



RobMoSys

H2020-ICT-732410

RobMoSys

**Composable Models and Software
for Robotics Systems**

Deliverable D3.3 – v1.0:

**Second motion, perception and world-model
stacks specifications**



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



Project acronym:	RobMoSys
Project full title:	Composable Models and Software for Robotics Systems
Work Package:	WP 3
Document number:	D3.3 – v1.0
Document title:	Second motion, perception and world-model stacks specifications
Version:	1.0
Delivery date:	28 June, 2019
Nature:	Report (R)
Dissemination level:	Public (PU)
Editor:	Herman Bruyninckx (KUL)
Authors:	Enea Scioni (KUL), Herman Bruyninckx (KUL), Nico Hübner (KUL), Marco Frigerio (KUL), Matteo Morelli (CEA), Dennis Stampfer (HSU), Christian Schlegel (HSU)
Reviewer:	Alessandro di Fava (PAL)

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

Executive summary

This Deliverable is an extended version of Deliverable D3.1 (“First motion, perception and world-model stacks specifications”). That latter Deliverable has been “kept alive” continuously, and has started to live its own useful life as RobMoSys dissemination instrument: several hundreds of students in various universities have already been using it as (non-exclusive) course material, and several H2020 and academia-industry projects have been using it as foundation.

The version of that “master document” from the day of submission of this Deliverable is attached. The rest of this document is a copy of the *Executive Summary* of the master document.

The major steps forward since the previous Deliverable are:

- all necessary software patterns and best practices that WP3 will need are present, and documented.
- major choices have been made for software and tooling that the software developments in WP3 are going to depend on. Major criteria have been:
 - maturity of the code base.
 - clear and unambiguous (“semantic”) interfaces and documentation that can be made 100% compliant with the RobMoSys meta models.
 - composability: small scale, single focus, no hidden “runtime”.
 - programmed preferable in the C language ecosystem, to allow full and efficient data and function accessibility.

The major items on this list are: cgraph, FlatBuffers, Lapack, SOEM, Lua.

The realtime foundations of the motion stack are now all being implemented together, for all relevant parts (world model, motion control, perception, monitoring, and task specification).

Cyber-physical systems consist of multiple sub-**systems** that **interact** with the physical world, but also with each other. They exchange **matter**, **energy** and **data**, but also more and more **information** and **knowledge**, formally represented as “data”. The ambition of this document is to find the **least amount** of **formally encoded knowledge models**¹ that are needed **to represent** all possible systems in such a way that, they can **be controlled** in **predictable**, **resilient** and **explainable** ways,² and still be **re-composable** in any type of **system architecture**.

This document is inspired by the **best practices** to be found in many successful and resilient societies and organisations created by humanity, and in the vast amount of **knowledge** in **physics**, **mathematics**, **computer science**, and **systems** and **control** theory.

¹This is the “*simple*” ambition of this document: each individual knowledge model is small enough to understand and comprehend fully in half an hour.

²This is the “not easy” part: it takes a lot of effort and iterations to get the envisaged integration of many simple things right, that is, correct, effective and efficient.

Robotic systems are the primary application target. This document advocates the design approach that considers the simplest robotic system consists already of *multiple* robots, each executing *multiple* tasks at the same time, for many of which they have to cooperate and to share resources, and with all their behaviour and actions realised by means of *multiple asynchronous* software activities.

The **system architecture** guidelines in this document build upon the **meta model** of the **task**: the **world model** plays the role of the only activity in the system that provides loose coupling between the other essential activities of **discrete and continuous control**, and **discrete and continuous perception**. Such **information architectures** must be made at various **levels of abstraction** (actuator control, proprioceptive and exteroceptive platform control) and their specification as **hybrid constrained optimization problems** supports **vertical and horizontal composition**, and deployment to heterogeneous software and hardware implementations.

This document explains how to use the 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.

After more than 50 years of evolution, the robotics domain has created a large amount of insights, technology, models and (open source) software. But most of those efforts have still to be consolidated into commonly supported and standardized components, with **best practice** architectures that can *guarantee* safe, secure, efficient and effective operation of robotics systems. From the systems perspective, the evolution and the *state of the practice* in robotics is very similar to that of other domains where ICT platforms play an ever increasing role: energy production and distribution, multi-modal logistics and traffic, manufacturing, medical instruments, etc.; this document uses the term **cyber-physical systems** for its Chapters and Sections that do not contain any knowledge that is specific to the robotics domain.

At the highest level of modelling abstraction, a cyber-physical **system** is a set of **activities** that provide **behaviour**, via which they change their own **state** and/or that of the **resources** they have to **share** with other Activities, via **interactions**. The words in bold are the core entities and relations, and they can be realised in the **physical** world (e.g., the electro-mechanical behaviour of robots or cars, the measurement principles behind sensors, the chemistry in a battery) as well as in the **cyber** world (that is, the composition of computational and communication hardware and software); the **transformations** between both worlds works via **transducers** that bring their own dynamics and interaction into the system design.

The **design** of all cyber-physical **applications** have some major challenges in common, which form the **theme** of this document: **composability**, **self-reflection**, **reactivity** and **explainability**. In ideal system design, the latter one should be a consequence of the former two, as soon as the design of components takes system-level integration and self-reflection into account as primary **design drivers**, and as an intermediary goal towards self-explainability of ICT platforms.

The size and the scale of integration of robotic systems grow beyond what single developers or development teams, can comprehend, create, validate and maintain. Contrary to what has happened in the ICT-driven applications that could be built on top of “the Web” platform, no “*giant companies*”³ have been created that have the funding and man power to realise such con-

³The so-called **GAFA giants**: Google, Amazon, Facebook and Apple.

solidation single-handedly, *and* to impose it on the rest of the world in a [monopolistic](#) way. The cyber-physical domains do not have such monopolistic giants, so, it is up to the community to put its act and hands together, and to create the **digital platforms** for the domains. That platform must be functionally effective and efficient, **commercially fair and exploitable**, and offered to the world as a set of extremely **composable** modules. Those modules' further development and maintenance can be shared in the open, while their focused exploitation can support the creation of innovation-driven (non-giant) companies. These composable modules are not only software libraries and components, but also models and documentation, as well as tools and *best practice* architectural patterns.

This document provides the foundations for the **modelling** of “**motion stacks**” for **component-based** robotic **systems**, including **control**, **perception**, **monitoring**, **task plan** and **world model** representations, functionalities, capabilities and activities.

Such “stacks” are essential parts of any **digital platform** for robotic systems, and their design has a direct impact on how various stakeholders can contribute to, exploit, and regulate such a platform, as well as the applications built on top of it.

A “component” is any type of computer-readable formal representation of the **composition** of smaller parts into a bigger whole. This can be as abstract as the composition of knowledge relations into a “knowledge base”, and as concrete as deploying executable software code into a process on an operating system. The ambition of the document is to explain how to do composition, and what are its best practices, irrespective of the form in which the composition will be used on a computer.

Most of the material introduced in this document is not restricted to the robotics domain only, since it applies to all so-called [cyber-physical systems](#) (CPS), for which engineers want to control parts of the physical world via [information and communications technology](#) (ICT). The major difference with “purely digital” ICT platforms (e.g., distributed financial databases, social media applications, e-commerce platforms) is that CPS directly impact (“control”) the real physical world, and this brings in a lot of extra constraints on its ICT components, more in particular the need for realtime feedback loops which include physical components that come with a dynamical behaviour that has not been designed by the system engineers.

The first part of the modelling focuses on the **generic foundations**, that the domain of robotics shares with a lot of other domains. More in particular, we need formal representations of (i) **entities**, **relations** and **property** graphs, (ii) at the levels of abstraction of [mereology](#), [topology](#), [geometry](#), and [dynamics](#), (iii) with separation of the concerns of **mechanism** (structure & behaviour) and **policy**, and (iv) with an explicit ambition to support the grounding of **knowledge** and **information** into **software** and **data** implementations, and to support the **reasoning** processes to create, compose, configure, validate and certify components and systems.

Two core [\(meta\) meta models](#) of the presented approach are: (i) the **Block-Port-Connector** paradigm to represent *all* structural relationships and compositions, and (ii) a formal model to represent the **Composition** of **data** structures, **functions** and **control flow** schedules in the algorithms that provide the behaviour of a computer-controlled engineering system.

The core [methodology](#) to link models to executed actions is **hybrid constrained optimization**. The first step in the methodology is to use the above-mentioned models 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:

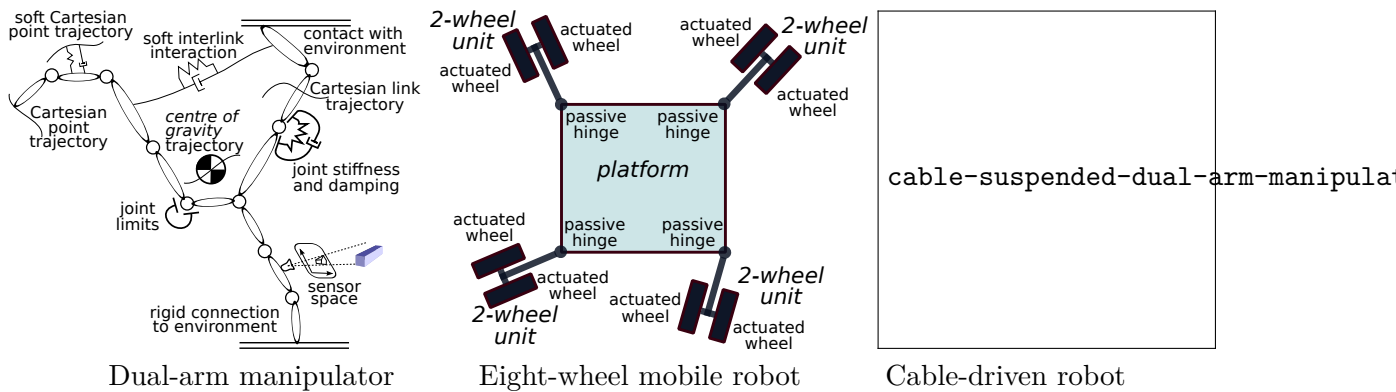


Figure 2: Sketches of various “advanced” robotic systems whose modelling is covered in this document.

symbolic constraint satisfaction (that is, **reasoning** on the “knowledge relations” that populate the application’s **context**), **continuous** constrained optimization (in “metric” domains like time, space, force, energy, . . .), and **discrete** optimization (that is, 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 the actuation setpoints to apply to the system. Very few application contexts *require* that the executed action is indeed the most optimal that exists, and are happy with a **satisfactory** solution [?]. The solution need also not be completely computed before the system is allowed to start acting, since any *feasible* solution that is available can already be used to get the system started; not in the least because only *actions* (and not optimizations) can help the system controller to assess how well it is realising its objectives in the real world. In addition, there is typically enough time *during the system’s operation* to run further iterations of the solver towards more optimal/satisficing outcomes.

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**”) [?]: a subsystem is called “stable” if it provides a “good enough” trade-off between (i) the **quality of the services** it delivers to the application, (ii) the **use of resources** it requires to provide those services, and (iii) its **robustness** against the disturbances that the application’s context will bring to the system.

The second part of the modelling brings in **robotics-specific** material, more specifically about the role of the “world model” as the sole coupling between (the models of) “plan”, “control” and “perception”, at any level of abstraction of a robotic **task**, Fig. 3:

- **world modelling**: what (uncertain) information does the robot system have available about how the world actually looks like?
- **plan**: how would we like the world to be changed by the robot system?
- **control**: what actions does the robot system undertake to realise the planned changes?
- **perception**: how can data provided by sensors be processed into information to update the world model?
- **monitoring**: which functions of the sensor data must be monitored to raise events when certain thresholds are exceeded?

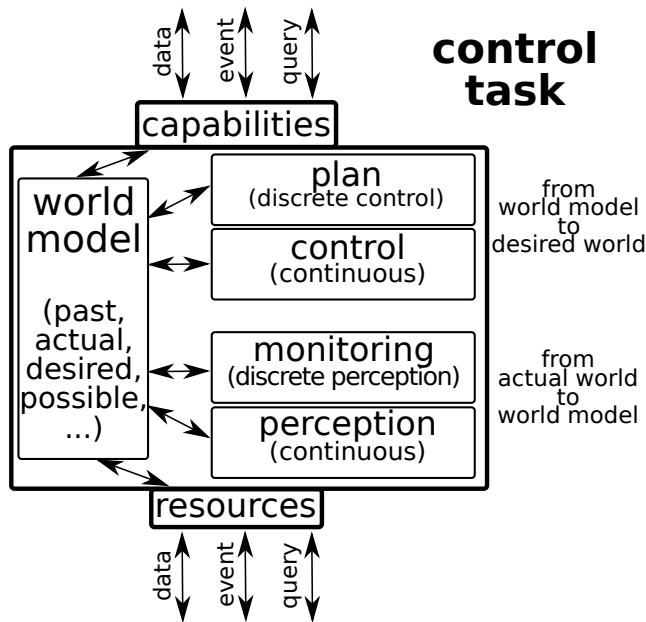


Figure 3: This figure sketches the composition of the different parts needed to realise a “task”. The arrows and the rectangles represent the composition primitives of *interconnection* and *containment*. Note that the figure is *not* representing a software component, but it represents the *structure* via which the parts are composed into a whole.

The **task** model describes the *capabilities* that are expected from the robot (e.g., to slide its hand over a table surface, or to drive through corridors in buildings), and the *resources* that it has available to realise those capabilities (e.g., actuators with limited torque generation power, or fingers with limped mechanical impedance and finite deformation limits). These capabilities and resources represent the **context** that provides meaning to the “*magic numbers*” in the models of the world, plan, control and perception.

The “top levels” of a motion stack model consist of geometrical entities, relations and constraints, from points and lines to **kinematic chains** with shape, **inertia**, toolings and sensors; for example, a dual-arm **mobile manipulator**, or a six-wheel planetary rover with **rocker bogies**. The “bottom levels” model the links between the kinematic chains and the actuators and the energy sources that must drive the chains’ motions. The major “behavioural” functionality to transfer energy between actuators and kinematic chain “**end effectors**” is that of the *hybrid dynamics solver*; it can compute all instantaneous motion and force transformations between the actuator space(s) and the Cartesian space(s) of all kinematic chains. Similarly to the **graph-based** models of kinematic chains, **perception graph** models compose sensors, sensing features, **data association** to object features, and constraints imposed by the task, the environment and the object properties; the generic “solver” in that context is **message passing over factor graphs**, playing a similar foundational role in adding behaviour to the models in perception as the hybrid dynamics algorithm does for motion of kinematic chains. The world model stack is a knowledge-based system, providing the **semantic entities and relations** to link the world model **information to the data** used in motion (planned and controlled) and perception.

The generic and robot specific model families share the same formalisation of their **mathematical**, **numerical** and **digital** representations, as well as the connection to **metadata** for **physical dimensions** and **units**.

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*, such that the development of models, tools, implementations and applications becomes methodological, transparent and scalable, but also stimulates 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 really 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 explain** what they are doing, **to reason** about their actions, 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.

Composable and explainable systems-of-systems: best practices and knowledge-based models for resilient holonic architectures in robotics and other cyber-physical systems

*“It’s simple, though not easy. . . ”*¹

Herman Bruyninckx

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

with contributions and ideas² by

Enea Scioni, Nico Hübel, Filip Reniers, Marco Frigerio (KU Leuven, Belgium)

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

René van de Molengraft (TU Eindhoven, the Netherlands)

2019-06-28

¹*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, or even dangerous.

²HB remains responsible for erroneous information and claims in this document.

Executive summary

Cyber-physical systems consist of multiple sub-systems that **interact** with the physical world, but also with each other. They exchange **matter**, **energy** and **data**, but also more and more **information** and **knowledge**, formally represented as “data”. The ambition of this document is to find the **least amount** of **formally encoded knowledge models**¹ that are needed to **represent** all possible systems in such a way that, they can **be controlled** in **predictable**, **resilient** and **explainable** ways,² and still be **re-composable** in any type of **system architecture**.

This document is inspired by the **best practices** to be found in many successful and resilient societies and organisations created by humanity, and in the vast amount of **knowledge** in **physics**, **mathematics**, **computer science**, and **systems** and **control** theory.

Robotic systems are the primary application target. This document advocates the design approach that considers the simplest robotic system consists already of *multiple* robots, each executing *multiple* tasks at the same time, for many of which they have to cooperate and to share resources, and with all their behaviour and actions realised by means of *multiple asynchronous* software activities.

The **system architecture** guidelines in this document build upon the **meta model** of the **task**: the **world model** plays the role of the only activity in the system that provides loose coupling between the other essential activities of **discrete and continuous control**, and **discrete and continuous perception**. Such **information architectures** must be made at various **levels of abstraction** (actuator control, proprioceptive and exteroceptive platform control) and their specification as **hybrid constrained optimization problems** supports **vertical and horizontal composition**, and deployment to heterogeneous software and hardware implementations.

This document explains how to use the 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.

¹This is the “*simple*” ambition of this document: each individual knowledge model is small enough to understand and comprehend fully in half an hour.

²This is the “not easy” part: it takes a lot of effort and iterations to get the envisaged integration of many simple things right, that is, correct, effective and efficient.

