a[10][8]; // an array of 10 elements,
              // each of type 'array of 8 int elements'

```

- **digital storage:** each of the above must be represented with a particular digital representation model, to describe how the information is encoded in bits and bytes in the **RAM** and the **hard disks** of computers, and in the **messages** that computer processes exchange the information with. For example, each **int** in the array above is represented by four bytes, with the **little endianness** arrangement.
- **physical units:** *all* of the above must be composed with a model of the relevant *physical units*. For example, the above-mentioned **floats** are of the type **length** and have a **unit** of **meter**.
- **uncertainty:** *all* of the above must be composed with a model of the possible *uncertainty*. For example, a **Point**'s uncertainty can be represented by the **standard deviation** of the numerical values in its digital representation.

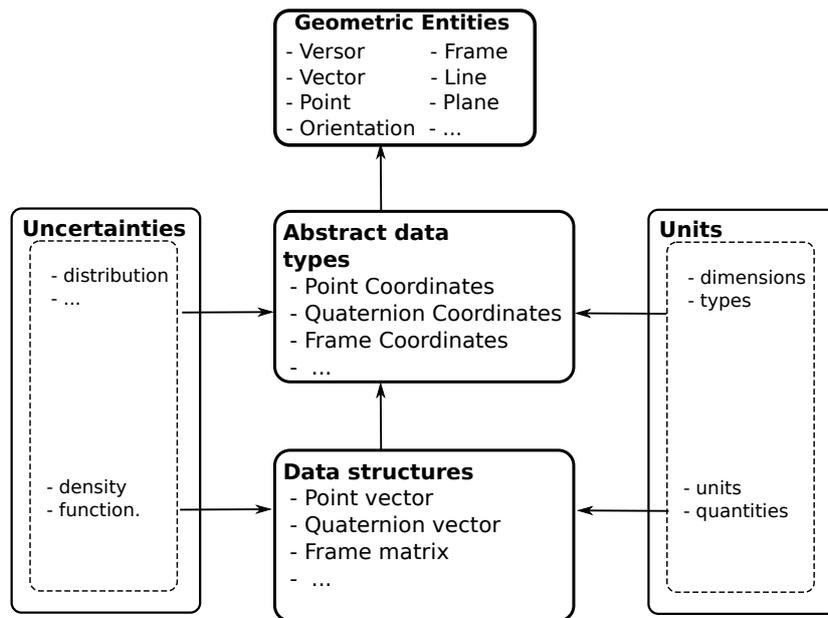


Figure 3.2: The mereo-topological model of the representations of geometric entities and relations, that is, the container entity for “mathematical” (i.e., *geometrical*), “abstract data type” and “data structure” representations. An arrow represents *composition* of complementary aspects in that numerical representation. Models of physical units *must* be composed always; models of uncertainties *can* be composed, when needed.

3.2 Meta models of Point–Polyline–Polygon geometry

This Section presents the meta models for the **geometry** that is relevant to cyber-physical system domains in which spatial information is essential; for example, robotics, avionics, manufacturing, or intelligent traffic systems. The major semantic concepts are (i) the **entities** of

points, **lines**, **line segments** and **polygons**, (ii) the **relation** of a **map** as an ordered or unordered **set of sets** of geometric entities, (iii) the **relations** of **instantaneous motion** of the entities, and (iv) the **rigid body** as a **constraint** on that instantaneous motion relation. The meta models are hierarchically structured into various levels of abstraction: (i) mereo-topological naming of entities and relations (Secs 3.2.1–3.2.4), (ii) abstract data types for symbolic model validation and model-to-model transformations (Sec. 3.2.6), and (iii) data structures to carry **coordinates** (Sec. 3.2.8). Each meta model on a “higher” level of abstraction can be represented by multiple meta models on a “lower” level. Hence, separating the definition of all these meta models helps **composability** of modelling efforts.

Modelling in this Section starts with the **Point** entity, and all other entities and relations are **compositions** of that **Point** model. (The **monospaced font style** is used in this Section to denote keywords in the meta models; for example, **Point**.) This model composability approach is motivated by:

- the desire to keep the number of “basic” *abstract data types* small.
- the desire to put entities, relations, and the constraints on those relations, into separate meta models, because there are many use cases for the separate meta models.
- the fact that most sensors in robotics *measure* points, and not lines, planes or bodies.
- the fact that for points the concept of *uncertainty* is uniquely defined, but not for lines, planes or bodies.
- the straightforward way to provide all possible kinds of “*attachments*” to compose into richer semantic structures.

Examples of the latter are *geometric chains* and *kinematic chains* (Chap. 4), or *maps* or *world models* (Chap. 6). The knowledge in the combination of this Section’s meta model for geometry and those for kinematic chains and “**OpenStreetMap**”-like world models, is sufficiently mature and complete to allow their formal representation to start a community-driven process towards a **vendor-neutral open standard**.⁹

3.2.1 Point entity and its composition relations

Point — A **Point** is the *zero-dimensional* element of the space **manifold**, so it has no properties of length, area, volume, or shape. None of this knowledge must be represented explicitly; just referring to the **mathematical meta meta models** suffices: does the **Point** live in a 2D or 3D space, and is this space Euclidean, affine or projective? In other words, the mathematical representation of a **Point** is just a **symbol**, to represent its **identity**. This document chooses the notation

$$\{ \text{Point} : \text{aPoint} \}, \tag{3.2}$$

to identify a model of a **Point**. With all **Semantic_ID information**, the full version of such a **Point relation** is depicted in Fig. 3.3. The following Sections will only use the short version.

⁹With clearly identified advantages with respect to the robotics *de facto* standard **URDF**: (i) all possible kinematic chains can be represented and not just tree structures with one-dimensional revolute or prismatic joints; (ii) any other meta model can be *composed* with the kinematic chain model without requiring that other meta model being visible or even known; and (iii) in particular, any type of actuator and control algorithm, including **multi-articular** configurations.

```

{
  [ { MMID: ["Geometry","E2"] },
    { MID: "Point" },
    { ID: "Point-E2-xy34s" }
    // a unique identity code
  ],
  { Arguments: [ {PointName: "aPoint"} ] }
}

```

Figure 3.3: Mereological model of a Point, with Semantic_ID metadata.

Vector — This **ordered** list of two Point entities adds three constraints: (i) the *ordering* that gives an **orientation** to the Vector, (ii) that list contains *exactly two* members (and not all the points on the line in between), and (iii) the two Points in the list are different. One of the Points gets the *attribute* of **start**, and the other that of **end**. These parameters are not a *property* but an *attribute*, because different contexts can give different **orientations** to the same Vector. The model is the composition of two Point models:

$$\{ \text{Vector} : [\{ \text{start} : \text{Point-E2-xy34s} \}, \{ \text{end} : \text{Point-E2-567j3} \}] \}. \quad (3.3)$$

The model above uses ID strings that refer to Point models as in Fig. 3.3. If desired, the shared MID and MMID information can be taken out of the Semantic_ID of each of the composite Points and put in the Semantic_ID of the Vector.

The following are Vectors with *constraints*:

- **Line_vector**: the **start** point is constrained to be *somewhere* on the Line through the **start** and **end** Points of the Vector.
- **Free_vector**: the set of Vectors starting in every point of the space, all forming parallelograms with every other Vector in the set. (Because of its dependency on the Parallel relation, this concept does not exist in projective space.)
- **Versor**, or **Direction_vector**: a **Free_vector** that represents just an **orientation**, that is, whose **length** has no meaning. Of course, the constraint remains that the **start** point is different from the **end** point.
- **Unit_vector**: a **Direction_vector** in the Euclidean space, whose **length** is constrained to be of *unit length*.

Their formal models are straightforward compositions of the Vector model, with the extra constraint relations.

Polyline — A polyline is an **ordered** set of Points. It is equivalent to an ordered list of Vectors, with the **constraints** that (i) the **start** point of one Vector is the **end** point of the previous Vector in the ordered set, (ii) each Point belongs to exactly two Vectors, except for the **start** point of the first Vector and the **end** point of the last Vector. The notation is:

$$\{ \text{Polyline} : [\text{aPoint-ID}, \text{bPoint-ID}, \dots, \text{zPoint-ID}] \}, \quad (3.4)$$

where aPoint-ID, bPoint-ID and zPoint-ID are the unique IDs of Points. **Length** is a scalar real-valued **property** of the Polyline, whose value is the sum of the **Lengths** of each

of the **Vectors**. The **orientation** of the **Polyline** is an **attribute**, that is also inherited by every **Vector** in the **Polyline**. The notation is

$$\{ \text{Polyline} : [\{\text{start} : \text{aPoint-ID}\}, \text{bPoint-ID}, \dots, \{\text{end} : \text{zPoint-ID}\}] \}. \quad (3.5)$$

Simplex (in a 2D space) is an **ordered** list of **two non-colinear Vectors**, with the **constraint** that both have the same **start** point. The notation is:

$$\{ \text{Simplex} : [\{\text{start} : \text{aPoint-ID}\}, \text{bPoint-ID}, \text{cPoint-ID}] \}. \quad (3.6)$$

The extension to a 3D space is obvious: just add one more **Point**, and adapt the **Semantic_ID** accordingly.

Orientation¹⁰ is the **attribute** of the ordering: the order *from* **bPoint-ID** to **cPoint-ID** being **positive** or **negative** is a *convention* given by the application context.

Frame is an entity that only has meaning in the *Euclidean* context, because it adds two **metric constraints**¹¹ to the **Simplex** entity: (i) the **length** of each **Line_segment** is the unity length, and (ii) the **Line_segments** are **perpendicular** (or **orthogonal**). The orientation of a **Frame** is typically an *essential* attribute, because of its use to represent **Coordinates** (Sec. 3.2.8), so the notation has somewhat different semantics than the **Simplex**:

$$\{ \text{Frame} : [\{\text{origin} : \text{aPoint-ID}\}, \{\text{end} : \text{bPoint-ID}\}, \{\text{end} : \text{cPoint-ID}\}] \}. \quad (3.7)$$

The extension to a 3D space is obvious: just add one more **Point**, and adapt the **Semantic_ID** accordingly.

The **Frame** is a spatial shape entity, but it is often just used for its attribute of **orientation**, in 2D and 3D spaces. (The **Simplex** too, for that matter.) Some common *policies* to represent **orientation** are:

- explicit ordering of the **Frame's Vectors**, via **semantic tags** that reflect *numerical* order (1, 2, and 3), *alphabetic* order (*X*, *Y* and *Z*), or *color coding* order (*R*, *G* and *B*, after the deeply established **RGB** color model).
- the binary **orientability** choice between **Right_handed** or **Left_handed**.

The notation for a **Frame-based Orientation** is:

$$\{ \text{Orientation} : \text{Frame}_a \}. \quad (3.8)$$

Polygon — This is a **Polyline** with the extra **constraint relation** that the two not yet connected **start** and **end** **Points** must now coincide. The notation (for an oriented **Polygon**) is:

$$\{ \text{Polygon} : [\{\text{start} : \text{aPoint-ID}\}, \text{bPoint-ID}, \dots, \text{zPoint-ID}, \{\text{end} : \text{aPoint-ID}\}] \}. \quad (3.9)$$

¹⁰**Right_handed** and **Left_handed** are often used synonyms.

¹¹These constraints are the same in 2D and 3D.

The **orientation attribute** is inherited by all **Polylines** inside the **Polygon**. The interior of the **Polygon** is represented by an extra **attribute**, namely one single **Point** outside the **Polygon** lines:

$$\{ \text{Polygon} : [\{\text{start} : \text{aPoint-ID}\}, \text{bPoint-ID}, \dots, \text{zPoint-ID}, \{\text{end} : \text{aPoint-ID}\}], \quad (3.10)$$

$$\quad \{\text{interior} : \text{iPoint-ID}\}$$

$$\}.$$

In the Euclidean context, **area** is a **property** of the polygon.

Polygon_pair — The **unordered** set of (i) a set of two **Polygons**, and (ii) a **Point** that has the *attribute* interior. The notation is:

$$\{ \text{Polygon_pair} : [\text{P1} : [\{\text{start} : \text{aPoint}\}, \text{bPoint}, \dots, \{\text{end} : \text{aPoint}\}], \quad (3.11)$$

$$\quad \text{P2} : [\{\text{start} : \text{APoint}\}, \text{BPoint}, \dots, \{\text{end} : \text{APoint}\}] \quad (3.12)$$

$$\quad], \quad (3.13)$$

$$\quad \{\text{interior} : \text{iPoint}\} \quad (3.14)$$

$$\}]. \quad (3.15)$$

In the Euclidean context, **area** is a **property** of a **Polygon_pair**, namely of its interior part.

Multi_Polygon — An **unordered** set of **Polygon_pairs**. In the Euclidean context, **area** is a **property** of the **Multi_Polygon**, as the sum of the areas of each **Polygon_pair**.

3.2.2 Extra composition relations in 3D

All of the 2D entities keep their meaning in 3D too. This Section introduces the extra 3D-only entities.

Polyhedron — This is the generalisation of the **Polygon** to a 3D shape with a surface that has no holes. That is, it consists of a set of **Polygons**, which all mutually share some **Polylines**. The notation is illustrated with the simple example of the **Polyhedron** in Fig. 3.4:

$$\{ \text{Polyhedron} : [\text{Po11} : [\{\text{start} : \text{Pnt2-ID}\}, \text{Pnt3-ID}, \text{Pnt4-ID}, \{\text{end} : \text{Pnt2-ID}\}], \quad (3.16)$$

$$\quad \text{Po12} : [\{\text{start} : \text{Pnt1-ID}\}, \text{Pnt4-ID}, \text{Pnt3-ID}, \{\text{end} : \text{Pnt1-ID}\}], \quad (3.17)$$

$$\quad \text{Po13} : [\{\text{start} : \text{Pnt1-ID}\}, \text{Pnt3-ID}, \text{Pnt2-ID}, \{\text{end} : \text{Pnt1-ID}\}], \quad (3.18)$$

$$\quad \text{Po14} : [\{\text{start} : \text{Pnt1-ID}\}, \text{Pnt4-ID}, \text{Pnt2-ID}, \{\text{end} : \text{Pnt1-ID}\}] \quad (3.19)$$

$$\quad] \}. \quad (3.20)$$

The model above could have been composed differently: some of the four **Polygons** could

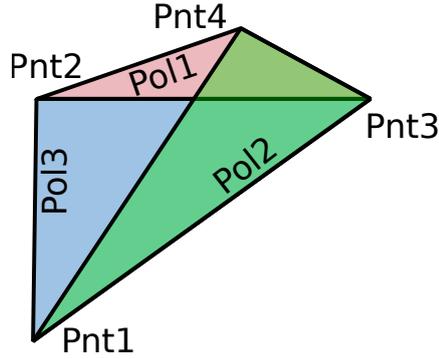


Figure 3.4: A Polyhedron as the composition of four Polygons, each the composition of three Points. (The fourth polygon is not shown in colour, in order not to overload the drawing.)

have their own model already, so that the Polyhedron can refer to them. For example:

$$\text{Pol1} : [\{\text{start} : \text{Pnt2-ID}\}, \text{Pnt3-ID}, \text{Pnt4-ID}, \{\text{end} : \text{Pnt2-ID}\}], \quad (3.21)$$

$$\text{Pol2} : [\{\text{start} : \text{Pnt1-ID}\}, \text{Pnt4-ID}, \text{Pnt3-ID}, \{\text{end} : \text{Pnt1-ID}\}], \quad (3.22)$$

$$\text{Pol3} : [\{\text{start} : \text{Pnt1-ID}\}, \text{Pnt3-ID}, \text{Pnt2-ID}, \{\text{end} : \text{Pnt1-ID}\}], \quad (3.23)$$

$$\{ \text{Polyhedron} : [\text{Pol1-ID}, \text{Pol2-ID}, \text{Pol3-ID}, \quad (3.24)$$

$$\text{Pol4} : [\{\text{start} : \text{Pnt1-ID}\}, \text{Pnt4-ID}, \text{Pnt2-ID}, \{\text{end} : \text{Pnt1-ID}\}] \quad (3.25)$$

$$] \}. \quad (3.26)$$

Volume remains a valid *property*, as is **orientation**.

Simplex, Frame — The 3D versions differ from the 2D versions in that (i) there are **three Vectors** in the Simplex or Frame, and (ii) their mutual constraint is on being **non-coplanar**.

Plane — A **plane** is the third axiomatic primitive of geometry (next to **Point** and **Line**), that represents a *surface* in the 3D space, or, in other terms, a two-dimensional *subspace*. Different ways to represent a **Plane** are:

- the *join* of three Points.
- the *join* of two intersecting Lines.
- these two Lines can come from two Vectors in a Simplex.

All three ways are equivalent in their composition of Points in somewhat different ways. A fourth Point, or a third Vector, can be used to determine the **orientation** of the Plane.

Half_space — This contains all the points in space which lie on one side of the *supporting* Plane. The defining Simplex of the Plane defines the **orientation** attribute of the Half_space. A common alternative orientation representation is to give one single Point outside the Plane.

3.2.3 Position and Motion relations of a Point

This Section extends the semantics of the **Point** entity and its mereological composition entities, with the relation of the **Motion** between two such entities. A **Motion** of entities is always **relative** with respect to each other, or relative with respect to themselves. Hence,

Motion is not a *property* or *attribute* of one single entity by itself, but it is the property of the *relation* between two entities. Hence, the same geometric entity can have several **Positions** and **Motions** at the same time: one for each other entity involved in a **Position** or **Motion** relation.

Motion has three parts: the **Position** relation, and the relations of **Velocity** and **Acceleration** that represent the first- and second-order variations over time of the **Position** relation. All geometric entities are compositions of **Points**, so it suffices, strictly speaking, to model the **Motion** of a **Point**. **Motion** has different interpretations, as a mapping over **time** and/or over **space**:

- **time mapping**: the **time derivative** of the **Position** of one particular entity gives its **Velocity**, and the time derivative of its **Velocity** gives its **Acceleration**.
- **spatial mapping**: two entities of the same type are a **Displacement** away from each other, irrespective of where they are, or whether or not they are moving with respect to each other. So, the **Motion** maps the first entity onto the second one.

Motion has **passive** and **active** semantics:

- **passive**: the relation between *two* moving entities has a set of properties that represent their *mutual* **Position**, **Velocity** and **Acceleration**.
- **active**: a **Displacement** can be interpreted as an **action** to move *one* particular entity from one **Position** to another **Position**.

The following paragraphs introduce the *notations* and *terminology* used in this document to represent **Motion**.

Position and **Displacement** — These are the relations between two **Points** that represents where they are with respect to each other in space. Both relations are mathematically equivalent, represented by a **Vector**, because this is already a geometric relation between a **start** point and an **end** point. The nomenclature **Position** and **Displacement** is used to indicate the following semantic interpretations of the **Vector**:

- **Position**: the two **Points** in the **Vector** are two **Points** with a different identity; the **Vector**'s geometric interpretation is that of the *instantaneous* relative position of these two **Points**.
- **Displacement**: the two **Points** in the **Vector** are twice the *same* **Point** (that is, with the same *identity*) but at different moments in time. The time scale is not represented explicitly, because it is not considered relevant. Only the *order* over time is relevant.

The notation of the **Position** of **Point** “ePoint” with respect to **Point** “fPoint” is an obvious extension of that of a **Vector**, with the tags **start** and **end** replaced by contextually more meaningful synonyms:

$$\{ \text{Position} : \{ \text{of} : \text{ePoint-ID} \}, \{ \text{with_respect_to} : \text{fPoint-ID} \} \}, \quad (3.27)$$

where **ePoint** and **fPoint** are two **Point** entities. Hence, a **Position** has a **direction** property, equivalent to the **orientation** property of the equivalent **Vector**. There exists an

(obvious) inverse relation:¹²

$$\{ \text{Position} : [\{\text{of} : \text{fPoint-ID}\}, \{\text{with_respect_to} : \text{ePoint}\}] \}, \quad (3.28)$$

that is, the position of point `fPoint` *from* point `ePoint`. The notation of the `Displacement` is a very simple extension to the `Vector`:

$$\{ \text{Displament} : \{ \begin{array}{l} \{ \text{Point} : \text{Point-ID} \}, \\ [\{\text{start} : \text{sPoint-ID}\}, \{\text{end} : \text{ePoint-ID}\}], \\ \}. \end{array} \} \}. \quad (3.29)$$

`Velocity`, `Acceleration` — These are the first and second time derivatives of the `Position` relation. The `Velocity` of Point “`ePoint`” with respect to Point “`fPoint`” is denoted as:

$$\{ \text{Velocity} : [\{\text{of} : \text{ePoint}\}, \{\text{with_respect_to} : \text{fPoint}\}] \}, \quad (3.30)$$

The velocity of a point with respect to itself is a physically valid relation.

3.2.4 Position and Motion relations of a `Rigid_body`

The `Motion` relation of a `Rigid_body` in Euclidean space is the set of the `Motions` of all of the `Points` in the `Rigid_body`. However, there is the `constraint` that the `Motion` preserves the `distance` between all these `Points`, and, *hence*, also `length`, `angle`, `area` and `volume`. Mathematicians have realised that this constraint allows to represent the above-mentioned possibly infinite set of `Motions` of all `Points` on the `Rigid_body`, to a finite-dimensional mathematical group structure, irrespective of how many points are considered in the `Rigid_body`. That group structure has **three dimensions** in `E(2)` and **six dimensions** in `E(3)`. For the `Position` part of a `Motion`, these groups got the names of, respectively, `SE(2)` and `SE(3)`, the **special Euclidean groups** of the `Displacements` of a `Rigid_body`. For the `Velocity` part, the dimensionalities of the groups are the same as for `Displacements`, and they are denoted by, respectively, `se(2)` and `se(3)`, the special Euclidean *algebras*; “algebra” indeed, because there is not just the *addition* operation in the group of `Velocities`, but also a *multiplication* operation, the so-called **Lie operator**. In all mentioned three-, respectively six-dimensional, spaces, any `Motion` relation can be separated into:

- two, respectively three, **translational** degrees of freedom of **one Point** that is chosen arbitrarily on the rigid body.
- one, respectively three, **rotational** degrees of freedom, independent of whatever choice of `Points` or `Frames` on the rigid body.

This observation was already made in the 19th century, by the French mathematician Michel Chasles (1793–1881), [20], who formulated¹³ the following theorem: the most general `Displacement` for a rigid body is a **screw motion**, [7, 9, 17, 46], i.e., there exists a line in space

¹²In other words, both `Position` relations are constrained by a higher-order relation, that of being each others’ inverse relations.

¹³The notion of twist axis was probably already discovered many years before Chasles (the earliest reference seems to be the Italian Giulio Mozzi (1763), [19, 43, 73]) but he normally gets the credit.

(called the **screw axis**, [9, 54, 84], or “twist axis”) such that the body’s motion is a rotation about the screw axis plus a translation along it. This theorem (and hence the screw axis relation) holds for *Velocities* too.

Although a rigid body is a composition of points, the *Euclidean metric* for points has no equivalent metric on the space of rigid bodies, [58, 67, 68]; in other words, the **distance** between two rigid bodies, the **length** of a rigid body **Displacement**, and the **magnitude** of a rigid body **Velocity**, are not well-defined properties of the **Motion**: these values change when one changes the **Point** on the **Rigid_body** that is used as reference in the **Displacement** or **Velocity** parts of the representations.

The following paragraphs summarize the notations and relations of the **Motion** of a **Rigid_body**.

Position — A **Rigid_body** (or **Simplex**, **Orientation** or **Frame**) is a geometric entity for which the above-mentioned **Rigid_body** constraint holds. As for **Points**, the representation of the **Position** of a **Rigid_body** or a **Frame**, is always relative to another **Rigid_body** or a **Frame**:

$$\{ \text{Position} : [\{\text{of} : \text{aBody-ID}\}, \{\text{with_respect_to} : \text{bBody-ID}\}] \}, \quad (3.31)$$

that is, the position of rigid body **aBody** with respect to rigid body **bBody**. Similarly for other combinations of **Rigid_body** or **Frame**, such as:

$$\{ \text{Position} : [\{\text{of} : \text{aBody-ID}\}, \{\text{with_respect_to} : \text{fFrame-ID}\}] \}, \quad (3.32)$$

and

$$\{ \text{Position} : [\{\text{of} : \text{fFrame-ID}\}, \{\text{with_respect_to} : \text{gFrame-ID}\}] \}. \quad (3.33)$$

Orientation — The above-mentioned **Position** between two rigid bodies **A** and **B** specifies *only* the three **translational** relative degrees of freedom between both bodies. The missing three degrees of freedom represent the relative **orientation** of the two rigid bodies. The notation is:

$$\{ \text{Orientation} : [\{\text{of} : \{\text{Rigid_body} : \text{aBody-ID}\}\}, \{\text{with_respect_to} : \{\text{Rigid_body} : \text{bBody-ID}\}\}] \}. \quad (3.34)$$

Pose — This is the name of the *composite* relation of the **Position** and **Orientation** of a **Rigid_body**. That composition adds no extra semantics, it’s just the *set* of both composing relations. The notation is symbolically identical to the **Position** and **Orientation** ones:

$$\{ \text{Pose} : [\{\text{of} : \{\text{Rigid_body} : \text{aBody-ID}\}\}, \{\text{with_respect_to} : \{\text{Rigid_body} : \text{bBody-ID}\}\}] \}. \quad (3.35)$$

Linear_velocity — The time derivatives of a **Position** of a **Rigid_body**. Again, the notation is symbolically identical to the ones above. The **Linear_velocity** is a **Line_vector** because it is constrained to point through the **Point** on the **Rigid_body** whose change in **Position**

is represented. That choice of `Point` is arbitrary. Hence, and in general, the same `Motion` of a `Rigid_body` can have an infinite amount of `Linear_velocity` attributes. The difference must be made semantically clear by adding an extra `semantic tag` to the `Linear_velocity` model, namely the ID of the `velocity_reference_point` that is chosen in the model.

`Angular_velocity` — The time derivative of the `Orientation` of a `Rigid_body`. Again, the notation is symbolically identical to the ones above. The `Angular_velocity` is a `Free_vector` since it does not depend on any choice of `Point` on the moving `Rigid_body`.

`Velocity of a Rigid_body` — The time derivative of the `Pose` of a `Rigid_body`. There are a large number of possible and equivalent representations. For example, the `Velocity` of at least three `Points` on the `Rigid_body`.

`Twist` — This is a very popular choice to represent the `Velocity` of a `Rigid_body`, due to the fact that it requires only the minimum possible number of independent variables to represent a motion, namely three in 2D and six in 3D. A `Twist` is indeed the combination of a `Linear_velocity` vector and an `Angular_velocity` vector. (In a 2D space, the latter reduces to the choice of a scalar.)

Similar formal models hold for the second-order time derivatives, that is, the time derivative of a `Twist`: `Linear_acceleration`, `Angular_acceleration`, `Acceleration` (or `Acceleration_twist`, or `Rigid_body_acceleration`). There are two different types of time derivative of a `Twist`, hence introducing the need for two extra *attributes*:

- **Eulerian** (or **ordinary**) time derivative: one looks at the matter that flows under one particular fixed point in space, and reports on the change of velocity observed at that place over time. So, the `Acceleration` is the difference between the `Velocities` of *two different* particles, at the *same place* in space but at *different instants in time*.
- **Lagrangian** (or **material**) time derivative: one follows one particular point fixed on a rigid body, reports on the change of that body-fixed point's velocity over time. So, the `Acceleration` is the difference between the `Velocities` of the *same particle* at *two different instances in time*, and hence also at *two different places* in space.

The difference between Eulerian and Lagrangian time derivatives is only relevant for `Accelerations`; to derive `Velocity` from changes in `Pose`, the Eulerian time derivative and the Lagrangian time derivative give the same results. Both derivatives also give the same result for `Acceleration` from *rest*, i.e., from zero initial `Velocity`. Different domains use these two different definitions, most often without explicit information, which results in **non-composability problems in robotic systems** that must integrate different domain choices.

3.2.5 Constraint relations on mereo-topological entities

The previous Sections provide *imperative* models: the geometric entity or relation is constructed by a “recipe”. This Section describes *declarative* ways to model geometric entities and relations, that is, the latter are describe by a set of *constraints*, and a *constraint solver* is needed to find or check the entity or relation. The constraint relations make use of only the mereo-topological semantics; constraint relations on numerical `Coordinates` values are introduces in the [Section on geometric chains](#).

Examples of such mereo-topological relations are depicted in Fig. 3.6: a distance between a Point $pPoint-ID_2$ and a Line $Line_1$ can be interpreted as the *shortest* distance between $pPoint-ID_2$ and another point lying on the Line, or as the length of the projection of $pPoint-ID_2$ on the Line. Table 3.4 and Table 3.5 resume the different interpretations of linear distances and angular distances.

Intersection (E, A, P):

Cross_ratio (E, A, P):

Parallel (E, A):

Orthogonal (E):

Projection (E, A):

$$\{ \text{Projection} : [\{ \text{from} : \text{AsSeenBy_entity} \}, \{ \text{to} : \text{FromEntity} \}, \{ \text{result} : ?\text{projection} \}] \}. \quad (3.36)$$

Distance, Length (E): In the Euclidean context, length of a Vector or Line segment (or the distance between the two Points in these entities) is a *property* of the Line_segment, with a *real scalar value* of physical dimension *length*.

Angle (E):

A Vector_field is a *set* of one Vector in each Point in (a subset of) space.

Line_segment — An **unordered** set of **two** Points is enough to represent *all* the points on the Line that runs through the two Points in the set; it is also enough to represent all the points *between* the two Points in the set, and that geometric entity is called a Line_segment. The notation is

$$\{ \text{Line_segment} : [\text{aPoint}, \text{bPoint}] \}, \quad (3.37)$$

where a and bPoint are Points.

Line: is an entity that has a *axiomatic* meaning, in the Euclidean, *projective* and *affine* contexts; that means that one cannot define it formally and completely from more primitive concepts. So, this document *represents* (not “defines”...) a Line as the *join* of two Points, that is, the linear combination of the **start** and **end** points in a Line_segment.

Half_line — The positive or negative part of a Line, derived from the defining Line-segment.

Half_space — This contains all the points in space which lie on one side of the *supporting* Line. The defining Vector of the Line defines the **orientation** attribute of the Half_space. A common alternative **orientation** representation is to give one single Point outside the

Line, to represent the interior of the Half_space.

(TODO: distance between Lines, [55]?; harmonize symbols of points and lines with previous subsections in this Chapter.)

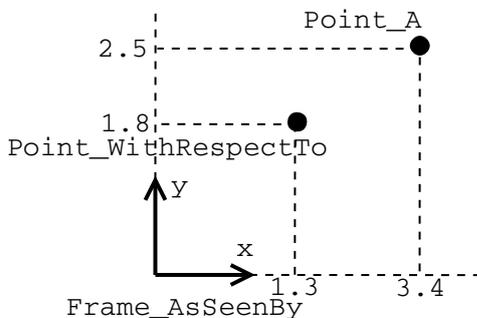


Figure 3.5: The three geometrical entities involved in a generic `Coordinates` abstract data type (in a 2D case): the Point (“Point_A”) whose coordinates are being represented; the entity `with_respect_to` which the `Position` coordinates are represented (in this case, this is also a Point, namely “Point_WithRespectTo”); and the numerical values of the coordinates are `as_seen_by` a reference Frame, with name “Frame_AsSeenBy”. The coordinate values are `x:1.1` and `y:1.3`.

3.2.6 Abstract data types for Coordinates of Position and Motion

An **abstract data type** (or **coordinate model**, or **coordinate representation**) adds (i) **numerical values** to the mereo-logical models of the previous Sections, and (ii) the **semantic tags** to identify the correct interpretation of those numbers. Recall that for each geometric entity and relation, there exist multiple equivalent possibilities to represent its `Motion`, and for each of these possibilities one can make multiple choices for coordinate model. In general, such choices already exist for each of the **three** essential semantic parts of a `Coordinates` model for a Point (Fig. 3.5 and Table 3.1):

- the identity (“Point_A”) of the Point whose `Coordinates of Motion` (i.e., `Position` or `Velocity`) are modelled.
- the identity (“Point_WithRespectTo”) of the Point **with respect to** which the first Point’s `Motion` is modelled.
- the identity (“Frame_AsSeenBy”) of the Frame in which the `Motion Coordinates` numerical values are computed. In other words, these numbers represent the `Motion as seen from` the Frame. They are computed as the `Projection of the Motion’s Vector` on the axes of the Frame.

For the specific case in which one represents the `Motion` of a `Rigid_body` by a `Twist`, two **extra semantic tags** are needed (Table 3.2):

- the **screw tag**, identifying the `Linear_velocity` and `Angular_velocity` vectors in the `Twist` model.
- the identity of the Point that serves as **velocity reference point** of the `Linear_velocity` part of the `Twist`.

Some common choices of velocity reference point are:

- a point on the entity whose `Motion` is represented.

```

{ Coordinate_Point_Entity_Frame_Array_Meter :
  { {
      relation: Position },
    {
      of: { Point: a } },
    { with_respect_to: { Entity: e } },
    {
      as_seen_by: { Frame: F } },
    {
      coordinates: [ {F.x: 1.2}, {F.y: -0,1}, {F.z: 3.2} ] },
    {
      units: meter },
    {
      ID: Coordinate143sGa },
    {
      MID: Coordinate_Data_Structure },
    {
      MMID: { Coordinates, Euclidean_space,
              Point, Entity, Frame, QUDT } }
  }
}

```

Table 3.1: Example of Coordinates model, in this case the Position of a Point.

```

{ Coordinate_Twist_Frame_Frame_Array_Meter :
  { {
      relation: Twist },
    {
      screw: [ linear_velocity: v], { angular_velocity: w} ] },
    {
      of: { Frame: b } },
    {
      with_respect_to: { Frame: r } },
    {
      as_seen_by: { Frame: F } },
    { velocity_reference_point: as_seen_by_frame_origin },
    {
      coordinates: [ {v: [ {F.x: 1.2}, {F.y: -0,1}, {F.z: 3.2} ] },
                    {w: [ {F.x: 0.2}, {F.y: 0.8}, {F.z: -2.2} ] } ] },
    {
      units: meter, seconds },
    {
      ID: Coordinatehd4if7 },
    {
      MID: Coordinate_Data_Structure },
    {
      MMID: { Coordinates, Euclidean_space,
              Twist, Frame, QUDT } }
  }
}

```

Table 3.2: Example of a Coordinates model of the Twist of a Frame "b" with respect to a Frame "r". The screw representation has a Linear_velocity and an Angular_velocity part. For the Linear_velocity and Twist of a RigidBody, the extra semantic tag velocity_reference_point is needed; its value is one out of an enumerated set of possibilities: of_point, on_screw_axis, as_seen_by_frame_origin, with_respect_to_point, etc.

- a point on the **with respect to** entity.
- the origin of the **as seen from** Frame.
- a point on the **screw axis**.
- a point that is **relevant** for the robot's current Task.

Geometrical relation	Abstract data type
Position	Position vector
Orientation	Euler-axis angle array Rotation matrix Euler angles array Roll-Pitch-Yaw angles array Quaternion array
Pose	Homogeneous transformation matrix Finite displacement twist array Screw axis array
Linear_velocity	Linear velocity vector
Angular_velocity	Angular velocity vector Rotation matrix time derivative Euler angle rate array RPY angle rate array Quaternion rate array
Twist Rigid_body velocity	Homogeneous transformation matrix time derivative six-dimensional twist array Pose twist array Screw twist array Body-fixed twist array Instantaneous screw axis array

Table 3.3: Commonly used abstract data type representations for geometrical relations. (TODO: update and complete.)

Because (almost) all geometric entities and relations are compositions of the `Point` entity, defining the [abstract data types](#) for all these entities and relations is rather straightforward, using the approaches of higher-order relations (Sec. 1.2.5) and `semantic_ID` metadata (Sec. 1.2.17), respectively; Table 3.1 shows examples. So, the complexity of modelling abstract data types does not come in the first place from the inherent complexity of the models, but from the large number of *different but equivalent choices* that are commonly used in practice. This *freedom of choice* is often the source of incompatibility, and of lack of composability, of models and implementations created by independent development teams, for the simple reason that not all choices are made explicit, and are not available in formalized form via which system software can check semantic compatibility at runtime, and introduce the appropriate model transformation when an incompatibility is identified. Table 3.3 summarizes the most common choices; the arguments in each composition come from the following list:

- the `name` of the composition relation (from Table 3.3). For example, `Position`, or `Twist`.
- `semantic_ID` metadata. This contains entries like `Geometry`, `E(3)`, and `QUDT`.
- the list of the *arguments* in the relation, each representing a composed entity in the relation. This includes the three or four [semantically tagged](#) entities, for example a `Frame`, that serve as *references* for, both, the `with_respect_to` and `as_seen_from` fields. Each argument can be a composition relation in itself, with its own `semantic_ID`. A common practice is “to raise” metadata that is the same in different arguments, to the highest level of composition in the relation; for example, it makes sense to use the same physical dimensions for all quantities in a composite model.

- the key-value pairs to represent the numerical values for each of the entities. For example, for the **Position** of a **Point** that means a set of *named* scalars values on the *real line*:

```
{ [ x: { type = "real" }, y: { type = real" } ] },
```

in 2D spaces, and a triplet

```
{ [ x: { type = "real" }, y: { type = real" }, z: { type = real" } ] },
```

in 3D spaces. The symbol "x" in the **Position** relation must refer to the argument of the reference with the same name. That is why its semantics is that of a “symbolic pointer”, because in the context of abstract data types, “to represent” just means “to be able to refer to symbolically”. For example, an application can already check formally whether the "x", "y" and "z" fields in a model indeed all have the correct type of "real". It can also add extra constraints, for example, a *range limit* on the "z" field.

No *concrete numerical* values are given in the representation (yet), just their symbolic type names. The numbers get filled in later, in the next level of composition, namely that of the **Coordinates data structures**.

- a set of key-value pairs that describe the *properties* of the composition relation. (Or, equivalently, the *attributes* that the composition relation gives to the entities it composes.) For example, the **start** and **end** tags for the two points in a **Line_segment**. And the **physical dimension** of the numbers: **length**, **length/time**, **angle**,...

No *physical units* are encoded, yet, just their symbolic names. Physical units, such as **meter** or **radian**, are filled in later, as *attributes* in the above-mentioned concrete **Coordinates data structures**.

There is seldom a unique abstract data type representation for a given geometric concept. For example, it *is* possible that more than three reference values are used to represent a geometric entity in 3D space; That means that there must be a *constraint* that links the four or more reference values, and that constraint must be modelled too. (The advocated way to do this is by referring to a meta meta model in the **Semantic_ID** where the constraint is formally encoded.) Another example comes from the fact that the abstract data type contains symbolic information about the **reference** with respect to which its model has meaning; that involves the choice of a reference **Frame**, and while **Cartesian reference frames** are a common choice, some use cases are better off with variants like, for example, **polar**, **cylindrical** or **spherical** references. In a **projective space**, one uses **homogeneous coordinate** representation, allowing also the points at infinity to be represented by finite coordinate values. In an **affine space**, one uses **Barycentric coordinates**, as soon as a **Simplex** is given as the basis for the coordinates.

3.2.7 Operators on Coordinates

Because **Coordinates** are not unique attributes of **Motion** of geometric primitives, there exist many transformations between equivalent **Coordinates** representations. This Section describes the different categories of such transformations, and the operators with which to

realise them.

Changes of Coordinates under changes of:

- `with_respect_to`: (TODO)
- `as_seen_by`: (TODO)
- `velocity_reference_point`: (TODO)

Addition of Velocity:

- the **Coordinates** of two **Twist** representations can be added, numerically, but **only if** all the semantic metadata of both representations are identical: same geometric entity, same `with_respect_to`, same `as_seen_by` and same `velocity_reference_point`.
- non-**Twist** representations of **Velocity** do not allow the numerical addition of their **Coordinates** representations. For example, for the representation of the **Velocity** of a **Rigid_body** by means of the **Velocity** representations of three or more of its **Points**, it does not make sense to add individual **Point**'s **Velocity** coordinates, because there are constraints to be satisfied between these **Velocity** representations.

From Velocity to Position, and back:

- the **exponential map** from **Velocity** to **Position** maps a **Velocity** to a **Displacement** (that is, a change in **Position**) of a geometric entity that corresponds to applying the **Velocity** on that entity for **one unit of time**.
- the **logarithmic map** is the inverse of the exponential map: it maps a **Displacement** to the **Velocity** that is required to cover the **Displacement** in **one unit of time**.

3.2.8 Data structures for Coordinates

A *data structure* model adds to things to the *abstract data structure* model:

- the **Coordinates** data structure of a concrete programming language with a set of *numerical values* representing the **quantity** of each coordinate. In the example of Fig. 3.5, the coordinates are an array with two values, the projections of the **Point**'s **Position** onto the x and y axes of a **Frame**.
- the semantic tags of the **Physical_units** of the numerical values; for example, **meter** or **radian**.

The mapping from the abstract data type to the coordinate data structure is *one-to-many*, because of the choices that can be made in, for example, (i) the *naming* and the *ordering* of the values in the data structure, (ii) the reference **Frame**, (iii) the physical units of each numerical value, (iv) how the data *access* is performed (e.g., via *named keys* or via *anonymous indexing*), (v) whether the numerical values of the coordinates are projections of the point on real-valued coordinate axes, or integer-valued indices on a grid,¹⁴ or (vi) whether several coordinate representations share the same units and reference frame.

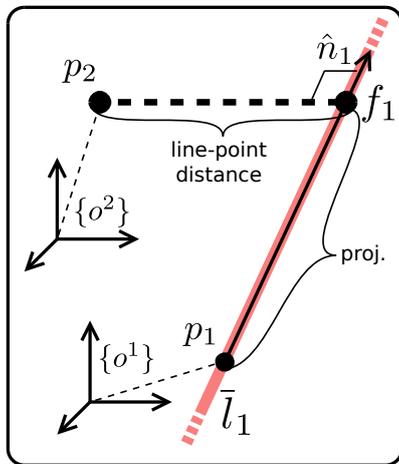
¹⁴The **TopoJSON** standard uses this approach.

One pragmatic approach in the choice of a coordinate representation is to use an already existing standard for the data structure part of the `Coordinates` model. **Prominent examples** are **Hierarchical Data Format** (“HDF5”), **GeoJSON**, or **Geography Markup Language**; Sect. 3.6 gives a more detailed overview. Standards like HDF5 cover multiple levels of abstraction, i.e., the abstract data type, data structure *and* storage representations.

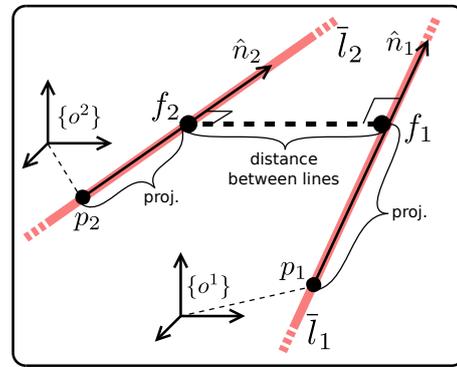
Important facts about coordinate representations are:

- a `Point` has no (need for a) coordinate representation; a unique symbolic `Semantic_ID` suffices.
- coordinates are needed in the *quantitative* representation of the `Position` of a `Point`. The coordinates represent the relative position of the `Point` to other geometric primitives.
- the same `Point` can be an argument in multiple `Position` data structures at the same time.
- the relation between a `Point` and the coordinate representation of a `Position`, is *not* that the `Point` has-a data structure of coordinates, but the other way around: the `Position` has has-a relations to, both, the `Point` abstract data type, and the data structure of the numerical coordinate values.

(TODO: `Coordinates` representations of all other geometric entities and relations.)



(a) Line 1 is expressed in $\{o^1\}$, Point 2 in $\{o^2\}$.



(b) Lines 1 and 2 are expressed in $\{o^1\}$ and $\{o^2\}$, respectively.

Figure 3.6: Graphical representations of the five possible relations between a point and a line 3.6a, and between two lines 3.6b.

3.2.9 Constraint relations on Coordinates — `Geometric_chain`

Robotic applications must keep track of the relative `Positions` and `Motions` of several geometric entities, while (i) some of these parameters are observed by sensors, and (ii) some of them are coupled by a model that, in general, is a **graph** of relative `Position` and `Motion constraint relations`. The combination of observations and constraint models allows the

Table 3.4: Summary of linear distance relations between point and line entities.

	point	line
point	point-point distance	
line	$\frac{\text{line-point distance}}{\text{projection of point on line}}$	$\frac{\text{distance btw lines}}{\text{projection (p1-f1)}} / \text{projection (p2-f2)}$
plane	point-plane distance	

Table 3.5: Summary of angular distances relations between versor and plane entities.

	versor	plane
versor	angle btw versors	
plane	incident angle	angle btw planes

robot to work with a “sufficiently full” world model, even when it can only observe a subset of all parameters in that model. An example from human driving: drivers have a mental model of a particular layout of traffic lights, traffic signs, and ground markings, and when they see one or more of those they can infer where to expect the others, even without spending perception efforts in that direction. Another example is a robot that estimates its **Position** with respect to a cup on a table (Fig. 3.7) by (i) observing how it moves with respect to the table, and (ii) remembering where the cup was on the table previously.

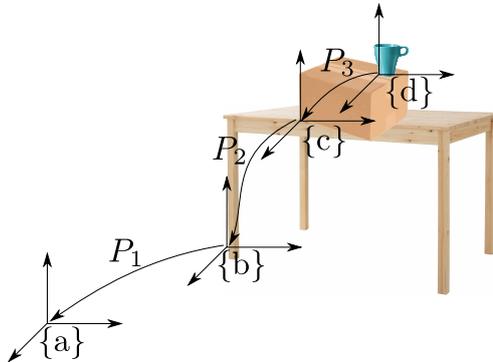


Figure 3.7: A **Geometric_chain** representing the relative **Poses** of a cup, a block and a table, where the constraints between them are of the **on-top-of** type.

Figure 3.7 shows **Frames** as representations of all geometric entities involved, but the constraint relations can also involve less than six-dimensional relations. For example, only a two-dimensional constraint can be used to represent that the block is resting on the table: one of its bottom **Points** must lie in the top **Polygon** of the table.

The simplest constraint graph is a **chain** of relations, sometimes called a **Geometric_chain**: there is a *sequential* order of some kind in which poses and their time derivatives are to be computed. For example, a cup can be placed on a block, that is itself lying on a table, that itself is standing on the floor, which itself is located in a particular room (Fig. 3.7). This situation represents the common case where the motions of several (rigid) bodies have relative motion constraints due to **natural** causes, such as gravity and the stiffness of the bodies’ materials. A [later chapter](#) describes the case of **engineered** motion constraints, in the form of “robots” or **Kinematic_chains**; any **Kinematic_chain** is also a **Geometric_chain**, but not vice versa.

3.3 Composition relations: Map as set of geometric entities

The sections above used *sets* of geometric entities, more in particular **Points** and their compositions. These compositions are denoted by **square brackets** "[" and "]", and they represent a **collection**. (An equivalent name is “*set*”.) The **Polyhedron** is clearly already a **collection** of **collections**, and this is the primary **mereological composition** of all geometric entities of Secs 3.2.1–3.2.2. However, there exist many geometric entities and relations that do *not conform*-to the Point-Polyline-Polygon meta model; for example, spheres, ellipsoids and cylinders; or **clothoidal** and **spline** curves. Each of those geometric *types* can also be given meta models, and each *instance* of these types can be **connected** to an instance of the Point-Polyline-Polygon type. So, the concept of **collection** of **collections** is a fundamental and generic composition relation for all sorts of meta models, especially but not exclusively, the geometric meta models.

This document uses “**Map**” as the generic name for any **collection** of **collections** of geometric instances, of whatever type. By definition, a **Map** can just be a set of other **Maps**. Figures 3.8–3.9 show examples of **Maps** (with only instances of the Point-Polyline-Polygon type). The **Maps** represent only some *geometry*, but the captions with the Figures already *interpret* this geometry in the context of an application; some of those additional *semantic tags* are described in Sec. 3.4.

That Section, and other later Sections, will add **constraints** (topological, geometric, as well as other types of constraints) to the mereological sets in **Maps**. According to the *best practices* advocated in this document, such extra semantics require new entity-relation meta models on their own, and those compositions will not (just) be called “**Maps**” anymore.

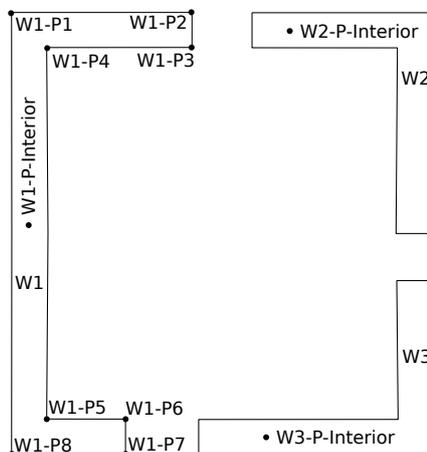


Figure 3.8: Map of the **walls** in a (part of) a building, as **Polygons**. In order not to overload the drawing, the naming of the **Points** is not complete, but just serves to give the gist of how a naming scheme could look like. The **Line_segments** are also not named explicitly.

(More or less) standardized meta models for **Maps** have matured in the **Geographical Information Systems** (GIS) contexts for **map making**. Two examples are the **GeoJSON** standard, and **OpenStreetMap**. Polygons are most often an approximation of the geometry of objects and relations in the real-world, but they represent a level of abstraction that is *always* relevant, in its own right, and as a basis to attach more detailed geometric abstractions to. For example, the **OpenStreetMap** database has a polygonal skeleton, to which a lot of “semantic tags” are attached; some of the latter have non-polygonal shapes, like for example traffic light icons.

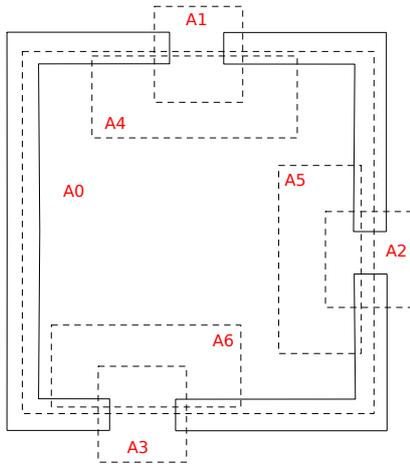


Figure 3.9: The Map of Fig. 3.8 is extended with some **areas** in the building. The areas are not physical, but serve as abstract conventions to allow humans and robots to indicate particular parts of the spatial domain.

3.4 Composition relations: `Semantic_map`, `World_model`

A `Semantic_map` is a `Map` composed with **semantic tags** (or “annotations”, or “metadata”) for some of the `Map`’s entities or relations. The meaning of the tags is relevant in the context of the `Semantic_map model`; the simplest version uses tag names that hint at application contexts, “stored” implicitly in the “background knowledge” of humans. Examples of such `Semantic_maps` are **bus or metro** lines, the **public road map**, or a ski area, all have their own specific tags: a metro station does not appear in a ski resort, while, conversely, a **green ski trail** does not make sense in a metro system, although both types of world models make use of the same geometric primitives.

An important **topological** constraint that a `Semantic_map` adds to a `Map` model is that of `Layer`, `View`, and `Model_diff`: collections of geometric primitives that “make sense” to be referred to together.

When the meaning of the semantic tags is modelled in the context of this document’s `Task meta model`, a `Semantic_map` becomes a `World_model`. Figures 3.10–3.11 give examples, where the **floor plan** semantic tags get meaning in an application that help robots navigate through an indoor environment.

The composition relations in world models are trivial and mereological, and not restricted to just the geometric primitives introduced above. Relevant extensions in the context of this document are those of **geometric chains** and of **kinematic chains**, but also tags that refer to the parts in the `Task meta model` (`plan`, `control`, `perception` and `monitoring`). This document also assumes that a `World_model` contains models of the **state** of the system it models; or rather, of **collections** of system **states**, because many applications need to represent various versions of reality: versions measured with different sensor sets; actual and desired versions for planning; multiple hypothesis of the world in perception; etc.

(TODO: resolutions, **state** of a world model, stream of (diffs) of **state**.)

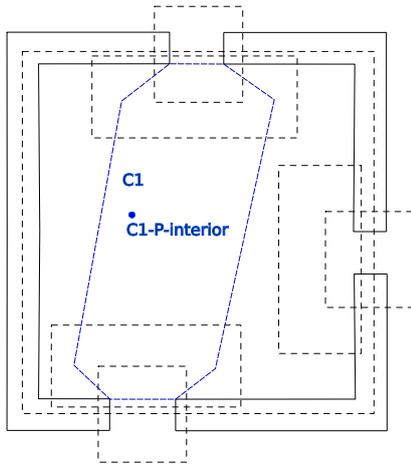


Figure 3.10: The Map of Fig. 3.9 is extended with one particular extra *area* that receives the semantic meaning of a *corridor*. That is, the semantic tag carries an **intention** of the purpose of that area.

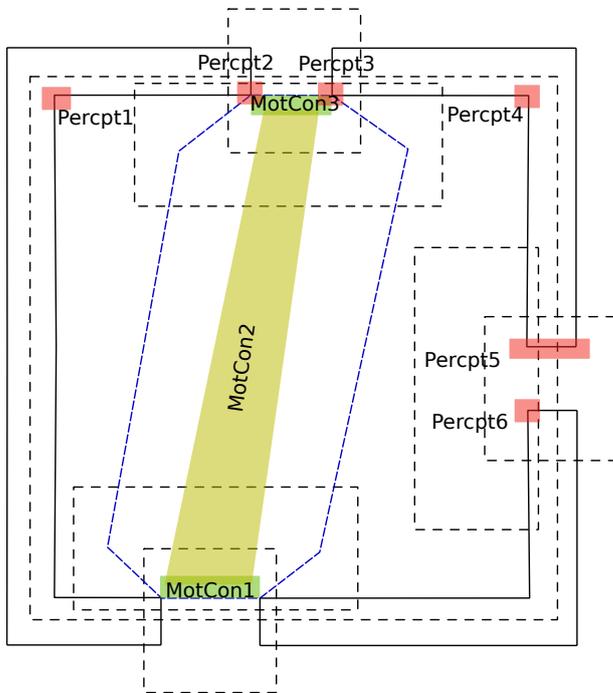


Figure 3.11: Map of the **motion** and **perception** areas that are relevant for a particular Task in the Map of Fig. 3.10. In order to follow that (virtual) “*corridor*”, the robot software should take into account the light-green “control” constraints, and the light-red areas that are best fits for the robot’s “perception” capabilities.

3.5 Uncertainty in geometric entities and relations

An uncertainty model represents the variability over the `Coordinates` representation (numerical, category, symbolic class,...) of a geometric entity or relation. It adds the extra semantics of a `Probability_distribution`, with abstract data type, numerical `Coordinates` representation, and a representation of physical units and dimensions.

An example is to represent the uncertainty on the `Coordinates` of the `Position` of a `Point` in Euclidean space with a **Gaussian probability distribution** (also called “normal distribution”), which can be numerically represented by its mean vector and its covariance matrix, Fig. 3.12. This model is also a composition of mathematical models (vector and matrix); additional constraints, like that the covariance matrix must be symmetric, are not modelled, for now. Similarly, a Gaussian mixture model can be created by composing an array of the

presented Gaussian models with the constraint used for scaling them appropriately.

There is little choice, in how to represent uncertainties on **Position**. The situation is more complex for most compositions of two or more **Points**, such as **Line_segments**, **Lines**, **areas**, **Frames**,... For example, representing the uncertainties on a **Line_segment** by means of Gaussian uncertainties on its two end points is *different* from representing it by means of a Gaussian uncertainty on its **start Point** together with an uncertain direction **Vector**.

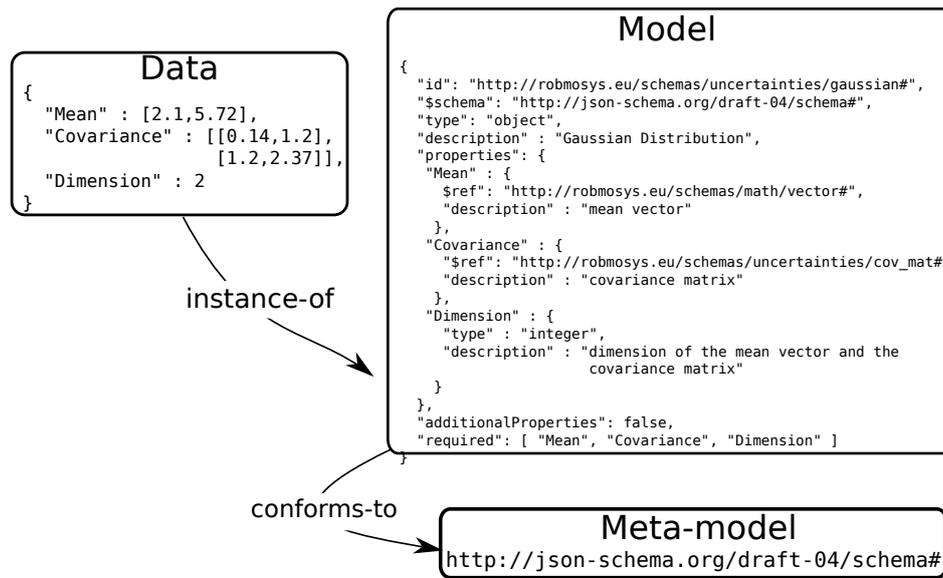


Figure 3.12: A valid data instance of a **JSON-Schema** model representing a Gaussian distribution. The schema is a composition of other schemas (for vectors and covariance matrices) and includes a few constraints on the data structure, such as the values required for the validation of the JSON document. Moreover, the schema **conforms-to** a specific meta-model of **JSON-Schema**.

3.5.1 Sources of uncertainty

Sources of uncertainties can be, but are not limited to:

- **uncertainties by construction**, which are those uncertainties that represent approximations over the description of the geometric entity attached to it. This is the case for mechanical constraints, such as the coupling tolerance in a joint;
- **sensor noise**, which is a property of the sensor but can be influenced by other factors such as environmental condition or robot motions;
- **process noise**, which represents the uncertainty in the modelling of a behaviour.
- **categorical confusion**, which represents the uncertainty in knowing to which category a particular entity belongs.

3.5.2 Covariance of a Frame has no meaning

An important **mathematical fact** is that $SE(3)$ does not have a bi-invariant metric; in engineering terms this means that it makes no sense to talk about the “distance” between two **Frames** or “to add or subtract” two **Frames**, or to compute their “mean”. Hence, also the “covariance matrix” on rigid body **Motion** or **Force** is void of any physical meaning. In practice this means that one *always* must introduce a weighing factor to balance the physical dimensions of the translational and angular parts, and this weighing factor is *always arbitrary*.

A representation of the uncertainty on a **Frame**, *with* consistent physical meaning, consists of *choosing three Points* on the **Frame** (e.g., the **Point** at the **Frame’s** origin and the **end Points** of two of the **Frame’s** three Cartesian unit **Vectors**), and adding a **Position** covariance matrix to the coordinate representations of all of them. This results in a *non-minimal* representation of the uncertainty; the constraints that have to be added reflect the facts that the two **Vectors** must have *unit length* and are *orthogonal*. Of course, this representation *also* introduces an arbitrary weighing between translation and orientation, albeit in an indirect way, via the choice of which points to select in the representation.

3.6 De facto meta model standards for Coordinates

Including specific built-in datatypes provided by different programming languages, there are several digital data representation models (and tools) available, each one covering a (set of) specific features or use-cases. In most cases, a digital data representation model can be described by means of an *Interface Description Language* (IDL), with the purpose of being cross-platform, and/or decoupling from the programming language that implements a certain functionality, thus allowing communication between different software components.

Section 11.10 gives an overview of common standardized “host languages” that can (hence) also be used to model **Coordinates** representations. But it is common to find a digital data representation format (meta-model) dedicated to a **specific framework** or communication middleware (e.g., **CORBA**, **DDS**); in the latter case, the digital representation instance is also called *Communication Object*. Nevertheless, the relations between the data, its digital data representation model, and the meta-model used to define a specific data representation holds among the different alternatives, as depicted in Figure 3.13; a concrete example is discussed in Figure 11.8.

The following Sections explain the popular **Coordinates** representation approaches in robotics.

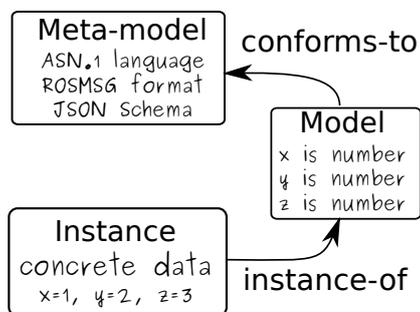


Figure 3.13: Digital representation models: a data structure in software is an **instance-of** a digital data representation model, which is a formal description that **conforms-to** a meta-model. A concrete example is shown in Fig. 11.8.

To evaluate the positive impact of a digital data representation meta-model (and its

underlying tools) as bones of a composable software solution, the following aspects must be evaluated:

- **expressivity** of a meta-model to describe different properties over the data, including:
 - basic, built-in data types available;
 - possibility to indicate *constraints* on the data structure;
 - customisation over the memory model to store the data (instance of the model);
- **validation**: availability of a formal schema of the digital data model, meta-model and tools to validate both data instance and model schema;
- **extensibility**: the possibility to extend (by composition) the expressivity level of a digital data representation model;
- **language interoperability** (also called neutrality): the capability of a model of being language-independent; this requires a specific compiler to generate code-specific form of the digital data representation model;
- **self-describing**: optional capability of injecting the model in the data instance itself (metadata), or at least a reference to it; this enables reflection and run-time features.

Below follows a non-exhaustive list of those models, with a special attention to models used in robotics.

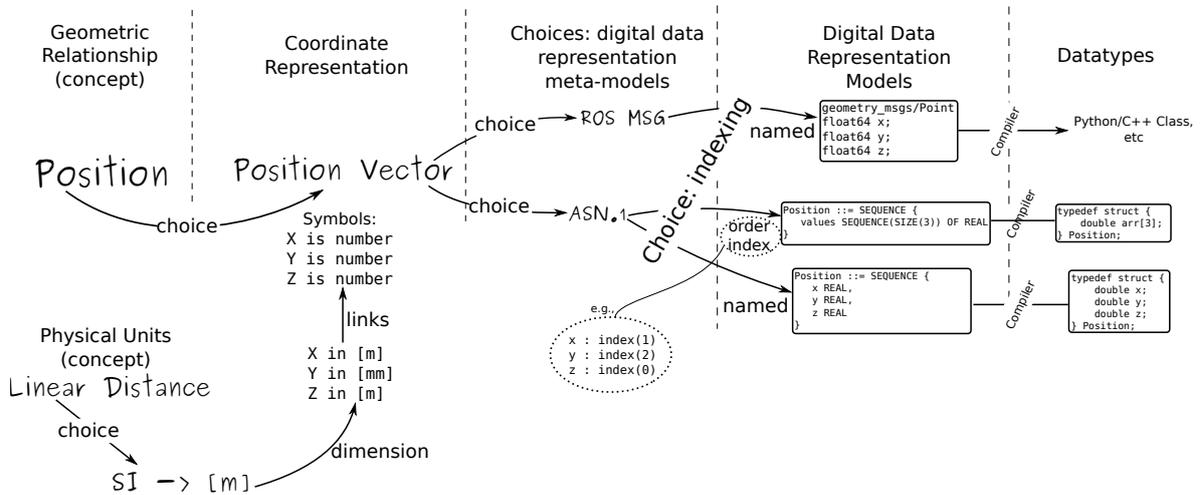


Figure 3.14: From geometric relations to digital data representation: choices for the grounding of the concept of *Position*.

3.6.1 ROS Messages

The **ROS message** is a digital data representation meta-model developed for the ROS framework, aimed to describe structural data for serialisation and deserialisation within the ROS communication protocol, namely ROS topics, services and actions. Precisely, the (non formalised) meta-model is strongly coupled with the chosen ROS communication pattern:

- ROS messages (`msg` format) for streamed `publish/subscribe` ROS topics;
- ROS services (`srv` format) for blocking `request/reply` over ROS;
- ROS actions (`action` format) for ROS action pattern.

The need of having a different data model for each communication mechanism provided reduces the degree of composability of the overall system, enforcing the component supplier to a premature choice. However, it is possible to compose message specifications from existing ones, such as services and action models are built starting from messages models. The expressivity of a ROS message description is limited with respect to other alternatives (e.g., ASN.1, JSON-Schema): it allows to specify different types for numerical representations, (e.g., `Float32`, `Float64` for floating-point values) but there is no support for constrains over a numerical value, nor specific padding and alignment information. Moreover, there is no built-in enumeration values, which is usually solved with few workarounds¹⁵. However, default values assignment is possible in the ROS message models. ROS messages are self-describing by means of a generated ID (MD5Sum) based on a naming convention schema of the message name definition and a namespace (package of origin). Language-neutrality is provided by the several compilers available within the ROS framework. However, there is no efficient encoding mechanism applied, reducing the compilation process to a mere generation of handler classes for the host programming language. Despite the technical shortcomings of the ROS messages, they are the likely most used digital data representation model in the robotics domain, due to the large diffusion of the ROS framework (which does not allow another data representation mechanism¹⁶).

3.6.2 RTT/Orocos typekits

The `RTT/Orocos typekits` for geometry are digital data representation models directly grounded in C++ code, which are necessary to enable sharing memory mechanisms of the RTT framework. However, it is possible to generate a typekit starting from a digital data representation model if a dedicated tool is supplied. For example, tools that generate a typekit starting from a ROS message definition exists.

3.6.3 SmartSoft Communication Object DSL

The `SmartSoft framework` provides a specific DSL based on the `Xtext` DSL tool of the `Eclipse Modeling Framework`. It describes a digital data representation for the definition of primitive data types and composed data-structures. The DSL is independent of any middleware or programming language and provides grounding (through code generation) into different communication middlewares, including CORBA IDL, the message-based *Adaptive Communication Environment*¹⁷ (ACE), and DDS IDL. Moreover, the tool designed around the SmartSoft Communication Object DSL allows to extend the code-generation to other middleware-specific or language-specific representations.

¹⁵An `UInt8` type with unique default value assigned for each enumeration item.

¹⁶It is possible to have other representations over ROS messages, e.g., JSON documents, by using a simple `std_msgs/String` message.

¹⁷see <http://www.cs.wustl.edu/~schmidt/ACE.html>

3.7 Differential geometry as meta meta model

Differential geometry (e.g., [18]) is the mathematical theory (including an unambiguous terminology and notation) of the geometry and dynamics of robotic systems (and other energy transforming cyber-physical systems), starting with the *manifold* of all positions, and the tangent and co-tangent spaces representing the velocities and forces, in the form of, respectively, the *tangent vectors* at a point on that manifold, and the *co-tangent vectors* (or *1-forms*) over each tangent space. The concept of a *jet* fits well to the geometrical combo of position, velocity and acceleration of the same point or rigid body.

(TODO: much more explanation about what understanding of differential geometry is exactly needed to exploit it constructively in the design of robotics and cyber-physical systems.)

Chapter 4

Meta models for a kinematic chain and its instantaneous dynamics

One way to describe the motion of a robot is as the instantaneous **transformation of energy** between the robot's **motors** (in the so-called “**joint space**”) and its various **end effectors** (in the so-called “**Cartesian space**”). The **mechanical structure** of the robot's **Kinematic_chain** thereby acts as the physical **energy transformer**. Each kinematic chain **constrains** the rigid bodies it connects **mechanically**, and this constrained behaviour is a semantic extension of the **Geometric_chain**: the *cause* of the constraints is related explicitly to mechanical entities. The major semantic contribution of this Chapter is to model (i) the **types** of motion constraints, (ii) the *abstract data types* that model their coordinates, and (iii) the behavioural relations of *mechanical dynamics*.

The next set of extensions make the link with the **task** meta model: (i) the **Forces** in the actuators of the chain are task *resources*, (ii) the **Motions** and **Forces** of the chains' **Rigid_bodies** are task *capabilities*, and (iii) **Attachments** allow to add **Force** and **Acceleration** constraint models. Together, these semantic entities suffice *to specify* any type of **instantaneous motion** that is physically allowed by the kinematic chain.

Later Chapters introduce even further extensions, which relate the specifications to (i) the *controlled execution* of the specification, and (ii) the *intended* effects of such control.

4.1 Meta models

This Section introduces the entities and relations that the meta models of the **Kinematic_chain** add to those of **geometric meta models**.

4.1.1 Mereology

The **mereo-topological** meta model of a **Kinematic_chain** has the following parts (Fig. 4.1):

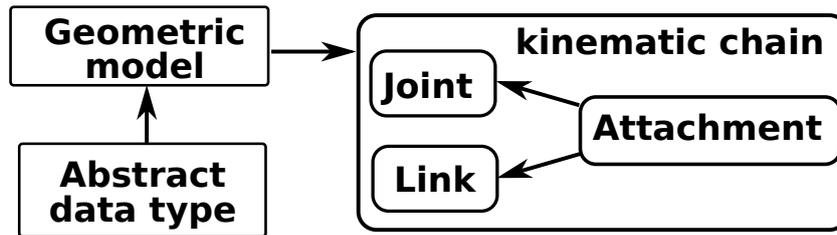


Figure 4.1: A `Kinematic_chain` model is a graph of `Link` and `Joint` entities, which are themselves defined as compositions of *geometric* entities (Sec. 3.2, Fig. 3.1) and their *abstract data type* representations. Each link can have one or more `Attachments`, to allow extensions by means of *model composition*, such as, geometric shape, dynamical properties, sensors, actuators, or handles for task specifications. This sketch only represents the *mereo-topological* parts of the `Kinematic_chain` meta model; an arrow represents a *is-part-of* relation, which is the *inverse* of the *has-a* relation.

- a collection of `Links`. `Link` is just another name for `Rigid_body`, in the semantic context of kinematic chains, that is, it refers to a `Rigid_body` that gets semantic tags to build a kinematic chain with.
- a collection of `Attachments` rigidly connected to each `Link`. An `Attachment` is a geometric entity (e.g., `Point`, `Vector`, or `Frame`) that serves as an argument in **composition relations** between `Links` and other relevant domain models. For example, at an `Attachment`, the geometrical properties of a `Link` can be connected to (models of) mechanical inertia, geometric shape, perceivable markers, or motors, sensors and tools.
- a collection of `Joints`, each being a special case of the mentioned composition relations, namely a **constraint relation** on the relative `Motion` between two `Links`. Major arguments in each such constraint relation are:
 - the `Attachments` on each `Link`. There should be one and only one such `Attachment` on one particular `Link` for one particular `Joint`.
Each `Link` can be constrained by more than one `Joint`; for example, most `Links` in robot are part of a serial connection of `Joints`.
One `Joint` can constrain more than two `Links`; for example, human joints like the shoulder have tendons that are attached to (and hence, constrain) multiple bones via multiple muscles.
 - the **type** of the motion constraint.

4.1.2 Types

The two common families of mechanical motion constraints are:

- **lower pair** motion constraints, with the one-dimensional `Revolute` (Fig. 4.2) and `Prismatic` joints as major representatives. These are **bi-directional** constraints (generating positive and negative constraint forces) and have a **state** variable (“*q*”) whose

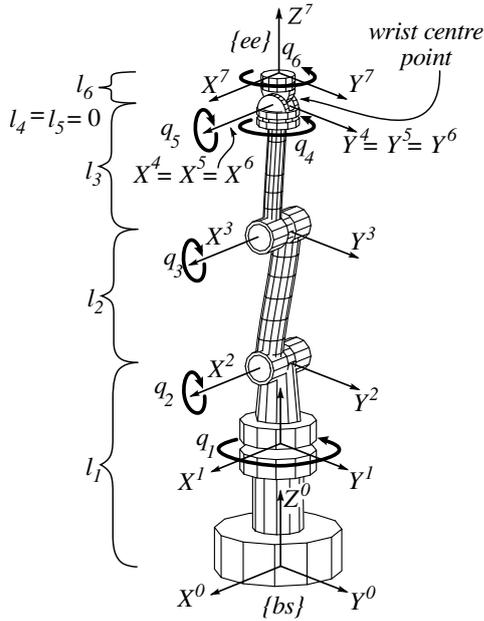


Figure 4.2: The most common kinematic design of industrial manipulator arms, using only revolute joints as bi-directional motion constraint between seven rigid bodies. This particular geometric configuration is a kinematic family parameterized by the symbols in the drawing. What is not symbolically represented on the drawing are the facts that (i) joints 2, 3 and 4 have parallel axes, (2) joints 4, 5 and 6 have intersecting axes, and (iii) joints 1 and 2 have orthogonal axes. All members of that family have the same mereo-topological model and the same geometrical parameters of the attachments of joints to links, but each member has different numerical values of these parameters.

value and its time derivatives are a one-on-one representation of the relative Motion of the two constrained Links:

$$\text{Position (Link}_1, \text{Link}_2) = f(q), \quad (4.1)$$

$$\text{Velocity (Link}_1, \text{Link}_2) = g(q, \dot{q}), \quad (4.2)$$

$$\text{Acceleration (Link}_1, \text{Link}_2) = h(q, \dot{q}, \ddot{q}), \quad (4.3)$$

- **higher pair** motion constraints, with *wheels* and *cables* as major representatives. These constraints do not have position-level state variables, but only velocity-level variables; hence their name of **non-holonomic** motion constraints: the constraint can not be “integrated” to an equivalent constraint formulation with position-level variables.

Higher pair constraints have some **uni-directional** constraint subspaces, in which constraint forces in only one direction are possible. For example, cables (Fig. 4.3), edge-surface contacts (Fig. 4.4) as in wheels, or vertex-surface contacts (Fig. 4.5).

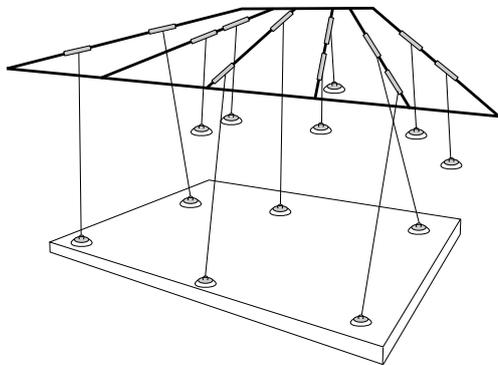


Figure 4.3: A cable support as uni-directional motion constraint between two rigid bodies. The mechanical framework on the top contains motors to control the length of the cables, as well as their position on the horizontal guides. The load is depicted at the bottom of the drawing, with unspecified “attachment tools” between cables and load.

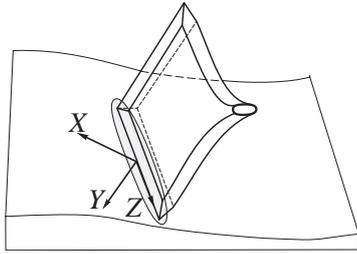


Figure 4.4: An *edge-surface* contact as uni-directional motion constraint between two rigid bodies. This model represents (part of) the geometrical abstraction of real-world motion constraints such as wheels or skates.

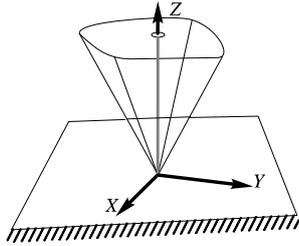


Figure 4.5: A *vertex-surface* contact as uni-directional motion constraint between two rigid bodies.

4.1.3 Coordinates and behavioural state for Motion

Coordinate representations of motion constraints are needed to formalise the geometric **motion behaviour** of a kinematic chain, that is, the **constraints** that the chain adds to the relative **Motion** (**Position/Pose**, **Velocity/Twist** and **Acceleration**) of its individual **Links**. Figure 4.2 shows an example of a geometrically parameterized model of a **Kinematic_chain**. The abstract data type for **Coordinates** is a collection with the following entities:

- the geometrical dimensions of the **Link**.¹ This is done via the meta models of Sec. 3.2.
- the **Coordinates** of each **Attachment**. These **Coordinates** use the same **as_seen_by** reference frame as the **Coordinates** in which the **Link**'s geometrical dimensions are expressed.
- the *type* of **Joint** that is meant to be connected to each **Attachment**.
- the *model* of the **Joint**'s mathematical expression as a motion constraint. Some parameters in that model come from the **Attachments** on the **Links** involved in the motion constraint, some come from the type of the **Joint**, and still others represent the motion limits of the specific **Joint** instance.
- the **state** of a **Joint** is the subset of the parameters in a **Joint**'s mathematical model that represent the **motion behaviour** of the **Joint**. That is, the numerical values of the actual relative **Position**, **Velocity** and **Acceleration** of the **Links** connected by the **Joint**.

The *type* of the **Joint** must be represented formally, to allow software tools to check whether (i) both **Attachment** and **Joint** have the correct geometric primitives in order to be composed together, and (ii) the motion constraint model below links its state variables to appropriate geometric entities.

¹A **Link** is considered to be a **Rigid_body**. Flexible links are not treated in this document. However, the mereo-topological modeling still applies, since rigidity is a geometric concept.

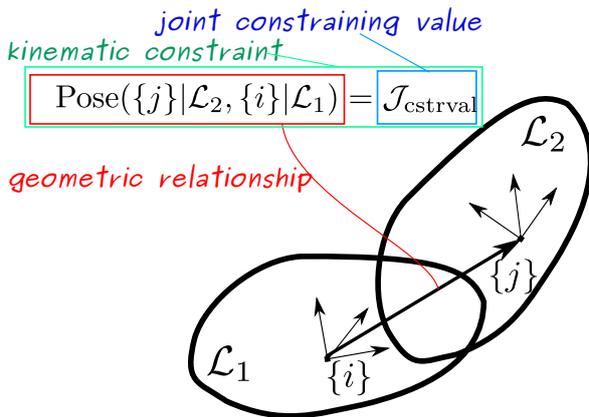


Figure 4.6: A generic model of the mathematical expression for the simplest Kinematic_chain, namely one with two Links and one Joint. The Joint has an unspecified type, but it has a state representation. That means that the relative Pose relation can be expressed as a *bijection* between relative Poses and one single joint position value $\mathcal{J}_{\text{cstrval}}$. The frames $\{i\}$ and $\{j\}$ serve as *Attachments*, fixed to the Links \mathcal{L}_1 and \mathcal{L}_2 , respectively.

For some Joint types (such as `Revolute_joint` and `Prismatic_joint`), a **finite** set of state parameters can be given: one `joint_position` for the relative Position, one `joint_velocity` for the relative Velocity, and one `joint_acceleration` for the relative Acceleration. The span of these numerical values is called the **joint space** of that Joint, and that joint space model (always) has constraints on the allowable *range* of the state parameters. The set of spatial configurations of the Links that correspond to the joint space range is called the **Cartesian space** of that Joint.

Figure 4.6 depicts the generic mathematical model for a Joint motion constraint that has a **state** representation. The drawing sketches only the position level of the motion constraint, with a `joint_position` state value $\mathcal{J}_{\text{cstrval}}$; the straightforwardly extensions to the velocity and acceleration levels include the time derivatives of the `joint_position` state. The *dimensionality* of that state parameter $\mathcal{J}_{\text{cstrval}}$ is a number between “6” (rigidly connected) and “0” (not constrained at all). The coordinate representation of the constraining value must be compatible with the coordinate representation of the geometric relationship. As mentioned above, the `Revolute_joint` and the `Prismatic_joint` have constraint expression as a *function* of just one variable. More complex Joints (such as the higher-pairs) do not have clearly defined Joint_positions, e.g., the **knee** and **shoulder** joints in human bodies, or the **unilateral motion constraints** of contacts or cables.

Because a Kinematic_chain is just a collection of geometric primitives with semantic tags that have meaning in the Kinematic_chain context, the formal representation of a **Semantic_map** is the obvious model primitive to apply. Table 4.1 shows an abstract example of a mereo-topological Kinematic_chain model.

(TODO: examples of concrete models; e.g., for a robot arm of the **321 kinematic family**. Make the model in Table 4.1 complete and consistent with the textual description.)

4.1.4 Geometrical operations — Forward, inverse and hybrid kinematics

Modelling the Motion behaviour of all Links in a Kinematic_chain requires no extra operations in addition to the geometrical ones already introduced in Sec. 3.2.7. At the level of abstraction of the whole Kinematic_chain however, extra operations are needed:

- *structural* operations of composition and decomposition of Kinematic_chain models. These operations are introduced in Sec. 4.5.