After more than 50 years of evolution, the robotics domain has created a large amount of insights, technology, models and (open source) software. But most of those efforts have still to be consolidated into commonly supported and standardized components, with *best practice* architectures that can *guarantee* safe, secure, efficient and effective operation of robotics systems. From the systems perspective, the evolution and the *state of the practice* in robotics is very similar to that of other domains where ICT platforms play an ever increasing role: energy production and distribution, multi-modal logistics and traffic, manufacturing, medical instruments, etc.; this document uses the term *cyber-physical systems* for its Chapters and Sections that do not contain any knowledge that is specific to the robotics domain.

At the highest level of modelling abstraction, a cyber-physical **system** is a set of **activities** that provide **behaviour**, via which they change their own **state** and/or that of the **resources** they have to **share** with other Activities, via **interactions**. The words in bold are the core entities and relations, and they can be realised in the **physical** world (e.g., the electro-mechanical behaviour of robots or cars, the measurement principles behind sensors, the chemistry in a battery) as well as in the **cyber** world (that is, the composition of computational and communication hardware and software); the **transformations** between both worlds works via *transducers* that bring their own dynamics and interaction into the system design.

The **design** of all cyber-physical **applications** have some major challenges in common, which form the *theme* of this document: *composability*, *self-reflection*, *reactivity* and *explainability*. In ideal system design, the latter one should be a consequence of the former two, as soon as the design of components takes system-level integration and self-reflection into account as primary *design drivers*, and as an intermediary goal towards self-explainability of ICT platforms.

The size and the scale of integration of robotic systems grow beyond what single developers or development teams, can comprehend, create, validate and maintain. Contrary to what has happened in the ICT-driven applications that could be built on top of “the Web” platform, no “*giant companies*”³ have been created that have the funding and man power to realise such consolidation single-handedly, *and* to impose it on the rest of the world in a *monopolistic* way. The cyber-physical domains do not have such monopolistic giants, so, it is up to the community to put its act and hands together, and to create the **digital platforms** for the domains. That platform must be functionally effective and efficient, **commercially fair and exploitable**, and offered to the world as a set of extremely **composable** modules. Those modules’ further development and maintenance can be shared in the open, while their focused exploitation can support the creation of innovation-driven (non-giant) companies. These composable modules are not only software libraries and components, but also models and documentation, as well as tools and *best practice* architectural patterns.

This document provides the foundations for the **modelling** of “**motion stacks**” for **component-based** robotic **systems**, including **control**, **perception**, **monitoring**, **task plan** and **world model** representations, functionalities, capabilities and activities.

Such “stacks” are essential parts of any **digital platform** for robotic systems, and their design has a direct impact on how various stakeholders can contribute to, exploit, and regulate such a platform, as well as the applications built on top of it.

A “component” is any type of computer-readable formal representation of the **compo-**

³The so-called *GAFA giants*: Google, Amazon, Facebook and Apple.

sition of smaller parts into a bigger whole. This can be as abstract as the composition of knowledge relations into a “knowledge base”, and as concrete as deploying executable software code into a process on an operating system. The ambition of the document is to explain how to do composition, and what are its best practices, irrespective of the form in which the composition will be used on a computer.

Most of the material introduced in this document is not restricted to the robotics domain only, since it applies to all so-called **cyber-physical systems** (CPS), for which engineers want to control parts of the physical world via **information and communications technology** (ICT). The major difference with “purely digital” ICT platforms (e.g., distributed financial databases, social media applications, e-commerce platforms) is that CPS directly impact (“control”) the real physical world, and this brings in a lot of extra constraints on its ICT components, more in particular the need for realtime feedback loops which include physical components that come with a dynamical behaviour that has not been designed by the system engineers.

The first part of the modelling focuses on the **generic foundations**, that the domain of robotics shares with a lot of other domains. More in particular, we need formal representations of (i) **entities**, **relations** and **property** graphs, (ii) at the levels of abstraction of **mereology**, **topology**, **geometry**, and **dynamics**, (iii) with separation of the concerns of **mechanism** (structure & behaviour) and **policy**, and (iv) with an explicit ambition to support the grounding of **knowledge** and **information** into **software** and **data** implementations, and to support the **reasoning** processes to create, compose, configure, validate and certify components and systems.

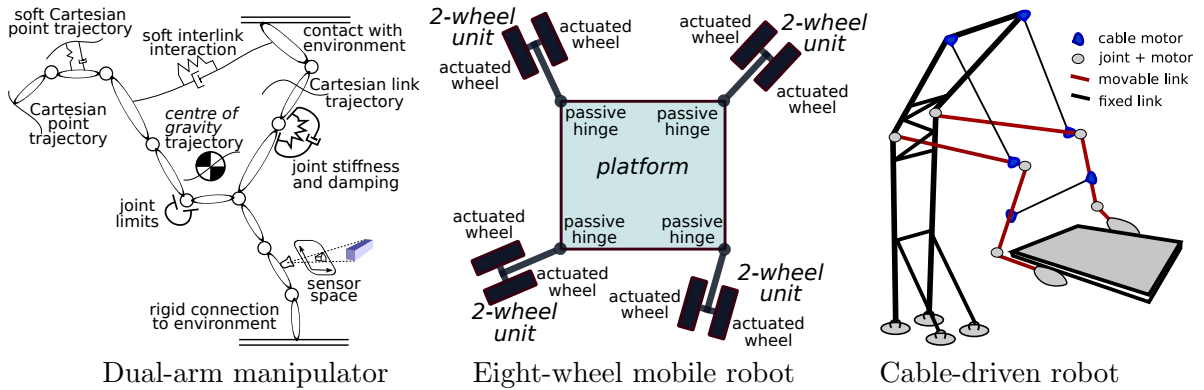


Figure 1: Sketches of various “advanced” robotic systems whose modelling is covered in this document.

Two core (meta) meta models of the presented approach are: (i) the **Block-Port-Connector** paradigm to represent *all* structural relationships and compositions, and (ii) a formal model to represent the **Composition** of **data** structures, **functions** and **control flow** schedules in the algorithms that provide the behaviour of a computer-controlled engineering system.

The core methodology to link models to executed actions is **hybrid constrained optimization**. The first step in the methodology is to use the above-mentioned models 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 (in “metric” domains like time, space, force, energy, . . .), and **discrete** optimization (that is, 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 the actuation set-points to apply to the system. Very few application contexts *require* that the executed action is indeed the most optimal that exists, and are happy with a **satisfactory** solution [77]. The solution need also not be completely computed before the system is allowed to start acting, since any *feasible* solution that is available can already be used to get the system started; not in the least because only *actions* (and not optimizations) can help the system controller to assess how well it is realising its objectives in the real world. In addition, there is typically enough time *during the system’s operation* to run further iterations of the solver towards more optimal/satisficing outcomes.

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**”) [47]: a subsystem is called “stable” if it provides a “good enough” trade-off between (i) the **quality of the services** it delivers to the application, (ii) the **use of resources** it requires to provide those services, and (iii) its **robustness** against the disturbances that the application’s context will bring to the system.

The second part of the modelling brings in **robotics-specific** material, more specifically about the role of the “world model” as the sole coupling between (the models of) “plan”, “control” and “perception”, at any level of abstraction of a robotic **task**, Fig. 2:

- **world modelling**: what (uncertain) information does the robot system have available about how the world actually looks like?
- **plan**: how would we like the world to be changed by the robot system?
- **control**: what actions does the robot system undertake to realise the planned changes?
- **perception**: how can data provided by sensors be processed into information to update the world model?
- **monitoring**: which functions of the sensor data must be monitored to raise events when certain thresholds are exceeded?

The **task** model describes the *capabilities* that are expected from the robot (e.g., to slide its hand over a table surface, or to drive through corridors in buildings), and the *resources* that it has available to realise those capabilities (e.g., actuators with limited torque generation power, or fingers with limped mechanical impedance and finite deformation limits). These capabilities and resources represent the **context** that provides meaning to the “*magic numbers*” in the models of the world, plan, control and perception.

The “top levels” of a motion stack model consist of geometrical entities, relations and constraints, from points and lines to **kinematic chains** with shape, **inertia**, toolings and sensors; for example, a dual-arm **mobile manipulator**, or a six-wheel planetary rover with **rocker bogies**. The “bottom levels” model the links between the kinematic chains and the actuators and the energy sources that must drive the chains’ motions. The major “behavioural” functionality to transfer energy between actuators and kinematic chain “**end effectors**” is that of the

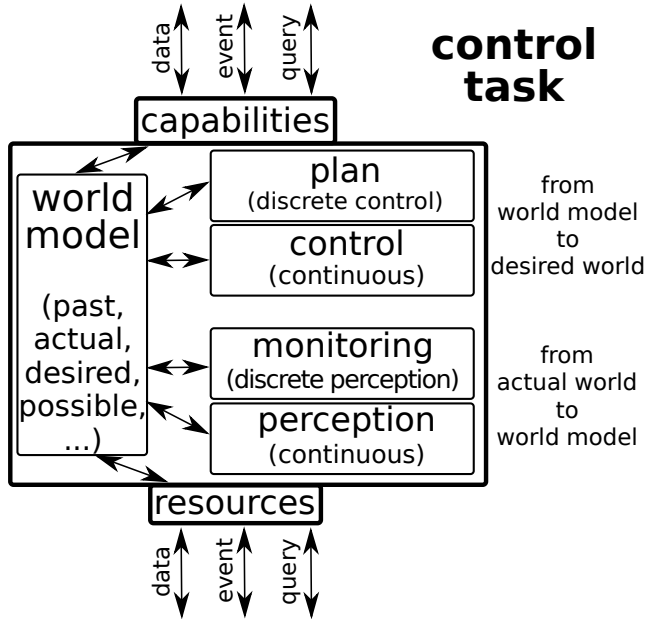


Figure 2: This figure sketches the composition of the different parts needed to realise a “task”. The arrows and the rectangles represent the composition primitives of *interconnection* and *containment*. Note that the figure is *not* representing a software component, but it represents the *structure* via which the parts are composed into a whole.

hybrid dynamics solver; it can compute all instantaneous motion and force transformations between the actuator space(s) and the Cartesian space(s) of all kinematic chains. Similarly to the [graph-based](#) models of kinematic chains, [perception graph](#) models compose sensors, sensing features, [data association](#) to object features, and constraints imposed by the task, the environment and the object properties; the generic “solver” in that context is [message passing over factor graphs](#), playing a similar foundational role in adding behaviour to the models in perception as the hybrid dynamics algorithm does for motion of kinematic chains. The world model stack is a knowledge-based system, providing the [semantic entities and relations](#) to link the world model [information to the data](#) used in motion (planned and controlled) and perception.

The generic and robot specific model families share the same formalisation of their **mathematical**, **numerical** and **digital** representations, as well as the connection to [metadata](#) for [physical dimensions](#) and [units](#).

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*, such that the development of models, tools, implementations and applications becomes methodological, transparent and scalable, but also stimulates 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 really 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 explain** what they are

doing, **to reason** about their actions, 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
1 Foundations of knowledge-driven engineering: meta modelling	17
1.1 Models for science and engineering	18
1.2 Modelling knowledge with property graphs	18
1.2.1 Mechanism: property graph for entities, relations and constraints	19
1.2.2 First-order relations	19
1.2.3 Higher-order relations	20
1.2.4 Properties and attributes	23
1.2.5 Policy: <code>semantic_ID</code> meta data	23
1.2.6 Mechanism for reasoning: property graph traversal	24
1.2.7 Policy: constraint, dependency and causality graphs	25
1.2.8 Best practices: quasi non-existent	25
1.2.9 Storage and reasoning in property graph databases	25
1.2.10 Host languages to store, exchange and query models	26
1.3 Mereology: <code>has-a</code>	28
1.3.1 Mereology: <code>has-a</code>	28
1.3.2 Topology: <code>contains</code> , <code>connects</code>	29
1.3.3 Role of mereology-topological models	30
1.4 Core relations in modelling: <code>conforms-to</code> and <code>is-a</code>	30
1.4.1 <code>conforms-to</code> hierarchy on relations	30
1.4.2 <code>is-a</code> hierarchy on properties	31
1.4.3 Composition and inheritance	31
1.4.4 Best practice: four levels in <code>is-a</code> and <code>conforms-to</code> hierarchies	31
1.5 Meta (meta) models and DSLs	32
1.6 Mechanism and policy	33
1.6.1 Policy: frameworks, middleware, solvers	34
1.6.2 Bad practice: interpreting attributes as properties	34
1.7 Declarative and imperative	34
1.8 Hierarchical and serial ordering: <code>scope</code>	35
1.9 Structural composition: <code>block-port-connector</code>	36
1.10 Design patterns	37
1.11 Dependency graphs	37
1.11.1 Mechanism: Directed Acyclic Graphs for partial ordering	38
1.11.2 Policy: temporal order, hierarchy, causality	38

2	Meta meta models for (robotic & cyber-physical) systems	39
2.1	Cyber-physical systems: interaction of matter, energy, information & data .	40
2.1.1	State of a system	41
2.1.2	State representation: ordinal, categorical, continuous, discrete, event	41
2.2	Taxonomy of scientific theories — Levels of representation	41
2.2.1	Mathematical representation	42
2.2.2	Abstract data type	42
2.2.3	Data structure	43
2.2.4	Digital storage	43
2.2.5	Electronic processing	43
2.3	Functions: composition of computations into algorithms	44
2.3.1	Mereotopological model	44
2.3.2	Mechanism: function, data, schedule, invariant and algorithm .	44
2.3.3	Computational state of an algorithm — Stack	46
2.3.4	Operational status of an algorithm — Flag	46
2.3.5	Policy: flags to coordinate synchronous schedules	47
2.3.6	Policy: closure in a context	47
2.3.7	Policy: deployment in BPC component	47
2.3.8	Policy: application programming interface	48
2.3.9	Policy: dataflow	48
2.3.10	Policy: functional programming	48
2.4	Activities: composition of algorithms into behaviour	49
2.4.1	Mechanism (meta model): event loop computations	49
2.4.2	Behavioural state of an activity — Mode	50
2.4.3	Policy: sequential, concurrent and distributed execution	50
2.4.4	Policy: work flow	51
2.5	Interfaces: interactions between behaviour in activities	51
2.5.1	Semantics of exchange instrument: buffer, queue, channel, socket	51
2.5.2	Semantics of continuous, discrete and symbolic information: data, events, and queries	53
2.5.3	Semantics of exchange direction: uni-directional and bi-directional . .	54
2.5.4	Patterns: Publish-Subscribe, Producer-Consumer, Request-Reply . .	54
2.5.5	Mechanism: Producer-Consumer stream	55
2.5.6	Policy: peer_activity and flow_control status flags	56
2.5.7	Composition of Producer-Consumer streams	57
2.5.8	Composition of data and meta-data streams	57
2.5.9	Submission-Completion streams	58
2.5.10	Publish-Subscribe & Request-Reply as stream architectures	58
2.5.11	Policy: message broker	59
2.5.12	Interaction state — Interface protocol phase	59
2.5.13	Mechanism: Conflict-Free Replicated Data Type (CRDT)	59
2.5.14	Mechanism: immutable data type	59
2.6	Components: composition patterns for activities and their interactions . . .	59
2.6.1	The 5C's: composition of behavioural roles	60
2.6.2	Bad practices	62
2.6.3	Mechanism: "5Cs"	63
2.6.4	Policy: separation of concerns in/between platform and application .	63

2.6.5	Policy: vendor-centric added value in Configuration, Coordination, Composition	63
2.6.6	Policy: Coordination, Orchestration, Choreography	63
2.7	Tasks: composition of behaviours for control, perception and world modelling	64
2.7.1	Mechanism: capability, resource, world model, plan, control, monitor, perception	65
2.7.2	Task meta model as semantic database	66
2.7.3	Policy: coupling via shared world model	67
2.7.4	Policy: continuous, discrete and symbolic Task models	68
2.7.5	Policy: hybrid constrained optimization problem specification (HCOP)	68
2.7.6	Policy: declarative solvers and imperative algorithms	68
2.7.7	Policy: sharing data, events, models	68
2.7.8	Policy: mission, service, skill, function	69
2.7.9	Vertical and horizontal composition	69
2.8	Event loop pattern: composition of asynchronous computational behaviour	71
2.8.1	The role of the event loop	71
2.8.2	“5C”-based programme template for an event loop	72
2.8.3	Mechanism: callback, source, event, context, poll, dispatch, pool	73
2.8.4	Policy: memory allocation of source, context, queues, workers	75
2.8.5	Policy: priorities	75
2.8.6	Policy: mediation	75
2.8.7	Policy: context deployment in threads	75
2.8.8	Policy: integration into software components	75
2.8.9	Software pattern realisations	76
2.8.10	Software anti-pattern realisations	77
2.9	Solver pattern: from declarative specification to imperative algorithm	77
2.9.1	Mechanism: relation, constraint, dependency graph, spanning tree, action, solver sweep	78
2.9.2	Policy: task-based hybrid constrained optimization	79
2.9.3	Policy: dynamic programming	79
2.9.4	Policy: static spanning tree pyramid	79
2.9.5	Policy: feasible and optimal solutions	79
2.9.6	Policy: sweep scheduling	80
2.9.7	Policy: tolerances	80
2.10	Finite state machine to coordinate activity modes	80
2.10.1	Structure & behaviour: state, transition, event reaction table	80
2.10.2	Higher-order FSM model: pre, per and post conditions	81
2.10.3	Mechanism of event handling: event queue, event processing, event monitoring, event loop	82
2.10.4	Mechanism: activity Coordination behaviour via monitors and callbacks	83
2.10.5	Policy: selection, priority, deletion	84
2.10.6	Constraints on event handling meta model	84
2.10.7	Policy: hierarchical state machine	85
2.10.8	Best Practice: Life Cycle State Machine for single activity deployment	86
2.10.9	Best Practice: FSM for behavioural state, Flag for operational status	87