```

{ Semantic_map :
  { {          ID : KC_ID_123XYZ },
    {          MID : Kinematic_chain },
    {          MMID : [ Geometry, Euclidean_space, Frame, QUDT ] }
    {          links : [ Link_1_ID, ..., Link_2_ID ] },
    { attachments : [ { Link_1_ID : [ Link_1_Att_1_ID, Link_1_Att_2_ID ] },
                      ...,
                      { Link_6_ID : [ Link_6_Att_1_ID, Link_6_Att_2_ID ] },
                      { Link_7_ID : [ Link_7_Att_1_ID ] }
                    ] },
    { joints : [ { {          ID : Joint_1_ID },
                  {          MID : Revolute },
                  { first : Link_1_Att_2_ID},
                  { second : Link_2_Att_1_ID }
                },
                ...,
                { {          ID : Joint_6_ID },
                  {          MID : Revolute },
                  { first : Link_6_Att_2_ID},
                  { second : Link_7_Att_1_ID }
                }
              ] ] } }
}

```

Table 4.1: Example of Coordinates model, in this case the Position of a Point.

- operations on the Coordinates of the *dynamical behaviour* of Kinematic_chains. These operations are introduced in Sec. 4.3.
- operators that *transform* between joint space entities and Cartesian space entities. These are further described in the paragraphs below.

One identifies the following three categories of the latter transformation relations, where a later one encompasses all of the earlier ones:

- **Position** transformations: some joint space position values are given, as well as some Cartesian position values, and the operator takes these as inputs and generates as outputs all the non-specified entities. An additional outcome is a measure of the *consistency* of all specified inputs.
- **Position–Velocity** transformations: similarly of for the Position-only transformation, but now with inputs and outputs also at the velocity level of abstraction.
- **Position–Velocity–Acceleration** transformations: idem, now including also the acceleration level of abstraction.

Special cases of these transformations are when the inputs consist of only a complete set of joint space values, or a complete set of Cartesian values; the terminology is, respectively, **Forward_kinematics** and **Inverse_kinematics**, with a semantic tag indicating the relevant

level of motion abstraction. The generic case is seldom used; this document names it the `Hybrid_kinematics`.

Formalization of these transformations is straightforward:

- add an `Attachment` for each of the input and output entities. Or reuse an already existing one; this holds in particular for the joint space specifications, because the model of a `Joint` already required the introduction of `Attachments`.
- add a `Specification` relation for each of the `input` entities, with as arguments:
 - the part of the `Attachment` that is used for the specification.
 - the symbolic tag `input`.
 - the `value` of the specified `input` entity, together with its physical `units`.
- add a `Specification` relation for each of the `output` entities, with as arguments:
 - the part of the `Attachment` that is used for the specification.
 - the symbolic tag `output`.
 - the symbolic `variable` of the desired `output` entity, together with its physical `units`.
- add two symbolic status variables, one `input_status` and one `output_status`, that encode, respectively, the desired and actual consistency of inputs and outputs.

It is indeed possible that the specified inputs can not give rise to a uniquely computable set of outputs (Sec. 4.1.5). The `input_status` symbol encodes the choices that can be made in computing the transformation; the `output_status` encodes the consistency that is realised by the transformation. These choices depend on the type of transformation, and on the numerical solver that is used to implement the transformation computations.

4.1.5 Inconsistency, redundancy, singularity

Because kinematic chain relations are **non-linear functions** of their constituent geometric entities, some input-output functions can be:

- **inconsistent**: the requested output can not be computed with the given inputs and the given kinematic chain model, because, for example, the chain has two loops when one is expected, or it has one loop when none is expected, etc.
- **redundant**: more than one output is consistent with the same input.
- **singular**: the computations to find the outputs are numerically ill-conditioned, because the physical energy transformation between Cartesian and joint space reaches an extremum.

While these insights are already part of the meta models, it is really only in *software* implementations that they pose challenges: simplistic implementations will crash because the problems often occur only in specific configurations of a kinematic chain, or with specific input-output combinations.

4.2 Meta models of mechanical dynamics: composing force and motion

A [previous Section](#) introduces the Motion behaviour of geometric primitives (Position, Displacement, Velocity and Acceleration); this Section adds the **dynamics** behaviour relations of the mechanical domain, that is, the relations between **force** and **Motion**, and any **composition** of such relations. In other words, in what ways can forces influence the motion of rigid bodies?

4.2.1 Abstract data types and data structures for dynamics

The *abstract data type* for a **Force** is (formally but not semantically) equivalent to that of a **Twist**: it has **Coordinate** representations that look very similar, but only the semantic tags for the same parts are different. It is sometimes given the name of **Wrench**, and is composed of the two parts of a **screw**: a force **Vector** in 2D or 3D, and moment **Vector** in 1D or 3D, respectively.

A similar observation as with **Twists** was already made in the early 19th century, by the French mathematician Lous Poinsot (1777–1859), [79], who formulated the following theorem: *any system of forces applied to a rigid body can be reduced to a single force, and a couple in a plane perpendicular to the force.* This is a formulation of the **screw axis** in disguise. Table 4.2 shows an example of a **Force** representation using the **screw_axis** model.

(TODO: complete formal semantics, with examples. Including transformation and mapping operators.)

4.2.2 Motion as result of constrained optimization — Gauss’ Principle

The motion of a rigid body under the influence of forces (including gravity) is represented by the **Newton-Euler equations**. This is a **procedural** approach to describe motion; **declarative** alternatives exist too, via the general, differential-geometric concept of a **geodesic motion** on a manifold equipped with a *metric*, the latter being the mass distribution in the manifold; or via the dedicated methods of **d’Alembert** [25], **Lagrange** [64], **Hamilton** [50], **Gauss** [42], **Jourdain** [57] or **de Maupertuis** [27]. The declarative (or “**variational**” or “**constrained optimization**”) approaches are overkill for the *free* motion of a rigid body, but become practical whenever *constraints* occur on the free motion of the body, such as via mechanical contacts.

The most generic “*Principle of Least Constraint*” is that of Gauss: it provides an optimization relation between (i) the instantaneous acceleration of a non-constrained point or rigid body, (ii) the geometrical properties of a **motion constraint**, (iii) the instantaneous acceleration that satisfies that motion constraint, and (iv) the constraint force.

The major modelling step in this context (that is, the composition of the inertial effects of two rigid bodies connected by a frictionless and rigid one-degree of freedom motion constraint) is the topic of Chapter 4.

4.2.3 Energy relations — Bond Graphs

The previous Sections describe knowledge relations about how force and motion are coupled via the “dynamics equations” of a **Point** or a **Rigid_body**. These relations represent the

instantaneous coupling between force and motion, but any mechanically moving system has **state** entities that determine the force-motion couplings over longer periods of time. **energy** is a major state entity, that appears at all time scales of **Motion**:

- **Acceleration_energy**: the map of mass and acceleration into a form **acceleration times mass times acceleration**. Hence, it is linear in the mass and quadratic in the acceleration. This form of “energy” can not be stored or dissipated, but is the **measure** that is optimized in Gauss’ principle, to find the instantaneous motion of a *constrained* point or rigid body.
- **Kinetic_energy**: the map of mass and velocity into a form **velocity times mass times velocity**. Hence, it is linear in the mass and quadratic in the velocity. This is one way of storing energy in a moving point or rigid body.
- **Potential_energy**: the map of mass and displacement/position into a form **g(mass, displacement)**, which is linear in the mass but *can* be nonlinear in the displacement. Two major examples in mechanics are: the potential energy of a mass in the gravity field, and the potential energy of a deformed spring. This is another way of storing energy in a moving point or rigid body.
- **Work**: the map of force and displacement into a form **force times displacement**. Hence, it is linear in both arguments. This mapping has the physical dimensions of energy, and can not be stored or dissipated, but is a **measure** to represent how much energy is needed to realise a particular non-instantaneous, finite displacement motion of a point or rigid body.
- **Damping/Friction**: the map of force and velocity into a form **h(force, velocity)**. It is linear in force but *can* be nonlinear in velocity. This mapping has the physical dimensions of energy, and represents energy dissipated in an instantaneous motion via mechanical friction or damping.

Physics has resulted in generic composition meta models that represent the (instantaneous) **exchange of energy** between rigid bodies (or non-mechanical equivalents in other **system dynamics** domains such as **electromagnetism**, **thermodynamics**, or **chemical engineering**). **Bond graph** theory is major example² of such a meta model, with its foundations built on the flows of energy via **Block-Port-Connector** mechanism.

4.2.4 Differential geometry: manifold, (co)tangent space, linear forms

Differential geometric entities and relations have been introduced superficially [earlier in this document](#), but now the domain of mechanical dynamics provides a good context in which the complexity of a differential geometric representation can add value to the modelling. Indeed, differential geometry has a rich nomenclature and formalisations with which the modeller can differentiate unambiguously between entities and relations that are often confounded in mainstream engineering contexts, and hence to bring structure in the models based on a solid scientific foundation. Here is an overview of the entities and relations to be used in models of the dynamics of mechanical systems:

²Unfortunately, its designers did not have the same *separation of concerns* principles in mind when formalizing the theory, which has led to too much coupling between the mathematical and the abstract data type levels of abstraction.

- **manifold** $M(P, p)$ of Positions p of a Point P .

In robotics, a “point” in the manifold can also be a line, a plane, and often also a `Rigid_body`. These manifolds are *not* **metric space**, since “the” distance between two elements in those manifolds has no unambiguous meaning.

- **tangent space** $TM(P, p, v)$ of all Velocities v of the Point P at a given Position p in the manifold M .
- **second-order tangent space** $TTM(P, p, v, a)$ of all Accelerations a of a Point P at a Position p in the manifold M that has the Velocity v in the tangent space at that Position. There is a **constraint** between p , v and a , in that a is the acceleration of P at p when v is already the velocity of the *same* Point P at the *same* Position p . This constraint relation is sometimes referred to under the name of “**jet**”, [24].
- the **composition** of Motion relations has the following properties:
 - Displacement compositions form a **multiplicative group** in the manifold M . (So, not the more familiar **additive** one!)
 - Velocity compositions form an **additive group** on TM . It has no natural metric, but does have a natural origin, namely the “zero velocity”.
 - Acceleration composes **additively** on $TTM(P, p, v)$, and depends nonlinearly on the Position p and the Velocity v . So, it is not really a group operation, just a continuous **manifold**.

- **co-tangent space** $\overline{TM}(p, f)$ of **linear forms** f at Position p that **map** Velocities v in the tangent space at p into a real scalar value w :

$$f : TM(p) \rightarrow \mathbb{R} : v \mapsto f(v) = \langle f, v \rangle = w. \quad (4.4)$$

In mechanics, f is called a **Force**, and the pairing w of force and velocity is **Power**, that has the physical units of **Energy** per unit of time.

- **mass (inertia)** \mathcal{M} is a linear mapping from a Velocity v from TM onto an element \bar{p} from \overline{TM} , called the **Momentum** of the mass \mathcal{M} with the Velocity v :

$$\mathcal{M} : TM(p) \rightarrow \overline{TM}(p) : v \mapsto \mathcal{M}(v) = m. \quad (4.5)$$

The relation between momentum m and force f is **Newton’s law**: force is the change of momentum over time. It is a **conserved quantity**.

The mass can also serve as a **metric** on the manifold of velocities, by combining the momentum map with the force-velocity pairing, applied to the *same* velocity. The result is indeed a **bilinear form** that maps that Velocity v from TM onto a real scalar w :

$$\mathcal{M} : TM(p) \rightarrow \mathbb{R} : v \mapsto \frac{1}{2} \langle \mathcal{M}(v), v \rangle = w. \quad (4.6)$$

The resulting energy is the **Kinetic energy** stored in the moving mass. It can serve as a **metric** on TM [18].

A mass \mathcal{M} can also serve as a metric on TTM , the space of **Accelerations**, and in this case the scalar is sometimes given the name of *Zwang* [42], or *acceleration energy* [101].

- **damping** \mathcal{D} is a (not necessarily linear) mapping from a **Velocity** v from TM onto a linear form f_d (“friction force”) in \overline{TM} .

$$\mathcal{D} : TM(p) \rightarrow \overline{TM}(p) : v \mapsto \mathcal{D}(v) = f_d. \quad (4.7)$$

Hence, the pairing of the friction force f_d with the velocity v which causes the friction maps into a real scalar w :

$$\mathcal{D} : TM(p) \rightarrow \mathbb{R} : v \mapsto \mathcal{D}(v) = \langle f_d, v \rangle = w. \quad (4.8)$$

The resulting energy is **Heat** lost in friction; it has physical units of **Energy**. Damping can only serve as a metric on TM when it is linear.

- **elasticity** \mathcal{K} is a (not necessarily linear) mapping from a **Displacement** d between two **Positions** on M onto linear form f_e (“elastic force”, “spring force”) in \overline{TM} .

$$\mathcal{K} : M(p) \times M(p') \rightarrow \overline{TM}(p) : (p, p') \mapsto \mathcal{K}(p, p') = f_e. \quad (4.9)$$

Hence, the pairing of the elastic force f_e with the displacement (p, p') which causes it maps into a real scalar w :

$$\mathcal{K} : M(p_1, p_2) \rightarrow \mathbb{R} : \text{Displacement}(p, p') \mapsto \mathcal{K}(p, p') = w. \quad (4.10)$$

The resulting energy is the **Potential energy** stored in a spring. Because the limit of a **Displacement** (p, p') is a **Velocity**, \mathcal{K} can serve as a **metric** on TM .

- **impedance**: the combination of one or more of the relations **mass**, **damping**, and/or **stiffness**. In general, **impedance** relations are **non-linear**, but in many engineering contexts, their **linear** approximations suffice:

$$f = M a, \quad (4.11)$$

$$f = D v, \quad (4.12)$$

$$f = K \Delta p. \quad (4.13)$$

4.3 Meta model for the mechanical dynamics of a kinematic chain

This Section links the **mechanical dynamics** meta model with **geometrical Coordinates** meta model. The composition represents the **physical phenomena** that:

- a **physical body** has **mass** and hence its (change in) motion is (*always*) related to an (inertial) force, via **Newton’s laws**.
- a force can (*in some situations*) act on a body via a **spring** or a **damper** connected between a force and a body.
- the interaction between force and motion is regulated by **mechanical energy constraints**. More in particular the **conserved** energy components of **potential** and **kinetic** energy, and the **tribologic** energy (friction, lubrication and wear) that is **dissipated** into **thermal energy**.

For a robot, the forces have “external” sources (interaction forces or gravity) and “internal” sources (torques at the joints generated by actuators connected to the robot’s **Kinematic-chain**).

```

{ Coordinate_Force_Frame_Array_Newton_Meter :
  { {
      relation: Force },
    {
      screw: [ force: f}, { moment: m} ] },
    {
      on: { Frame: b} },
    {
      as_seen_by: { Frame: r} },
    { moment_action_point: as_seen_by_frame_origin },
    {
      coordinates: [ {f: [ {r.x: 1.2}, {r.y: -0.1}, {r.z: 3.2} ] },
                    {m: [ {r.x: 0.2}, {r.y: 0.8}, {r.z: -2.2} ] } ] },
    {
      units: newton, meter },
    {
      ID: Coordinatehd4if7 },
    {
      MID: Coordinate_Data_Structure },
    {
      MMID: { Coordinates, Euclidean_space,
            Force, Frame, QUDT } }
  }
}

```

Table 4.2: Example of a `Coordinates` model of the Force on a Frame "b" with respect to a Frame "r", and using the `screw` representation.

4.3.1 Coordinates for dynamics state

`Coordinates` representations for these forces are formally very similar to, but obviously semantically different from, the `Coordinates` representations for `Velocity` and `Twist`:

- **Cartesian Force**: these `Coordinates` have already been introduced in Sec. 4.2.1, as the **dual** concept of the Cartesian `Twist`.
- **joint state space torque or force**: many robot designs have motors that act on the same mechanical axis as the one that realises the 5D motion constraint, or equivalently, that drives the 1D motion freedom. So, the Cartesian `Force` that is generated by the motor to drive the relative `Motion` between the two `Links` constrained by the mechanical axis is faithfully represented by one *scalar* number, called (joint space) **torque** or **force**, for rotational or linear motors, respectively.

The formalizations of these `Coordinates` are analogous to those of `Motion`, just differing in the relevant semantic tags, see Table 3.2.

4.3.2 Coordinates and behavioural state for impedance

“**Impedance**” is the collective term for the mechanical mappings of mass, elasticity and damping. Their **models** are easily composed with the `Kinematic_chain` model:

- one `Attachment` is added to a `Link`, to represent the abstract data structure of the `Link`’s rigid body **mass**. That abstract data structure is *always* a **matrix** (“inertia tensor”, “mass matrix”,...), because the mapping between `Velocity` and `Force` is always *linear*, as is Newton’s Law that connects `Acceleration` to `Force`.
- one `Attachment` is added to each of two different `Links`, to serve as connection arguments in elasticity and damping relations between the two `Links`.

- the mapping relations of *elasticity* (between **Displacement** and **Force**) and of *damping* (between **Velocity** and **Force**) are **higher-order relations** that are typically *non-linear* functions of the **state** of the **Motion**. In practice, their **linear approximations** in the form of a **stiffness matrix** (and **Hooke’s Law**) or a **damping matrix** are popular.

The formal **Coordinates** representations of all of the above semantic primitives are straightforward, either as models of matrices, or as symbolic expressions of mapping functions.

(TODO: make formal models explicit, with examples.)

4.3.3 Geometrical operations revisited — The role of “virtual dynamics”

It seems logical to decouple the geometrical and dynamical meta models of kinematic chains, because the models of masses, springs and dampers (introduced in this Chapter) can be composed onto the geometric models (of Chap. 3) via **Attachment** primitives. But dynamical relations do occur already at the geometric level, albeit in disguise:

- whenever the specification of a (forward, inverse, or hybrid) kinematics transformation does not lead a unique solution, **redundancy** and **singularity** relations are to be introduced to determine the relative magnitude of the contribution of various components to the solution of the problem. Such trade-offs are typically realised by introducing “weighing” or “cost” functions with the components as arguments.
- the mathematical formulation of these weighing and cost functions reveals that they *must* have the physical dimensions of mass or elasticity.

Hence, the behavioural meta model for kinematic chains couples the geometrical and (mechanical) dynamical entities and relations together, all the time; the difference at the geometrical level of abstraction is that the impedances introduced by the solvers are artificial and arbitrarily chosen by the solver programmers; that is, they typically do not correspond at all to the real mechanical properties of the kinematic chain under consideration.

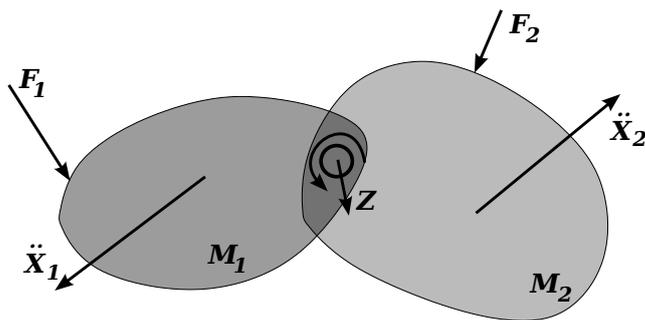


Figure 4.7: The entities involved in the dynamic behaviour of a (revolute) joint constraint. The spatial direction (Z), force (F) and acceleration (\ddot{X}) have *six* degrees of freedom; the mass (M) is a 6×6 relation between them.

4.3.4 Dynamic relations under a 5D motion constraint

The simplest possible **serial composition** in a kinematic chain has *one* single bi-directional **Joint** between two **Links**, and that **Joint** has only *one* degree of motion *freedom*. In other words, the **Joint** *constrains* the two connected **Links** in five degrees of freedom, that is, the generated constraint forces span a five-dimensional space. Figure 4.7 sketches the components involved in the dynamical behavioural relations for such a **Joint**. (The Figure shows an

unactuated revolute joint (that is, there is no motor present on the mechanical joint axis), but adding a joint torque is obvious. (TODO)) The modelling for other types of Joints uses similar entities, relations, and constraints.

The forces \mathbf{F}_1 and \mathbf{F}_2 on both links are connected to their accelerations $\ddot{\mathbf{X}}_1$ and $\ddot{\mathbf{X}}_2$, their inertias \mathbf{M}_1 and \mathbf{M}_2 , and to the Vector \mathbf{Z} representing the joint axis, by the following **dynamical constraint relation** (expressed as six-vector *mathematical* representations, and not *coordinate* representations):

$$\mathbf{F}_1 = \mathbf{M}_1^a \ddot{\mathbf{X}}_1, \quad (4.14)$$

$$\text{with } \mathbf{M}_1^a = \mathbf{M}_1 + \underbrace{\left(\mathbf{I} - \mathbf{Z} (\mathbf{Z}^T \mathbf{M}_2 \mathbf{Z})^{-1} \mathbf{Z}^T \right)}_{\mathbf{P}} \mathbf{M}_2. \quad (4.15)$$

\mathbf{M}_1^a the so-called *articulated body inertia* [37], i.e., the increased inertia of Link 1 due to the fact that it is connected to Link 2 through an “articulation”, that is, the revolute joint in this case. \mathbf{M}_1^a of Link 1 is the sum of its own inertia \mathbf{M}_1 and the **projected** part \mathbf{P} of the inertia of the second Link. The term “projection” is adequate because the matrix \mathbf{P} satisfies the conditions for being a **projection matrix**, namely the **idempotence** relation $\mathbf{P}\mathbf{P} = \mathbf{P}$. These algebraic properties are those of a weighted **Moore-Penrose pseudo-inverse**, [72].

4.4 Meta model for instantaneous motion of kinematic chains

Figure 4.8 (left) gives a schematic overview of all the variables needed to represent the **state** of a kinematic chain: position, velocity, acceleration and force, in each of the joints as well as of each of the links in the chain. Of course, all these variables are interconnected by the geometric and dynamics constraint relations introduced by the joints, gravity, and/or contacts with the environment. These constraints are sometimes called the **natural** constraints, because they are introduced by the physical world and are to be satisfied all the time. In addition to the natural constraints, robot controllers introduce **artificial** constraints, by means of artificially chosen forces and/or acceleration energy constraints at the joints and/or the links.

4.4.1 Gauss’ Principle of Least Constraint

Several centuries of research on the equations of motion of mechanical systems have led to the insights that *all* natural and artificial constraints on the **instantaneous motion** of (a connected or not-connected set of) rigid bodies can be formulated by one single theory, namely **Gauss’ Principle of Least Constraint** [42, 80, 101, 102]. The consequence of this Principle is that there exist three, and only three, possible ways to change the **instantaneous motion state** of a kinematic chain (Fig. 4.8, right):

- joint space forces and/or torques on the Links, generated by the actuators at the Joints. These are sometimes called the “**posture control**” torques/forces, because a major use cases for them is to control the internal posture of the kinematic chain.
- Cartesian Forces on the Links, caused by “active” force sources, or resulting from “passive” contacts with the environment.

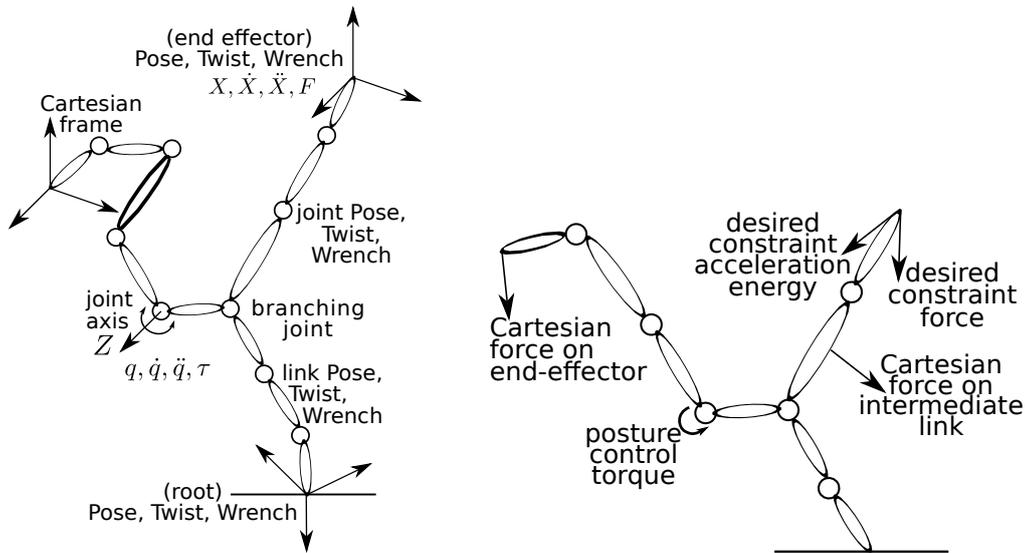


Figure 4.8: The entities in a geometric and mechano-dynamical model of a kinematic chain. On the left, all the type of variables that make up the *state* of the chain. On the right, the three possible types of *drivers* for the instantaneous motion (*change of state*) of a chain: (i) a torque generated by a motor at a **Joint**; (ii) a Cartesian force at a particular **Attachment** on a **Link** (with or without a mechanical contact at that **Attachment**); and (iii) a constraint on the allowable Cartesian acceleration energy at a particular **Attachment** on a **Link**.

- **constraints on the Cartesian acceleration** of Links. The constraint function to optimize in this case is the **acceleration energy** \mathcal{Z} (from the word “Zwang” in the original German literature). \mathcal{Z} is the sum of all terms of the form **acceleration times mass times acceleration**, which is the “second-order” version of the similarly expressed *kinetic energy*.

4.4.2 Specification of instantaneous motion

Because the physics of mechanical dynamics provides three complementary ways to make a kinematic chain move, it is appropriate to introduce a meta model to represent these physical facts as **instantaneous motion drivers** for robots. The formalization of Cartesian and joint space forces has already been introduced in earlier Sections, but a meta model for an acceleration constraint needs a bit more work. The solution uses the approach of **Lagrange multipliers**:

- the constraint is modelled indirectly, by the introduction of **unit constraint forces** that counteract any acceleration in the constrained directions.

Figure 4.9 gives some examples of how Cartesian motion constraints on a **Link** can be represented as a collection of constraint **Forces** on the **Link**.

- the **acceleration energy** that the constraint forces are allowed to generate, needs to be specified.

In the most common case, the allowed acceleration energy will be zero, representing the desire not to have any acceleration in a particular direction at all.

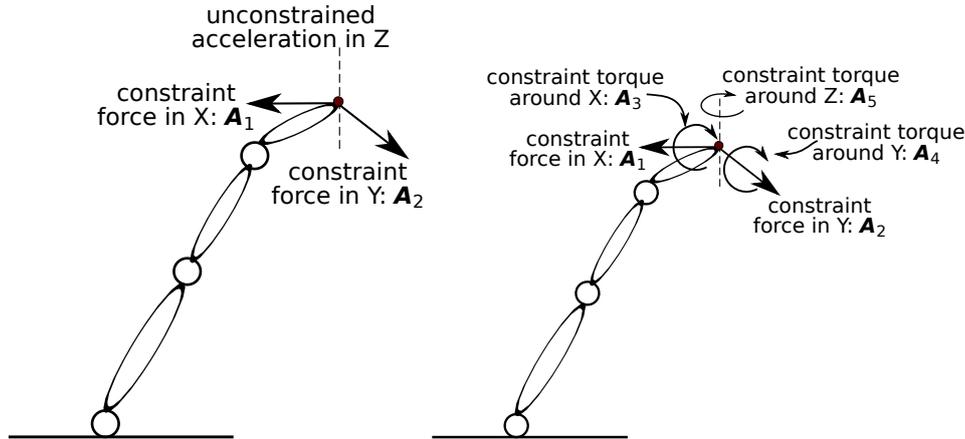


Figure 4.9: Two examples of artificial motion constraint specification via acceleration constraints. On the left: the last `Link` of the chain is constrained to have one of its `Points` move on a vertical line (irrespective of the orientation of the `Link` with respect to that line), by introducing two constraint forces that push the `Point` towards the line whenever it deviates from it. The right-hand specification adds constraint torques around all three orientation directions, in order that also the full orientation of the `Link` remains unchanged during the motion.

- the **magnitudes** of the constraint forces are then to be computed by solving the constrained dynamical equations.

Section 4.8 explains the algorithmic foundations for such solvers.

In practice, acceleration constraints are not popular to specify motions, and velocity constraints are used more often (and sometimes even position constraints). However, because only acceleration-level constraints have physical meaning, the velocity and position specifications constraints only make sense when composed with the acceleration-level physics via a *constraint controller* [11].

The formalisations of forces and accelerations have already been introduced in Chap. 3, which leaves only `acceleration_constraint` and its `acceleration_energy` as the new semantic relations to be modelled. The mereo-topological formalisation of a **instantaneous motion specification** of a `Kinematic_chain` then becomes a simple composition of the models of the `Kinematic_chain` with:

- the `Attachments` to connect motion drivers to.
- the motion driver specified in each `Attachment`.

Table 4.3 gives an example of such a mereo-topological specification model. The extension with abstract data type models is done in Sec. 4.8.

4.4.3 Operations: forward, inverse and hybrid dynamics transformations

A specification of the drivers for an instantaneous motion of a `Kinematic_chain` is equivalent to specifying an **operation** on the chain, namely the transformation between the forces represented in the motion drivers and the forces one, and instantaneous (change in) motion

```

{ A_motion_specification :
  { {
      ID : Motion_Spec_ID_XX64Hy },
    {
      MID : [ HybridDynamicsMotionDrivers, KC_ID_123XYZ ] },
    {
      MMID : Kinematic_chain }
  { motion_driver : [ { { attachment : Link_1_Att_2_ID },
                      { force : joint_torque_ID_34df } },
                    { { attachment : Link_4_Att_1_ID },
                      { acceleration_constraint :
                        { dimension: 2 },
                        [ { unit_constraint_force :
                          { ID : CF_1 },
                          { attachment : Link_4_Att_1_ID.x } },
                        ],
                        { unit_constraint_force :
                          { ID : CF_2 },
                          { attachment : Link_4_Att_1_ID.y } },
                        ] } } }
  }, ] }, } ] } }
}

```

Table 4.3: Example of `Motion_driver` model, with one joint torque and one acceleration constraint (in the X and Y directions of a `Frame` attached to a `Link`. The symbols used in the model refer to the `Kinematic_chain` with ID "KC_ID_123XYZ", Table 4.1.

of, each of the `Links` in the chain. For now, this transformation is still *implicit*, since one needs a `solver` to make the transformation explicit. A solver that accepts all forms of motion drivers is sometimes called a **hybrid dynamics solver** [37]. When only joint torques are provide as inputs (together with the `Motion` state of the `Kinematic_chain`), the transformation is known under the name of **forward dynamics**. Similarly, in the **inverse dynamics** formulation, only the Cartesian forces are provided as inputs, together with the `Motion` state of the `Kinematic_chain`.

4.5 Composition and decomposition of `Kinematic_chains`

The **primitive** `Kinematic_chain` has one single `Joint` connecting one `Attachment` on each of two `Links`. Compositions of this primitive result, in general, in a **graph** of interconnections, and this Section describes the entities and relations pertaining to such graphs of interconnected `Joints`.

4.5.1 Composition — Serial, branch, loop

The following categories of **composition relations** are relevant:

1. `serial_composition` (Fig. 4.10): *either a primitive* `Kinematic_chain`, *or* an already existing `serial_composition` to which one connects an extra `Link`, via a `Joint` to a `Link` in the `Kinematic_chain` that has already only *one* other `Joint`.

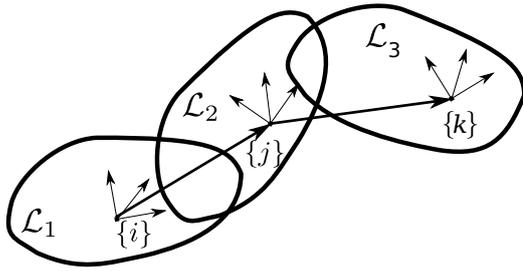


Figure 4.10: Serial composition of kinematic chains.

The result is a **strict order** of all Links, Attachments and Joints in the composite Kinematic_chain. The first and the last Link in this list get the semantic tag `leaf_link`. The order does not have an absolute *sense*, in that there is no reason to call one of the `leaf_links` the “first” and the other the “last”. This *sense* is often added by models that compose the Kinematic_chain into a particular context, where that sense has a natural meaning. For example, it is natural to give one `leaf_link` in the industrial robot arm of Fig. 4.2 the tag “0” (the one that is the “base” of the robot, bolted to the ground), and count up from there till the other `leaf_link` (the “end-effector” of the robot, to which `tools` are connected).

Serial composition does not require new semantic operations; the composition relation of the *mass matrices* of two serially connected Links has already been introduced in Sec. 4.3.4.

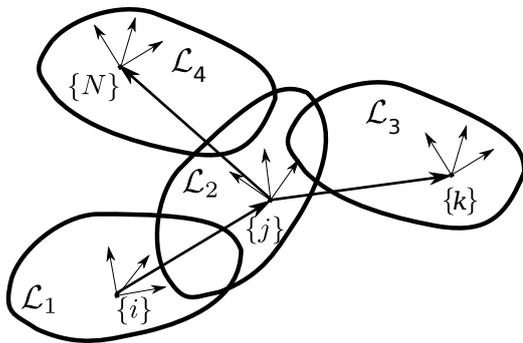


Figure 4.11: Branch composition of kinematic chains. The composite chain has three branches, and Link \mathcal{L}_2 is the `branch_link`.

2. `branch_composition` (Fig. 4.11): the connection of one Kinematic_chain to another not yet connected Kinematic_chain via a Joint between (i) a `leaf_link` in the former Kinematic_chain, and (ii) a non-`leaf_link` in the latter Kinematic_chain.

Both Links lose their `leaf_link` tag; the latter Link gets the `branch_link` tag instead (in case it does not already have that tag). Each of the connected Kinematic_chains is a `branch` of the composite Kinematic_chain.

Branch composition requires one new semantic operation, namely the composition the *mass matrix* of a branch Link and the mass matrices of two of its branches. This composition is linear, so the Coordinates of the mass matrices can be added, as soon as all their semantic tags are identical, that is, all coordinates use the same velocity reference point, the same as-seen-by reference, the same physical units, the same ordering of linear and angular parts.

3. `loop_composition` (Fig. 4.12): this is a composition in which a `leaf_link` is connected

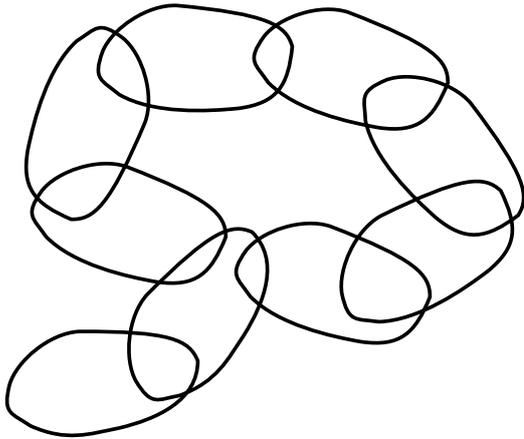


Figure 4.12: Loop composition of kinematic chains.

via a `Joint` to a different `Link` in the `Kinematic_chain` to which the `leaf_link` belongs. The `leaf_link` loses its semantic tag. The other `Link` loses that tag too, in case it was one before the connection; if it had already two or more `Joints`, it gets the semantic tag of `branch_link`.

Closing a loop has a non-trivial impact on the operation of composing the mass matrices. There is not one single way of realising this operation, because an arbitrarily chosen “cut” of the loop into two branches is required (Sec. 4.5.2), after which the mass matrix of the cut link must be divided over the two branches, the serial composition of mass matrices is to be applied to each branch, and the branch composition of mass matrices has to be applied at the branch link of the cut loop.

Of course, higher-order composition relations are possible too; for example, loops within loops, like one finds in the `musculo-skeletal` structure of animals and humans.

4.5.2 Decomposition — Spanning tree

Many applications need to work with only those parts from a `Kinematic_chain` that are relevant to the applications’ tasks; for example, only one arm of a `two-arm mobile manipulator` is needed to grasp an object. The semantic relation of the `Spanning_tree` supports such decomposition of a `Kinematic_chain` into sub-chains. It composes a model of a particular kinematic graph with another kinematic graph model (the so-called “`Spanning_tree`”) to **select a particular view** on the original graph, and has the following properties:

- *tree*: the `Spanning_tree` graph is composed of only serial chains connected at branching links, and each of them are sub-graphs of the original graph. For example, one way to map a kinematic graph onto a `Spanning_tree` is by cutting all of its loops.
- *semantic tags*: each serial sub-chain and each branch point get a `Semantic_ID` that identifies them as part of the original graph and as part of the `Spanning_tree` derived from it.

The model of the `Spanning_tree` stores the information about which cuts were made, and adds an extra `Attachment` at both ends of the cut. This allows to add **loop closure** relations, that mathematically represent the information about how to close the original loops again,

or, equivalently, how the state of a `Spanning_tree` violates the motion constraint relations of the original kinematic graph.

One particular `Kinematic_chain` model can be mapped onto multiple `Spanning_tree` models at the same time. The concept of the `Spanning_tree` is relevant for all topological models, with serial, tree as well as graph topologies.

4.5.3 Policy: iteration via sweeps

A tree topology structure simplifies not only the mechanical construction of a robot, but also the computational solvers needed to make computations with its kinematic state. For example, a tree topology has only one single path between any two of its nodes, which simplifies computational *iterations* over lists of `Links`, `Joints` and `Attachments`. The *hybrid dynamics solver* makes extensive use of different sweeps.

The symbolic, model-centric equivalent of the computational iterator over a data structure is the *database cursor*, to make *graph traversals* over a property graph more effective when one has to solve a series of queries on the same graph, i.e., the same `Kinematic_chain`.

For the above-mentioned reasons, the majority of commercially used robot mechanisms *have* already a tree topology, and even the simplest form of a tree, a serial topology. For similar reasons, the majority of numerical solvers are designed to work on tree topologies only; that means that applications that need real graph topologies must compose their task specification with a `Spanning_tree` policy.

4.5.4 Policy: input-output causality assignment

The meta models represent *relations* between `motion` entities, which are the **a-causal** *mechanism* describing how the properties of the various entities in the motion are related. Most applications require causal relations, or *input-output functions* as they are called more often, with the following **arguments**:

- a *model* of the kinematic chain;
- the *list* of geometric primitives which are *given* as inputs; and
- the *list* of geometric primitives which must be *computed* as outputs.

One single a-causal relation gives rise to many conforming input-output functions, one for each combination of input and output choices.

4.6 Taxonomy of kinematic chain families

From a modelling point of view, it makes sense to introduce “families” of models, as a simple mereological higher order model for **classification**, because (i) the kinematic chain structure of most robots falls within one such category, and (ii) each category has a specific numerical solver, optimized for the particular geometric properties of the family. Concretely speaking, a `Kinematic_family` relation collects the constraints that are shared between all members of the family, such as: the number of joints, the type of the joints, the singular configurations, and the abstract data types representing joint space and Cartesian space state.

This Section introduces the top-level members of the `Kinematic_family` *taxonomies*.

4.6.1 Serial chains

Serial kinematic chains, or “arms” (Fig. 4.13):

- Kinematic_family::Serial-321: traditional industrial robots
- Kinematic_family::Serial-3-X: many “cobots” like Universal Robot, or Mabi.
- Kinematic_family::Serial-313: KUKA iiwa, ABB YuMI,...
- Kinematic_family::SCARA
- snake or “elephant trunk” family.

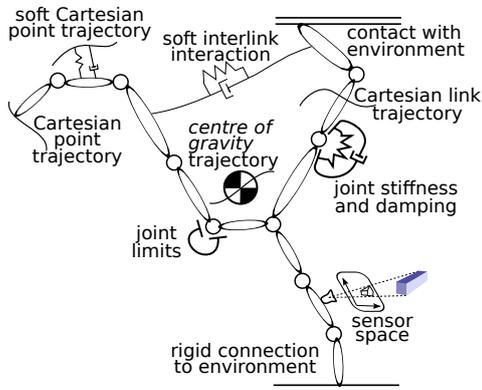


Figure 4.13: Sketch of the kinematic chain model of a dual-arm manipulator, with all(?) possible motion constraints on links and joints.

4.6.2 Mobile platform chains

- DifferentialDrive
- Holonomic:
- EightWheelDrive: (Fig. 4.14)

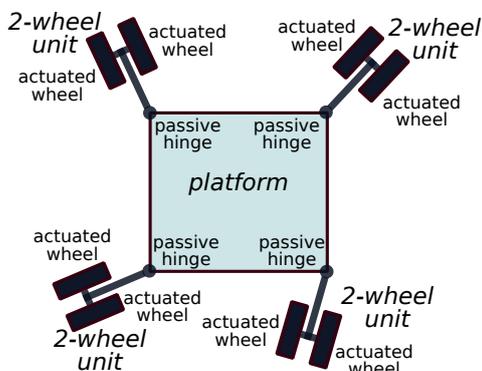


Figure 4.14: Sketch of the kinematic chain model of an over-actuated mobile robot: eight actuated wheels, connected in so-called 2WD (“two-wheel drive”) pairs, that in turn are connected to a rigid platform via passive revolute joints with a caster offset. The design drivers behind this platform are (i) passive backdrivability in all configurations, and (ii) holonomic motion behaviour in all configurations.

4.6.3 Parallel chains

- **Delta:**
- **Stewart-Gough:**

(Fig. 4.15):

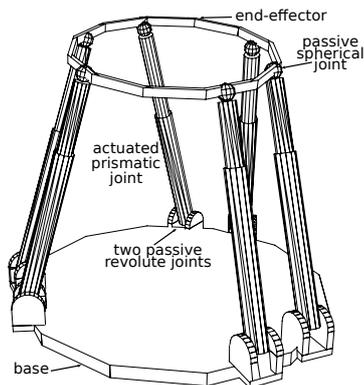


Figure 4.15: Sketch of the kinematic chain model of a parallel kinematic chain, with six legs each with six 1D joints.

4.6.4 Multirotor chains

4.6.5 Hybrid chains

featuring one or more “loops” in the chain’s topology.

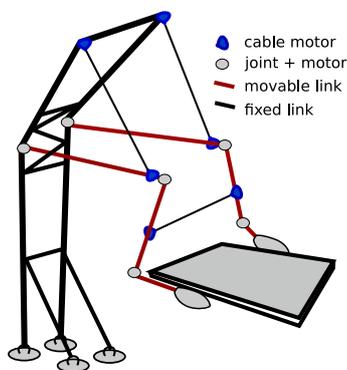


Figure 4.16: Sketch of the kinematic chain model of a cable-driven robot, with a “hybrid” topological structure.

4.6.6 Cable-driven chains

Fig. 4.16,

4.7 Taxonomy of motion capabilities

Most existing robot systems have intended functionalities that are the **composition** of two or more of the following:

- **mobility:** to take care of the **locomotion** towards targets over the earth, driven by wheels, legs, propellers (in air as well as in water),...

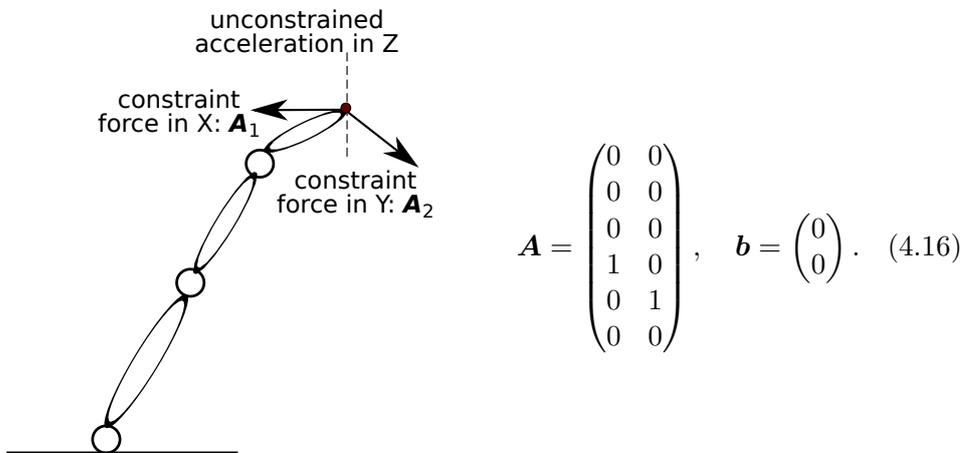
- **balance**: the extra motion capabilities to do any combination of the above capabilities, at the same time, while keeping the robot in a desired range of configurations that the task context considers to be “in equilibrium”.
 - **reach** (approach, accede, enter,...): the “arms” or “manipulator” navigate towards targets in local 6D space.
 - **orient**: the “wrist” can direct grasped objects locally in various directions.
 - **grasp** (encompass, lift, support, hold,...): the “gripper” or “hands” to manipulate objects by *force closure*, *form closure*, or *non-prehensile* holding.
 - **touch**: to sense and manipulate by *form features* such as nails, finger tips, or whiskers.
- The robotics domain has (informally) introduced *semantic tags* like “**mobile manipulators**”, “**humanoids**”, “**gantries**”, or “**welders**”, as specific compositions in particular application domains. How *to specify* the desired motion behaviour of a kinematic chain brings in the application’s task context, and is the subject of Chapter 5

4.8 Solver meta model for hybrid kinematics and dynamics

Section 4.3 introduced the entities and relations needed to specify the instantaneous motion of an ideal kinematic chain. This Section adds the information about the functions and **abstract data types** needed to create a **solver algorithm** that computes such an instantaneous motion from the model and its specification.

4.8.1 Mechanism–I: motion drivers

Section 4.4 introduced the mereo-topological models for the three different “motion drivers” with which to specify the instantaneous motion of a kinematic chain: joint torques, Cartesian forces, and Cartesian acceleration energy constraints. Acceleration energy is represented as a product of acceleration and inertia, whose formalisation with **abstract data types** gets the form $\ddot{\mathbf{X}}^T \mathbf{M} \ddot{\mathbf{X}}$; or, equivalently, of force and acceleration, of the form $\mathbf{F}^T \ddot{\mathbf{X}}$. In the Figures below, the matrix \mathbf{A} represents the matrix of the constraint force basis, that is, the set of “unit” versors along the spatial directions in which the acceleration constraints can generate constraint forces. For example, to constrain the motion of a reference *point* on the segment *partially* in the vertical direction, the constraint matrices can be chosen as follows:



The columns of \mathbf{A} are constraint forces in the horizontal X and Y directions, that must keep the acceleration in those directions to zero; the three rows of zeros on the top indicate

the absence of acceleration constraints on the rotational degrees of freedom. The \mathbf{b} vector is used in a *motion task specification*, indicating that one wants zero acceleration energy to be generated/consumed in the constrained directions.

A second example is about moving the segment vertically *without allowing rotations*. Hence, the constraint matrix \mathbf{A} now represents five constraint forces:

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}. \quad (4.17)$$

That means that the constraining forces and moments are allowed to work in all directions, except the vertical Z direction.

Here is the “traditional” case of giving the segment a *desired acceleration energy* \mathbf{b}_d in full 6D:

$$\mathbf{A} = \mathbf{1}_{6 \times 6}, \quad \mathbf{b} = \mathbf{b}_d. \quad (4.18)$$

4.8.2 Mechanism–II: procedural sweeps

All of the solver algorithms introduce an ordering in their computations, constrained by the structure of the kinematic chain. These control flow *schedules* are commonly referred to as “sweeps”; Fig. 4.17 depicts the three sweeps in the generic example of a tree-structured kinematic chain. Algorithm 1 summarizes the computations³ that provide the “inverse dynamics” of a serial robot, bolted to the ground, with N joints, one external force, and one acceleration constraint.

The *core properties* of such hybrid dynamics solvers are that:

- there are three **drivers** in the hybrid dynamics solver that can make the kinematic chain move; these are visible in the second last line in the solver Algorithm, via (i) *joint torques* $\boldsymbol{\tau}$, (ii) the *external forces* \mathbf{F} on links, and (iii) the Lagrange multipliers $\boldsymbol{\nu}$ generated by the *constraint acceleration energies* \mathbf{b} .
- when a *Cartesian force* is given as *input*, the chain’s *motion* (first of all, acceleration, but via simple integration also velocity and position) is computed as an *output* of the solver. It is often appropriate to introduce a *monitor* in the third sweep, to check whether that computed motion is not violating any specified task constraints.
- when an *acceleration constraint* is given as *input*, the resulting *force* is computed as an *output*. As in the previous case, a *monitor* can be introduced in the third sweep to check whether this computed force is not violating any specified task constraints.
- the chain has a **dynamic singularity** if the articulated mass matrix projection goes to infinity, that is, one or more of the D scalars is close to zero.

³The bookkeeping of the indices is not yet fully consistent in the presented pseudo-code. . .

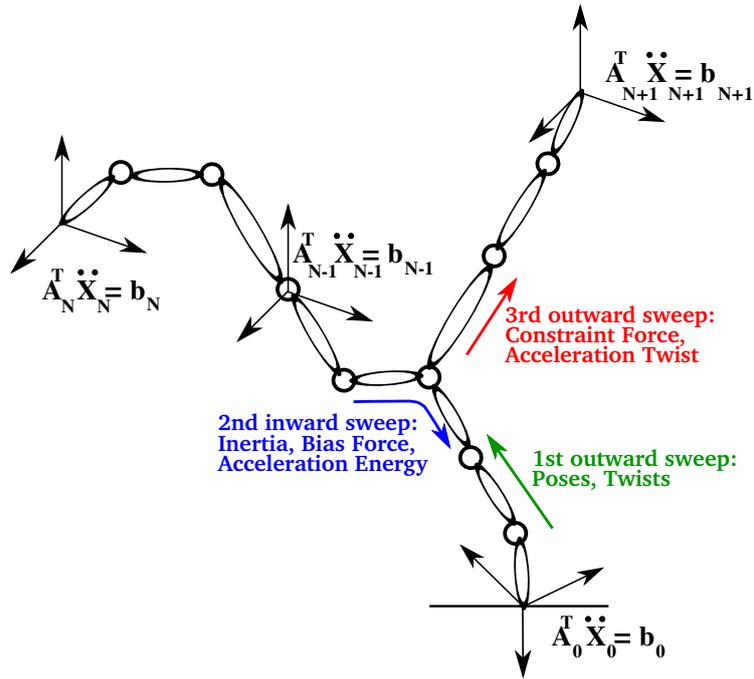


Figure 4.17: A hybrid dynamics solver uses the topological structure of the kinematic chain to schedule all the behavioural functions required to answer a particular query.

- the τ_i are the (only) coupling with the physical actuator dynamics, where electrical power consumption or torque limits have to be specified, monitored, and/or optimized.

4.8.3 Policy: scheduling options in the third sweep

- when solving for the Lagrange multipliers ν , one can **weigh** the various acceleration constraint drivers.
- **prioritization** between the three types of drivers can be done by sequentially scheduling third sweeps for each of them separately: later third sweeps “win” over earlier ones.
- the effect of gravity can be computed separately from the effects of the motion drivers.
- joint torques caused by **joint friction and/or elasticity** can be added to the τ_i drivers, as *feedforward* functions.
- after the second sweep, one has **factored** the whole kinematics and dynamics in pieces that can be (linearly) **composed** together in **various** ways in third sweeps.

For example, one could do a linear search to find the acceleration energy driver \mathbf{b} that gives a desired constraint force (via a re-solving of the linear system of equations connecting the \mathbf{b} to the Lagrange multipliers ν), or, conversely to find a Cartesian force \mathbf{F} that generates a joint torque with a desired magnitude and sign, or one that just does not saturate the joint actuators.

Algorithm 1: Hybrid dynamics solver, according to Popov-Vereshchagin [102]