2.10.10	Best Practice: Flag, Traffic Light, Petri Net, for inter-activity coordination	87
2.10.11	Bad and Good Practices in FSM usage	89
2.10.12	Transition systems for the runtime creation of FSMs	89
2.11	Data, uncertainty and information	90
2.11.1	Mechanism: Bayesian probability axioms	90
2.11.2	Mechanism: Bayes' rule for optimal transformation of data into information	90
2.11.3	Policy: belief propagation	90
2.11.4	Policy: hypothesis tree for semi-optimal information processing	91
2.11.5	Policy: mutual entropy to measure change in information	91
2.12	Formal representation languages: mature standards	91
2.12.1	QUDT and UCUM	91
2.12.2	JSON and JSON-Schema	92
2.12.3	JSON-LD	92
2.12.4	RDF1.1	93
2.12.5	Abstract Syntax Notation One (ASN.1)	93
2.12.6	Hierarchical Data Format — HDF5	93
2.12.7	FlatBuffers, Protocol Buffers	93
2.12.8	BLAS, LAPACK	93
2.12.9	DFDL	94
2.12.10	XML Schemas	94
2.12.11	Differential geometry	94
3	Meta models for the geometry of rigid bodies and polygonal worlds	95
3.1	The meta meta models	95
3.1.1	<code>semantic_IDs</code> for geometry in 1D, 2D and 3D	96
3.1.2	Geometric relations in projective, affine and Euclidean spaces	97
3.1.3	Non-Euclidean space for rigid bodies	98
3.1.4	Differential geometry	99
3.1.5	Qualitative spatial relations	99
3.2	Taxonomy of geometric meta models	99
3.3	Levels of representation in geometry	102
3.4	Meta models of <code>Point-Polyline-Polygon</code> geometry	103
3.4.1	<code>Point</code> entity and its composition relations	104
3.4.2	Extra composition relations in 3D	107
3.4.3	<code>Position</code> and <code>Motion</code> relations of a <code>Point</code>	108
3.4.4	<code>Position</code> and <code>Motion</code> relations of a <code>Rigid_body</code>	109
3.4.5	Constraint relations on mereo-topological entities	112
3.4.6	Abstract data types for <code>Coordinates of Position and Motion</code>	113
3.4.7	Operators on <code>Coordinates</code>	117
3.4.8	Data structures for <code>Coordinates</code>	118
3.4.9	Constraint relations on <code>Coordinates</code> — <code>Geometric_chain</code>	118
3.5	Meta models of mechanical dynamics: composing force and motion	120
3.5.1	Abstract data types and data structures for dynamics	120
3.5.2	Motion as result of constrained optimization — Gauss' Principle	121
3.5.3	Energy relations — Bond Graphs	121

3.5.4	Differential geometry: manifold, (co)tangent space, linear forms . . .	122
3.6	Composition relations: Map as set of geometric entities	124
3.7	Composition relations: Semantic_map , World_model	126
3.8	Uncertainty in geometric entities and relations	127
3.8.1	Sources of uncertainty	127
3.8.2	Covariance of a Frame has no meaning	128
3.9	De facto meta model standards for Coordinates	129
3.9.1	ROS Messages	130
3.9.2	RTT/Orocos typekits	131
3.9.3	SmartSoft Communication Object DSL	131
4	Meta models for a kinematic chain and its instantaneous motion	132
4.1	Meta models	132
4.1.1	Mereology	132
4.1.2	Types	133
4.1.3	Coordinates and behavioural state for Motion	135
4.1.4	Geometrical operations — Forward, inverse and hybrid kinematics . .	136
4.1.5	Inconsistency, redundancy, singularity	138
4.2	Meta model for the mechanical dynamics of a kinematic chain	139
4.2.1	Coordinates for dynamics state	139
4.2.2	Coordinates and behavioural state for impedance	139
4.2.3	Geometrical operations revisited — The role of “virtual dynamics” .	140
4.2.4	Dynamic relations under a 5D motion constraint	140
4.3	Meta model for instantaneous motion of kinematic chains	141
4.3.1	Gauss’ Principle of Least Constraint	141
4.3.2	Specification of instantaneous motion	142
4.3.3	Operations: forward, inverse and hybrid dynamics transformations . .	143
4.4	Composition and decomposition of Kinematic_chains	144
4.4.1	Composition — Serial, branch, loop	144
4.4.2	Decomposition — Spanning tree	146
4.4.3	Policy: iteration via sweeps	147
4.4.4	Policy: input-output causality assignment	147
4.5	Taxonomy of kinematic chain families	147
4.5.1	Serial chains	148
4.5.2	Mobile platform chains	148
4.5.3	Parallel chains	149
4.5.4	Multirotor chains	149
4.5.5	Hybrid chains	149
4.5.6	Cable-driven chains	149
4.6	Solver for hybrid kinematics and dynamics	149
4.6.1	Mechanism–I: motion drivers	150
4.6.2	Mechanism–II: procedural sweeps	151
4.6.3	Policy: scheduling options in the third sweep	152
4.6.4	Policy: free-floating base	153
4.6.5	Policy: tasks in the mechanical domain	153
4.6.6	Policy: deployment in an event loop	154
4.6.7	Composition with dynamics of resources	154

4.6.8	Composition with dynamics of perception and world model	154
4.6.9	Composition with motion trajectories	155
4.6.10	Dynamics of serial kinematic chain	155
4.6.11	Dynamics of branched kinematic chain	155
4.6.12	Dynamics of kinematic chain with a loop	155
5	Meta models for robotic tasks	156
5.1	Simplest task: proprioceptive guarded motion	156
5.2	Action, actor, actant (object), activity, agent	158
5.3	Natural hierarchies in task models	159
5.3.1	Hierarchy in motion capabilities	159
5.3.2	Hierarchy in world model scope	159
5.3.3	Hierarchy in perception graph	161
5.3.4	Hierarchy in cascaded control levels	161
5.3.5	Hierarchy in energy transformations	162
5.4	Mechanism: task specification as constrained optimization problem	163
5.4.1	Policy: specify as objective function or as constraint	164
5.4.2	Policy: integrating multiple levels of abstraction in one task	165
5.4.3	Policy: event loops for task execution	165
5.4.4	Policy: monitoring for hybrid constrained optimization	165
5.4.5	Policy: iterating feasible solutions during task execution	166
5.5	Bad practices in Task specification	167
5.6	Task examples: indoor and outdoor robot driving	167
5.6.1	Geometric world models for control & perception integration	168
5.6.2	Example task plan: escape from a room	170
5.7	Semantic task specification: domain-specific languages	174
5.7.1	Task ontology	174
5.7.2	The importance of task ontology standardization	174
5.7.3	Task as composition of natural and artificial constraints	175
5.7.4	Task meta model realises the “dependency inversion principle”	175
5.7.5	Semantic mobile robot motion primitives	176
5.7.6	Semantic robot arm motion primitives	177
5.8	Vertical and horizontal composition of Tasks	177
6	Meta models for world modelling and its integration in tasks	179
7	Meta models for control of continuous time behaviour	180
7.1	Mereology of control behaviour: feedback, feedforward, and adaptation	180
7.2	Mechanism: state and system dynamics relations	182
7.3	Policy: model-reference adaptive control (MRAC)	184
7.4	Policy: model-predictive control (MPC)	184
7.5	Mechanism: setpoint, trajectory, path and tube control inputs	185
7.6	Policy: control progress objective	185
7.7	Policy: PID, sliding mode, gain scheduling and ABAG	185
7.8	Mechanism: cascaded control loops	187
7.9	Mechanism: asynchronous distributed control	187
7.10	Policy: optimal control	189

7.11	Policy: behaviour tree for semi-optimal control	189
7.12	Event loops revisited: control behaviour composition	189
7.12.1	Example: one-dimensional position control	191
7.12.2	Policy: hybrid event control	192
7.12.3	Policy: throughput and latency	193
7.12.4	Policy: realtime activities via the “multi-thread” software pattern	193
7.12.5	Policy: event loops for Task control	195
8	Meta models for perception and its integration in tasks	196
8.1	Mereo-topological meta model: the natural hierarchy in robotic perception	197
8.2	Policy: (data) association	198
8.3	Perception example: robots driving in traffic	198
8.3.1	Escape from a room	198
8.3.2	Ego-motion estimation with accelerometer, gyro and encoder	199
8.3.3	Ego-motion estimation with visual point and region features	199
8.4	Mechanism: Bayesian information theory	200
8.5	Geometrical semantics in perception	202
8.6	Dynamical semantics in perception	202
8.7	Policy: tracking, localisation, map building	202
8.8	Mechanism of information update: Bayes’ rule	203
8.9	Mechanism of perception solver: message passing over junction trees	204
8.10	Policy: dynamic Bayesian network	204
8.11	Policy: Factor Graphs	206
8.12	Mechanism: composition of point and region features	206
8.13	Policy: feature pre-processing	206
8.14	Policy: deployment in event loop	206
9	Meta models for holonic, resilient & explainable system architectures	207
9.1	Robustness, resilience and explainability as system design drivers	208
9.2	Architectures: hierarchy, heterarchy and holarchy	209
9.2.1	Information architecture	209
9.2.2	Software architecture	210
9.2.3	Hardware architecture	211
9.2.4	Digital platform	211
9.2.5	Meta models for robotic stacks	211
9.2.6	“Sense-Plan-Act”, “Three-Level”, “Subsumption”	212
9.2.7	Role of middleware	213
9.3	Autonomy and decision making	213
9.3.1	Sheridan’s ten levels of system autonomy	213
9.3.2	Explanation levels for autonomous decision making	213
9.3.3	Links with horizontal and vertical integration	214
9.4	Safety	214
9.4.1	Best practice: safety PLC	214
9.5	Security	214

10	Meta models for information architectures	216
10.1	Running example: two-wheel driven mobile robot	217
10.1.1	Hardware architecture	218
10.1.2	Typical tasks	219
10.2	Pattern: task control at three levels of abstraction	219
10.2.1	Proprioceptive guarded motion holon	221
10.2.2	Exteroceptive guarded motion control holon	221
10.2.3	Map-based guarded control holon	222
10.2.4	Inter-holon stream architecture	222
10.3	Specialisations of Producer-Consumer stream model	222
10.3.1	Stream with heterogeneous data chunks	222
10.3.2	Composition of stream with object pool	223
10.3.3	Multiple producers, multiple consumers	223
10.3.4	Ownership, garbage collection, and blocking policies	224
10.4	Best practices	224
10.4.1	One LCSM per activity	224
10.4.2	Every entity and relation has one owner	225
10.4.3	Explicit causality in data access policy	225
10.4.4	Every task model is a shared resource	225
10.5	Pattern: mediation in peer-to-peer activity interactions	226
10.5.1	Mechanism: the mediator activity	226
10.5.2	The mediator pattern for task-resource trade-offs	227
10.5.3	The mediator pattern in human society	227
10.5.4	Resilience via higher-order dependency relations	228
10.6	Information architecture examples	229
10.6.1	Proprioceptive motor-wheel-platform control	229
10.6.2	Exteroceptive platform control	229
10.6.3	Safe interactive control over a network	229
11	Meta models for software architectures	230
11.1	Running example: mobile robot with two actuated wheels	231
11.2	Mechanism: implementing streams by coupling two ring buffers	231
11.2.1	Software design	232
11.2.2	Policy: composition of data and meta data streams	236
11.2.3	Policy: time series stream	236
11.2.4	Policy: composition of stream and memory pool	236
11.2.5	Policy: pipe line	237
11.2.6	Policy: heartBeat/watchDog mediation for “lazy” stream	237
11.2.7	Pattern: event loop with ring buffer and scatter-gather I/O	238
11.2.8	Policy: contiguous data for producer and for consumer	238
11.2.9	Best practices	238
11.2.10	Standards and software projects supporting streams	239
11.3	Mechanism: stream with wait-free maximal freshness	239
11.4	Policy: event loop mediation for multiple stream buffers	241
11.5	Mechanism (operating system): thread, process, shell, container, cloud	242
11.6	Pattern: composition of (a)synchronous threads	243
11.6.1	Software design	243

11.6.2	Bad practices	245
11.6.3	Good practices	245
11.6.4	Policy: mediation on shared resources	246
11.7	Mechanism: programming language operators	246
11.7.1	Async/await	246
11.7.2	Iterator	246
11.7.3	Maybe	246
11.7.4	Memory barrier	246
11.7.5	Atomic and lockfree operators	247
11.7.6	Bad practices with locks	247
11.7.7	Bad practices in communication	247
11.8	Mechanism: core, system-on-a-chip, computer, cloud	248
11.9	Policy: framework plug-ins versus library composition	248
12	Skill architectures for the composition of Tasks	249
13	Skill architectures for two-arm manipulator robots	250
14	Skill architectures for mobile robots	251
15	Skill architectures for cable robots	252

Chapter 1

Foundations of knowledge-driven engineering: meta modelling

Any **formal representation of knowledge** consists of **models** built on the **axiomatic foundation** of **entity** and **relation**: an entity represents “stuff”, “things”, “primitives”, “atoms”, . . . , and a relation represents a dependency between properties of entities [20]. The **meaning** of a model must/can be formally represented by relating the model to one or more **meta models**: the latter contain the representation of all the relations that constructs in the model must satisfy in order to be “well formed” and “meaningful”. 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; in common practice, one 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 in two complementary ways: the graphs that make up a knowledge system are **queried** via **graph matching** or **graph traversal**, with the latter supporting the **higher-order reasoning** required for **explainable** robotic systems.

This Chapter describes the modelling concepts in the core of any **knowledge-based** and/or **model-driven engineering**¹ approach towards a digital platform for the robotics domain, or for any other cyber-physical systems domain for that matter. **Major challenges** in doing the modelling “right” are (i) to define the **levels of abstraction** that are considered essential in a particular domain (that is, which entities and relations to include, and which ones to neglect), (ii) how **to switch between these levels** at the “right” moment, (iii) how **to create** knowledge bases for computers **to reason** on the models contained in those servers, (iv) the **formal representations** in “**host**” **languages** for encoding the knowledge, and (v) how to reach the **standardization** required for realistic multi-vendor interoperability.

¹For 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.

1.1 Models for science and engineering

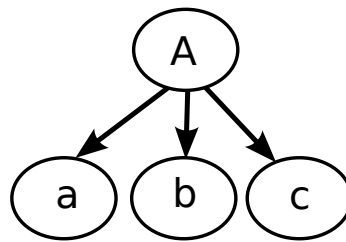
“Modelling” is the mental activity of the human scientist or engineer to make an artificial language to represent, in a formal way, the properties of (real-world as well as abstract) **entities** and of the **relations** between them. A model provides *structure* to a chosen **domain**, to make *explicit* which things are relevant in that domain, and how these relevant things influence each other. Hence, a “model” is a **collection** (or, “set”) of entities and relations to represent **scope** (“what is important?”), **interaction** (“what influences exist between entities?”), and **behaviour** (“how do the properties of entities and relations influence the behaviour of what they represent?”).

Scientists strive for models that allow **to analyse reality**; engineers strive for models with which they can **design artefacts** in such a way that the models can feed machines **to implement** the artefacts in the real world. Engineering models typically make use of scientific models; the inverse is never(?) the case.

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 a modelling activity is to represent **context** and **composition**, in such an explicit way that one can add new models to already existing ones without changing the meaning of the latter, or its formalisation. 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.

Figure 1.1: Directed graph, with anonymous edges between named nodes.



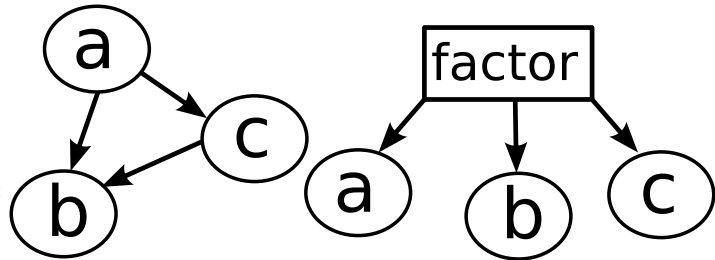
1.2 Modelling knowledge with property graphs

Knowledge is the **interconnection** between **data**, **information** and **meaning**, and formal representations of knowledge have been given names like **knowledge base**, **knowledge graph**, **semantic database**, or **ontology**. The **structural model** is a (directed) **graph**, Fig. 1.1, that is, nodes connected by edges, and each edge has a direction. The **mereological** part of the knowledge representation adds **meaning** to the graph’s nodes and edges. The **behavioural model** of a knowledge graph represents how the knowledge can be exploited

(activated, reasoned with,...) by selecting the parts of the knowledge graph required to answer “queries.”

Mereology, [holonomy](#) and [meronomy](#) are all very related terms, that can be used interchangeably, to denote the symbolic relations that humans understand to exist between “parts” and “wholes”. This document uses the term “mereologic” to represent this symbolic relation whenever *models* are considered, and the term “holonic” when [holarchy architectures](#) are considered (as alternative to the [hierarchical](#), [homoarchival](#) or [heterarchical](#) designs. The latter [interpretation](#) finds its origins in the seminal works by [Arthur Koestler](#) and [Herbert Simon](#). Its applications in robotics and manufacturing appeared in the 1990s [79, 81].

Figure 1.2: “Plain’ graph on the left, and hypergraph, or factor graph, on the right. The normal graph can only represent a relation between *two* nodes, while the hypergraph can represent *n-ary* relations between *more than two* nodes.



1.2.1 Mechanism: property graph for entities, relations and constraints

A [graph](#) (Fig. 1.1) is the simplest structure to model that some entities (represented by **nodes**) are related (represented by **edges**).

A [factor graph](#), or [hypergraph](#), (Fig. 1.2) extends the graph model with a **second type of node**, namely a node that can connect to *more* than two other “plain” nodes. Some domains call the extension a *factor node*, some call it an *hyperedge*, illustrating the fact that its semantics is equally well motivated by considering it as an extension of the “node” concept or as extension of the “edge” concept. Anyway, in the context of knowledge modelling, hyperedges represent **relations with any number of arguments**. For example, factor graphs can represent [S-expressions](#), which are the basis of [context-free languages](#). In the context of knowledge representation and reasoning, this is the foundation behind many computer languages that provide reasoning capabilities, such as [Prolog](#) or [Lisp](#). Examples: (i) for [queries](#), or (ii) for [algebraic](#) relations.

A [property graph](#) [6] extends the hypergraph model with a **third type of node** (or edge), the **property node**, to represent the **properties** of an entity node or a factor node (Fig. 1.3). Again, this addition is a cheap way to increase the semantic richness of a graph even further, without adding any structural complexity; it does add a structural *constraint*, in that a property node should only be connected to one single other (entity or relation) node.

1.2.2 First-order relations

This document adopts the **axiomatic** basis to represent [first-order knowledge models](#) with **property graphs**: **nodes** represent **entities**, **edges** represent **relations**, and both have **properties**. 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. This step from normal graphs to factor graphs allows richer semantic expres-

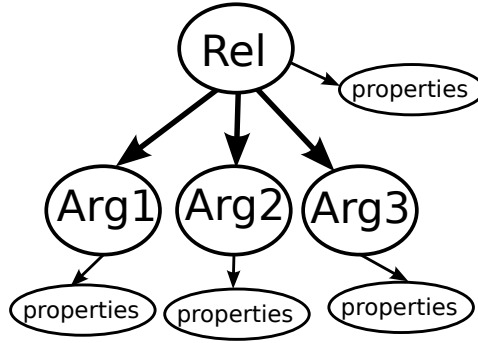


Figure 1.3: Property graph, with an edge linking every named node to a node containing **key-value pairs** (or any **abstract data type** for that matter), representing the properties of the named nodes. The property graph is a *directed graph* representation of the **mereological** model of a relation with three arguments, all with their own properties. The entity **Rel** has three other entities as its parts, **Arg1**, **Arg2** and **Arg3**, and each of them has a **properties** entity as a part. The arrows are seemingly still anonymous, but each arrow does represent a particular **has-a** relation, and hence has an *identifier* of that relation as its *property*. These identifiers have been omitted in the drawing, because they depend on the concrete relation that is modelled, and that is information that is missing in this conceptual sketch.

sivity, without the need to add any new structural primitive, and with just a small extra complexity in the bookkeeping of the type of node.

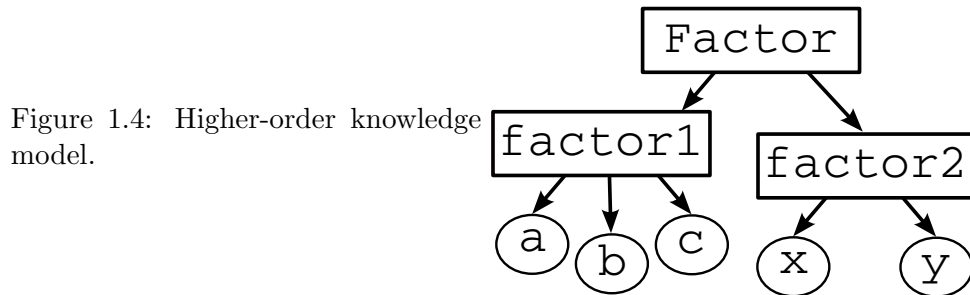


Figure 1.4: Higher-order knowledge model.

1.2.3 Higher-order relations

A second **axiomatic** basis of knowledge representation is that all application contexts of realistic complexity need to represent not only first-order but also **higher-order** relations because every application needs a **context** in which **to interpret** entities and relations (Fig. 1.4), [37, 63, 72]. Of course, 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.

From a **structural** point of view, an higher-order relation just represents a *topological connection* between the properties of other entities and/or relations; from a **semantic** point of view, the higher-order relation encodes **knowledge** about **why** the connected properties are connected, about **how** their connection must be used, about **what variations** are allowed, etc. Generally speaking, a higher-order relation models the **role** that each of its argument

plays in the relation, given the specific context in which the higher-order relation is created.

(TODO: theory of mind [89]: what do agents know about what other agents know? [action languages](#), [transition system](#), [abstract rewriting systems](#), OMG's M0–M3 meta modelling, [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.)

Examples of higher-order relations

- *intent and configuration of tasks at various levels of control*: every robotic system has motors to create motion, and the control of those motors is driven by a particular goal of the application (the “intent” of the task). The intent, *together* with the “capabilities” of the robot system, result in relations from which the *configuration* of a control algorithm is to be derived. There are various “levels” of such control relations, from the level close to the motors to a level of inter-factory logistics. Each higher-order control relation “[closes the world](#)” for the lower levels; but this closure should not be made strictly hierarchical, because that limits [adaptability and composability](#) with other “worlds”.
- the concept of a [singularity](#) in the configuration of the [kinematic chain](#) of a robot relates the values of the chain's joint positions with the chain's geometrical properties
- 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.
- ...

Types of higher-order relations

- [transitive](#), [converse](#), [identity](#), [associative](#), [commutative](#), [symmetric](#), [asymmetric](#), [reflexive](#) or [equivalent](#) relations.
- [constraint](#) (Fig. 1.5) 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**². 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 (Fig. 1.6). 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.

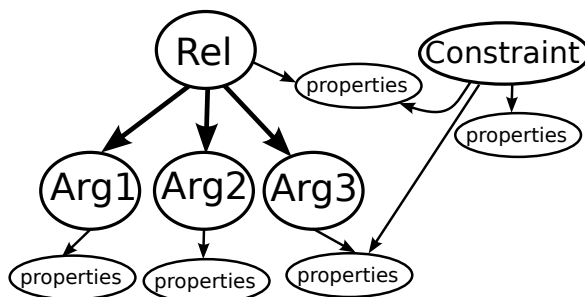


Figure 1.5: A **directed acyclic graph** representation to model a **constraint** between the properties of a relation and one of its arguments. The constraint is a (higher-order) relation in itself, with its own properties.

²The terminology for these relations is another name for what this document has called the **is-a** relation, and its inverse.

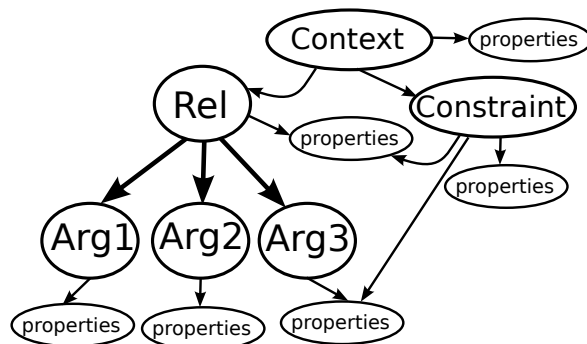


Figure 1.6: *Directed acyclic graph* representation that adds a **context** relation to the model of Fig. 1.5. Such a context can influence the meaning/interpretation of all the relations and constraints within the context, and is yet again another (higher-order) relation in itself, with its own properties.

1.2.4 Properties and attributes

In *entity-relation* models (Sec. 1.2.1), and hence also in their most flexible *property graph* representation, both entities and relations have *properties*: a set of *key-value pairs* that contain the “data” that is needed to understand the *meaning* of the entity or the relation. “*Attribute*” is another term that is often used to denote that “data”. This document chooses to introduce the following semantic difference between both terms, and the choice reflects the *etymology* of the terms:

- **properties** of an Entity are the “data” that that Entity “possesses” or “owns”, and without which the Entity has no meaning, in whatever context the Entity is used.

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

- **attributes** are the “data” that are given to an Entity by a Relation that involves it. More in particular, attributes are the properties of the *role* of an Entity in the Relation.

For example:

- the color 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 made part of.
- the position of a rigid body in space is always relative to other bodies or references.
- 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.

In this document, there is in fact little need to use the term “attribute”, because of the fundamental role played by the Entity-Relation duality: any Relation is immediately considered to be an Entity in itself, and the attributes that a Relation gives to an Entity are modelled as properties of the Relation node in the property graph model.

1.2.5 Policy: semantic_ID meta data

Higher-order relations imply the need for so-called *reification* in the modelling, which means that *relations* become *entities* themselves. So, among other application-specific properties, they have their own *unique identifier* to refer to, so that they can be used as arguments in

other relationships. In summary, the computer representations of *all* the “digital twins” and their first-order and higher-order interconnections can be realised with the same [mechanism](#) of the *property graph*; this document suggests to adopt the *policy* to give nodes (and hence also edges) the following **semantic ID abstract data type**:

- **ID**: a [unique identifier](#) with which the edge or relation can be referred to in other models.
- **MID** (“model ID”): a unique identifier that points to the model that provides the immediate context in which to interpret this entity, relation, constraint or context. One can also call this the “*type*” of the node.
- **{MMID}** (“meta model UUIDs”): a **set** of unique identifiers that each point to one of the meta models with which to interpret the model.
- **{outE}** (“outgoing edge UUIDs”): a **set** of the composition of (i) the unique identifier for an outgoing edge, together with (ii) the ID of the node to which it connects.
- **{inE}** (“incoming edge UUIDs”): a **set** of the composition of (i) the unique identifier for an incoming edge, together with (ii) the ID of the node to which it connects.

The pragmatic cost of a semantic ID is low: a unique identifier can just be an integer, whose uniqueness need only hold in the context of its meta models. In addition, every composition requires, in principle, only one extra contextual identification per composing *container*.

The motivations behind adding semantic meta data to models in a systematic way are that (i) the topological structure “model”–“meta model”–“meta meta model” *is* domain knowledge that is worth representing explicitly in itself, and (ii) sooner or later, any model will have to be connected to new information, and any system will become part of an even larger system. At that moment, the new bigger context will have to be able to refer to already existing models and “reason” about them, to any level of detail, *and* without having to change anything to the already existing representations. The semantic richness of a knowledge graph increases significantly every time such **higher-order knowledge**, or any new [abstraction](#), are added to the graph. These are just other set of nodes and edges, that 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.

There are some standards that can be used to represent the unique identifiers: *Universally Unique Identifiers*, [Universal Resource Identifiers](#), and [International Resource Identifiers](#).

1.2.6 [Mechanism](#) for reasoning: property graph traversal

A property graph is the **structural** representation of a knowledge model.³ The **behavioural** part consists of [graph matching](#) and [graph traversal](#). These are activities [to query](#) the graph for answers. Graph matching gives a template graph as input, and outputs (the set of) matching sub-graphs in the knowledge graph; the input in graph traversal is a data structure (“programme”) that encodes at which node to start the query answering, and in which order to follow edges and nodes further in the graph to find the answer. The simplest form of graph traversal is the one that works on tree structures; the popular [serialization formats](#) such as

³An early standard, [Topic maps](#), has apparently been forgotten by the community.

XML and JSON⁴ have tree traversal query language, e.g., XPath (and its superset XQuery) and GraphQL.

Obviously, graph traversal queries contain extra, **higher-order knowledge** about the system compared to graph matching queries, and are a possible approach towards realising the **holy grail** of **higher-order reasoning**, which mainstream reasoning frameworks, like the ones built around Prolog or OWL, cannot provide, because they remain at *first order*; indeed, Prolog or OWL do not have the semantics to represent relations on Prolog or OWL relations.

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 **Arg3** node in the relation **Rel**, one can follow the arrows from the **Rel** node to the **Constraint** node and its **properties** node. Note that the direction of the arrows reflect *meaning* in a relation in a model, but these arrow directions do not constrain the traversal through the model’s graph that is stored in the graph database; the latter always add two links for each directed arrow in a model, to allow to travel in any direction between the nodes that the arrow connects.

1.2.7 Policy: constraint, dependency and causality graphs

Some important instances of the property graph meta model are the **constraint graph**, the **dependency graph** (Sec. 1.11), and the **causality graph**. They all formalize constraints that exist between the properties of nodes or relations; but a dependency graph is a special case of a constraint graph, because “dependency” is a semantically richer type of “constraint”, and a causality graph is a specific type of dependency, in that it represents *cause-and-effect* dependencies.

1.2.8 Best practices: quasi non-existent

In contract to *first-order reasoning*, the literature and the state of the practice on higher-order knowledge graphs and graph traversals is scarce, e.g., [6, 27, 73], and practically useful software tools are not known to the authors. Hence, there are not enough use cases out there to identify common policies, let alone bad or good practices.

1.2.9 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 [6]. More in particular, they directly support the mereological and topological aspects of *entity-relation models*: they keep track of which entities are connected to which other ones, and of the properties of each and every node; and they support the specification and execution of queries via graph matching and/or graph traversal.

Of course, this interlinking of models via property graphs must stop somewhere; in the case of (robotics) software systems, this “grounding” takes place when a piece of concrete software is composed with the set of models that reflects the software’s behaviour, and a human expert has validated that this implementation is correctly realising what is represented in the models. (The software artefacts themselves are *not* stored in the graph database, but only

⁴An insightful and concise comparison between XML and JSON can be found [here](#).

their *meta data*.) 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 ROS-based educational robotic systems than for ESA’s 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: code configuration and generation; model validation (*“does the system specifications conform to the application’s requirements?”*), verification (*“does the system implementation conform to its specifications?”*), or 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; etc. If all models are stored as directed property graphs in a graph database, reasoning is realised by means of the graph matching/traversal query language of the graph database. Some examples of tools that support such graph traversals are [Gremlin](#), [SPARQL](#), or [Cypher](#).

1.2.10 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 recent modelling languages were born in the context of “the Web”, namely [RDF](#) and [JSON-LD](#), that have [built-in support](#) to represent *named directed graphs*, via their keywords for **context** and **unique identifiers**. 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 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”.

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 Mereo-topological levels of abstraction

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.

1.3.1 Mereology: has-a

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

$$\text{Relation}_x \text{ (Entity_1, Entity_2, Entity_3)}, \quad (1.1)$$

with a lot of background knowledge. The simplest interpretation (often called the “highest” [level of abstraction](#)), is that of its [mereology](#), which just represents the **parts** that make up the model, 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** (Fig. 1.6) 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.

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 1.5 extends the model with a *constraint* on the relation; the constraint brings in “loops” in the representation. The further extension with a *context* is shown in Fig. 1.6. 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.

1.3.2 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} \). \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.

1.3.3 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, 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.

1.4 Core relations in modelling: conforms-to and is-a

In a previous Section, the term “highest level of abstraction” was used somewhat sloppily, because in practice, there is *always* a higher level of abstraction, in any modelling effort. Not in the least because the science of mathematics is always driving formalizations and abstractions further and further. This document considers two relations as the core of *any* level of abstraction in knowledge modelling: the **is-a** relation, for abstraction on *properties*; and the **conforms-to** relation, for abstraction on *relations*.

1.4.1 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. It is important to stress that it is *not* required that *all* constructs in a model must conform to the constraints of the meta models, because this “underconstraining redundancy” (also known as the **open-world assumption**) is key to allow composition with any new model extensions and new corresponding meta models (*if* these do not contradict constraints introduced by earlier meta models!).

For example, Equation (1.1) was introduced as a *mereological* model, since 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 most difficult but also 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.)

1.4.2 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.)

1.4.3 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.

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.

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

No model of the real world, or of an engineered system to control 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 *abstraction*: any use of the model, in whatever context, will have "to make abstraction from" the non-modelled aspects. The other, complementary, way to reduce the correspondence between a model and

the real-world entities that it represents, works by reducing the **level of resolution** 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.

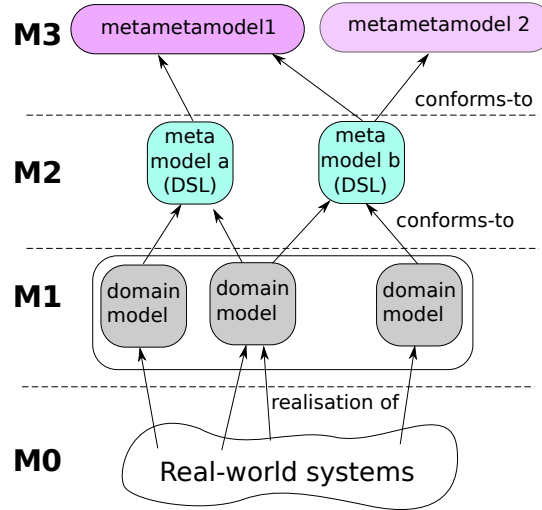


Figure 1.7: The partial order in meta models. A meta model language is often also called a **Domain-Specific Language**, or DSL. The **mnemonics** of the M0–M3 abbreviations in the modelling order relation is that the M stands for “model”, and the number represents the number of M’s.