```

begin
  // outward sweep, to compute the motion state:
  for  $i \leftarrow 0$  to  $N - 1$  do
     ${}^{p_i}T = {}^{d_i}T {}^{p_{i+1}}T(q_i)$  ;
     $\boldsymbol{\omega}_{i+1} = \boldsymbol{\omega}_i + \dot{q}_{i+1} \mathbf{Z}_{i+1}$  ;
     $\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{r}^{i+1,i} \times \boldsymbol{\omega}_i$  ;
     $\ddot{\mathbf{X}}_{b,i+1} = \begin{pmatrix} \dot{q}_{i+1} \boldsymbol{\omega}_i \times \mathbf{Z}_{i+1} \\ \boldsymbol{\omega}_i \times (\mathbf{r}^{i+1,i} \times \boldsymbol{\omega}_i) \end{pmatrix}$  ;
  // inward sweep, to compute the force and acceleration factorization:
  for  $i \leftarrow (N - 1)$  to 0 do
     $\mathbf{P}_{i+1} = \mathbf{1} - \mathbf{M}_{i+1}(\mathbf{Z}_{i+1}^T \mathbf{M}_{i+1}^a \mathbf{Z}_{i+1})^{-1} \mathbf{Z}_{i+1}^T$  ;
     $\mathbf{M}_i^a = \mathbf{M}_i + \mathbf{P}_{i+1} \mathbf{M}_{i+1}$  ;
     $\mathbf{F}_i = \mathbf{P}_{i+1} \mathbf{F}_{i+1} - \mathbf{M}_{i+1}^a \mathbf{Z}_{i+1} (\mathbf{Z}_{i+1}^T \mathbf{M}_{i+1}^a \mathbf{Z}_{i+1})^{-1} \tau_{i+1} + \mathbf{F}_i^b + \mathbf{F}_i^e$  ;
     $\mathbf{A}_i = \mathbf{P}_{i+1} \mathbf{A}_{i+1}$  ;
     $\boldsymbol{\beta}_i = \boldsymbol{\beta}_{i+1} + \mathbf{A}_{i+1}^T \left\{ \ddot{\mathbf{X}}_{i+1} + \mathbf{Z}_i D_i^{-1} \left( \tau_{i+1} - \mathbf{Z}_i^T (\mathbf{F}_{i+1} + \mathbf{M}_{i+1}^a \ddot{\mathbf{X}}_{i+1}) \right) \right\}$  ;
    with  $D_i = \mathbf{Z}_i^T \mathbf{M}_i^a \mathbf{Z}_i$ , and  $\boldsymbol{\beta}_N = 0$  ;
     $\mathbf{Z}_i = \mathbf{Z}_{i+1} - \mathbf{A}_{i+1}^T \mathbf{Z}_i D_i^{-1} \mathbf{Z}_i^T \mathbf{A}_{i+1}$ ,  $\mathbf{Z}_N = 0$  ;
  // Lagrange multipliers of acceleration constraint forces:
   $\mathbf{Z}_0 \boldsymbol{\nu} = \mathbf{b}_N - \mathbf{A}_0^T \ddot{\mathbf{X}}_0 - \boldsymbol{\beta}_0$  ;
  // outward sweep to compute joint torques and link accelerations:
  for  $i \leftarrow 1$  to  $N$  do
     $\ddot{q}_i = (\mathbf{Z}_{i-1}^T \mathbf{M}_i^a \mathbf{Z}_{i-1})^{-1} \left\{ \tau_i - \mathbf{Z}_{i-1}^T (\mathbf{F}_i + \mathbf{M}_i^a \ddot{\mathbf{X}}_{i-1} + \mathbf{A}_i \boldsymbol{\nu}) \right\}$  ;
     $\ddot{\mathbf{X}}_i = \ddot{\mathbf{X}}_{i-1} + \ddot{q}_i \mathbf{Z}_i + \ddot{\mathbf{X}}_{b,i}$  ;

```

4.8.4 Policy: free-floating base

(TODO)

4.8.5 Policy: tasks in the mechanical domain

This Section explains how to configure the properties of the different sweeps in a hybrid dynamics solver to satisfy various types of mechanical Tasks, i.e., motions.

(BEGIN TODO:

- How can one let the robot do two (or more) tasks at the same time, and give relative priorities to the different tasks?
“Task” means: instantaneous motion, via force and or acceleration.
- How can the hybrid dynamics solver be used on a robot which does not have a torque control interface, but only a velocity control interface?
- What is the dynamic equivalent of a *kinematic singularity*?

Remember: at the velocity level, and for a serial robot, a singularity was defined as a joint space configuration in which the Jacobian matrix loses rank; or where some Cartesian forces require no joint torques to be supported; or where some joint space velocities result in no Cartesian velocity.

- Is it possible that acceleration constraints are not consistent with each other? How would one find out? What can one do about this situation?
- How can the algorithm be used to model the propulsion of a ship?
- How can one find out whether joint limits are violated? What can one do about this situation?
- Give an example where applying a force somewhere on the robot will not make the robot accelerate in that direction. What can you do to guarantee such minimum acceleration? How is this guarantee dependent on joint limits?
- How can one separate the parts of the joint torques that contribute to compensate gravity from those that work against the inertia of the robot to make it move?
- How can you find out to what extent two instantaneous motion specifications on the same kinematic chain are (in)consistent?
- How can you find out whether one could regenerate energy in a particular joint motor?

END TODO)

4.8.6 Policy: serial kinematic chain

(TODO: queries for forward, inverse, hybrid transformations; motion, force and solving sweeps.)

4.8.7 Policy: branched kinematic chain

(TODO: queries for forward, inverse, hybrid transformations; motion, force and solving sweeps.)

4.8.8 Policy: kinematic chain with a loop

(TODO: queries for forward, inverse, hybrid transformations; motion, force and solving sweeps.)

Chapter 5

Meta models for robot tasks

Users of robotic systems expect these systems to have the **capability** to execute the quasi infinite amount of types of **tasks**, that they know are possible with the specific type of **resources** available to the robot, and in the specific type of **environment** in which the robot and its task-related **objects** live.

This document **structures** this infinite task design space for *systems*—and the **task specifications** that are relevant there—as the composition of task designs for sub-systems that can not be made any smaller without losing the meaning of “*being responsible for a task*”. Such tasks can be the ones the system’s customers pay for (e.g., the assembly of a product), but their realisation requires the execution of lots and lots of smaller, more invisible, tasks, such as perception and control algorithms, condition monitoring, kinematic transformations, etc.

A task specification is a model of how to use the robot’s **capabilities**, to change the **actual state of the world**, into a **desired state (of information)** of the world. The granularity of entities and relations in a *task specification* of a (sub-)system depends on the choice of entities and relations in the *world model* available in that (sub-)system.

Because each sub-system has a different view on what the world is, this Chapter defines a **task model** of a *system to conform to* the **composition** of task models of *sub-systems* by means of the following complementary **meta models** of **mereotopological** type:

- the **Task-Skill-Resource** meta model represents the **knowledge** to couple all perception, control and decision making activities in a system of systems.
- the **perception for task control** meta model represents the **hierarchy** in the **composition** of “control”: the proprio-ceptive control, extero-ceptive control, and cartho-ceptive control of one single robot system, and the **shared** perception and control of human-robot and robot-robot interactions.
- the **constrained optimization** meta model represents a common approach to specify a task.

The relevant meta models of **motion**, **perception**, and **control** are introduced in other Chapters. A task model *depends* on these models, but also the *inverse dependency* holds: the requirements of the robot’s task influences how the robot is going to move, to perceive and to control.

The **capabilities** model of a robotic system is then the composition of models from several of the above-mentioned structures. Such a capabilities model serves as a **data sheet** of a robotic system’s behaviour, and even as a **contract** between providers and customers of the robotic system.

5.1 Continuous, discrete and symbolic task models

One fundamental hypothesis in this document is that **all** formal representations of tasks, and for **every** concept they represent, contain¹ entities and relations of **all** of the following three complementary types:

- **continuous**: the relations are continuous functions of parameters in the system’s entities, such as the time, space, force, effort, cost, . . . aspects of a task.

For example, the [kinematic and dynamic equations](#) of a robot’s mechanical structure. Or the mathematics of a [PID controller](#).

- **discrete**: the relations are discrete functions of the systems’ entity parameters, that is, they represent the conditions when, why and how to select the “right” continuous system and task models.

For example, the logical decision making functions in [Petri Nets](#) or [Finite State Machines](#).

- **symbolic**: the relations encode non-measurable dependencies between continuous and discrete models, such as:

- insights that *explain* why to configure some [magic numbers](#) in the continuous and discrete models, and with which values.

For example, the maximum torque of an actuator can be limited because the required currents could overheat the wires, based on a thermal model of the motor.

- representation of the *intention* of a task.

For example, when making a left turn, a car driver may first move the car a bit more to the right of the lane, in order to later be able to make a smoother turn. Or, a dual arm robot hands over an object from the right hand to the left hand, because it will need the latter in a later part of the task.

- representation of the *context* of a task.

For example, the above-mentioned maximum motor torque could be lowered when a robot gets near a human, or when it enters a specific zone in a factory, in order to reduce the risk of damage in case of a collision. Or, a dual-arm robot may carry a load closer to its body, and move closer to its statically most stable posture, whenever that load is very fragile, risky, and/or expensive.

There is a clear hierarchy in these three types of relations:

- symbolic relations are [higher-order](#) relations on discrete and continuous entities and relations, but also on other symbolic relations.
- discrete relations are [higher-order](#) relations on continuous entities and relations, but also on other discrete relations.

¹Or rather, “*should contain*”, because almost no robotic systems nowadays already satisfy this hypothesis.

- continuous relations can be **higher-order** relations on other continuous entities and relations.

5.2 Use case: indoor navigation with a two-wheel driven robot

Because of the high level of structure that one finds in buildings, the indoor navigation of a mobile robot (of the **two-wheel driven** kind depicted in Figure 5.1) is possibly the simplest non-trivial application for which to provide examples that *ground* the concepts, entities and relations of this Chapter.

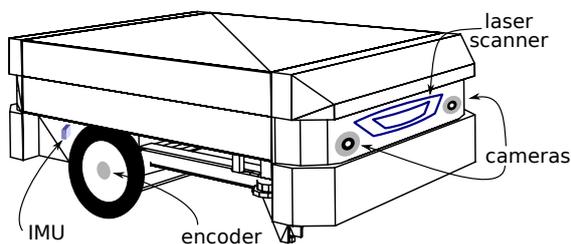


Figure 5.1: A popular hardware architecture of a two-wheel driven mobile robot: a **2D laser scanner** and two **cameras** are mounted at its front, encoders on its wheel motors, and an *Inertial Measurement Unit* on its body.

This Section gives some concrete examples of **task models**: what **resources** are available to specify tasks for, and what are the perception, control, monitoring, world modelling, . . . activities in such robots, at three common levels of **system boundaries**: **proprio-ceptive**, **extero-ceptive**, and **cartho-ceptive** tasks.

5.2.1 Resources

The following hardware is rather common in mobile platforms:

- the motors are **brushless DC** (BLDC) motors, using **field-oriented control** (FOC), embedded in a stand-alone **servo drive**. Hence, *constraints* come from **torque saturation**, **overheating** with too high currents, or **charge depletion** of the batteries.
- the **motor drives** are interfaced to an **industrial PC**, that runs a (possibly **real-time**) version of the **Linux kernel**. One of the PC's processes executes the tasks that the robot gets from its users, linking control and perception functionalities. Another process is a server (e.g., a **Node.js** instance), responsible for all networking with computers elsewhere in the local network; for example, to schedule and dispatch tasks for one or more robots, and to monitor the execution. There can be another process, for (possibly) **web browser-based** **graphical user interface** (GUI). Hence, *constraints* come from avoiding (i) to overload the available RAM, CPU cores, and disk space, and (ii) to violate control loop timing **latency**.
- the interfacing between drives and PC uses a real-time communication platform, such as a **CAN bus**, or an **industrial ethernet** protocol, e.g., **EtherCAT**. Normal ethernet is used for the communication between the PC and the “outside world”. *Constraints* come from runtime interfacing latencies.
- the robot has **rotary encoders** on its motors; an **inertial measurement unit** (IMU) and a **2D laser scanner** on its body, and two **colour cameras** embedded in its front **bumper**. The latter is equipped with **proximity/contact sensors**. The latter sensors are interfaced via

general purpose IO pins that are memory mapped; the encoders and IMU are interfaced via the EtherCat bus, that also provides an estimate of the motor current; the Lidar and cameras use USB.

Constraints come from discretization errors, random and systematic “noise”, and nonlinearities such as saturation or deviations from linear scale measurements.

A notable resource that is most often *not* present in indoor mobile robots is a mechanical suspension, let alone of the rocker-bogie type.

5.2.2 Proprio-ceptive task specification

The robot’s tasks only take its own body into account, so the world model consists of the relative positions of relevant points on the robot: motors, sensors, points on the body used in the task specification, . . . Example tasks are:

- *monitor* which motions the robot makes when actuating the motors. Actuation can be via torques or velocities. The motion can be represented by spline or clothoid curves of one or more points on the robot body.
- *control* the motion of the robot towards task goals. The specification of such goals could be as simple as involving only one single point of the robot (e.g., *move one meter further*), or as complex as involving multiple references on the robot and multiple configuration spaces (e.g., *keep the two corner points of the robot on its left side on a straight line when moving forward, while pushing with a maximal torque of XX Newtons at the front side of the robot*).

5.2.3 Extero-ceptive task specification

This level extends the world model with landmarks in the environment, that the robot is capable of *perceiving*, so that tasks can include them in their specifications, Extero-ceptive generalisations of the example tasks above are:

- *perceive the motions that the robot makes with respect to the corner of the corridor that is on the left in its direction of motion.*
- *control the mentioned pushing motion of the robot while keeping its left side at least YYcm away from the wall on its left.*

5.2.4 Cartho-ceptive task specification

This level adds landmarks on the map to the robot’s world model, that is, tasks can be specified using world model features that are not directly perceivable by the robot’s sensors. Cartho-ceptive generalisations of the example tasks above are:

- *estimate how long it will take for the currently ongoing motion of the robot to reach its destination two corridors away on the map.*
- *control the mentioned pushing motion of the robot while keeping its left side at least YYcm away from the wall on its left, and anticipating that someone could show up from the corridor intersection that is coming up next, because the map has tagged this corridor as “busy”.*

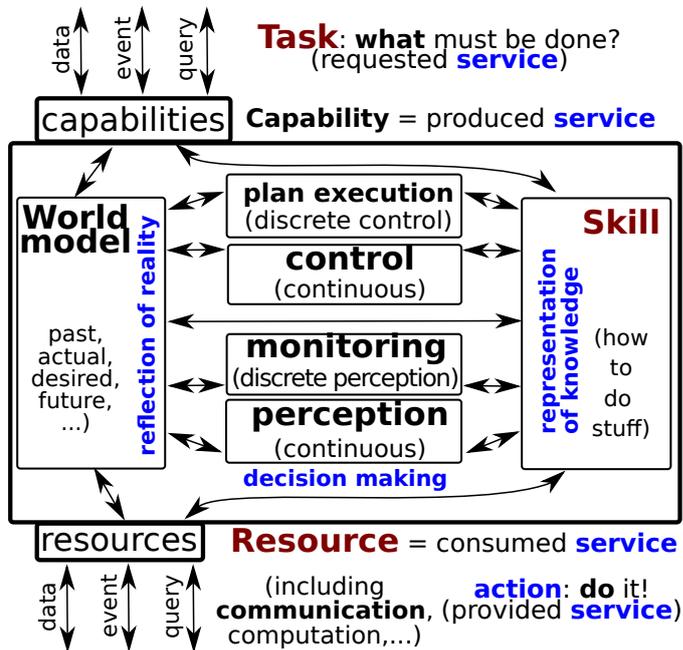


Figure 5.2: The mereo-topological part of the Task-Skill-Resource meta model. Its major relations are the ones that are *not* there: the essential activities in a robotic system (plan, control, perception and monitoring) **do not interact directly**. They interact indirectly via (i) information exchange with the *world model activity* about the *current* state of the world, and, (ii) decision making delegated at runtime to the *Skill activity* because that activity has the knowledge about the different ways how the robot can *detect* and *change* the state of the world.

5.3 Task-Skill-Resource: activities needed for task execution

This Section introduces the entities and relations (more concretely, the **activities** and their **interactions**) whose composition yields a **Task execution** model (Fig. 5.2). (A later Chapter provides a concrete *architecture* for the actual execution of the *declarative model* of this Section.) The following list of activities is (*paradigmatically* thought to be) *exhaustive*, but not all parts have to be present in every robotic application:

- **reality reflection** (Sec. 5.3.1): capabilities, resources, world modelling, perception, and monitoring.
- **decision making** (Sec. 5.3.2): plan execution, and knowledge-based reasoning.
- **interaction and communication**, (Sec. 5.3.3): via data, events and queries.

Each decision making activity makes use of models that represent the world with **symbolic**, **discrete** and **continuous** entities and relations, Sec. 5.1. The “perception” and “control” at these three complementary levels of representation are often called, respectively, (knowledge-based) **reasoning**, **supervisory** or **discrete event** control, and **continuous** control.

Obviously, these three control levels are always **inter-dependent**, so one needs two complementary types of activities to deal with these dependencies:

- **Skill: to coordinate** the other activities. The design of a Skill is based on the knowledge about how to execute a task given the available resources, about how to interpret the contextual situation, about which trade-offs can be made in perception and control, and, especially, about how to realise robustness of the task execution against **disturbances**.
- **World model: to produce** information about the state of the world to the decision making activities, and **to consume** the data about the state of the world from perception activities.

The mentioned **dependencies** between the activities are of the following types:

- **constraints** on the behaviour of an activity, or on the interactions between two or more

activities. The task specification can prescribe some constraints to be satisfied before execution can start (**pre-conditions**), but also while task execution is in progress (**per-conditions**), or only after the task execution has finished (**post-conditions**). The satisfaction of the latter type of constraints is often the *reason why* a task execution ends.

- **tolerances** on these constraints, to be “**guarded**” during task execution.

5.3.1 Reality reflection: capability, resource, world model, perception, monitoring

The **reality reflecting** parts [97] of the *Task-Skill-Resource* meta model are needed to **create, configure, store, update and process** the state of the **real world**:

- **resources**: the model of the robot’s mechanical structure, equipped with actuators, sensors and tools; and of the *constraints* of what can be provided by the resources that are necessary for the execution of the specified task. For example, mechanical strength, energy availability, computational and communication hardware properties, etc., together with the *quality of service* metrics which represent how well the resources are being used, and the *costs and risks* connected to that resource usage.
- **capabilities**: the *types of tasks* that the robot can execute, including the *constraints* of what the *execution* of the specified task can deliver to users, together with the *quality of service* and *task progress* metrics, that represent how well those capabilities are being provided.
- **world-model**: the *information* the robot has about the world around it, and that *needs to be shared* between several activities. There can be multiple “versions” of the “world” present at the same time, for example, past, present, and future states. The **world-model** is also the place to remember the past, and to predict desired or possible future worlds, both with various degrees of uncertainty.
- **monitor**: the model of the relations on world model parameters with which to check whether the *actual task* behaviour corresponds sufficiently enough to the *expected* behaviour. This is the **discrete** version of **perception**, that is, the one that triggers *events* in the **task** behaviour as soon as the mentioned checked turns out to be negative.
- **perception**: the model of how sensors can provide the *actual* information needed to create and/or update the **continuous** parameters in the world model, and in the monitors.

5.3.2 Decision making: plan, control, knowledge-based reasoning

Decision making in the *Task-Skill-Resource* meta model is done in:

- **plan execution**: the **supervisory control** of coordination and configuration of all other activities, according to the task specification. A *plan* is a model of which activities (perception, control, etc.) to execute, in which sequence and/or in which mutual concurrency, and under which conditions they can start or they must stop. The **plan** is the **discrete** part in the overall **control** behaviour, that is, the one for which the actions

are triggered by *events* via which the **task** tries to go from the actual model of the world to a desired world model.

- **control**: the model of how to move the robot towards the targets specified in the task, taking into account the constraints introduced by the task specification and the resources. This model contains the **continuous** aspects of **control**, that is, the one for which the actions are triggered by *data* streams containing real numbers. The latter, sooner or later, always end up with the sensors and actuators in the robots' hardware.
- **reasoning**: to support the reality reflection and decision making activities, whenever there are no **hard coded** solutions are available. In other words, finding out what **magic numbers** to set, in which activities, to which “right” values, based on knowledge that designers have encoded symbolically, about the application and about the dependencies between the activities.

In other words, this reasoning uses the information that represents (i) the *context* in which the other models are used, and (ii) the couplings (“**associations**”) between these models. In other words, the information about how the magic numbers in all these models are connected to each other and to the real world, for each specific set of capabilities and resources. For example, texture or color information of an object in the world that is optimal for a particular type of sensor (processing) to detect. This knowledge is accessible symbolically from the world model via **semantic tags** that “point” to the knowledge relations.

A *plan* is a *model* used by the decision making activities, that contains the information to configure, coordinate and execute *activities* when specific *guards* are violated. One particular plan can be realised in many different ways, even given exactly the same resources. This is where the **Skill** comes in, with a model of the knowledge about **how** to realise a task. One Skill model can be “better” than another one, in a *specific* application context, because it provides better choices of the many **magic numbers** and algorithms inside the same Task model.

5.3.3 Interactions: world model, skill, perception and control activities

The **interactions** on the Task meta model appear as follows:

- **inter-activity interactions via World model and Skill**: the connections between the decision making parts take place only indirectly,² via interactions with the **World model** and the **Skill knowledge model**: the former to provide them with the “state of the world”, the latter to provide the “state of the knowledge”, and both are indispensable to base decisions on.
- **interaction channels of data, event and query**: an **information architecture** that conforms to the Task-Skill-Resource meta model will introduce several information channels between the activities that implement the activities in that meta model, of the types described in Sec. 2.9.

²At least on the mereo-topological level of system abstraction: in the **information architecture** description of the system, it is possible that a *direct* data channel is established between, say, a perception and a control activity. But that operational decision is then made by the activity in charge of, say, the world model.

Queries are the most “symbolic”, knowledge-based type of interactions: each query is a complete *model* that is being exchanged between activities, allowing for higher levels of declarative interaction descriptions. In the context of this Chapter, *Task specification* models are the most relevant type of queries. And the **task queue** pattern is the obvious one to use in the interaction. Typically, the *execution* of a task specification involves several *submission* and *completion* interactions.

For example, the success of the “Web” is based on the fact that full HTML models are communicated, which are “task specifications” for the browser: the HTML models *what* to visualise, and the **browser engine** interprets that model and decides on *how* to execute the visualisation. The HTML “task model” is in itself also a very good example of a **composable** model: an HTML file in itself composes other web standard formats, such as SVG or JPEG. Even when a receiver can not “render” the full model, the composition semantics is clear enough (i) to allow local decision making about what to render or not, and (ii) to communicate back a “status report” to the sender explaining which parts of the sent model gave problems.

The interactions can be *physical* (via the mechanical tools on the robot’s structure), or *data driven* (via “device drivers” directly coupled to the hardware), but all but the simplest of applications involve a decent amount of *symbolic information* communication, via data flows, event broadcasts, and query solving.

5.4 Perception and control composition: proprio-, extero-, cartho-ceptive tasks

Robot tasks grow in complexity together with the **complexity of the tasks**; or more concretely, with the complexity of the task **plans**. That complexity obviously grows with the size and the semantic richness of the world models, and the symbolic **knowledge** that is available about these semantics. This Section introduces one particular set of **levels of abstraction** in task models, where each level represents one particular way of **closing the world** knowledge. There are few *scientific* arguments to motivate the statement that the described task categories are “the” relevant ones; but there are several *pragmatic* arguments:

- the level of certainty that the robot has about its own state is significantly higher than what it has about the world around it.
- the variability in the world around it is significantly higher than the variability inside the robot.
- what the robot can *sense* determines what it can decide for itself, also in the case that connections with external activities are not available.
- information about the world that the robot can *only* get via external connections must be used in different task execution activities than the one using the robot’s own sensing, because access to that external information comes with less guarantees.
- every external information task execution activity *can* be deployed outside of the robot’s hardware. While a task execution activity using the robot’s own sensors for the robot’s own motion *should* be deployed on the robot’s hardware.

5.4.1 Proprio-ceptive tasks

Proprio-ceptive tasks represent plans that refer only to the robot’s **instantaneous hybrid dynamics** behaviour. In other words, the **kinematic chain** of the robot *is* the **world model**, and its behaviour is that of an (electro-)mechanical energy transforming system, using only its own **proprio-ceptive** perception capabilities and its own actuators’ control capabilities.

Examples of proprio-ceptive tasks are: the instantaneous motion under the influence of a pushing force at any of the links in the kinematic chain, or the instantaneous open loop motion under the influence of torques applied at the joints, possibly together with instantaneous (artificial) acceleration constraints. Some monitoring use cases are: the motion makes the robot reach the planned position in space, as far as this can be interpreted by the robot’s own sensors, or the motion stops when a contact transition is detected via current, force, tactile or IMU sensors mounted at various attachment points on the kinematic chain.

The semantics of the **word stem** “proprio” is that of “ownership”, in other words, everything that is “one’s own”. This does not only refer to the *sensing* and *actuation* that the robot can do without the help of any other machine, (via its own **fieldbuses** and **general purpose IO**), but also to the *decision making* and *communication* that it has full ownership of.

5.4.2 Extero-ceptive tasks

In **extero-ceptive** tasks, sensors like **cameras** or **laser range finders** localise and track **object** features in the environment, and the control activity adapts its proprio-ceptive task execution behaviour accordingly. In other words, the range of the **robot’s extero-ceptive sensors**, and the information processing capabilities of the sensor signal processing, determine what the world *is*.

5.4.3 Cartho-ceptive tasks

The sensors localise and track **object** features in the environment that are indicated as semantic tags on the map, In other words, the **robot’s extero-ceptive sensors** determine what the world model *is*, together with an **environment-centric map**. The *world model* contains features of robot-external objects, with their robot-centric perception affordances; for example, *visual servoing*.

5.5 Mechanism: task specification as optimization, satisfaction and reasoning under constraints

Previous Sections introduced **mereo-topological structures** that represent the complexity of robots acting to realise tasks in their environment, with representations in **continuous, discrete and symbolic** domains. This Section adds one particular methodological approach to add **behaviour** to that structure, because that approach fits very well to this document’s emphasis on **composability**: a task is formulated as a **search problem** under **constraints**.³ Two flavours of solvers (that is, **search algorithms**) are relevant:

³The system developers should aim for a representation with has enough information to create a **feasible state space** that is (i) a faithful representation of all task requirements and resource and environment constraints, and (ii) **computationally tractable**.

- **optimizing** solver: one wants to find the *best* solution, where the quality of the solution is represented by the problem’s **objective function**.
- **satisficing** solver: one is satisfied with a solution that is “*good enough*”, i.e., the search stops as soon as *a* value of the objective function has been found that is above the threshold value of the desired minimum quality.

In many cases, the *optimizing* approach implies that computational resources are spent on reaching a result that is (i) not needed, and (ii) sensitive to small disturbances in the problem formulation. The role of *tolerances* in the problem specification is to indicate how well constraints and objective functions are to be realised by the solvers. With that information available, the solver’s computational complexity can be reduced, because the optimization algorithm is allowed to return a result as soon as it finds one that satisfies the problem formulation in an *adequate*, satisficing, way.

5.5.1 Specification in the continuous domain: constrained optimisation

For the continuous aspects of a system, the methodology is to formulate a task specification as a *(hybrid) constrained optimisation problem*:

Hybrid constrained optimization problem	
task state in the task domain	$X \in \mathcal{D}$
robot continuous motion state	$q \in \mathcal{Q}$
desired task state	(X_d, q_d)
objective function	$\min_q f(X, X_d, q)$
equality constraints	$g(X, q) = 0$
inequality constraints	$h(X, q) \leq 0$
tolerances	$d(X, X_d) \leq A$
solver	algorithm computes q
monitors (Boolean functions of X)	decide on switching

Its *outputs* are the *instantaneously desired changes* in the robot’s continuous motion state $q \in \mathcal{Q}$, that is, torques or velocities. Its *inputs* are the following entities, relations and constraints, [10, 14, 28, 65, 76, 77, 88]:

- **configuration space**: all the parameters in the models of the structures (hierarchies and information associations) in the previous Sections. For example, the joint space parameters of a robot and its actuators, q , and Cartesian space parameters X .
- **desired configuration** (X_d, q_d) : the sub-sets of the whole joint and Cartesian configuration spaces that the execution of the task should have as its outcome.
- **objective function(s)**: relations $f(X, X_d, q)$ on these parameters that the task execution is expected to optimize. For example, the closeness to a target area, the distance to obstacles, the progress of the task execution, or the energy consumption of the robot,
- **constraints**: equality relations $g(X, q) = 0$ and/or inequality relations $h(X, q) \leq 0$, on the configuration space parameters, that must be satisfied during the execution of the task. For example, joint limits, singular configurations in a kinematic chain, non-collision, or remaining within dedicated areas.

- **tolerances:** inequality relations $d(X, X_d, q, q_d) \leq A$ that describe how well the constraints have to be satisfied, and/or how far the objective functions have to be optimized.
- **monitors:** the algorithms that compute the values of specified tolerances, and that have [magic numbers](#) to turn these values into *events*
- **solvers:** the algorithms that take all of the above inputs, and compute the instantaneous “drivers” for the robot’s actions (motion as well as perception).
- **hybrid:** the solver algorithms also react to monitor events, and then (possibly) switch their solver algorithms.

For each particular **domain**, one must fill in the *types* for f , X , q , etc., as well as a particular *type* of solver. For each particular **application** in that domain, one has to fill in specific constraints, objective functions, monitors and tolerances. One concrete example of a constrained optimization problem is that of [hybrid kinematics and dynamics](#).

5.5.2 Specification in the discrete domain: constraint satisfaction

A **constraint satisfaction** problem is the discrete complement of the [continuous constrained optimization](#) problem. Its *inputs* are *dependency relations* between the behaviour in [activities](#). For example:

- partial ordering constraints between activities, about when each of them can or must start and end. Such as opening the gripper of the robot only *after* the object in that gripper has been placed on the table. Or the ordering of the [three sweeps](#) in a dynamics solver.
- protocols to access shared resources and to communicate information. Such as the transfer of ownership in [interaction streams](#).

Its *outputs* are [computation schedules](#) and coordination models that make the task execution activities satisfy the dependency relations. These coordination models are a composition of the previously introduced coordination primitives: [flags](#), [flag \(arrays\)](#), [Petri Nets](#), and [Finite State Machines](#).

Robotic systems have such a large variation in discrete satisfaction problems that no generic problem formulation can be provided, as is the case in the continuous domain.

5.5.3 Specification in the symbolic domain: reasoning

Semantic reasoning is the symbolic complement of [continuous constrained optimization](#) and [discrete constraint satisfaction](#). Its *inputs* are knowledge relations that represent and link the behaviour in the system’s activities; for example:

- traffic rules: how do traffic signs and signals influence the motion of a robot.
- the algorithms that a robot has at its disposal to solve particular perception and control problems.
- the physical pre-, per- and post-conditions connected to actions of the robot.
- qualitative specifications of spatial actions, [31, 61, 62, 63].

Its *output* is a complete and consistent set of dependency relations that provide all information needed to create the discrete and continuous optimization/satisfaction task specifications.

Again, no generic problem formulation exists, because of the huge variation in use cases and contexts.

5.5.4 Policy: task requirement as objective function or as constraint

One should be careful about what to use as objective functions in an optimization problem, for several reasons:

- functions like *time*, *energy consumption*, or *safety*, are *derived* quantities: their values can not directly and predictably be influenced by the actuator signals q . Hence, it's often better to select them as *inequality constraints*, that are guarded by monitor activities during the task execution, to allow the plan to switch to another control approach when the *realised* time, energy, etc., fall outside of the *specified* boundaries.
- motion and effort variables are more directly influenced by the actuators, hence it is typically easier to use objective functions that combine one or more of such variables. For example: deviations from geometric paths; or predicted violations of tubular regions after a certain time horizon in the future.
- it is mathematically easy to specify a *multi-objective optimization*, via a weighted combination of various objective functions. However, this *requires* the choice of weighing factors, butch are often impossible to derive from the task context in a unique or deterministic way. Hence, the weighing factors often remain very arbitrary, and not motivated by knowledge insights into the task challenges.

A primary example of such difficult weighing choice is the trade-off in the task specification between:

- **utility**: the value that a successful execution of a task brings to the system.
- **cost**: the investment in resources requierd to realise the task.
- **risk**: the expected extra cost of the task execution when that execution can not proceed in the expected optimal way.

The resulting **best practice** is to choose **one single objective function** per state in the Task plan, and to foresee other control states to switch to whenever one or more of the other *monitored* (but not *optimized*) functions exceed specified tolerances.

5.5.5 Policy: dimensionality reduction through task constraints

Robots can live in high-dimensional configuration spaces, with many sensors and actuators, many dynamic objects in the world each with its own state, etc. The task specification can reduce that complexity by introducing constraints between several dimensions. For example, every **physical contact** between the robot and part of its environment is an opportunity to reduce the complexity of specification, interpretation, monitoring and control:

- the number of parameters needed to model the motion of the robot in *physical* contact are reduced compared to the unconstrained situation, because many motion parameters are then also *mathematically constrained* to belong to a sub-manifold of the unconstrained manifold.
- the task can introduce *control* with the explicit intention to keep the robot on that sub-manifold, [11].

- the task can introduce a *plan* to solve a six-dimensional uncertainty as a series of lower-dimensional control problems, [95].

For example, specifying the motion of a point on the robot’s end-effector on a surface is a two-dimensional problem, and not a three-dimensional one anymore. Letting the robot find the corner of a box can be done using a plan that, first, lets the robot find *any* point on the box by specifying its motion in a two-dimensional plane until a contact is detected, and then moves over that contact plane until it finds the plane’s edge, and so on.

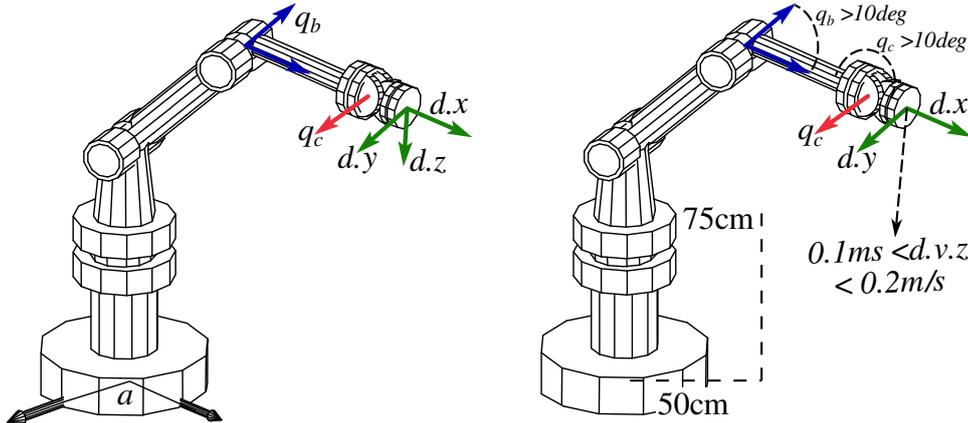


Figure 5.3: Example of a proprio-ceptive guarded motion. The model of the robot’s `Kinematic_chain` is extended with the Attachments *a*, *b*, *c* and *d*, via which the task is formally specified as a set of desired/monitored parameter ranges.

5.6 Policy: reactive task models — Guarded actions

Figure 5.3 shows examples of robot task specifications, in the form of so-called **guarded actions**. These are often also called “guarded motions” in the common case where the robot action consists of only motion. This task specification approach goes a long way back in the history of robotics [52, 71, 106], in a version in which the *objective function* in the **hybrid constrained optimization** specification is replaced by motion drivers based on feedforward and/or feedback control of the robot. In those days, solving constrained optimization problems was too computationally expensive. However, the approach is still relevant, because it is a simple (and hence effective) way to realise a **task model**:

- the robot’s model *is* the world model.
- the plan has only one action.
- the control has a constant specification.
- perception use only state variables from inside the robot controller.
- the monitoring part of the model means that the motion goes on until a particular relation between observed and specified state variables (that is, the “guard”) is satisfied.

This task model must, obviously, be composed with a lot of other things, such as:

- *algorithms* with which to realise control, perception, and monitoring.
- *discrete control* and *world model updating*.
- *relations* with which to fill in and adapt **magic numbers** at runtime.
- *relations* that add constraints from robot and environment.

Ideally, *all* magic numbers in the specification come from contextual relations, such as **PreCon** provides the “pre-conditions” magic numbers, **Spec** provides them for the “per-condition” specifications, and **Post** provides them for the “post-conditions”.

```

CONTEXT: Pre, Spec, Post
MOVE: d.origin in direction d.z
WHEN:    // pre-conditions
    d.origin further than Pre.[50 cm] away from a.origin
    d.z is larger than Pre.[75 cm]
WHILE:   // per-conditions
    keeping d.origin.speed between Spec[0.1 m/s] and Spec[0.2 m/s]
    keeping d.origin further than Spec[50 cm] away from a.origin
    keeping q.b angle larger than Spec[10 degrees]
    keeping q.c angle larger than Spec[10 degrees]
UNTIL:   // post-conditions
    d.z is smaller than Post[75 cm]

```

This reactive approach has the advantage that it does not rely on the specification of a **desired trajectory**; on the contrary, the trajectory is the *result* of the execution of a **motion-generating behaviour** together with a set of constraints on that behaviour. The above-mentioned historical example relies on a rather restricted world and robot behaviour model (namely [proprio-ceptive control](#)), because of the computational and tooling limitations of robotics systems of the 20th century. The same methodology still applies to systems with a lot higher configuration spaces, as demanded by 21st century robotics applications. Or rather, the methodology is *even better* suited to scale than its *planned trajectory* alternative, because it need not take all constraints and/or motion degrees of freedom into account at the time of *specifying* the motion, but only the ones that are (observed to be) relevant during the *execution* of the motion.

Figure 5.4 sketches another specification example, this time for a mobile robot moving inside of a room:

```

CONTEXT: MPre, MSpec, MPost
MOVE:
    d1.force in direction d1.z
    d2.force in direction d2.z
WHEN:    // pre-conditions
    d2.origin further than MPre.[100 cm] away from Line(X3,X4)
WHILE:   // per-conditions
    keeping Line[d1,d2] within MSpec[Tube(X1,..,X6)]
UNTIL:   // post-conditions
    d1.origin OR d2.origin is closer than MPost[150 cm] to Line(X2,X3)

```

5.7 Task composition

If a robot system gets more motors and sensors, it can, in principle, perform more tasks at the same time. Of course, such task composition has many facets, is not unique, and inevitably comes with **dependencies** between the composed tasks that must be dealt with.

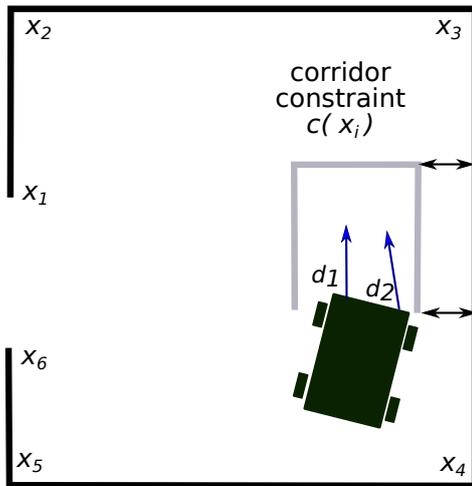


Figure 5.4: Example of a guarded motion specification to make a mobile robot drive within a “corridor”, that is defined with respect to the wall of a room. The specification uses the IDs of relevant points in the world model: d_1 and d_2 are the points on the robot where (virtual) actuation forces can be applied by the control system; x_1, \dots, x_6 are the corner points of the room.

5.7.1 Horizontal composition: sharing world models

Some tasks involve more than one robot, or one robot can execute more than one task at a time; for example: dual-arm tasks; platooning of cars or boats; etc. The various activities involved in the task execution **share** *at least* a part of the world model, but in most cases also parts of perception, control and monitoring, Fig. 5.5. This results in:

- access to **shared resources** (such as the world model) must be coordinated between activities.
- if (the execution of) the plan, control, perception and monitoring models is done by multiple activities, their internal “state machines” must be adapted to guarantee this coordination.
- constraints and objective functions are *fused* in some parts of the task specification’s *plan*.
- tolerances might be adapted to the specific context of a specific composition, and hence the monitors that are connected to checking the tolerance violations.

Overall, the same [constraint-based control methodology](#) applies as for the composed tasks individually, and in many cases “all” that has to be changed are the *Coordination* and *Configuration* parts (Sec. 9.3.4) in the system’s architecture.

5.7.2 Vertical composition: resources become capabilities, and vice versa

In the simplest form of vertical composition, the *capabilities* of a lower level become the *resources* of a higher level, Fig. 5.6. And the composite task specification **adds constraints** to the various capabilities and resource models; for example, when using the task specification meta model of [constraint-based optimization](#):

- *higher level* adds constraints and objective functions to *lower level’s* [constraint optimization](#) problem.
- and vice versa.
- there is a need to introduce *extra* constraints and objective functions because of the *coupling*.
- hence, also extra tolerances and monitors are *needed*.

For example, an electrical motor influences the mechanical joint motion control via motor

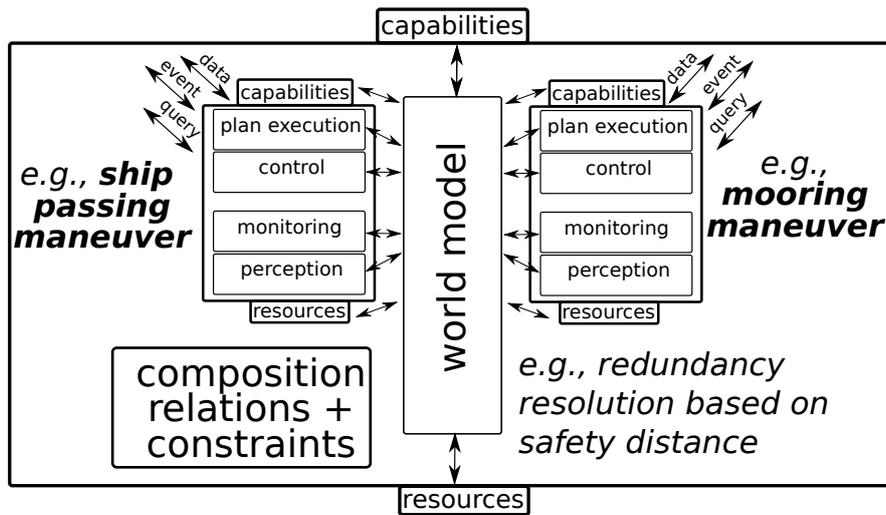


Figure 5.5: Horizontal composition of tasks, illustrated in a context of **shipping** via **inland waterways**.

heating and energy efficiency constraints; and that mechanical joint motor control influences the kinematic chain motor control via position or torque limits. The objective functions to be optimized for the motors and for the kinematic chain can be designed independently of these constraints, and it is only by the *solution* of the whole constrained optimization problem that the integration takes place. That is, the *monitoring* of the constraints gives rise to switches between several optimization problems to be solved. This approach yields a very **composable** way of sub-system integration, and the overall system behaviour emerges from the individual task executions' behaviours with high predictability, except for (i) the exact time on which the system will react, and (ii) the exact sequence of states that the system will evolve through.

5.7.3 Strategic, tactical and operational task levels

The following terminology is sometimes used to refer to particular vertical composition levels:

- **strategic**: decisions about what investments in resources are needed to create profit.
- **tactical**: decisions about how to realise these strategic investments best.
- **operational**: decisions about which existing resources to deploy to create required capabilities.
- **supervision**: decision about accepting the performance of provided capabilities, or to adapt them.
- **coordination**: execution of capabilities, monitoring, and reconfiguration.
- **control**: continuous time execution control.

5.7.4 Multi-tasking: coordinated, orchestrated, choreographed

The following three types of task execution coordination models are needed:

- **coordinated** tasks: **one** sub-system (a robot or not) provides all other robots or non-robot sub-systems, online, with (i) individual task specifications, and (ii) the events to coordinate the local executions.

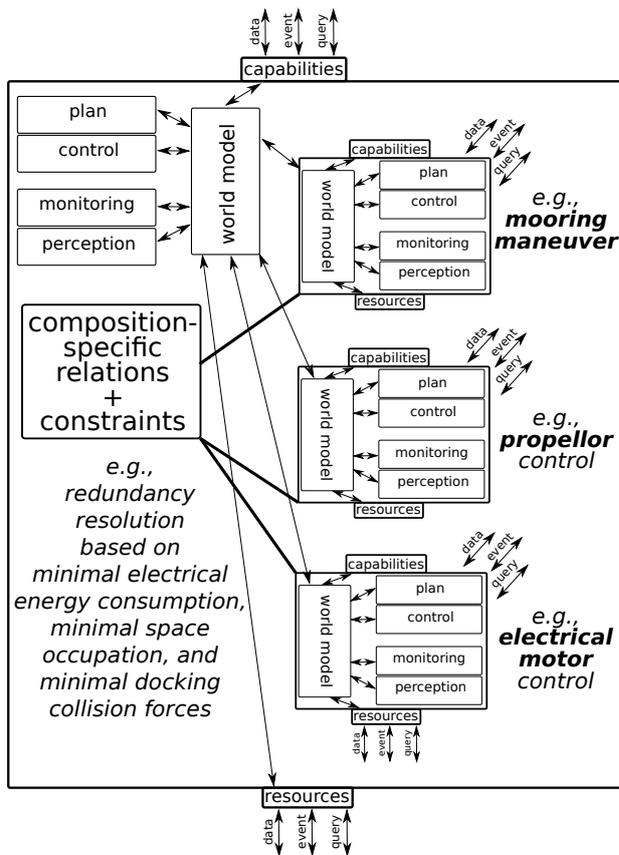


Figure 5.6: Vertical composition of tasks.

- **orchestrated** tasks: all motion subsystems have already the task specifications **on-board**, and only the coordination events must be communicated.

For example: a robotic manufacturing cell, where all robots gets the assembly programs from the cell supervisory system, together with the events to trigger their execution and (re)configuration.

- **choreographed** tasks: all subsystems generate coordination events **themselves** based on their **sensor-based observation** of the other platforms; hence no communication is needed but only perception.

For example: human-aware robotic manufacturing cells, where the reactions of the robots to the presence of humans in their neighbourhood are pre-programmed (or, better, modelled), but the coordination events inside and between robot control systems are to be generated by the latter control systems themselves.

5.7.5 Shared control of task execution

Robotic systems are increasingly expected to work together with humans, even in physical contact with each other. The following levels of so-called **shared control** have crystalized over the years:

0. **No assistance.** The human provides all physical power to move an object.

1. **Unconstrained force amplification.** The human’s power is captured in a device, and one or more degrees of freedom of the captured human input is amplified by the robot.
2. **Partially constrained force amplification.** The robot’s power amplification is constrained by virtual “fences” or “surfaces”.
3. **Autonomous obstacle avoidance.** The robot adds extra motions to the ones put in by the human, to avoid obstacles that it can perceive via its autonomous sensing capabilities.
4. **Partially autonomous trajectory following.** The robot adds extra constraints to the motion by a virtual “trajectory” (only the geometrical part of it, not the timing part!), from which it deviates as little as possible.
5. **Full autonomy.** The robot does the full control.

All of the above cases deal only with the **continuous control** aspects of a task. Even in the *Full autonomy* case, there is still a *sharing* of the control for the **discrete control** aspects: humans do the perception and monitoring in the continuous task spaces, and their inputs are not interpreted as continuous domain power setpoints, but as discrete switches between different continuous control modes for the robot.

5.8 Task specification, data sheet, contract, responsibility and commitment

Models of tasks are necessary to represent *what* a system has to achieve as results. But systems are more than just about functionality and performance: **non-functional requirements** are very important to assess the added value of a system deployed in a particular application context. This Section introduces some relevant higher-order relations that connect a functional task description to some non-functional **contracts** that the system subscribes to when it accepts *to execute* a task description. Contracts come in various shapes and forms, and the purpose of this Section is not to be exhaustive, but the following two categories are **common**:

- **commitment**: the task executing party accepts a **best effort** contract, in which it guarantees that it uses the agreed-upon amount of **resources** to reach the task requirements.
- **responsibility**: the task executing party accepts the **liability** to realise the agreed-upon **outcome** of the task.

In practice, contracts are sterile relations between two parties, unless there are quantifiable ways to assess the extent to which each party respects the contract. No new *mechanism* is needed to represent a contract model: the generic **higher-order relation model** suffices. Indeed, also the non-functional *requirements* for a task must be transformed into a set of *constraint* relations, with *tolerance* relations on top. “All” the contracting parties have to do is to agree on:

- *how to measure* these tolerances.
- what *costs* are implied by which level of tolerance violation.

5.9 Taxonomy of action, actor, actant, activity, agent

This Section introduces terms that (sometimes) appear in the literature as complementary *stakeholders* in task representation and execution.

The “**action**” noun is a **semantic hypernym** for the two nouns **motion** and **perception**, and it represents a **model of what happens in the world**. (The same action model can have various interpretations: “actual”, “desired”, “possible’,...) Action models appear in *all* robotic systems, at various **levels of abstraction, resolution or granularity**, encoded with various (often hierarchically) **interconnected higher-order relations**, and with a large variety of “performance”.

More semantic concreteness comes from the identification of (i) the **actor** that is responsible to execute the **action**, and (ii) the **actants** (that is, **objects**) that are required to make the actoin succeed, or that the actor has to take into account as possibly impacting the successful execution of the action. Any robot can move itself (hence the actor and tha actant are the same), but a hand grasps “something”, that is, the grasped object is the actant; a pinch grasp is performed with only the thumb and the index finger. This example makes it clear that the **spatio-temporal scope** of each term becomes smaller if the term is attached “deeper” in the hierarchy.⁴

The terms action, actor and object, as introduced above, represent **knowledge models** in this document. This knowledge is used in a an **activity**, which is a **process** that implements the execution of an action, with “physical processes” or “digital twins” for actor, actant(s) and action.

Finally, the name (**software**) **agent** is given to the **system** of activities that belong together, as being executed by one single physical system in the real world.

5.10 Bad practices in task specification

(TODO: intantaneous reactive control, e.g., via potential fields; weighing of translation and rotation; weighing of objective functions and constraints; not making parameters dependent on the context, but use as fixed **magic numbers**; solving problem to the optimum, every sample again instead of tracking solutions and using a satisficing approach; defining the error functions as instantaneous errors on setpoints in position, velocity and acceleration; neglecting the fact that constraints on joint velocity have seldom physical or economical sense.)

5.11 Use case: semantic indoor navigation (revisited)

This Section extends the short **introductory description** of indoor robot navigation, towards **semantic** navigation. Fig. 5.7. The inspiration comes from the **traffic system**, or “driving” in general, as a major societally relevant systems-of-systems example. One can assume that everyone is acquainted with the a formal representation of (i) tasks and of (ii) the perception, control and monitoring required for the execution of those tasks. That is, recognizing building features or traffic signs, localizing them in their spatial context, and inferring their influence on the robot’s current actions.

⁴The oldest(?) reference that explicitly introduces such natural hiearchy of **increasing order of intelligence with decreasing order of precision** is [86].

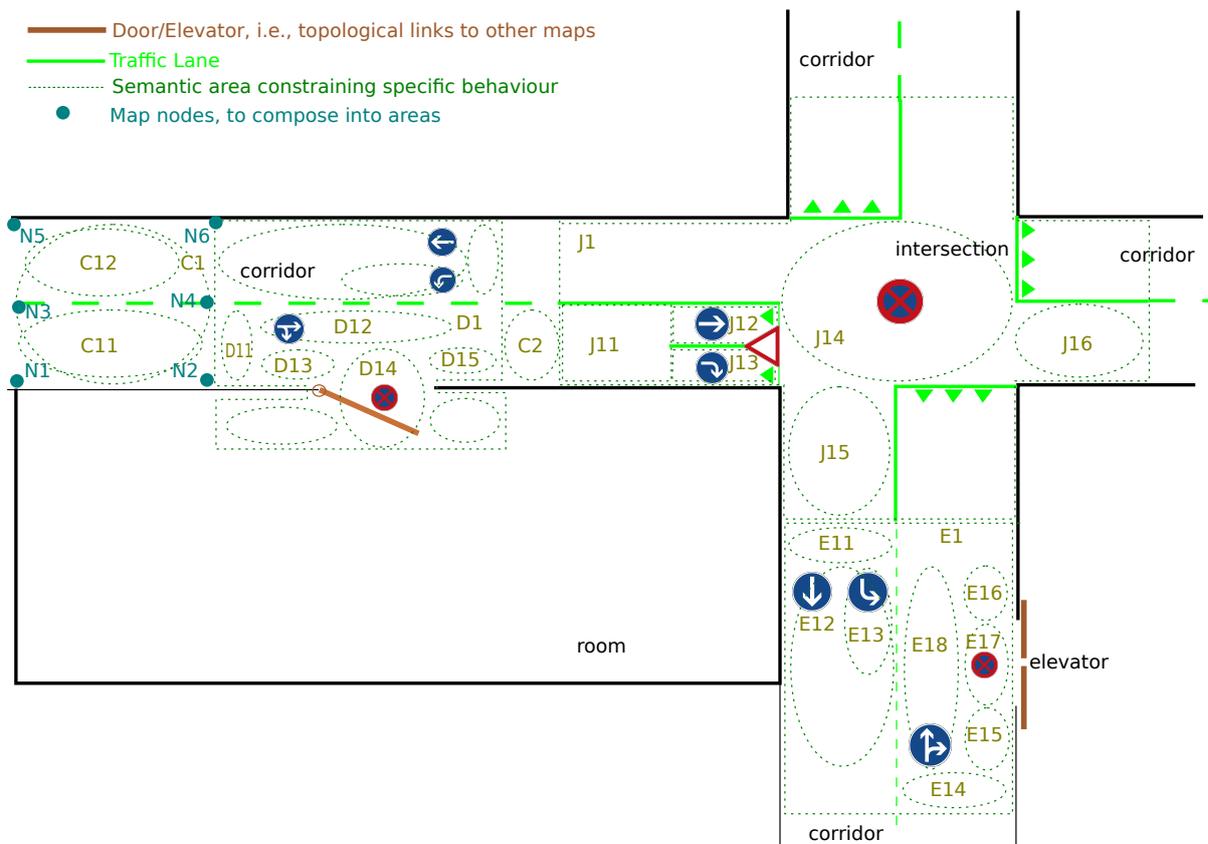


Figure 5.7: *World model* of an indoor “two-corridor-with-intersection” area: the solid black lines represent the **geometrical** properties of walls, and the red lines represent doors. The other map entities are the **semantic tags** of the traffic signs and markings. (Figure courtesy of Nico Hübel.)

Traffic lanes, signals and signs are amongst the most familiar **semantic tags** worldwide: all drivers are trained **to interpret** those **symbols**, and to let them influence (i.e., constrain, as well as optimize) their driving behaviour with cars and bikes, and their traffic participation as pedestrians, individual as well as groups.

The symbols model the *constraints* that every robot must respect when it drives through the area. Each such **symbolic** tag is, directly or indirectly, attached to one or more **geometrical** features of the map. These symbols model “motion” in a fully **declarative** way, because they do not prescribe **how** a traffic user should move, but rather which relations the actual motion is expected to satisfy. The real-world **instantiations** of the traffic symbols are **designed to be perceivable** by human drivers in (almost) all environmental and weather conditions. One of the meanings attached to traffic symbols is the area they cover in the world, and the shape and extension of these areas are **designed** to fit to the **control bandwidths** that can be safely expected of all actors that take part in the traffic. Together, a traffic layout is an **architecture** of semantic traffic primitives in the world that (almost) guarantees that **humanly controlled systems** can drive safely and efficiently. (That is, as long as the latter reduce the risk during participation in traffic.)

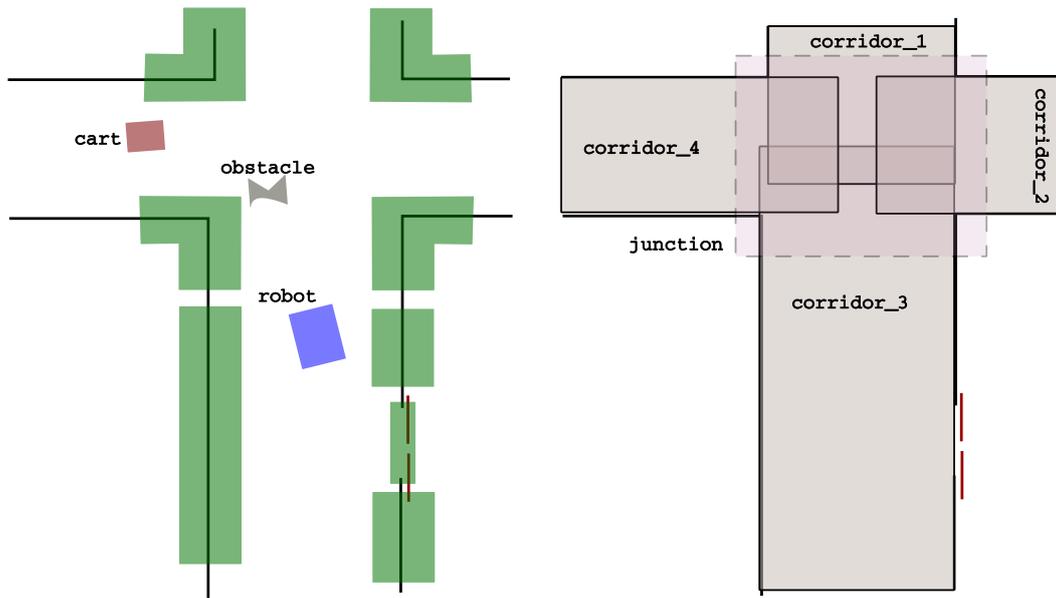


Figure 5.8: The semantic “base layer” of a world model, with several types of “tags”. Left: perception tags (in green) for a laser range finder with sensor processing capabilities that can detect straight wall segments and rectangular corridor corners. The drawing also contains the tags (and the spatial extension) of objects that are relevant for the Task: the robot of course, and the `cart` that it has to navigate to; but also an unidentified `obstacle`. Right: five (overlapping) `free space` areas, one for every `corridor` that ends up in the `junction`.

The main purpose of this Section is to explain how *engineered systems* can make use of the `task meta model`. The first step in that direction is to add the **geometric** level of abstraction, Fig. 5.8: the **world model** is filled with (models of) the entities in the environment: the location and shape of the traffic areas, and the shape and motions of the robot’s **kinematic chain**. In addition, the world model gets relations that link some of its geometric primitives to perception capabilities that are present in the robot control system: there are some “walls” in the environment of the robot that its laser scanner sensor processing algorithm can detect and track, so that the position of the robot in the world model can be updated when new sensor information comes in.

5.11.1 Geometric world models with semantic tags for control & perception

Geometry. Figure 5.7 depicts a **world model**, with information at the *geometric level of abstraction*: it has **geometric primitives** such as **points** and planar **polygons**, and it has **semantic tags** (each attached to a geometric primitive) to model **landmarks** (i.e., *task-relevant* places in the world) that have **features** (i.e., *task-relevant* properties of a landmark used in **motion** and **perception** models). The Figure sketches an indoor area of two intersecting corridors, with doors to rooms and elevators. These doors and walls form **geometric constraints** for any **motion** that robots execute in the modelled world, because they have to steer clear from collisions with these “hard” world landmarks.

Plan. The next step after the modelling of the world, is the modelling of the task plan. The simplest form of such a plan is a *finite state machine*, with in each state a choice of a model for the *control*, the *perception* and the *monitoring* that the robot is expected to realise in that state; the monitoring provides the *events* to trigger a state change in the *plan*.

Figure 5.9 sketches how to use world model landmarks to attach some essential tags used in a **plan** model: a “*tube*” is connected to some tags in the traffic model, to indicate the area within which the *control* must keep the robot, while its *perception* measures whether it is making “good enough” progress towards some other traffic tags; various *monitors* will follow the approach to one or more of these target tags, and signal the *plan* when the robot has “reached” one of them. More concretely, the robot should (i) not drive into the natural constraint of the wall but follow it, (ii) satisfy the artificial constraint of the traffic lane, and (iii) be ready to stop in time in front of the intersection.

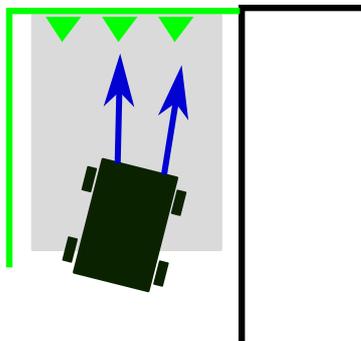


Figure 5.9: Semantic world model, extended with a motion **control** specification, in the form of a **tube** (the shaded gray area that represents the constraints of the control) and *motion drivers* (the blue arrows that represent (artificial) forces that generate the **instantaneous motion** of the robot).

Control. In symbolic form, the control model can be as simple as the two blue arrows in Fig. 5.9: the arrows represent two driving forces:

- the *origin* of the symbolic force driver arrow is a reference to a physical attachment point on the robot;
- the *end* of the arrow is a reference to some of the target tags in the world model.

More concretely, in the Figure, the driving forces are directed towards the semantically tagged area of the intersection of this corridor with the junction area. In a full formal representation, this “pointing to” corresponds to a topological connection between the instances of object features on the real robot and the real environment.

The Figure also sketches another type of semantic tags, representing a **right of way** constraint. This tag has an *area* of influence, that is indicated symbolically by the green line segments.

These symbolic forces, and the symbolic constraint, are inputs to a motion controller, that transforms them to actual actuating torques on the motors, maybe by solving a **hybrid constrained optimization problem** (HCOP) in which the symbolic relations are used to (i) select the parts that must be included in the HCOP as relations, and (ii) some **magic numbers** in those relations. In this way, the control algorithm remains simple and constant, because it is rather straightforward **to inject** the context-dependent information (that is embedded in the world model) into the algorithm.

The geometry-based semantic tags are **not enough** to specify the controller’s HCOP completely. Here are some examples of missing information, that must still be *injected* too:

- *progress measure*: the geometry can provide information about the **direction** of the motion, but information about what is an appropriate **speed** must come from other types of knowledge. For example: maximum, minimum and/or optimal energy-motion curves for the robot; time-sensitivity of the task (e.g., a rush order); safety regulation.
- *tube following policy*:
- *obstacle avoidance policy*:

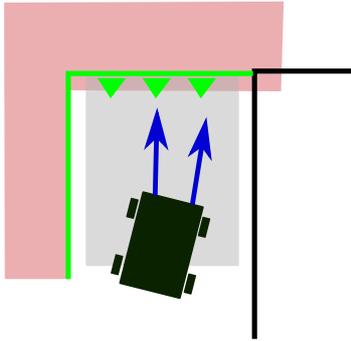


Figure 5.10: Semantic world model of Fig. 5.9, extended with the **local perception tags**: the green area focuses the perception of tracking a “wall” feature, at a resolution high enough to support the motion control; the red area focuses the monitoring on the detection of “any” feature in the direction of motion to react to, at a resolution low enough to react in time.

Perception. Some (possibly different) semantic tags in the world model also serve the robots’ **perception**. For example, Fig. 5.8 (Left) represents the **perception** tags for a simple laser range finder type of sensor: each tag is an *area* connected to a landmark on the world model (e.g., a *wall*) and the area indicates the “attraction region” in which *all* LIDAR measurement points can, or should, be **associated** with the wall landmark. Figure 5.10 adds a second perception tag: the nearby “rest” of the environment in the direction of motion, to be monitored for the presence of “obstacles”; that is, *any* LIDAR measurement point in that area makes the monitor “fire”. That signal can be taken up by the plan coordinator, to switch to another part of the **plan**.

An important added value of the represented task *knowledge* is that all the sensor data that comes from beyond this local horizon *need not be processed*, because it does not have an impact on the currently executed parts of the **plan**. In resource-constrained applications such as robotics, it is indeed as important to know what *not* to spend effort on as it is to know what *must* be done.

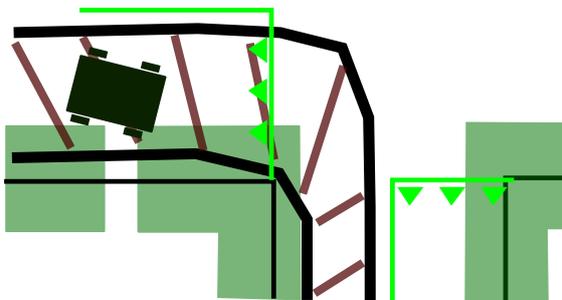


Figure 5.11: Semantic world model, with a task horizon that covers the sequential composition of two or more sub-tasks. More in particular, given this world model areas to the robot controller suggest that, in its current motion execution, it should already take into account that “something” might come from “behind the corner”. A possible control behaviour that would profit from this knowledge can make the robot move around the corner, further away from that corner than it would have done without the “preview” information.

Preview control. A more advanced task plan can include a *preview control* model, as depicted in Fig. 5.11: one can compose two or more sequential sub-tasks “to look ahead” to the next area on the map that the robot has to drive through, and to provide more extensive natural and artificial constraints that come with this extended context. The composite task plan is again a “tube” as in Figs 5.9–5.10, but now one with a bend around the next expected corner. Again, nothing changes in the perception, control and monitoring functionalities, because all extensions are added to the world model, and to the plan.

5.11.2 Specification of an activity to realise a task specification

Activities are an important target for task specification, because the declarative geometric specifications of the previous Section must, in one way or another, be transformed into *actions* of the robots, and *activities* are the system architecture primitives where that transformation responsibility resides. More in particular, an activity adds the following to a specification model:⁵

- *algorithms*: to interpret the raw sensor data in the context of the task specification, and to generate setpoints for the actuators. These core algorithms are complemented by monitoring algorithms, that evaluate how much “progress” is made towards the task goals.
- *coordination*: a robotic system of a realistic complexity has many sensing, control and monitoring algorithms active concurrently. Hence, there must be a coordination of which algorithms to run, on which data. Such coordination has, of course, and algorithmic basis itself.
- *communication*: different activities can run concurrently on the same CPU, or in parallel on different CPUs or machines. Hence, some data must be exchanged between them, either via coordinated access to shared memory, or via message passing mechanisms. All of these also involve their own dedicated algorithms.
- *event loop*: all of the activity parts described above result in a set of functions to be executed on a set of data structures, and the event loop is the place where the functions are actually executed.

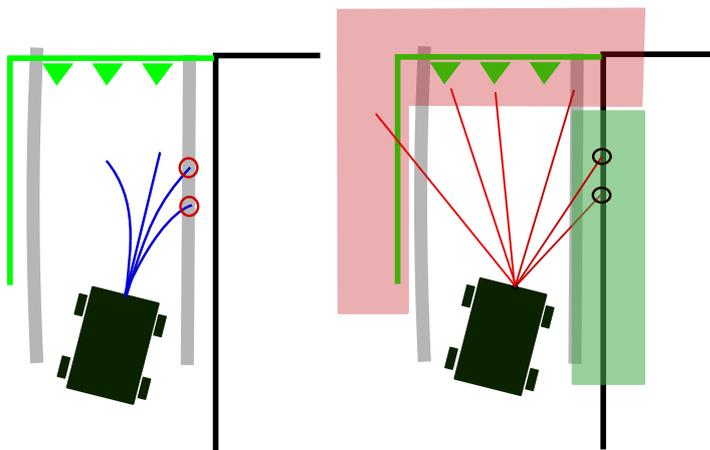


Figure 5.12: Left: control part of the task. The best trajectory for the controller to choose is the one that (i) comes closest to the end target area of the currently traversed section of the world, and (ii) is furthest away from intersection with the (artificial) “task tube”.

Right: perception part. The localisation of the robot is the estimate that (i) makes the simulated world fits best to the sensor measurements, and (ii) deviates minimally from the previous localisation estimate.

⁵The details of the components in an activity are explained in more details elsewhere in the document.

Figure 5.12 sketches possibly the simplest set of activities that warrants to be called a “robotic application”. When the sensors are limited to **encoders** on the wheels, and a **LIDAR** on the robot, activities of the following kind are needed:

- *motion activity*: the Figure sketches the primitive motion that the robot can realise, that is, when applying constant setpoints to the its motors, in **open loop**, the robot will realise a curved trajectory, as represented by one of the blue curves in the Figure. A possible **feedback control** activity works as follows. The controller tries different motor setpoints, on a *model* of its motion behaviour, which results in the series of blue curves depicted in the Figure. It then selects that trajectory which has the “best” motion within the “tube” specified in the task model. In this example, the “best” trajectory is the one that comes closest to the end of the current world model segment available for motion.

If that “best” one results in the robot moving too slow towards the end point of the segment, the *speed* setpoint is increased, for example with an **ABAG**-like controller. If too many simulated trajectories intersect the “tube” on the left or on the right, the *steering angle* is increased, also with an **ABAG**-like controller. The waveforms of the **ABAG** controllers are to be chosen by the control developers.

- *sensing activity*: the Figure sketches the usage of a laser scanner, with two “hits” encircled in black. These are used to update the position of the expected wall “line”, for example via a (Bayesian) localisation algorithm. The latter is also a constrained optimization problem, because it finds the *minimum* error between the measurements on the one hand, and, on the other hand, the modelled world with the estimated parameters of the wall filled in.
- *world modelling activity*: this activity updates the information it already has about what objects are present in the robot’s environment, and where they are. The information comes from, both, a *map* that is available a priori, and the sensor processing of the **LIDAR**, often via one or more levels of **data association**.
- *monitoring activities*: the task specification has modelled the desired progress of the robot, in a model of the task that was specified on the a priori map. One or more monitoring activities compute to what extent all of the constraints in the task specification are satisfied. (But also to what extent the required resources are available for the task execution.) This can be done in many different levels of the **data association** (e.g., wheel unit, platform, environment), and considering many different features (e.g., motion, effort, time, oscillations, etc.).
- *communication activities*: all of the above-mentioned activities share one or more data structures, via interaction with the world model. That interaction is a form of “communication” between these activities.
- *coordination activity*: when the *control* and *perception* errors are “too big”, as quantified by the *monitoring* activities, the coordination activity reacts to this situation by selecting another combination of *control*, *perception* and *monitoring*. as available in the *plan* model.

5.11.3 Example task plan: dock to cart in next junction left

This Section gives an example of what the just-mentioned *coordination activity* could look like, when the context of the task specification is broadened a little bit, from the *single action* context of the previous Section to the context depicted in Fig. 5.13.

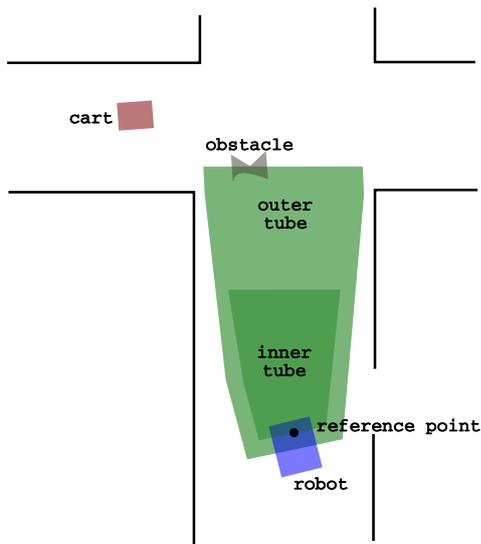


Figure 5.13: In the environment sketched in Fig. 5.8, this drawing sketches the first of three “tube” specifications, for the task of moving the robot to the cart. This first part must bring the robot to the junction.

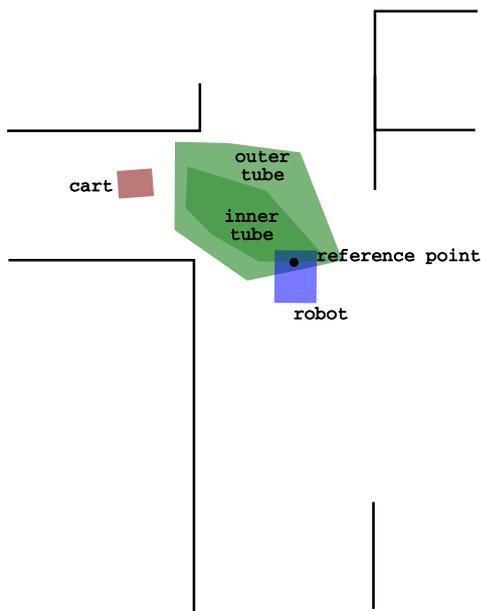


Figure 5.14: The second “tube” specification makes the robot traverse the junction, towards the corridor on the left.

The broader context requires the **composition** of several of the “primitive” task specifications introduced before. The task **plan** can have the three sub-tasks of Figs 5.13–5.15.

The first specification wants to bring the robot to the end of `corridor_3` in which it currently resides; it uses an “inner” and an “outer” tube to reach this goal. The robot must cross the `junction` towards the `corridor_4` where the `cart` is, but because the sensing and world modelling activities have placed an `obstacle` on the map, the specification in Fig. 5.16 has been added to the plan. That one first passes the obstacle “on the right”, before starting the third “nominal” task specification, Fig. 5.15. All specifications are materialized as particular instantiations of the generic inner-outer tube template.

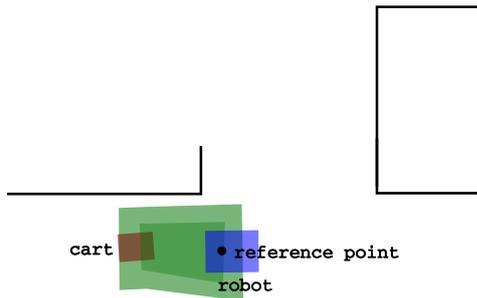


Figure 5.15: The third “tube” specification makes the robot move towards the cart.

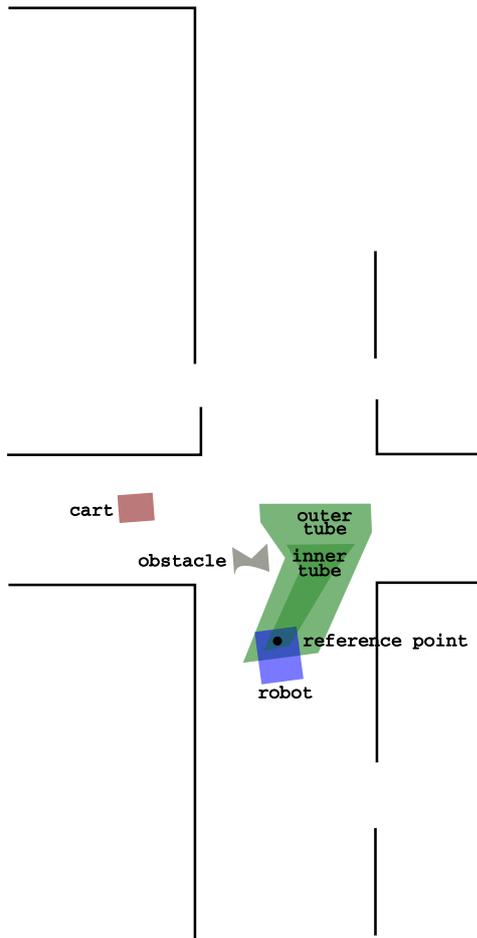


Figure 5.16: When in the task specification of Fig. 5.13, an obstacle shows up somewhere in the areas covered by the “tubes”, and obstacle avoidance task has to be introduced. This could be done as sketched in the drawing, which is just another specific instantiation of the generic inner-outer tube template.

5.11.4 Example task plan: escape from a room

This Section provides another example of composite task specification, linking the various specification parts in the [generic Task model](#): *to escape from a room*, Fig. 5.17.

One of its simplest possible **plans** has the following nominal *sequence* of **states**:

1. initialize sensors and motors, without motion control, perception or monitoring;
2. move forward till wall is detected, with the simplest possible control and monitoring;
3. move while *following* wall *on the right*, with somewhat more extensive control and monitoring, and with wall detection perception.
4. turn right at first large enough hole, with extra hole detection perception and monitoring.

5. stop.

The sequence above only represents the *nominal* plan, that is, it is not ready to cope with an execution of the robot's actions that would bring it in other states than the mentioned sequence. A **robust** plan, i.e., one that can cope also with non-nominal executions, must have monitors in all of its states, to check whether the sensor measurements still satisfy the assumptions that hold in each plan state, within a task-specific tolerance. Typically, a robust plan requires an order of magnitude more design efforts than a nominal one.

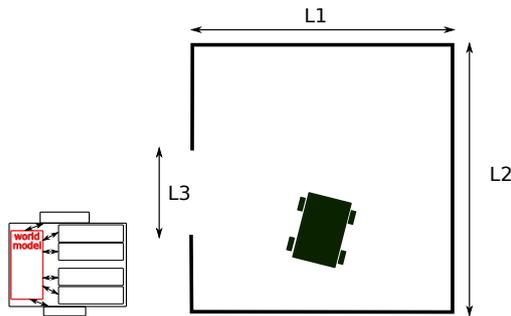


Figure 5.17: The **task** of the robot is to drive out of a rectangular room with a hole in one of its walls. The size and position of room and hole are unknown. The small figure on the left refers to the generic task model of Fig. 5.2, and indicates its parts that are involved; in this case, only the *world* is being modelled.

The **resources** available to realise the task are assumed to be:

- *laser range finder*: it provides at regular intervals in time an array of rays, regularly spaced in a range of orientations, and indicating the free space within a minimum and maximum range of distances.
- *encoders*: they provide the change of the robot's wheel rotations over time, and hence an estimate of the instantaneous velocity of the robot.
- *velocity control*: it tries to realise the instantaneously specified desired velocity of the platform, via control of the corresponding wheel velocities.
- *effort value*: one scalar that represents the percentage of “full” available power used for the current motion.
- *keyboard button*: events from the keyboard of the human operator.

At **initialization**, the following knowledge is **assumed** to correspond to the real environment of the robot:

- the robot is *inside* a room.
- the room has a *rectangular* shape as in the figure, with unknown lengths of the walls.
- the room has *one door*, *wide enough* to let the robot pass through.

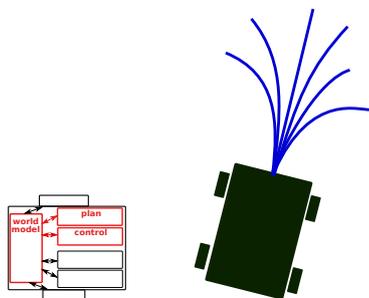


Figure 5.18: A possible *control* strategy for the escape-from-room capability in Fig. 5.19: to select from between a discrete set of pre-computed open loop trajectories, each corresponding to one particular time-invariant input at the actuators.

The **control** part of the task can as simple as making a selection between various “open loop” motion trajectories, Fig. 5.18:

- when a set of constant speeds is applied to each wheel, the result is a set of known trajectories of the robot in the near future. These trajectories can be obtained from a model only, or can be identified on the real robot.
- the sparsity and density of these trajectories can be chosen, in a plan-directed way, to reduce the computational requirements to a level that does not allow to separate between trajectories that are closer together than the sensing resolution, or than the tolerance allowed in the task specification.
- time and space horizons can be chosen for the trajectories, again in a plan-directed way.
- the *control* action can then be as simple as *selecting* the best open loop trajectory and apply the corresponding wheel constant velocities.

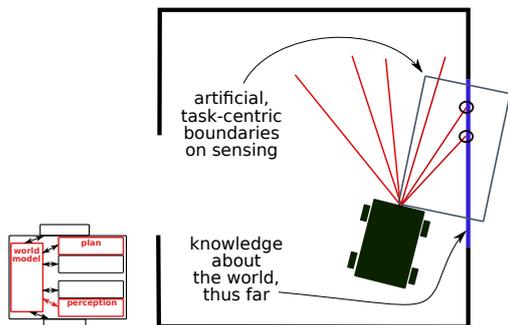


Figure 5.19: A possible *perception* strategy for the escape-from-room capability in Fig. 5.19.

The **perception** part does not need all the data provided by the distance sensor, since it could use the following algorithm (Fig. 5.19):

- select a *region of interest* (the grey box in Fig. 5.19) that fits to the *plan*, because the latter is only interested in the right-hand side of the robot.
- fit a *line* through a *large enough* cluster of measurement.
- do this over a *time window* of measurements.

In other words, perception is done by means of the least-squares fitting of a line of limited length, through a clever, plan-directed selection of current and previous hits of the scanner rays with obstacles.

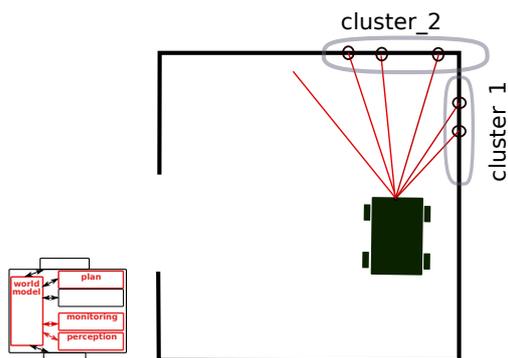


Figure 5.20: The *monitoring* strategy for the escape-from-room capability in Fig. 5.19 must follow the *wall on the right* (which is needed for the motion control) and look out for a *wall in front*.

The **monitoring** deals with finding which of the following four *hypotheses* gets most support from the sensor data:

1. one can fit a *wall on the right*, by looking only at a *local* horizon of measurements, as expected by the *task* context;
2. a further horizon in the *forward* direction is needed:
 - (a) to monitor whether there is “something”, to react to in the *plan*;

- (b) to find another *line cluster*, orthogonal to the first one, to update the *world model* with a new *corner*.
- the leftmost rays can be discarded, because they are outside of the scope of the *plan*, which reduces the computational load.
 - all* measurements *could* be *neglected* until needed again, based on the *planned* speed of the motion.

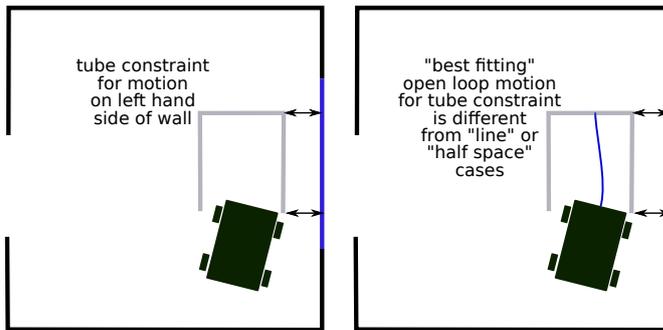


Figure 5.21: Left: a possible *motion specification*, in which a “tube” constrains the allowed motions, but does not command an explicit motion trajectory. Right: a corresponding *control* choice.

The **motion specification** needed in the **control** can be as simple as the “tubes” in Fig. 5.21:

- the robot is allowed to move anywhere inside a **tube** at some distance from the wall.
- it must make progress towards the next **waypoint** which is at the closed end of the tube.

The controller then selects one of the open loop trajectories of Fig. 5.18 that fits best, according to a task-specific metric. **Alternatives** for the **control** design are depicted in Fig. 5.22.

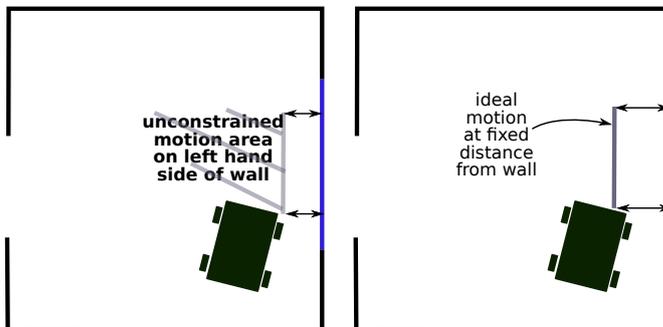


Figure 5.22: Two alternative controller approaches for the motion specification in Fig. 5.21. Left: an open half space, to the left of the wall. Right: a line trajectory at a specified distance from the wall.

The **world model** entities and relations needed in the **plan** are as follows:

- room has four **corners**, one door, and five walls.
- each **wall** is represented by two **nodes**, that is, a point in the 2D plane.
- the **robot** has a **pose** with respect to the room features.

This gives rise to the topology of Fig. 5.23. The geometrical properties are rather straightforward:

- every **node** gets two coordinate numbers, being its position in the room’s **frame1**.
- every **corner** gets a property tag representing that it is a straight angle.
- the position of the **robot** is given by the coordinates of its local frame in the room frame.

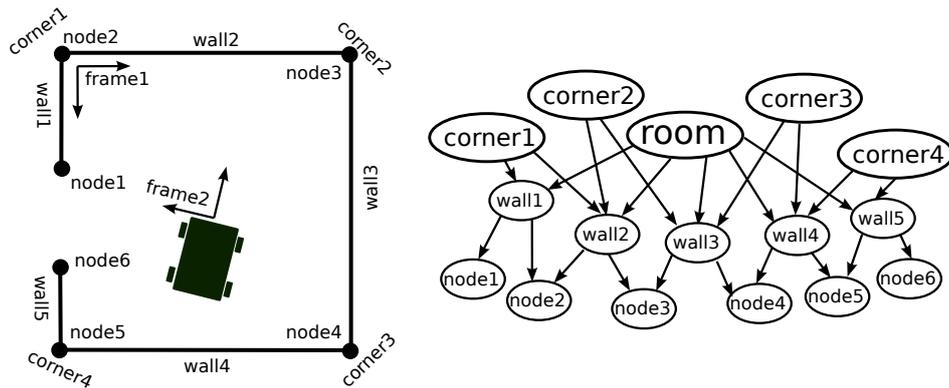


Figure 5.23: Right: topology of *room* model entities (“data structures”) and relations (**has-a**, and **connects**). Left: geometrical properties of all entities. The robot’s pose in the room can be represented numerically by position coordinates of the frame attached to the robot with respect to the frame attached to the room.

5.12 Domain-specific task specification languages

This Section gives examples of *domain-specific languages* that bring the specification of tasks (as introduced in the previous Sections) in line with the terminology used in particular application or technology domains. The provided examples are far from standardized, yet.

5.12.1 Task ontology and its standardization

Modelling all relevant task-level compositions is a huge undertaking, but will hopefully and eventually result in a (*standardized!*) collection of domain/application specific **ontologies**. This undertaking is not a main focus of this document, but all of the document’s contents serves as a foundation for that undertaking. The good news in the short-term is that even the “poor” mereological top of this structure as introduced here is very useful to support discussions between human developers, not in the least to make the scope of their developments explicit.

Sooner or later, the **stakeholders** in the **robotics domain** must **agree on a *standardized ontology*** for all these terms, because that is the **only** way to realise a vendor-neutral digital platform that can serve as the basis for composable innovation. (After more than half a century of robotics industry, there are still no significant results in the direction of semantic standardization of the field.) The second added value of creating a widely supported domain ontology with a **hypernym–hyponym hierarchical structure** is for the robot controller software to exploit: **only** when the mentioned ontological information is available, at **runtime** and in **formal representations**, one can expect robots **to reason** about their actions on the “most appropriate” level of abstraction, **to assess** whether what they are doing corresponds to what they are supposed to do in their current **task**, and **to adapt** their action plan accordingly.

5.12.2 Semantic mobile robot motion primitives

Abstracting a bit from the concrete examples in the previous Section, the following semantic motions could form the basis of a mobile robot’s “**platform** motion stack capabilities”:

- *start-to-cruise* (and its *inverse*, *cruise-to-stop*): how to get the robot start its motion and reach a “cruising” motion behaviour, when all it has to do is to drive “straight ahead” within its current lane, and that lane continues till “far” beyond the dynamic bandwidth of the robot.
- *cruise through tubular area*: the *world model* for this semantic motion has landmarks on the robot are geometrically constrained by a “tubular” area in the Cartesian space.
- *overtake obstacle during cruise*: this is a composite cruising task, with large-changing motion capabilities added. Reference [45] contains already a very worked-out formalization of this semantic motion primitive, at a mereo-topological level of abstraction that fits to that of this Chapter.
- *cruise to approach*: this is an extended version of the *cruise-to-stop* primitive, in that the “approach target” semantic tag adds extra constraints on the motion behaviour, such as optimal/expected relative motion positions and orientations, and relative motion speed profiles.
- *approach to stop*: similarly, this semantic primitive adds extra stopping behaviour, determined by properties of the approached target.
- *approach to turn right/left in tubular area*: this primitive adds extra behaviour of how to connect two *cruise through tubular area* motions, one before and one after a “crossing”. The tubular area in the *world model* has specific landmarks that guide the robot to make a right (or left) turn. Examples are: traffic lane indicators on the ground; or “walls” in the built environment as well as in the natural environment (trees, bush, river,...).

5.12.3 Semantic robot arm motion primitives

The mobile robot example in the previous Sections is conceptually the easiest to grasp, because the world is mostly flat, *and* the shape of the robot is mostly constant. For arm-based robotic systems, the full 3D Cartesian space and the full n D joint space are to be taken into account, but at the mereo-topological level of abstraction, very similar semantic motion primitives can be defined. Figure 5.24 sketches some simple examples, with two levels of resolution in representing the mentioned configuration spaces.

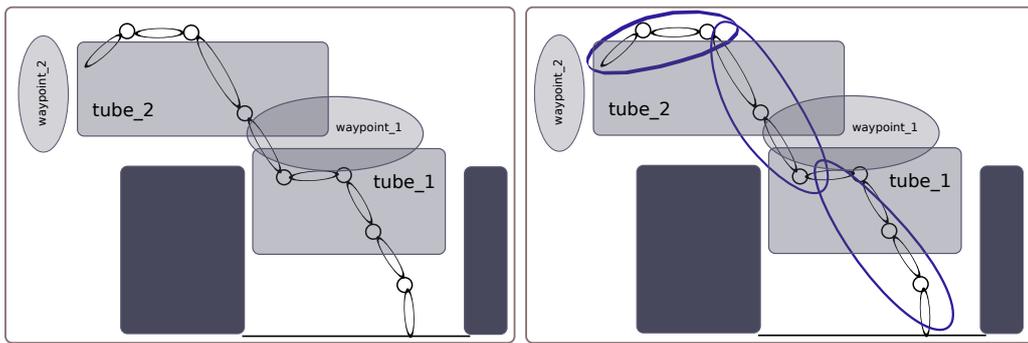


Figure 5.24: Model of a “high” (left) and a “low” (right, in blue) kinematic resolution motion plan for a serial robot arm during a sequence of tubular motion between obstacles.

Chapter 6

Meta models for dynamic semantic maps and situational awareness

The role of **world models** in a system is to provide **context** to a component, that is, information that the component can not deduce from its own models, actions and sensors. This “definition” is far from constructive, because it refers to what is *not* there. In addition, it is time-varying, because, for example, a robot component can extend the models it has “on board” with some of the world model parts, reducing the role of the latter. Or rather, not the *role* of the world model, but the *place* in the system where it is located. Indeed, this document’s approach is to foresee a role for *a* world model in *any* component, and to avoid relying on *the* world model being available for all components in the system, at all times. In other words, this Chapter introduces mechanisms and policies *to create* world models, *to query* them, *to update* them, and *to connect* them. The *architectures* of how and where a system introduces world models is the subject of [later Chapters](#).

The [paradigmatic](#) parts that this document provides to describe the role and the design of world models, are those of:

- the [association hierarchy](#) in perception.
- the [association hierarchy](#) in control.
- the [Task-Skill-Resource](#) meta model.

Any composition of models in one or more [levels](#) in the three above-mentioned structures gives rise to the need to give a world model context to that composition. [Geometry](#) is (again, paradigmatically) chosen as the [spatial “database”](#): where are objects with respect to each other in time and space. That database serves as the foundation for all other relations in a world model:

- *geometry–geometry*: the representations of the *shapes* of objects and robots, and how they shape the *motion constraints* between objects.
- *geometry–perception*: the representations of how properties of objects are *detectable* in sensor data.
- *geometry–motion*: the representations of how properties of objects are *targets* of the robots’ motions.
- *geometry–task*: the representations of *actual, desired, hypothesized, . . .* states of the world, depending on the task requirements.

A world model is called **complete** if its relations provide **causal explanations** for all **magic numbers** in the model.

Chapter 7

Meta models for continuous control

Control is that part in a system’s **task model** that is responsible for realising the behaviour of the system. This Chapter focuses on the **continuous** domain¹ only, that is, on the control activities that:

- **put energy into the system**,
- by deciding what the actuator signals are, that can bring the “world” from its *actual* state to its *desired* state,
- with algorithms specified in the **plan**.

This Chapter introduces the **mereo-topological** meta model of control (that is, feedback, feedforward, adaptive, and predictive control), and some of the natural structures and “best practices” in the continuous control domain.

Discrete control is the natural complement to continuous control: during any ongoing continuous control activity, one has **to monitor** all the **constraints** whose violation indicates that it might be time to make the decision to switch to another controller. This kind of decision making about switching control behaviour is what this document calls *discrete control*. For this Chapter, the discrete control comes in only superficially,² via (i) the monitoring algorithms that work on the same (or closely connected) continuous system configuration space as the control algorithms, and (ii) the simplest form of discrete control decision making, namely the **Finite State Machine** (FSM).

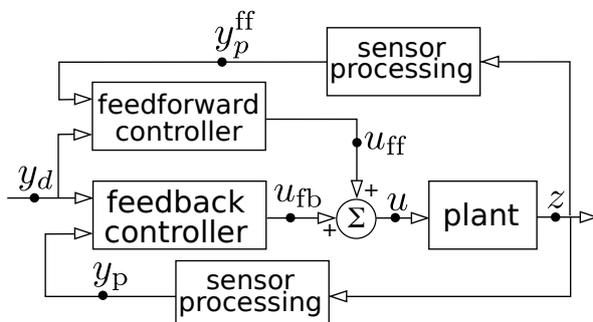


Figure 7.1: Feedback and feedforward control both contribute to the actuation signal u that is given to the plant. And they both can work with *processed* versions (y_p and y_p^{ff}) of the current state z of the plant.

¹“Space” or “effort” versus time, for example.

²Sections 2.7, 2.8 and 2.10 provide more in-depth discussions on the coordination mechanisms for discrete control.

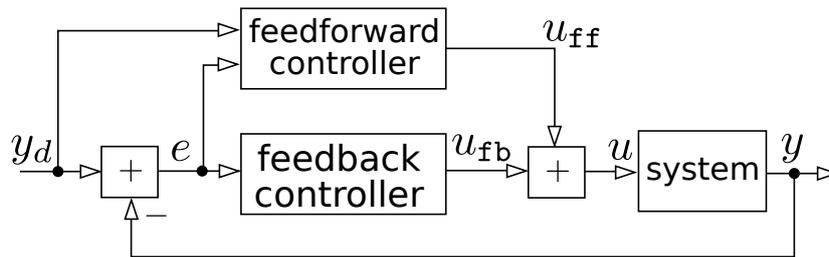


Figure 7.2: Feedback and feedforward control loops: simple version.

7.1 Mereology: feedback, feedforward, predictive, adaptive, preview

Figures 7.1–7.2 show the simplest case of a control system, that is, with only feedback and feedforward contributions; Figures 7.3–7.5 show somewhat less simple cases. All of them consist are a **composition** of the following contributions to the actuator input signals:

- **feedback**: this function generates the actuator signals based on the so-called “**error**” between the desired state of the world and the actual state of the world. (As far as that actual state can be **estimated** from the raw sensor information.)
- **feedforward**: this function computes actuator signals based on (i) the actual state of the world, (ii) a **model of how the system is expected to react** to potential **instantaneous** actuator signals (in other words, a **simulation** of the system), and (iii) the **smallest predicted error** over all the evaluated actuator signals.
- **predictive**: this is like feedforward but the simulation-based optimization is not done instantaneously, but over a chosen **time horizon** into the future, Fig. 7.4. Often, one also chooses **to limit the search space** of actuator signals to consider in the optimization,
- **adaptive**: this function does not generate actuator signals itself, but it changes the control **behaviour** indirectly, by adapting one or more **parameters in the models** of the feedback and/or feedforward functions. The basis of adaptation is the **observed history** of control signals and system states.
- **preview**: this function changes the **structure** of the control model, because it adds a model of (part of) the control to be realised in the **next task** in the sequence of task specifications.

Parameterizing the feedback, feedforward, predictive, adaptive and preview functions provides for higher configurability in the control system, in particular to reuse the same implementations in larger ranges of operational conditions.

Typical examples of adaptive control adapt one parameter in the feedback model, or in the feedforward model:

- many feedback loops first perform a first-order **low-pass filtering** on the measurements or on the error signals, and the **gain** of that filter can be adapted to the observed **noise level** of the signals.

- relevant feedforward parameters are the values of the (one-dimensional) **inertia**, damping/**friction**, or **elasticity** in the mechanical part of the system. The value of that parameter in the feedforward model can be adapted on-line by monitoring the *trend* of the feedback error: is the error is “systematically positive”, the parameter can be decreased, and when the error is “systematically negative” it can be increased. Only one of the three mentioned mechanical parameters can/should be adapted in any one-dimensional adaptive controller. For example, if one tries to compensate for friction and inertia by changing one only feedforward parameter, one can end up in a situation where neither of both values is compensated well.

An example of preview control is that, when moving a robot hand towards a door handle, one can already start controlling the opening of the hand, as well as its orientation with respect to the door, starting from a certain parameterized distance to the door. The result *could* be that the hand is already better aligned when it is going to have to open the door in the next sub-task.

A preview controller *can* improve the execution time of a task by “looking ahead”, but (i) the performance improvement may be hard to guarantee, and (ii) it definitely comes with extra computational costs.

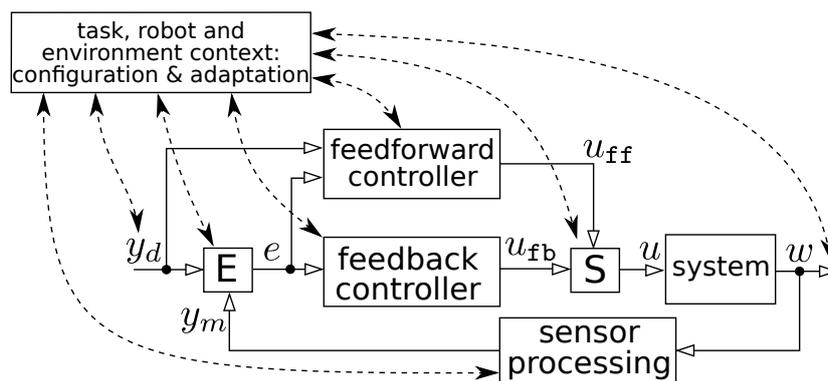


Figure 7.3: Feedback and feedforward control loops, in a somewhat less simple version: it indicates that all of the “magick numbers” in a control scheme can only be given their appropriate values when the full context of the application is taken into account. In other words, a controller should never be designed in isolation. Nevertheless, this sketch does exactly this, because it takes the *setpoint error* as the main input to the controller computations. Hence, it considers only the *instantaneous error* between desired and actual state value y . (This is a more **context-dependent** version of Fig. 7.1.)

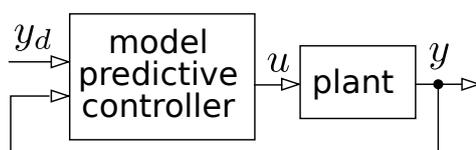


Figure 7.4: Predictive controller.

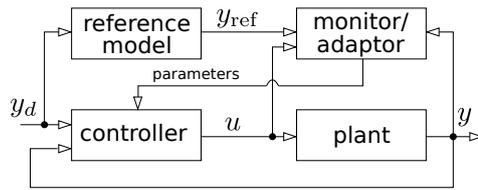


Figure 7.5: Model-reference adaptive controller.

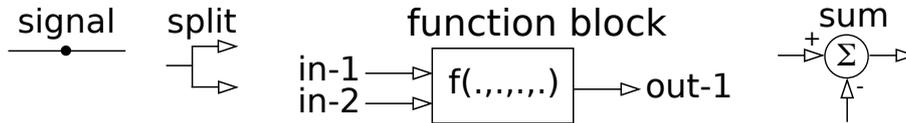


Figure 7.6: The four entities in the controller meta model that conform to the algorithm meta model: the data blocks `signal` and `split`, and the function blocks `function-block` and `sum` block.

7.2 Information associations in control

Figure 7.7 depicts the natural hierarchy in control of mechanical systems.

(TODO:)

7.3 Mechanism: state and system dynamics relations

Any controller (or **control diagram**) is the composition of only four entities (Fig. 7.6) that conform-to the `algorithm` meta meta model:

- **signal**: the **data** that “flows” between two **function** blocks, that is, the input and output parameters of the functions. The numerical representation of a signal is a traditional **data structure**.
- **signal split**: special case of a “**signal**” that appears in quasi every control diagram, and that represents the fact that the *same* signal is used as inputs to more than one **function** block. The split has a *direction*: there is only one connection to an output of the **function** block that creates the **signal**, but the **signal** can be the input of more than one **function** block.
- **function block**: this is a *pure function*, with one or more **signals** as inputs and one or more **signals** as outputs.
- **sum** block: this is a special case of a **function** block, that appears in quasi every control diagram, and that gives as its output the sum of all its inputs, each possibly with a “-1” sign inversion.

The following concepts are common to all controllers, so the meta model adds them as first-class modelling primitives:

- **setpoint**: an entry point of a controller. Its meaning in the context of an application is that this signal gives the *desired* value of the **plant** state. The **setpoint** has no further connections at its own “input” side, at least not inside the scope of the current control

- **plant, or system:** this is the part of the real world whose **state** the **control-diagram** tries to influence, and with which it interacts via **sensors** and **actuators**, These convert physical values into digital **signals**, and back.
- **feedback loop:** a **function** block that takes **setpoint** and **measurement signals** as inputs, and computes a **signal** that can be used by an **actuator**. The topology of **feedback** is a “loop” because the **plant** couples the **measurement** and the **actuation**.
- **feedforward chain:** a **function** block that takes **setpoint** signals as inputs, and uses a *mathematical model* of the **plant** dynamics to compute a **signal** that can be used by an **actuator**. It is *not* a loop, but a serial composition of function blocks.
Of course, *adaptation* does bring a loop structure, but adaptation is a **higher-order** composition relation.
- **sensor:** a **function** block that gives **signal** values to some physical values of the **plant**.
- **actuator:** a **function** block that converts **signal** values in the controller to physical values of the **plant**.
- **adapter:** a (higher-order) **function** block that takes a **state** of the **control-diagram** as input, and computes a **signal** that another **function** block in the **control-diagram** can use to change the value of one or more of the parameters it uses in its computations.
- **monitor:** a (higher-order) **function** block that takes a **state** of the **controldiagram** as input, and computes an **event** for the application software that uses the controller. That event is not used in the **continuous** **controldiagram** itself, but it *can* give rise to a change in one or more parts in the **control-diagram**, via the **discrete** control part of the system.
- **control-diagram:** this is the **composition** of all of the above, which conforms to the constraints of the controller meta model, discussed below. It has **one trigger** entry point, to execute all computations inside the diagram. To this end, every **control-diagram** contains a data structure to represent its **schedule**. In other words, the **controldiagram** is the model of the function that (i) takes the **output state** of the **plant** as one of its arguments, (ii) takes the *desired state* of the **plant** as its **setpoint** argument, and (iii) from them, computes the **input** that should be applied to the **plant** to make the latter evolve towards the desired **state**.
- **delay, or buffer:** signals with different **timesteps** are sometimes used as arguments in the same function. Hence, the controller function must provide a mechanism to store signals for a finite amount of time, and these are the **buffers**. They contain functions and data structures.

In general, a **control-diagram** model is a **cyclic graph**, because of the presence of feedback loops. However, there is one very important property of that graph: each cycle corresponds to a **different sample-time**, hence, time is the appropriate entity to **unroll** the cycle into an **event loop**,

Properties of control behaviour (that (only) show up during the **execution** of the corresponding event loop implementation) are:

- **latency**: the difference between the desired timestamp of the execution, and the actual one.
- **loop-time**: the duration of the execution of once cycle of a **controldiagram**. Ideally, that duration is zero, because that would mean that the actuation signals are applied immediately after the sensor have been read. Of course, this is not feasible in practice. The best one can achieve is that the **looptime** is “sufficiently smaller” than the smallest time constant in the system that one wants the controller to react to. Of course, these **magic numbers** are fully dependent on the application context.
- **jitter**: the statistics of the **latency** and **loop-time** values over a certain period. Especially the worst values are important, because they give rise to the largest **disturbances** that the execution of the control introduces to the desired controlled system behaviour.

Not all **compositions** of entities in the meta model result in meaningful **control-diagrams**, since the following **constraints** must be satisfied:

- **feedback loop constraint**. The following chain *must* be present: **setpoint**, **feedback**, **actuation**, **plant**, **measurement**.
- **feedforward chain constraint**. The following chain *must* be present: **setpoint**, **feedforward**, **actuation**.
- **cascaded feedback loops constraint**. More than one **feedback loop** can be present in the same **control-diagram**, and the proper composition of an “inner” and an “outer” loop requires that the **setpoint** for the “inner” loop is an **actuation** signal of the “outer” loop.
- **adapter chain constraint**. Any **adapter** has at least one **state** as its input, and its output goes into a **feedback**, **feedforward** or **monitor** block.
- **monitor chain constraint**. Any **monitor** has at least one **state** as its input, and has no output that goes into a **feedback**, **feedforward** or **adapter** block.

7.4 Mechanism: optimal control

The control meta meta model as introduced above, still allows multiply ways of *how* the control computations are realised. **Constrained optimization** has become a popular approach, because:

- it forces designers **to formalize explicitly** what their design objectives really are.
- it is a perfect fit in the **Hybrid Constrained Optimization Problem (HCOP)** formulation of robotic tasks, introduced in Sec. 5.5. That formulation is *even more generic*, because it integrates the **symbolic**, **discrete** and **continuous** aspects of system behaviour design. The current Chapter deals with only the continuous part of this integrated approach.
- **monitoring** of the values of the objective functions and the constraints in this continuous part is the link from the continuous control part to the **discrete control** part. The link in the other direction comes from the **plan**, where the events are one aspect of the information used to make the decisions to switch continuous control behaviour.

- **reasoning** on symbolically represented knowledge is the third **symbolic control** part. The “feedback”, “feedforward” and “adaptation” at that level correspond to deriving which symbolic relations are relevant to which decisions, and in which way exactly they influence the decision making.

In other words, it brings in the **contextual information** that helps to formulate the *application-*, *environment-* and *platform-*specific objective functions and constraints for the continuous and discrete controllers.

Figure 7.8 depicts the generic formalisation (“meta meta model”) of a continuous-domain constrained optimization problem with which to compute a control signal:

task state & domain	$X \in \mathcal{D}$
desired task state	X_d
robot/actuator state & domain	$q \in \mathcal{Q}$
objective function	$\min_q f(X, X_d, q)$
equality constraints	$g(X, q) = 0$
inequality constraints	$h(X, q) \leq 0$
tolerances	$d(X, X_d) \leq A$
solver	algorithm computes q
monitors	decide on switching

Figure 7.8: Generic formulation of a *constrained optimization problem* (COP).

The **domain** fills in the *types* for f , X , q for a particular “robot”, and a particular *type* of solver. The **application** then adds choices for the *parameter values* for f , X, \dots , and the concrete solver and monitor *implementations*.

(TODO: concrete examples.)

7.4.1 Policy: PID and pole placement for linear systems

Textbooks on control theory (in both the “analog” and the (discrete) “**state space**” versions) most often work under a very hard assumption, namely that the system to be controlled has an **open loop dynamical model** that is **linear**. That means that the **superposition principle** is assumed to hold: any linear combination of control signals gives rise to the linear combination of the reactions of the controlled system to the individual control signals. No real-world system satisfies this assumption, for a large number of different reasons.

PID control and **pole placement** are two mainstream approaches in analog and discrete control design. The methodologies to design such controllers are optimization-based too, (albeit **offline** and not online, as is the ambition of this document): the methodology selects the control parameters as the “best” trade-offs that can be made between **gain and phase margin**, for **any** possible combination of setpoints and input signals. In other words, to keep the error value as small as possible, under all circumstances, and to keep the deviation of the signal *shape* of the controlled system as close as possible (in the *time domain*) to the signal shape of the setpoint.

7.4.2 From PID and pole placement to optimal control

This Section introduces a non-exhaustive list of assumptions that the PID or pole placement approaches make, and for which it is often not very difficult to find alternatives. A major reason for the sustained popularity of the traditional control approaches is **human inertia** against change, culminating in the slow update of control theory curriculum at universities. This document stimulates⁴ the transition from the “offline optimized” control paradigm of PID and pole placement to the “online optimized and monitored” control paradigm of optimal control.

- control action scales linearly with magnitude and sign of error.

This most often leads to undesired large control actions for large errors.

- every sample time, the full controller is executed, and the horizon over which the control reacts is very short.

Typically, that horizon is only one time sample deep, but it is possible to extend the state of the system with its states at different time instants.

Indeed, some signals do not change significantly during that short period. More in particular, the computation of the “error”, and transforming that error into a feedback actuator control signal, can often be done at much lower time rates than updating the feedforward actuator signal; especially for mobile robots, the same feedforward actuator signal can be held for a long time and feedback must only be applied when the robot runs a risk of going out of its “lane” somewhere in the future.

- there is no semantic difference between a *positive* and a *negative* error.

In mobile or flying robot applications, a symmetric reaction to an error implies a loss of actuation energy: when the robot is moving a bit too fast, there is seldom a need for the controller to slow down actively (and hence spending energy on that braking) because the friction that is always present in the system will do that braking job for free.

- *every* error must be reacted to, or, equivalently, *only a zero error* makes sense to strive for.

Many applications can tolerate a **dead zone**, because reacting to errors smaller than a relevant threshold adds no value.

- *optimality* is the enemy of **good enough**.

All optimizations (whether they are done offline as in pole placement, or online as in optimal control) are based on models, and models only. That means that the quality of the “optimum” depends heavily on (i) the quality of the model (does it contain the “right” parameters?) and (ii) the quality of how well one knows the values of these parameters. Hence, spending a lot of efforts on finding the optimum most often makes very little sense. It can even be dangerous because many optimization problems have a high sensitivity in the optimum: a small change in parameter values can give rise to a large change in the optimum. For example, the “best” road for a mobile robot to follow

⁴Without forgetting that there are indeed still many simple control challenges for which the traditional approaches are impossible to beat.

when traversing rough terrain is very sensitive to whatever road quality parameters one chooses, and even more to the measurement principles and devices one has to rely on to find those parameters.

The following Sections describe alternative control design approaches, that want to cope with one or more of the above-mentioned simplifications explicitly.

7.4.3 The role of PID and linear control in robotics

A PID controller deals with **one signal only**. In other words, it represents a **SISO** (Single-input single-output) control system.

In any robot of somewhat realistic complexity, the forces generated by each motor influence all “error” signals. In other words, these are **MIMO** (Multiple-inputs multiple-outputs) control system. The good news is that the knowledge exists **to decouple** all controlled degrees of freedom, via a **dynamics solver**. Such a solver computes the **feedforward** part, and this yields *not* a linear system but a *linearized* system:

- when computing the control efforts *in the right* order, each of them requires solving only linear equations.
- but *superposition* does not hold.

The result is something that looks a lot like linear control but isn’t. Not in the least because, despite the decoupling, the inertia that is felt by each motor changes with the configuration of the robot, and this introduces an (often significant) non-linearity. *Parking a car* is an illustration of this non-linear configuration dependency: at any moment in time, the car can just *not* move in a direction orthogonal to the orientations of its wheels, Fig. 7.9.

Daily experience tells us that the real controller must be a **hybrid controller**: it solves the control problem of parallel parking as a sequence of *back-and-forth* tasks, with feedforward actions on the steering and driving wheels, with monitors to stop the selected feedforward action, and continuing the process until the car is “well enough” into the parking spot.

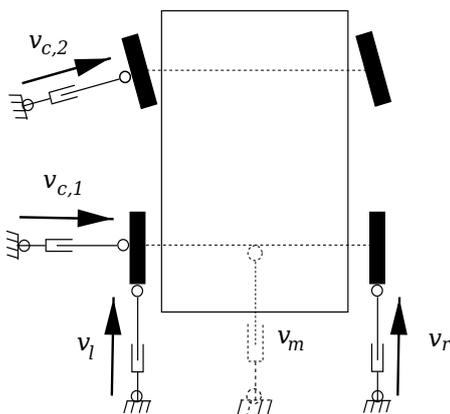


Figure 7.9: A mobile robot, or a car, are major examples of systems that can not be controlled by one single linear controller: there is no way to move the car in the directions orthogonal to its wheel orientations, and that direction changes non-linearly with the steering angle.

7.4.4 Policy: model-reference adaptive control (MRAC)

One particular **policy** of optimal control has been given the name **Model Reference Adaptive Control** (MRAC).

Its computation uses a model of the desired plant behaviour, to adapt, “optimally”, some parameters in the feedforward and feedback models.

7.4.5 Policy: model-predictive control (MPC)

Another particular *policy* of optimal control has been given the name **Model Predictive Control** (MPC).

It is a special case of considering the design of a controller as an **optimal control** problem. Its computation uses a cost function that includes samples along the *full* predicted trajectory.

7.5 Mechanism: setpoint, trajectory, path and tube inputs

This Section introduces a *natural order* in ways to specify the goal of a controller, that is, in choosing the type of **input** that the controller accepts, The order is from *most constrained* to *least constrained*, where the latter gives the control designer the most degrees of freedom in the design process:

- **setpoint**: only one single **instantaneous value** of the **desired state** of the plant is being used in the control computations. In other words, the **control horizon** is only one time instant “deep”.
- **trajectory**: instead of just one instantaneous desired value for the **plant state** as input to the control design process, a trajectory of desired **plant state** values at multiple sample times over a certain horizon is used. This gives the designer more freedom to spread the inevitable control error budget over a larger space.
- **path**: another mechanism is to use a **path** instead of a trajectory. This is a less constraining input, because the **time is not imposed**. In other words, the **state** is *constrained* to follow the geometry of the path in state space, but not any timing along that path.
- **tube**: this is the least constraining input, because the controller can now also deviate from a given path, as long as the resulting path keeps the plant state inside a “tube”, or “region”, in the **state** space.

The design choice about what is the *input* to a controller defines, implicitly, also the interpretation of what is expected as an **acceptable (control) error**.

7.5.1 Policy: composition of optimal control and tube inputs

The optimal control approach of Sec. 7.4 allows the designer to make the choice of “error model” explicit, by adding corresponding constraints to the problem formulation. In other words, the PID and pole placement methodologies require generically valid **stability** functions, while the optimal control methodology allows the designers to make their specific choice of what it means to have a “stable controller” in the specific context of their application, and even to adapt the control behaviour to the concrete state of the system.

This flexibility comes with (guaranteed) extra costs: the obvious computational costs to be paid by a more complex online solver, but also often less obvious costs of forcing the

designers to come up with motivations about what are the “appropriate” specific choices to make.

The flexibility also comes with (potential) added value: the controller need not be designed to react to *all possible* cases with one and the same control function, but it can adapt its instantaneous behaviour to the actual state of the optimization problem: More in particular, the controller can react differently depending on, for example:

- which constraints are violated.
- the sign and/or trend of the error.
- the selection of error function.
- the closeness to actuator saturation.
- the measurement range accuracy and/or noise level of the sensors.

Another form of flexibility that (potentially) adds value lies in the choice of solver for the constrained optimization problem. Some examples of this flexibility are:

- **stop anytime** solver: when a given computational budget is used up, the best available solution to the optimal control problem *at that moment* is used as the input to the actuators.
- **good enough** solver: many tasks do not require the control to be executed optimally, so the solver can stop as soon as the solution it has found is “good enough”.

From an [explainability](#) point of view, it makes sense to add the information of what is “good enough” as an extra set of constraints to the optimization problem. Indeed, that allows to introduce monitors around these constraints, that are hence as close as possible to the source of the decision making.

7.5.2 Policy: control progress objective or constraint

For setpoint and trajectory control, the *objective* of the controller is the same: to reduce the **error** to zero. But *path* and *tube* specifications are not constrained enough to determine the behaviour of the controller completely by a “zero error” condition. Indeed, a **progress specification** must be introduced explicitly in the optimal control formalisation of the controller:

- **progress objective function**: the closer the executed motion of the robot brings this objective function to its extremum, the better the progress.
- **progress constraint**: as long as the executed motion of the robot results in this constraint to be satisfied, the current control action can go on.

7.5.3 PID alternatives: sliding mode, gain scheduling, ABAG

Section 7.4.1 explained which set of assumptions are hard constraints behind the PID control design methodology. This Section gives an overview of control design approaches that allow to loosen some of these constraints explicitly:

- an **error area** (instead of an error value) is used **to select** (Fig. 7.10):

- the **feedforward** part, e.g., in **sliding mode** control.
- the **feedback** part, e.g., in **gain scheduling**.

It is simple to formalise these areas as inequality constraints in the optimal control model.

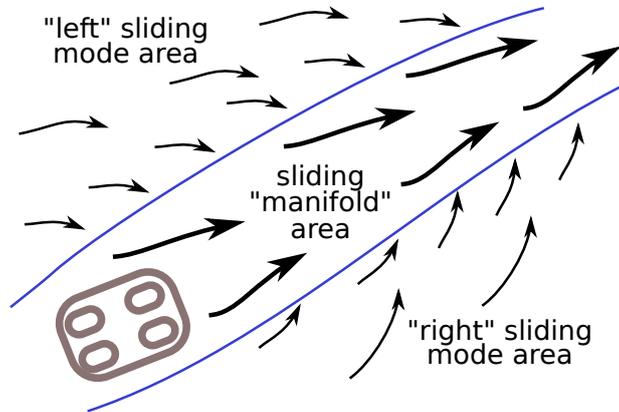


Figure 7.10: The concept of *sliding mode control*: the state space of the robot is divided into areas, each with a particular *a priori* determined control approach. In the strictest version of the concept, the middle area is the specified *trajectory*.

- the **trend** in the error is used **to adapt** the **feedforward** part, with **ABAG** control [40] as a representative of this approach.

The control computes a **bias** (the “B” in “ABAG”) that represents the control signal needed to keep the plant in its current state, and that follows the trend in the error at “slow” speed, while there is also a **gain** (the “G” in “ABAG”) to react “fast” also to the direction of the error. (The “B”s in “ABAG” indicates that both parts are **adaptive**.)

- the reaction to an error **need not be a linearly scaled version** of that error, but any “appropriate” **waveform** can be chosen instead.

Both sliding mode and (a slightly generalized version of) ABAG allow to decide what is “appropriate”, depending on where the system is with respect to the desired area, and/or how far away the actuators are from **saturation**.

- the control waveform can also be chosen to guarantee a **limited frequency** content. This is one of the ways to improve **stability**).
- the *saturation* criterium can also be applied to the **magnitude** of the control signals. This criterium can be composed with the above-mentioned criterium of “distance” to state space areas.
- one can apply saturation limits on the feedforward and feedback signals **separately**, Many PID implementations also use this approach, known as *gain* are **limited** to chosen maximal values, *before* they act on the control signal and not afterwards, as in the case of the controller’s **anti-windup policy**.

An attractive feature of a PID controller is that it requires only *one single and simple algorithm*, irrespective of the error or the setpoint. The flexibility in the other control approaches

comes at a price (because the algorithms are more complex), but also with a benefit, because only marginally extra effort is required to make a controller **hybrid and adaptive**.

7.5.4 PID versus ABAG

An ABAG control algorithm is almost as simple as a PID algorithm, and also their application contexts are similar:

- both can only be applied to one-dimensional control problems, only.
- both have a small number of parameters to tune: P, I and D for PID, and B, and G for ABAG.
- G has the same role as P: it is the *feedback* control signal that reacts directly to an error.
- in the control contexts of this document, models of the dynamic behaviour of the system-under-control are not the exception, but the rule; and, hence, the D part is replaced by a *model-based feedforward* part.
- the B term in the ABAG has a similar role as the I in PID: it is the (*error-based*) *feedforward* signal that builds up (or down) over time depending on the **trend** of the error. For an ideal system in **steady state**, the feedforward contribution via B or I realises the full control, without any contribution of the feedback part G or P. For example, it generates the force to compensate for gravity, or for friction.
- the contribution of the integral term in a PID grows with, both, the *magnitude* of the error, and the *duration* of the error. The result is often more difficult to predict than the impact of the B term in an ABAG controller:
 - the adaptation of B on the basis of the error can be made more deterministic than the adaptation of the I. For example, by allowing only a known waveform type, and with saturation. Hence, stability is easier to guarantee.

In the PID case, the building-up (and reduction) of the I term depends on the concrete error *magnitude*, which can not be predicted offline.
- ABAG has **anti-chattering** built-in: the explicitly specified **dead zone** around the zero value of the error makes the controller react only after the magnitude of the error is larger than the specified threshold. The dead zone need not be symmetric for positive and negative error values.

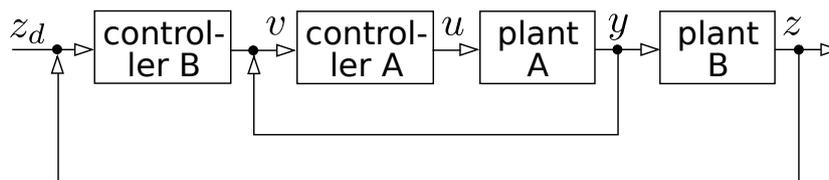


Figure 7.11: Cascaded control loops.

7.6 Mechanism: cascaded control loops

The natural hierarchy in the physical domains that are relevant in robot system control, and especially the differences in the natural time constants in these domains, leads to the **best practice of cascaded control loops**, Fig. 7.11: the innermost loop deals with the control of the fastest physical time scale (in particular, the DC-to-AC conversion), and the loops around it cover the next time constants in increasing order: torque, acceleration, velocity and position. In every loop, a new part of the plant dynamics must be taken into account; e.g., the inner loop sees the electrical dynamics of a motor, and the loop around it also sees the mechanical inertia.

(TODO: figures.)

7.6.1 Composition of energy and cascaded control

The hierarchy in **cascaded control loops** has the following parts:

- **power inverter**, with a typical⁵ **time constant** of 1/100.000th of a second.
- **electrical motor**: transforms electrical energy and power into **mechanical work and torque**, via mechanisms such as **field oriented control**, with a typical **time constant** of 1/10.000th of a second.
- **mechanical acceleration**: the generated torque causes acceleration of the attached mass; the time scales required in robotic applications lie around 1/1000th of a second and up.
- **mechanical velocity**: time-domain integration of acceleration results in velocity; again, an order of magnitude less in required time scale is order.
- **mechanical position**: further integration into position is the final step in the mechanical level of abstraction, with again an order of magnitude lower time scale.

The *battery* is not part of the cascade hierarchy because:

- its time scale is determined by the *chemical dynamics* of conversion of chemical energy into electrical energy, and this is orders of magnitude slower than the dynamics inside the electrical DC-to-AC energy conversion.
- its energy production need not follow the time scales of the control, since that is the responsibility of the **DC-to-AC power inverter**.

For the sake of concreteness, the example of a *battery-driven mobile robots* (e.g, Fig. 7.12) is used to illustrate the structure in the various energy transformations that occur in a robotic system:

1. **battery** as an electrical **energy source**: it can provide electrical energy (current and voltage) via well-known impedance relations, and with constraints on maximum and minimum power, voltage levels, temperature dynamics, etc.
2. **energy transformation** between AC or DC **electrical energy** into AC energy for (a)synchronous motors: the battery energy is transformed into mechanical energy via impedance relations of multi-phase, multi-pole motor models, and with constraint curves for torque, speed and efficiency.
3. **energy transformation** via a **mechanical transmission** from the electrical motor to the mechanical joint: the joint “consumes” not only the electrically generated torque

⁵At least for electric motor actuators.

for its own motion, but the dynamics of the whole chain are coupled in. A transmission can, itself, introduce extra (mechanical, thermal, . . .) dynamics, for example, via friction and elasticity, heat generation, . . .

4. **energy transformation** between the **joints and kinematic chain** to produce **Cartesian space motion** of end effectors (and other link attachment points): these are the “hybrid dynamics” introduced in Sec. 4.8.
5. **task specification** relations between **Cartesian attachment points on a robot and motion targets in the environment**: *only* when the robot is in contact with objects in the environment, the interactions are dominated by mechanical relations of the same type as those of the kinematic chain, but contact-less “interactions” are often *specified* as *artificial* constraints of the same type as the physical constraints.
6. **task specification** relations between **multiple moving robots**: again, these coordinated motions are not impacted by physical relations, but only via *artificial* ones.

The model semantics speak about “energy transformation”, because that is the more declarative and non-instantaneous way of expressing the relations, instead of the imperative and instantaneous terms “force” and “velocity”. (This is also the difference in approach represented by the (equivalent!) **Newton-Euler** and **Euler-Lagrange** theories of dynamics.) Instantaneously, the energy is indeed transmitted via *forces* (Fig. 7.12), and this is a very important fact that *must* be represented faithfully in all models (and hence also software). Indeed, different sources of force can be *added* together instantaneously, which is a major instantiation of the **composability** ambition of the modelling efforts. Position-based relations (positions and poses and their time derivatives), however, are *not* composable consistently in an additive way.

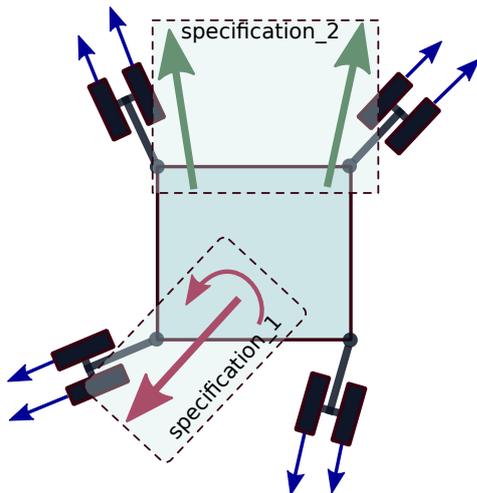


Figure 7.12: Transmissions of forces between actuated wheels (the blue “traction vectors” along the wheels’ rolling direction) and the (red) force and moment on the platform the wheels are attached to. (The green duo of forces represent an alternative way to specify desired/actual inputs to the robot platform.) This relationship model is composable because the combined effect of all actuator forces are physically realised by simple vector addition.

- **gyration**: transformation of chemical, electrical, hydraulic, . . . energy into *mechanical* energy.
- **transformation**: between the various types of *mechanical* energy, linking torque, acceleration, velocity, position and trajectory.

7.7 Mechanism: asynchronous distributed control

The Sections above made the **assumption** of **synchronous control**:

- all computations can be done in *zero time*.
- the computations are executed at the *right time*, say in 1kHz loop.
- the *scheduling* of the execution of the computations is the same every time one computes the whole control loop.

This assumption does not hold anymore for many modern machines, like cars or robots, that have multiple **fieldbuses** inside, and many of the computations (e.g., sensor processing) must run on separate processes or even computers. In addition, demands are shifting towards **system-of-systems** applications, in which separate machines come together in temporary systems and must realise tasks together; for example:

- multiple **tugboats** maneuvering a tanker.
- multiple *cranes* moving same load.
- multiple *drones* transporting same load.
- getting cars into and out of a *platoon*.

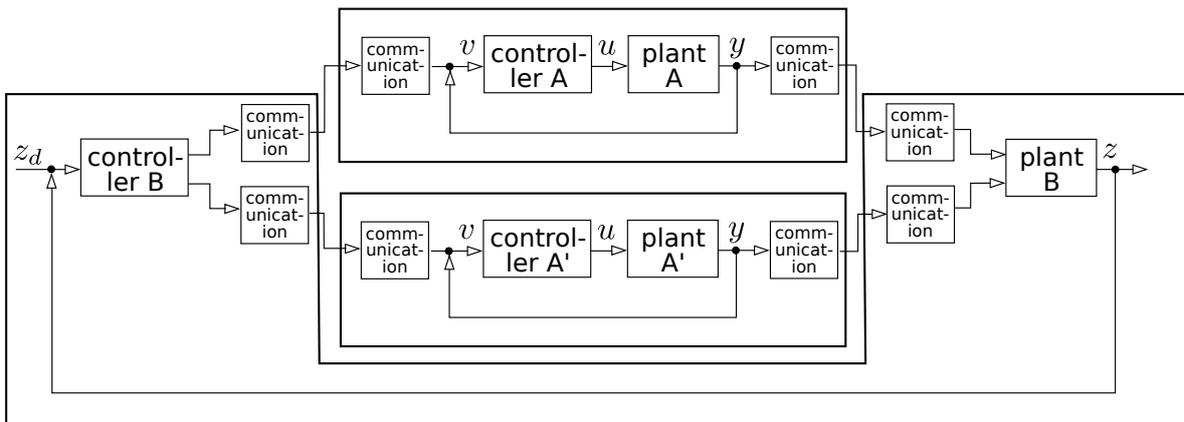


Figure 7.13: Many modern systems must rely on *communication* between sub-systems, in order to realise cascaded control loops.

So, such control loops involve *communication* between control computing processes, Fig. 7.13. Such a **distributed cascaded control** architecture introduces **asynchronicity** into the control problem:

- the closing of *feedback loops* is disturbed, because the latest state information is not available at the theoretically ideal time.
- there is now a need for *monitoring*: each subsystem must monitor how well its own control responsibilities are progressing with respect to what the overall system expects from it. It must provide this information to the other system components, and in turn must use the similar progress quality information that it receives from the other components.

- the result is the need for *mediation*: each subsystem must have a (safe, effective) reaction to the situation where the control progress, of itself or of its peers, is “not good enough”.

The result is that every distributed controller becomes an **hybrid event controller**:

- each sub-controller needs a *Finite State Machine*, with different control configuration in each state.
- all sub-controllers must also send *events* to each other, *to coordinate* their FSMs.

7.8 Mechanism: behaviour tree for semi-optimal control

A **behaviour tree** is a mathematical model, with a limited but very composable number of entities and relations, to decide what next *action* to take in a control loop. (It is a more specific version of a **decision tree**, focused on “control”.) It trades off optimality of the quality of the solution for speed of finding a feasible solution. The knowledge encoded in a behaviour tree model is typically known (i) to be “good enough” in particular use cases, and (ii) reflects experience in how to detect the (or rather, “a”) relevant use case via a series of decision making conditions that are fast to compute.

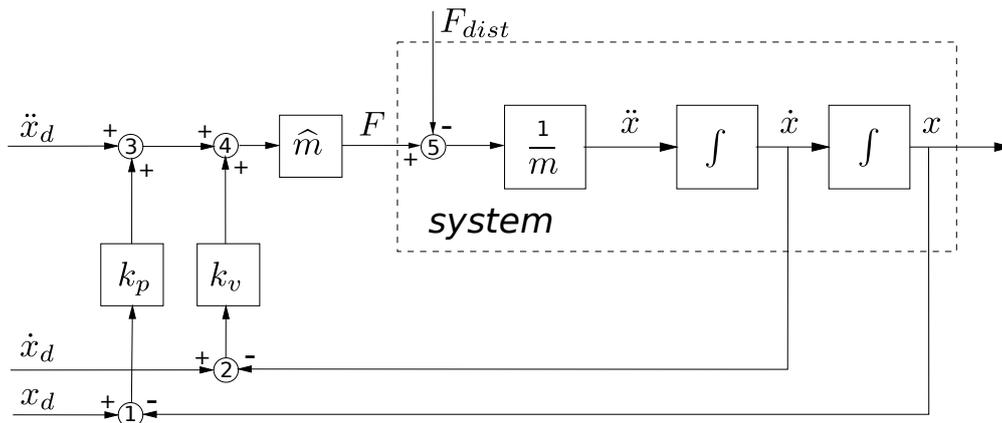


Figure 7.14: A one-dimensional position controller, generating the actuating force F from the desired position x_d , velocity \dot{x}_d and acceleration \ddot{x}_d , via nested velocity and position feedback loops with gains k_v and k_p . The “world model” of the controller consists of an estimate \hat{m} of the moved mass.

7.9 Event loops revisited: control behaviour composition

“Control” is an essential part of all robotics and cyber-physical systems, and the composition of the material introduced by all previous Sections now allows to model the meta model of controllers. In summary, a controller event loop (with a simple example depicted in Fig. 7.14) composes the **task**, **algorithm**, **Finite State Machine**, control diagram, and **event loop** meta models, and adds specific **policies** (i.e., model configurations):

- control diagrams as in Fig. 7.14 represent the **dataflow** model of an algorithm. The **structural model** is a **graph**, but typically unambiguous ways exist to find its **spanning tree** (typically *cutting* the diagram at the “summators” 1, . . . , 5 in Fig. 7.14), with equally unambiguous ways *to serialize* it into a **schedule**.
- dataflow buffers (i.e., the “arrows” in the control diagram) are often just “one deep”, since the controller is only interested in the most recent version of measured data (“*Last Write Wins*”), such that older versions can be overwritten when new measurements arrive. However, modern controllers see an increasing use of *Model-Predictive Control* (MPC) or *Moving-Horizon Estimation* (MHE), which require data flow buffers of size $N > 1$.
- if necessary, pre-processing of measurement data takes place in the **prepare** step for each loop, and gives the result as “new measurement” to the control loop. Such pre-processing can consist of averaging operations, or curve fitting, or other types of **observers** or estimators, like the MPC or MHE approaches mentioned above.
- several nested (or cascaded) loops can exist (e.g., Fig. 7.14): the natural **causality hierarchy** is to schedule the computations of an inner loop more frequently than those of an outer loop.
- many variables computed in control loops must be **monitored**, and these computations must be integrated in the “right way” into the scheduling of all other control loop computations.
- similarly, some monitors do not only generate *events* to trigger *discrete changes* in the control loop configuration, but also *adaptation* of some *continuous parameters* in the controllers, such as feedback gains or model parameters.
- last but not least, **real-time** requirements of the application have an impact on the model of the event loop.

7.9.1 Example: one-dimensional position control

For example, the event loop of the one-dimensional position controller depicted in Fig. 7.14 specializes this generic pattern as described below. The *data blocks* are the arrows in the control diagram:

- *state* of the system $x(t)$, i.e. position of the mass m . The state changes continuously over time, but the controller only needs the most recent versions of the state measurements.
- *setpoint inputs* $x_d, \dot{x}_d, \ddot{x}_d$, e.g. desired position, velocity and acceleration of the mass.
- *measurement inputs* x and \dot{x} , e.g. actual position and velocity of the mass.
- *feedback gains* k_p, k_v , i.e. proportional position/velocity control gains computed, for example, via pole placement (off line) or an observer/adaptor combination (on line).
- *outputs* of control is desired acceleration, reached after summation “4”.
- feedback output is transformed, via *feedforward* multiplication by the *estimated* mass \hat{m} , into force F to system actuators.

- *disturbance* force F_{dist} applies after control, at summation “5”. This is not a summation that is performed in software, because it is realised by nature, in the real world. The measurement actions (that turn real-world values into digital numbers) are not depicted explicitly.
- that real-world *system* is depicted in the Figure within the dashed rectangle. It is *modelled* to be a perfect double integrator with real mass m .

The *function blocks* are the rectangles in the control diagram, and they represent a multiplication; each circle represents a summation function block. The arrows in the diagram represent *data blocks*, but also some of the rectangles have data inside, e.g., the estimated mass rectangle. The high-level *schedule* that realises the controller’s *event loop* (by triggering *function blocks*) is the following:

```
when triggered // = OS executes controller every, say, 10 milliseconds
do {
  communicate() // read desired position/velocity/acceleration
                // from input data block(s)
                // read actual position/velocity from sensors
  schedule()    // trigger function blocks, in the following order:
                // sums 1 & 2, multiplications k_p & k_v,
                // sums 3 & 4, multiplication \hat{m}
  communicate() // write computed control force to actuator data block
}

```

It is possible that the computation of the control loop generates *events* itself. Or rather, such events are generated in *monitor* functions that are not shown explicitly in the Figure, and that the `schedule()` function adds to some data blocks in the controller. For example:

- when an *error* between desired and actual state parameters is too large.
- when the *trend* of the error is undesired, e.g., always positive.
- when the computed control force F is too large for the actuators.
- when the actual execution sample time deviates too much from the desired one.

It is possible that the computation of the control loop must react to *events* that come from the outside (and that are different from the *timer events* that most control loops rely on). For example:

- a new motion plan is started, so that some control parameters must be reset, such as the setpoints and the gains.
- the current plan is interrupted, so that the controller must bring the system to a safe stop as quickly as possible. In practice, this boils down to the controller starting a new motion plan itself.

Hence, the control loop event queue must be extended with `coordinate()` functions to react to (and/or generate) events, and `configure()` functions to realise the reconfigurations triggered by the coordination execution. The “safe stop” functionality would require the addition of extra functions blocks, hence by a new *schedule*.

Implementations of control loops used to require no asynchronous Communication, but just synchronous reading and writing from data in the memory of the computer; the **Programmable Logic Controller** (PLC) works like this, and while it still is *the* workhorse of the automation industry, all modern versions implement the asynchronous and *hybrid* variants. The key hardware-supported technology here is **memory-mapped IO**. But most modern robotic systems now have one or more **field buses**, such as **CAN**, **EtherCat**, or another **Industrial Ethernet** variant, so some **asynchronous** Communication parts have become necessary in the event loops. Such **software architectures** require at least two asynchronously running activities: the field bus **device driver** takes care of the communication over the network, and writes/reads messages into the data blocks that the controllers use in the their event loops.

Interrupts are another very important source of events in control systems; many modern interface devices can be configured to generate events to which the operating system will react. The application can configure the operating system to schedule a specific **interrupt handler** function as soon as the interrupt arrives. (The above-mentioned communications most often work in such an interrupt-driven way.)

7.9.2 Policy: hybrid event control

No realistic **task** can be realised by just one single control loop, and so-called **hybrid event controllers** are needed:

- the **continuous** control behaviour is realised by feedback control loops, like the one introduced above, or by a **constraint optimization solver**.
- some **discrete** control behaviour is added, often in the form of a **Finite State Machine**, where each state executes a different continuous controller, together with other continuous time and space computations, such as monitors, observers, adapters, etc. Transitions between continuous controller modes are triggered by events, generated by the actually running continuous controller itself, or by external activities.

Obviously, such hybrid controllers fit perfectly in the **event loop** approach, since that structures the computations, communications and configurations of the control loops, the FSMs, the event triggering and processing, with synchronous as well as asynchronous activities.

7.9.3 Policy: throughput and latency

Applications require a variety of controllers, and one of the major design trade-offs is that between optimising the controller's **scheduling** for either of the two following *Quality of Service* measures:

- **throughput**: the **more data is processed**, the better. This is important when the behavioural performance of the controller depends on the amount of information that can be extracted from the raw sensor data.
- **latency**: the **faster functions** are executed, the better. This is important when (i) the natural dynamics of the real-world system under control is “fast”, and/or (ii) the control design method requires “exact” timing of the controller computations since the behavioural performance of the controller depends on it.

In many applications, Tasks have a need for both types of computations, the former typically to update their *world models*, and the latter to realise their *feedback control*. An **often seen adaptation** of the **generic high-level control schedule** first does the feedback control as fast as possible and only then spends the remaining computing cycles to world model updating:

```
when triggered
do {
  communicate()      // read only sensor data needed for control actions
  schedule-feedback() // now do all Tasks' feedback control actions
  communicate()      // write computed control efforts to hardware
                    // read extra sensor data needed for world model updates
  schedule-updates() // now update all Tasks' world models
  coordinate()       // only now process events that could
  configure()        // trigger reconfigurations
  communicate()      // do all remaining non-control communications
}
```

7.9.4 Policy: real-time activities via the “multi-thread” software pattern

Many robotics and cyber-physical systems contain one or more activities whose execution must be **predictable** (“**deterministic**”, “**real-time**”) with respect to the computational resources they have available:

- **time**: the execution must take place within a small tolerance of the ideal instance in time. The two key performance measure are *latency* and *jitter*.
- **memory**: the execution must respect consistency constraints on the data structures that are operated upon by the various dataflows used in the activities. A key performance measure is *mutual exclusion*, or *locking*.
- **interrupts**: **interrupts** can preempt most of the software activities on a computer, so real-time applications must configure the interrupt capabilities appropriately. The common configuration options are: to inhibit (“mask”) some interrupts before a real-time activity is launched, to inhibit interrupts on the set of cores that share cache memories with the real-time core, or to assign only the real-time relevant interrupts to the CPU core on which the real-time activity is running.

The **good practice** solution, Fig. 7.15, splits the event loops of these activities into multiple parts, each in a separate **thread**, and all contained within the same **process**:

- the **mediator** thread is the one that comes with the process that is deployed in the operation system, to create the other threads in the process, and to manage their *Life Cycle State Machines*:
 - *resource creation & deletion*:
 - *resource configuration*:
 - *capability configurations*:
 - *running capabilities*:
 - *pausing capabilities*:

- the **real-time** thread executes (i) all *Computations* that must be executed *immediately*, (ii) all *Communications* that are done via non-blocking memory-mapped I/O, and (iii) the *Coordination* that is triggered by the real-time event loop itself and must be dealt with immediately (e.g., deciding to switch to a fail safe control mode).
- the **workers** are other threads, each with one single responsibility, such as (i) feeding the real-time thread with the dataflow it needs, (ii) getting the real-time dataflow and distribute it to the registered clients higher up in the control stack, and (iii) getting the diagnostic information back, to allow for online or offline analysis of the control performance. (Without loss of generality, the latter can be seen as just a special case of (ii).)

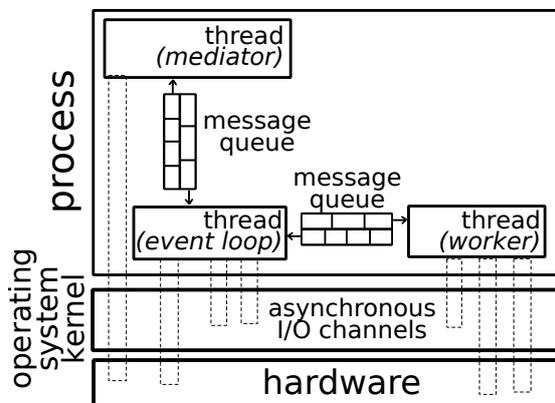


Figure 7.15: Multi-threaded process architecture for the concurrent and parallel execution of activities. The “message queues depicted in the figure represent any type of fast and local inter-thread communication; e.g., lockfree buffers, circular buffers, etc.

The real-time performance of multi-threaded design depends to a large extent on the choice of buffers between both threads. The two common policies are to use either a **locked** buffer approach (by means of a *mutex* or another **locking mechanism**), or a **lockfree** buffer approach.

The first thread is some called the **hard** real-time thread, and the other ones the **soft** real-time thread. These adjectives have no absolute meaning. A major **best practice** design requirement is that there can be **only one hard real-time thread on the whole computer**. And giving that thread the highest priority allowed by the operating system is just a *necessary* but *not sufficient* condition for reaching this requirement. Putting the hard real-time part on a dedicated computer system that runs no other software, is often the only really deterministic design. The state of the technology allows to make dedicated chips (e.g., **FPGAs**) that can even do away with an operating system.

7.9.5 Policy: event loops for task control

The Task meta model is the core structure for the design of cyber-physical systems, so it is necessary to identify and model the impact every specific task model has on the design of the event loops in its activities. More concretely, all property graph “arrows” in Fig. 5.2 must be turned into decisions on how to exchange model data in an activity, synchronously or asynchronously. So, the generic event loop structure of Sec. 2.5 gets specialisations of the generic `communicate()`, `coordinate()`, `configure()` and `compute()` functions, with explicit references to the (interactions between) the task meta model aspects of control, monitoring, plan, world model and perception.

Chapter 8

Meta models for perception and its integration in tasks

Perception is *dual* to control, in many aspects and for all engineering systems, so that both control and perception “stacks” can be seamlessly integrated, at all levels of abstraction. However, in a robotics context the amount of perception opportunities (that is, sensors with sensor data processing activities) is huge. However, the information and software architectures for the integrated control-and-perception stacks are copies of those for the control stacks in themselves.

Previous Chapters focused on the *motion specification and control* aspects of a robotic stack. Motion in a robotics context always requires various forms of *perception*: specifying and controlling motion requires access to information about how “the world looks like” at any given moment, and that requirement can only be achieved if the (task-relevant part of the) world is perceived. Examples of such close integration between motion and perception are visual or force-based tracking of the interaction between a moving robot and its environment. So, it does not make much sense to develop all stacks independently, or to deploy their software implementations in only loosely coupled components: the way how things are perceived by robots, how robots are perceived by other agents, or how robots can/should move, depends to a large extent on how the world around the robots looks like, and on what information of that world can be provided by the sensor-based perception; similarly, motion is in many cases important to help perception, especially to improve *observability* of the world model updating process; finally, the task capabilities that a system offers, help to focus the perception to those sensor-processing efforts that are relevant to make progress in the task execution.

The term “stack” refers to the *hierarchical information/model structure* of all entities and relationships involved in perception. The first, **mereo-topological**, step in that direction is sketched in Fig. 8.1. This Chapter first explains that mereo-topological model in more detail, and then adds the meta models with structure and behaviour at the geometrical, dynamical, information theoretical levels of abstraction, using, amongst others, **Bayesian information theory** as a scientific foundation of this meta modelling. The connections are made for the **task-centric** integration between motion, perception and world modelling, allowing to model explicitly how task specifications can add artificial constraints on the perception behaviour.

8.1 Information associations in perception

This document introduces the association relations sketched in Figure 8.1, to structure the influence of knowledge representation to a task’s perception and control activities, at various *levels of abstraction*. Indeed, the association relations come from *knowledge* about what are the “best” *magic numbers* to use in the activities’ algorithms.

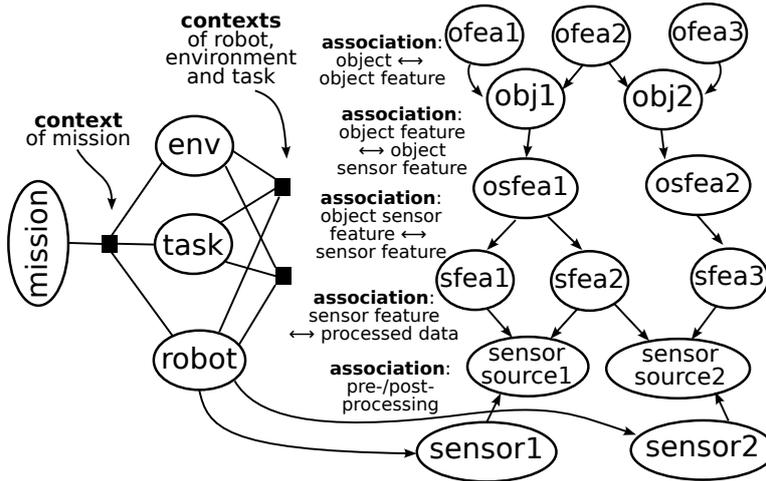


Figure 8.1: A structure to underly the complex dependencies in the *associations* between various levels of perception in robotic systems. This is the perception complement of the control association hierarchy of Fig. 7.7

8.1.1 Pre-processing

Examples of pre-processing operations on raw sensor data:¹

- low-pass filter,...
- conversions RGB to HSL/HSV or grayscale,...
- image pyramid.
- Discrete Cosine Transform, chirplet transform,...
- ...

For example, when knowing the time of day, one can choose better values for brightness thresholds. Or if one has an idea about the passive dynamics of a physical system, one can select better filter bandwidths in the pre-processing.

8.1.2 Pre-processed sensor data association to sensor feature

Examples of sensor features:

- PWM settings for an electrical actuator drive.
- dark-light transition expected in an image.
- corner detection, SIFT or SURF visual features,...
- entropy/texture quantification.

Again, the “higher-order” knowledge behind the association relations are of the following type:

- knowledge about the visual texture differences that can be expected between a sensor feature and its surroundings.

¹Or, in the other “direction” on the actuator setpoints coming out of a control algorithm.

- knowledge about the maximum current allowed in an actuator, or the best resolution that a sensor can provide.

8.1.3 Sensor feature association to object sensor features

Different parts of an object have different “features” for each particular sensor (and/or sensor processing algorithm). Examples are:

- surface **friction** model.
- object edge as separator between two surfaces with different **texture** and/or different **illumination conditions**.
- object region segmentation.
- object corner/surface detection.

Examples of the “higher-order” knowledge behind the association relations are:

- knowledge about expected visual differences in line thicknesses of **visual markers** on a road.
- knowledge about the materials of the floor, the walls, and doors in an indoor corridor.

8.1.4 Object sensor feature association to object feature

Examples of the association relations linking different *functional* parts of an object with different *object sensor feature* parts

- the handle is painted green while the door is painted blue.
- the border of the table is where the contact force of a gripper sliding over the table **drops suddenly**.

Obviously, the above-mentioned examples of knowledge come with specific *quantitative values* of the mentioned properties or effects.

8.1.5 Object feature association to object association

A particular object is recognized is a “sufficient” number of its object features have been detected, in the expected relative order. For example:

- the pattern of doors and windows in an indoor corridor.
- the location of holes and edges in an object that is to be **assembled** in a **manufacturing** cell.

Examples of the

hyperref[sec:magic-number]magic numbers in these association relations are: knowledge about relative sizes, shapes, or scales of the objects’ surfaces.

8.1.6 Association to task, environment and robot context

The objects are part of a particular task, that a particular robot has to execute in a particular environment. For example, delivering logistic goods in an indoor environment, or assembling an aircraft’s wing.

8.1.7 Association of a mission with its tasks, environments and robots

The overall mission of a system determines which of the above-mentioned association relations are relevant or important. Examples of where this “intentional context” matters is the decision

making of whether or not to continue an ongoing task execution, or to deploy more or less robots in it.

8.2 Mereo-topological meta model: the natural hierarchy in robotic perception

The perception stack model has *structural* parts and *behavioural* parts. The structural part uses *property graphs* to model the fact that n-ary relations exist between entities in the stack. These structural relations conform to the [Block-Port-Connector](#) meta model. The behavioural models describe the dependencies between the values of the properties in the connected entities, and these dependencies can be continuous, discrete, or hybrid. For robotic systems, every perception model has n-ary *relations* between *entities* of the following types:

- **sensor**: to describe the properties of the **data** generated by sensor devices.
- **actuator**: to describe the properties of the **data** to be provided to actuator devices to make them put energy into the system.
- **features**: relations between *sensor* data and *object* properties that play a role in the context of a *task*, or between *object* properties and their role in a *task* to determine how the *robot* should move, or both of the above within one single relation. “*Task*” can be replaced by a composition of entities in the models of the *task*, the *robot* and the *environment* in which the previous two operate.
- **objects**: have properties that can be linked to data features, for sensing as well as actuation, and to the tasks that describe what robots have to do with them.
- a **robot** model represents the sensing and actuation capabilities and resources of robotic devices and systems.
- **environment** entities, are often relevant for parametrizing perception algorithms (e.g. camera parameters due to lighting conditions) or to select appropriate sets of sensors (e.g. during fog or rain outdoors or when encountering a dark indoor area during night or in the basement). Obviously, this part of the perception stack contains the links to world modelling; an important development within the project will be the “right” separation and composition of perception modelling and world modelling.
- the **task** plays a crucial role in constraining the selection of all other entities ranging from limiting object types that are relevant during that task to the selection of the perception features that need to be detected.
- the **mission** model makes choices of which task, robot and environment models must be used together to realise “long-living” applications.

8.3 Policy: (data) association

Association relations represent the inherent uncertainty of the inference process that must decide which (sets of) “features” at a lower level of the perception hierarchy are “caused” by which (sets of) “properties” at a higher level. Such association relations appear (at least!)

in four different complementary ways, as indicated in Fig. 8.1: between sensors and features, between features and objects, between objects and task/robot/environment, and between a mission and the tasks, robots and environments it requires. Even a small number of possible choices in every association relation results in a huge number of uncertainties in the overall system model; this complexity is most often very much underestimated by the human mind.

The term **data association** is most often reserved for the association between the sensor data and feature properties.

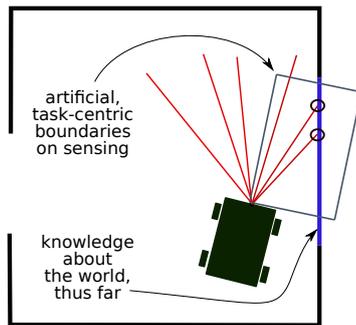


Figure 8.2: Application example of the *escape from the room* task (Sec. 5.11.4) to illustrate various levels in the perception stack hierarchy of Fig. 8.1.

8.4 Perception example: robots driving in traffic

This section provides “running examples” for this Chapter’s modelling an application conforming to the perception stack meta model.

8.4.1 Escape from a room

Assume the robot has a *laser scanner*, *encoders* on the wheels, and a *cameras* (one looking down to the floor, to use its texture for self-localisation; one looking to the ceiling, for similar purposes; and one looking forward). Part of the sensor models describes the physical units, the mathematical, numerical and digital representations of the data that the sensors produce. For the camera this is a matrix with dimensions defined by the sensors resolution property; each of the values in this matrix is a vector containing the RGB values, which are in turn chosen to be represented as integers between 0 and 255. The laser scanner has a similar representation, but with the 2D RGB image replaced by a 1D vector of depth values.

The digital representation of the camera image is used by one or more *segmentation* algorithms. For example, one based on color is configured with the size and color properties of the expected features in the room; assume that there is a wall with a round green drawing. While the system architect would only choose the type of algorithm, the system builder needs to choose a specific implementation here (e.g. in which color space to look for “green segments”). Also the grounding what “green” means in terms of regions in a color space needs to be grounded in a digital representation (potentially by linking to an ontology describing colors in various spaces). In addition, also the *environment conditions* play a role, because the perceived color depends not only on the object properties but also the lighting conditions. The output is a set of green regions, which some algorithms might use as prior knowledge for the next iteration.

To simplify the data association problem, it is assumed that only the circle with the highest probability will be used. This circle has a state which is represented as its centroid

and diameter. By only looking at the numbers shown in Fig. 8.2, it is difficult to say that these numbers are in image or pixel coordinates. Therefore, it is again important to point at the meta model describing the digital representation and the semantics of the data.

This centroid and diameter found in the camera image are then used as an input to a **Kalman Filter** (some additional pre-processing is not displayed). A Kalman Filter is a generic, “platform”, algorithm that needs to be configured with a process and a measurement model, initial conditions, as well as noise parameters. These are configured from the sensor model, task model, and object model. Please note that, in contrast to the perception stack, the task is not explicitly shown in this figure since it is influencing the overall architecture and choices. A Kalman Filter requires a state to work on, which is a (dynamically changing) property of the ball. Again, its digital representation is important as is the semantical context like the frame its position is expressed in (see motion stack).

The green round feature typically has many properties that can also change with every new application. Therefore, the suggested structure allows them to be composed with the “ball” while keeping their semantic context by pointing to the models they conform to. The number of possible object properties is huge and will have to grow over time.

8.4.2 Ego-motion estimation with accelerometer, gyro and encoder

(TODO: link the proper time derivatives of the trajectory of the plan with the corresponding levels in the sensors: linear acceleration in the accelerometer, angular velocity in the gyroscope, and wheel position in the encoder. Then do a *least-squares* parameter identification for each of the sensors separately, or by weighing them all together with the uncertainty magnitude of each individual sensor source.)

8.4.3 Ego-motion estimation with visual point and region features

(TODO: point features are abundant in vision, at the detriment of regional features, that are often more difficult to compute, more dependent on the application, and on other features. Typical regional features are: entropy, geometric or texture patterns, and spatial and temporal frequencies, often on the raw pixels but also on pre-processed pixel values.)

8.5 Probability: composition of data and uncertainty

Using formalized models to represent one’s knowledge about the state of the world, and about the relations that exist between the time evolutions of interacting entities in the world, can help in formulating the “right” task control problem. However, defining the model is only half of the story: because every model contains **parameters**, one has to estimate the “real” value of these parameters, based on (i) the measurement data from sensors, and (ii) the relations that link these measurement data to the model parameters.

8.5.1 Mechanism: Bayesian probability axioms

Bayesian probability theory is a scientific paradigm for **information processing**. Its **axiomatic** (hence, fully **declarative**) foundations have been laid in the 1960s [48, 56]. These *axioms for plausible Bayesian inference* are:

I Degrees of plausibility are represented by real numbers.

II Qualitative correspondence with common sense.

III If a conclusion can be reasoned out in more than one way, then every possible way must lead to the same result.

IV Always take into account all of the evidence one has.

V Always represent equivalent states of knowledge by equivalent plausibility assignments.

They **result** in the well-known mathematics of statistics, with **random variables** and **probability density functions** (PDF) as major entities, and the **chain rule** and **Bayes' rule** as major relations. How to measure the information contents in a PDF was also explained axiomatically [48], resulting in the primary role of **logarithms** as the natural **measures of information**. These axiomatic foundations are as follows:

I $I(M:E \text{ AND } F|C) = f\{I(M:E|C), I(M:F|E \text{ AND } C)\}$

II $I(M:E \text{ AND } M|C) = I(M|C)$

III $I(M:E|C)$ is a strictly increasing function of its arguments

IV $I(M_1 \text{ AND } M_2:M_1|C) = I(M_1:M_1|C)$ if M_1 and M_2 are mutually irrelevant pieces of information.

V $I(M_1 \text{ AND } M_2|M_1 \text{ AND } C) = I(M_2|C)$

A **model** is a set of relations between entities in a domain, and the model for **information** (or **uncertainty**) is a *Probability Density Function* (PDF) $p(X, Y, \dots)$ over the parameter space of a selection of the properties in the model:

- *discrete* PDF: parameter space has only *finite* number of possibilities.
- *continuous* PDF: parameter space is continuous.
- *hybrid* PDF: parameter space combines discrete and continuous sub-spaces.

Information representation is *subjective*, because a PDF is a multi-dimensional, single-valued function $p(X, Y, Z, \dots)$ that describes the probabilistic relationship between the variables X, Y, Z, \dots **in a model** M :

$$p(X, Y, Z, \dots | M)$$

and the model M is a *chosen* representation of the *chosen* relationships (assumptions, constraints, ...) between the variables X, Y, Z, \dots . So, the PDF represents what the *system* “*knows*” about the world, *not* what the world really *is*. *Engineers* have to *choose* what mathematical representations to use, for domain as well as for information!

Information structure = graph:

- **node** contains **variables**, with representation of their uncertainty.
- **arc** (edge, link, arrow, ...) contains **(probabilistic) relationship** between variables in connected nodes.
- terminology: **Bayesian network**, belief network, **factor graph**.

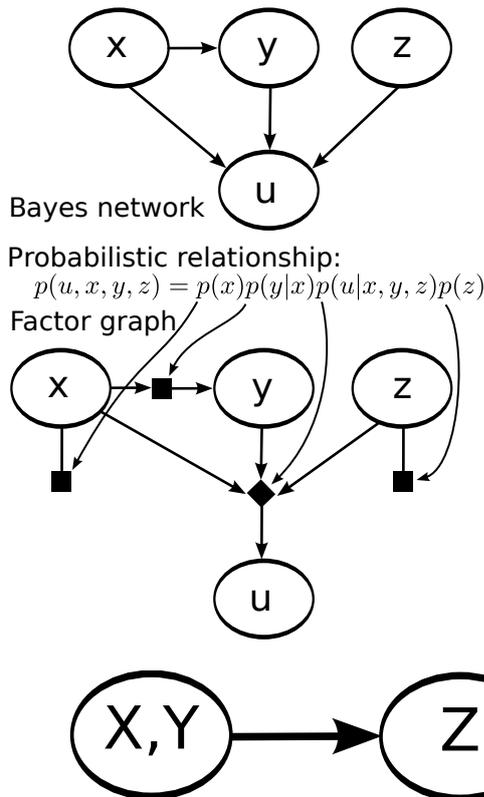


Figure 8.3: Example of a probabilistic model, in Bayesian network form (top), as a factored conditional probability density function (middle), and as a factor graph (bottom).

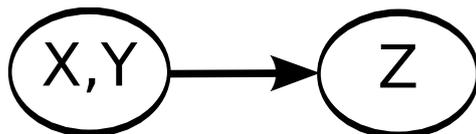


Figure 8.4: Simplest Bayesian network.

- same *real-world* system can have various graphical *models*.

The most important arcs are the ones that *are not there!*

The simplest Bayesian network is one with just one directed arc, Fig. 8.4. The *explicit* relationship $Z = f(X, Y|\Theta)$:

- “if I know something about X and Y , so what can I now then predict about Z ?”
- arrow direction: “easy” to *calculate*
- is not necessarily *physical causality*
- *factorizes* joint PDF via *conditional PDFs*:

$$p(x, y, z) = p(z|x, y)p(x, y).$$

Mathematical representation of a PDF: a single-valued, positive function $p(x)$ + density “ dx ” around the value x . What really counts is the “**probability mass**” (“**expected value**”) over a certain domain D :

$$D = \int_D p(x) dx$$

Extra “property” of PDF: **integral** over complete configuration space of $x = 1$:

- value “1” is **arbitrary choice/convention!**
- only **relative** value of probability mass is important.

Measure of (change in) information:

- **mutual information, relative entropy** of *two* PDFs P and Q :

$$H(P||Q) = \int_X \log \frac{dP}{dQ} dP.$$

- “how much does information change when new data becomes available?”
- “no information” does not exist \rightarrow always *relative*!

Figure 8.5: Simplest PDF: Gaussian (or **normal**) PDF, with **mean** $\mu = 0$ and **variance** $\sigma = 5, 10, 20, 30$.

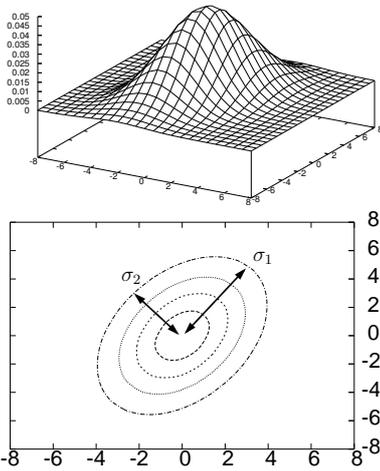


Figure 8.6: 2D Gaussian.

Mean μ , Covariance P :

$$\mu = \int \mathbf{x} p(\mathbf{x}) d\mathbf{x}, \quad P = \int (\mathbf{x} - \mu)(\mathbf{x} - \mu)^T p(\mathbf{x}) d\mathbf{x}$$

(μ is vector)
(P is matrix)

Advantages of Gaussian PDF representations:

- only two parameters needed (per dimension of the domain).
- information processing is (often) analytically possible.

Disadvantages:

- mono-modal = uni-variate = only one “peak”.
- extends until infinity = never zero.

Efficient extensions:

- sum of n Gaussians: can have up to n peaks.
- exponential PDFs: $\alpha h(x) \exp\{\beta g(x)\}$: analytically tractable.

Sample-based PDF is an **approximated PDF** by means of samples with a weight:

Operations on PDFs (e.g., Bayes' rule) reduces to operations on samples. For example, “integral” becomes “sum”:

$$\int \phi(x)p(x)dx \approx \frac{1}{N} \sum_{i=1}^N \phi(x^i) = \sum_{i=1}^N w^i \phi(x^i)$$

8.5.2 Mechanism: Bayes' rule for optimal transformation of data into information

(TODO: [110].)

8.5.3 Policy: belief propagation

(TODO: explain how the structure of a Bayesian graphical model provides a declarative way to solve the model. Junction tree, message passing. [59, 75])

8.5.4 Policy: hypothesis tree for semi-optimal information processing

(TODO: [23, 83].)

8.5.5 Policy: mutual entropy to measure change in information

One of the major **choices** within the large family of logarithmic functions was the following:

$$H(p, q) = - \int p(x) \ln \left(\frac{p(x)}{q(x)} \right) dx,$$

where both $p(x)$ and $q(x)$ must be *strictly positive*. The function $H(p, q)$ is **asymmetric**, hence it is not a distance function, or **metric**. The reason why it is a “major” choice is that it focuses on the **relative** change in information between two probability density functions, instead of aiming for an absolute measure, which does not make much sense. $H(p, q)$ is known under several names: *mutual entropy*, *mutual information*, or **Kullback-Leiber divergence**.

8.6 Geometrical semantics in perception

8.7 Dynamical semantics in perception

8.8 Policy: tracking, localisation, map building

1. **tracking**: how does an identified object's position in the world change over time?

2. **localisation**: where is the robot in the world?

Recognition is perception of the same type as localisation, but intended to know where particular objects are in the robot's environment.

3. **map building**: what is the map of the world?

The modelling (and hence also computational) complexity increases roughly with an order of magnitude with every category of perception.

8.9 Mechanism of information update: Bayes' rule

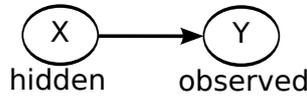
The essential role of Bayes' rule: “*Inverse probability*”:

$$p(x \text{ and } y|H) = p(y \text{ and } x|H)$$

$$(\text{product rule}) \Downarrow (\text{product rule})$$

$$p(x|y, H)p(y|H) = p(y|x, H)p(x|H)$$

$$\Rightarrow \boxed{p(x|y, H) = \frac{p(y|x, H)}{p(y|H)}p(x|H)}$$



Bayes' rule, for the inclusion of new data:

$$\boxed{p(\text{Model params}|\text{Data}, H) = \frac{p(\text{Data}|\text{Model params}, H)}{p(\text{Data}|H)} p(\text{Model params}|H)}$$

$$\text{“Posterior} = \underbrace{\frac{\text{Conditional data likelihood}}{\text{Data Likelihood}}}_{\text{“Likelihood”}} \times \text{Prior.”}$$

Data: *observed*; Model parameters: *hidden*

All factors are functions of *model parameters*, except $p(\text{Data}|H)$ = often just “*normalization factor*.”

Bayes' rule: important properties:

- $p(M|D, H)$: **function** of M , D , and H .
- PDF on Model parameters “in” \Rightarrow PDF on Model parameters “out.”
- Integration of information is *multiplicative*.
- Computationally intensive for general PDFs.
- Easy for discrete PDFs and Gaussians. (And some other families of continuous PDFs.)
- $p(\text{Data}|\text{Model params})$: requires known table or mathematical function $\text{Data} = f(\text{Model params})$ to predict Data from Model.
- Likelihood is *not* a PDF.
- **Optimal Information Processing and Bayes's Theorem**, Arnold Zellner, *The American Statistician*, 42(4):278–280, 1988.

8.10 Mechanism of perception solver: message passing over junction trees

The *message passing algorithm in factor graphs* [13] plays a similar role in perception as the *hybrid dynamics solver* of Sec. 4.8 does for motion. For example, Kalman or Particle Filters, Bayesian networks or Factor Graphs, ARMAX or Butterworth filters, are algorithms with very similar structural properties as the hybrid dynamics solver (Sec. 4.8), as far as they pertain to the “sweeps” over tree structures, and their generation by reasoning about the structural relations (graph interconnections) and the functional constraints (“dynamic programming” solvers of constrained optimization algorithms).

(TODO: much more details and examples.)

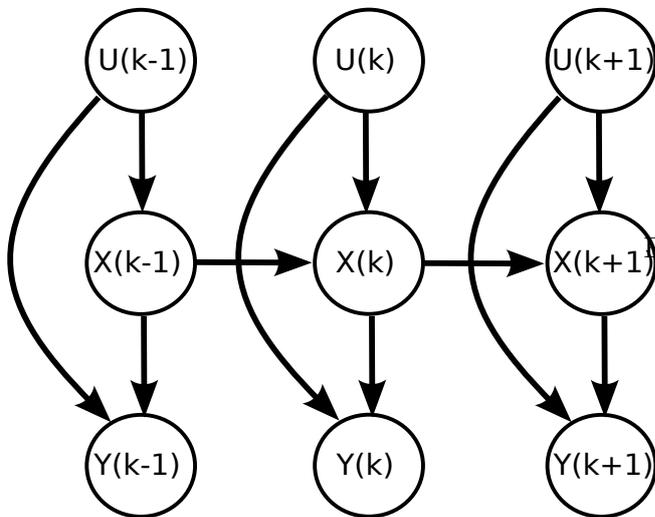


Figure 8.7: Dynamic Bayesian network.

8.11 Policy: dynamic Bayesian network

Figure 8.7 sketches a typical dynamic Bayesian network, where one of the arrows represents *evolution over time*.

Variables:

- U : control inputs
- X : state information
- Y : measurements

Arrows:

- motion model: $X(k+1) = f(X(k), U(k+1))$
- measurement model: $Y(k) = g(X(k), U(k))$

Multiple arrows can be represented by one function.

“1st-order Markov” = “time”-influence only *one step* deep.

A dynamic Bayesian network is the probabilistic extension of the representation of a physical control system:

$$\begin{cases} \frac{dx}{dt} = f(x, \theta, u) \\ y = g(x, \theta, u) \end{cases}$$

- x : domain values.
- t : time.
- θ : model parameters (PDF, relationships).
- u : input values.
- y : output values.
- f : state function, or “*process model*”
- g : output function, or “*measurement model*”

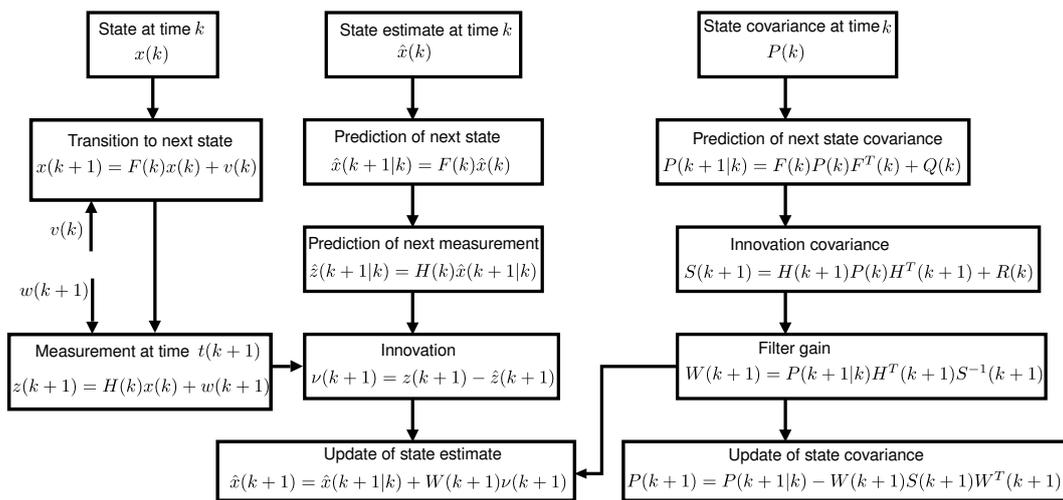


Figure 8.8: Computational schema of the Kalman Filter.

The simplest dynamic network is the *Kalman Filter*.

Required inference: given Y and U , update X .

Assumptions: fast analytical solution possible!

- Process model: $x_{k+1} = F x_k + Q_k$.
- Gaussian “uncertainty” on x_k : covariance P_k .
- Gaussian “process noise”: covariance Q_k .
- Measurement model: $z_k = H x_k + R_k$.
- Gaussian “measurement uncertainty” on z_k : covariance R_k .

Typical application: *tracking* = adapting to *small* deviations from previous values.

Second simplest dynamic network: Particle Filter for localisation.

- functional relationships $f(\cdot), g(\cdot)$: can be *non-linear*.
- PDF *representation*: samples.
- *each sample* is sent through f and g separately, and then a new PDF is reconstructed.

So, a *numerical* solution is needed. **Typical application:** *localisation* with *large* uncertainties.

8.12 Policy: Factor Graphs

8.13 Mechanism: composition of point and region features

8.14 Policy: feature pre-processing

8.15 Policy: deployment in event loop

Chapter 9

Meta meta models for holonic and explainable system architectures

Previous Chapters introduced many types of **components**: functions with their data arguments, algorithms, activities, subsystems, streams, coordination mechanisms, etc. Their focus was on:

- the design insights that support **composability**: by **decoupling** the aspects of computation, communication, coordination and configuration of a component, that component's behaviour is not only **more predictable**, but also **more flexible** to compose with other components.
- the definition of a **data sheet**, that is, a formal model of a components structure and behaviour that represents everything other components have to know to interact with it.
- the application of these insights in relevant **domains**: kinematics and dynamics, perception, control, etc.

This Chapter focuses on the **composition** of existing components, into a new **component-based** system, providing hooks for **system-level trade-offs** in how **to configure** the components in such a way that the composite system realises the **compositionality** property. The underlying design hypothesis is that the more *composable* the components in a system are, the more *compositional*, that is, *predictable*, the behaviour of the system becomes.

The key **meta meta**¹ level *building blocks* of compositional systems are:

- the **Block-Port-Connector** for *connectivity*, **hierarchical** and **heterarchical**.
- the **mediator** for **hierarchy** in *decision making*.
- the **DOM** (Document Object Model) for **hierarchy** in interaction.
- the **stream** for **heterarchy** in interaction.

The key **meta meta** design driver is to introduce as much *hierarchical* structures into a system architecture as possible, because hierarchy is a major approach to deal with **complexity**.

¹That is, the concepts and relations in this Chapter are the ones which one uses whenever one wants to compare different architectures with each other, or to assess whether they can be composed together into a bigger architecture.

The **holon**² meta model composes all of the above, with as key **meta meta** design driver the need to be able to assign **ownership** of resources to one and only one holon at the same time. The underlying design hypothesis is that clear ownership allows to create sub-systems that can behave as full-blown systems themselves, always ready to form larger systems with other holons, all by themselves. The holon structure also provides an *heterarchical context* in which “loosely coupled” components interact via *streams* to *make decisions*

The DOM and holon meta models serve complementary architectural purposes, and their composition is symmetric: a holon can have an internal DOM architecture, and a component in a DOM model can be a holon.

Making all of the above **digital resources** with their own **symbolic metadata**, is key to support their **runtime** creation, configuration, discovery and introspection, to achieve performance without compromising (re)composability. The **architectural mechanism** here are **symbolic representations** for **indirection** and **dependencies**. This document advocates symbolic indirection, for all three core aspects of programming:³ *data, functions, and control flow*.

This Chapter only introduces the *meta meta level* concepts about **architectures**, and one still has to add (i) concrete entities and relations to make the meta models in a particular domain, and (ii) concrete constraints before its meaning can be transformed between two “implementations” of the same meta model. So, designers of cyber-physical systems can create this concreteness via digital⁴ platforms for their customers, that consists of a repeated application of the meta meta level patterns, inside:

- **information** architectures with *activities* to process the *knowledge* and *data* of the customers’ applications.
- hardware architectures with *devices* and *communications* between them.
software architectures of **components** large and small (that is, algorithms, activities, streams and sub-systems), with their (often *dynamically changing*) **peer-to-peer** interactions.

Below are the **system-level properties of architectures** that are relevant at the *meta meta level*. A **resilient** system has an architecture of so-called **holons**, that is, components that:

- **remain operational** under *any* (lack of) interaction with peer systems (that can come and go dynamically).
- can always **decide for themselves** when and how to switch to what kind of **graceful degradation** behavioural state(s).
- can **explain** their decisions, to whatever **peer** holon that is interested in knowing,
- can build up **trust** in their mutual interactions with *identified* holon peers.

The **sustainability** of a system, however, depends on more than just the above-mentioned technical aspects. Some very important factors are:

- the **eco-system interactions** (social, economic, ethical) of the **stakeholders** (humans, organisations, and markets) in an **application domain**.

²On mainstream, over-simplified incarnations, a holon structure is often also called *agent*, or *digital twin*. The simplification often comes from neglecting the hierarchical and ownership aspects.

³**Templates** a popular approach is many programming languages to achieve the same goals, but they cover only data and functions, not control flow.

⁴Or **business**, or **computational**.

- the **safety** that the system can offer, to itself, its users, and its environment.
- the **authentication & authorisation protocols**.
- (hence) the penetration and acceptance of **open standards**, first, and of **free and open-source software** (FOSS), second.

9.1 Hierarchy in knowledge versus hierarchy in architecture

Knowledge representations are powerful because they contain many **graph-structured associations** between entities and relations of the domain that is modelled. Many of these knowledge relations have a **tree-structured skeleton**, representing the “*key*” associations in that domain, in a hierarchical way (that is, with *parent-children* connections). An example of such a tree skeleton is the representation of the **built environment**, where the **hierarchy of containment** shows up very clearly: the root-level (or highest ancestor) “container” is the *world*, with *continents* and *countries* as relevant branches (or descendants). The tree branches further via containment relations of *country*, *province*, *city*, *street*, *building*, and *floor*. The floor is a natural end point (“*leaf*”) of the tree structure, because it is a graph-structured interconnection of *corridors*, *rooms*, and *doors*.

System architectures, however, have a complementary purpose: knowledge relations are *static* pieces of information, but eventually, it will be *software* that realises the system’s *behaviour*. More importantly, *decisions are made*, based on the actual *state* of the system. Of course, these state variables are linked to knowledge relations, but the latter’s graph structure is seldom the best way to structure the *dependencies* between *software components*. The latter must be given a context of execution, in which one should try to introduce hierarchy to allow efficient *decision making* about the software component behaviour. That hierarchy must (i) reflect (parts of) the knowledge, faithfully, and (ii) influence the system’s behaviour in a deterministic way.

This document adopts the following meta model of hierarchical dependencies between software components:

- the change of (some) properties at a higher level component in the hierarchy *trickles down* to components at lower levels in the tree.
For example, if a building is sold, the ownership of everything inside that building changes. If one uses the same template model for different houses, parameters like number of floors and heights of ceilings should only be changed at the top level. Or if a corridor is given fire or security doors, the usage of all rooms inside is impacted.
- a behavioural “event” in a lower level component *propagates up* towards higher level components, but also to *siblings* in the tree.
For example, if the door of a room is locked, which other parts of the building should be informed. At what level will the city’s fire brigade be called when a fire has started in a particular room. Or where is the decision taken to let a robot change its planned navigation task through a building when it detects “obstacles” on its way.

9.2 Data sheets: digital resources (holons, twins, platforms) with metadata

This document strives for software components that are aware that they have a model and a set of meta models, and that can use them in “dialogues” with other components, via one of its ports. Conceptually, the contents of a data sheet is straightforward, namely the enumeration of all models that represent the behaviour of the component, as visible by other components:

- **Semantic_ID**: a (set of) data structures (strings, numbers, binary blobs, . . .) that identify the component uniquely.
- **ports**: the *data structures* available via each port, the *interaction protocols* that the port follows, and the *constraints* that may occur on the behaviour of different ports.
- **resources**: a symbolic model of the resources the component has available to provide its services to others.
- **operational range**: constraints on “how good” the component can realise its behaviour.

9.3 Components: heterarchy in activities and interfaces

Any cyber-physical application consists of a (potentially large) number of **activities**, in *physical* processes (that is, in nature, or in man-made hardware devices such as mechanisms, sensors and computers), and/or in *cyber* processes (that is, in software). The activities **interact** with each other, in “cyber space” via the software and communication *services* they provide to each other, but also in “physical space”, via the actions they perform on physical *resources* that they share.

The *component* is *the* building block of the *system architects*, allowing them to compose activities and their interactions into compositions:

- that can be **deployed** in the system (again, for both the physical and cyber versions),
- without a need to know about the exact workings of these activities and interactions in the **inside** of the component.

Several other terms are used to represent the concept of a “component”: **service, sub-system, holon, agent, process, node, server, . . .**. In addition, this document uses the term “component” as a *pars pro toto* or *synecdoche*, to refer to the functionality *compositions* introduced elsewhere in this document: **functions, algorithms, activities, and interactions**. Of course, the quality of a component in a system depends to a large extent on the **composability** scope with which the components’ interior has been designed.

9.3.1 Components, composability, compositionality

This document advocates a way to design activities and interactions such that the architected system keeps the **compositionality** property whenever it is built from components that have the **composability** property. The objective is that knowing the component architecture, together with the *data sheets* of the behaviour of the activities in, and the interactions between, the components, suffices **to predict** the behaviour of the system. A well-done composition architecture should, ideally, behave itself as a new “primitive” component in a larger system.

This document *designs* the mechanisms of *function, algorithm, activity, interaction* and

component with the *same* composability-compositionality design drivers. When done right, this allows

- to let a system **explain** all of its decisions, at all times.
- **to instrument** the system with monitoring functionalities.
- **to reconfigure** all of the above, even at runtime.

The *differences* that *need* to be introduced between the five mechanisms, cover the variety in **deployment** constraints. That is, the deployment of information architecture building blocks onto software architecture building blocks, and the deployment of software building blocks onto the building blocks provided by computer hardware (CPU cores, networks, memory) and by the operating systems (processes, threads, system calls, inter-process communication, shared memory regions).

9.3.2 Vertical and horizontal composition

Two complementary ways exist in system composition:

- **vertical** (or, “**internal**”) composition into one single **component**, of **activities** at various **levels of abstraction**, and at various levels in the **association hierarchy**.

“Single” means that (i) all the composed activities are always deployed together, and (ii) they are not visible anymore as individual activities to the “outside world”, because only the top-level component interacts with that outside world.

Vertical composition is also possible, with the same design approach, for various *algorithms* in one *activity*, and for various *functions* in one *algorithm*.

- **horizontal** (or, “**external**”) composition of **components** into a (**distributed**) **system**.

“Distributed” means that:

- components communicate over networks (in the broad meaning of the term).
- oftentimes so-called **middleware** components bring extra integration challenges into the system (with as expected added value their provision of tailor-made solutions).

9.3.3 Lifecycle of compositions

Composition of activities into components, or components into systems, need not be static. Especially in robotics contexts, the variability in tasks and environments increasingly require runtime re-composition. This can only be realised if re-configuration is explicitly designed-in as one of the the **lifecycle phases** of a composition. Re-configuration encompasses horizontally and vertically composed behaviour inside a component, as well as the behaviour of a component that is made visible to the component’s outside.

9.3.4 The 5C’s: Composition of 4C behavioural roles in a component

This Section presents the **5C** component meta model,⁵ [81, 100], consisting of the following “*four + one*” **mereological** entities and relations:

⁵The 5C model of this Section has its **acronym** in common with that of [66]; while there is a thematic overlap, the meaning of the latter 5C model corresponds more to the “levels of abstraction” discussed in Sec. 1.4.1 and following.

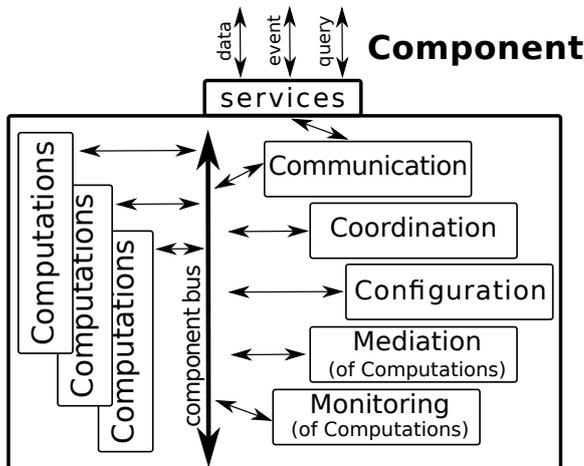


Figure 9.1: The mereo-topological model of a *Component*, structured with the *5C* meta model.

1. **Computation**: the **algorithmic** entities that realise the components “**functional**” behaviour, that is, changes of the component’s “state” in (software representations of) time, space, energy, quality,...
2. **Coordination**: the **logic** entities that realise the component’s “**discrete**” behaviour, that is, they process **events** and decide whether or not the component has to change its **Computation** behaviour.
3. **Configuration**: the **logistic** entities that turn such behaviour-changing decisions into reality, taking into account the **constraints** imposed by the **resources** that the component is using. Indeed, a component can not always change its behaviour from one execution time to the next, because the component relies on specific behaviours of other hardware and software components. Often, configuration requests trickle down to other “resource” components, and there is *asynchronous state* involved in the process of realising a (re)configuration.
4. **Communication**: the **interaction** entities that **exchange** data, events and models between **concurrently or asynchronously executing** components, respecting specific constraints on consistency, ordering, memory usage, timing, etc. A *Component*’s internal exchange of information can be organized as one or more (**software**) **buses**, including **stream buffers** in shared memory.
5. **Composition**: every **architectural** relation that **couples** a collection of the four entities above into one **component**.

The first **4Cs** only have meaning in the context of **Composition**. For example, it only makes sense to separate the **Configuration** aspects of a component or a system, *if and only if* the following conditions are *both* satisfied:

- there is a design driver to compose that components with other components, now or later, or whenever.
- that composition *requires* the computations of both components to be configured *together*.

In other words, it only makes sense to spend efforts in dissecting components into **4C** parts if later compositions of these components introduce **dependencies** between one or more of these parts, between two or more of the components. If a component already comes with **4C** decoupling, it is *potentially more composable* than a monolithic component.

The total **5C** meta model has only **qualitative** value: all it can do is to make developers

aware *that* each (software) component **has-a** specific set of parts with the above-mentioned roles. A value that it does not have, is to explain *how* to realise these roles with concrete software components, neither **declaratively nor imperatively**. The **DOM models** are examples of *declarative* representations of architectures; the **event loop** is an example of an *imperative* representation of an architecture that conforms to the 5C meta model. Both are *models*, and hence *not* a **software architecture**. That means that:

- these models of architectures can be **pre-processed before configuration and deployment**. This allows, for example, to optimize the amount of **Communication**, **Configuration** and/or **Coordination** components that are needed in a (sub)system.
- the architecture of a deployed (sub)system can be **adapted at runtime**, *if* (i) the models of all **Components** are available, and (ii) the **Configuration** component can deal with online *queries* for re-composition.
- the model is an excellent (because structured and explicit) **documentation** for human developers to discuss the system design, and its trade-offs.

9.3.5 Types of Computation

The following *types* of **Computation** are so generic (at least in robotics contexts) that this document includes them in its *cyber-physical domain* extension of the generic component meta model :

- **monitoring**: a computational component with a “**higher-order**” role, in that:
 - it exchanges data with one or more of the computational components that are responsible for the *services* that the *Component* must provide.
 - it computes the **Quality of Service** (QoS) with which these computations are realizing their *expected* service behaviour, and
 - it **fires an event** when particular QoS thresholds are reached.
- **mediation**: a computational component with another “**higher-order**” role complementary to the *Monitors*:
 - it exchanges data with the **Component’s Configuration**.
 - it has the extra **application-specific** knowledge about how various **Computations** are to be **traded-off** when the Quality of Service of the **Component** goes beyond its configured thresholds.
 - it takes the **decision** to trigger **Configuration** of some of the *Components* to react according to the application’s policy. The latter are system-level trade-offs, that the system developers have configured into the Mediation component itself.
- **scheduling**: many algorithms in robotic and cyber-physical systems have high flexibility, because they cover a diversity of computational tasks, large and small, that must be realised “*together*”, constrained by some *inter-dependencies*. For example, the torque control loops of all motors in a machine; various visual feature detectors in different places of a camera image; distributing “work” to various “workers”; ordering the access to shared resources; etc. Such computations are often called **scheduling**.
- **dispatching**: a computation that must:
 - decide which resources (activities, algorithms, components, hardware,...) are responsible for the execution of a schedule.
 - monitor the actual execution.

- fire the necessary events to keep the system updated about the progress of the schedule execution.

There can be multiple **Monitoring** and/or **Mediator** components in each **Component** model, the same **Monitor** and/or **Mediator** can get data from several **Computations**, and the same **Computation** can provide data to several **Monitors** and/or **Mediators**.

9.3.6 Types of state: state, status (flag), and their changes (event)

“State” is one of those over-used terms in **science** and **technology**, and many other terms are used as (almost) synonyms (“mode”, “phase”, “status”, “flag”,...). This document chooses to use three terms (*state*, *status*, *event*), with the following semantics:

- **state of an algorithm**: an algorithm (or computer programme) stores data in variables, and their value can be stored to **persistent storage** at any time, and re-loaded in the computer later to continue the algorithm’s execution. (This assumes all functions in the algorithm are **without side-effects**.)
- **state of an activity**: this state represents the single *behaviour* that an **activity** is executing at a given moment in time.

The state is meant **to be used internally** in an activity’s own finite state machine, (**FSM**), that is, the data structure that let it decide about which **behaviour to realise** now, or next.

- **state of a interaction**: represents the progress in the agreements (**protocols**) that activities have about how to interact with each other, physically or **digitally**.
- **state of a component** (or system): the composition of the states of all algorithms, activities and interactions in the component.
- **status**: represents the **value of a condition** (symbolic, numerical or logical relation) on some parameters in states of the above-mentioned types.

A status is represented by a (**status**) **flag**, which is meant **to be observed synchronously** by other activities, to let algorithms in different activities decide which **functions to execute** now, or next. The **Petri Net** is the meta model of a very common approach of this type of coordination.

- **event**: represents the fact that **something has changed** in a state or a status. That “something” can be the value of a variable (e.g., the distance of a robot to an object), but also the behavioural state of an activity, or the status flag of an algorithm or data structure.

An event is meant **to be communicated asynchronously** towards other activities, to allow their **coordination to react** to what happens elsewhere.

9.3.7 Modes of coordination: coordinated, orchestrated, choreographed

One possible trade-off to make in the design of a system is that between (i) the amount and latency of the communication required to coordinate components, and (ii) the *autonomy* of each component. This trade-off is represented by the following modes of coordination that components can engage in:

- **coordinated**: *all* components react only when they receive an explicit event to do so, and they expect that event to come with all information about how to react.
- **orchestrated**: less events have to be broadcasted, and each carrying less information, because all components share the same “**score**”. That is, a *model* that represents (i) the

expected sequencing of events, and (ii) the expected reaction of each component.

- **choreographed**: the components have a *Plan* of their mutual coordination, and they have *Computation* components that *observe* the behaviour of other components, and *Monitors* that can recognize when the reconfigurations that are scripted in the *Plan* must be executed. Ideally, this can happen without the need to broadcast one single event between the components.

The descriptions above use *events* as the triggers for coordination of *behaviour* of components, but the same three coordination types hold for *flag*-based coordination between multiple *activities* in one or multiple components.

9.3.8 Mechanism and policy in system composition

One of the major pragmatic problems to compose components into systems is that components often come with **hard-coded configuration** choices (Sec. 9.3.4). The reason most often being that the component was developed with just one particular application context in mind, and for which the chosen configuration was (hopefully) “optimal” and/or “obvious”. So, another **mereological** aspect of component modelling to improve **composability** is **to separate** (i) the description of a component’s “mechanism” from (ii) the description of the “policy” with which that mechanism is used in a particular application context:

- **mechanism**: **what** does a component (algorithm, process, agent, piece of functionality,...) **do**, irrespective of the application in which it is used? Often, mechanism is subdivided into its **topological** sub-parts of structure and behaviour:
 - *structure*: how are the parts of the model/software connected together?
 - *behaviour*: what discrete and continuous “state changes” does each part realise?
- **policy**: **how** can the structure and behaviour of the component be **configured**, to adapt its functionality to the particular application it is used in? In its simplest form, a policy just configures some “magic number” parameters in the model/code of a library or component system. In more complicated forms, the whole architecture and interfaces of an application are optimized towards the particular application context.

9.3.9 Policy: vendors add value in Configuration, Coordination, Composition

The *Component* meta model has a handful of different roles, but Fig. 9.1 already hints at the fact that the amount of models and code that will eventually have to be used for all components is, by far, concentrated in the *Computations*. The other components can have very little content, but that content is the one where vendors can make the difference between the *generic* service implementations and their unique selling point and commercial added value.

9.3.10 Bad practices: interpreting attributes as properties

The concepts of “policy” and “attributes” are often encountered together, because the latter are, by definition, *always* the result of a policy decision: the policy *decides* what value to give to the attribute. Here are some all too common examples (hence the name *bad practice*) where an attribute was set by a component designer *as if it were a property* of that component:

- a control gain is *not* a *property* of a *controller*, but an *attribute* given by the *task* that needs the controller to improve a particular *task-dependent* performance metric.
- *colour* is not a property of an *object*, but of the *relation* that connects the material properties (texture, paint, . . .) of that object with the *lighting conditions* and the *visual perception properties* of a camera.
- *the* shape is not a property of a link in a kinematic chain, because various applications will require other shape representations for the same link. For example, to make fast computations with only a first-order accuracy, or to add a specific mesh for collision deformation simulation, etc.
- the identity of the peer components with which a component interacts, is an attribute that is given to the component via some sort of **mediation**. The same goes for the streams via which the component interacts.

9.3.11 Block-Port-Connector for components, activities and functions

Several Sections above introduced the major building blocks for “behaviour”: **functions** and **activities**, composed together into **components**. At each level of composition, a component must be able to expose a selected part of its *inside* behaviour to the *outside* via **ports**, so that that behaviour can be *connected* to the behaviour or other components. At the same time, ports realise **information hiding**: the *inside* behaviour can be hidden, when this is advantageous (i) to reduce system-wide complexity, and (ii) to leave *freedom of choice* to the implementation of the insides. This composition freedom can be done in many different ways, and the *forces* that determine the behaviour of the composition are:

- *encapsulation, information hiding, security*. For example, in the context of **functions**, the representation of these various forms of data “*protection*” is possible by dedicated constraints on the accessibility of the **D-blocks**. This is best done by composition of the algorithm’s **A-block** with a **BPC model** in which some **Ports** are **connected** to selected **D-blocks**, **F-blocks** and/or **S-blocks**, and those **connect** relations get **attributes** that model the kind of “*protection*” to be composed in.

In other words, the *closure* of the algorithm is not defined by the functional developer, at design-time, but by the component supplier, who provides the port-based view on the algorithm that fits best to the concrete context in which its functionality is to be used.

- *run time adaptability*: there is no hard technical constraint that prevents the just-mentioned port-based *views* to be adapted at runtime.
- *resource management*: the execution of an algorithm makes use of two resources of the computer hardware, namely its *memory* and its *CPU*. Because the meta model contains explicit and complete information about the memory requirements (size as well as access constraints) of *D-blocks* and the execution sequences of *F-blocks*, an application developer can provide tooling to deal with possible **exhaustion** of those resources; e.g., the various management policies for **data buffers** or **stacks**, and for **iterators** or **execution schedules**.

9.4 Document Object Model: hierarchy in coordination & configuration

This Section introduces the **DOM**, or **Document Object Model**,⁶ meta model, using a **tree structure** to represent the **hierarchical** set of **dependencies** in the **configuration** and **coordination** aspects of the **5C meta model** of “tightly coupled” components. The DOM concept serves two partially overlapping purposes:

- the DOM structures the decision making about configuration and coordination.
- the structure is a **tree**, so efficiency of implementations is guaranteed.
- the DOM model is part of a component’s **data sheet**.

The DOM meta model comes with the two naturally fitting meta models of *cascading* configuration and *bubbling* event handling.

9.4.1 Good and bad practices for hierarchical models

Hierarchy is good for system design, because the predictability, liability and monitoring of decision making increases when the **context** in which decisions are made is very hierarchical. However, hierarchy is also a too tempting structure to adopt without thorough motivation, so developers must be careful:

- to support the good parts of hierarchy, that is, the structure that it brings in the **information** required for decision making.

For example, if a torque controller component detects **overheating** of one of the actuators in the robot, the hierarchical context model helps to decide which other control levels should be informed about this condition.

- to avoid the bad parts of hierarchy, that is, to **hard code decision making** into a hierarchy of components, whenever such a strict structure does not reflect the requirements of the application.

For example, in the above-mentioned motion control context, one should not assume at a high level of the hierarchy that all actuators have the same overheating limits.

The major cause of why hierarchy is good or bad, lies in the **amount of interactions** that is needed between components

1. to reach a decision, and
2. to adapt the behaviour of every component impacted by the decision.

Human society has several realisations of both good practices for using, or *not* using, hierarchy. For example, the strict hierarchies in the military and multi-national corporations, where generals or CEOs can not interact productively with all soldiers or workers in their organisation. Such constructive interaction *are* possible and constructive in the **heterarchical, flat** structures of **startup companies** or **grassroots social movements**.

9.4.2 Mechanism: elements, tags, attributes, and containment

A DOM *model* is a *hierarchical structure* (Fig. 9.2) of (an explicitly identified subset of all) system components, that represents the **scope** and **structure** of the decision making

⁶It would conform to this document’s semantics better if the “*M*” in “*DOM*” were to stand for “*meta model*” instead of “*model*”, but the term has already achieved widespread acceptance such that a deviating nomenclature, however small, would introduce unnecessary confusion.

command hierarchies between these components. Standardized DOM meta models exist,⁷ through which one can change, at runtime, the hierarchical structure that a conforming model represents. The most widespread use of DOMs is now in Web browsers, graphical user interfaces, and declarative markup languages⁸ like HTML5, SGML or GML.

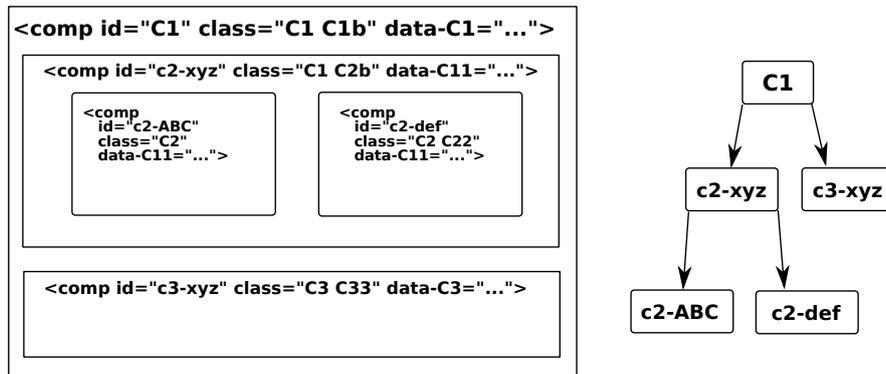


Figure 9.2: Example of tree-structured hierarchy in a Document Object Model. The tree structure provides the *scope* for the *configuration tags* (*class*, *data-...*) in the component descriptors, and for the *event handling* (see Fig. 9.3).

Figure 9.2 shows the mereo-topological entities and relations of a DOM model:

- **tags, elements, and nodes:** an *element* is syntactically scoped by an opening tag `<comp...>` and a closing tag `</comp>`, and it represents a *component* in the DOM model. That component’s contents, between the tags, consists of a tree structure of *nodes*, that represent the *properties* of the component. Properties of a components are fully “owned” by that component: their values depend on that component and only on that component, but not on the position of that component in a system, nor even its position in a DOM hierarchy;
- **attributes:** a starting `<comp...>` tag contains the *attributes* of the component, that is, its meaning, role, purpose,... *attributes* have a type: *id*, *class*, or *data*. A component comes with attributes of itself, and it inherits other attributes from the component higher up in the DOM hierarchy. When several of the component’s ancestors provide the same attributes, the one of the closest ancestor takes precedence.
- **containment relation:** the components are ordered according to a tree hierarchy. This *tree* structure involves a form of the *Block-Port-Connector* meta model, where “connected-by” means “is-contained-in”, or *is-child-of*. The obvious inverse connection relations are “contains” or “is-parent-of”.
- **shadow DOM relation:** the configuration and coordination dependencies between the full DOM model tree and a sub-tree can be *encapsulated* explicitly. This “shadow” barrier is a boundary of the propagation of configuration and coordination through the tree structure.

⁷As well as rich ecosystems around them, providing software tooling and programmatic interfaces for offline and online use.

⁸Because XML is the most common host language for DOM markup languages, the example in this Chapter also adopt an XML syntax. Semantically, the meaning of the elements and attributes can be mapped one-on-one to the JSON-LD host language that was used in previous Chapters.

A common *syntactical shortcut* is the introduction of *custom tags*:

```
<comp id="C1" class="CustomClass C1b" data-C1="...">
```

is shortened to

```
<CustomClass id="C1" class="C1b" data-C1="...">
```

The containment relation is depicted graphically in Fig. 9.2, but it is a *best practice* in text-only representations to use the nesting of matching tags to represent containment:

```
<comp id="C1" class="C1 C1b" data-C1="...">
  <comp id="c2-xyz" class="C1 C2b" data-C11="...">
    <comp id="c2-ABC" class="C2" data-C11="...">
    </comp>
    <comp id="c2-def" class="C2 C22" data-C11="...">
    </comp>
  </comp>
  <comp id="c3-xyz" class="C3 C33" data-C3="...">
  </comp>
</comp>
```

9.4.3 Policy: paths, diffs and multi-tree

Because it is built with only tree structures, implementations that work with DOM models can rely on *fast indexing* of elements and tags, represented by a simple grammar, such as *XPath* or *UNIX file paths*. For example, starting from the top of the hierarchy of the DOM model above:

```
C1/c2-xyz/c2-ABC
```

Or with relative paths, for example starting from *c2-def* to reach its *sibling* *c2-ABC* or its “nephew” *c3-xyz*:

```
../c2-ABC    ../../c3-xyz
```

Such paths also allow for efficiently finding *structural differences* (“*diffs*”) between two DOM model trees.

Finally, one can allow one particular component to appear in several DOM models at the same time, as a means to provide different contexts to that component. The more precise semantics of “to be in” is that several DOM models can *refer to* the same component via that components unique ID. In other words, it is straightforward to extend a DOM model from a single tree to a *multitree*. This policy is:

- **composable**: (i) one model’s *root* component can be added to any other DOM tree as a new *leaf* node, and (ii) such a composition can add new class or data attributes, or overrule existing ones, respecting the standardized cascading constraints.
- **compositional**: dependencies are explicit and strictly hierarchical.
- **robust**: the same model entities (like class attributes) can be defined at different levels of the DOM hierarchy, and this *redundancy* gives system designers opportunities for extra consistency checks during composition of DOM models.

9.4.4 DOM Cascading Style Sheets for top-down configuration

Figure 9.2 shows the tree-structured containment of component representations in a DOM model. The *cascading* semantics of the attributes has the following: *mereo-topological* entities and relations:

- *inheritance* of values of attributes is top-down according to the tree structure. The class attributes are interpreted as Semantic IDs of meta models, so multiple conformance applies to their composition. (TODO: give examples.)
- *selectors* can be added to fine-tune the inheriting behaviour.
- *variables* can be added to configure the same values in different elements. They can be computed at runtime, and stored in local or remote storage.

Both meta models are fully declarative, hence supporting **composability** and **compositionality**.

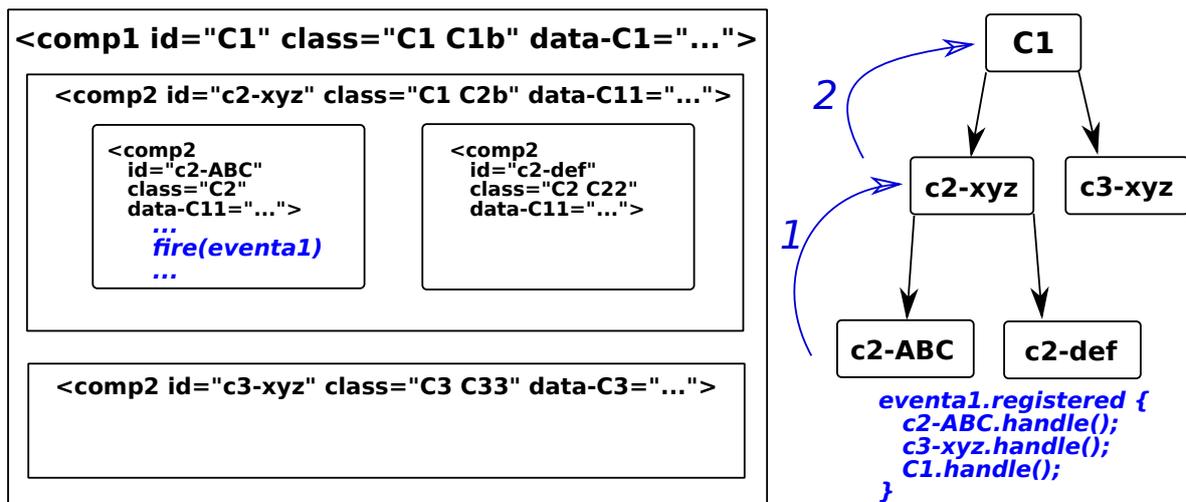


Figure 9.3: Sketch of how event bubbling can work in the DOM of Fig. 9.2

9.4.5 DOM events for top-down and bottom up coordination

Figure 9.3 shows propagation of event handling up and down the containment tree structure of a component DOM model. The mereo-topological entities and relations of events are:

- **event**: has one unique type, and a (possibly ordered) list of targets that have registered to be informed about the occurrences of the event.
- **event_target**: a component that wants to be informed about the occurrences of an event.
- **event_dispatch**: the mechanism that delivers the occurrence of a particular event to each target that is registered to that event.
- **event_listener**: the function that a target executes when the event is dispatched to it.

(TODO: more details and examples.)

9.4.6 DOM model for relation-constraint-tolerance models

The composition pattern of relation, constraint, and tolerance, has a DOM formalization as sketched in this generic example:

```
<relation id="R1" // A relation of type rela
```

```

    class="relA"           // with two arguments
    data-pars="max">      // and one constant parameter
  <property id="propA" class="role-input1"></property>
  <property id="propB" class="role-output1"></property>
  <property id="MaxVal" class="role-parameter" value="max"></property>
</relation>

<relation id="C1"         // A constraint relation of type
  class="constraint1"     // "constraint1" with parameter cM
  data-constr-par="cM">  // on two properties of relation R1:
  <property id="arg1" class="role-subject"> #R1/propA </property>
  <property id="arg2" class="role-object"> #R1/propB </property>
  <property id="limit" class="role-parameter"> cM </property>
</relation>

// A constraint relation of type "cnstrX", with parameter cc,
// on one property of relation R1:
<relation id="C2" class="constraint cnstrX" data-constr-par="cc,dd">
  <property id="arg1" class="role-subject"> #R1/propB </property>
  <property id="limit" class="role-parameter"> cc </property>
</relation>

// A tolerance relation of type tolZ on two properties
// of constraint relations C1 and C2:
<relation id="T1" class="tolerance tolZ" data-tol-par="tol1">
  <property id="arg1" class="role-p1"> #C1/data-constr-par/cM </property>
  <property id="arg2" class="role-p2"> #C2/data-constr-par/cc </property>
  <property id="limit" class="role-tol-limit"> tol1 </property>
</relation>

// A tolerance relation of type tolX on one property
// of constraint relation C1:
<relation id="T2" class="tolerance tolX" data-tol-par="tol2">
  <property id="arg1" class="role-X1"> #C1/data-constr-par/cM </property>
  <property id="limit" class="role-tol-limit"> tol2 </property>
</relation>

```

Not surprisingly, all these expressions share the same DOM structure, which is that of a *relation*. Their semantic differences are expressed by the *attributes*. The **hierarchy** that we want a DOM **model** to represent is *not* in the hierarchy of the textual order of the above-written **DOM document**, but in (i) the **lexical nesting of the elements** in every single expression, and (ii) the **dependencies between parameters** in the different expressions. Indeed, the **constraint** and **tolerance** expressions *refer to* the **relations** they pertain to, using their symbolic *IDs*. So, the hierarchically “highest” expressions are the **tolerances**, although they happen to be written “lowest” in the DOM document above; anyway, they could have been placed anywhere in the DOM document, for that matter.

9.4.7 Graph connections between DOM model trees

By means of the systematic use of **symbolic pointers**, or **Semantic IDs**, constraint and tolerance relations as in the example above can be introduced between entities and properties of relations that reside in more than one DOM tree structure, and more than one DOM

document. The result is that they introduce **graph-structured** couplings between **tree-structured** DOM models. Hence, it is the system designers responsibility to encode the knowledge about the system, its applications and its context, with the “right” composition of:

- DOM models. They bring **efficiency** of the DOM model-based “reasoning”, because of their tree structure, and
- constraints. They bring **semantic richness**, because their graph structure allows to link anything to anything, everywhere.

SHACL and **ShEx** are two **W3C** standards that use a DOM structure themselves to represent constraint or tolerance relations, hence introducing graph-structured relations between DOM trees. So, the **composability** of the DOM meta meta model is clearly present.

9.4.8 Platform Independent/Specific Model, Domain-Specific Language

(TODO: **PIM**, **PSM**, **DSL**)

9.5 Block-Port-Connector for connectivity

An *architecture* determines how *to couple*, structurally and behaviourly, two or more activities, while allowing, at least, *to configure* and *to introspect* the status of the architectural couplings. The structural part of the architecture is realised by connecting the ports on activities, via which they interact with each other. In other words, a *model* of the architecture adds concrete contents to the **Block-Port-Connector (BPC)** *meta* model, and that concreteness comes from selecting one of the mechanisms, policies and/or best practices introduced in previous Chapters. For example, a Port provides the *data chunk* model and the status of the *coordination* protocol that were chosen for the particular **circular buffer** implementation of the inter-activity interaction of its own activity Block.

The system developers make such concrete BPC models for the different phases in the life of their system:

1. *design*: the structure and behaviour of activity interactions can already be represented, implemented and tested without the physical resources being available.
2. *deployment*: the BPC model should contain all information needed to deploy the above-mentioned implementations onto the physical system.
3. *configuration and coordination*: the BPC model should indicate which of its properties can be reconfigured at runtime, and what the status is of any coordination protocol that the activities are involved in.
4. *introspection*: the status of the BPC model can be *queried* at runtime, also by activities that are not part of the ones involved in the BPC model.

In order to keep the best practices this document advocates for **composable modelling**, the more concrete BPC model in a higher-numbered phase just adds an higher-order relation to the lower-numbered phase model. has the following design:

- structure of connectivity: a representation of which Ports are connected to which Blocks and which Connectors, with obvious type and cardinality constraints.
- structure of exchanged data: a representation of the data structures that travel through Ports and Connectors.

- coordination of connection: a representation of the flags visible via the Ports, and of how they connect to the Petri Net or Finite State Machine inside a Connector or Block. (TODO: more details and examples.)

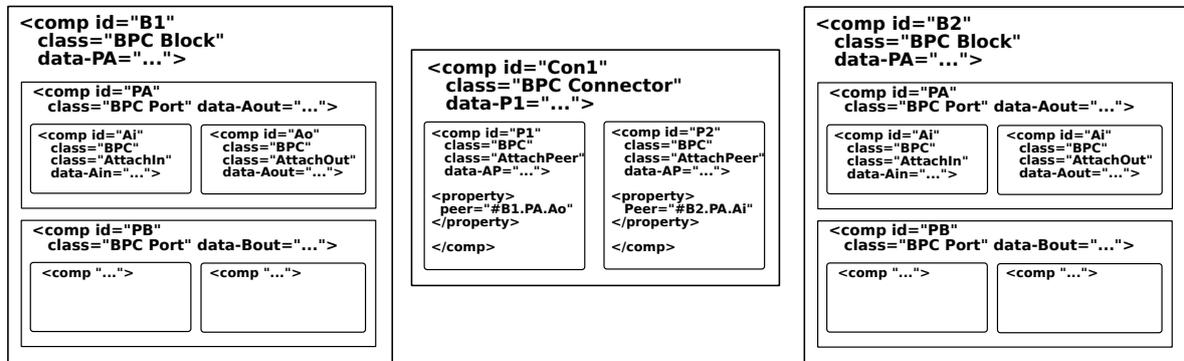


Figure 9.4: Example of DOM model for two Blocks being connected by a Connector, via their Ports. The Connector “points” to the Blocks’ outside-facing Port attachments via the symbolic pointers #B1.PA.Ao and #B2.PA.Ai.

9.5.1 DOM model

Figure 9.4 gives DOM model examples for two Blocks connected via one Connector. This approach of adding attachments to the entities that are connect decouples the connected entities in such a way that they only have to provide to other entities the *possibility* to be connected, without having to know the details or the purpose of the connection. This conforms to, both, the [property graph](#) and the [Block-Port-Connector](#) meta models.

(TODO: explain more.)

9.5.2 The mediator pattern in human society

Human society has introduced the mediator pattern many centuries ago already, by building systems-of-systems of high complexity with societal architectures that are now considered as “obviously” resilient subsystems. Some examples where an extra “mediator” organisation or entity has been introduced to cope with the complexity of the increasing number of interactions:

- in the context of *tasks* to organise the many interactions between humans into a loosely coupled societal hierarchy: persons organise themselves in families, they form neighbourhood, make up villages, organised into a metropolitan area around a central town, all assembled in regions and countries.
Each of these “political” structures has its own “government”.
- in the context of *tasks* to provide building infrastructures: rooms are connected by corridors, that form wards with some form of functional cohesion, aligned into floors, that form wings of buildings.
Each of these building structures has its own “floor and task plans”.
- in the context of *tasks* to organise industry: devices are interconnected with high-performance local networks into work cells; those form more flexibly and loosely com-

municating lines; these lines, in turn, are laid out in plants; that are logistically interconnected and coordinated as a company.

Each of these factory compartments has its own “foreman” for task execution **supervision**.

Other examples of societal instantiations of the mediators are: the *coach* in a team of athletes and staff, the *CEO* of a company, the *mayor* of a town, the *architect* of a large public infrastructure construction, etc.

9.5.3 The mediator pattern in cyber-physical systems

Examples where the pattern has been applied in engineering systems are:

- a motor control service for robot end-effector tasks: the mediator deals with the specificities of the kinematic chain that links the motors to the tasks, such as redundancy resolution, singularity avoidance, energy optimization, etc.
- system of systems integration where the communication performance *in* each system is high, while it is low *between* systems. For example, a production line in a manufacturing plant where the different devices in each work cell can communicate via optimized channels and components developed by the same team, while the coordination between the work cells can only make use of the less mutually optimized communication and behaviour coordination of independently developed machines from different vendors.
- **single-page applications**, such as a **graphical user interface** (GUI) for a very stateful system of systems but an extremely stateless GUI.

9.5.4 Two-ways symbolic indirection via third-party broker

(TODO: pointer is for binary (hence, fast) indirection in compiled code and data; scripts are for one-way symbolic-to-binary conversions at runtime via **interpretation** and **bytecode**; URL/IRI is for symbolic (hence, slow) indirection between models. This Section explains how both can be integrated, with the following structure:

- the indirection is *two-ways*.
- the mapping is done via *querying* the services of a *third party*, a so-called **broker**.

Hence, models must be available and editable at runtime. This is an extra cost at development time, but only a **one-off** cost. So, once the models are available at runtime, many interesting things become available. Also, many of the **bells and whistles** that differentiate “advanced” programming languages from C “just” become model-to-model transformations that can be shifted from compile time to runtime; for example: templates, event handling (including **bubbling** or **capturing** in a containment hierarchy), closures,...

9.5.5 Policy: extend a mediator into a broker

In its simplest form, the mediator activity has “just” the responsibility *to coordinate* other activities. The interaction primitives needed for such coordination are simple and **lightweight**: **flags**, **flag** (arrays), **Petri Nets**, and **Finite State Machines**. All coordinated activities must be *registered* with (or, *subscribed* to) the **stream** that exchanges the small data chunks of the just-mentioned coordination primitives. It is only a small increase in activity complexity to let the activity behind the stream take more responsibilities; for example, to become a

- **data centric broker**: in addition to doing the bookkeeping of registered peer activities, the broker version of a mediator also provides services on top of the raw data: *buffering, filtering, time stamping, delivery guarantees* in **pub-sub** communication patterns, etc.
- **application centric decision maker**: in addition to the brokering, the application also “outsources” (some forms of) decision making from the activities to the mediator activity. Because that latter activity knows about the interactions between all other activities, it is the logical place to make system-wide trade off decisions, such as **priority**; *voluntary data loss*, **curation** or **recovery**; **garbage collection**;; **dependency injection**; etc.

The more traditional⁹ *broker* pattern works well under the assumptions that (i) the broker can remain the same all the time, and (ii) the communication “middleware” loses no data, ever.¹⁰ The *mediator* pattern offers more opportunities to introduce policies to cope with violations of both assumptions.

9.5.6 Reification of a Connector and its Ports

The **mediator pattern** is essential to allow systems to deal with *dependencies* between sub-systems, by hosting all the *decision making* about the dependency coordination inside a dedicated activity. One way to look at this pattern is to consider it as the **reification** of the Connector of the **Block-Port-Connector** meta model, adding the following information to itself and/or to each of its Ports:

- *ID*: because the Connector becomes an activity in itself, it can be logged, monitored, inspected, . . . Of course, that requires that the Connector (and its activity) can be unique identified.
- *behaviour*: a Connector can have internal behaviour, while still representing itself via a stateless Port.
- *higher-order dependencies*: when a system becomes larger and/or more complicated, it is inevitable that system developers want to reconfigure a Connector that was static before, in order to take higher-order dependencies into account that did not exist before. For example, to adapt the size of a buffer inside a Connector to a change in the latency and bandwidth constraints available in a newly deployed communication capability.

(TODO: more details and examples.)

9.6 Mediator: hierarchy in decision making for inter-dependent components

Mediators structure the **decision making** in component systems:

- to realise **tasks** with a **quality control** that **satisfies** requirements.
- to use **resources** with an **efficiency** that is **affordable**.
- to be **robust** against those **disturbances** that are **relevant**.

⁹Standardized in the 1990s, with **CORBA** (*Common Object Request Broker Architecture*) and **Data Distribution Service (DDS)** as major representatives.

¹⁰Typically, middleware projects provide one or more of the following three policies to guarantee message delivery: **exactly-once**, **at-least-once** or **at-most-once**. Guaranteeing such policies always comes with high costs, under conditions of communication disturbances.

The **mediator pattern**¹¹ represents an architecture in which one activity is dedicated to satisfying the **constraint relations** (or “*dependencies*”) between the behaviours of two or more activities. In order to coordinate the activities explicitly, each application domain must add concrete models of the behaviours, the dependencies and the interactions via which the activities can influence each others’ behaviours. This Section only deals with the aspects that hold for all application domains; the Chapters on **information** and **software** architectures provide the more concrete use cases to which the mediator architectural pattern is applied.

The **core design** of the mediator pattern is that an **extra activity** is introduced for **every inter-activity coordination** that must be introduced whenever:

- the activities involved share a resource.
- none of them must, or can, be aware of the presence of the others.

Hence, the individual activities do not feel the complexity of their dependency and influence on the other activities.

9.7 Holons for heterarchical decision making in task execution systems

Systems are made available because they offer *capabilities* to realise **tasks** that add value for customers while making good use of the *resources* that the customers want to pay for. (**Robotic tasks** are specific instances of the more generic task concept in this Chapter.) From a system architecture point of view, the essential aspect of a task is the **ownership** that “someone” has to take about *every* task and *every* resource. It is this context in which the concept of a **holarchy** (or *holonic architecture*) was born, [60, 90], first of all as a model of *resilient* organisational systems-of-systems, in a socio-economic context.

Holons care about the two other aspects of composition in the **5C** meta model, namely **communication** and **computation**.

In the context of the “**5Cs**” meta model, a holon architecture of a (composite) component adds **communication and computation** to the hierarchical DOM structure for *coordination and configuration*: only the top-level component owns the interfaces for communication and behaviour towards the “outside world”, *or* it is the unique component in its DOM structure that sets the configurations for all communication interactions of all other components in its DOM. This communication structure is *heterarchical*, because the communications/interactions between holons has no specific structure in itself, so there is no first-class role for “hierarchy”.

9.7.1 Resilient, stable, robust sub-systems

This document uses the terms **stable** or **robust** as synonyms for *resilient*. **Anti-fragile** systems are outside of the scope of this document, for now; that is, the document focuses on *known unknowns* only. An alternative conceptual classification that is relevant to this Chapter is that of the **Cynefin framework**: this document deals with the *clear* (“best practices”) and *complicated* cases (“good practices”); its design concepts support the structuring of *complex*

¹¹The article on this pattern on the Wikipedia does not cover the “(meta) meta” and “coordination” aspects, but focuses on the software engineering aspects only. This document’s focus is on the former aspects, as well as on more aspects than data communication, more in particular, on *coordination* and *composition*.

cases (“emergent practices”), but not of *chaotic* (“novel practices”) or *disorder* cases (“no practices”).

This document redefines the mereo-topological interpretation of these concepts in the context of the architecture of engineering systems:

*A system has a **resilient** architecture if its behaviour is **explainable**, and remains so under any change in the behaviour of any of the systems it interacts with, as well as any change in the topology of its interactions.*

9.7.2 Dependencies in architectures: hierarchy, heterarchy and holarchy

(TODO: hierarchy for knowledge and context, heterarchy for command and control; holarchy: architecture with, both, explicit allocation and responsibility about ownership of everything, and **theory of mind** awareness of each holon’s individual role in an **society** of **peers**.)

9.7.3 Explainability: causal driver for robustness, resilience, anti-fragility

Explainability is the first step in a **hierarchy** of system **behavioural resilience metrics**; **adaptability**, **predictability** and **dependability** are next. And **guaranteeability** is the holy grail. A **working hypothesis** of this document is that the explainability of an architectural design of a system is proportional to the level of **knowledge concentration** that the designer can achieve in providing one **unique mediator** component that makes the decisions about how to coordinate its own system-level behaviour with that of all of its internal subsystems, as well as with all of the external peer systems. In other words, explainability can only be achieved when *all* decisions the system makes to adapt its behaviour to its context, are made in one single place, based on one consistent combination of (offline) *knowledge relations* and (online) *world model data*. Of course, trying to apply this design driver blindly, by **centralising** decision making, has time and again proven not to be a good approach; the more resilient architectures have “holarchies” of resilient sub-systems, with the “right” loose coupling of their behaviours and, especially, their decision making.

The system architects’ responsibility remains huge, because computer reasoning can’t¹² deliver the **wisdom** of deciding the subsystem boundaries where robustness, explainability or resilience can be optimized.

9.7.4 Key Performance Indicators

(TODO: robustness: explainable reaction to known disturbances; resilience: explainable reaction to unknown disturbances, or rather, to disturbances that one can *detect* but not *classify*. How to measure performance? Benchmarking/KPIs for task, environment, and technology, separately and composed.)

9.8 Autonomy as explainable decision making

Like most other terms in this Chapter (“architecture”, “safety”, “system of systems”,...) *autonomy* has been given several different definitions, although none of them is sufficiently constructive to be used as a **refutable** design driver. This Section introduces such a refutable

¹²This is a very **refutable** claim. Readers are kindly invited to provide such a refutation.

design for autonomy, by associating autonomy with explainable decision making; the underlying (refutable) hypothesis is that a system can only be called “autonomous” if it can explain *why* it makes its decisions.

9.8.1 Endsley’s five levels of system autonomy

Already in 1987, **Endsley** [33] identified **five levels of autonomy**. Albeit in the context of computer support for fighter pilots, her autonomy levels are relevant to all cyber-physical and robotics systems too; the domain of **autonomous cars**, has taken Endsley’s classification as its inspiration for its own **autonomy levels** specification.

Level	Description
5	Computer makes decisions, with the human completely out of the loop.
4	Computer makes recommendations to the human which it will carry out unless the human vetos.
3	Computer makes recommendations to the human which it will carry out if the human concurs.
2	Computer makes recommendations to the human which he may choose to act on or not.
1	Computer offers no assistance, human makes all decisions & actions.

9.8.2 Sheridan’s ten levels of system autonomy

Sheridan refined Endsley’s scale to **ten levels of autonomy** [74], Table 9.1. Endsley [35] refined this scale further.

Level	Description
10	Computer does everything autonomously, ignores human.
9	Computer informs human only when it sees fit.
8	Computer informs human only if asked.
7	Computer executes automatically, and informs human when necessary.
6	Computer allows human restricted time to veto before starting action.
5	Computer executes suggested action if human approves.
4	Computer suggests one single alternative.
3	Computer narrows alternatives down to a few.
2	Computer offers a complete set of decision and action alternatives.
1	Computer offers no assistance, human makes all decisions & actions.

Table 9.1: Sheridan’s ten levels of system autonomy [74].

9.8.3 Explanation levels for autonomous decision making

Endsley’s and Sheridan’s scope was limited to the interaction between one *single* machine and one *single* human. Modern robotic and cyber-physical systems must extend this scope to *systems-of-systems* contexts, with *multiple* agents, multiple tasks, multiple resources, multiple vendors, multiple regulators, and multiple machines. This document introduces a definition of “autonomy levels”, Table 9.2, based on the **dialogue** with which a system **explains its decisions** to other agents, human as well as artificial. In other words, this scale is *constructive* (it

says *how to measure* whether a level of autonomy is reached), while the scales of Endsley and Sheridan are *qualitative* (it is still only a human who can assess whether a system has reached a particular level of autonomy). The granularity of the levels is designed to allow incremental *step change* developments in autonomy, for very focused decision making challenges. Note the huge technical challenges to go from “level 4” to “level 5”, and, especially, from “level 6” to “level 7”. These steps introduce two subsequent levels of **empathy**: (i) to reflect on one’s own actions and put them in the context of the **user** for whom a Task is being executed, and (ii) to create and maintain also the world models of **other systems**, and to reason in their place.

Level	Description
One system — One task	
1	What am I doing?
2	Why am I doing it?
3	How am I doing it,...
4	...and how well am I doing it,...
4b	...and why am I doing it this way?
4c	...and how do I decide to stop doing it?
One system — Multiple tasks	
5	What could I be doing instead,...
5b	...and still be useful,...
5c	...and how do I decide to switch what I am doing?
6	What is threatening my progress,...
6b	...and how can I make myself resilient,...
6c	...and how do I decide to add a particular resilience?
Multiple systems — Multiple tasks	
7	What progress of others am I threatening,...
7b	...and how can I make myself behave better,...
7c	...and how do I decide to adapt a particular better behaviour?
8	What other machines and humans can I cooperate with,...
8b	...and how do I find out how we can coordinate our cooperation,...
8c	...and how do we decide, together, what coordination to adopt,...
8d	...and how do we monitor our coordination,...
8e	...and how do we decide that someone has cooperation problems?

Table 9.2: Systems’ decision making explanation levels. They represent the various degrees to which systems can (i) perform self-diagnosis monitoring and Coordination, (ii) *explain* their autonomus decision making, and (iii) adapt in order to increase resiliency [69].

9.8.4 Role of meta models in horizontal and vertical task composition

A system’ decision making levels of Table 9.2 are agnostic to the scale of the system, so they are also relevant for any type of horizontal and vertical composition of **tasks**. If such a composition is constructed with knowledge-driven hybrid constrained optimization (KHCOP), the on-line solvers of such KHCOP problems have already answers to some of the questions in Table 9.2, because decisions are made in the Coordination state machines and Task progress

is monitored semantically via constraint violation checks.

This is the *first level* of explainable decision making: every decision is associated with a set of constraint violations. The meta model of the system represents the associations between these constraints explicitly and symbolically. System designers must try to add also the *second level* of explainable decision making, by the higher-order associations that represent the **reasons why** these constraints have been configured to be the ones to monitor. Task compositions are a major source of this higher-order association: every composition is introduced for a reason, and representing that reason explicitly and symbolically is the way forward. This document’s **Task-Skill-Resource** meta model already links three complementary sources of such higher-order associations: the requirements of the task, the situational context of the environment, the limitations of the resources, and the knowledge relations used in the skill that composes them together.

(TODO: examples.)

9.9 System safety as explainable decision making

(TODO: explain why safety is a system-level aspect, and composes parts of the HCOPs in all the current Tasks. The role of the LCSM as the model for an independent “**safety PLC**”.)

9.9.1 Best practice: safety PLC

(TODO: third-party monitoring, detection and reaction to solve first-party’s unsafe conditions.)

9.9.2 Best practice: active safety behaviour

(TODO: first-party monitoring, detection and reaction to prevent third-party’s unsafe conditions.)

9.10 System security as explainable decision making

Security is a system- and application level aspect, because (i) no middleware can be trusted, and (ii) no task specification can be trusted. Hence, and in addition to the best practice in encrypting all communications, the application has to engage in *dialogues* with all parties that provide or consume data and task specifications. The dialogue consists of back-and-forth questions and answers with a large enough variety in encrypted keys to exclude man in the middle problems. In this document’s broader context of knowledge-driven systems engineering, these dialogues do not impose a lot of extra overhead because a lot of messages are already exchanged for other “non-functional” purposes, such as heartbeats, resource mediation, and task coordination.

(TODO: lot more concreteness.)

Chapter 10

Meta models for information architectures

An **information architecture** of a system is the design layer between an **application's task requirements** and the *software* and *hardware resources* that the system has available to realise these application requirements.

This Chapter applies the **mediator** pattern to create information architectures for core sub-systems and components of robotic and cyber-physical systems. Internally, a mediator is **primarily a hierarchical structure**, that relies on:

- a **DOM model** of the hierarchical dependencies between the information structures that the mediator must provide a decision making context for.
- a **Life Cycle State Machine** to coordinate the creation, operation and removal of the coupling.
- **event loops** providing tree-structured control flow models to compose the *activation* algorithms and streams together inside activities.
- **streams** add **heterarchical** data exchange connections between the tree structures in one, or between more, mediator hierarchies in all asynchronous activities involved in the coupling.

An information architecture represents the **structure** and **behaviour** of an application in a way that **does not depend** on concrete choices¹ of *software* (programming languages, development tools, communication middleware or protocols, operating systems, . . .) or *hardware* (CPU architectures, memory layouts, communication networks, . . .). However, an information architecture **does depend** on the application's *tasks*, because it must guarantee the **completeness**, **correctness**, and **semantic explainability**² of how the activities in the architecture realise the application. An information architecture is also the place to embody **design contracts** between the developers and customers of an application.

An information architecture has three complementary objectives:

1. to embed **synchronous** behaviour inside **activities**.

¹To couple the information architecture to all these “platform” choices is the task of the **software architects**, who must make the system *work*, and work *efficiently*.

²The **buzzword** of this decade is **explainable AI**. Like with all buzzwords, the motivation and expectations behind it are absolutely relevant and real. But most of its realisations do not live up to these expectations.

2. to execute activities **asynchronously** inside **components** with **interfaces**.
3. **to compose** these components, so they **interact** to form a system.

When done right, a higher granularity in the interfaces and a richer behaviour in the interaction mechanisms, allow a higher **composability**, at the cost of more components and more complicated interfaces and interactions. Each particular architecture design **trades-off** the system's **reconfigurability** with its **compositionality**. A higher reconfigurability comes with extra costs (at design time as well as deployment time and runtime) but (should) result in a system that can adapt more to changing application requirements. Each particular architecture design is **constrained** by the application's task requirements (that introduce **artificial constraints**) and by the resources (that bring **digital and physical constraints**).

This document's approach to reach the architecture design objectives is:

- to create the application's **task models**, and the **dependencies** between the models of the multiple tasks that can run in sequence or concurrently .
- to create the **state models**,³ to represent the application behaviour.
- to create **functions** that turn task and state models into behaviour.
- to decide which behaviour to realise in **activities**.
- to realise that behaviour in an **event loop** inside each activity, with (preferably) **lock-free mechanisms** for data exchange.
- to decide which activities to put together in **components**.
- to make components **discoverable**, with **queryable interfaces**.
- to create flags, Finite State Machines and Petri Nets **to coordinate** the **asynchronous** execution of activities and components.

10.1 DOM models in robotics geometry

This Section and the following introduce **DOM models** for *information architectures*⁴ in several of the robotic sub-domains introduced in earlier Chapters. The purpose of these DOM models is to represent *parts* of an information architecture with a *tree structured* formal representation. The selection of which parts in a particular domain are best suited for such efficient modelling (because of the tree structure) while not compromising the semantic expressivity of the model (which, in general, comes from graph structures), is continuously open for debate and adaptation.

10.1.1 DOM model for polygonal shapes

Representation of 2D and 3D shapes is very important in robotics; all of the other subsections below rely on shape representations in one way or another. There are two major, complementary approaches to represent shapes of **solid** objects:

- **Constructive Solid Geometry**: a solid object is modelled as a set of Boolean operators on primitive solids. For example, one starts with a cube from which one subtracts two orthogonal cylinders, whose axes are intersecting the centre of the cube and run through the outer boundary of the cube.

³Often also called "world models" in this document

⁴The **software** representations, tools and implementations are introduced in a later Chapter.

- **Boundary Representation (B-Rep)**: a shape of a solid object is modelled as connected surface elements. The simplest form uses **polyhedrons** (in 3D) as a connection of planar 2D **polygons**. This representation must explicitly represent what is “inside” and “outside” of the solid that is represented by the polygonal faces, and one must take care “to close” the solid appropriately. Many of the domains discussed below can do with non-solid polygonal shapes, such as lines or frames.

The DOM model for polygons and polyhedrons already exists in a very mature and standardized form, as the **SVG 2.0** (Scalable Vector Graphics) DOM meta model. These are the relevant parts from that standard for this Chapter (whose details are not repeated in this document because they are available in detail online):

- **DOM Point**: points are the obvious core primitives in polygonal shape models. The SVG standard supports points not only in **Euclidean space**, but also in **projective space**, that is, a point can lie “at infinity”, and Euclidean shapes can be transformed into **perspective** views.
- **basic shapes**: the **elements** of rectangle, circle, ellipse, line, polyline, polygon.
- **geometry properties**: symbolic names for point and shape coordinates, like centre points of circles, or width and height of rectangles. These land in the DOM model as **attributes** in the elements. For example:

```
<rect x="1" y="1" width="1198" height="398" />
```

```
<polygon points="350,75 379,161 469,161 397,215
                423,301 350,250 277,301 303,215
                231,161 321,161" />
```

These examples show *fully specified* DOM models: each of the two elements has all of its necessary geometric parameters filled in with concrete numeric values. Both are compositions of *point* coordinates:

- the **<polygon>** element has a **points** attribute, that is an *ordered list* of, in this case, 2D **x** and **y** point coordinates. The element has the implicit⁵ constraint that the first point also acts as the (not represented) last point in that list, in order to close the polygonal shape.
- the **<rect>** element has attributes for the **x** and **y** point coordinates of its (implicitly modelled) top-left corner point, and its **width** and **height** attributes provide the information to reconstruct the other three points of the rectangular polygon. The orthogonality of the lines at the four corners of the rectangle is also not explicitly represented, but symbolically, in the name “rectangle”.
- **composition entities and relations**: meta data definitions, templates, hierarchical grouping, symbols and symbolic pointers. That is, the following **tags**: **<g>...</g>** (that realises the *tree structure* containment), **<defs>...</defs>**, **<use>...</use>**, and the *functions* **url()** (for symbolic referencing), **calc()** (for computations on symbolically represented parameters), and **transform()** (for **moving and deforming** the basic shapes into arbitrary shapes, at arbitrary positions and orientations).

One single **shape** element can host multiple *attachments* to connect other elements to, other **shapes**, as well as non-geometric elements. For example, a robotic application can need a model of a **kinematic_chain** that has a low resolution as well as a high resolution model of the **shape** and **inertia** of its links. The low resolution version can just have a **stick figure**

⁵*Implicit* means that this constraint is in the *meta model*, not in the model itself.

shape model, and a **point mass** inertia model.

10.1.2 DOM model for coordinates

The **SVG DOM** of the previous sub-section has a meta model for coordinates. (**QUDT** provides the *knowledge relations* for coordinates. As mentioned **elsewhere**, both types of modelling are highly complementary.) Figure 10.1 depicts a simple example of the composition of two DOM models to represent a rectangle:

- one for the SVG DOM model of the rectangle represented only with *symbolic* coordinates. This model makes use of the CSS `url()` function, that acts as a *symbolic pointer*.
- one for a coordinates DOM model containing the *numerical values* for the symbolic coordinates.

Both DOM models are *symbolically* connected by the “symbolic pointer” `symRec1`.

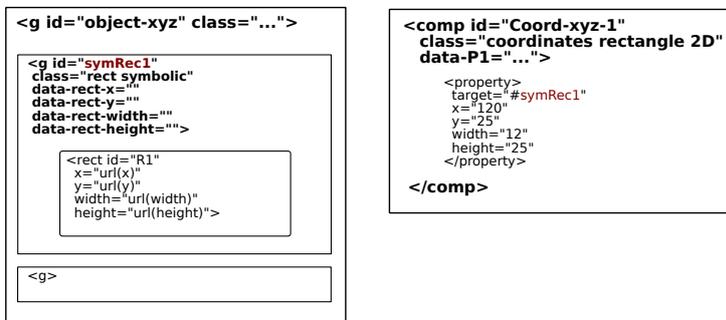


Figure 10.1: Two DOM trees, one for a rectangle conforming to the SVG DOM meta model, and one for the coordinates conforming to this document’s DOM meta model. The symbolic connector between both is the `symRec1` identifier.

One single geometric entity can have multiple **coordinates** DOM elements attached to it. From a modelling point of view, this is as simple as introducing multiple **Connectors** from the same geometric **Blocks** to different coordinate **Blocks**.

10.1.3 DOM model for geometric and kinematic chains

The DOM model below is an example of a very simple **geometric constraint**, the **kinematic chain**, built from one revolute joint between two rigid links, each providing some **attachments**, in the form of a **frame**. Again, the textual order of the DOM elements below does not matter, but the hierarchy in the cross-links does. The cross-links, or “symbolic pointers”, are represented in the DOM document by putting a **number sign #** in front of the ID of a DOM element. In the example below, `#L1` is the symbolic representation of the link L1 in an expression in another element.