1.5 Meta (meta) models and DSLs

One model can **conform-to** multiple meta models (Fig. 1.7); for example, the model of a kinematic chain must follow the rules of valid kinematic connections, but must also provide mathematical representations of its various parts that have compatible geometric coordinates and physical units.

Since a modelling language, or meta model, is also a model in itself, it has (possibly multiple) meta meta models, that is, the formal representations of the entities, relations and constraints that govern the semantics of the modelling language. the **conforms-to** relations introduces a partial ordering between meta models. Being a meta model, or a meta meta model, is not an absolute **property** of a modelling language, but a relative **attribute** given to it in the scope (“container”) of three modelling “levels”: it is a **topological** relation between a model, its meta models, and those meta models’ meta meta models. In practice, the “hierarchy” of meta modelling is not very deep, since one very quickly ends up with “pure mathematics” as formal languages; key examples being **linear algebra**, **graph theory**, **predicate logic**, **differential equations**, or **mechanics**.

A meta model language is often also called a **Domain-Specific Language**, or DSL (Fig. 1.7, and several domains are cooperating on making their own set of DSLs or **schemas**);

the meta meta model languages are (typically) independent of a particular application domain, for example: mathematical models, or [XML Schema](#). The *property graph* is this document’s main meta meta model for formal modelling of entities and relations. and the mereological and topological representations introduced in earlier Sections of this document are at the meta model level. Later Chapters will apply such (meta) meta modelling to the context of models for the domain of robotics: property graphs, mereology and topology are all meta meta models for (robotics) domain-specific languages for kinematic chains, motion control, geometry, dynamical systems, physical units, etc.

A very popular set of meta/meta models is that of the [Meta-Object Facility](#), which models the **instance-of** relations between **class** and **object** entities. It is a major foundation of, for example, UML, but its practitioners often try to apply it to represent all possible knowledge, information or data relations, even the ones that are not connected via the semantics of “inheritance”.

Other established domain specific language ecosystems exist in [Computer-Aided Design](#), in the form of [ISO 10303](#) (also known as “Step”) for production, and [building and construction industry data](#).

For practical use, it is not mandatory to have formally represented meta models, and certainly not meta meta models, unless one has to perform **model-to-model transformations**, for example, to convert measurement data from the [International System of Units](#) (SI) to the [imperial system](#), or to convert [Euler angles](#) into [quaternions](#). Such transformations can only be verified formally if both “ends” have formal meta models (DSLs) *and* these DSLs conform to the same meta meta models. In the XML world, model-to-model transformations can be done with the [XSLT](#) standard; this includes the possibility to to model-to-model transformations in the JSON world, by first doing a JSON-to-XML transformation, then the XML-to-XML transformation, and finally back from XML to JSON.

1.6 Mechanism and policy

One of the major pragmatic problems to compose components into systems is that components are often provided with [hard-coded configuration](#) choices (Sec. 2.6.1). 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](#) the description of a component’s “mechanism” from the description of the “policy” with which that mechanism is used:

- **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 it **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.

1.6.1 Policy: frameworks, middleware, solvers

Two major instantiations of the coupling between *mechanism* and *policy*, in the domain of software engineering, are [frameworks](#) and [middleware](#): they provide software libraries that **optimize the “usability”** of the software in particular application contexts, by making already all of the policy choices that are relevant in that application context. The advantage is that the developers only have to make choices about the behaviour of their applications. The disadvantage is that these choices are most often hidden inside the framework, and hence **compromise the “reusability”** and composability of the framework/middleware, if one or more of the policy choices are not optimal (or even feasible) within a somewhat different application context.

1.6.2 Bad practice: 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. 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 *priority* is *not* a *property* of a *thread*, but an *attribute* given by the *process* that composes the thread together with other activities.
- 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.
- *resource allocation* is not to be done *in* a *Computation*, but *for* the component that requires the resource to help realise a *task-dependent* functionality.
- *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, since 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.

1.7 Declarative and imperative

Models consist (Sec. 1.2.1) of *relations* between *entities*, with *constraints* between *properties* and *attributes* of entities and relations. Some of these relations or constraints represent **structure** (Sec. 1.3.2) between a set of other entities, relations, constraints, properties or attributes. For example:

- the *order* in which *actions* must be *executed*, sometimes called the [control flow](#).
- the *dependencies* between *concepts*, e.g., [hierarchy](#), [taxonomy](#), [priorities](#), etc.

In general, such set of structural relations can be represented by a graph in itself, a so-called [dependency graph](#) (Sec. 1.11). There are two major complementary ways to turn the information represented by such a dependency model into “actions”:

- **imperative**: the ordering structure is determined explicitly, at **design time**. So, at **run time**, the determined “**recipe**” is executed, as is.
- **declarative**: the dependency graph is available at run time, together with (i) a **solver** program that *computes* the “optimal” ordering (by solving **constraint satisfaction** or **constrained optimization** problems), and (ii) 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 (and linked, extensible, configurable, verifiable,...) at runtime, which improves **composability**, at the cost of more execution time and memory.

1.8 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 tow 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**, since 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.

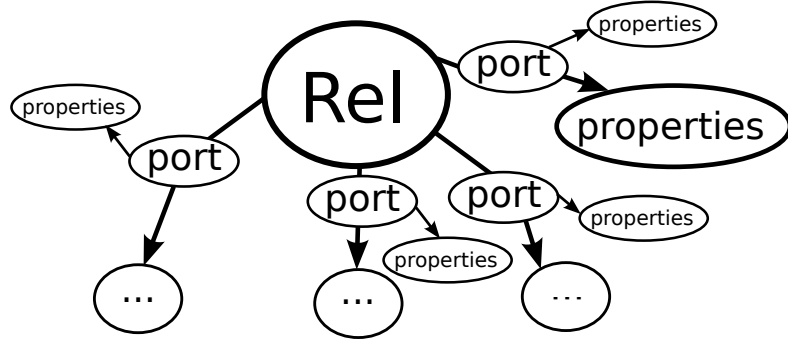


Figure 1.8: A directed graph representation of the **topological** model of the relation in Eq. (1.1). The arrows represent **connects** relations, which add extra structural information compared to the mereological model, namely the connection with a specific “port” entity that describes the *role* 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.

1.9 Structural composition: block-port-connector

This formal representation of structural composition extends the generic [Block-Port-Connector](#) (BPC) meta meta model, [75]:

- **Block:** every Relation is a Block, so **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 a concrete **instance-of** an argument with a concrete **instance-of** the Block and Ports mentioned above. Its types must, of course, match with those in the Ports.

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.

To link the *outside* and *inside*, the Ports must 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. The constraints on both ends of this Connector are just(?) type compatibilities.

1.10 Design patterns

A (software) **design pattern** is a general **reusable solution** to a commonly occurring problem within a given **context**.⁵ 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**”, which 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 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.

1.11 Dependency 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-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 → relation → constraint → dependency → causality.

⁵The concept comes from architecture (the bricks-and-mortar form of architecture, that is), via the **seminal work** of [4]. 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.*”

1.11.1 Mechanism: Directed Acyclic Graphs for partial ordering

1.11.2 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.
- ...

Chapter 2

Meta meta models for (robotic & cyber-physical) systems

All engineering domains have a **lot of knowledge in common**, using [physics](#), [mathematics](#), [computer science](#) and [systems-and-control](#) theory, to represent all possible **interactions between matter, energy, data and information**. This common knowledge is to be formalized into **meta meta models**, the “higher-order” knowledge which forms the basis for the concrete modelling of all cyber-physical systems *domains*, including *robotics*. The ambition of this document is to find the **least amount of such models** needed to represent **behaviour composition** of **activities** and their **interactions**.

Engineering systems, hence also robotic applications, contain parts from various physical domains (mechanical, electrical, thermal, etc., Fig. 2.1), and the *optimal composition* of “components” into “systems” is a major responsibility of application developers. Their ambition should be to maximize two complementary aspects:

- **composability**: this is the extent to which a **component** makes all of its “5Cs” (see Sec. 2.6.1) **separately configurable**, to increase the opportunities **to reuse** the component in any kind of system.
- **compositionality**: this property of a **system** reflects the **predictability** of the behaviour of that system, hence making the system more easy **to use** as a composition of “components”, as soon as the interconnections and the individual behaviours of the composing components are known.

Neither of both (“[non-functional](#)”) aspects can really be measured objectively and directly, or even be modelled explicitly. Hence, this Chapter introduces a collection of more concrete [best practices](#), (software) design patterns, concepts and relations, that help human developers **to bring structure** in the complex process of developing models and software for new components and new systems, hence (hopefully) leading to higher composability and compositionality.

The presented material is at the **meta meta model** level, and limited to only the **mereotopological** parts¹ of [domain-specific knowledge](#). The **behavioural levels of abstrac-**

¹That is, the composition of the [mereological](#) and [topological](#) parts.

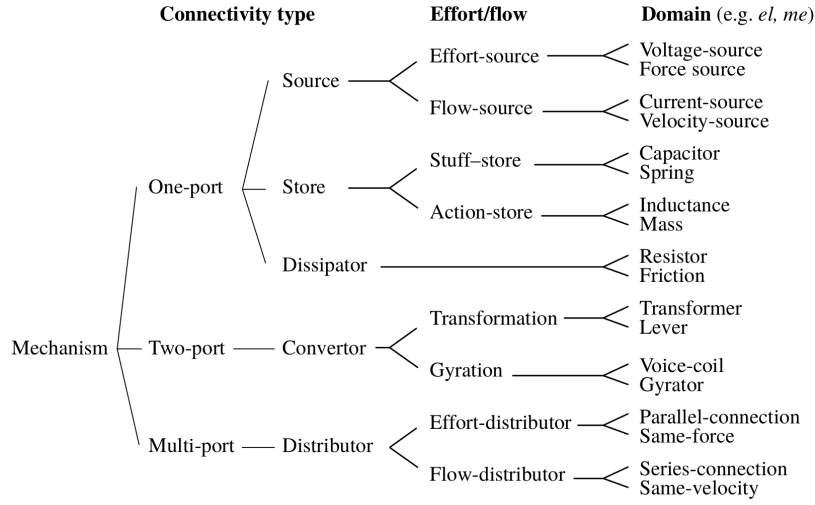


FIGURE 8. The taxonomy of physical mechanisms. The properties discriminating between the classes after branching are printed above the branch points. The classes on the right give some examples of mechanisms in the electrical and mechanical domain.

Figure 2.1: The topological relations between the various mereological top-level entities, according to [13].

tion will be added in Chapters 3 and following, where “robotics” will start to appear as a specific subset of 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:

- 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, and very complete and 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** [87] (or, “**discrete**”, or “**software**”) parts has, still, a much shorter history, and

a lot less completeness, consensus and harmony has been achieved in the domain of [software engineering](#).

2.1.1 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.
- *computations*: all the information needed to pause algorithmic computations for a while, and resume then at a later moment in time.
- *control flow*: to represent a set of conditions that are (not) all satisfied at a particular moment in time, and whose values determine which direction an algorithm is going to take at that moment.
- *plan—behaviour*: the information about what activities the system should run at a particular moment in time, in each mode in which it has to realise a particular behaviour.
- *interaction*: activities that interact often follow a protocol of interaction that has various phases.

2.1.2 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 Taxonomy of scientific theories — Levels of representation

Any engineering activity makes use of a body of [scientific disciplines](#), and human engineers spend a lifetime on increasing their understanding of the *structural relations* between (a subset of) these disciplines. Because of the sheer magnitude of scientific knowledge available, computer-controlled systems require the most structured possible formalisation of all these relations, in order to keep the scope of automatic reasoning to (somewhat) [tractable](#) levels.

For the purposes of this document, the [mathematical](#) and [information theoretic](#) domains are highly relevant. Especially the former is a very mature domain, that comes with the expected structure; for example, allowing to differentiate between the geometric, algebraic and analytical parts of mathematical models. It is beyond the (current) scope of this document

to formalize the taxonomy of scientific disciplines, but the structural relations will show up in many places, often in still too implicit ways.

The modelling in this document requires **formal representation** of the entities and relations for, both, the physical and the informational parts of the cyber-physical system. One (but not the only) **hierarchical structure**² that relates formal representations is that of the **levels of abstraction**³ described below. The hierarchy comes from the observation that each of these levels is a (not *the*) meta model of the level below, and has the level above as one of its own meta models.

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

An **abstract data type** is a **symbolic** form of a **knowledge-graphs** that also provides entities and relations to store **numerical values** of the **coordinates** of (some of) the symbolic parts. 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 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

²Hierarchy in models is a key driver for efficient reasoning on the models, because searching for “more” or “less” abstraction need only be done in one given direction through the models. So, whenever such a hierarchy can be discovered in a domain, it makes sense to make it explicit. This turns out to be a difficult exercise in practice, since many modelling standards have introduced hierarchies just for the sake of efficiency but not because they are a faithful reflection of the reality one wants to model. For example, all models that claim to be “object oriented” but violate one or more of the **SOLID** principles.

³At least, for *computer-controlled engineering* systems, since 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**.

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

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*, that is, a particular **programming language** is chosen.

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.). Examples are built-in types as *integer* values, *floating-points* values, *string* and their *composition*, e.g., arrays and unions in the **C** language. That **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*.

2.2.4 Digital storage

One essential part of computing with data structures is **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**.

2.2.5 Electronic processing

With modern computers, the **performance** of computations is strongly influenced by the electronic architecture of *cores and caches*. 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*.

2.3 Functions: composition of computations into algorithms

This Section provides the **mereological** and **topological** parts of a modelling language to describe algorithms as “composite functions” of the traditional [90] parts of *data structure*, *(pure) function* and *control flow*. The description is independent of how the algorithm is deployed in different components, of its implementation mechanisms, and of the programming language used. The major added value in this meta model lies in the *separation of concerns* of representing data, functions and control flow *explicitly* (and hence *separately*), *including* the **dependency constraints** between them. Especially the *control flow* and *constraint* parts are seldom available as **first-class citizens** of a meta model; both are essential for **solver** algorithms with a large amount of runtime adaptability.

In *all* of the “5Cs”, some form of algorithm has to be performed, and not just only in the “Computation” ones. The algorithms in “Coordination” are typically just **Boolean functions**; the “Communication” algorithms are typically **protocol stacks**; “Configurators” execute **configuration scripts** or **parse configuration “files”**. What ends up in the “Computations” parts is the rich variety of algorithms that belong to a particular application domain. Later Chapters in this document introduce several of them in a motion control context.

2.3.1 Mereological model

A mereological representation 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 entities are **data structures** (i.e., to represent the **computational state** and the **computational configuration** of an algorithm) and **functions** (i.e., instructions, or computations), and the mereological relation is **execution** (that is, a function links the data structures that are the *input* variables of a function to its *output* data structures).

The first topological relation is **(function) closure**, or “data binding”: a function is connected to the data structures that serve as its input and its output data.

The second topological relation is **control flow**, that is, the ordered progression through states (equivalent to the order of execution of functions), which is also called the **schedule**.

The third topological relation is a **(function) invariant**: it represents a constraint between (some of) the input and output data of a function that must be satisfied every time a function has been completely executed, independently of the values of the input data.

2.3.2 Mechanism: function, data, schedule, invariant and algorithm

The **entities and relations** in the meta model are direct representations of data structures, functions and control flow, with a computational schedule that composes them into an algorithm:

- 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** that represent a **function**, that is, the computational element that has D-blocks as its arguments, with some of them having the **role** of “inputs”, others of “outputs”, and some have both roles. both roles).

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

- **S-block**: the **Relation** that represents the **scheduling** constraints (or *control flow*) of a collection of **F-blocks**, that is, their appropriate execution order.
- **A-block**: an **algorithm** is a composition **Relation** (or an “architecture”) of all of the above, together with a collection of constraint relations, such as invariants and closures. An architecture is a D-block in itself, that contains the **UIDs** of the (i) composing blocks, (ii) the dependency graph relations, and (iii) its own **semantic meta data**.

F-blocks and D-blocks are *connected* to each other, *because* an F-block *changes* some data in its D-block arguments. So, there is a need for **data access constraints** to model the requirements that the order of execution of two or more functions must satisfy if one wants to guarantee correctness and consistency of the data structures they operate upon. 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. 1.7): 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 for 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.

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. Arguments (input and output values) of the

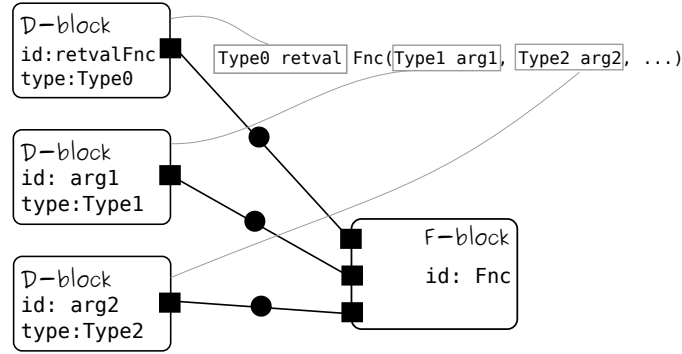


Figure 2.2: A function prototype and its graphical representation as a F-block connected to a set of D-blocks.

function are *ports* (in *Block-Port-Connector* terms) connected to D-blocks; Figure 2.2 shows an example. The structural part of the meta model [conforms-to](#) the [Block-Port-Connector](#) meta model (BPC): 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. 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.