```
<link id="L1" class="rigid">
  <property id="attachment.a" class="attachment frame">/property>
  <property id="attachment.b" class="attachment frame">/property>
  <property id="attachment.c" class="attachment frame">/property>
  <property id="attachment.d" class="attachment frame">/property>
</link>

<link id="L2" class="rigid">
  <property id="attachment.a" class="attachment frame">/property>
  <property id="attachment.b" class="attachment frame">/property>
</link>
```

```

<relation id="J12" class="constraint geometry motion joint revolute">
  <property id="proximal" class="role-proximal-attachment">
    #L1/attachment.a </property>
  <property id="distal" class="role-distal-attachment">
    #L2/attachment.b </property>
</relation>

<geometric_chain id="Chain1-2" class="kinematic">
  <property id="JointList" class="joint-list"> [#J12] </property>
</geometric_chain>

```

The models of all the elements are still limited to the [mereo-topological](#) parts only, that is, the minimal amount of symbolic relations required to be able to speak about something in which a domain expert can recognize the semantics of a “kinematic chain”. The element at the top of the DOM hierarchy, `<geometric_chain>`, conforms to all the classes that its children need:

- the `geometric_chain` is a `kinematic chain`, that is, only a specific set of geometric motion constraints are allowed in the model.
- the `<link>` is a geometric entity of class `link` and that link is `arigid` solid.
- the `revolute joint` is a `constraint` on the relative motion of both links.

Because of the CSS cascading property, it is strictly speaking not necessary to repeat the classes for each of the children; but it helps in storing the components of the full DOM model in separate documents, or in streaming them over a communication channel, without losing the full semantic context.

The next paragraphs will compose, step by step, extra semantics to the minimal model above. These steps illustrate how composition is done, while (i) keeping *semantic hierarchy*, and, at the same time, (ii) being robust against any accidental *textual nesting*. “Accidental” means that the meta model developers were not aware of (i) the fundamental difference between both hierarchies, (ii) the fundamental differences between [properties and attributes](#), and (iii) the artificial constraints that textual nesting introduces.⁶

The first composition step is that of a `kinematic_chain` model that has more than one `joint`. This does not add a new hierarchical level, but just an element on the already existing `joint` level of the DOM. So, the model below can reuse the model above, and just add the new model elements: a new `link`, with ID `L3`; a new `joint`, with ID `J23`; and a new `geometric_chain`, with ID `Chain12-23`. The previously modelled entities and relations are incorporated by referring to their IDs.

```

<link id="L3" class="rigid">
  <property id="attachment.a" class="attachment frame"></property>
  <property id="attachment.b" class="attachment frame"></property>
</link>

```

⁶[URDF](#) (Unified Robot Description Format) is a popular DOM model in robotics, whose design includes many DOM modelling errors. The major cause of the problem is putting *attribute* elements *inside* the node of which they are attributes, instead of introducing a *dedicated* attribution element to encode that particular relation between a node and each of its attributes. (For example, URDF elements can have only one single set of coordinates, because they are represented as a property element nested *inside* a geometric element.) For both attributes and properties, the URDF design neglects the *cascading* and [multiple conformance](#) features of state of the art DOM models. (For example, the DOM model can not be used in 2D and 3D Cartesian worlds without significant changes, while in principle all that is needed is to change an attribute flag from 3D to 2D, only in those elements that are impacted by the difference.)

```

<relation id="J23" class="constraint geometry motion joint revolute">
  <property id="proximal" class="role-proximal-attachment">
    #L2/attachment.a
  </property>
  <property id="distal" class="role-distal-attachment">
    #L3/attachment.b
  </property>
</relation>

<geometric_chain id="Chain12-23" class="kinematic">
  <property id="JointList" class="joint-list ordered"> [#J12, #J23] </property>
</geometric_chain>

```

The composition above is of the **serial** type. The two other types are **branch** and **loop**.

A **branch** connects three **serial** chains via the same link. In other words, one link is connected not to one but to two (or more) other links:

```

<link id="L4" class="rigid branch">
  <property id="attachment.a" class="attachment frame"></property>
  <property id="attachment.b" class="attachment frame"></property>
  <property id="attachment.c" class="attachment frame"></property>
</link>

<relation id="J34" class="constraint geometry motion joint revolute">
  <property id="proximal" class="role-proximal-attachment">
    #L3/attachment.a
  </property>
  <property id="distal" class="role-distal-attachment">
    #L4/attachment.a
  </property>
</relation>

<relation id="J4X" class="constraint geometry motion joint revolute">
  <property id="proximal" class="role-proximal-attachment">
    #L4/attachment.b
  </property>
  <property id="distal" class="role-distal-attachment">
    #LX/attachment.a
  </property>
</relation>

<relation id="J4Y" class="constraint geometry motion joint revolute">
  <property id="proximal" class="role-proximal-attachment">
    #L4/attachment.c
  </property>
  <property id="distal" class="role-distal-attachment">
    #LY/attachment.a
  </property>
</relation>

<geometric_chain id="Chain1234XY" class="kinematic branch">

```

```

    <property id="JointList" class="joint-list ordered">
      [#J12, #J23, #J34, [#J4X, #J4Y] ]
    </property>
  </geometric_chain>

```

A **loop** is a serial chain that closes on itself. Assuming we have a `kinematic_chain` with five links and five joints, and the first link is the one that is *arbitrarily* chosen to be the one where the loop is closed, this could be (the relevant fragment of) the DOM model:

```

<geometric_chain id="Chain12-23" class="kinematic loop">
  <property id="JointList" class="joint-list ordered">
    [#J12, #J23, #J34, #J45, #J51]
  </property>
  <property id="loopLink" class="chain-loop-link arbitrary"> #L1] </property>
</geometric_chain>

```

10.1.4 DOM model for shape and inertia kinematic chain links

Any `shape` from a `world model DOM` can be used, after it has been given an `attachment` entity:

```

<shape id="Shape1" class="shape geometry 3D">
  <property id="attachment" class="role-proximal-attachment">
    attachment.S
  </property>
  <property id="geo" class="simplex polygon geometry 3D">
    <svg> ... </svg>
  </property>
</shape>

<relation id="L1-Shape1" class="constraint attachment geometry shape link">
  <property id="link" class="role-link-attachment">
    #L1/attachment.c
  </property>
  <property id="shape" class="role-shape-attachment">
    #L1/attachment.S
  </property>
</relation>

```

An `inertia` model example is given below. The actual values in the inertia matrix are not yet filled in, but made linkable via the `Mass1.geo` symbolic ID:

```

<shape id="Mass1" class="inertia geometry 3D">
  <property id="attachment" class="role-inertia-attachment">
    attachment.I
  </property>
  <property id="geo" class="inertia geometry 3D matrix">
    <matrix> #url() </matrix>
  </property>
</shape>

<relation id="L1-Mass" class="constraint attachment geometry inertia link">

```

```

<property id="link" class="role-link-attachment">
  #L1/attachment.d
</property>
<property id="inertia" class="role-inertia-model">
  #Mass1/attachment.I
</property>
</relation>

```

10.1.5 DOM model for actuator and transmission in kinematic chain

A mainstream robot has one actuator, one transmission and one 1-D joint per link. The transmission is (often a constant) *geometric* factor, the **gear or torque ratio**, multiplying the *actuator torque* into the *joint torque*, that is applied to the link. The *dynamic* properties of a transmission are its *elasticity* and *inertia*.

```

<transmission id="trXX" class="geometry 1D">
  <property id="attachment" class="role-transmission-attachment">
    attachment.T
  </property>
  <property id="gear-ratio" class="geometry 1D" data-gear-ratio="3.4">
  </property>
</transmission>

<relation id="J12-Actuator" class="constraint attachment geometry joint transmission">
  <property id="joint" class="role-link-attachment">
    #J12/attachment.a
  </property>
  <property id="actuator" class="role-actuator-attachment">
    #trXX/attachment.T
  </property>
</relation>

```

(TODO: but **multi-articular** transmissions (for example, differential drive torque splitters, or finger tendons), require some extra relations.)

An **actuator** DOM model has the following form:... (TODO: similar to Inertia: there is one actuator connected to a transmission.)

10.2 DOM model for numeric, symbolic and semantic maps of world, building, and floors

Many robotic applications need to represent the location of a robot, on a particular floor of a particular building, somewhere in the world. DOM models for location on the world are already mature and standardized, in the domain of **GIS** (Geographic Information Systems); **OpenStreetMap** is a very accessible example.

The state of the practice in modelling buildings and floorplans is worse. For buildings, the **BIM** (Building Information Modelling) standard exists, but it is optimized for use in building design and construction, and is still difficult to apply to the sensing and control contexts of robotics.

For **floorplans** (that is, one single layer in a building), this document introduces the following suggestion for DOM models:

- **base layer**, with *numerical values* for the coordinates of all “corner” landmarks in the map, that is, walls, doors, and windows.
- **symbolic layer**, with (i) the *symbolic DOMPoint* names for these landmarks, and (ii) the *symbolic constraint representations* of the geometrical shapes that connect these landmarks, that is, the *rectangles* and *polygons* of Sec. 10.1.1.
- **semantic layers**, adding meaning to selections of entities in the symbolic layer, that is, to represent **rooms, doors, corridors and elevators**, etc.

Figure 10.2 sketches the concep. Of course, each of the three composing DOM models can consist of a tree hierarchy itself.

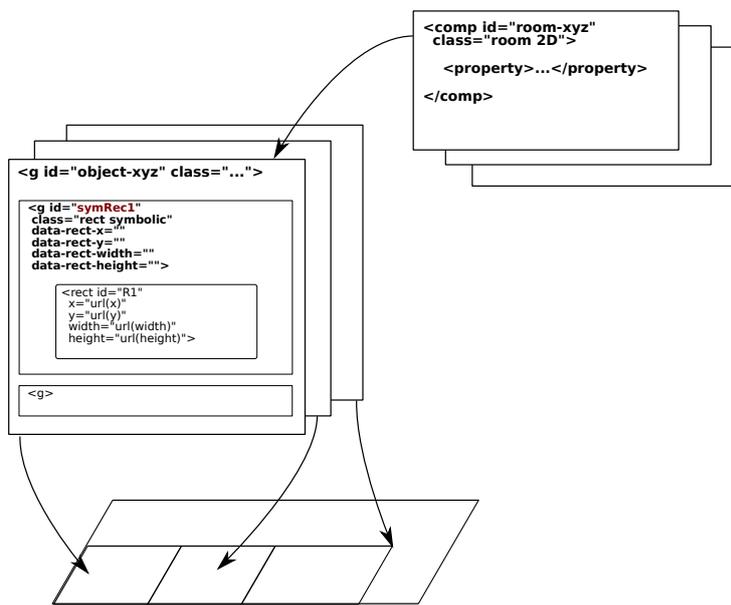


Figure 10.2: Sketch of a DOM model of three layers of other DOM models trees: (i) a basemap with *coordinates* of polygonal shapes, (ii) a map with the *symbolic* names of all relevant corner points on the basemap, and (iii) a map with *semantic* entities and relations between names on the symbolic map.

10.2.1 DOM model for semantic localisation and navigation

(TODO: component classes and tree, that add landmark models to plans; coordination and configuration; model CRUD interaction.)

10.2.2 DOM model for motion tasks

(TODO: component classes and tree; coordination and configuration; data and event interaction streams.)

10.2.3 DOM model for instantaneous motion specification in kinematic chains

Section 4.4.2 introduced the meta model to specify the instantaneous motion of a kinematic chain. The entities and relations introduced there are straightforward to encode in a DOM model:

- Cartesian forces and acceleration constraints on a link need an **attachment** entity on that link.

- actuator torques on a joint are specified in the `attachment` entity that *is* already on the joint.

The formalisation in Table 4.3 is straightforward to encode with the DOM meta model grammar used in this Chapter.

(TODO: chain-wide constraints of redundancy and under-actuation; support chains with loops and with multi-joint transmissions; **agonist-antagonist** actuation.)

10.3 DOM model for cascaded control loops

(TODO: component classes and tree; coordination and configuration; data and event interaction streams.)

10.4 DOM models for activities

This Section adds concrete DOM models to the meta models of the five building blocks for activities: **data**, **functions**, **algorithms**, **streams** and **event loops**.

10.4.1 Data, functions and algorithms

Figure 10.3 sketches a DOM for a **data type**. Standardized DOM meta models with (more or less) the semantics of what this document advocates exist, such as **Apache Arrow** or **HDF5**. So, this document does not try to suggest new alternatives.

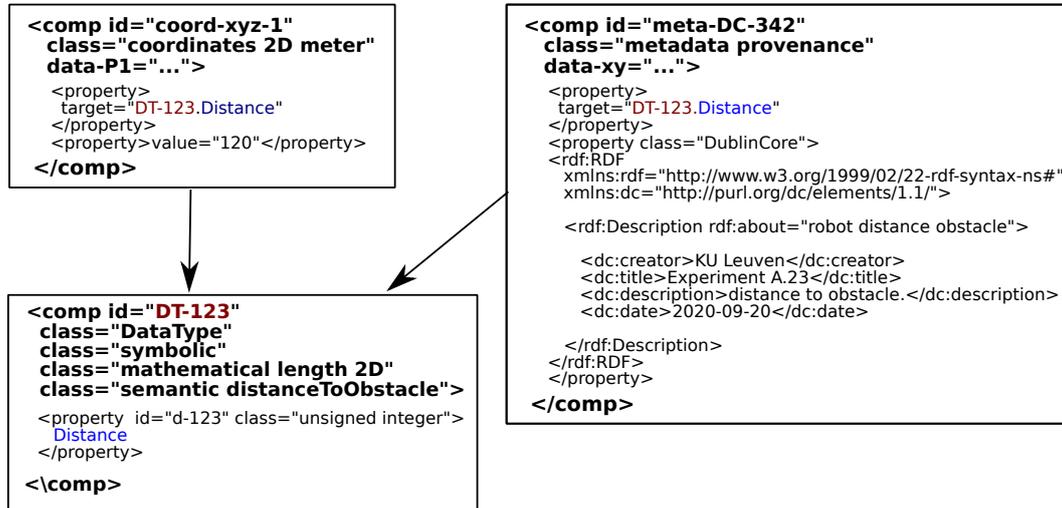


Figure 10.3: Sketch of a DOM model for a **data type**, with a relation to (i) a **digital structure** DOM for its coordinates, (ii) a **metadata** DOM.

Figure 10.4 sketches a DOM model for a **function**, and Fig. 10.5 that for an **algorithm**. The latter DOM model adds *dependencies* between functions; its DOM model contains representations of the following architectural **constraints** on the execution order of the functions in the algorithm:

1. constraints on the **serial and parallel** execution of functions inside an algorithm.

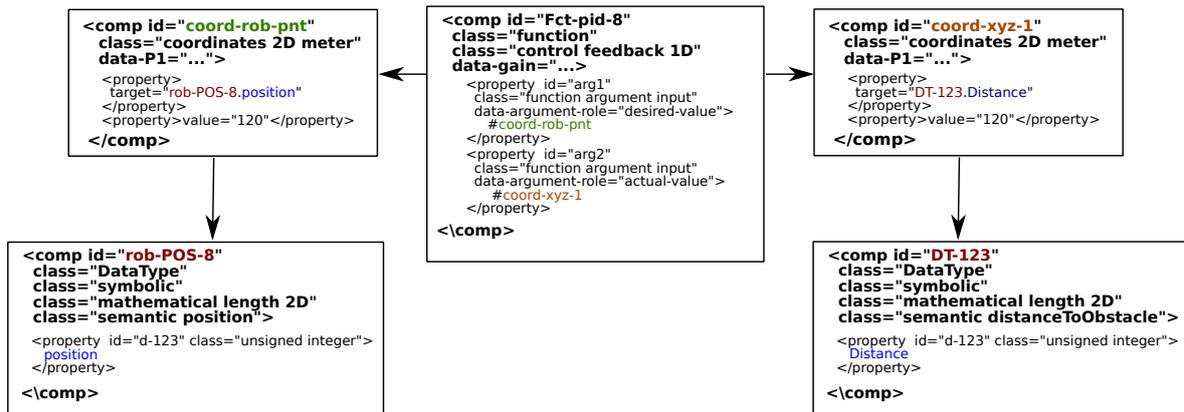


Figure 10.4: Sketch of a DOM model for a **function**, with three arguments, each being a data structure as in Fig 10.3. The relation to the function’s metadata (not shown in the drawing) is similar as in the latter Figure. The arrows and colours are not part of the textual DOM, but are added to the drawing to help the reader follow the symbolic pointers inside the various DOM elements.

2. constraints on the **conditional branching** in an algorithm’s **control flow**.
3. constraints on the access to the data arguments of the functions.
4. **Life Cycle State Machine** constraints on the selection of which (sub)algorithm to activate at any moment.
5. **Petri Net** constraints on the configuration of *flags* in the algorithms. These are part of the above-mentioned conditional branching, for all those conditions whose logical value depends *also* on data that is updated by algorithms “elsewhere” in the system.

The first two items are not in the focus of this document; and to the best of the authors’ knowledge, no widely supported standards exist. However, these constraints, and also the third one, are just special imperative cases of the more declarative models of state machines and Petri nets. These more complex mechanisms are *inevitable* anyway whenever algorithms, functions and data have to be composed in concurrent or asynchronous deployments. So, the generic DOM model of the *algorithm* entity is as depicted in Fig. 10.5

10.4.2 Activities, streams and event loops

An **activity** composes:

- one or more **streams**, that bring in, or get out, the data for the algorithms inside the activity.
- one **event loop**, that serializes the **4Cs** of the algorithms.

Streams and *event loops* couple data and functions together, in **heterarchical** ways; hence, they are the complement of the **hierarchical** structure of *data*, *functions* and *algorithms*. The *activity* is the “**holon**” where all this comes together, more in particular the decisions to configure and coordinate streams and event loops; this decision making in itself is structured in a hierarchical DOM, although the streams and event loops themselves represent *heterarchical* couplings.

Figure 10.6 sketches the structure of the DOM of a **stream**:

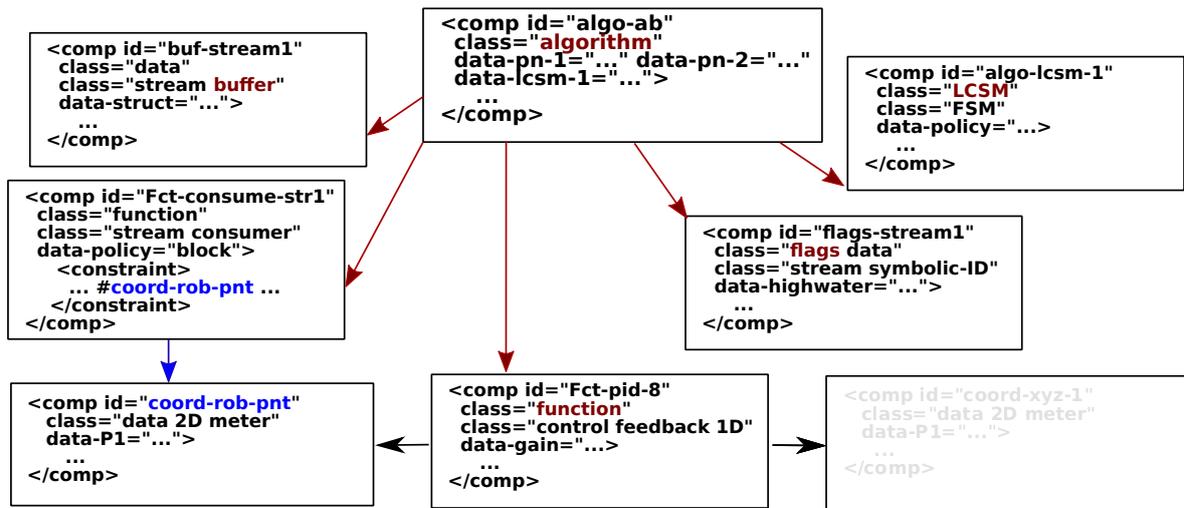


Figure 10.5: Sketch of a DOM model for an **algorithm**, where the top-level **algorithm** element is the parent composition of several children components. One of those children, the stream’s buffer, provides a constraint on the first data argument of the function. Access to the function’s second data argument is not constrained. The stream’s *produce* function is taken out of the scope of this Figure.

- the stream *coordinates and configures* the stream’s (i) buffer, (ii) its coordination flags, and the (iii) *consume* or *produce* functions that get data in and out of the buffer.
- the activity must *connect* the stream functions to the data that the algorithm functions work with, making sure that data consistency *constraints* are met.
- there are many *mechanisms* and *policies* in streams whose configurations must be set appropriately for the application context.

... (TODO: include figure.)...

Figure 10.6: Sketch of a DOM model for a **stream** which coordinates the *data access* between two functions: (i) a “useful” function for the application (i.e., the one in Fig. 10.4) and (ii) a “infrastructure” function that consumes the stream for the “useful” function and makes the stream data accessible.

Figure 10.7 sketches the structure of the DOM of an **event loop**:

(TODO: DOM of event loop; DOM of activity composing the DOMs of stream and event loop.)

... (TODO: include figure.)...

Figure 10.7: Sketch of a DOM model for an **event loop** which coordinates the *execution order* between functions.

10.5 Architectures for data and information exchange

There are only a few different ways of how components can exchange data: by accessing the same data structure in a shared memory partition, or by sensing the data structure over a communication channel. Both are complementary, because the latter mechanism eventually needs the former mechanism too: the communicated message is received by an operating system and that has to bring the data structure into the memory space of a component.

10.5.1 Activity = Block + Port, Interaction = Connector

The meta models for [activities](#) and [interaction channels](#) have already been introduced before, and they [conform to](#) the [Block-Port-Connector](#) meta model as follows:

- activity: provides the Port and Block parts of the BPC meta model.
- interaction channel: provides the Connector part of the BPC meta model.

(TODO: more details and examples.)

10.5.2 Symbolic indirection for composability *and* performance

(TODO: make the [mechanism](#) of generic architecture more concrete by adding LCSM and Configuration symbols to the “outside” facing parts of the component’s ports; mapping between symbolic IDs: the “inside” facing part translates the symbols presented at the “outside” in symbols that are compatible with the prots used inside the component. This is an instance of the [key-value](#) associative array or database, which applies to more [abstract data types](#) than just lists of keys and values.)

10.5.3 Policy: producer-consumer channel specialisations

The meta model of *channels*, or *streams* can be adapted to many application contexts, to optimize specific trade-offs in interactivity: performance, robustness, resource awareness, predictability, . . . This Section introduces a set of specialisations that fit well to important robotics use cases.

Heterogeneous data chunks

Streams interconnect producer and consumer activities with a permanent interaction channel. It is not uncommon that the contents of the information exchanged between both activities changes over time; often even from message to message. Therefore, the abstract data type of the data chunk in the stream can be different for every message. The stream meta model can still provide its ownership transfer and loose-coupling properties as follows: the data in its data chunks are not the “real” message data, but just pointers to whatever data structure corresponds to each message (Fig. 10.8).

10.5.4 Pattern: submission–completion for journaled CRUD interfaces

A common type of interaction between activities are the so-called **CRUD** operations (Create, Read, Update, Delete) on the [abstract data types](#) and [data structures](#) that represent models, events and data. The *mechanism* to support such CRUD interactions between asynchronous

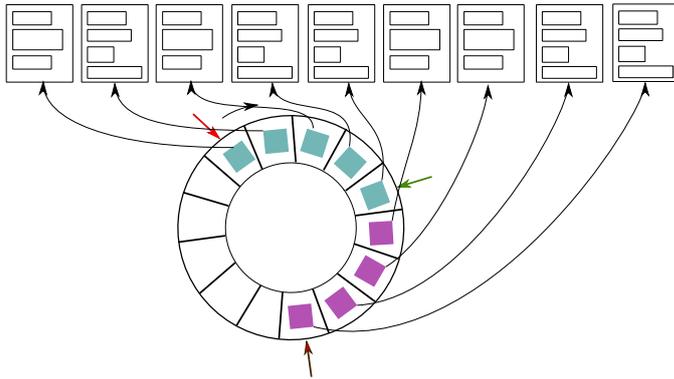


Figure 10.8: Stream with heterogeneous data chunks. The stream buffer is used only for efficient and effective ownership transformation of pointers to any type of data structure.

activities is the [Submission-Completion pattern](#), (Fig. 10.9) and more in particular an approach such as [write-ahead logging](#), in which the stream buffer acts as the “[transaction log](#)”, or the “[journal](#)”.

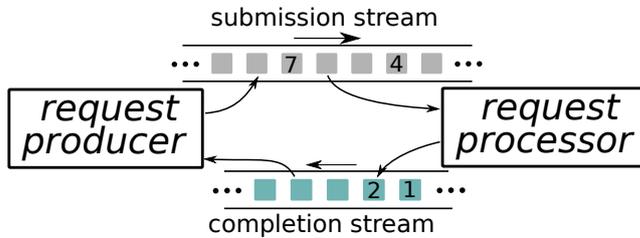


Figure 10.9: Submission-Completion streams as a mechanism to realise CRUD operations between producer and consumer.

This mechanism originates in data bases technology, and (hence) fits well to all “world model” aspects in robotic systems: control, perception and monitoring activities need to exchange information with the world model, and the latter must serve the needs of multiple interacting activities at the same time, while still guaranteeing a good trade-off between data consistency and latency in reacting to the requests. The trade-off can be steered by:

- **granularity** in the streams. Both in number of streams that one single activity can establish with the data base, and in magnitude of the transactions. For example, a robot can make one transaction to the world model per control cycle, or it only updates the world model after it has travelled through a complete “area” on the map.
- **task-level coordination protocols**. CRUD operations should be robust against an *identified* set of communication and computation disturbances to the system behaviour. Examples of disturbances are: delays in communication, non-availability of resources, or saturation of streams.

The system architects decide about the envisaged robustness: in many use cases, it makes little sense to design the trade-off to higher performance than needed in the application. For example, many robotics systems can tolerate the loss of large parts of a world model, because they have the possibility to (re-)execute tasks that can make the system recover from such a failure.

Despite the obvious observation that tasks, activities and CRUD operations are essential design primitives, they often remain overlooked as first-class design drivers, and very few *software frameworks* support task-level models. The reasons are manifold:

- it is difficult to express, explicitly, the knowledge relations that link platform resources on the one hand, and task capabilities, and performance and robustness requirements, on the other hand.

- system design spans several levels of abstraction and requires many scientific and engineering disciplines to be integrated. Hence, task-centric trade-offs put high demand on the knowledgeability of the system designers.
- many a software project starts *bottom-up*, around some algorithms and middleware that the developers can get “to work”; this often results in the proverbial “**law of the instrument**”: the instrument designers bring in (implicitly and unwillingly) artificial constraints on the applicability of their software, and hence it is they who put limits on system performance, not the system designers.
- and, often as a result of all of the above, the design of coordination protocols, that guarantee consistency under disturbances, is a tough design challenge.

Multiple producers, multiple consumers

In many applications, each activity must interact with multiple other activities, and often share the same information between these multiple activities. The easiest way to make an architecture for this situation is to introduce information streams that have more than one producer and/or more than one consumer (Fig. 10.10).

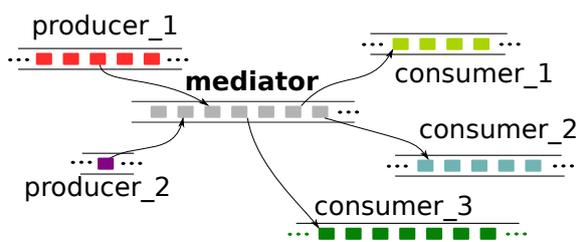


Figure 10.10: Single stream with multiple producers and consumers. Such an architecture requires explicit coordination between all producers, and another explicit coordination between all consumers, because *ownership* of each data chunk in each shared stream is not clear.

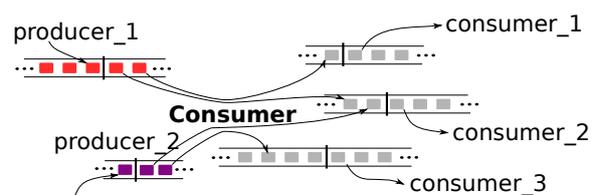


Figure 10.11: An equivalent architecture to Fig. 10.10 (*only* for the left-hand side of that Figure, covering the sharing of the “grey” stream): **one** intermediate **Consumer** activity is introduced in the architecture, and that activity is the **single** reader on the multiple producer streams, an also the **single** producer on each of the consumer streams.

This approach introduces several **data chunk ownership complications**:

- how to decide which producer is allowed to update which data chunk in the stream.
- when is the ownership of each chunk transferred.
- how to decide which consumers get access to the same data chunk.

Figure 10.11 sketches a more **explainable and composable** design. The price to pay is:

- to introduce an **extra consumer** activity, whose sole purpose is to be the **only consumer** on each of the producer streams, and the **only producer** on each of the consumer streams.

This solves, both, the ownership and access coordination problems.

- to introduce an extra copy operation for each data chunk.

Indeed, the “green” coloured consumer streams on the right-hand side of Fig. 10.10 are not directly produced from the grey “shared” stream in the middle, but first the “local” grey streams in Fig. 10.11 are filled for each consumer before that latter one can process those data chunks to produce its own “green” stream.

The price to pay is smaller than it seems at first sight:

- the relative cost of adding extra memory gets closer to zero the more complicated the application becomes, and the latter context is exactly the one the suggested approach provides a solution for.
- on modern hardware, the introduction of the intermediate consumer can result in efficiency *gains*: it reduces the problem of **cache misses** because the data storage is more local to each activity.
- the **head-of-line blocking** problem is avoided, because the suggested policy implements a **virtual output queuing** technique.

10.5.5 Pattern: world model as blackboard

World models are *designed* to decouple all “internal” activities in tasks, so the most deterministic way to integrate several tasks is by sharing and coordinating selected parts of the “composite” and “component” world models. One end of the sharing spectrum is realised by a **blackboard architecture**: all activities that have to share world model information, read and write it on the “**same “map”**”, so activities can see everything from each other. The other end of the sharing spectrum has no shared world model at all: all activities have their own internal world model, and exchange world model information via **communication**, one-on-one and on a *need to know* basis. The “forces” that determine where to position an application in this spectrum are: communication cost; model consistency; robustness against loss of interaction; specialisation of component functionalities; intellectual property rights; etc.

10.5.6 Pattern: world model as semantic database

For any somewhat realistic application, the **tworld model** contains several hundreds to several thousands of entities, relations and constraints, with an order of magnitude more *parameters*. It is appropriate to call this composition a “**semantic database**” (Sec. 1.2); the added value with respect to a normal database is:

- the parameters in the data are linked together with relations that have semantic meaning, with several **higher-order** relations and constraints, via the many interconnected levels of abstraction and types of information.
- **queries** on the database can use semantic terms reflecting the **intention**, **causality** or **dependency** that holds between query arguments; of course, *if* these higher-order relations have been added to the models. Hence, one can get explanations about *why* the query yields the results it does, or about *the context* in which the answer holds.
- the provided knowledge links can be exploited in the computation of the **query answer**, because **graph traversal** becomes possible, instead of the more general *graph matching*. The latter is the default “solver” for relational database, while the former becomes more and more mainstream in graph databases.

In the context of robotics systems, this means that the **coupling** between **control** and **perception** can be adapted to the **context** provided by the *Task-Skill-Resource* paradigm:

- the expected capabilities.
- the available resources.
- the past, actual, and expected state of the environment.

For example, a robot controller can switch its perception algorithms depending on the knowledge it has about which features in the environment fit best to, both, the available sensors and

sensor processing software, and the required feedback and monitoring in the motion control loops. More concretely, when driving through a corridor in a hospital or office building, the robot can actively search for the *semantic tags* that have been put in the building with the explicit purpose of guiding its users towards the various destinations. A similar situation holds for outdoor navigation (car and truck driving, or plane and helicopter take-off and landing) where traffic signs and signalling are put up to increase the **freedom from choice** of the traffic participants, and hence their “cognitive load” connected to taking part in the traffic.

10.5.7 Best practices in activity and channel design

This Section introduces and motivates some best practices in the design of activities interfaced with channels.

Requests instead of Commands

If activities behave as if they are master of any interaction they are involved in, they typically interpret a “request” as a “command”.

Activities interact, because they need to influence each other’s behaviour, like letting one activity do something for another activity. Developers who consider every such interaction as a friendly request, think more quickly about introducing *dialogues* between the activities. A dialogue makes it easier for peer activities to adapt to their mutual expectation and performance, at runtime, all the time, and after a shared decision making of what is a “good” trade-off between both peers’ objectives.

One LCSM per activity

Each activity has its own “life”, independent of any other activity, and that “life” must be a **singleton**. Hence, there is one and only one **Life Cycle State Machine** in each activity. Only via a full LCSM, an activity can reconfigure, at runtime, the resources it owns. (Of course, that is only true *if* that resource has the functionality of runtime reconfiguration.)

Activities are not just essential as information architectural entities to represent “life” of their own, but also to give “life” to the system: they interact with each other to realise a system’s desired behaviour. Two necessary (but not sufficient) conditions to make sure that such interactions produce **predictable behaviour** are that (i) each of the interacting Activities is itself in its own internal state that is designed to support the interaction, and (ii) it can then communicate explicitly about the interaction with the other Activities it is interacting with.

(TODO: example: *EtherCat* communication can not be used in real-time before appropriate **handshake** is performed, and the master as well as all slaves go to the same state in the standardized EtherCat protocol state machine.)

Every entity and relation has one owner

It is the owner who decides about the policies on which CRUD operations are allowed to act on the data of the entity or the relation. Ambiguous ownership leads to the wrong activities configuring resources, at the wrong moment.

(TODO: examples: actuator or sensor; task; state of a solver; resource allocation; etc.)

Explicit causality in data access policy

Activities must interact with each other to realise tasks. And each task brings (models of) causal dependencies between task functions operating on task data: a particular function should only operate on a particular abstract data type when certain conditions are met. Such causal relationships must be modelled explicitly, by relations representing so-called *data access policy* constraints.

For example, the *real-time* thread in a dual-thread activity is the one who does the *writing* from the source of the data to any shared data structure, because it is the creator, and hence the owner, of that data. When the *non-real-time thread* would do the copying, the data transfer could be interrupted by preemption of that thread.

(TODO: more examples.)

Every task model is a shared resource

The **model of a task** is a property graph, that links information in the task's control, perception, plan and monitoring parts together with information of how the world looks like, at any moment in the task's lifetime. That means, the runtime version of the task model represents **state** in the system. State information is only useful if it is *shared*, but updating state information in a distributed execution context implies a risk of making the information inconsistent. Hence, it is the information architects' major responsibility to think thoroughly about which activities are allowed to operate on which parts of the runtime task model, and how various "competing" operations are coordinated inside and between activities. In other words, the task model must be considered as a *shared resource*, and all relevant design patterns must be applied. The start of this design is to identify (explicitly and formally) what are the *information access dependencies* in the system. Such dependencies often go further than coordinating each operation on each shared part of the task model, because there is often also "state" in *sequences* of the CRUD operations themselves: some sequences must be applied **atomically**, or not at all; some sequences can be interleaved ("pre-empted") by specific other sequences; etc..

The **execution** of a task consumes **platform resources** in often very indirect and hidden ways, via control and perception activities, and it is almost impossible to deploy different tasks in a system without causing interference at the resource usage level.

10.6 Mediator architectures for interacting activities

10.6.1 Mechanism: execution of event handling in an event loop

The events are the coupling of an *activity's* behaviour with the behavioural context of the whole *system*, via the data structures of an **FSM**. A system's **software architecture** needs the **computational mechanism** of the **event loop**, to turn the above-described abstract data types into actions. The common terminology of this computation is **event handling**. Its meta model is a special case of **stream handling**, with the events forming the **task queue** and the coordinating activity playing the **worker**. So, the computational entities and relations, that

add to the previously introduced structural and behavioural entities and relations of the FSM meta model, are:

- **event_queue**: the *structural relation* that represents a **stream buffer** of **events** that the FSM has received and must still process, or that the FSM has fired and must still send out.
- **event_processing**: these are activities that *produce* and *consume* the event stream buffers, to realise the *behaviourial relation* between the input **event_queue** stream and (i) the output **event_queue** stream, and (ii) the **transitions** stream. **Boolean algebra** is the mathematics of the algorithm that *computes* the behavioural relation.

The reactive nature of the FSM implies that an event data structure in the event stream is deleted, after that event has been processed. (Similarly, a Petri Net removes **tokens** from **places** when a **transition** fires; but the algorithm flags that are connected to the texttttokens are not removed: only their values are changed by a firing Petri Net.)

- **event_monitoring**: **event_processing** is a first-class activity, so it is obvious that it can receive its own set of monitors. Typical monitor functions are:
 - detecting whether the FSM is “running in cycles”. For example, the same nominal task execution plan is always interrupted by the same non-nominal situation, and the same error recovery plan that leads to the same task FSM state that the task plan execution started from.
 - event tracing: an event can be fired each time a *particular sequence* (“trace”) of events has occurred. For example, **activity_1** sends an event, after **activity_2** and **activity_2** have fired an event.

This monitoring is one of the *causes* that fires, from the inside, events to which the FSM can react to.

The computations needed for event queueing, processing and monitoring are realised by an **event_loop** that “wakes up” at configured moments in time to execute the computations. This execution requires **computational state**⁷ of its own, which adds extra parameters to the FSM meta model:

- *life cycle* parameters:
 - **current_state**: the ID of the current state of the FSM.
 - **initial_state**: the FSM **state** in which the FSM starts after its event processing activity has transitioned from its **deploying** state to the **active** state (Sec. 2.10.8).
 - **final_state(s)**: one or more FSM **states** in which the FSM ends its **active** state and transitions back to its **deploying** state.
 - **state_history**: the **stream** of state IDs that the FSM execution has moved through.
- extensions to the externally visible **event reaction table**:

⁷Note the fundamental differences between “state” as a representation of type of behaviour, and “state” as the values of the parameters in the algorithms that implement the behaviour. It is a pity that no widely accepted nomenclature exists to separate these two meanings of the word “state”.

- **onEntry**, **onExit**: these events are added to the table for each **transition**, so execution behaviour can be triggered every time the **state** at the start of the **transition** is *exited* and the **state** at the end of the **transition** is *entered*. It is a best practice to use two events, because this allows the behaviour to be “owned” by the separate **states**, and not by the **transition**.
- **onTrigger**: the **event_loop** triggers the **current_state** at regular intervals in time, and execution behaviour can be connected to such triggers by adding a **transition** from that **state** to itself.
- a list of **callbacks**: a callback is a function that the **event_loop** executes in lieu of the *coordinated* activities; the latter have *configured* their callbacks first, by **registering** these functions as clients for the **event_processing**, together with the event reaction table that represents the events to which these callbacks will react.

A typical usage is to realise reconfiguration of an activity, required by a state change in the FSM.

10.6.2 Mechanism: ownership, loose coupling, semantics

Below are generically relevant **cross-cutting responsibilities** to all above-mentioned **design decisions**, that have to land somewhere in activities in the information architecture:

1. **ownership**: the decision about which activity owns the information in the system, not in the least the information to access the resources in the hardware (often via the operating system). Complemented by the decision under which conditions ownership can (or should) be transferred to other activities; often only for limited periods in time.
2. **loose coupling** of data, solvers and activities: let the activities *observe* the behaviour of each other during their interactions, and let them react to it instead of hard-coding that interaction upfront.
3. **semantic meaning** of the information: it is the mediator activity that links the architectural aprts to documentation and offline/online tooling, to guarantee that information is interpreted in an unambiguously correct way, by human developers as well as by the mediated activities.

10.6.3 Mechanism: activities exchange data via channels

The **activity** and the **channel** are the **mechanisms** (“components”) with which system architects realise **tasks**. The one-line description of the **behaviour** of activities and channels is that, respectively, they *process* and *exchange data*.⁸

A **task model** is the **causal** trigger of any system design:⁹ it provides the **purpose** (or, **intention**) of the system’s behaviour, by representing **what** has to be realised by the robot. A **skill model** represents the knowledge about **how** to realise a task, hence it adds constraints to that task model. A task and a skill are *not* activities in themselves, but they constrain the

⁸In the terminology of the **block-port-connector** meta meta model, activities are the **blocks**, channels are the **connectors**, and both interface data via **ports**.

⁹Often, hardware architecture decisions bring in hard causalities too. For example, via the protocols of network communications.

behaviour of the activities and channels that **execute** a task's **plans** to guide them to the “best possible” outcome in the “most robustly possible” way.

The design process to represent the task requirements explicitly, results (almost by definition) in a **plan** of *how* to realise the task. Possibly, the resulting plan is only partial, and/or too abstract to be composed with a skill without any further design efforts.

Internally, an activity composes the following building blocks:

- one **Life Cycle State Machine** (LCSM), to *coordinate* its outwards-facing behaviour.
- one or more **algorithms**, to realise its internal behaviour.
- zero or more **Petri Nets**, to *coordinate* these algorithms.

Externally, an activity needs to be composed with

- one **event loop**¹⁰ in a thread, to trigger the computations in its algorithms, FSM and Petri Net(s).
- one or more **exchange patterns**, to manage the (shared) data that its algorithms produce and/or consume.

When an activity switches state in its LCSM, it typically

- executes a different set of algorithms.
- interfaces with a different set of channels than the ones it had been interfacing with in the current LCSM state.
- uses a different set of configuration parameters for all of the above.

More concretely, for each state of the LCSM, the activity model provides a **concrete (model for an) implementation** of the 4C's function types:

- **communicate()**: realises the loose coupling with channels, in that it (i) implements the *asynchronous* **produce()** and **consume()** functions for all of the channels that must be interfaced in the current LCSM state of the activity, and (ii) adapts the *port* configuration through which the algorithms inside the activity access these data structures *synchronously*.
- **configure()**: realises the changes in some **magic numbers** in all other function types, whenever a change in LCSM state requires such a reconfiguration.
- **coordinate()**: this comes in two flavours:
 - **coordinate(&LCSM)**: decides upon *externally visible* behavioural state changes, via the “state data” of the LCSM.
 - **coordinate(&PetriNet)**: synchronises *internally* the execution of algorithms, via the “state data” of a Petri Net.

The LCSM and Petri Nets are not fully decoupled: some changes in Petri Net flags should result in LCSM events, and the other way around. Such *conversion functions* are part of the “shared level” of all coordination functions.

- **compute()**: executes the functions required for the current activity behaviour, and driven by its current data state.

¹⁰The event loop is owned by, and deployed in, a “**thread**”, which is a mechanism of the *software architecture*, not the information architecture (which is the subject of this Chapter).

The particular **composition** of all of the above 4C's functions makes up for the *fifth* part of the 5C computational paradigm for behaviour. The 5C composition level also adds the `communicate()` functions that connect some of the *interaction channels* to outward-facing *Inter-Process Communication* channels.

10.6.4 Policy: discovering and connecting peers

(TODO: ICE, signalling, SIP, SDP, **perfect negotiation**, etc.)

Interactive dialogue — Ownership synchronisation

Figure 10.12 depicts how the stream buffer mechanism can be used to realise a **dialogue** between two activities. The figure shows an ordered sequence of six moments in time (t_1 to t_6), and at each of these moments, one and only one of the two activities (the “green” one, or the “purple” one) is owner of the data chunk that both activities share. Hence, the owning activity:

- reads what the other activity has provided as an update to that the data chunk the previous time it had ownership.
- interprets that update and implement its own response update.
- transfers ownership back to the other activity.

The stream's buffer is just “one deep”, in this case. The buffer need not be used to store the “message” data, but it must store the “pointer” to that structure, because it is the ownership to that pointer that is exchanged between both activities. The pointer value itself can change everytime the ownership is changed; hence:

- there is a large flexibility in what data structure is used in the dialogue.
- the same **mechanism** of “one-deep streams” can be used for all situations where the ownership of any kind of **resource** must be managed in a deterministic and efficient way. More in particular, it is an alternative for **mutexes** and other locking mechanisms.

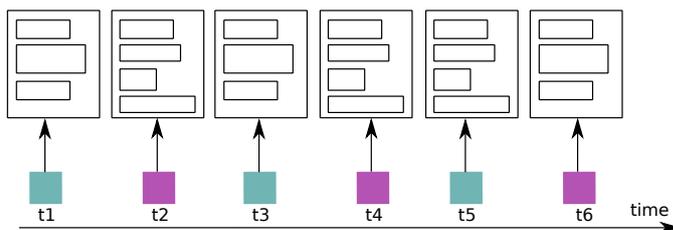


Figure 10.12: Stream support for a *dialogue* between two activities, each one updating the same “message” data chunk in turn: the stream mechanism can guarantee deterministic transfer of *ownership* back and forth between the activities.

10.6.5 Mediator architecture for Task-Skill-Resource

(TODO: explain information architectures that **conform** to the **Task-Skill-Resource** meta model (Fig 5.2 repeated [here](#) for convenience). The architecture must provide explainable trade-offs to support the following relations:

- the **specification** of a task.
- the **decisions** being made in the **activities** of **perception** and **control**: they must deliver the expected outcome, **satisficing** the specified **quality**, **robustness**, and **situational awareness**, [34].

- the **bounded rationality** [91, 98] **costs** incurred by using the available **resources**, including **non-monetary utilities** such as **safety** and **explainability**.

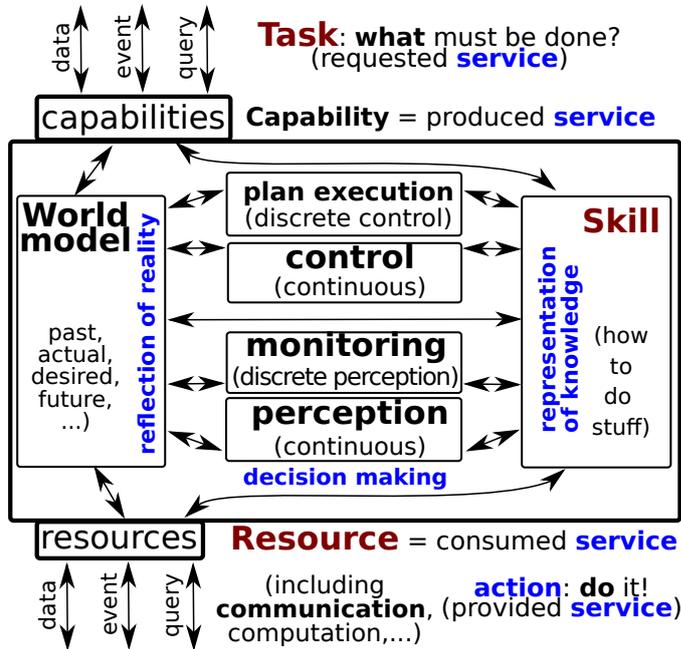


Figure 10.13: Copy of Fig. 5.2, representing the Task-Skill-Resource meta model.

10.6.6 Task queue mediator activity

10.6.7 Task queue with worker pool

10.6.8 Coordination and configuration of architectures

Pattern: composition of flags, Petri Nets and FSMs

(TODO: give examples from the “Running example” context, where this kind of composition has proven advantages in the trade-off between coupling for efficiency, and decoupling for composability. Several hypotheses in the world model are updated concurrently, until one or more of them “win” and the others can be stopped.)

Pattern: configuration – monitoring – decision making – reacting

(TODO: this problem occurs very often, and the solution has a generic architecture. A “mediator” configures a number of activities, with monitors to detect when the logical conditions become true that are needed in a decision making, and after which the decision is available to the activities to react to. The (multiple) monitors, the (singleton) decision making, and the (multiple) reactions are all configured together, with one single “Task” purpose.)

10.7 Architectures for concurrent algorithms with configuration and coordination

Many functionalities in robotics systems are provided by algorithms that need to coordinate their internal behaviour:

- *resource configuration:*
- *functionality configuration:*

10.7.1 Pattern: splitting a solver into scheduling and dispatching

Many solvers, such as **quadratic programming** or **kinematic and dynamic solvers**, consist of multiple **stages**, or sub-algorithms. For example, first converting the problem formulation to a canonical form, then finding a feasible solution, and finally iterating towards the optimum. Nevertheless, their *implementations* most often *run to completion* without allowing to be preempted between the different stages. Only few solvers are designed with such *stop anytime* execution behaviour in mind. The latter ones are a natural fit for architectures based on **task queues and event loops**. The pattern that has emerged in this context splits the solver in two complementary parts:

- **scheduler**: the computations that decide which parts of the total algorithm *are ready to be executed*. The outcome of the scheduler is that this list is added to the *task queue* of the activity in which the solver is deployed.
- **dispatcher**: the computations that *actually execute* a task on the task queue.

This pattern allows to compose the execution on one event loop of several algorithms from several programmes. The added value is that:

- whenever there is something to be executed in a complex activity, it can be executed as soon as the activity gets assigned to a CPU.
- the dependencies between several algorithms (that is, one algorithm needs a partial result of another algorithm) can be optimized by a **mediator** activity that has *knowledge* about these dependencies.

A robotics context in which this pattern is highly effective is that of the integration of **sensor fusion**, **moving-horizon estimation** and **model-predictive control**: all of these parts require information about the kinematic and dynamic state of the robot to which the sensors are attached and for which the controller is working, but a “run to completion” design of the algorithms results in the same kinematic and dynamic computations to be repeated several times.

10.7.2 Collaborative pre-emption in iterative solvers

(TODO: linked to the **scheduler-dispatcher** pattern in information architectures. make 4Cs of the solver such that they allow “to break” the event loop after every iteration (or a coherent part of it), and schedule progress monitoring for the solver.)

10.7.3 Caching and memoization in solvers

(TODO: explain trade-off between redoing computations and storing intermediate computational results: **memoization**.)

10.7.4 Cascaded control loops

10.7.5 Adaptive control loops

10.7.6 Tasks as hybrid constrained optimization algorithms

Representing a **robotic task** as a **hybrid constrained-optimization problem** yields a **declarative** specification, that is, it is a model (generated off-line or online) that expresses the logic of the (desired) task execution without describing its **control flow**. The **solver** generates (at runtime, taking the latest sensor information into account) the **imperative** (or, “**procedural**”) flow of control actions (actual setpoints in joint position, velocities, accelerations or torques to send to the actuators) or plans (more detailed declarative task models, with a more focused scope in time and space), required to realise the task. The above holds both for control-based approaches [3, 70] (online, reactive, but maybe suffering from local minima problems) and for plan-based solvers [65] (typically, but not necessarily, offline, less reactive, but exploring a bigger search space).

The term “task” was used in the paragraphs above as a container term for each of its parts: plan, control, monitoring, perception, capabilities and resources. Often, the configuration spaces of two or more of them are taken together. A common example appears when specifying **active perception** tasks: the plan contains motions that have as sole purpose to improve the perception and monitoring aspects of a task. This is what humans do, often unconsciously, for example when double checking their location in an environment, they focus their attention to a series of important landmark, in a particular order and with a frequency of revisiting the relevant landmarks, that depends on the tolerances required for that localisation.

Mechanism: relation, constraint, dependency graph, spanning tree, action, solver sweep

A typical solver has the following **mereo-topological** entities :

- **domain**: the set of all “states” of the problem under study.
- **optimization-relation** and **constraint**: these relations represent which state combinations are, respectively, desired or not allowed.
- **dependency-graph**: the graph that represents dependencies between several **optimization-relations** and **constraints**. For example, there can be an *order* in inequality constraints, or an hierarchy in optimization functions.
- **spanning-tree**: one particular way of ordering the dependencies in a tree structure. Most solvers spend time on creating such a tree structure, to have an efficient way of structuring their computations.
- **action**: any combination of allowed data and function on the **domain**. The goal of the solver is to find a control flow on such **actions** that brings the system from an initial state to a state that satisfies the **optimization-relations** and **constraints**.
- **solver-sweep**: a solver algorithm typically “sweeps” one or more times over the **spanning-tree**, where:
 - each node crossing corresponds to the scheduling of a particular **action**.

- a termination condition is checked to stop the solver as soon as the constructed set of actions (that is, the resulting imperative algorithm) has reached a “good enough” solution.

Policy: dynamic programming

Dynamic programming is, often, the most difficult algorithm design pattern, since it *exploits* the knowledge about which intermediately computed data structures should be given the status of “state”, because they will be reused at a later time. Obviously, that knowledge is very domain and application dependent, so few generic insights exist. With one major exception: the physical world satisfies many *conservation principles* (e.g., for energy or momentum), so any intermediate result that represents a conserved property is a natural candidate to become a status variable in the algorithm.

The design *forces* are: to maximize runtime adaptability, in dependencies between data, functions and schedules.

This is a broad family of “declaratively specified” algorithms (or *solvers*, Sec. 5.5), and hence they come close to exploiting all features of the presented algorithmic meta model.

Policy: static spanning tree pyramid

(TODO: all memory statically allocated, functional dependencies modelled as spanning trees, schedules modelled as spanning tree over spanning trees; configuring, preparing, dispatching;)

Policy: feasible and optimal solutions

Any dependency relation can be “hard” or “soft”, specifying whether one desires to find feasible or optimal solutions:

- **feasible:** each constraint relation must be satisfied.
- **optimal:** the *deviation* from the constraint relation is optimized according to a *cost function*.

Satisfying versus **optimizing** solvers. The former form of solver uses the iterations towards the most optimal solution, until the current intermediate solution is already adequate (“good enough”). That is, within a specified, set of the constraints. The latter form of solver only returns a solution when this solution is the optimal one. Of course, both solver types should also stop as soon as they find out that the solution can not be computed, for some reason.

Policy: sweep scheduling

The schedule of the solver’s computations can be determined statically or dynamically:

- **static:** the spanning tree is computed once, and deployed as a static dispatch data structure.
- **dynamic:** the spanning tree can be (re)computed at runtime, allowing for dynamic reconfigurations of the solver specification.

Policy: tolerances

- numerical accuracy of the constraint satisfaction;
- number of iterations to find solution.

10.7.7 Control of kinematic chains

(TODO: scope of this Section.)

10.7.8 Policy: deployment in an event loop

The discussions around Algorithm 1 in Sec. 4.8 have made it clear that the computation of the hybrid dynamics solver, and any of its many variations, are just instances of an **event loops**, with (maximum) three iterations. As illustrated by later Sections, also other computations can be *composed* into the same event loop, extending the iterations over longer time horizons, e.g., for sensor fusion, control, world model updating, and/or task monitoring.

(TODO: explain which event loop schedules correspond to: prioritization between inputs, optimal control, guaranteeing joint limits or Cartesian force limits)

10.7.9 Composition with dynamics of resources

(TODO: how to compose which parts of the hybrid dynamics solver for a kinematic chain with the dynamics of actuators and energy resources, if the latter also come with a Task description in the form of a constrained optimization problem.)

10.7.10 Composition with dynamics of perception and world model

Most kinematics and dynamics solver libraries offer programming interfaces that are only solving the kinematic and dynamic state entities (hence, they use joint space and Cartesian space entities as inputs and outputs), and have a *fixed control flow* that is optimized for this purpose. However, many robotics applications need to “fuse” the motion computations with computations for perception, control, planning and world modelling, Fig. 6.

For example, a visual servoing task requires a perception algorithm that has access to the instantaneous pose and velocity of the camera, when that is attached somewhere on the kinematic chain, so that the motion inputs can be used in the control flow of the vision processing *and vice versa*. The good news is that the presented models for kinematic chains and their algorithmic solvers have a high level of decoupling between models and control flows, via the “sweeps”, in which compositions with other types of solvers (control, estimation, etc.) can be composed. So, a major composition modelling and tooling that still needs to be developed, is that to compose algorithmic work flows.

10.7.11 Composition with motion trajectories

(TODO: what to add to the instantaneous motion tasks in order to be solve for non-instantaneous motions, such as trajectories and paths; special traditional cases of linear and circular arc trajectories.)

10.8 Architectures for proprio-, extero- and cartho-ceptive control

This Section introduces reference information architectures for the three main control “levels” in robotics systems. The emphasis is on the similarities in the architectural *structure* of components, and less on the (obvious) differences in the *behaviour* of the components.

10.8.1 Proprio-ceptive architectures

(TODO:)

10.8.2 Extero-ceptive architectures

(TODO:)

10.8.3 Cartho-ceptive architecture: robot, edge/fog, and cloud

Robotic systems of some realistic complexity are **distributed** over several robots, networks and computers. Inevitably, those systems’ behaviour and interactions are also distributed over *space* and *time*. Hence, their architecture must include **cartho-ceptive** control, too. The top-level architecture of such distributed systems is more generically applicable than to robotics systems only, and the following **mereo-topology** can be borrowed from **cloud computing**:

- on-board: internally, the robot architecture must share realtime data and action coordination between its components that provide access to its resources, and that execute its tasks.

So, internally, there is a world model (or, “chart”, or “map”) to extend the **proprio-ceptive** and **extero-ceptive** control of *one single robot* to an environment that it has to share with other devices. In other words, the single robot must add **situational awareness** to its own control.

- **fog** or **edge**: the architecture supports peer-to-peer and *instant* data sharing and action coordination between:
 - a *set of robots* which have to be aware of each others’ presence and actions.
 - one or more *edge holons*, that help the robots in their control tasks.

The edge devices can be in the infrastructure (e.g., a traffic control tower for drones; a lock controller for autonomous boats; or a traffic light controller on a crossroad that is used by mobile robots), or can be provided by dedicated third-party service providers (e.g., companies specialised in dynamic maps and charts, or in ICT services).

The world model serves to coordinate the **cartho-ceptive** control of one or more robots; not in the least by having one peer holon providing “local” world model services.

- **cloud**: *big* data and client-server interactions with individual robots (or their “edge holons”), without providing services to instant and direct robot-to-robot interactions.

Depending on the purpose of the cloud in the robotic system, the world model can serve any of the above-mentioned control levels. The most obvious, “big data”, one is that of providing and updating maps to all robots in the system, and/or to collect data from

all robots to provide application services to third parties. For example, meteorological data gathered from autonomous measurement robots on the oceans or in the air. But other more “instant data” use cases are viable too. For example, off-board processing of data and computing intensive jobs, like [point cloud registrations](#).

The 5C aspects of the architectures are as follows:

- *communication*: one communication access point per peer, that is responsible for discovery and session initiation; it can decide to set up dedicated streams between some of its internal components with external peers.
- *coordination*: coordinated, orchestrated or choreographed, depending on the required or allowed autonomy of the decision making in each peer. Mixed designs are possible, for example to let each peer decide for itself, distribute its intentions in the edge, so that others can react when they expect that a potential conflict or error can occur. It may make sense to foresee an arbitration in the architecture, to delegate the decision making to a “third party”, possibly with appropriate legal authority.
- *configuration*: the edge device can decide to switch on or off some shared decision making settings for all participating peers. For example, different safety zone distances for clear skies, foggy conditions, or night; or different priority rules in some zones because some construction or maintenance operations are executed there.

The data and information model decisions that have to be made in the architectures are as follows, and are very application domain specific:

- *base map*. These are typically *metric* maps, that is with numerical coordinates for all landmarks in the map. Landmarks most often are static over “longer” time intervals. For example, [nautical chart](#), such as the [ECDIS](#) standard. Or street maps (e.g., from [OpenStreetMap](#)) or [indoor](#) maps (e.g., from or [IndoorGML](#) from the [Open Geospatial Consortium](#)). Or warehouse and assembly line layouts. And even assembly graphs generated by a [Computer-Aided Manufacturing](#) system.
- *semantic tags and layers*: any application brings in application-specific extra landmarks, or “tags”, on the base map, and even several layers of such tags, each providing different “semantic contexts” to the robots.

Some examples are semantic layers of traffic signs and markers added to road maps; the dynamic version of this would show the actual traffic, possible with information about the status of each robot on the map and its intended actions for the near future. The [Middle Size League](#) of the RoboCup competition is an example where very dynamic maps are needed to allow a team of robots to play a game of soccer.

- *perception relations* with semantic tags: every robot must be able to connect the raw data from its sensor to the relevant “tags” on the map.

Figure 10.14 shows an example of a base map with semantic tags in a perception layer. The tag is the “symbolic pointer” to more information about how the objects where these tags are attached to, show up in the sensor data of a particular sensing technology.

- *control relations* with semantic tags: each robot must be able to link the actual status of the map to the decisions it makes about its actions.

The semantic tags in Fig. 10.14 can also provide links to the information about the maneuvers a robot boat is allowed to perform in the corresponding areas.

- *differences* between models and *model updating*: in dynamic open worlds, no map is ever complete or consistent, so peers can provide suggestions to improve the current contents of the map.



Figure 10.14: Base map of the city harbour of Leuven, with a perception layer of four semantic tags: the bridge pillars, the quay walls, the harbour master office, and the mooring area for boats.

10.8.4 Situational aware architecture in the “edge”

This Section introduces a **reference information architecture** for the cartho-ceptive control of a varying number of peer robots in an **edge** composition, where *situational awareness* is a shared common task. The presented reference architecture is rather “empty”, in that the **largest part of the concrete design** consists of **application-specific** data structures and coordination protocols. To make the architecture somewhat tangible, the “edge” is considered to consist of a city harbour as in Fig. 10.14, with a couple of vessels as robotic peers, and the harbour master as the local navigation authority peer. The latter is also in charge of the opening and closing of the bridge over the harbour, which will be a “task” that the vessels can ask it to execute at their behalf. Although explained in the application context of “autonomous shipping”, the design of the reference architecture is very similar to other robotic contexts, such as logistic AGVs in a warehouse, or multiple robot arms sharing an assembly cell.

Base map: this is as depicted in the [Figure](#). The list of semantic tag layers that are available to the vessels is to be standardized by the inland waterways authorities; not just the list, but also the information models that can be accessed through the tags. All vessels are obliged to have a particular version of the standardized map on board, in their own extero-ceptive navigation pilot.

Communication: the vessels are obliged to use the communication channels and messaging protocols prescribed by the waterways authorities. This involves several steps, of discovery, session initiation, authentication, heartbeating, decision logging, etc. Each of these steps could require the use of different channels and protocols. Two major responsibilities for the communication are (i) to support each peer in the edge network to update its dynamic map with information from the other peers, and (ii) to enable the coordination of the shared decision making needed to let each vessel execute its intended journey, while sharing the area with other vessels, in an explainable and legally compliant way.

Situational aware perception: each vessel extracts from its own internal extero-ceptive control activity the subset of landmarks, vessels and “obstacles” that it recognizes; Fig. 10.15 sketches what that could mean in the city harbour case. It communicates that information in

its edge network, as its **dynamic annotated map**. It also obtains similar dynamic annotated maps from the other peers. So, it can improve its own assessment of its environment, and assess what the other vessels in the neighbourhood are aware of themselves. The quality of each vessel's dynamic map, and hence the self-localisation of that vessel on the base map, can be scored by the number of perceivable tags on the base map that have actually been recognized.



Figure 10.15: A sketch of what the dynamic annotated map for a vessel looks like. The vessel has recognized four of the tagged landmarks on the base map; this recognition is represented by the big V. The harbour master office is sensed via its *4G communication*. The two bridge pillars are both recognized in its *Lidar data*. The recognized part of the quay wall (also from *Lidar data*) is indicated in bold red lines. *Some* occupancy of the boat mooring area has been recognized, but no individual boats or objects could be identified in the *Lidar data*.

Situational aware task execution: on top of the situation awareness provided by the above-mentioned dynamic map, each vessel:

- decides about its own navigation task.
- makes a “trajectory” of planned maneuvers in the near future available on the dynamic map.
- adds its own *control limit zones* around these planned maneuvers, that is, two zones around the hull of the vessel in which it (i) can still avoid collision when another vessel or obstacle would enter that zone, and (ii) can only react with an emergency stop, without guarantee for collision avoidance.

This input allows for a **preview control** approach: the task decisions of each vessel can be taken with information about the *intended future* maneuvers of other vessels, Fig. 10.16. Of course, this concept requires the standardized coordination of all vessels' individual tasks. For example, via **distributed consensus** coordination protocols, such as:

- *delegate MAS* [53]: peers interact individually with a selected number of other peers in the edge network, and take full responsibility for the decision making.
- **promises**: robots decide to cooperate voluntarily in their mutual action coordination by publishing their intentions (“promises”) to each other.
- *mediator*: the peers communicate their intentions to a (unique and agreed upon) peer (for example, the local harbour master), that has the authority to make decisions in name of the involved peers.

Both approaches can use similar methods, such as **auctions**.



Figure 10.16: The red and the green vessel communicate their intended maneuvers for the four next periods of 30 seconds. This allows all peers in the edge network to *preview* the change in the actual situation, and to engage in arbitration protocols when needed. One of the preview locations of the green vessel is annotated with a green and an orange zone, indicating the zones in which the vessel’s own navigation control can, respectively, can not, avoid collision when another vessel or obstacle enters that zone.

In summary, the major message of this Section is that:

- the reference architecture of a situational aware “edge” network of peers is very **structured and conceptually simple**.
- it can only work reliably, predictably and with clear liability, if **dozens** of application-specific protocols and data structures are **standardized and compulsory** for all peers.

10.9 Common architectures for robotic task execution

10.9.1 Task-Skill-Resource

(TODO: identify holons and mediation; lazy concurrent solver algorithms; submission-completion streams.)

10.9.2 Sense-Plan-Act, Three-Layered, Behaviour, Subsumption

(TODO: Three-Layer architecture [5]; TLA, three-tier; decisional/deliberation = planning (symbolic control), execution = decision (discrete control), reaction/functional = (feedback control.) Identification of too strong couplings.)

10.9.3 Task-Skill-Resource versus Sense-Plan-Act

This document advocates the *Task-Skill-Resource* paradigm, as a replacement of the mainstream **Sense-Plan-Act** paradigm. The former let systems *start* any Task execution with a *plan of guarded motions*, and with an explicit *world model* (“**map**”) that contains **geometrical** as well as **symbolic** tags. Both are pieces of *knowledge* one can expect to be available, and that one can expect “agents” to interpret correctly in the context of the full map. The

system then uses its sensors that provide data (i) to guard how well the plan can be realised, (ii) to suggest updates to the world model, (iii) to associate the symbolic tags with real-world geometric entities, (iv) to adapt the current plan, and (v) to decide to look for a better plan in the Task's plan repository.

The Sense-Plan-Act paradigm, on the contrary, expects systems *to generate* their plans themselves; most often in the form of nothing more than a *trajectory*, because trajectory control is the only control action that can be realised in a sensor data-only context. Indeed, it is only when the *context* of the motion is available, that the context-dependent knowledge of a Skill can be introduced. In other words, the shortcoming of SPA is *not* in that the robot can not *react* to changes in the environment during motion execution (this is as easy as starting another SPA cycle, whenever deemed appropriate), but that the choices made in determining the *act* model can not influence the *sense* or *plan* choices anymore. This reduces the opportunities for task-dependent configuration of sensing and planning.

Chapter 11

Meta models for software architectures

A **software architecture** is the design layer between (i) the **application-centred information architecture** and (ii) the **physical resource-centred hardware architecture**. The software architecture complies to the functional constraints of the application (in, both, information and hardware), and links them to the non-functional system requirements of (first) **loose coupling**, (then) **composability**, (then) **performance**, and (finally, maybe) **explainability**. Software architects decide about which **software components** (or *sub-systems*) to create, to structure the design trade-offs between (i) reaching the above-mentioned objectives, and (ii) conforming to the *information architectures* that are being made available.

The core mechanisms of **composable architectures** that receive a software design model in this Chapter are:

- **symbolic indirection**, for runtime discovery, inspection, and reconfiguration.
- the **event loop**, for runtime reconfigurable reactivity.
- the **multi-threaded component process**, for deterministic ownership of data and resources, and reconfiguration of software deployments.

In combination with the core *information* architectural patterns and best practices, they form the *software* basis for all of the following system challenges:

- which **data structures** to implement, and in which granularity.
- which **functions** to implement to process these data structures.
- how to compose functions into **algorithms**, in a way that can be realised in **any** programming language.
- which **programming language** fits best to which deployment context.
- which **algorithms to deploy** in which **activities**.
- which **activities** to deploy in which **event loops**.
- which **solvers** to use to **schedule** algorithms for execution.
- which event loops to deploy in which **threads**.¹
- which threads to deploy in which **processes**.

¹Or to run them on the **bare metal** of the CPU. The term “thread” is used as a *pars pro toto* for every software mechanism that can **host the execution of code**.

- which threads to schedule on which **CPU cores**.
- which processes to start from which **shells**.
- which processes to deploy on which **processor**.
- which **channel** implementations (e.g., **streams**) to deploy between which activities, choosing the “best” policies for **buffering** (preferably in shared, **contiguous memory**), and **inter-process communication**.
- how to realise the **ownership** constraints of information and hardware architectures.
- which libraries to choose, in which programming languages.
- which operating system(s) to choose.
- which **coordination** to introduce between (algorithms deployed in) activities, for (i) their **access** to shared data, (ii) their **synchronization** of their execution schedules, and (iii) their deployment on **multicore CPUs** and on **caches**.
- which quality to aim for in data **freshness** and **consistency**.

This Chapter focuses on the **system level** aspects of software; in other words, **computer science** and **software engineering in the large**. That does not imply that it neglects **programming in the small** [29, 30]; on the contrary, but it assumes each system software practitioner to master already those software engineering practices. In its examples, this Chapter uses the following established technologies: the **C** programming language and **standard library**; the **Linux/UNIX** operating system; **POSIX threads**; and the **Bash shell**.

The list above may seem overwhelming at first sight, and it is a good indication of:

- the **complexity** of designing a software architecture.
- (because of) the high number of **trade-offs** than can (or rather, must) be made.
- the **business model** opportunities that differentiate the “bad” system architects from the “good” ones. The latter can optimize a system just by adapting some “magic numbers” in the system’s **configuration** and **coordination** components, because they *understand* why their adaptations are appropriate.

This document brings in **very strong structures**, and conforming to them systematically and consistently reduces the design process to a methodological exercise of making well-identified trade-offs in well-identified design contexts. In addition, the same structure does not only support the *architecture* development, but also the *component configuration* in the system. The bad news, however, is that that potential has not yet materialised, for several reasons:

- software projects do not yet exist that provide the components that conform to the composable software architectural structures of this Chapter.
- the formal models of the architectural structures have not yet been created, let alone been harmonized, let alone been standardized.
- (hence) tooling to support the software configuration process does not yet exist either.

Data structures are in most cases the design decision with the highest impact on:

- the semantic unambiguity of the meaning of the data.
- the efficiency of function implementation, buffering and interfacing.

11.1 Mechanism: mapping between symbolic IDs and memory addresses

(TODO: CRUD data structures and indexing functions; hash table, splay tree, hierarchical bitfields,...)

11.2 Mechanism: bitfields for flags, FSMs and Petri Nets

Earlier Sections introduce the information-level coordination primitives of [status flags](#), [Petri nets](#) and [state machines](#). This Section explains their *implementation* by means of *bit fields*.

11.2.1 Bit fields

A *bit field* is a very efficient data structure

- **to store** a rather limited number of *flags* or *enumerations*, in *unsigned integers*. Hence, the number of flags that can be represented is limited by the number of bits in the integer; typical sizes are 8, 16, 32 or 64 bits.
- **to process** event logic by means of *masking* and *bitwise operations*.

Hence, the computer representation of the states, the transitions and the events, can happen in the following way:

- the **states** are encoded in a bit field.
- the **state transitions** are encoded in a bit field.
- the **state-transition-state** connections are encoded in composite data structure with three bit fields:
 - one bit field represents the “start” **state**.
 - one bit field represents the **transition**.
 - one bit field represents the “end” **state**.
- the **events** are encoded in a bit field.
- the **event-to-transition** table connected to one particular transition is a composite data structure, with two fields:
 - one bit field represents the **transition**.
 - one bit field represents all **events** that trigger that **transition**.
- the **transition-to-event** table connected to one particular **transition** is a composite data structure, with two fields:
 - one bit field represents the **transition**.
 - one bit field represents all **events** that are triggered by that **transition**.

With these data structures, the event processing runs as follows:

- *firing* of events happens by adding the event bits to a bit field, and *pushing* it onto a *stream ring buffer*.
- *processing* of events happens by *popping* an event bit field from a *stream ring buffer*, using bit operators to get the individual events out, and reacting to the events in a *switch statement*.

11.2.2 Dictionary of linked lists

(TODO: representation in *associative array* to implement *symbolic indirection* between external and internal “names” of flags; advantage of runtime adaptability, at the cost of runtime

memory allocation.)

11.2.3 Events or flags?

A status flag for a finite state machine can make *less* state information observable than there is available internally, and with different symbolic tags; for example, it is possible that the status flag for a [Life Cycle State Machine](#) shows only the values `inactive` and `active`, where the former holds the FSM is in any of the sub-states of the `deploying` super-state, and the latter holds for all the sub-states of the `active` super-state.

Events are connected to changes in state and status flags, as, both, “sources” and “sinks”: a state/status change can give rise to the firing of an event, and the processing of an event can cause a state/status change.

The abstract data type of all of the above-mentioned concepts can be `enums` (“[enumerated values](#)”) that is, they can take one of a finite number of possible symbolic names.

Events are most appropriate in the [Coordination](#) and [Orchestration](#) coordination modes, and status flags for the [Orchestration](#) and [Choreography](#) coordination modes. The overlap in the [Orchestrated](#) mode is as follows:

- events for the coordination of activities *not* in the same process address space, because there state must be *communicated*.
- status flags for the coordination of activities *in* the same process address space, because there state can be *observed*.

11.3 Mechanism: buffer, queue, socket

The [unique identifier](#) of a data chunk in a stream is a *symbolic name* of the data chunk, or, equivalently, a *symbolic pointer* to the data chunk. There are several complementary (and not mutually exclusive) ways the “pointer” mechanism is being used to realise *semantic classes* of activity interactions:

- (shared memory) **buffer**: both activities share the same memory space, and can use symbolic pointers to the data chunks in the buffer.

The *advantage* of the shared memory mechanism is that the data need not be copied, and all of the available information is accessible, directly and repeatedly.

The *disadvantage* is that there is no clear ownership of the data, hence one must add *data access constraints* to all activities that share the information. These constraints must be followed by all functions that use the information, so they interleave their execution in a predictable, consistent, and resilient way; the result is equivalent to the introduction of a stream:

- the stream imposes a strict order, to be followed by the interleaving.
- each data chunk contains the symbolic pointer that is available to the owner of the data chunk.

Most often, these constraints are to be satisfied *by discipline* of the programmers, instead of being guaranteed by construction, in programming languages or software tools.

- (message) **queue**: this mechanism works between two activities in the same address space or not, and both rely on the services of the operating system. That acts as a

mediator, by taking care of (i) copying the data to and from the streams in the address space of both activities, via **write** and **read** operations, and (ii) deciding which data to forward from a producer to a consumer.

A special case of message queues are *event queues*: one activity can have to react to events from one or more other activities, so a stream buffer is a good solution for each “event queue” between two activities, and the data chunk for an event typically consist of nothing more than a symbolic tag.

The *advantage* of a message queue is that the developers of the reading and writing activities need not bother about the data access constraint: the data must be *copied* anyway, to be sent over the “wire”.

The *disadvantage* is that the execution of each activity has some side effects: the data comes only available one by one, in the strict *first in, first out* order of the **queue**; higher memory usage because of the data copying; and non-deterministic pre-emption of their execution, because the “**locking**” of the buffer pointers can happen implicitly behind the screens.

- **socket**: when activities do not share directly accessible memory,² the message queue mechanism is extended further: the “local” message queue is not really one single queue, but the data that is written to it is sent to another computer or process via a communication channel, where it is copied to the local message queue of the program there. The interfacing activities can still use the same **send** and **receive** operators.

The *advantage* is the same as for message queues, i.e., the implicit satisfaction of the data access constraint. For so-called “**embarrassingly parallel**” applications, it *might* be that the overall time to do computations is reduced, because more cores can be used at the same time and the relative costs of communicating the data structures is dwarfed by the time needed for computations on the data.

The *disadvantage* is that the communication *overhead cost* is increased: the data must be copied several times; communication takes longer the “further apart” the hardware cores that execute the communicating processes; the channel between both processes must be kept alive in a **socket**. A second disadvantage is that “*the*” *state* of a data structure does not exist, because of the many **copies to be kept consistent**, all the time.

The operating system on a computer is the natural place to embed all of the above-mentioned mechanisms, exactly *because* it is “just” mechanism that applications can and should compose with their specific policies. Major aims for “mechanism” developments at the platform level is to strive for (i) standardization, (ii) performance, (iii) resilience, and (iv) security. Realising these competing goals together requires a huge effort, which is another reason to share this effort by all stakeholders of the platform. Examples illustrating this context in the case of operating systems are the inter-process communication projects **D-bus** (between processes on the same computer) and **WebRTC** project (between processes on different computers). In robotics, the **ROS** project tries to achieve the same role, but it has yet reached industry-grade maturity in any of the four design goals mentioned above; it also conforms to almost none of this document’s best practices and design patterns.

²That already is the case on one single computer, where *processes* have separated memory access, realised by **hardware**.

11.4 Mechanism: stream with wait-free ring buffer

The meta model of the *stream buffer* (Sec. 2.9) represents how two or more activities interact with each other (asynchronously, and in **producer** and **consumer** roles) with the exchanged data buffered in a strictly ordered **stream** of data chunks. This Section extends the **information architecture** model of the stream with software design aspects (Fig. 11.1 and Fig. 11.2). The design covers the **single producer/single consumer** case, with serial **pipeline composition** of stream buffers.

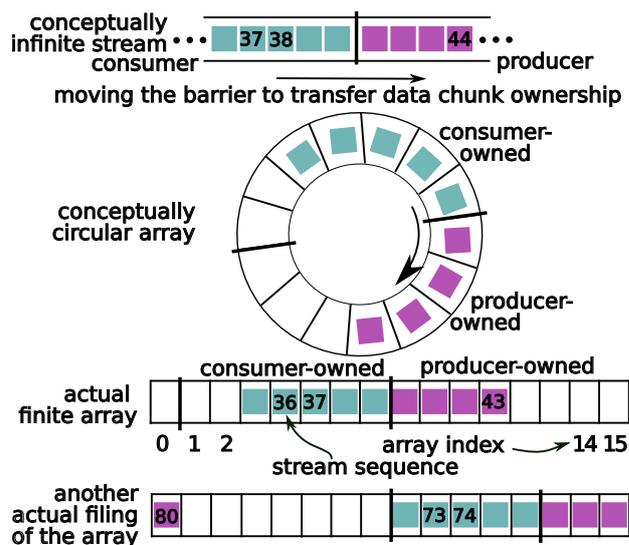


Figure 11.1: The information-architectural primitive of the **producer-consumer stream** (top), is mapped onto the conceptual software model of a *ring buffer* (center), and implemented with a finite array (bottom). The ring buffer array has two contiguous areas: one where the producer has data chunk ownership (in the examples, this area “flows over” to the beginning of the physical arrays), and one where the consumer has data chunk ownership. Each data chunk that is entered into the stream gets a new stream sequence number, counted by a perpetually incremented integer number.

11.4.1 Design overview

These are the essential features of the design:

- the **data structure** of the **ring buffer**³ allows to map a conceptually infinite stream onto a **finite array** in computer memory.
- more than two activities can be composed in a **pipeline**, by connecting the producer-consumer ring buffer pattern back to back,⁴ and deploying them onto one and the same array.
- one activity can be active in more than one part of the same pipeline. In other words, it can **own** the data chunks in more than one contiguous sub-buffer.
- each such contiguous sub-buffer with unique ownership is called a “**Port**”.
- an activity **transfers ownership of data chunks** to the next activity in the pipeline, by moving the pointer to the oldest part of the sub-array that it owns, further into that area. In other words, it shrinks its own Port at its oldest side. “Older” means: a lower value of the monotonically increasing “logical” stream sequence number.

³“Circular buffer” is a commonly used synonym.

⁴To the best of the authors’ knowledge, this mechanism was first introduced in the **Disruptor** variant of the ring buffer.

The ownership transfer mechanism is **symmetric** between all activities, because the “un-owned” area of the ring buffer is considered to be owned by the **first activity** in the pipeline (sometimes called the *source* of the stream).

To allow activities to improve the **throughput** performance, the following **status flags** can (optionally) be added to a *Port*:

- **Activity**: this is an **enumerated type** with which an activity informs the other activities about its actual streaming behaviour on that Port:
 - **Active**: the Port owner has stream activity in its current behavioural state.
 - **Pausing**: the Port owner is ready to stream, but waiting for (an unidentified) trigger by another activity.
 - **Inactive**: don't expect any stream activity from the Port owner.
- **HighWater** and **LowWater**: these are two **Boolean** flags, via which a producer (respectively, a consumer) informs its consumer (respectively, its producer) that its part of the buffer is getting too full (respectively, too empty), and, hence, it invites the other activity to become more active on the stream.

Every activity on a stream is free to neglect the information in the status flags of the other activities' Ports.

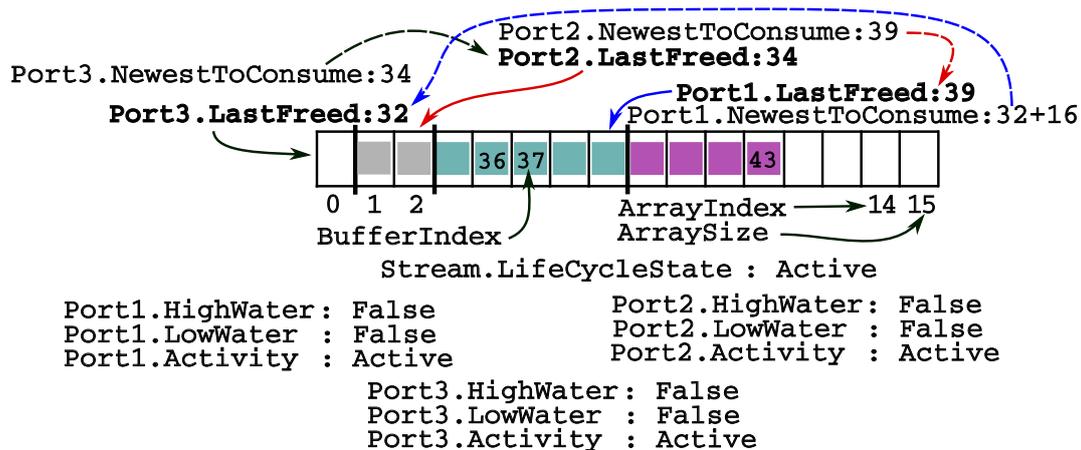


Figure 11.2: Example of an instance of the abstract data type of a `StreamBuffer` with three activities having a `Port` on the buffer. The first activity owns a lot more data chunks than it has already processed; the latter are identified with coloured slots in its `Port`, the former with blank slots.

11.4.2 Block-Port-Connector conforming meta model of a Stream buffer

The stream buffer meta model (Fig. 2.20, Fig. 11.1 and Fig. 11.2) conforms to the **Block-Port-Connector** meta meta model:

- **Block**: each of the activities that are active on the stream buffer, that is, that have a `Port` on it.
- **Connector**: the array of the stream buffer, and the list of the `Ports` that are connected to it.

- **Port**: the data structure that the activity in each Block needs to access the data chunks in the Connector buffer.

The following [abstract data types](#) are the core of a Stream buffer:

```
// The "Connector" meta model of a stream buffer :
StreamConnector : {
  Integer      : Size ; // number of entries (>0) in the buffer's array
  MemoryAddress : Start ; // pointer to the array's first data chunk
  Type         : ... ; // type of a data chunk; depends on the application
  StreamPort   : [...] ; // ordered list of Ports of activities using the buffer
  LifecycleState : ... ; // state in a Life Cycle State Machine
}

// The "Port" meta model for an activity using the stream:
StreamPort : {
  StreamConnector : ... ; // Connector to which this Port is attached
  BufferIndex      : LastFreed ; // last chunk whose ownership was transferred
  * BufferIndex    : NewestToConsume ; // pointer to the "LastFreed" of the producer
  // i.e., the newest chunk available

  Enum           : Activity { Active, Inactive, Pausing } ;
  Boolean        : HighWater, LowWater; // backpressure flags
  Integer        : HighWaterThreshold, LowWaterThreshold ;
  // thresholds on buffer usage before it is "too full/empty"
}

```

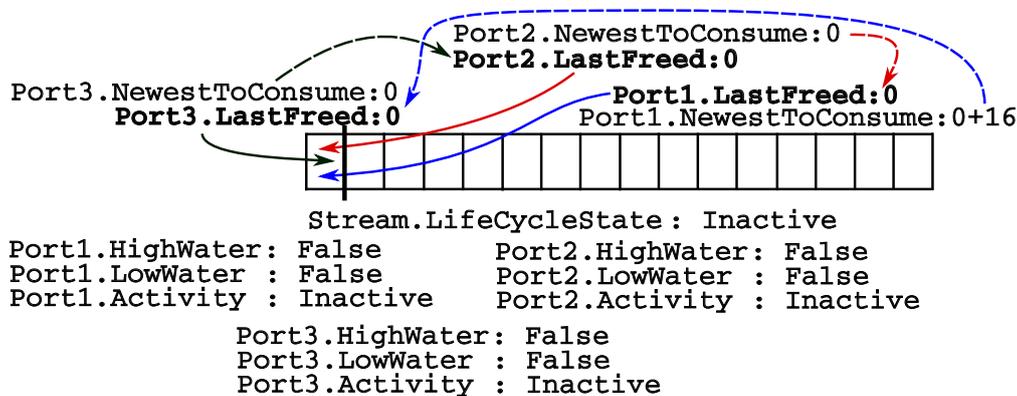


Figure 11.3: Empty/initial state of a Stream ring buffer with three activities. That means, that the whole buffer is owned by the first activity's Port1. Note that all indices into the *array* are 0, but the logical index into the stream buffer for the last Port is the buffer Size 16.

11.4.3 Implementation design

This Section designs a stream buffer for the **particular but high-impact case** of a **single owner** activity of every Port in the stream buffer. All activities share the memory needed for

the array underlying the stream buffer, but their behaviour can be **deployed** in a *thread-safe way*, in the same thread, in different threads, in different memory-sharing processes, or on different cores. The number P of activities acting on the buffer can be arbitrary large, thanks to the design decisions described here:

- the **stream's buffer** (which is conceptually an **infinite** list of entries of the same type) is mapped onto a **finite array**.⁵
- a **data_chunk** model represents the unique type of each entry in the stream.
- the **sequence number** of a data chunk in the stream buffer is an **integer number**, the `BufferIndex` k .

The sequence number is **incremented perpetually**, every time a new entry is produced. “Perpetually” is not completely true, but, even at a filling rate of one million entries per second, a 64-bit integer counter can guarantee that a **buffer overflow** will not occur for more than half a million years.⁶

- at initialisation of the stream buffer, all data chunks are owned by the activity whose **Port** is the first in the ordered list of **Ports** (Fig. 11.3).
- at all times during the life time of the stream buffer, the array is divided in P **contiguous parts**, when there are P activities acting on the buffer. (The simplest case has $P = 2$, with a *producer* activity and a *consumer* activity.) The ownership of all contiguous parts is **always** the order in which the **Ports** have been defined.
- all all times, an activity can access **all** the array entries in the **Port** that it owns, and do so in **random access** order. This access semantics is different from a **queue**, that has more constrained “**push**” and “**pull**” access semantics to only the first and last entries.
- an activity can **transfer ownership** of **any number** of the array entries that it owns, and do so in **one operation**, as long as it respects the **strict ordering** constraints on the list of **Ports**.
- the buffer becomes conceptually **circular** by equating the successor of the last entry in the **buffer's** array to the first entry in the **buffer's** array.
- the **buffer's** array has an **integer number** N of entries, represented by **Size**.
- referring to an entry in the array requires an **integer number**, called an **arrayIndex**. Such **arrayIndex** numbers are not constants, but are constrained to the **inclusive interval** $[0, N]$.
- by making the **Size** N a power of 2, the **arrayIndex** i of an entry in the stream can be computed cheaply from the latter's **BufferIndex** k , via a **modulo** operation: $i = k \bmod N$. Indeed, a $\bmod 2^N$ operator is implemented as a very fast **bit shift**.

⁵The software realisation of the stream as an array has two parts: an **abstract data type**, and a **data structure**. This Section focuses on the former model; the result has enough details to serve as documentation for a transformation into concrete data structures in concrete programming languages.

⁶ $2^{64}/10^6/60/60/24/365 = 584\,942$ years. A 32bit integer, with a sample rate of 1000Hz, only lasts for a couple of weeks: $2^{32}/1000/60/60/24/365 = 1/7.5$ years.

The example in Fig.11.1 has $N = 16$, so that `BufferIndex 36` has `arrayIndex 4`, and `BufferIndex 80` has `arrayIndex 0`.

- the array is positioned somewhere in the **RAM memory** of a process, and that address is pointed to by the `Start` memory address integer.
- in addition to the **logical index** into the buffer (of type `BufferIndex`), each activity also needs **two pointers** (of the `arrayIndex` type), to point into the **physical** array memory (Fig. 11.2):
 - **LastFreed**: points to the entry in the owned part of the array whose ownership was last transferred to a “downstream” activity.
The *data chunk* this pointer is pointing to is, hence, **not owned** by the activity anymore; the *pointer* itself *is*.
 - **NewestToConsume**: points to the **LastFreed** pointer of the “upstream” activity. That pointer then points to the newest entry for which ownership has been transferred to this activity.
The ownership situation is the opposite as for **LastFreed**: the *data chunk* this pointer is (indirectly) pointing to **is owned** by the activity, but the *pointer* itself *is not*. The latter should only be **observed** by the activity that owns **NewestToConsume**.
- the *mechanism* of transferring ownership is the same for all **Ports**, but the **first Port** (the stream buffer’s **source**) has some specific *implementation* constraints:
 - for its **NewestToConsume** index, it must always add the **Size** of the array to the value it reads from the last **Port**’s **LastFreed** index, to make the difference between an empty and a full array unambiguous.
 - it is the only **Port** confronted with the constraint that the underlying array is *finite*. So, sooner or later, it will not have enough slots available to receive its produced data chunks, and it is then forced to make a decision about which **data chunks to remove** from the array.
- this particular choice of pointers and ownership simplifies the implementation of the **operators** to transfer data chunks from one activity to another, because that can be done without locking, in a thread-safe and **wait-free** way. The producer activity can indeed **change** the **LastFreed** pointer, without risking a **race condition** with the consumer activity, because the ownership transfer operation involving **LastFreed** does not overwrite any consumer-owned variables.
- ownership transfer just involves changing pointers, and **Compare-and-Swap (CAS) operations** on such 64-bit integers are supported as **atomic operators** by all operating systems, and even natively by CPU hardware architectures.
This guarantees **correctness**.
- putting all the `arrayIndex` pointers in their own **cache line** (filling the rest of the cache line with unused “padding” bytes) guarantees that they can be updated independently without causing each other’s cache lines to be refreshed.
This improves **performance**, without compromising **correctness**.

- the following **data consistency constraints** must hold at all times:
 - **Size** should never change.
 - **Start** should never change.
 - the buffer can not be filled to more than its capacity, that is, the **LastFreed** pointers of subsequent activities should never overrun each other.
- each activity acting on the buffer can apply a **backpressure** policy, with which to set the value of its **HighWater** and **LowWater** status flags. They indicate to the activity's downstream and upstream activities whether it wants them to fill or empty their parts of the buffer, and can be computed by introducing two local integers, **HighWaterThreshold** and **LowWaterThreshold**, as follows:

$$\text{HighWater} = (* \text{NewestToConsume.BufferIndex} - \text{LastFreed.BufferIndex}) \quad (11.1)$$

$$> \text{HighWaterThreshold},$$

$$\text{LowWater} = (* \text{NewestToConsume.BufferIndex} - \text{LastFreed.BufferIndex}) \quad (11.2)$$

$$< \text{LowWaterThreshold},$$

Both status flags should be initialized to **false**, indicating that no backpressure policy is used.

- the status flags for backpressure and for **Activity** are of an **enumeration type**. These can be read and written atomically on all hardware platforms, hence, updating the status flags does not introduce race conditions.

(TODO: add descriptions of the operators on the data structures.)

11.4.4 Policy: late binding of data chunk type

(TODO: just allocate an array of bytes, to be interpreted by the application, even at runtime.)

11.4.5 Policy: sharing a stream between processes

On many hardware architectures, the operating system can **map** that physical memory space onto the virtual address spaces of different processes. The latter can then **share** the same physical memory, hence, stream buffer use cases are not limited to deployments in one single process only.

11.4.6 Policy: composition of data and metadata

The software-centric additions to the information architecture description in Sec. 2.9.6 are:

- the metadata **data_chunk** data structure must be defined. In the worst case, every data chunk requires its own metadata chunk, so the size of the metadata stream is that of the data stream.
- (hence) the metadata stream does not need another **backpressure** support or status flags in addition to the ones in the data stream.
- the ownership of the producer and consumer pointers in the metadata stream is with the same producer and consumer activities.

The former is a major responsibility of the information architect.

11.4.7 Policy: time series

Time series (or “data streams”, or “event streams”) are a very important use case of streams with metadata. (At least) three variants of the data structure of the metadata exist:

- the *singleton* metadata structure: each entry in the data stream buffer has its own time stamp, and it is part of the data chunk.
- the *metadata stream* version (Fig. 2.22). Each data chunk in such a metadata stream buffer has fields to represent:
 - the *range* of stream sequence numbers for which the following two metadata pieces of information hold.
 - the **provenance**: how was the data created, where does it come from, etc.
 - the **time representation** for each range.
- the *one-on-one metadata stream*: the metadata stream buffer has the same size as the data stream buffer, and each chunk in the meta buffer contains the metadata of one data chunk in the data buffer.

(TODO: Apache Arrow software and models; role of developing standards on **getusermedia** (sources, streams, tracks and channels; capabilities and property settings; constraints between streams, tracks and channels), **audio**, **WebRTC**.)

11.4.8 Policy: composition of stream and memory pool

This Section brings the information-level stream specialisation of the composition of the stream model with an **object pool** to the more concrete software level. That is, “objects” are assumed to be **serialized** into arrays, Fig. 11.4.

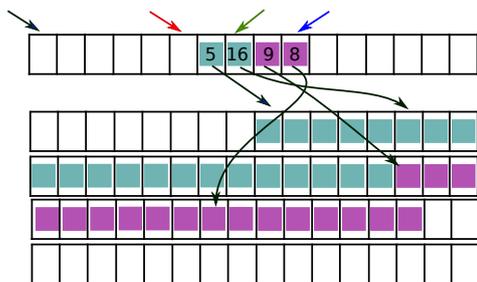


Figure 11.4: Stream buffer with memory pool. Each data chunk in the stream buffer contains two integers: the address pointer into the memory pool of the array that contains the serialized data chunk of the “real” data, and the size of that array. The stream buffer is used for the ownership transformation of these pointers from producer to consumer.

11.4.9 Policy: heartBeat/watchDog mediation for “lazy” stream

In a context in which the producer and consumer activities are **distributed** over several processes or computers, the status flags for **backpressure** and **peer activity** of one peer can not be read synchronously by the other peer. So, that information has to be sent via communication messages. Then it makes sense to add **heartbeat** events: when there has not been a communication for some time⁷ an event is sent, by the producer and/or by the consumer, to indicate that they are still engaged in the stream communication but have had

⁷That **timeout** is a configuration parameter.

no need to send over data chunks for some time.

The `watchDog` approach serves a complementary use case:

- a third **mediator** activity is introduced.
- it waits passively for **heartBeats** from producer and consumer.
- when they do not come in for a particular period of time, the mediator sends a **watchDog** message to the “delayed” peer, to trigger it in reacting with a **heartBeat**.
- if that also does not activate all involved peers, the mediator fires reconfiguration events for its subsystem of the whole system.

11.4.10 Policy: contiguous data for producer and for consumer

Keeping the buffer memory of the producer and of the consumer contiguously together in memory can improve **memory access performance**. However, the implications of this particular implementation of a stream buffer are:

- when adding entries to a sub-stream makes the array overflow, one has **to copy all** entries to the beginning of the array.
- to guarantee that this can always be done, the size of the ring buffer array must be **triple** the size that is guaranteed to the consumer or producer to own.

Indeed, even with an almost full ring buffer, the producer activity must still be able to copy its sub-stream from the “end part” to the “start part” of the ring buffer array, so it is possible that the consumer substream must be copied first towards the “middle part” because it still occupies the “start part” of the array.

In summary, this design should only be chosen if the application has a much more frequent need for *processing* the data in the buffer than for *filling* the buffer. For example, for algorithms that can exploit the specific vector processing capabilities of the CPU hardware.

11.4.11 Mechanism (software): stream with wait-free maximal freshness buffer

The stream buffer mechanism of Sec. 11.4 has the advantages of:

- very clear and efficient ownership protocol.
- high reactivity of producer and consumer to the status of their interaction.
- “random access” to all data chunks that can fit in the buffer.

There is an activity interaction use case, however, that is not well served: the interacting activities want only the most fresh data, and all “old” versions of the data chunk can be deleted. Major examples of that use case are where one activity needs:

- images from a camera: an image can be large, and (hence) take a lot longer to process than the sample time of the camera. So, processing all images would be more *time* consuming, and *memory* consuming than needed.
- results of repetitive database queries: an activity can subscribe to updates from a database, but only wanting to process the last update it received.
- data in a real-time control loop: almost by definition of “real-time control”, old data is not useful anymore.

Applying the stream buffer mechanism to these use cases results in situations where the producer has transferred ownership of so many data chunks that:

- it can itself not get rid of the latest information that it has available because the buffer is full.
- the consumer needs to look at too many data when it decides to check the buffer again.

This Section introduces a more suitable mechanism for the presented use case, which uses the same software engineering building blocks as the stream buffer, namely:

- atomic switching of 64-bit “pointers”.
- perpetually incrementable sequence numbers,
- that double as offsets pointers into a datachunk array, after applying a modulo function with the length of the array.
- that modulo function is very efficient if the array has a length which is a power of two.

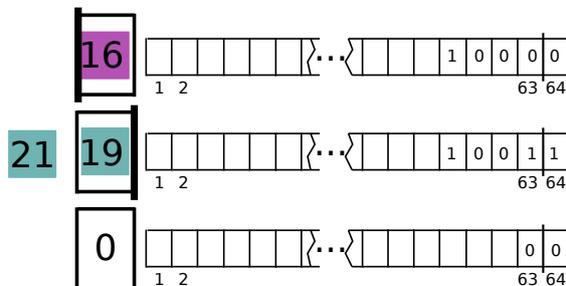


Figure 11.5: Stream with optimal freshness. Both producer and consumer can write, respectively read, the latest available version of the shared data structure. In the example, the consumer is still working on data chunk “16”, while the producer has already made data chunk “19” available for reading and is now starting to fill the empty slot with data chunk “21”. The bold vertical bars represent ownership: it’s at the left when the consumer owns the data chunk, and at the right when the producer owns it. This information is encoded in the 64-bit sequence number, by “sacrificing” the last bit: that is “0” when the chunk is consumer-owned, and “1” when it is producer-owned.

Figure 11.5 illustrates the approach, that uses **conditional transfer of ownership** [51]:

- instead of one array buffer with multiple sequence-ordered spots for data chunks, the mechanism uses **three data chunk slots**. This is semantically identical to saying that it uses three stream buffers in parallel, each with a **Size** of “1”.
- the producer always uses only **uneven** sequence numbers.
- when the consumer wants to read a new data chunk, it checks the three slots in whatever order, and when it finds a newer one than the slot it consumed last time, it **increments** the sequence number by one, in an **Compare-and-Swap** atomic operation.
- that makes the sequence number **even**, indicating to the producer that the consumer has **accepted the ownership** that the producer has offered. That **offer** was **conditional**, because the producer can decide to clear a slot that it filled previously, *and* that the consumer has not taken ownership of. Both logical conditions can be checked atomically, by the “trick” of the even/uneven sequence numbers.
- the consumer now starts consuming that slot.

- *after* the consumer has consumed the slot, it **decrements** the sequence number by one, so that it becomes **uneven** again.
- this means that the consumer has transferred ownership of that datachunk back to the producer, *and* that the slot represents “old” data. (Indeed, the producer has already used that old sequence number.)
- instead of letting the producer increment its sequence counter by “1”, and fill that in in the first old data chunk that it finds (that is, one with an even sequence number) it now **increments** its sequence number by “2” before filling it in.
- the procedure above implies that the producer can *always* have (i) one slot available to the consumer, and filled with the latest version it was able to write, and (ii) one slot available to itself, to start filling with new data.
- after it has filled a new slot (and before it tries something else), it visits the other two slots and if it finds the (maximally) one with an uneven sequence number, it resets that sequence number to “0”. That means that slot is tagged as being **empty**: any slot filled with available data has a sequence number of at least “1”.

This mechanism guarantees that the producer must never wait to fill a slot with the latest version of its data chunk, and that the consumer must never wait to read the latest available data chunk. Hence, the semantics of the “wait free” and “maximal freshness” in the name of the mechanism. The worst that can happen is that the producer trails the consumer in visiting the three slots in such a way that it just has made a new slot available but the consumer does decide to take the older one. The consumer can compute how many data chunks from the producer it has missed: it is *half* of the difference of the last sequence number and its previously read sequence number.

11.4.12 Standards and software projects supporting stream channels

Many established internet protocols are special (simplified) cases of the **Stream** meta model. For example, **SCTP** at the **application layer**, **RTSP** at the **transport layer**. Implementations for these standards come with mainstream operating systems.

At the time of writing, the **WHATWG stream standard** is reaching maturity. It includes an explicit meta model, as well as a reference implementation in Javascript. The ZeroMQ ecosystem provides a (partially conforming) model and an implementation in the form of its **exclusive pair pattern**. Both implementations have chosen for the *policy* of blocking the producer or Consumer when their stream buffer is full or empty. This document follows a less constrained meta model, allowing other policies, such as: to overwrite the oldest or the youngest entries, or to compact a stream.

11.5 Mechanism: event loop

An event loop can be realised with various primitives that encapsulate the asynchronous interactions behind a synchronous communication interface:

- **callback**: a composition of a data structure and a function. It is the responsibility of the **context** to guarantee that the data is in a consistent state whenever the function is called.

- **source**: the composition of a **callback** with the metadata required for *mediation* within (a context inside) an event loop, to allow “optimal service” to be given to all callbacks. Some mediation relations are: priorities between the execution order of callbacks, or the information needed to decide whether some callbacks can/should not be executed any more.
- **event**: this data structure is filled in, either by the application itself, or by the operating system (and hardware); events are read out by the event loop (in its `communicate()` phase) to find out whether or not, and what, data is available from synchronous and asynchronous sources. The synchronously executing algorithms of the application can fire events, but their handling can (often) be realised completely within the application activity, without support from the operating system.
- **context**: the composition of **sources** that adds two relations to the meta model: (i) the sources over which to do the mediation, and (ii) all the data and constraints needed to execute all the functions in the sources.
- **poll**: the function that a **context** is executing on a **source** (or rather, on the *streams* produced by a **source**), to read the information about what data an asynchronous I/O source has available for reading, or has sent out to “somewhere else”.
- **prepare, check, dispatch, finalize**: these are *bookkeeping* functions, that are (possibly) executed early in each iteration through an event loop, and in that order,⁸ for each individual I/O **source**:
 - **prepare**: do **source**-specific configurations to make it ready for polling, if needed.
 - **check**: read some “register” data in a **source**’s local data structure, to find out whether it is ready to be polled in this ongoing run through the event loop.
 - **dispatch**: actually execute the **poll** and corresponding **callback** function.
 - **finalize**: do **source**-specific configurations to “clean up” a polled **source**, if needed.

Not all functions need to be defined for every **source**, and when they exist they need not necessarily be called every loop iteration. These decisions are part of the *policies* of the event loop *mechanism*.

- **loop**: this is a data structure representing (i) the *order* in which the above-mentioned functions are executed, on their respective **sources**, and (ii) the maximum *time* that it wants to be blocked on a **poll** before it pre-empts that **poll** and continues with the rest of the functions in the loop.

11.5.1 Event loop design trade-offs

An event loop allows the **application** developer (and not the operating system) to **configure** the balance between the following **design forces**:

- **localising synchronization of *side effect-full* data exchange**. Side effects inevitably take place, via (hardware and software) mechanisms like **interrupt handlers**, **mutexes**, **condition variables** or “**lock-free**” and “**wait-free**” buffers, etc. The pattern’s solution is to *copy* the data from/into asynchronous sources into/from a “thread-local”

⁸Functions from several sources can be taken together, respecting the same overall order.

storage to which only the event loop thread has access. The above-mentioned mechanisms are explicitly recognizable in the programme code, so that it is at least *possible* to identify the areas in the code where side-effects can not be avoided.

- **event handling latencies.** There are almost no robotic applications in which **asynchronous I/O** is not present, because lots of sensors and actuators have to be interfaced, and processes must communicate. The event loop pattern provides application developers with one **callback object** per I/O channel, and has a **mechanism** (i) to select which I/O channels to deal with at any particular iteration through its “loop”, and (ii) to configure how long it wants to wait on the channel.
- **prioritization of event handling.** The above-mentioned selection of I/O channels is done with **priority queues**, for which the configuration is again under the explicit control of the application developers.
- **off-loading processing.** The application developers can configure a **thread pool** of “*worker threads*”, in addition to the “*main thread*”, to let each long-running event handler be dealt with in a separate “worker”, and to make the results accessible to the main thread via a **message queue**.

11.5.2 Event loop with ring buffer and scatter-gather I/O

The software architecture in this pattern has the following parts:

- the *ring buffer* data structure (Sec. 11.4)
- an event loop (Sec. 2.5) that executes the producer activity in one process.
- an event loop that executes the consumer activity in another process.
- both event loops also run the activities that realise the **vectorized I/O** (also called *scatter-gather I/O*) that is needed at both producer and consumer sides.

The scatter-gather activities are needed because each message in the stream between both processes can contain chunks from different consumers and/or producers in each process, so the messages have to be composed before sending and decomposed after receiving, and their parts copied to the correct local destinations.

Some **memory management** and **compute kernel** hardware has this functionality built in.

11.5.3 Runtime configurable instrumentation

(TODO: setting or clearing flags like “DEBUG”, “Log-Level-i”, ..., allow to schedule the corresponding instrumentation functionalities inside an algorithms nominal event loop.)

11.5.4 Multi-rate time series streams

The **event loop** is the computational work horse in software activities, and the following use case presents itself in many control systems:

- the event loop services multiple activities, each with one or more solvers, that each process one or more time series

- **multi-rate sampling**: the iterations for several solvers run at an order of magnitude difference in time scales. For example, a current control loop at 5kHz, a torque control loop at 1kHz, a platform velocity control loop at 100Hz, and a Finite State Machine for the task `plan` at 10Hz.
- the exact frequency is not so important; the order of magnitude is.
- many of the time series require **timestamping**.
- many use cases want to process the various data streams together, with time as the major index.

If system designers have gone through the effort of making models for each activity and stream, because tooling can use the models to exploit having the overview of everything that has to happen in each iteration of the event loop, **to configure** the latter’s implementation with the following optimizations:

- one time stamp can serve all streams.
- all provenance metadata need to be recorded only once.
- the timing in the multi-rate sampling can be chosen to be **powers of two**. For example, 8Hz, 124Hz, 1024Hz and 4096Hz, instead of the above-mentioned 10Hz, 100Hz, 1000Hz and 5000Hz. This allows efficiency gains in the selection of which solvers to trigger in each iteration: the 64-bit integer that the event loop uses to count its “ticks” just needs an efficient **modulo operation** for this selection.
- the overhead of **vectored I/O** (or *scatter-gather I/O*, of individual data chunks in individual streams) can be avoided by serializing all data chunks to or from one single I/O stream.

Tooling support to do this efficiently, network-order independently, while keeping **direct access**, exists in mature projects like **FlatBuffers**.

- often, a solver in an individual activity needs two forms of **iteration**:
 - incrementing its “tick”, to select the next entry in the stream buffer to read from and/or to write to.
 - iterating over its own algorithmic serialization. For example, **to visit** all links and joints in a kinematic chain.

The event loop can take care of the former, efficiently, using the **fetch-and-add** operation to set the starting position of the second iterator to the corresponding position in the stream; the solver can then just increment that second iterator, as if it were starting from zero.

11.6 Architecture to compose task queues, workers, event loops, and solvers

This Section introduces the components in the **computational pattern** that is the software counterpart of the similarly named *concepts* of Sec. 2.6.7, and the *information architecture* building blocks of Chap. 10:

1. **task queue**: data structure that lists “what has to be done”.
2. **solver**: computes “who does what”.
3. **worker**: serves as “the agent that does it”.
4. **event loop**: triggers the workers to actually “do it”.

11.6.1 Pattern: composition of “task queue” and “library API” patterns

System designs need a composition of

- concurrent and asynchronous activities deployed in threads, with
- serialized and synchronous function calls on data structures.

The coordination primitives of FSMs and Petri Nets work only for activities, because they are *declarative* models of *what* has to be done, and not *imperative* control flows that dictate *when* that has to be done. The declarative design approach fits naturally with “task queues”; the imperative design approach fits naturally with “APIs of libraries”. Both are needed in most systems. Only the systems that require that all activities are deterministically defined at compile time, can profit from a “library-only” design. Or rather: from a design where the only declarative part comes from the operating system’s management of resources, and the access to these resources. The latter kind of systems are most often encountered in deeply embedded applications (because of the lack of resources to solve the declarative descriptions in series of function calls), or in aerospace and avionics (because the high verification and certification expectations in these domains have, for the time being, favoured architectures where the software architecture is so-called “statically allocated”. Nevertheless, the latter domains are also the ones that pioneered the introduction of formal FSMs and Petri Nets, because these are the declarative models whose behaviour can be fully proven mathematically. At least, when the decision making is close to first order logic.

The event loop can rightfully be called a “software pattern”, because there exist already various realisations, with mature and large-scale application track records. Here are some of the largest realisations, with **industry-friendly open source licenses**:

- **libuv** event loop, powering the **Node.js** real-time web applications platform, and with the **V8** Javascript engine.
- Java, JVM, **Disruptor** event loop.
- **GLib Main Event Loop**:
The FSM **states** of the loop.
The event loop loops over callback **sources**, each providing **prepare**, **check**, **dispatch** and **finalize** functions; there three default tyeps: timers and file descriptors (the OS does the waiting and the loop polls for ready events) and idle ones, that are always ready to be dispatched (= run):
The loop can also deal with also “child thread” signals: CTRL-C etc.
- **SystemD event loop**
States: **states**
- **ZeroMQ based event loop**. It has a fixed attachment between sources, context, and thread.
ZeroMQ offers **inproc messages queue**, which fit in the event loop context to support inter-thread communication, between the main loop and its workers.

11.6.2 Policy: frameworks, middleware, solvers

Two major instantiations in the domain of software engineering of the coupling between *mechanism* and *policy*, are **frameworks** and **middleware**: they **optimize the “usability”** of software by implementing the policy choices that are relevant in a particular application context. The advantage is *freedom from choice*: developers only have to make choices about the behaviour *of their application*, and not anymore about the *policies* of the *software basis* they rely on. The disadvantage is that these choices are most often hidden inside the framework, and hence **compromise the “reusability” and “composability”** of the application. if one or more of the policy choices are not optimal (or even feasible) within a different application context.

11.6.3 Composition of stream with object pool

This use case is a variant on the previous one, that fits well to an application context where the data structures for all messages must be allocated statically. This use case makes use of the **object pool** pattern, where a fixed number of instances of a fixed number of data structures are allocated in advance, and the producer of the stream just has to select a free one from the pool; the consumer must free that entry afterwards.

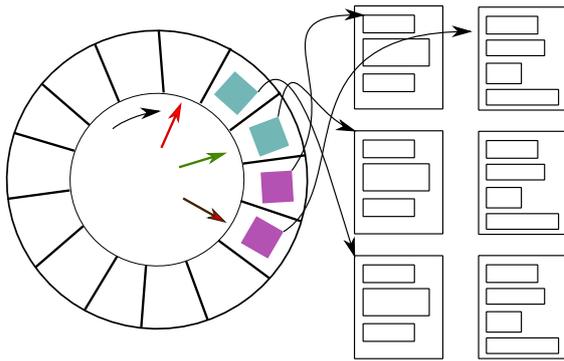


Figure 11.6: Stream with object pool. The data structures on the right are statically allocated, and they are only “borrowed” temporarily and “returned” at runtime, without being deallocated. The stream buffer is used only for efficient and effective ownership transformation of the pointers to the various available object data structures.

11.7 Architecture for hierarchical activity deployment

Robotic and cyber-physical systems can contain hundreds or thousands of software components, in the form of algorithms and activities. In order to bring structure in this abundance of components, system designers must search for *hierarchies*, wherever possible. Many of those hierarchies have already been identified, *and* modelled in the **system** and **information** architectures. This Section adds a very important hierarchy, naturally imposed by the structure of computer hardware and operating systems.

11.7.1 Hardware hierarchy: core, system-on-a-chip, computer, fog, cloud

Computer hardware provides hard constraints for a software architecture to comply to, caused by the following parts: CPU, **memory**, **bus**, I/O, and **network**. The CPU part comes in some variants on most modern hardware, depending on the degree of sharing memory (“caches” and “RAM”) and communication hardware; but the following hierarchy is clear and absolute, hence very useful:

- **core**: one CPU with some local “cache” memory that it completely under its own control.
- **processor**: a collection of **cores** that share some caches, and some buses to the RAM memory.
- **system on a chip** (or “*computer*”): composes several cores and peripheral hardware on one single integrated circuit, and coordinates their access to hardware shared communication buses.
- **computer**: a composition of several of the above computational mechanisms, integrated with communication, memory, IO and file storage functionalities.
- **fog** or **edge** system: a composition of computers that are under the management of one single system owner, and are (hence) deployed together, sharing an owner-specific network in an owner-specific location, with the **network-connected storage**, **authentication** and **network security** protocols that can be enforced by the owner.
- **cloud**: composition of several of the above, with the extra complexity of forming a system with multiple owners and stakeholders.

(TODO: formalize the [DOM Model](#).)

11.7.2 Software hierarchy: runtime, thread, process, shell, container, cloud

A similarly natural and strong hierarchy comes with all **operating systems**, offering **abstraction** and **resource sharing** of the above-mentioned computer hardware, to facilitate **software portability**. The hierarchy is as follows, from “bottom” up, that is, starting with the software entity that has the least contextual information about all software dependencies of the executed application:

- **runtime**: the “infrastructure” code that a compiler adds to the source code of an application developer, to facilitate generic functionalities, e.g., event handling. That means that there is an (most often implicit) [DOM model for event bubbling](#). There is one runtime per compiled binary, and the compiler has configured the sharing of the runtime objects that can be accessed from functions that have been *compiled* together. (TODO: explain role of WebAssembly standard, to make “runtimes” also composable and configurable during execution; **WASM** implementations.)
- **thread**⁹: the thread is the operating system mechanism that couples the functions in the compiled software code of an application to a CPU on the computer hardware. The thread **owns** very little: just two, but essential, interfaces between operating system and application, that allow activities **to share** the same **synchronous execution context**:
 - **thread ID** (OS centered): the unique identifier via which the *operating system* can **schedule and dispatch** the execution of the thread on a CPU.
 - **function pointer** (application centered): the address of the **event loop** function that composes algorithms in various activities into one single synchronous computational context. When access to various resources provided by the operating system must be coordinated, or when the execution of several tasks has to be coordinated, the event loop function is pointing to a **resource mediator** activity in the *application*.

Both identifiers are **attributes** of the thread and not properties: their values are as-

⁹More detailed introductions can be found [here](#) and [here](#).

signed by, respectively, (i) the operating system when *creating* the thread, and (ii) the application when *configuring* the deployment of an activity in the thread. The latter also sets other relevant attributes to configure the behaviour of a thread:

- conditions under which the thread wants to be scheduled, with *timing* and *scheduling priority* being major ones.
- constraints on **swapping** of the thread's RAM that the operating system should respect.

A **fiber** is a special type of thread: the operating system uses **cooperative multitasking** to determine when to execute which fiber, while threads are scheduled via **preemptive multitasking**.

A **coroutine** is another related concept, providing cooperative multitasking, but this time organised via a **programming language runtime** (e.g., **Lua** or **Go**) and not the operating system.

The thread is also the primitive to host the data connected to the **runtime** (re)configuration¹⁰ and **introspection** of the activities that it executes.

- **process**: the **composition** of the operating resources required by all threads. That is, the process **owns** the memory spaces needed for:
 - the **stacks** of the threads.
 - the **static** and **heap RAM** memory needed by the activities executed in the threads.
 - the **handles** (or, **file descriptors**) of all I/O resources, that is, handles to the **file system** and the **inter-process communication** channels needed by the activities executed in the threads.
 - the coordination of how threads are sharing the **CPU cores** and **virtual memory space**.
 - **compile time** configuration in **global variables** of the virtual address space shared by threads.

In other words, the process is the composition primitive that makes threads **share the same process context**. One process can start up other processes, via system calls to the operating system; this functionality and its ownership and access context is the same as for the *shell* below. In other words, processes can be composed in a **DOM model** of indefinite depth.

- **shell**: operating systems provide the shell primitive as a shared **context** for several processes to be active in. The same process code can be started from within different shells, but get another set of **environment variables** that the process uses to configure its behaviour. In other words, the shell is the composition primitive that makes processes **share the same operating system context**.

It is also the primitive to configure the **deployment**: in which order, and with which context, to launch several processes in an application.

- **container**: has a similar **composition** role as processes for threads, but now between operating systems and the computational hardware (which continues below in this hierarchy list).

In other words, the container is the composition primitive that makes shells **share the same operating system instance**.

- **kernel**: this is *the singleton* primitive responsible for the coordination of all processes (or containers) in their access to the physical resources on one computing platform.

¹⁰Some compilers offer support for runtime configuration composition, via the concept of **thread-local storage**.

Process, container and kernel can conceptually be considered as hierarchical versions of the same type of resource coordination primitives. All of them can be **configured** by an application, at runtime via shell scripts, or at compile time via **kernel configuration settings**.

The information architects provide an architecture of activities and algorithms, and the software architects must then make sure that each activity executes correctly in a thread somewhere on a processor, irrespective of the timing and the number of times that execution is **preempted** by the operating system.

11.7.3 DOM model for activity deployment hierarchy

The entities and relations in this Section have already received significant modelling attention, for example in the **AADL** standard and supporting **software and tools**. Unfortunately, AADL (and existing alternatives) violates almost all “best practices” advocated in this document, such as: separation of mechanism and policy, inversion of control, or composability. This Section provides a **DOM meta model** that does conform to the composability and compositionality ambitions of this document. It starts with the DOM models that have been introduced before to represent the **hierarchies inside an activity**.

(TODO: tooling from the Web; **CRUD operations** on a DOM model; role of a **virtual DOM**; visualisation; diffing and streaming.)

11.7.4 DOM model for interaction streams

The **previous Section** introduced a DOM model for the *hierarchy* in activities. This Section covers the complementary software architecture model, namely that of the *heterarchy* of the communications between activities.

Not surprisingly, this model will not have much hierarchy in tree-structured *containment*, but consists mainly of *constraints* between the ports in the activities.

(TODO: classes of streams and their protocols (PubSub, RPC, SubmissionCompletion,...; coordination and configuration; model CRUD interaction.)

11.7.5 Pattern: composition of process, thread, activity and algorithm

The primitives a software architecture *has* to work with are determined by the *operating system* (OS) paradigm. That is set of design decisions to provide efficient access to the computational, storage and communication resources of computer hardware. (That set is surprisingly homogeneous over different OS providers.)

Previous Chapters in this document presented a (the...?) complementary paradigm to design a system in the form of an information architecture. The primitives there are functions, algorithms, programmes and activities.

This document advocates one particular **pattern** about *how to compose* both architectures:

- each process has $N > 1$ threads, where the “main thread” takes the responsibility to mediate all resources, in both the information and software architectures.
- each thread runs one event loop, who is the worker that executes the tasks required in all the activities.

- the decision about which activities the event loop has to execute at a given moment in time is realised with a *Finite State Machine* data structure.
- the event loop is also the OS-centric primitive to which the activities can outsource their “waiting” for the progress in each others’ programmes.
- each activity has $K \geq 1$ programmes, each having $L \geq 1$ algorithms.
- the decision about which parts of its algorithm(s) a programme makes available as tasks for the local event loop to execute at a given moment in time, is realised with a *Petri Net* data structure.

This is a [pattern](#) and not just a [best practice](#), because the outlined design allows many trade-offs still to be made, and is hence more than just an observed fact.

11.7.6 Mechanism: property graph implementation

(TODO: different design choices depending on deployment context: as a service in a database, or as a thread safe library in an event loop.)

11.7.7 Pattern: process composing multiple threads

This Section explains how to create a software process, according to this document’s generic design drivers of **model-based**, **composability** and **explainability**. The components in the composition of a process are: [threads](#), [stream buffers](#), [finite state machines](#), [Petri Nets](#), and [event loops](#).

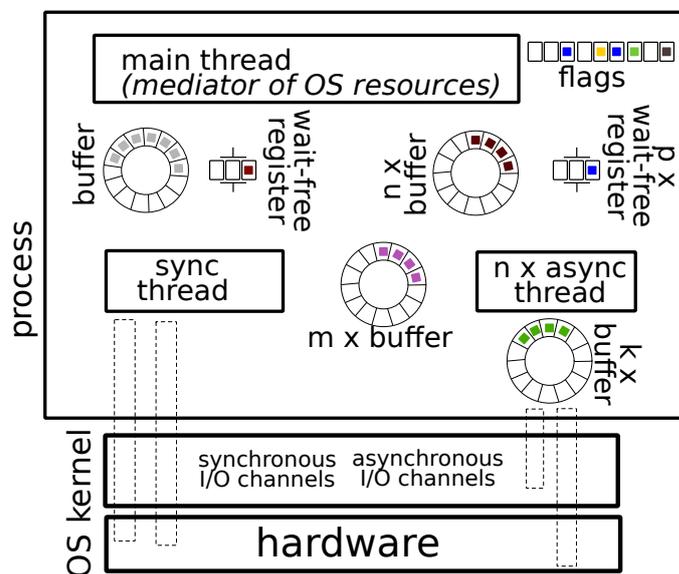


Figure 11.7: This process architecture is a pattern to let synchronous and asynchronous activities run in several threads: one *main*, one *synchronous thread*, and one or more *asynchronous threads*. All threads exchange their information via a composition of three primitives: **ring buffers** (when ordered “memory” is required), **registers** (when just the “most fresh” value of a variable matters), and **flags** (variables that can be written and read “atomically”). The number of threads and buffers, registers or flags need not be the same.

Design overview

Figure 11.7 sketches the **multi-threading architecture** that is common to the **deployment** of activities and **interaction channels** (“streams”) in an **information architecture**, in threads and buffers in a **software architecture**:

1. one **main** thread.

This is not a *choice* but a *constraint* imposed by the technology: starting a program from an operating system **shell** or **script** inevitably implies executing the **main** “thread” of the process, and there is only one such main thread. So, system designers don’t have to do anything special, it’s just there by default. Where they *do* have a choice is in deciding *which activities* the **main** thread will execute. This document let the **main** thread execute the activities in the threads, in their **deploying** state of the application’s [Life Cycle State Machine](#).

2. one **synchronous** thread, or “*real-time*” thread.

The **event loop** of this activity relies on the assumption that *none* of its functions will ever **block**. (That assumption need only hold when the activity is in the **running** state of its [Life Cycle State Machine](#).)

3. one or more **asynchronous** threads.

The role of each of these threads in the process is to serve as

- a stream interface towards activities than run in other processes (including device drivers for hardware, and local area networking).
- a coordinator for a set of other activities (via event processing on the event streams to and from these activities).
- a mediator for a set of other activities (by being the first consumer of their streams, able to decide that the activities need not process parts of their streams).

So, each of the **asynchronous** threads interfaces with one or more other activities, and is allowed to *block* while doing so. In order not to let the **synchronous** thread wait, a stream buffer between an asynchronous thread and the synchronous thread has received ample resources at deployment time.

4. one ring buffer (with status flags) for every stream.
5. a set of bitfields for each FSM and Petri Net.

The pattern can be adapted to the number and types of cores offered by the CPU hardware: it is possible to assign one synchronous thread to each of the cores, under the constraint that one does not deploy more than one such thread on that core.

Behaviour of the main thread

This document advocates to use the **main** thread exclusively to configure, monitor and mediate resources owned by the operating system:

- creating **threads**.
The **POSIX threads** standard and implementations provide a basis for this role.
- configuring their scheduling properties.
- creating the streams that buffer data between threads.
- configuring the “Ports” of the threads that must access the streams, as producer or as consumer.
- creating the finite state machines that coordinate the behaviour of each thread.
- configuring their initial states.

- operating-system owned communication such a **disk access**, **networking** and **I/O devices**. For example, via **standard streams** and **pipelines**, but also **environment variables** or the **main thread's command-line arguments**.

Behaviour of the synchronous thread

The *non-blocking* requirement on the *synchronous thread* must be realised *by design* (of both hardware and software):

- its communication needs with the hardware happens with technology that allow synchronous execution. For example, real-time networks like Ethercat or CAN, or memory mapped I/O.
- its interactions with other threads in the process runs always via stream buffers.
- it gets the **highest priority** from the operating system scheduler, and/or
- it is deployed on its **own private CPU core**.
- its **virtual address space** is locked into RAM (to prevent that memory from being paged to the **swap area**), via system calls like **mlock()**.
- the **interrupt** lines on the system are **masked**, except for the interrupts that the thread needs itself.
- by definition of the word, the “highest priority” can (or, rather, *should*) be given¹¹ to this one thread only, because if two threads are given the same numerical priority, only one of them can really be executed first.

Support of operating systems

Operating systems offer support for **resource reservations**, of CPU, **memory**, and I/O. For example, the Linux operating systems offers software support in the form of, for example, **mmap**, **mlock()**, **cgroups**, or **systemd**.

Operating systems differ in the set of software entities that are **owned** by a process on the one hand, and by each of its threads on the other hand. **This document** gives a lot of concrete details, more in particular about the differences between **POSIX** and **Linux** threads. Especially the wide variety in **unique identifiers** (“IDs”) in the (non-formalized) meta model of the process is an indication of its central role in any software architecture.

Implementation design: the main() function

statically allocated memory

main() function:

```
#include <stdlib.h>
int main(int argc, char **argv, char **envp);
{ ... }
```

Start in **creating** state:

- check statically allocated resources for all threads.

Transition to **configuring_resources** state:

¹¹Here is the **POSIX** way of configuring priorities.

- setting initial state of all threads' LCSMs.
- initializing all streams: pointers and status flags.

Configuration options:

- **#define macros** in the source code.
- **command line** arguments (via the **argv** pointers).
- **environment variables** (via the **envp** pointers).
- **configuration files**.
- **stdin**.

Configurability options

(TODO: it is a best practice to strive for software implementations with high levels of **configurability**, e.g., compile time configuration of functions, deploy-time configuration of binaries, start-up time configuration of processes, and runtime configuration of activities.)

Policy: mediation on shared resources

The **mediator** can rightfully be called a “software pattern”, because there exist already various realisations, with mature and large-scale application track records. Here are some of the common realisations in the domain of ICT infrastructure:

- **bandwidth throttling**.
- **CPU throttling**.
- **process throttling**.
- **garbage collection**.
- **memory pool**.
- **thread pool**.
- *isolation of runtime*: all the runtime's data are copied, such as the heap, calling stack and garbage collection, with a different *context* for each *application activity*.
The application's “shared” and “global” data are copied for each of a number of concurrent activities. For example, **V8:Isolate** and **V8:Context** in the **Chrome V8** runtime engine. The isolation and context pattern is in itself *not sufficient* for *thread-safe* execution!)

(TODO: best practice: every shared resource needs a mediator activity for **resource management**.)

11.8 Best and bad practices

This Section discusses the **composability** and **compositionality** aspects of commonly used architectural choices.

11.8.1 Best practices

- the *single-writer principle* helps to make (i) **atomic mutation semantics** easier to realise, and (ii) more efficient to execute, by reducing **cache misses**. Indeed, a data item is owned by a single **execution context** for all **mutations**.

- **lock-free and wait-free** algorithms for data *sharing* [4] help to *avoid* the involvement of the operating system, such that applications have a larger impact on their own behaviour.
- **garbage collection**: the decision making about what to do with a full or an empty stream buffer is best done in a *mediator activity* that is owned by the ring buffer, instead of by producer and consumers themselves; for the simple reason that only then ownership of the data is unambiguous. For example, in tracing, audio processing or control problems, the **common policy** is to overwrite *part of* the oldest data at the producer’s side with newly arrived data, even when the consumer has not yet freed up the buffer array part that it occupies. Other policies are:
 - *discard*: drop the newly available data chunks.
 - *compaction*: the data in the buffer that has been produced but not yet consumed (or rather, not yet *claimed* by a consumer) is reduced, according to an application-specific policy rule.
 - *negotiation*: the amount of data per chunk is reduced, so more data can fit in the same stream. For example, **WebRTC** follows this approach.
 - *clear* the whole buffer, because the *completeness* of the stream is not guaranteed anymore.
 - *block* the activity until the buffer is not full or empty anymore.

All policies can be composed. This use case can also be dealt with without giving the stream buffer its own first-class activity: the producer is owns its side of the buffer, so it can *execute* the mediation actions itself.

- **Safety PLC** pattern: (TODO: separate component (function, activity, thread, computer,...) that checks a number of flags, heartbeats, time outs,..., at higher speed and lower latency than the real-time control component(s), and that can shut down the application independently. For example, by cutting power to the energy-introducing parts; and maybe then “rebooting” the failed control components.)

11.8.2 Framework plug-ins (bad) versus library composition (good)

Frameworks are one of the most popular ways towards digital platforms, for some good reasons: *code reuse*, *freedom from choice*, *best practice implementations*, and *tooling*. However, frameworks inevitably make trade-offs towards *usability* within a particular domain, **erroneously assuming** that “*easy*” will imply “*simple*”.

Composability in a framework is typically provided via so-called *plug-in* interfaces: the framework provides several places in its code base where developers can *register* their own functions, that will later be called by the framework’s *runtime engine* at the “right” time. This mediator pattern implementation works fine, as long as the framework can provide its users with all services they need. However, frameworks are typically poorly composable themselves with other frameworks or components, because:

- their plug-in interface offers only one single level of composition hierarchy. It is then not possible for developers to introduce their own functions to couple “state” at two different plug-in interfaces.
- a framework’s runtime engine typically expects that plug-ins adapt to the framework’s policies and protocols, and not the other way around. (In other words, frameworks often consider themselves as an “endpoint of integration”.) So, it is for example not possible

to configure the runtime engine to share resources with non-framework activities.

- one major cause is that their runtime’s *event loop* is not a user-accessible implementation primitive. Instead, they opt for major “easy” hard-coded policies, such as:
 - dedicating one whole I/O channel to each *port* instead of multiplexing several channels on one single port.
 - no access to the internal multi-threading architecture. (*If* there is one, to start with.)
 - lock-in into one single programming language, forcing single-language, compile time only, composition.
 - the lack of explicit support for user-accessible mediation and ownership.
- the configuration of computations is forced to be done *in* a plugged-in computation, instead of in the component for which the plug-in provides functionalities.
- similarly for the coordination between several plug-ins: the framework forces each plug-in to make the decision that its behaviour has to change, or let the framework do this. This prevents the framework-independent behavioural coordination *between* plug-ins.
- introducing “easy” coordination instruments to let a plug-in choose its own *priority*, while such a priority is *not* a *property* of a component, but an *attribute* given by the *process* that composes several components.

The more composable approach is to replace the runtime engine part of the framework, and *generate* an application-specific one, by (i) composition of functions and data from pure libraries, and (ii) generating the code for the application’s runtime, starting from composable implementations of software architecture patterns. Examples of the latter are the event loops and the inter-process communication patterns, for which framework runtimes often have made static, immutable choices.

11.8.3 Bad practices in robustness

The challenge of the software architects is to balance the *software* and *hardware performance* (which is a loosely defined composition of *efficiency*, *efficacy*, and *effectiveness*), with the **productivity** of the *developers* (which is loosely defined as realising a predictable system quality with predictable effort). One major performance challenge is to implement conceptually perfect information processing capabilities with imperfect resources available on computers. Here are some examples of system designs that turn out to be bad practice, because they neglect particular disturbances to performance and predictability:

- the *infinite data extension* of “time series” data streams, while a computer has only finite and/or non-**shared memory** resources.
- the *infinitely fast interaction* between activities, while a computer has only finite resources for computation and communication.
- the *perfect abstraction of an activity*, while computer-implemented architectures must deal with the large *cost vs performance* variations presented by the building blocks offered by **operating system** and **hardware**.
- the *perfect semantic consistency* of information exchange, while **programming languages**, **libraries** and **frameworks** often have different semantic interpretations of data structures and functions.
- **resource contention**, **concurrency**, and *end-to-end latencies* caused by multi-core/multi-computer execution contexts.
- let threads decide themselves about their **priority** and **processor affinity**.

These decisions must rather be made at a system level where the trade-off between *several* threads can be made, in the context of the *full* application.

- let the “event loop” be realized by the operating system.
That operating system does not know anything about the application’s performance indicators, so it should not be asked to schedule the application’s threads. Instead, that responsibility is to be taken in the application’s event loop(s): only there, the right decisions can be made about when activities can be pre-empted, and by which other activities.

11.8.4 Bad practices in resource locking

The sample code in Table 11.1 combines three bad practices of using locks:

- *multiple locks around the same critical section*: (TODO)
- *nesting locks*: (TODO)
- *blocking operations inside lock*: (TODO)