2.3.3 Computational state of an algorithm — Stack

(TODO: data structure needed to make the mapping from input data structures to output data structures unique. When the computational state is stored, the algorithm can be restarted at any later moment by restoring that computational state. For a *pure function*, all computational state is in its arguments.)

2.3.4 Operational status of an algorithm — Flag

(TODO: a **flag** is the data structure needed to decide on the control flow inside an algorithm, that is, which of a finite number of “computational branches” to take. A flag has an **enum** (“enumerated values”) as its abstract data type, that is, the flag can take one of a finite number of possible symbolic names. The literature does not offer commonly adopted semantic differences between the terms “state” and “status”. An algorithm *can have* multiple flags at each moment in time.) Duality between flag and event; more in general, between a stream of events and a (database) table of flags: the latter is a snapshot of flags that store the value of all event at a particular point in time; the former represents the change in state of the table.

Flags represent snapshot of behavioural state in algorithms, activities (FSM), interaction (interface protocol state), etc.)

2.3.5 Policy: flags to coordinate synchronous schedules

(TODO: schedule of functions; dispatching of schedules; flags used to represent [operational status](#) of an algorithm, so they are the declarative way to influence control flow of dispatching via reacting to flags, set by other algorithms, or even other activities. Flag separates the concern of *deciding* to react, and to effectively *executing* the reaction.)

2.3.6 Policy: closure in a context

The design choices in the meta model's mechanism are very “[low level](#)” so flexible enough to allow various control flow policies: [multiple entry and exit points](#), [multiple dispatch](#), [partial application](#), [callbacks](#), [iterators](#) and [currying](#), and [closures](#). For example, the *A-block* mechanism extends the concept of [closure](#) in several ways:

- 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](#).

2.3.7 Policy: deployment in BPC component

Recall the meta model requirement that F-blocks should not have *side effects*; thus, no internal state should be allowed, or in other words, an F-block must expose any internal D-block through [ports](#). This prevents [information hiding](#), thus enabling better composability and *reusability* of the modelled algorithm, at the cost of increased complexity. If an *application* requires information hiding, it can realise this at the BPC level of its components, so leaving all composition freedom to the implementation of its algorithms. 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*. 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*.

2.3.8 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.

2.3.9 Policy: dataflow

Some applications have very few data consistency constraints, so it does not matter *when* an individual function accesses an individual data structure. **Dataflow programming** has emerged as a pattern in this context.

The design *forces* are: to maximize computational throughput; to maximize information hiding for the *control flow*; to allow some form of runtime adaptability and resource management of the *data buffers*.

The *dependency graph* models the order in which data has “to flow” through function blocks, hence it declaratively determines the control flow of the actual flow computations; that scheduling is derived at compile time or at deployment time; policies of buffer overflow management are available. In practice, only *trees* or *Directed Acyclic Graphs* give rise to predictable performance of dataflows, and, often more importantly, to intuitive programming interfaces that rely on connecting ports in function blocks.

2.3.10 Policy: functional programming

Functional programming has emerged as an algorithm composition policy that makes the functions *first-class citizens*, over data.

The design *forces* are: to maximize information hiding for the *dataflow*; to allow some form of runtime adaptability and resource management on the *callback event loop*.

The dependency graph models the order in which functions are to be *composed*, hence it declaratively determines the control flow; the scheduling (possibly via *dispatch tables*) is derived at deployment time; policy of preventing starvation of function invocations.

2.4 Activities: composition of algorithms into behaviour

For every physical and information-processing component in a robotics/cyber-physical system holds that it **has-a behaviour** that lasts for some time. For example, the motion of the air that flows around a quadrotor drone and through its propellers (physical behaviour), or the control of the lift force of the quadrotor by steering its rotor velocities (information-processing behaviour). This document uses “**activities**” as the collective name for entities that **realise** (“implement”, “execute”, ...) such behaviour over time. This document introduces the following hierarchy in **types** of information-processing:

- **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 **composition** of several **functions** and **function** schedules which share data in a fully **synchronous** way. That is, the functions are scheduled in a **serialized** way, and all data is available exclusively to any function in any schedule.
- **program**: the **composition** of several **algorithms**, some of which may be executed **concurrently**. That is, care is to be taken that one function does not write data at the same time that another one is reading that data, since this **asynchronous** execution of functions can introduce inconsistencies in the data they share amongst themselves.
- **activity**: the **composition** of a **program** with the **ownership** of a particular **resource**. *Ownership* means that:
 - every resource has one and only one owner at every moment in time. But ownership can be transferred.
 - the values of the properties of the resource’s model can only be changed by functions in the activity.
 - other activities that need access to the resource must do so via **queries** to the owning activity. It is possible that read access is granted in a way that requires no explicit query for each access, possibly via a *protocol*. Protocols exist for transfer of ownership (“Producer–Consumer”, “Broker”), and sharing of access (“borrowing”).
 - “*the*” state of a resources exists only in its owning activity, and other activities just have *copies* of that data, which can be mutually inconsistent.
 - when the owner of a resource is stopped, the resource is not accessible anymore, *or* the owner has a protocol with which it transfers ownership explicitly.

In other words, an **activity** does not only **communicate** data with other activities, but also **coordinates** the access to the resource under its responsibility.

2.4.1 Mechanism (meta model): event loop computations

The *computational architecture* that fits best to the **activity** meta model is that of the **event loop**, that is, the combination of synchronous and asynchronous *communications* and *computations*. The event loop pattern is so important in itself that it is explained in more details in Sec. 2.8.

2.4.2 Behavioural state of an activity — Mode

(TODO: the extra data structure in addition to the computational state of all algorithms inside an activity; that is, the data with which to determine which algorithms to run under what conditions. An activity *can and should be* in one and only one mode at each moment in time.)

2.4.3 Policy: sequential, concurrent and distributed execution

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 programs in threads, processes, cores, SoCs and clouds must take **composable** decisions about:

- **parallelization** of their functions over several computational cores connected by CPU busses 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 design 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.

Such parallel and distributed execution of activities is supported by **threads** and **processes**, relying on inter-activity communication mechanisms. The latter are commonly better known as **inter-process communication**, although several of its mechanisms also hold for *threads* (e.g., **shared memory** or **message queues**) or for cores, SoCs and clouds (e.g., **sockets**). These mechanisms rely on **operating system services** for (i) scheduling and managing resources for computation and communication, (ii) configuring realtime performance parameters, (iii) enabling synchronous and asynchronous access to peripheral devices, and (iv) serialisation/deserialisation and encryption of the data structures used in communication.

2.4.4 Policy: work flow

(TODO: **BPMN**, roles, lanes,...)

2.5 Interfaces: interactions between behaviour in activities

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 symbolic activities (task, performance,...) whose interactions they control. So, two complementary mechanisms must be available:

- the **exchange** of information between activities.

This is such an important aspect of system design that it merits its own “first-class citizenship” role in this document’s meta modelling scope.

- the **production and consumption** of information.

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 information interaction is realised via the so-called **interfaces** of the activities. This Section explains, at the **mereo-topological** level, how the **producer/consumer communication pattern** is composed with the activity pattern, to lead to a composite pattern, the **Stream**, that models **interactions** between activities.

2.5.1 Semantics of exchange instrument: buffer, queue, channel, socket

Interfaces exist to give one **activity** access to information in another activity. The **pointer** is the core technical mechanism (in software representations, that is) to support such access: the “owning” activity provides the “using” activity not with the desired information itself, but with another piece of information that indicates where the latter can find the desired information. This pointer can be the *address* in the RAM memory where the information’s

data structure is stored. It can also be a *symbolic name* (“UID”) of the data structure; in this case, the “using” activity must translate the symbolic name into a concrete address, e.g., via a *key-value* lookup table. There are several complementary (and not mutually exclusive) ways the “pointer” mechanism is being used to realise *semantic classes* of interfacing:

- **buffer**: both activities share the same memory space, and can use the (symbolic or address) pointer directly in their own code.

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. Most often, the constraints are only 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 buffers 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.

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.

- **channel** combines the advantages of buffer and queue: activities share a buffer (in the RAM-based shared memory of the process in which both are active, or via “*file based*” *shared memory* offered by the operating system), but at any moment in time, the *ownership* of two halves of the buffer lies unambiguously with either the producer or the consumer, and never with both.

In other words, there is a communication of ownership (“**move**”) of a buffer half from producer to consumer.^{4,5}

- **socket**: when activities do not share directly accessible memory,⁶ 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

⁴The Go programming language uses this approach natively, via its policy “*Do not communicate by sharing memory; instead, share memory by communicating.*”.

⁵For channels, a terminology is used that fits closer to the context of “communication”: “producer” becomes “transmitter”, and “consumer” becomes “receiver”.

⁶That already is the case on one single computer, where *processes* have separated memory access, realised by *hardware*.

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. A key example illustrating this context in the case of operating systems is the [D-bus](#) inter-process communication project; or the [WebRTC](#) project in the context of multi-media streaming over “the Internet”. In robotics, the [ROS](#) project tries to achieve the same role, but it has not yet reached industry-grade maturity in any of the four design goals mentioned above.

2.5.2 Semantics of continuous, discrete and symbolic information: data, events, and queries

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 the following three semantic types of “data” to cover all use cases:

- [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.
- **events** for the **discrete** world: activities must be able to influence each other’s **control flow**. This typically happens via sending “events” or “signals”, or to look at the value of “flags”. These are (small) data structures whose semantics is to represent the discrete (changes in the) state of the world. More concretely, the represent that “*something has happened*” or that “*some condition has a particular status*”.
- [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*.

Of course, from the point of view of the technical act of creation, storage or communication, *events* 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 meta data that explains how to interpret the data.

2.5.3 Semantics of exchange direction: uni-directional and bi-directional

Interfaces in the cyber context most often exist of a hierarchy of [communication levels](#), and even though an interface at one particular level may be *uni-directional*, communication at other levels can still be *bi-directional*.

For example, communication of data over a [TCP/IP](#) connection has a uni-directional semantics at the top level (as reflected in the naming of the `send()` and `receive()` operations) but at a lower level, [acknowledgement](#) (“ACK”) messages are sent in the other direction too.

2.5.4 Patterns: Publish-Subscribe, Producer-Consumer, Request-Reply

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.

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.
hence, one-way, send-and-forget	hence, two-way, backpressure and polling stream
topic centred: each interaction involves a data structure of the same type	message centred: every interaction can be a composition 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
Quality of Service by brokerage middle-ware	Quality of Service by application

A simple “hybrid” between both exists: [Request-Reply](#), which merits to be identified as a third major pattern. 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](#).

2.5.5 Mechanism: Producer-Consumer stream

The Producer-Consumer *pattern* has become dominant in “the Web”, and more in particular the composition of that interface pattern with the *bi-directional* exchange policy, and the *channel* data structure and ownership semantics. The reason for its success is the balance between

- the possibility of producer and consumer to adapt their behaviour to the actual status of their interaction,
- the asynchronous and loosely-coupled nature of such reactions,
- and the somewhat more complex design and implementation than simpler alternatives, like, say, *pub-sub*.

This document identifies this composition as the **stream** meta model (or “**producer_consumer_stream**”). Its mereo-topological and behavioural aspects are (Fig. 2.3)

- a **producer activity** acts as the **source** of data **chunks**;
- the **stream** is an (ordered) **data structure** of data chunks;
- a **consumer activity** acts as the **sink** that takes data **chunks** out of the **stream**;
- both producer and consumer can access and update all **chunks** in the stream part they own.
- the producer **transfers ownership** of one or more of its parts of the stream to the consumer, without leaving any holes in its ordered structure, and at the moment that fits best to its own behaviour.
- the stream has a **barrier data structure**, that indicates the separation between the (“upstream”) part of the stream owned by the producer, and the (“downstream”) part owned by the consumer.

The barrier is owned by the producer, and it can move it to anywhere in the upstream part of the stream.

- the mechanism can be implemented in software with high performance, not in the least because the clear ownership property allows [lock-freedom](#) and [thread-safety](#).

This mechanism is simple, and offers many configuration opportunities to realise solutions for almost all use cases of interacting activities. Its impact on two of this document’s major drivers, **composability** and **explainability**, cannot be overestimated. The Chapters on [information](#) and [software](#) architectures illustrate this claim by explaining many use cases in more detail.

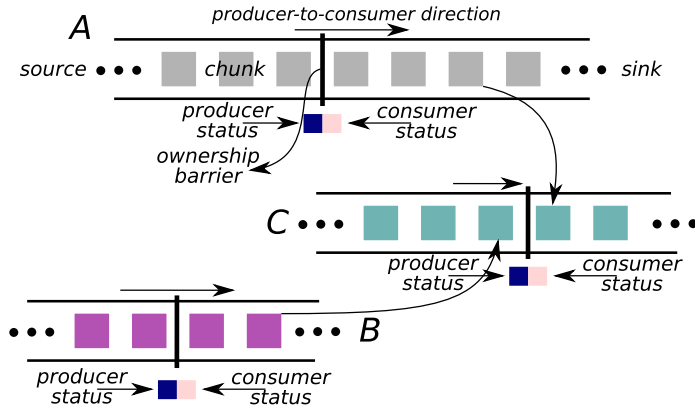


Figure 2.3: Example of *streams* and their compositions, including the status flags and ownership barrier to coordinate the interaction between producer and consumer.

2.5.6 Policy: `peer_activity` and `flow_control` status flags

The behaviour of the producer and consumer activities in a stream can be made more explicit, by composing the stream meta model with two complementary types of status flags.

The `peer_activity` status flags consist of 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), **pausing** (that is, producing/consuming activity can restart at any moment), or **requesting** (that is, the consumer is waiting for the producer to become active, or the other way around).

The `flow_control` status flags consist of again one flag for the **producer** and one flag for the **consumer**, which they use **to inform the other peer** about whether or not one peer is ready to let the other peer act on the stream. Indeed, a producer could fill up a stream faster than a consumer can process it, or the other way around. So, it makes sense to let both agree on a (stream) `flow control protocol`, to help them keep the behaviour of their shared stream predictable. This information can be conveyed by means of the `flow_control` status flags as follows:

- **push flow control**: this producer-owned status flag can either be `stopConsuming` (which represents the message to the consumer of “*please, stop consuming for a while*”), or `resumeConsuming` (which represents the message “*please, start consuming again*”).
- **pull flow control**: this consumer-owned status flag can either be `stopProducing` (which represents the message to the producer “*please, stop producing for a while*”), or `resumeProducing` (which represents the message “*please, send some more chunks*”).

The *pull* policy fits best to a *slower-consumer-than-producer* use case, while a *push* policy fits best to a *slower-producer-than-consumer* use case. The reaction mechanism is sometimes called **backpressure**, and, in general, it uses both policies intermittently, to adapt to the context.

In summary:

- **flow_control**: information produced by one activity about the **desired** activity of the **other**.

- **peer_activity**: information produced by one activity about the **actual** activity of itself.

Together, they provide an activity with a (flexible, ubiquitously deployable, and simple to understand) mechanism to adapt its behaviour to the status of its interactions with all other activities in the system.

For system architects, the added value of the combination of both types of status flags is to keep a lot of the *decision making* about **inter-activity interactions** inside the boundaries of each activity itself, hence increasing the level of **dependency inversion** and **inversion of control**, which are both advantageous to the **explainability** of system architectures.

2.5.7 Composition of Producer-Consumer streams

The **stream** is a mechanism to interconnect the behaviour inside two activities, optimizing the *loose coupling* between both activities, both behaviourally and structurally. Hence, **composition** of streams is simple, Fig. 2.3:

- one activity can be the consumer of one or more streams, and at the same time the producer of one or more other streams.
- any stream can have one or more producers, and one or more consumers.
- hence, the simplest system architecture is one in which each activity's behaviour is that of a **stream processor**: the **stream_chunks** it outputs as a producer, are processed versions and/or compositions of the **stream_chunks** it inputs as a consumer.

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

Figure 2.4 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.

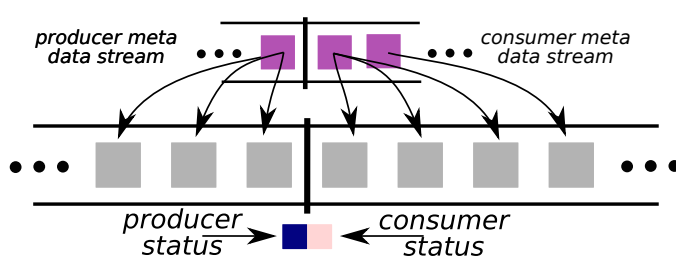


Figure 2.4: Producer-consumer stream composed with an extra meta data stream. Each chunk in the latter encodes meta data for a contiguous series of chunks in the former. Both streams share the same status flags, and ownership barriers.

[Time series](#) are a common example of requiring metadata, namely the information about (i) the time that each chunk in the stream was produced, and (ii) the clock that was used to generate the time samples. For example, indicating [time zone](#), [temporal resolution](#), etc.

2.5.9 Submission-Completion streams

An extension of [Request-Reply](#) towards *series of related* (CRUD) requests is the pattern of **Submission-Completion streams**:⁷ producers of requests add them to the submission stream, and come back later to collect the result from the completion stream, Fig. 2.5.

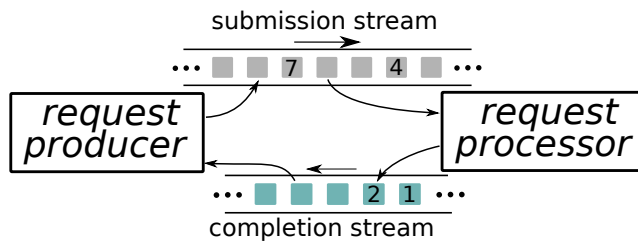


Figure 2.5: Submission-Completion streams. The processed stream goes to the original producer; the order of the items in the *completed* stream can be different from the order in the *submitted* stream.

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](#); [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](#).

2.5.10 Publish-Subscribe & Request-Reply as stream architectures

The [Publish-Subscribe](#) and Request-Reply communication patterns can be realised as particular cases of stream compositions:

⁷The simplest version of this pattern is the [Command stream](#).

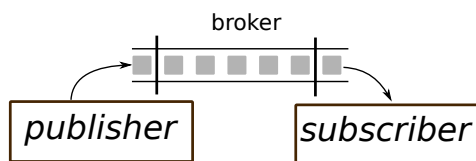


Figure 2.6: *Publish-Subscribe* communication realised by a streams composition. Both publisher and subscriber have ownership of only one single entry in the stream.

- *Publish-Subscribe* (Fig. 2.6): the publisher activity has a stream to the broker activity, without flow control or backpressure support, and with the extra policy that the publisher 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.

2.5.11 Policy: message broker

A typical *use case* of a stream (in the particular context of a [networked environment](#)) requires the producer and consumer *to register* with (or, *to subscribe to*) a stream, and to make use of the stream’s services to realise the data (and hence ownership) transfer. This policy implies that the stream must be more than just a data structure, namely an independent peer activity in itself; in other words, it becomes a *broker*.

2.5.12 Interaction state — Interface protocol phase

(TODO: the extra data structure in addition to the behavioural states of the activities involved in an interaction; that is, the data with which to determine which data to exchange via the interface, and when. Many interfaces have a dedicated protocol that formalizes the different **phases** of the interaction.)

2.5.13 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.)

2.5.14 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*.)

2.6 Components: composition patterns for activities and their interactions

Every [digital platform](#) consists of a (potentially large) number of components, providing “*services*” to each other. This Section provides a meta model of the architecture of such

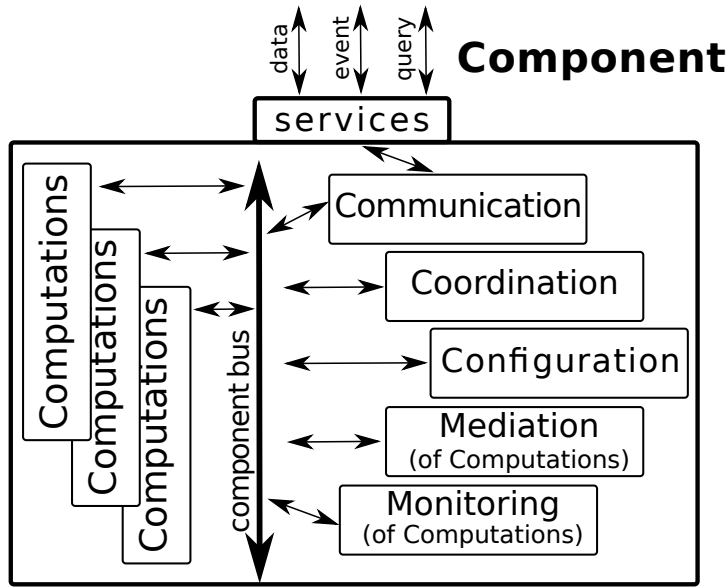


Figure 2.7: The mereo-topological model of the *Component* primitive.

components, designed for this distributed context, with **dynamically changing runtime lifecycles** of, both, the servicing components and the application being served. The design allows:

- to deploy all functionalities and patterns presented in all the Sections of this document.
- to use only a subset of its parts without changing its capability of serving in a composable application system architecture.
- to be a sub-system in itself, repeating the exact same component architecture composition internally.

2.6.1 The 5C’s: composition of behavioural roles

Since quite some time already, “robots” can not be built anymore with just one single **computer program** being responsible for their control. Not in the least because, within the set of all cyber-physical systems, “robots” are the devices from which the highest amount of **flexibility** is expected, both in its (hardware and software) resources and in its task capabilities. Hence, it makes more sense “to engineer a **digital platform**” that must control a robot system, than “to program” a robot. There is no sharp definition of what a robotic digital platform is, or should be. That lack of definition is not really a problem, because the above-mentioned expectations in variability and flexibility imply that the set of software and hardware components (“services”, “processes”, “nodes”, “agents”, or whatever one wants to call them) cannot be fixed at design time of the system anyway.

The art of system engineering hence consists of finding the “right” granularity in component behaviours, and the “right” architecture of their interaction structure, such that complicated but reliable digital control systems can be created, offering predictable capability performance even though various components can come from **different, competing vendors**. A necessary condition for the latter expectation is that all models (of structure, behaviour and interaction) are built with neutral, open and formalized DSLs (Sec. 1.5).

The simple **5C**⁸ component⁹ meta model [66, 82] that any system design should conform to, contains the following **mereological** entities and relations:

1. **Computation**: the “**processing**” entities that realise the “**continuous**” behaviour of the component, that is, to transform the (streams of) input **data** into (streams of) output data and events.
2. **Coordination**: the “**logic**” entities that realise the component’s “**discrete**” behaviour, that is, to process incoming **events** so as to decide whether or not the component has to change its Computation behaviour.
3. **Configuration**: the “**logistic**” entities that turn such behaviour-changing decisions into practice, taking into account the **constraints** imposed by the **reources** that the component is using. Indeed, most often a component can not change its behaviour from one execution time to the next, because the component relies on specific behaviours of other hardware and software components. Hence, configuration requests often trickle down to other 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 busses**.
5. **Composition**: the “architectural” relations that link a given collection of components together into one or more “super” components (or “sub-systems”, or “systems”), in such a way that all of the above-mentioned “Cs” are provided again to the “next” level of component/system composition.

A Component typically need already the following *types* of Composition, because the represented roles are generic:

- **Monitoring**: each monitoring component is a Computational component by itself, but it has 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,
 - to compute the **Quality of Service** (QoS) with which these computations are realizing their *expected* service behaviour, and
 - **to fire an event** when particular QoS thresholds are being reached.
- **Mediation**: each **mediation** is a Computational component by itself, but it has a “**higher-order**” role complementary to the *Monitors*:

⁸The 5C model of this Section has its **acronym** in common with that of [51]; while there is a thematic overlap, the meaning of the latter 5C model corresponds more to the “levels of abstraction” discussed in Sec. 1.3.1 and following.

⁹The document uses the name “component” somewhat as a *pars pro toto* or *synecdoche*, that is, to indicate the parts of a system, all of its sub-systems, as well as the whole system itself. This is in agreement with this document’s major goal to present **composable** designs.

- it exchanges data with the *Component’s Configuration*,
- because it has the extra **application-specific** knowledge about how various *Computations* are to be **traded-off** when the QoS of the *Component* goes beyond its configured thresholds, and
- then to take the **decision** to trigger the *Configuration* component to **reconfigure** some of the *Components* to react according to the application trade-offs configured in itself.

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*.

The above-described meta model is only **declarative**, in the sense that it just makes developers aware *that* each (software) component **has-a** specific set of parts with the above-mentioned roles, but it does not explain *how* to realise these roles with concrete software components. The following Sections explain, step by step, the approach to eventually develop **imperative** architectures that conform to the 5C meta model; the following Chapters then complement the architectural aspects with structures and behaviours of the concrete application domain of robotics.

The meta model is a *model*, and hence *not* a **software architecture**. That means that:

- the concrete models of systems can be **pre-processed before configuration and deployment** to optimize, for example, 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.

2.6.2 Bad practices

- *Configuration* is not to be done *in* a *Computation*, but *for* the component in which the *Computation* provides a *task-dependent* functionality.
- similarly for *Coordination*: a *Computation* must never make the decision that its behaviour has to change, even if it provides all the data needed in the *monitoring* that fires the event to (possibly) trigger the decision.
- try to *schedule* asynchronous activities, by introducing a new activity with the responsibility to *trigger* the other activities by means of sending execution triggering messages.

(TODO: declarative, so only “don’t do” relations can be given; no *Configuration* in *Computation*; no *Computations* in *Coordination*;)

2.6.3 Mechanism: “5Cs”

2.6.4 Policy: separation of concerns in/between platform and application

The meta model allows a lot of variability within the same strict and semantically grounded structure, and the “*forces*” that drive concrete designs into different directions are:

- **separation of roles** in the digital platform: this has been explained in Sec. 2.6.1.
- **separation of application and platform**: the knowledge about what is “optimal” for the application is deployed in components with an explicitly identified role, separated from those that contain the knowledge about what is “optimal” for the digital platform. Of course, such separation need not imply full information hiding: the application must be kept informed about how well the platform is realising its capabilities, and the platform should be informed about the *Quality of Service* desired by the application.

The domain of software engineering has a term for the *bad practice* of too much coupling, and the meaning of that term fits very well to modelling too: two software modules are *connascent* if a change in one module implies the other module to be modified accordingly, in order to maintain the overall correctness of the system.

2.6.5 Policy: vendor-centric added value in Configuration, Coordination, Composition

The *Component* meta model has a handful of different roles, but Fig. 2.7 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 component 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.

2.6.6 Policy: Coordination, Orchestration, Choreography

One possible trade-off that one can want to make in the design of a system is that between (i) the amount and latency of communication that is expected to coordinate the behaviour execution between several *Components*, and (ii) the *autonomy* of each *Component*:

- **Coordination**: *all* components have been designed to react to events with explicitly identified names, the events are broadcasted to *all* components, and each reconfiguration must be triggered by a broadcasted event.
- **Orchestration**: a lot less events have to be broadcasted, since all components share the same “*score*” with the expected sequencing of events; it then suffices to broadcast synchronization events only, at a much lower rate and not necessarily in a broadcast to all components.
- **Choreography**: the components have *Computation* components that observe the behaviour of other components, and *Monitors* that can recognize, with internal computations only, when reconfigurations are needed; ideally, this can happen without the need to broadcast one single event between the components.

2.7 Tasks: composition of behaviours for control, perception and world modelling

Section 2.6 presents the *platform-centric* meta model for *Components*, whose ambition is to create composable software *services* (irrespective of the application domain the services are designed for) that are to be deployed in operating system processes, on top of hardware *resources*; Sections 2.3, 2.4, 2.5 and 2.8 present how to interface *functionalities* in *event loops*, executed in *activities*, and deployed in threads and processes to provide the services accessible at the ports of software components. All of the above are *generic*, i.e., application and domain independent, and this Section presents that last missing domain/application-specific piece, namely the **mereo-topology** of how an **application** must represent entities and relations of its domain by means of the **task meta model** (Fig. 2.8). That is, to model:

- the **capabilities** that the application wants to offer to its “users”. (In the context of this Chapter, those “users” most often are other Task models, not physical/human *end users*).
- how to realise those capabilities with the **resources** it has available.
- the **strong structure** in the (small set of) **types of functionalities** the application must compose together in one single task.
- the **strong structure** in how several Task models can be **composed**, *horizontally* (i.e., Tasks at the same *level of abstraction*) as well as *vertically* (i.e, Tasks at different levels of abstraction).

The formalisation of how to compose the task model parts (the “information architecture”) with the platform-centric models referred to above (the “software architecture”) is dealt with in the *system architecture* Sections.

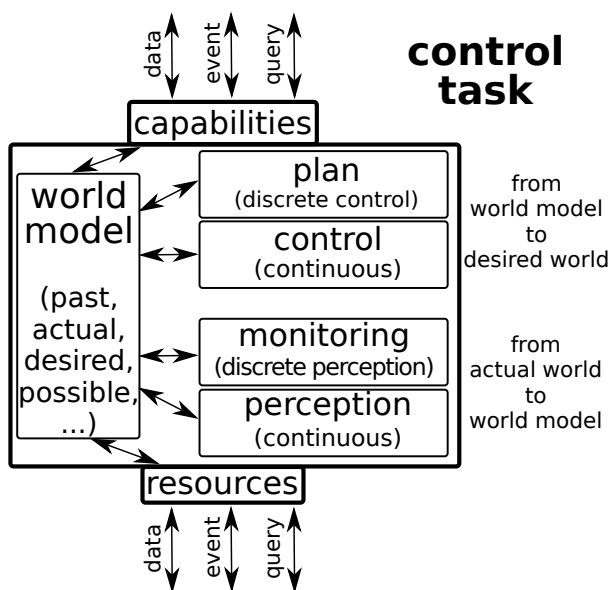


Figure 2.8: The mereo-topological model of the *Task* primitive. Note that the drawing does *not* represent a software component, but a *model of the composition* of the parts needed in a Task, and of where interactions between these parts occur. (Of course, the software components that will, eventually, implement Task executions, will be *structured* along the Task model structure as much as possible.)

2.7.1 Mechanism: capability, resource, world model, plan, control, monitor, perception

Figure 2.8 shows the mereo-topological graph of the Task meta model. The **mereological** part of that Task meta model has the following types of **entities** (and nothing more!):

- **resource**: the *constraints* of what can be provided by the resources that are necessary for the execution of the modelled 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.
- **capability**: the *constraints* of what the task execution can deliver to users, together with the *quality of service* metrics which represent how well those capabilities are being provided.
- **world-model**: all the **information** that **needs to be shared** by all the other models in a Task. In other words, it contains the extra information that represents the *context* in which these other models are used; e.g., the information about how the “magic numbers” in all these models are connected to each other for each specific set of capability requirements and resource constraints, and how these are related to information about the *physical* world.

There can be multiple “versions” of the “world” present at the same time (e.g., past, present, and future **states**), because the **world-model** is also the place “to memorize” the past, or to host possible desired or future worlds, both with various “uncertainty” versions.

The **world-model** is not just there to represent information about the external physical world, but it also contains **semantic tags** with (links to) information that is relevant for the other parts (plan, control, monitor, perception); for example, texture or color information of an object in the world that is optimal for a particular type of sensor (processing) to detect.

- **plan**: the model of which “motions” (or “actions”, in general) to be executed in which sequence, and under which conditions. The **plan** is the **discrete** version of the **control**, 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 realise the plan, in the actual world, and with the actual task requirements and resource constraints. This model contains the **continuous** aspects of **control**, that is, the one for which the actions are triggered by *data* streams.
- **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.
- **perception**: the model of how sensors can provide the *actual* information needed to create and/or update the **continuous** parameters in the world model.

The **topological** part of the Task meta model has the following **relations and constraints**:

- **separation of concerns via world model:** this constraint on the allowed connections between the mereological parts in a Task meta model, is a **major design axiom**, because it implies that all interfacing between **plan**, **control**, **monitor** and **perception** functionalities takes place only indirectly, via interactions with only the **world model**. In other words, the world model is the place to store all the **state**, that is, the information that one must remember from the past, to act in the present, and to predict the future.

“Interactions”, “to store” and “to remember” suggest communication, data base functionalities, or shared memory, but that is a too constraining view: what is described in this Section are the *models* of the entities and their relations, and not software realisations that conform to these models.

- **data, event and query** interaction models, to support the information interaction required for, respectively, inter-agent streaming, asynchronous reactive broadcasting, and discovery and configuration of inter-agent cooperative behaviour. (See Sec. 2.5 for more details.)

A similar remark as above holds here too: despite the suggestive names, these are models of what type of interaction is required, and they do not represent the software in communication middleware or the operating systems that realises these interactions..

Most applications have to support multiple **tasks**, to multiple users, and with dynamically changing requirements for both. Hence, the task primitive mechanism must allow various *composition policies*; Figure 2.9 is an illustration of one such composition over multiple of the *levels of abstraction* discussed in this document, and the next Section explains the trade-offs available in composition.

2.7.2 Task meta model as semantic database

For any somewhat realistic application, composite task models as in Fig. 2.9 will contain several hundreds to several thousands of entities, relations and constraints, so it is appropriate to call it 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, via the higher-order interconnections between them.
- queries on the database can (hence) use semantic terms reflecting the **intention**, **causality** or **dependency** that holds between query arguments. In other words, one can get explanations about *why* the query yields the results it does, or about *the context* in which the answer holds.
- as a further result of the semantic contents of the database links, knowledge can be exploited also in the computation of the query answer, because **graph traversal** becomes possible instead of the less efficient but more general *graph matching*. The latter is the default “solver” for relational database, while the former becomes more and more standard in graph databases.