```
struct { int a; int b; } dataA;
struct { int a; int b; } dataB;
...
mutex_lock(mA);
mutex_lock(mB);
f1(&dataA);
f2(&dataB);
printf("A: %d, B: %d \n",dataA.a,dataB.b);
mutex_unlock(mA);
mutex_lunock(mB);
```

Table 11.1: Code example combining three bad practices in using locks.

11.8.5 Bad practices in communication

- to use only the **publish-subscribe** communication pattern. Events often profit from a *broadcasting* policy, and queries about models require one-on-one *dialogues* via **submission-completion** queues. Neither of those use cases is supported well with publish-subscribe.
- to neglect the **CAP** and **PACELC** theorems, that state that:
 - the assumptions of Consistency, Availability, and Partition tolerance can not be satisfied at the same time.
 - Consistency and Latency are contradicting requirements.

These theorems, and the **exactly-once semantics** of communication, are major *constraints* for realising consistency of data exchanged between asynchronous activities. Or rather, once system designers are aware of the difficulty to realise all aspects involved, they will look for architectures that are **robust** against (combinations of) communication disturbances.

- to neglect the **fallacies of distributed computing**, *even* between processes on the same computer.
- to communicate *state* information back and forth between distributed components, even when it is possible to deploy all the components’ functionalities into the same process

and (hence) to store the shared state in a shared data structure.

11.9 Mechanism: programming language operators

11.9.1 Async/await

(TODO: **async/await** programming language construct to make some forms of asynchronous programming look very much like synchronous programming; discussion on where it fits in the meta models of activities, event loops, and streams. explain problem with cancelling of a function when it is already waiting on a promise, and refer to Submission-Completion as a *task queue* mechanism with a better support for task cancelling; discussion on when to use it and when not; examples needed, for example in graph traversal solvers.)

11.9.2 Iterator

(TODO: **iterator** helps to separate the structure of the data from the operations, in a declarative way.)

11.9.3 Maybe

(TODO: **Maybe** type, to represent not-yet-known data.)

11.9.4 Memory barrier

(TODO: **memory barrier**, to order access to fast memory.)

11.9.5 Atomic and lockfree operators

Replace every synchronous “mutex” area with an asynchronous stream, or with **seqlocks**.

Memory barrier operations need to be inserted,

The problem can also be solved by introducing a small ring buffer with lock+value structures, because then the write/read of the value *depends* on the sequence number and compilers will not **reorder instructions**.

A single-writer stream can be done without locks; multi-writer is seldom needed because it can be replaced by the same consumer for all of the producers in a single writer-reader form, and that consumer makes one or more new composed stream.

Read-Copy-Update is another approach that trades off locking for more copies of written data.

11.9.6 Mechanism: Conflict-Free Replicated Data Type (CRDT)

(TODO: different activities can concurrently change data structures, and the changes are merged and distributed automatically without the need for a central server which “owns” the data structure; only some data structures have the **CRDT** property.)

11.9.7 Mechanism: immutable data type

(TODO: concurrency becomes a lot easier to make predictable if any data that is created new will never have to change anymore, because *reading* of data can never lead to inconsistencies; explain where this particular type of CRDT makes sense in system architectures, and when (not) to use it. Aspects of *garbage collection* and *compaction*.)

11.10 Modelling languages: mature implementations & tools

The meta and meta meta models of all previous Chapters must, sooner or later, be encoded in concrete “programming” languages. For some of the representations, some such concrete encoding languages have matured into world-wide, vendor-independent standards. Most formalizations have been developed for human developers only, and not for [higher-order](#) reasoning by computers. This Section gives some domain and vendor neutral examples, for which mature implementations and tool support exist.

11.10.1 QUDT and UCUM

The *quantity-unit-dimension-type* meta model has already been formalized several times, for example in the **QUDT** initiative [104]. This ontology is nicely composable with the levels of abstraction hierarchy:

- mathematical and abstract data type representations: the **T**(ype) and **D**(imension) parts in QUDT are *linked* as semantic tags to each “type” of thing that one wants to represent. The *type* in QUDT is the same as the type in the mathematical and abstract data type representations. For example, the distance between two points in space, or **Maxwell’s equations**. The *dimension* in QUDT adds annotations to (parts of) types such as **length**, **time**, or **voltage**.
- data types and digital representations: as soon as one makes a concrete choice of how to represent things concretely on computers, one automatically introduces the **Q**(quantity) and the (physical)**U**(nit) parts of QUDT.
- **T** and **D** always come together at the same level of abstraction, as do **Q** and **U**.

The **W3C** and the **Eclipse** eco-system promote a model similar to QUDT: **UCUM** (the *Unified Code for Units of Measure*).

11.10.2 JSON and JSON-Schema

JSON-Schema is a schema to formally describe elements and constraints over a **JavaScript Object Notation** (JSON) document. Instead of relying on an external DSL, a JSON-Schema is also defined as a JSON document. In turn, the JSON-schema must conform to a meta-schema, which is also defined over a JSON document. A concrete example is provided in [Figure 11.8](#).

JSON-Schema is considered a composable approach, since (i) JSON supports associative array (only strings are accepted as keys) and (ii) JSON-Schema supports JSON Pointers (RFC 6901) to reference (part of) other JSON documents, but also objects within the document itself. This allows to compose a schema specification from existing ones, and to refer only to

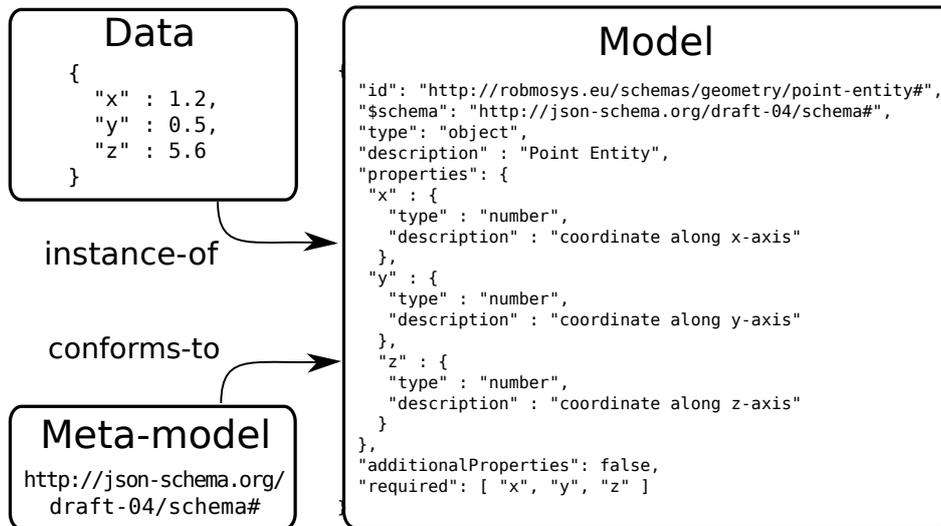


Figure 11.8: A valid data instance of a JSON-Schema Model representing three coordinates. The schema includes few constraints on the data structure, such as the values required for the validation of the JSON document. Moreover, the schema **conforms-to** a specific meta-model of JSON-Schema (draft-04).

some specific definitions. JSON-Schema is used in web-technologies and it is very flexible in terms of requirements needed to be integrated in an application. However, it is verbose with respect to other alternatives, as well as not efficient in terms of number of bytes exchanged with respect to the informative content of a message. In fact, JSON-Schema does not provide native primitives to specify hardware-specific encodings of the data values. However, it is possible to compose a schema that cover that roles, in case that the backend component can deal with them. Figure 11.8 shows a example of a typical workflow with JSON-Schema. As a final remark, JSON-Schema is not limited to describe JSON documents, but also language-dependent datatypes.

11.10.3 JSON-LD

JSON-LD, “*JSON for Linked Data*”

(TODO: outline of features; how to use it to represent property graphs, `Semantic_ID` (“@context” for meta model *connections*, “@type” for meta model *conforms-to* relations, and “@id” for model ID; how to formulate queries as graphs on the property graph, with *given* and *queried* property parameter values; how to do graph traversals to answer queries.)

11.10.4 RDF1.1

RDF:

(TODO: relation to JSON-LD: minor difference in flexibility *to name* graphs; lack of keywords to represent `Semantic_ID` natively.)

11.10.5 Abstract Syntax Notation One (ASN.1)

ASN.1 (“*Abstract Syntax Notation One*” is an IDL to define data structures in a standard, code-agnostic form, enabling the expressivity required to realise efficient cross-platform serialisation and deserialisation. It has its origins in telecommunication, in the early 1980s. ASN.1 models can be collected into “modules”, which can be composed between themselves as well. This feature of the ASN.1 language allows better composability and re-usability of existing models. However, ASN.1 does not provide any facility of self-description, if not by means of the naming schema used by the compiler to generate a data type in the target programming language. Originally developed in 1984 and standardised in 1990, ASN.1 is widely adopted in telecommunication domain, including in encryption protocols, e.g., in the HTTPS certificates (X.509 protocol), VoIP services and more. Moreover, ASN.1 is also used in the aerospace domain for safe-critical applications, including robotics applications. For example, an ASN.1 compiler is included in *The ASSERT Set of Tools for Engineering* (TASTE), a component-based framework developed by the European Space Agency (ESA). Several compilers exist, targeting to different host programming languages, including C/C++, Python and Ada.

11.10.6 Hierarchical Data Format — HDF5

HDF5 is another internationally standardized format, with a maturity similar to QUDT, offering meta model, models and reference implementations for all sorts of *abstract data type* representations and transformations.

11.10.7 FlatBuffers, Protocol Buffers, Apache Arrow

FlatBuffers and **Protocol Buffers** are more recent but well-supported alternatives. Their designs have been optimized for efficient runtime data processing and messaging and *self-description*, but not really for knowledge representation and reasoning.

A promising recent development in the direction of improving these deficiencies is the **Apache Arrow** project; it has formal schemas for all binary data structures, pays very special attention to support efficient random access to all binary data, *and* has tooling to convert the former automatically to code that realises the latter.

11.10.8 Common Trace Format — CTF

The **Common Trace Format** is a binary trace format, that originates from the Linux kernel, in the context of *event streaming* and *processing*.

11.10.9 BLAS, LAPACK

BLAS and **LAPACK** are other mature ecosystems of models, tools and software, to provide the *linear algebra* aspects of representations and operations in geometry.

11.10.10 DFDL

DFDL (Data Format Description Language, “Daffodil”): (TODO: XML, less developed for scientific abstract data types like multi-dimensional arrays;)

11.10.11 XML Schemas

Similarly to JSON-Schema, XML Schemas (e.g., XSD) are models that formally describe the structure of a Extensible Markup Language (XML) document. XML schemas are very popular in web-oriented application and ontology description, but also in tooling and hardware configurations (e.g., the *EtherCAT XML Device Description*).

11.10.12 PROV-O provenance ontology

(TODO: W3C's [PROV-O: The PROV Ontology](#).)

11.10.13 Functional Mockup Interface (FMI)

FMI is an industry-driven standard for the representation and the **co-simulation** of dynamical systems. The host language is XML. The standard tries to be self-contained, which means it repeats a lot of more generic concepts and standards, such as the representation of physical units and types (covered by QUDT), or the digital representation of values (8, 16, 32 or 64 bit unsigned integers, etc., as covered by programming languages). It has fallen into two major traps of “bad practice” modelling, i.e., *inheritance instead of composition* and *properties instead of attributes*. This gives rise to some complementary negative effects:

- *reification* is not built in, hence there is no support to represent higher-order relations (constraints, tolerances, . . .) on parts of a model.
- a property of a **CoSimulation** type is, for example `canReturnEarlyAfterIntermediateUpdate`, but that is a behavioural property of an activity that executes a **CoSimulation**, not of its model.
- software tools to support the standard must be huge, because the standard is huge.
- slow innovation, because of the many internal model dependencies. For example, the standard is not adapted in mainstream robotics systems, because the latter have a focus on *runtime* execution, and less on *co-simulation*. Although the overlap between both areas is huge, the co-simulation dependencies are very “hard-baked” into the standard, making it less appropriate for reuse in domains with another focus.

Chapter 12

Skill architectures for eight-wheel mobile robots

Mobile robots must have a high degree of actuation redundancy, on a passively backdrivable mechanical platform, to deliver efficient trade-offs between task performance and physical safety of humans, environment objects and themselves. This Chapter illustrates robotic systems design by means of the example of a very over-actuated platform, namely one with **eight wheels**, deployed in four groups of **two wheels** sharing the same **axle**.

This Chapter presents the application context of an overactuated eight-wheel mobile platform (Fig. 12.1). The focus is on the geometric, quasi-static and dynamic relations that link the forces on the platform to the torques on the wheels, i.e., traction forces in each two-wheel swivel-caster drive unit. The addition focus is on a platform built as a *composition* of several such units. The chosen platform is overkill for some applications, but it contains all relevant modelling and control aspects, including overactuation, and (hence) redundancy resolution, passive joints, and passive platform backdrivability. *Control* of the platform takes place at, at least, six levels of abstraction:

- the selection of each platform for a particular logistics mission in the organisation.
- the navigation through an indoor environment with areas with different navigation constraints, as in Fig. 5.7.
- the local motion of the platform that realises the navigation, as in Fig. 5.9.
- the motion of the four two-wheel units inside the platform that it uses for its proprioceptive motion control.
- the torque/speed control of the individual wheels inside each unit.
- the electrical-mechanical energy transformation control between mains (or battery) and wheel, via power amplifiers and battery management control.

Controllers at all levels interact in multiple ways, as represented in the **task meta model**:

- the transportation control must decide where the platform must move through the environment and what type of control mode is needed.
- the “force” and “motion” control of the platform must decide about how to divide the platform driving force over the various two-wheel units, which act as “force” driver resources.
- the control of each two-wheel unit must, in turn, decide about how to actuate both

individual wheels taking into account the energy efficiency of their torque generation and the friction constraints of their tyres.

12.1 The eight-wheel drive mobile platform

Figure 12.1 sketches the ideal kinematic chain model of an overactuated mobile robot platform. It has one rigid body¹ as a “platform”, and four identical driving units with each two actuated wheels in a symmetric configuration around a **swivel caster** connection to the platform. The advantage of this design is its *passive backdrivability*: when all power falls away, the platform can still be moved by a human, in all directions and with moderate force. The disadvantages are (i) more complex mechatronics (mechanics, electronics and control), and (ii) the need for active compensation when driving on laterally inclined surfaces.

Assumption: throughout this Chapter, the ground is assumed to be *horizontal*, and the wheel axles too. This means that a non-actuated robot is in a stable equilibrium, and, hence, *no control* is a **passively safe** mode of **failure**.

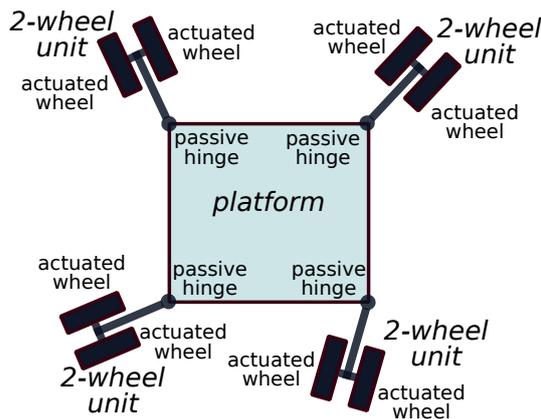


Figure 12.1: Kinematic model of an overactuated mobile robot: eight actuated wheels connected, in so-called 2WD (“two-wheel drive”) pairs, to a rigid platform via passive revolute joints with a caster offset. (The magnitude of the offset is much smaller than depicted, in real-world implementations.)

The mobile platform is a representative of a *complicated* robot kinematic chain model, because it embodies all of the following non-mainstream properties:

- *underactuated*: the hinges with which each 2WD unit is connected to the platform have no actuator. In principle, they also might have no encoder, so that their orientation must be *estimated*.
- *redundant*: the platform has *three* motion degrees of freedom (two for translation in the ground plane, and one for rotation), but *eight* actuators.
- *floating base*: no part of the robot that is rigidly fixed to the environment, so the robot has no “natural” origin in space.
- *non-holonomically constrained*: the wheels have rolling constraints, which are instantaneous acceleration-level constraints that can not be “integrated” into position constraints.

¹This is an idealisation, since the mechanical platform will have some passive degrees of freedom to allow all wheel units to touch the ground at all times, and under comparable load conditions.

- *N-DOF joints*: the platform is actuated by four forces, each generated by two actuated wheels *in parallel*, hence, there exist no *spanning trees* in the kinematic structure that makes solving the kinematics and dynamics linearizable into 1-DOF problems: the force distribution over both wheels must be solved together as a 2-DOF problem.
- *Cartesian friction and slippage*: the forces in the actuators are not guaranteed to be transmitted to the platform in full, since the wheels can slip, and they also lose force transmission efficiency via friction.
- *partially observed*: because of slippage, the sensors on the actuators are not sufficient to measure, or even estimate, the full motion state of the robot. Sensor fusion with extra sensors, like **Inertial Measurement Units** or cameras, must be introduced.

12.2 Topological navigation and metric motion tasks

The *application context* of this document is the usage of overactuated mobile platforms in indoor environments. Figure 12.2 sketches a typical area in such a context, with “corridors” coming together in “intersections”, and having “doors” to “rooms” and “elevators”. There are several complementary types of “tasks” (and, hence, also corresponding types of “control”, “perception”, “world modelling” and “monitoring”), depending on which *level of abstraction* that one looks at the available fleet of robots: *mission*, *navigation*, *platform motion*, *2WD motion* and *wheel motion*.²

The logistics unit in the organisation has the *mission* to provide the right goods at the right time at the right place; to reach that goal, it must decide (“control”) which platforms to deploy, where and when, and in what type of configuration. Each individual mobile platform has the *task* to *navigate* through such environments. Maps of the environments contain *no explicit motion trajectories* for any particular type of robots, but only the *declarative constraints* that *any* moving robot should respect when it wants to navigate through the environment without colliding into it. This document uses the term “*navigation*” to denote a sequence of connected “semantic areas” that the robot traverses,³ and the term “*motion*” to denote the (timed) “geometrical path” that the robot platform travels through.⁴ The term “*motion*” has itself *three complementary* meanings, depending on whether it pertains to (i) the platform as one rigid body, (ii) one single of the four 2WD units attached to a platform, or (iii) each individual wheel in one such 2WD unit.

The mobile platforms have *to control*, both, navigation and motion, by making the platform/units/wheels move in such a way that the desired/expected navigation or motion is realised. Navigation control pertains to the activity to decide which areas a robot should traverse; this is a *symbolic constraint satisfaction* problem that deals with topological navigation. The metric level of abstraction is on the *motion control*, formulated as a **hybrid constrained optimization** problem in the *continuous* (metric) domains of time, space, force, energy,...

A mobile platform can use three complementary types of *motion* control (at all the motion control levels of the **platform**, the **2WD units**, and the individual **wheels**), depending on

²This document does not consider the task levels “below”, such as battery control and power convertor control, not the one “above”, such as the decisions about investment and maintenance in robotic technology, or about their integration into the overall logistics operations.

³Hence, this is a *topological* abstraction of the robots’ motions.

⁴Which is the *metric* representation of the robots’ motions.

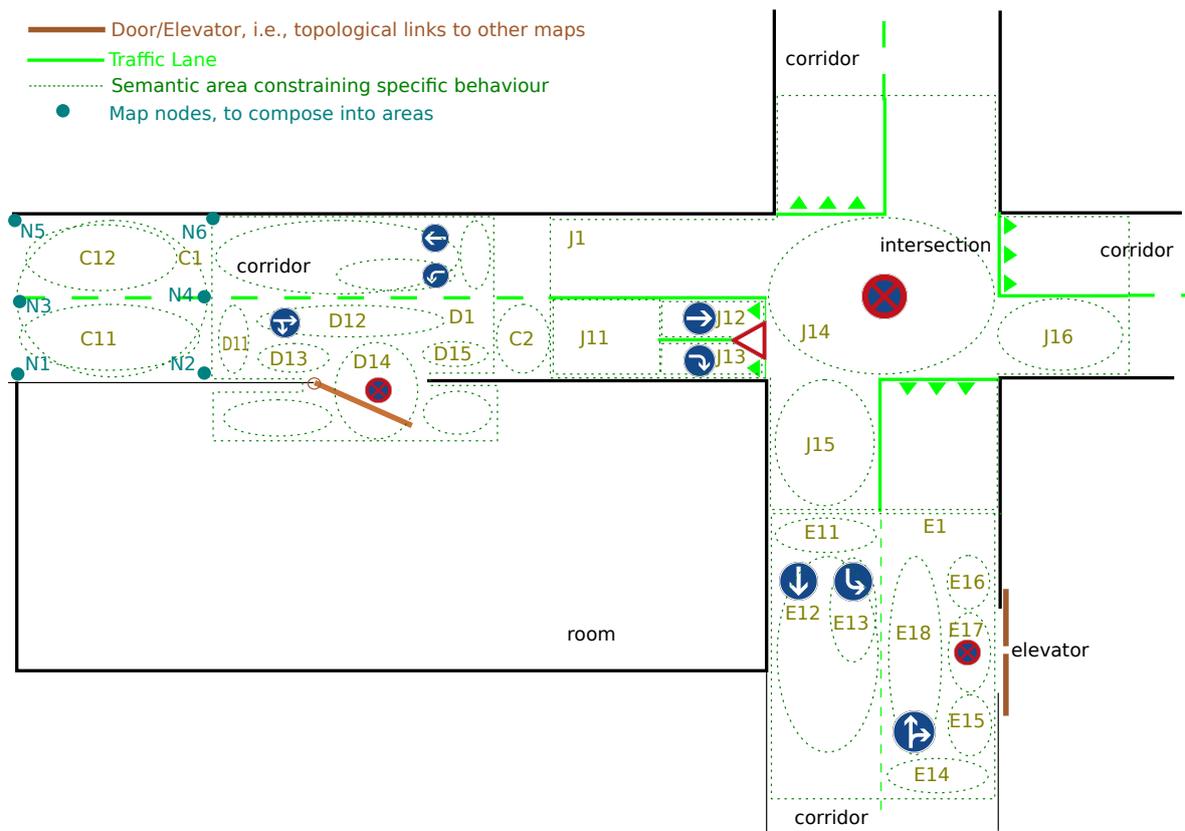


Figure 12.2: A typical indoor environment to deploy mobile platforms in, with a lot of (virtual) *semantic tags* (inspired by real-world **traffic signs and signals**) to describe the *intended* navigation usage of parts of the area (such as “traffic” lanes and stopping, halting or waiting areas), and of objects (such as doors or elevators). The drawing does not show *perception* and *action* tags, which are both necessary in a robotics application context.

what are the *inputs* of the navigation task specification and the *measures* with which the *progress* (or “*utility*”) of the task execution will be evaluated:

- “*force*” control: the *inputs* are desired force and torque applied somewhere on the platform, together with a target area or tube within which the control must move the platform. The output of the control are desired forces to be created by each of, respectively, the two-wheel units, or the individual wheels. The progress is measured by how well the resulting motion fits in the target tube.
- “*motion*” control: the *input* is the desired “motion”, that is, acceleration, and/or velocity, and/or position of some reference points on the platform. The output of the control are, either, desired forces, or desired motion, to be realised by each of, respectively, the two-wheel units, or the individual wheels. The *progress* is measured as (some sort of time-averaged) error between desired and actual motion.

- “*impedance*” control: the *inputs* are desired velocity, and/or position, and the stiffness and/or damping that a “disturbing agent” will feel when physically interacting with the system. The *output* of the control are desired forces to be realised by each of, respectively, the two-wheel units, or the individual wheel. The *progress* is measured as (some sort of time-averaged) error between desired and actual motion.

Of course, every task comes with specific extras on top of the mentioned control inputs and outputs:

- one or more *cost* functions, e.g., energy consumption, time, distance, number of changes in control modes, smoothness of the motion, maximum actuation efforts, etc.;
- one or more *risks*, e.g., how big would be the extra cost incurred by the system if the current task specification can not be realised;
- *constraints* on the admissible values of some parameters;
- *tolerances* on these parameters (and on their constraints) that represent the solutions that are “good enough”.

12.3 Platform navigation and motion modes

The choice of a particular control approach is represented by a set of *modes* that a mobile platform can be in:

- *cruising*: the robot traverses an area that has no declarative constraints (“semantic tags”) that can result in “discrete” or “large” transitions in motion speed or heading.
The appropriate continuous control is “force” control to keep the platform within the area’s *declarative constraints*, and the discrete control to decide whether or not to bring some 2WD units towards *pushing* configurations;
- *maneuvering*: the robot moves within one particular semantic area, with the goal to reach one specific sub-area (with a “large” tolerance), and this is expected to result in discrete transitions in motion speed or heading.
The appropriate continuous control is impedance control, with discrete wheel configuration switching control.
- *positioning*: the robot must reach a specific geometric *position* (that is, an “area” with “small” geometric tolerance) within one particular semantic area.
The appropriate control is motion control, with discrete wheel configuration switching control.
- *2WD configuring*: no control is active at the *platform* level, since the state of the 2WD units has not yet been established. Hence, the control activity is first to be defined and realised at the “*joint*” level, that is, that of the individual 2WD units.
- *wheel configuring*: before any motion can be controlled, one first has to configure the controllers of the *individual wheels*.

The order in which these modes have been presented is not a coincidence: it is a hypothesis in this document that the order reflects a *strict hierarchy*: the *nominal* task in a “higher” level can be extended by one or more “lower” level tasks that, together, represent a *pre-emption* of the nominal higher-level execution. For example, a *cruising* task can be refined with an *overtaking maneuver*, that consist of: (i) maneuvering from one “lane” to a neighbouring lane, (ii) cruising in that lane till one has moved “sufficiently” further than the overtaken obstacle, and (iii) maneuvering back to the original lane. The “pre-emption task” should *not replace* the original cruising, but it puts the three mentioned maneuvers on the “*stack*” of the robot controller, so that the original motion can be *resumed* at any time that the current (and hence also all of the remaining) pre-emption maneuvers are not needed anymore.

12.4 2WD force transmission

This Section zooms in on (the quasi-static model of) one two-wheel unit connected to the platform via a swivel caster. Figure 12.3 shows the pushing and pulling forces that can be transmitted (quasi-statically) from the wheel traction forces to the platform. The model assumes that both wheels are identical, that they can provide a maximal force of F^n , that both wheels are a distance $2 \times r$ apart, and that they are a distance l away from the revolute joint connecting the wheel unit to the platform (represented by the black circle in the middle of the figure).

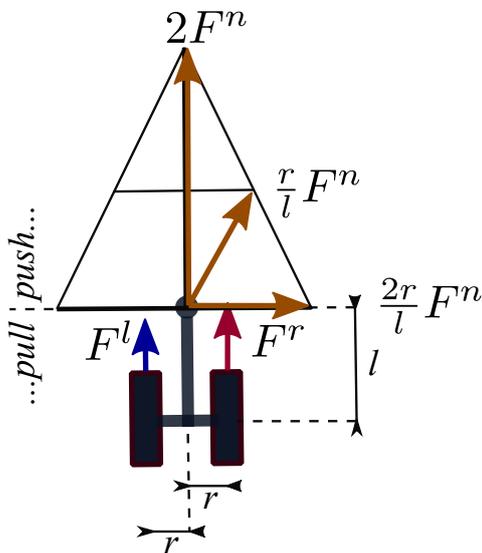


Figure 12.3: The forces transmitted between the traction forces on the wheels to platform, which is connected through a frictionless revolute joint that transmits only forces and no torques. The “triangle” spans the domain of all possible forces that can be exerted on the platform by varying the wheel forces. When one of the wheel forces is larger in magnitude than the other one, and opposite in direction, the wheels pull the platform, otherwise they push it. The three orange forces on the drawing represent the tangential force components resulting from the cases in which $F^l = F^n$ and F^r equals, respectively, F^n (vertical force), zero (middle force), and $-F^n$ (bottom of the triangle). Only the triangle in the “push” direction is shown, that is, when the resultant of F^l and F^r is positive in the direction from the wheels towards the platform.

Because of the revolute joint, the wheels cannot transmit a *moment* to the platform, but only pure forces when represented with respect to the center of the joint. The maximal force in the direction of the wheels is $2 \times F^n$; this forms the top of the triangle. If the magnitude of both wheel forces become smaller, but still equal in magnitude, the resulting force on the platform is situated lower on the vertical centre line of the triangle. If their magnitudes are not equal, a

force component off this centre line is generated, because it must offset the moment generated by the unbalanced wheel forces. When the left wheel force remains at F^n and the right wheel force decreases, the resulting platform forces generate the right-hand border of the triangle. When the right wheel force is $-F^n$, both wheels create a pure moment of $2rF^n$ around the centre point in between both wheels, where r is the distance between that centre point and the point where a wheel force is applied to the ground. At the revolute joint on the platform, this pure moment results in a pure force of $\frac{2r}{l}F^n$, in the horizontal direction. When the right wheel force is zero, the wheel torque is half the value of the previous case, so the horizontal offset force is $\frac{r}{l}F^n$.

The force transmission is inversely proportional to the *castor offset* length l . When that length goes to zero, the transmission goes to infinity. Geometrically speaking, the side of the triangle becomes more and more horizontal, and for $l = 0$, it becomes impossible to generate a horizontal force component by the two wheel forces. This limit case corresponds to the traditional **differential wheeled robot** kinematics.

Magic numbers:

- the geometric parameters of the model in Fig. 12.3;
- the geometry of the wheel, that is, the wheel diameter;
- the mechanical dynamics of the wheel, that is, mostly, its mass, its rotational inertia, and the friction of the wheel bearing.
- the electrical dynamics of the wheel, that is, mostly, its torque and velocity constants, torque-speed efficiency curve, and its maximum torque and current.
- the width of the contact patch;
- the friction of the wheel, both rolling and sliding;
- the slip properties, but these are a composed property of the interaction of the wheel with the material of the ground surface.

The mapping from two wheel torques to the force transmitted to the platform via a passive revolute is bijective, and it has no singularities. Hence, no “pseudo-inverse” or “redundancy” choices need to be made, introducing no extra parameters. Of course, there are saturation limits to torque/force transmissions, due to friction at the wheels and actuation limits in the motor.

12.5 2WD efficiency of force transmission

The triangle in Fig. 12.3 also helps to discuss the **efficiency** of the force transmission from wheel motors to platform motion. The “efficiency” η^f is being defined here as the *ratio* between the *absolute magnitudes* of (i) the platform force F_i^p that wheel unit i contributes to the overall force on the platform, and (ii) the resultant force vectors F_i^r and F_i^l of the forces

in the right and left wheels of the unit:⁵

$$\eta_i^f = \frac{\|F_i^p\|}{\|F_i^r\| + \|F_i^l\|}. \quad (12.1)$$

The triangle shows that the wheel forces can generate forces on the platform whose *magnitudes* are larger than the magnitudes of the wheel forces; but there are some “hidden” costs behind this observation:

- *instantaneous power* must be conserved.

Since the *work*, or *power*, generated by applying the same forces for a small distance or time should be conserved, larger forces result in smaller motions on the platform than on the wheels. That is because the force on the platform feels a higher instantaneous inertia than just the wheel unit inertia. So, nothing is magically gained here with respect to the propulsion of the platform. On the contrary: as illustrated in Fig. 12.4, the efficiency of the transmission goes down with increasing angle between the wheel centre axis and the force vector on the platform created by the wheel unit.

- the *overall energy* required to move the wheel units must be considered over non-instantaneous trajectories.

Since the forces generated at the wheels accelerate, both, the platform and the wheel units, any force on a wheel unit that “points away” from the desired motion of the platform will start accelerating the wheel in the “wrong” direction. Sometime later, the energy put into this acceleration will have to be counteracted with energy to accelerate the wheel in the “right” direction.

- the *friction* and acceleration forces can fight each other.

Since several wheel units are driving the platform at the same time, and since uncontrolled external forces can be acting on the platform at any time, the forces *generated* at one wheel unit are not the only ones that *move* the unit. The result is that *wheel-twisting* sliding friction is inevitable at the point of contact between the wheel and the ground, and hence energy is lost in this way.

- *traction slip* can occur.

The driving forces on a wheel can become larger than the friction between wheel and ground surface.

From the discussion above, it is clear that there are multiple definitions of “efficiency”. For example, some slippage in the traction wheels decreases *energy consumption* efficiency, but can improve *motion progress* efficiency.⁶

In the “push” case, all forces required to realise the platform force are also accelerating the wheels in the direction of further alignment with the desired platform force; in the “pull”

⁵An *energy-based* efficiency ratio can only be defined when a *full dynamic model* of all hardware components is available.

⁶Indeed, the fact that a wheel does not slip implies that it has the potential to provide more traction, so one can only be sure to have obtained maximum wheel traction if a “small” amount of slippage occurs.

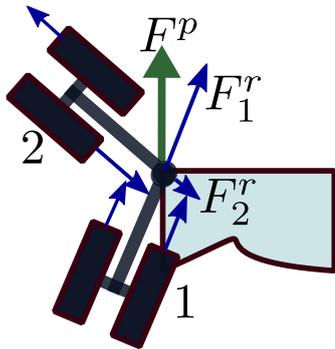


Figure 12.4: The angle between the wheel centre line and the line of the generated force on the platform determines the “efficiency” of the transmission of forces at the wheels to the resulting force on the platform.

case, the wheel accelerations are in the opposite direction. The obvious and most optimal case of 100% efficiency is not depicted, and that is the case in which the centre line of the wheels is parallel to the platform force, so all forces are pointing in the same direction *and* friction is limited to just the **rolling friction** (which is an inevitable physical fact, that can not be improved by control).

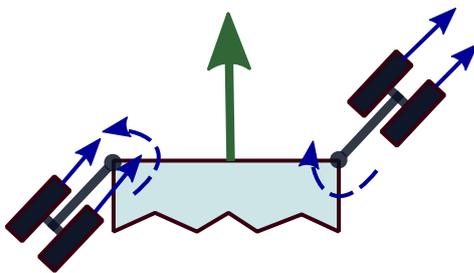


Figure 12.5: The left-hand side wheel unit is pushing the platform, the right-hand side wheel unit is pulling it. The dashed arrows indicate the motion direction of the wheel unit as a result of the forces applied to each wheel.

Magic numbers:

- the parameters in the various possible efficiency functions;
- the thresholds for the decision making to actuate a wheel or not;
- the wheel slippage parameters.

12.6 Platform pushing is (locally) stable, pulling is unstable

Figure 12.5 sketches the forces on the wheels that are needed to generate a desired platform force, once in a “push” configuration (left) and once in a “pull” configuration (right). In the push case, the forces *improve* the *alignment* between wheel unit and platform force, but they *deteriorate* the alignment in the pull case.

This observation only holds *instantaneously* and *locally*: it might be the case that the current task has the explicit intention of turning the pulling wheel unit around into a pushing configuration, and then multiple types of trade-offs can be made between realising this mode change and contributing to the platform propulsion.

Magic numbers:

- the parameters in the various possible friction functions of the wheels with the ground surface;
- the thresholds for the decision making to push or to pull with each wheel unit.

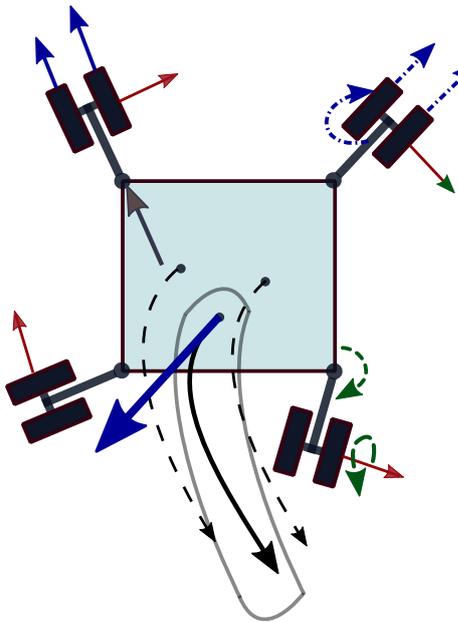


Figure 12.6: The various (physical and virtual) forces acting on the mobile platform: the small blue arrows are active traction forces on the wheels (the F^r and F^l of Fig. 12.4), which result in the grey force on the platform (the F^r of Fig. 12.4); the small blue dashed arrows are passive friction forces (rolling and sliding); the large blue arrow is the (desired or actual) force on the platform body; the red small arrows represent the ideal acceleration constraint forces, orthogonal to the nominal rolling direction of the wheels; and the small green dashed arrows are the friction forces in the bearings of the wheels and of the passive revolute joint in the **swivel caster** suspension to the platform.

The full and dashed black arrows are desired motion trajectories for some selected reference points on the platform; the gray “tube” around the centre one represents the *tolerance* that the motion *specification* allows the motion *controller* to have at runtime.

12.7 Platform force distribution over all 2WD units

The previous Sections looked at the force interactions between one single wheel unit and the mobile platform. However, that platform has *four* wheel units, with in total *eight* actuated wheels (Fig. 12.6). Physically, all wheels transmit their forces to the platform in parallel, so there are *five degrees of actuation redundancy*, and it becomes the responsibility of the *platform motion controller* to decide *how to distribute*, over the four wheel units, the desired force and torque needed for the motion of the platform. The trade-offs to be made in force distribution are: (i) how much *internal platform force*⁷ the wheel units should produce, and (ii) how much force and energy each wheel unit is expected to contribute to the platform motion.

⁷Internal platform forces do not contribute to the acceleration of the platform, since they are the forces that are compensating each other between the driving wheel units. In general, this is a loss of energy, but there might be use cases in which it is useful to use internal forces to make the motion control of the platform *artificially more stiff*: a human trying to push away the platform will feel a higher resisting force than in the case of a non-actuated platform..

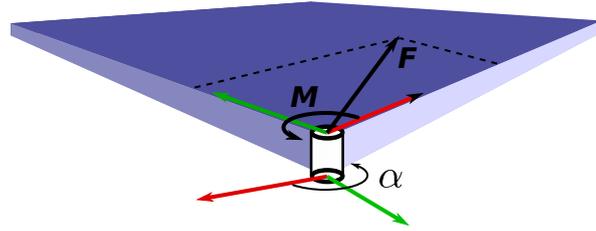


Figure 12.7: The revolute joint that connects the wheel unit to the platform transmits translational forces only. That is, assume that (F_x, F_y, M) represents a general force on the platform (expressed in the reference frame on the platform whose origin coincides with the wheel suspension point), then the force F will be transmitted from platform to wheel unit, but the moment M will not. In the other direction, a wheel unit can only transmit a force F to the platform. The angle α represents the rotation between the (arbitrarily chosen) reference frames on, respectively, the platform and the wheel unit axis.

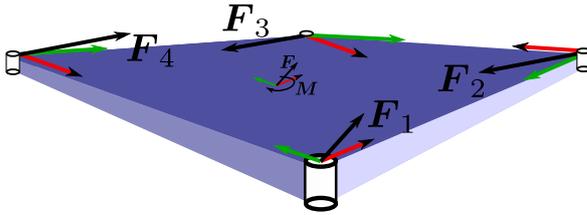


Figure 12.8: The total force F and moment M on the platform is the sum of the forces $F_i (i = 1 \dots 4)$ contributed by all four wheels.

12.7.1 Mechanism: forward force composition from wheels to platform

The following (quasi-static, linear) *force mapping* is a (quasi-static) *physical constraint* that must always be satisfied:

$$\underbrace{F^p}_{3 \times 1} = \begin{pmatrix} F_x^p \\ F_y^p \\ M^p \end{pmatrix} = \underbrace{G}_{3 \times 8} \underbrace{\begin{pmatrix} F^1 \\ \vdots \\ F^4 \end{pmatrix}}_{4 \times (2 \times 1)} = G \begin{pmatrix} F_x^1 \\ F_y^1 \\ F_x^2 \\ F_y^2 \\ F_x^3 \\ F_y^3 \\ F_x^4 \\ F_y^4 \end{pmatrix}, \quad (12.2)$$

where F^p represents the force (two components) and torque (one component) on the platform, and F^i represents the force on the platform generated by the combination of right and left wheel in the i th wheel unit. This force has only two components (Fig. 12.7), since the moment component is not transmitted through the revolute joint that connects the wheel unit to the platform.

G is called the *forward force matrix*, because, *physically*, it adds the forces F^i generated by the four wheel units. The *numerical values* in the G matrix depend on the following geometrical parameters and choices, Fig 12.9:

- the reference frame $\{p\}$ that is chosen to express F^p .

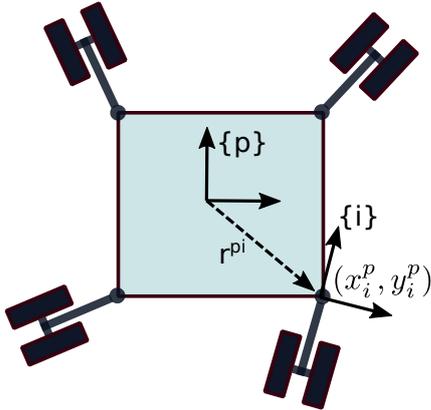


Figure 12.9: For the coordinate representations, and their transformations, two reference frames are relevant: the $\{p\}$ frame fixed to the platform, and the four $\{i\}$ reference frames, with origin in each of the wheels, and with their orientation fixed to the attached wheels. The vector r^{pi} connects the origin of the $\{p\}$ frame to the origin of the $\{i\}$ frame. The coordinates (x_i^p, y_i^p) are those of the attachment point of the joint in the $\{p\}$ reference frame.

- the reference frames $\{i\}$ that are chosen for each of the F^1 .
- the geometric parameters of the kinematic chain formed by the wheels attached to the platform. In Fig 12.9, these are the coordinates (x_i^p, y_i^p) for each of the four wheel units.
- the choice of physical units; e.g., meters and radians.
- the choice of digital representation; e.g., first two 32-bit floats for the force component, followed by on 32-bit float for the moment component.

(The latter two choices are not visible in the Figure.) Taken all this together, G has the following form:

$$G = \begin{pmatrix} c_1 & s_1 & c_2 & s_2 & \cdots & c_4 & s_2 \\ -s_1 & c_1 & -s_2 & c_2 & \cdots & -s_4 & c_4 \\ -x_1 s_1 & y_1 c_1 & -x_2 s_2 & y_2 c_2 & \cdots & -x_4 s_4 & y_4 c_4 \end{pmatrix}, \quad (12.3)$$

$$\text{with } c_i = \cos(\alpha^i), s_i = \sin(\alpha^i). \quad (12.4)$$

12.7.2 Mechanism: inverse force distribution from platform to wheels

In the platform control, the *inverse force mapping* is required, that is, a method **to distribute** each *desired* platform force F^P onto four desired wheel unit forces F^i . This is as simple as “inverting” Eq. (12.2). However that inverse is not uniquely defined, since G is not a square invertible matrix, for two reasons:

- *redundancy*: there are an infinite number of wheel forces F^i that realise any given platform force F^P , so one has to choose a **policy** to determine the distribution. This boils down to “minimize” the internal forces between the wheel units according to some arbitrarily chosen **metric**.
- *singularity*: the wheel units are not one-to-one transformers of force, since there is the **constraint** that the moment component is not transmitted.

With these physical insights, in combination with other arbitrary choices that have to be made, the force distribution problem has the following *conceptual* solution:

- first, decide which wheel units will have *to contribute actively*, and which others will be used in unactuated, **coasting** mode.

- *weigh* the contributions over the wheels with the *apparent inertia*⁸ felt at each swivel attachment point.
- *scale* this inertia-weighted distribution via the *efficiency* (Sec. 12.5) that each active wheel unit can provide.

The conceptual solution can be realised in many different ways, for which multiple decisions have to be made: when to let a wheel coast or not, which version of “efficiency” is used, what is the allowed tolerance, etc. The *inertia weighted pseudo-inverse* formulation looks as follows:

$$\min_{F_x^i, F_y^i} (GF^i - F^p)^T W^p (GF^i - F^p) + \sum_i (F^i)^T W^i F^i. \quad (12.5)$$

W^p and W^i are the translational parts only of the full Cartesian space inertia matrices of the platform, expressed in the frames $\{p\}$ and $\{i\}$. Only the translational force components F_x^i and F_y^i (in each of the four wheel units) are being optimized over, so, the minimization searches for 4×2 numbers, that realises the desired platform force and moment $F^p = (F_x^p, F_y^p, M^p)$, keeping the magnitudes of the force components F_x^i and F_y^i as low as possible weighted by the platform inertia felt at each wheel unit attachment point. One could add the following four scalar **constraints** to the minimization in Eq. (12.5):

$$\forall i, M^i = 0, \quad (12.6)$$

representing the fact that, at the attachment points, moments can not be generated by the wheels anyway.

Magic numbers:

- the parameters in the various possible weighing functions;
- the parameters in the motion tubes;
- the parameters in the time and space horizon of a non-instantaneous motion specification;
- the *epsilon* and *maximum number of iterations* parameters in the SVD algorithm;
- the choice of the motion reference points on the platform.

12.7.3 Mechanism: force distribution at the wheels

The Section above explains how to find the 2D force (F_x^i, F_y^i) to be generated by each wheel unit i , to realise a desired force and moment on the platform. This Section adds the properties of a 2WD unit, Secs 12.4–12.5. On one hand, this means to add two more **constraints** to Eq. (12.5):

- the **efficiency** of a unit to transmit forces from the wheel actuators to the platform attachment point must be above a certain threshold.

⁸The “apparent inertia”, or “articulated body inertia” [37, 102] can only be computed if a full dynamics model of the platform is available.

- the **friction** at each of the wheels constrains the maximum magnitude of the transmitted force.

On the other hand, the 2WD level of abstraction also adds a possible extra **objective function**, that represents the desire to find solutions to the force distribution problem that change minimally over time:

$$\min_{F_x^i, F_y^i} (GF^i - F^p)^T W^p (GF^i - F^p) + \sum_i (F^i)^T W^i F^i + \sum_i (\Delta F^i)^T W^i \Delta F^i, \quad (12.7)$$

where ΔF^i is the difference between the currently computed force magnitude and the one that was applied at the previous sample instant.

12.7.4 Policy: numerical force distribution solvers

When one opts for a *linear* algorithm to solve the force distribution problem, the following linear algebra insights⁹ are relevant:

- *pseudo-inverse*: a non-square linear set of equations $Ax = b$, like Eq. (12.2), has a solution $x = A^\dagger b$. Depending on whether A is of full row rank or column rank, the analytical expression for A^\dagger is $(A^T A)^{-1} A^T$ or $A^T (A A^T)^{-1}$.
- *minimization problem*: this pseudo-inverse solution x is also the solution of the following “least squares” problem:

$$\min_x (Ax - b)^T (Ax - b) + x^T x. \quad (12.8)$$

In other words, it finds the “smallest” x that is mapped by A to the “closest” possible point near b .

- *weighted pseudo-inverse*: in most cases, the “squares” in the equation above must be replaced by “weighted least squares”, because of dimensional homogeneity¹⁰ or task specification trade-offs:

$$\min_x (Ax - b)^T W (Ax - b) + x^T K x, \quad (12.9)$$

with W a *metric* in the *range space* of b , and K a metric in the *domain space* of x .

- *inertia weighted pseudo-inverse*: nature provides a particular way “to solve” Eq. (12.9), where W is a Cartesian-space inertia matrix and K its joint-space equivalent, e.g., [46, 102].
- *Singular Value Decomposition solver*: all of the problems above can be solved via (variants of) the numerically optimal *SVD* algorithm, e.g., [96].
- *runtime iteration solver*: in general, solving an SVD problem involves *numerical iterations* until **convergence** to a user-specified “epsilon”; in a robotics context, one could think of spreading these iterations over subsequent controller time instants, using the last computed iteration as a *feasible* initial solution for the next iteration, with somewhat changed problem parameters.

⁹More technical details can be found in linear algebra textbooks such as [47, 93].

¹⁰Indeed, in many cases, different components in x and/or b have different physical units. This holds for the forces (with Newtons as units) and torques (with Newton-meters as units) in the platform dynamics context of this document.

12.7.5 Mechanism: force trajectory objectives

The previous Sections considered an **instantaneous** problem only: the optimizations only consider the actual values, and have no horizon of several sample instants, to the future as well as to the past. In many use cases, introducing a **finite horizon** makes a lot of sense, to optimize effects that need some time to be visible and/or controllable, such as, for example, oscillations, dampings, or hysteresis effects. The generic answer to this is to introduce constrained optimization over a horizon, which is known as **Model Predictive Control** (MPC). This approach adds a term in the objective functions of Eq. (12.7) and its “predecessors”, for each time instant in the horizon, and that means an exponential increase in the computational complexity of the solvers. *The* approach to reduce this combinatorial explosion is to introduce **parameterized trajectories** that have only a small number of parameters. In other words, by “sampling” over the parameters, a variety of trajectories is generated, for each one the objective function is evaluated, and then the optimal one is selected. This has several advantages, that reinforce each other:

- **link with control and plan:** these parts of the *platform-level* Task meta model often often already make use of parameterized trajectories, which can be reused inside the force distribution problem.
- some **constraints and/or objective functions** can be **satisfied by design**, because the generated trajectory already satisfies them.
- **contracts** about **expected and actual progress:** a side effect of using simple local trajectories at the platform-level of the motion, is that also each 2WD unit gets condensed information about what the platform control expects from the 2WD control. This information can be used as a contract between both controllers, and as a reference to measure progress of the control execution at both sides.

Examples of short-horizon trajectories are: constant velocity, constant acceleration, or **clothoid** (linear acceleration changing) curves, as sketched in Fig. 5.18; all have *known* oscillation and smoothness properties.

12.8 Mechanism: declarative platform motion specification via tubes

A complementary approach to solve the platform control problem not instantaneously, but over a certain “horizon” in time and space, is depicted in Fig. 12.10. Instead of using a *procedural* specification in the form of a parameterized curve, it introduces a **declarative** specification in the form of a “*tube*”. The control must choose platform force/torque drivers $F^p(t_i), i = 1, \dots, N$ over a finite time horizon of N samples, with which to drive the platform in such a way that its motion is (expected) to remain within the specified tube. There can be more than one tube at the same time, as long as they overlap, since each one represents the *tolerance* within which the motion of a selected reference point on the platform must remain.

For online use, it makes sense to extend such “forward-looking” *time series* $F^p(t_i)$ of driver forces over a particular horizon of N instants, with a “backward-looking” time series of recently passed positions $p^p(k), k \in -1, \dots, -M$ of M instants (Fig. 12.10). The reason is that there is a high uncertainty about where exactly the platform was in the recent past,

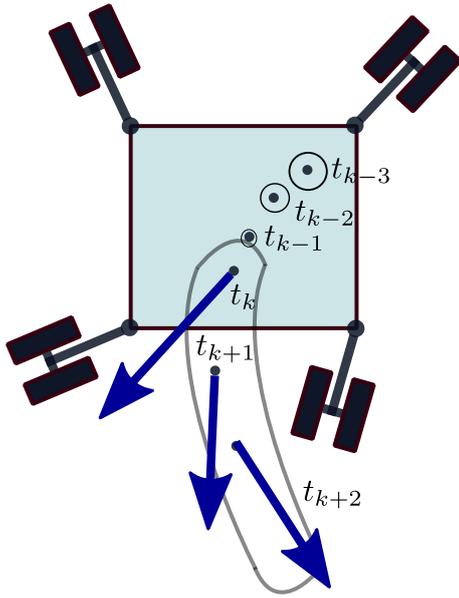


Figure 12.10: At the current time instant t_k , the platform controller primitives are (i) the current desired total force at a chosen reference point on the platform, (ii) the expected n future such force specifications (with $n = 2$ in this example), and (iii) the last m positions that the platform reference point has moved through (with $m = 3$ in the example). The circles around the latter points refer to the uncertainty in knowing their coordinates.

due to the high number of disturbances that can occur; e.g., slippage, internal forces between wheel units, external forces, etc.

12.9 Policy: 2WD control modes, task progress, energy, slippage

By means of the methods explained in the previous Sections, the platform control generates, for each 2WD unit, a local (in time and space) motion specification, together with an indication of the progress that it expects each two-wheel unit to make within that control horizon. The responsibility of the 2WD controller is then to find an “optimal” trade-off between:

- realising that progress. In the context of the [Task meta model](#), this is the *capabilities* part of the 2WD Task control level.
- minimizing the energy to do so. This is the *resources* part, where the electrical motor and battery are the resources. (Or similar components for other types of actuation.)
- minimizing the slippage of the wheels. This is the sole responsibility of the 2WD Task controller, not to be shared with any other control level.
- optimizing the efficiency of the unit’s force transfer to the platform. Also the sole responsibility of the 2WD Task controller.

A 2WD unit has the following *modes* (with qualitative descriptions of their suggested behaviour) to determine the choice of trade-off:

- *pulling*: only the “outermost” wheel provides traction, while the “innermost” wheel coasts. Otherwise, the wheel unit would make its own position evolve into one with worse efficiency.
- *pushing*: both wheels provide traction forces. The magnitude can be made proportional to the relative efficiency of this 2WD unit compared to the other units.

- *from pulling to pushing*: both wheels coast, while the platform is driven by the other 2WD units, until this unit’s efficiency has reached a certain threshold, after which it can start applying traction.
- *from pushing to pulling*: both wheels apply traction, irrespective of the unit’s efficiency and the actions of the other units, with an *open loop* trajectory that should turn the unit around its swivel point. Of course, what the other units are doing brings disturbances to the expected evolution of the wheel unit turning.
- *unknown*: because of the partial observability of the platform (Sec. 12.1), the configuration of a wheel unit with respect to the platform might not be known with sufficient certainty to decide on a control mode.

12.10 Policy: hybrid platform control

The approach presented in this Section assumes that one has good *a priori* knowledge about how the platform behaves under given wheel actuation in the various *control modes* of, both, the platform and each of the 2WD units. The discrete part of the Task control, the plan, makes the decision to switch between such control modes. For example, the following configurations¹¹ of the wheel units correspond to different force transformations from wheels to the platform:

- all wheel units are “pushing” (Sec. 12.6), and with similar “efficiencies”.
- one wheel unit is “pulling”, the others are “pushing”;
- one wheel unit is “pulling” and must transition to “pushing”;
- the platform must transition between two of the possible “motion” control modes: position, velocity, force, impedance, energetic effort, . . . , with or without “tube” tolerances on the setpoints/ranges specified in the control domain space.
- the platform starts a “motion” from a stand-still situation, with none of the wheel units being aligned, neither amongst themselves, nor with the intended platform motion direction;
- . . .

The relevant *prior knowledge* for each mode has not yet been identified, let alone formalized in computable algorithms, but the commonalities between all approaches are the following:

- one selects a particular platform control mode;
- one selects particular wheel actuator forces, based on some variant of a *lookup table* or the solution of a constrained optimization problem;
- one sums the effects of all wheel units via Eq. (12.2), which results in a driving force and torque on the platform;
- one integrates this over a time interval, to have a prediction of the resulting motion;

¹¹This set is not exhaustive, yet.

- if that prediction satisfies the task requirements “well enough”, one applies the computed trajectories to the real wheels;
- when it is not good enough, one increments the feedforward trajectory in one or more of the wheel units, based on the known trends between changes in trajectory specification and changes in the monitored task quality outputs.

Probably the simplest control mode is that of “*all wheel units are pushing and are well-aligned*”. Since this mode is comparable to driving a traditional car, the above-mentioned *open-loop selection* would boil down to something of the following kind:

- to make the platform change the radius of a “circular motion”, all wheel units get the same *desired increment in direction* of their platform force F_i^p ;
- to make the platform increase its “average speed”, all wheel units get the same *desired increment in magnitude* of their platform force F_i^p .

Such control approach (based on discrete increments on the inputs together with a horizon-based monitoring/prediction of the resulting open loop trajectory) does not fit very well with the traditional work house of feedback control, namely the **PID controller**, but rather with something like **sliding mode control**, or **ABAG** control, [40].

Magic numbers:

- (TODO)

12.11 Navigation: topological motion specification and control

The control level “above” that of the platform *motion* is that of platform *navigation*, that is, to decide how to move the platform through an *environment* as a series of *areas* (represented by semantic tags or “*waypoints*”) to pass. Figure 12.11 sketches the situation, in which a *map* of that environment exists, via which *semantic motion constraints* are indicated; the “legends” that is used in the Figure is that of the mainstream outdoors road signs. At that level of abstraction, a “path” consists of the semantic tags of the “traffic” primitives on the map, together with some sort of *progress measure* that fits to that chosen level of abstraction.

Magic numbers:

- (TODO)

12.12 Information architecture

Figure 5.2 shows the graph of relations that connect the core components in the *model* of any task specification, at any of the levels of abstraction discussed in this document (mission, navigation, platform motion, 2WD unit motion, and wheel motion, Sec. 12.2).

The following paragraphs provide a reference software architecture for the relevant concurrent mobile robot activities of wheel control, 2WD unit control, platform motion control, and navigation control, together with their hardware drivers and behaviour monitors. (TODO)

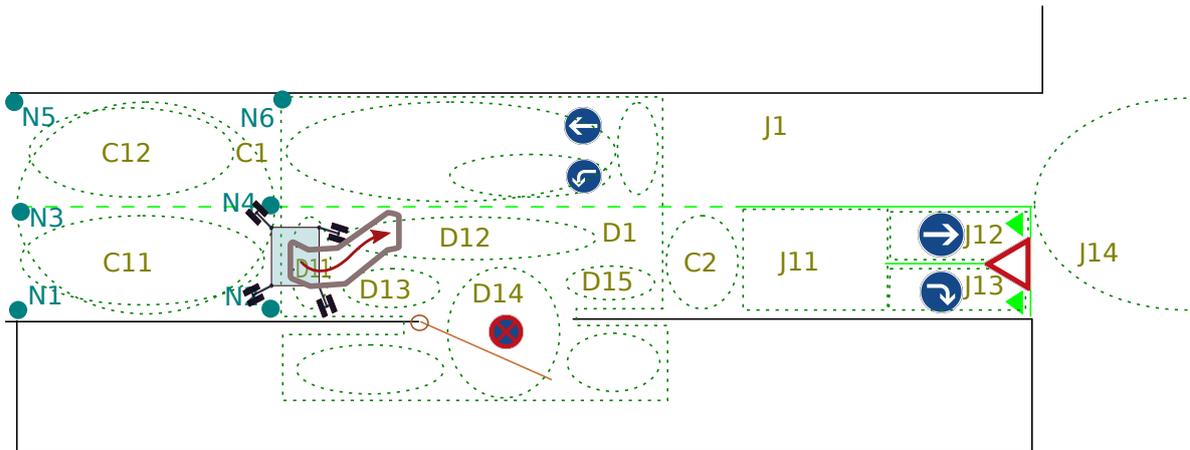


Figure 12.11: In *cruising* mode (Sec. 12.3), the navigation controller has to solve the problem of generating a “tube” trajectory for the platform, taking into account the semantic primitives in the map. The example above show a possible “tube” to bring the platform from the semantic area “D11” to the semantic area “D12”.

Magic numbers:

- (TODO)

Bibliography

- [1] ACKOFF, R. L. Towards a system of systems concepts. *Management Science* 17, 11 (1971), 661–671.
- [2] ACKOFF, R. L. From data to wisdom. *Journal of Applied Systems Analysis* 16 (1989), 3–9.
- [3] AERTBELIËN, E., AND DE SCHUTTER, J. eTaSL/eTC: A constraint-based task specification language and robot controller using expression graphs. In *Proceedings of the 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Chicago, IL, USA, 2014), IROS2014, pp. 1540–1546.
- [4] AL BAHRA, S. Nonblocking algorithms and scalable multicore programming. *Communications of the ACM* 58, 7 (2013), 50–61.
- [5] ALAMI, R., CHATILA, R., FLEURY, R., GHALLAB, M., AND INGRAND, F. An architecture for autonomy. *The International Journal of Robotics Research* 17, 4 (1998), 315–337.
- [6] ALEXANDER, C., ISHIKAWA, S., AND SILVERSTEIN, M. *A pattern language: towns, buildings, construction*. Oxford University Press, 1977.
- [7] ANGELES, J. *Rational Kinematics*. Springer, 1988.
- [8] ANGLES, R., ARENAS, M., BARCELÓ, P., HOGAN, A., REUTTER, J., AND VRGOČ, D. Foundations of modern graph query languages. *ACM Computing Surveys* 50, 5 (2017), 1–40.
- [9] BALL, R. S. *Theory of screws: a study in the dynamics of a rigid body*. Hodges, Foster and Co, Dublin, Ireland, 1876. Reprinted 1998, by Cambridge University Press.
- [10] BARTELS, G., KRESSE, I., AND BEETZ, M. Constraint-based movement representation grounded in geometric features. In *13th IEEE-RAS International Conference on Humanoid Robots* (Atlanta, Georgia, USA, October 15–17 2013).
- [11] BAUMGARTE, J. W. Stabilization of constraints and integrals of motion in dynamical systems. *Computer Methods in Applied Mechanics and Engineering* 1, 1 (1972), 1–16.
- [12] BÉZIVIN, J. On the unification power of models. *Software and Systems Modeling* 4, 2 (2005), 171–188.
- [13] BISHOP, C. M. *Pattern Recognition and Machine Learning*. Springer, 2006.

- [14] BORGHEGAN, G., SCIONI, E., KHEDDAR, A., AND BRUYNINCKX, H. Introducing geometric constraint expressions into robot constrained motion specification and control. *IEEE Robotics and Automation Letters* 1, 2 (July 2016), 1140–1147.
- [15] BORGIO, S. Euclidean and mereological qualitative spaces: a study of SCC and DCC. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09)* (2009), pp. 708–713.
- [16] BORST, P., AKKERMANS, H., AND TOP, J. Engineering ontologies. *International Journal on Human-Computer Studies* 46 (1997), 365–406.
- [17] BOTTEMA, O., AND ROTH, B. *Theoretical Kinematics*. Dover Books on Engineering. Dover Publications, Inc., Mineola, NY, 1990.
- [18] BURKE, W. L. *Applied differential geometry*. Cambridge University Press, 1992.
- [19] CECCARELLI, M. Screw axis defined by Giulio Mozzi in 1763. In *9th World Congress IFToMM* (Milano, Italy, 1995), pp. 3187–3190.
- [20] CHASLES, M. Note sur les propriétés générales du système de deux corps semblables entr’eux et placés d’une manière quelconque dans l’espace; et sur le déplacement fini ou infiniment petit d’un corps solide libre. *Bulletin des Sciences Mathématiques, Astronomiques, Physiques et Chimiques* 14 (1830), 321–326.
- [21] CHAUVEL, F., AND JÉZÉQUEL, J.-M. Code generation from UML models with semantic variation points. In *International Conference on Model Driven Engineering Languages and Systems* (2005), no. 3713 in Springer Lecture Notes in Computer Science, pp. 54–68.
- [22] CHEN, P. P.-S. The entity-relationship model—Toward a unified view of data. *ACM Transactions on Database Systems* 1, 1 (1976), 9–36.
- [23] COX, I. J., AND LEONARD, J. J. Modeling a dynamic environment using a Bayesian multiple hypothesis approach. *Artificial Intelligence* 66 (1994), 311–344.
- [24] CRAMPIN, M., AND PIRANI, F. A. E. *Applicable Differential Geometry*, 3rd ed., vol. 59 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1988.
- [25] D’ALEMBERT, J. L. R. *Traité de Dynamique*. 1742.
- [26] DE LAET, T., BELLENS, S., SMITS, R., AERTBELIËN, E., BRUYNINCKX, H., AND DE SCHUTTER, J. Geometric relations between rigid bodies (Part 1): Semantics for standardization. *IEEE Robotics and Automation Magazine* 20, 1 (2013), 84–93.
- [27] DE MAUPERTUIS, P. L. M. Accord de différentes lois de la nature. In *Oeuvres, Tome IV*. Olms, Hildesheim, Germany, 1768.
- [28] DE SCHUTTER, J., DE LAET, T., RUTGEERTS, J., DECRÉ, W., SMITS, R., AERTBELIËN, E., CLAES, K., AND BRUYNINCKX, H. Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty. *The International Journal of Robotics Research* 26, 5 (2007), 433–455.

- [29] DEREMER, F., AND KRON, H. Programming-in-the large versus programming-in-the-small. In *Proceedings of the international conference on Reliable software* (1975), pp. 114–121.
- [30] DIJKSTRA, E. W. Letters to the editor: go to statement considered harmful. 147–148.
- [31] DONDRUP, C., BELLOTTO, N., HANHEIDE, M., EDER, K., AND LEONARDS, U. A computational model of human-robot spatial interactions based on a qualitative trajectory calculus. *Robotics 4* (2015), 63–102.
- [32] EHRIG, H., ERMEL, C., GOLAS, U., AND HERMANN, F. *Graph and Model Transformation. General Framework and Applications*. Monographs in Theoretical Computer Science. Springer, 2015.
- [33] ENDSLEY, M. R. The application of human factors to the development of expert systems for advanced cockpits. 1388–1392.
- [34] ENDSLEY, M. R. Toward a theory of situation awareness in dynamic systems. *Human Factors 37*, 1 (1995), 32–64.
- [35] ENDSLEY, M. R. From here to autonomy: Lessons learned from human–automation research. *Human Factors 59*, 1 (2017), 5–27.
- [36] ENGELS, G., LEWERENTZ, C., SCHÄFER, W., SCHÜRR, A., AND WESTFECHTEL, B., Eds. *Graph Transformations and Model-Driven Engineering*. Springer, 2010.
- [37] FEATHERSTONE, R. *Rigid Body Dynamics Algorithms*. Springer, 2008.
- [38] FLORIDI, L. The method of levels of abstraction. *Minds & Machines 18* (2008), 303–329.
- [39] FLORIDI, L. *The philosophy of information*. Oxford University Press, 2011.
- [40] FRANCHI, A., AND MALLET, A. Adaptive closed-loop speed control of BLDC motors with applications to multi-rotor aerial vehicles. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Singapore, 2017), pp. 5203–5208.
- [41] FRANKEL, T. *The Geometry of Physics*. Cambridge University Press, Cambridge, England, 1996.
- [42] GAUSS, K. F. Über ein neues allgemeines Grundgesetz der Mechanik. *Journal für die reine und angewandte Mathematik 4* (1829), 232–235.
- [43] GIBSON, C. G., AND HUNT, K. H. Geometry of screw systems—1. Screws: Genesis and geometry. *Mechanism and Machine Theory 25*, 1 (1990), 1–10.
- [44] GLANVILLE, R. The purpose of second-order cybernetics. *Kybernetes 33*, 9/10 (2004), 1379–1386.
- [45] GLEZ-CABRERA, F. J., ÁLVAREZ BRAVO, J. V., AND DÍAZ, F. QRPC: A new qualitative model for representing motion patterns. *Expert Systems with Applications 40* (2013), 4547–4561.

- [46] GOLDSTEIN, H. *Classical mechanics*, 2nd ed. Addison-Wesley Series in Physics. Addison-Wesley, Reading, MA, 1980.
- [47] GOLUB, G., AND VAN LOAN, C. *Matrix Computations*. The Johns Hopkins University Press, 1989.
- [48] GOOD, I. J. A derivation of the probabilistic explanation of information. *Journal of the Royal Statistical Society (Series B)* 28 (1966), 578–581.
- [49] HALFORD, G. S., WILSON, W. H., AND PHILLIPS, S. Relational knowledge: the foundation of higher cognition. *Trends in Cognitive Sciences* 14, 11 (2010), 497–505.
- [50] HAMILTON, W. R. On a general method in dynamics. *Philosophical Transactions of the Royal Society*, II (1834), 247–308. Reprinted in Hamilton: The Mathematical Papers, Cambridge University Press, 1940.
- [51] HERLIHY, M. P. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* 13, 1 (1991), 124–149.
- [52] HILL, J. W., AND SWORD, A. J. Manipulation based on sensor-directed control: an integrated end effector and touch sensing system. In *17th Annual Human Factors Society Convention* (1973).
- [53] HOLVOET, T., WEYNS, D., AND VALCKENAERS, P. Delegate MAS patterns for large-scale distributed coordination and control applications.
- [54] HUNT, K. H. *Kinematic Geometry of Mechanisms*, 2nd ed. Oxford Science Publications, Oxford, England, 1990.
- [55] IMIYA, A. A metric for spatial lines. *Pattern Recognition Letters* 17 (1996), 1265–1269.
- [56] JAYNES, E. T. *Probability Theory: The Logic of Science*. Cambridge University Press, 2003.
- [57] JOURDAIN, P. E. B. Note on an analogue of Gauss’ Principle of least constraint. *Quarterly Journal of Pure and Applied Mathematics* 8L (1909).
- [58] KARGER, A., AND NOVAK, J. *Space kinematics and Lie groups*. Gordon and Breach, New York, NY, 1985.
- [59] KIM, J. H., AND PEARL, J. A computational model for combined causal and diagnostic reasoning in inference systems. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence* (Karlsruhe, Germany, 1983), pp. 190–193.
- [60] KOESTLER, A. *The Ghost in the Machine*. 1967.
- [61] KUIPERS, B. Modeling spatial knowledge. *Cognitive Science* 2 (1978), 129–153. Reprinted in *Advances in Spatial Reasoning*, Volume 2, Su-shing Chen (Ed.), Norwood NJ: Ablex Publishing, 1990.
- [62] KUIPERS, B. An intellectual history of the spatial semantic hierarchy. In *Robotics and Cognitive Approaches to Spatial Mapping*, vol. 38 of *Springer Tracts in Advanced Robotics*. 2008, pp. 243–264.

- [63] KUIPERS, B., MODAYIL, J., BEESON, P., MACMAHON, M., AND SAVELLI, F. Local metrical and global topological maps in the hybrid spatial semantic hierarchy. In *Proceedings of the 2004 IEEE International Conference on Robotics and Automation* (New Orleans, U.S.A., 2004), ICRA2004, pp. 4845–4851.
- [64] LAGRANGE, J. L. Mécanique analytique. In *Oeuvres*, J.-A. Serret, Ed. Gauthier-Villars, Paris, France, 1867.
- [65] LAGRIFFOUL, F., DIMITROV, D., BIDOT, J., SAFFIOTTI, J., AND KARLSSON, L. Efficiently combining task and motion planning using geometric constraints. *The International Journal of Robotics Research* 33, 14 (2014), 1726–1747.
- [66] LEE, J., BAGHERI, B., AND KAO, H.-A. A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems. *Manufacturing Letters* 3 (2015), 18–23.
- [67] LIPKIN, H., AND DUFFY, J. Hybrid twist and wrench control for a robotic manipulator. *Transactions of the ASME, Journal of Mechanisms, Transmissions, and Automation in Design* 110 (1988), 138–144.
- [68] LONČARIĆ, J. Normal forms of stiffness and compliance matrices. *IEEE Journal of Robotics and Automation RA-3*, 6 (1987), 567–572.
- [69] MADNI, A. M., AND SCOTT, J. Towards a conceptual framework for resilience engineering. *IEEE Systems Journal* 3, 2 (2009), 181–191.
- [70] MANSARD, N., AND CHAUMETTE, F. Task sequencing for sensor-based control. *IEEE Transactions on Robotics* 23, 1 (2007), 60–72.
- [71] MASON, M. T. Compliance and force control for computer controlled manipulators. *IEEE Transactions on Systems, Man, and Cybernetics SMC-11*, 6 (1981), 418–432.
- [72] MOORE, E. H. On the reciprocal of the general algebraic matrix. *Bulletin of the American Mathematical Society* 26 (1920), 389 and 394–395.
- [73] MOZZI, G. *Discorso Matematico sopra il Rotamento Momentaneo dei Corpi*. Stamperia del Donato Campo, Napoli, 1763.
- [74] PARASURAMAN, R., SHERIDAN, T. B., AND WICKENS, C. D. A model for types and levels of human interaction with automation. *IEEE Transactions on Systems, Man, and Cybernetics. Part A: Systems and Humans* 30, 3 (2000), 286–297.
- [75] PEARL, J. Fusion, propagation, and structuring in belief networks. *Artificial Intelligence* 29 (1986), 241–288.
- [76] PERZYLO, A., SOMANI, N., PROFANTER, S., GASCHLER, A., GRIFFITHS, S., RICKERT, M., AND KNOLL, A. Ubiquitous semantics: Representing and exploiting knowledge, geometry, and language for cognitive robot systems. In *15th IEEE-RAS International Conference on Humanoid Robots* (Seoul, Republic of Korea, November 2015).
- [77] PERZYLO, A., SOMANI, N., RICKERT, M., AND KNOLL, A. An ontology for CAD data and geometric constraints as a link between product models and semantic robot task descriptions. In *Proceedings of the 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Hamburg, Germany, 2015), IROS2015.

- [78] PHILLIPS, S., HALFORD, G. S., AND WILSON, W. H. The processing of associations versus the processing of relations and symbols: A systematic comparison. In *Seventeenth Annual Conference of the Cognitive Science Society* (1995), pp. 688–691.
- [79] POINSOT, L. Sur la composition des moments et la composition des aires. *Journal de l'Ecole Polytechnique* 6, 13 (1806), 182–205.
- [80] POPOV, E. P., VERESHCHAGIN, A. F., AND ZENKEVICH, S. L. *Manipulyatsionnyye roboty: dinamika i algoritmy*. Nauka, Moscow, 1978.
- [81] RADESTOCK, M., AND EISENBACH, S. Coordination in evolving systems. In *Trends in Distributed Systems. CORBA and Beyond*. Springer-Verlag, 1996, pp. 162–176.
- [82] RAMM, E. Principles of least action and of least constraint. *Gesellschaft f. Angewandte Mathematik und Mechanik (GAMM)* 34, 2 (2011), 164–182.
- [83] REID, D. B. An algorithm for tracking multiple targets. *IEEE Transactions on Automatic Control AC-24*, 6 (December 1979), 843–854.
- [84] ROTH, B. On the screw axis and other special lines associated with spatial displacements of a rigid body. *Transactions of the ASME, Journal of Engineering for Industry* 89 (1967), 102–110.
- [85] SAINT GERMAIN, A. D. Sur la fonction s introduite par M. Appell dans les équations de la dynamique. *Comptes Rendus de l'Académie des Sciences de Paris* 130 (1900), 1174.
- [86] SARIDIS, G. N., AND STEPHANOU, H. E. A hierarchical approach to the control of a prosthetic arm. *IEEE Transactions on Systems, Man, and Cybernetics* 7, 6 (1977), 407–410.
- [87] SCHULZ, M. Decisions and higher-order knowledge. *Noûs* 51, 3 (2017), 463–483.
- [88] SCIONI, E. *Online Coordination and Composition of Robotic Skills: Formal Models for Context-aware Task Scheduling*. PhD thesis, IUSS Ferrara 1391, University of Ferrara, Italy and Department of Mechanical Engineering, KU Leuven, Belgium, April 2016.
- [89] SCIONI, E., HÜBEL, N., BLUMENTHAL, S., SHAKHIMARDANOV, A., KLOTZBÜCHER, M., GARCIA, H., AND BRUYNINCKX, H. Hierarchical hypergraphs for knowledge-centric robot systems: a composable structural meta model and its domain specific language NPC4. *Journal of Software Engineering in Robotics* 7, 1 (2016), 55–74.
- [90] SIMON, H. *The Sciences of the Artificial*. MIT Press, 1969.
- [91] SIMON, H. A. Behavioral model of rational choice. *The Quarterly Journal of Economics* 69, 1 (1955), 99–118.
- [92] SINACEUR, H. B. Facets and levels of mathematical abstraction. *Philosophia Scientiae* 18, 1 (2014), 81–112.
- [93] STRANG, G. *Linear Algebra and its Applications*, 3rd ed. Harcourt Brace Jovanovich, San Diego, CA, 1988.

- [94] TOLK, A., ADAMS, K. M., AND KEATING, C. B. Towards intelligence-based systems engineering and system of systems engineering. In *Intelligence-Based Systems Engineering* (2011), vol. 10 of *Intelligent Systems Reference Library*, pp. 1–22.
- [95] TOSI, N., DAVID, O., AND BRUYNINCKX, H. DOF decoupling task graph model: Reducing the complexity of touch-based active sensing. *Robotics 4*, 2 (2015), 141–168. Special Issue “Representations and Reasoning for Robotics”.
- [96] UDWADIA, F. E., AND PHOHOMSIRI, P. Generalized LM-inverse of a matrix augmented by a column vector. *Applied Mathematics and Computation* 190, 3 (2007), 999–1006.
- [97] VALCKENAERS, P. ARTI reference architecture — PROSA revisited. In *Service Orientation in Holonic and Multi-Agent Manufacturing* (2018), no. 803 in *Studies in Computational Intelligence*, pp. 1–19.
- [98] VALCKENAERS, P. Perspective on holonic manufacturing systems: PROSA becomes ARTI. *Computers in Industry* 120 (2020), 103226:1–14.
- [99] VAN LEEUWEN, J. On Floridi’s method of levels of abstraction. *Minds & Machines* 24 (2014), 5–17.
- [100] VANTHIENEN, D., KLOTZBÜCHER, M., AND BRUYNINCKX, H. The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming. *Journal of Software Engineering in Robotics* 5, 1 (2014), 17–35.
- [101] VERESHCHAGIN, A. F. Gauss principle of least constraint for modelling the dynamics of automatic manipulators using a digital computer. *Soviet Physics Doklady* 20, 1 (1975), 33–34. Originally published in *Dokl. Akad. Nauk SSSR*, Vol. 220, No. 1, pp. 51–53, 1975.
- [102] VERESHCHAGIN, A. F. Modelling and control of motion of manipulative robots. *Soviet Journal of Computer and Systems Sciences* 27, 5 (1989), 29–38. Originally published in *Izvestiia Akademii nauk SSSR, Tekhnicheskaya Kibernetika*, No. 1, pp. 125–134, 1989.
- [103] VON DER BEECK, M. A comparison of Statecharts variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, H. Langmaack, W.-P. de Roever, and J. Vytöpil, Eds., vol. 863 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1994, pp. 128–148.
- [104] W3C. QUDT (Quantities, Units, Dimensions, and Types). <http://www.qudt.org>.
- [105] WIENER, N. *Cybernetics or Control and communication in the animal and the machine*. MIT Press, 1948. 1894–1964, PhD at Harvard at the age of 18.
- [106] WILL, P. M., AND GROSSMAN, D. D. An experimental system for computer controlled mechanical assembly. *IEEE Transactions on Computers* 24, 9 (1975), 879–888.
- [107] WINFIELD, A. F. T. Experiments in artificial theory of mind: From safety to storytelling. *Frontiers in Robotics and AI* 5 (2018).

- [108] WIRTH, N. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.
- [109] YUNT, K. On the relation of the principle of maximum dissipation to the principles of Jourdain and Gauss for rigid body systems. *Transactions of the ASME, Journal of Computational and Nonlinear Dynamics* 9 (2014), 031017–1–11.
- [110] ZELLNER, A. Optimal information processing and Bayes's theorem. *The American Statistician* 42 (1988), 278–284.