In the context of cyber-physical systems, and robotics, this means that the **coupling** between **control** and **perception** can be adapted to the context provided by (i) the expected

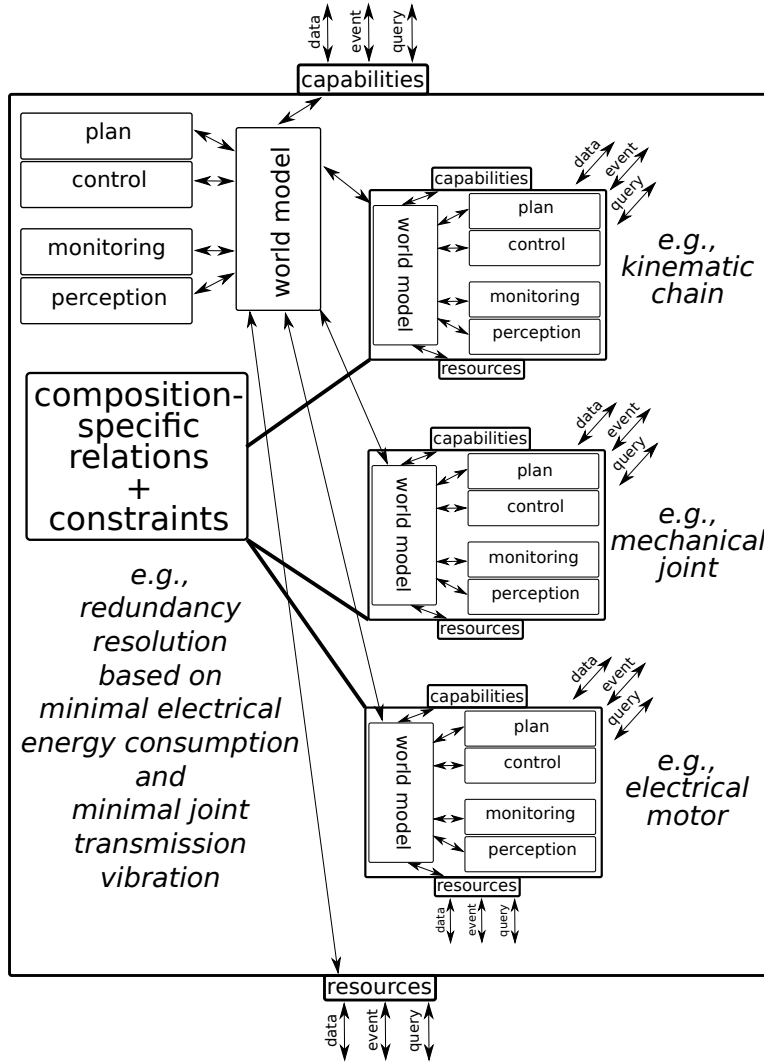


Figure 2.9: Composite task, over multiple levels of task specification, linked to three of the most common *levels of abstraction* of a robot’s kinematic chain. The *task composition pattern* is repeated at the composite level, adding the **knowledge** about which additions to make at the composite level, and which interconnections (constraints, new relations) to add with the parts that are already present in the composed task models. Again, note that the figure represents the composition of *knowledge relations*, and not that of software components.

capabilities, (ii) the available resources, and (iii) 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 all outdoor navigation tasks where traffic signs and signalling are available, such as in car and truck driving, or plane and helicopter take-off and landing.

2.7.3 Policy: coupling via shared world model

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.

2.7.4 Policy: continuous, discrete and symbolic Task models

Most robotic systems have three complementary knowledge representation (and integration) needs:

- **continuous**: the knowledge represents the time, space, effort, cost,... aspects of a Task.
- **discrete**: the knowledge represents when and why to select another continuous Task model, and how to switch “smoothly” between them.
- **symbolic**: the knowledge represents the insights about which “magic numbers” to configure into the continuous and discrete Task models, and how these magic numbers are interconnected by (higher-order) relations and constraints.

(TODO: examples in control (feedback/feedforward, FSM, strategy) and perception (association on data, information and task levels.)

2.7.5 Policy: hybrid constrained optimization problem specification (HCOP)

(TODO: how to fill in the HCOP specification with Task meta model structures)

2.7.6 Policy: declarative solvers and imperative algorithms

Computations that use constraint-based **solvers** are **designed** to be more deterministically composable than ones with only imperative algorithms.

(TODO: a solver generates a *plan* at runtime, with a more limited horizon (in time, space, resources, capabilities) than the *Task* in whose context it works. The different policies in this respect model which solver and which horizon to choose, under which contextual situation.)

2.7.7 Policy: sharing data, events, models

Communication infrastructure is **designed** to be shared by many processes and/or computers, so a lot can be gained by deploying several task-level “client-to-server” communication channels onto the same infrastructure channel, and to do the same with the data, event and/or query messages that are to be exchanged through these channels. “Queries” are, in fact, complete *models* that are being exchanged between components, allowing for higher levels of declarative interactions. For example, the success of the “Web” is based on the fact that full HTML models are communicated, which 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.

2.7.8 Policy: mission, service, skill, function

(TODO: explain how these different names in the literature conform to the same meta model, but within different scope and levels of abstraction.)

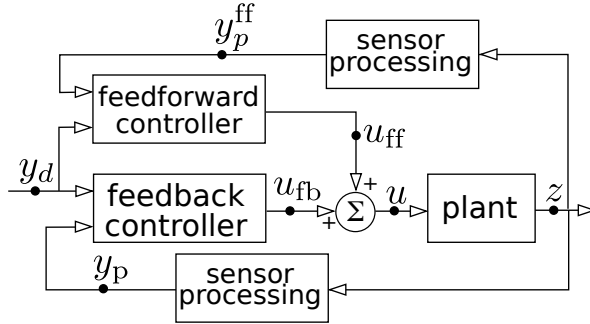


Figure 2.10: Feedback and feedforward control both contribute to the actuation signal u that is given to the plant. And they both can work with *processed* versions (y_p and y_p^{ff}) of the current state z of the plant.

2.7.9 Vertical and horizontal composition

The [Task meta model](#) is a major part of the modelling methodology of this document. This Section explains how its design makes it a candidate for composition, both “horizontally” and “vertically”.

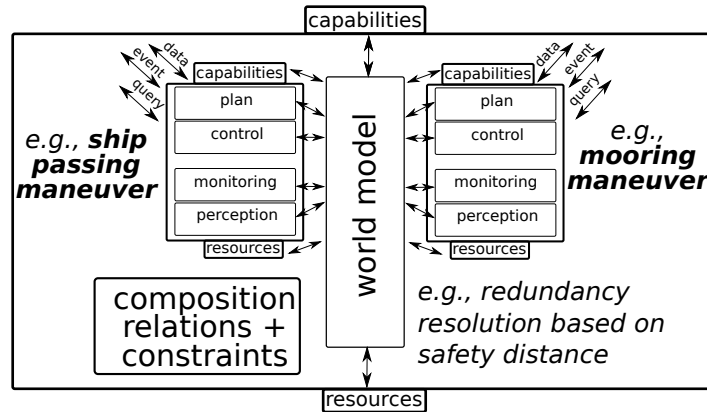


Figure 2.11: Horizontal composition of Task models, in an inland waterways shipping context.

Horizontal composition — Task activities **share** *at least* a part of the world model, but in most cases also parts of perception, control and monitoring, Fig. 2.11. 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 HCOP-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. 2.6.1) in the system’s architecture.

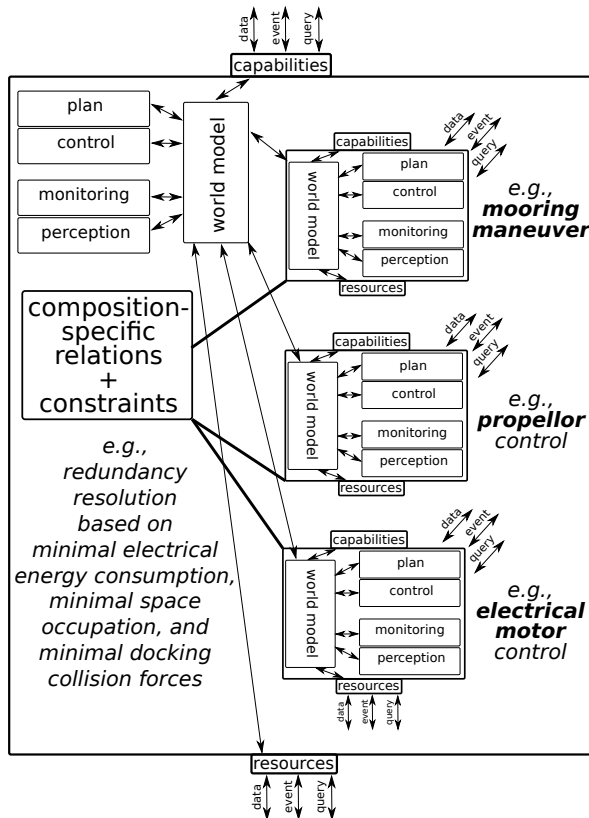


Figure 2.12: Vertical composition of Task models.

Vertical composition — Tasks **add** to each other’s models, Fig. 2.12:

- *higher level* adds constraints and objective functions to *lower level*’s COP.
- 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*.

But again, the same HCOP-based control methodology applies as for the composed Tasks individually, and not only the *Coordination* and *Configuration* parts are adapted, but also new *Communications* and *Computations* are to be introduced.

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.
- **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.

2.8 Event loop pattern: composition of asynchronous computational behaviour

The event loop is the mechanism to compose the execution of (i) **synchronous** algorithms (e.g., sensori-motor controllers), together with (ii) a set of other **asynchronous** activities (threads and processes, physical or digital) with which the algorithms have to interact (e.g., via digital or analog I/O, or via networking). Some instantiations of the pattern are (i) the “*Web*” with “browsers”, “servers” and [Single Page Applications](#) (SPA), (ii) the [Programmable Logic Controller](#) (PLC) workhorse of the automation industry, (iii) the realisation of the [software components](#) in [service-oriented architectures](#), and (iv) [computer games](#).

2.8.1 The role of the event loop

The nominal context in which **algorithms** are designed is that of so-called **synchronous** execution: (i) the order in which every function executes—and, hence, the order in which every access to data takes place—is the order in which it is programmed in the code of the algorithm, and (ii) the functions have [no side-effects](#), that is, they only change the values of the data structures that are in the directly visible [scope](#) of the programme code. The execution context of **components**, however, is typically **asynchronous**: just by looking at the code, one can not know when new data will arrive at the Ports of a component, or when new data provided by a component will reach the Ports of other components; in other words, it is best to assume that each of the functions in the asynchronous parts of a software system can have side effects.

Since algorithms are to be deployed inside of components, there must be a way to connect both worlds together, and the **event loop** is an **architectural pattern** to realise this. (It is a specialisation of the [reactor pattern](#) and [proactor pattern](#), in that it adds particular ordering policies to the execution of all functions that it manages.) The [event loop pattern](#) has three main *goals*: (i) **to decouple** the synchronous parts from the [asynchronous](#) ones, (ii) to provide a **computational context** to store “state” in a [thread-safe](#) way (that is, to guarantee that it is changed by any side-effect of any of the other functions), and (iii) to allow the **application** developer (and not the operating system) **to configure** the balance between the following system design *forces* of **non-blocking asynchronous execution** and **sequential synchronous execution**:

- **data consistency in algorithms**. Many *Computations* rely on guarantees that the data they work with is changed only in deterministic ways, under their own full control.

For example, there is a lot of *state* in *task planning*, *perception*, or *discrete* and *continuous control* algorithms. Hence, access of the computational functions to the data must take place in a *synchronous* way: the order in which the function operators are programmed is also the order of the changes to the data they manipulate, *and* there is no other function changing the data “behind their back”.

- **localising the synchronization of the *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” 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.
- **identity of ownership of data.** While it is inevitable that “state” variables are copied, for various reasons and to various asynchronous activities, a system design can only be (understood or proven to be) correct if each piece of data has one and only one, uniquely identified, owner. That owner must have the possibility to decide when copies of data are allowed to enter into, or to exit from, asynchronous interaction channels, and when and how to update the “state” it owns on the basis of asynchronously incoming requests to change that “state”.
- **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 of asynchronous 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*.

2.8.2 “5C”-based programme template for an event loop

A **mereo-topological**¹⁰ version of the event loop, using only component-centric 5C entities, is given in the following pseudo code:

```
when triggered // by operating system, which deals with all
                // asynchronous side effects.
do {            // the control flow structure of the event loop.
  communicate() // get all "messages" with events & data, filled in
```

¹⁰This model represents *behaviour*, as a mereo-topological composition of 5C entities, which represent *types* of behaviour themselves.

```

        // by other asynchronous activities.
coordinate() // handle the events in these messages, and
            // decide which ones to react to.
configure()  // some events imply reconfiguration of computations.

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.
communicate() // the computations above can generate events & data that
            // other asynchronous activities must know about.

sleep()      // the loop deactivates itself, until the shortest deadline
            // that was requested in all of the steps above.
}

```

The sequence of functions in the code above is not a hard constraint, and the exact selection and serialization order of these functions in an event loop is to be configured by the application developer. Each application context indeed comes with a set of dependencies between the order in which computations can be executed, and those *declarative* constraints are to be “solved” (in principle, every time the event loop is triggered) to create a *procedural schedule* (Sec. 2.3) that encodes the *bookkeeping structure* of the algorithmic **control flow** in the event loop.

The power of the event loop pattern, as described in the code above, is that it *stimulates* developers to separate the 5C concerns in an extremely simple way. That way has also proven¹¹ 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 not of *preemptive multi-tasking*. One necessary, but not sufficient, condition to satisfy this constraint is that all functions in the algorithms have no side effects, and access only data that is local to the event loop. Often, that data is stored in the *buffers*¹² needed for the *asynchronous I/O* in the `communicate()` parts of the event loop.

2.8.3 Mechanism: callback, source, event, context, poll, dispatch, pool

Section 2.8.2 gave a mereo-topological representation about how the event loop pattern orders the *behaviour* it must realise for all the activities that it serves. This Section introduces another **mereo-topological** model for event loops, this time explaining the entities and relations —both, *computational resource*-centric and *algorithm*-centric— that together form the mechanism each event loop is built from:

- **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.

¹¹For example, many web servers and web browsers work with an event loop architecture.

¹²For example, for UDP message passing, or TCP/IP byte streams.

- **source**: the composition of a **callback** with the meta data 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 by the operating system (and hardware) and read by the event loop to find out whether or not, and what, data is available from asynchronous sources. Also the synchronously executing algorithms can fire events, but their handling can be realised completely without support from the operating system.

Events are the mechanism via which to *separate the following concerns*: *monitoring* the behaviour of a system for specific “things to happen”, and *reacting* to what has happened. The event mechanism allows to include several policies, for both separated activities: multiple monitoring activities can generate the same event; multiple activities can react to the same event; activities can react in different ways to the same events; and the connections between sources and sinks of events can be mediated and configured, even at runtime.

- **context**: the composition of **sources** that adds two relations to the meta model: (i) the sources over which to do the mediation are clearly identified in relation that the event loop execution must take into account, and (ii) all the data and constraints needed to execute all the functions in the sources must be accessible in a local data structure owned by the event loop, so that the callback handling can take place in a thread-safe way.
- **poll**: the function that a **context** is executing 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 four functions are (possibly) executed in each iteration through an event loop, in that order,¹³ 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 the **poll** and continues with the rest of the functions in the loop.

¹³Functions from several sources can be taken together, respecting the same overall order.

- **message-queue**: when some **polling** has been off-loaded to a dedicated worker thread, the **loo** need not read the original **source**’s registers (that part of the job is being done by the worker), but it “polls” the message queue that is filled by the worker with the data it polled. In contract to asynchronous I/O, the message queue is a local data structure, which is in principle always ready to read from (possibly with empty data as a result).
- **attach-source**: connect a particular **source** to one single **context**.
- **attach-context**: connect a particular **context** to one single **thread**.

2.8.4 Policy: memory allocation of source, context, queues, workers

Since the number of asynchronous **sources** is often not known in advance, and can vary at runtime as well as their readiness for polling, different implementations make different choices of which of the above-mentioned data structures to store in data segments, on the stack or on the heap. The choices are typically motivated on the basis of trade-offs between memory usage and execution time.

2.8.5 Policy: priorities

A second obvious policy to think of is that of attributing priorities to various **sources**, and to the events that their asynchronous I/O devices provide.

2.8.6 Policy: mediation

The policies above must be mediated, and (possibly each iteration of the loop), one can choose different trade-offs between *capability needs* on the one hand (performance, consistency, flexibility,...), and *resource usage* on the other hand (quality of service, “energy” consumption, latency,...):

- one can select to skip **prepare**, **check**, and/or **finalize**;
- the more dynamic one wants to make these choices, the more latency can be expected;
- one can change **attach-source** and **attach-context** at runtime.

2.8.7 Policy: context deployment in threads

The worker threads are a mechanism that still requires decisions to be made about when to use workers, how to distribute **sources** over them, and how to configure their in-process message queues.

2.8.8 Policy: integration into software components

Figure 2.13 shows an *architecture* to implement the **internals** of a **software component**, as one single *process* with three types of *threads*:

- **main event loop**: this is where the behaviour of the component is being computed. It offers a “thread-safe” context to the algorithms that realise the behaviour.

- **worker thread:** there is one worker thread for each asynchronous I/O channel that can block for a “long” time.
- **mediator thread:** this is where the (possibly many) *quality of service* measures are monitored, and in case that the performance (in, both, I/O communication and internal computations) goes beyond the QoS boundaries, one or more of the above threads is triggered to change its behaviour, via a message on an event queue.

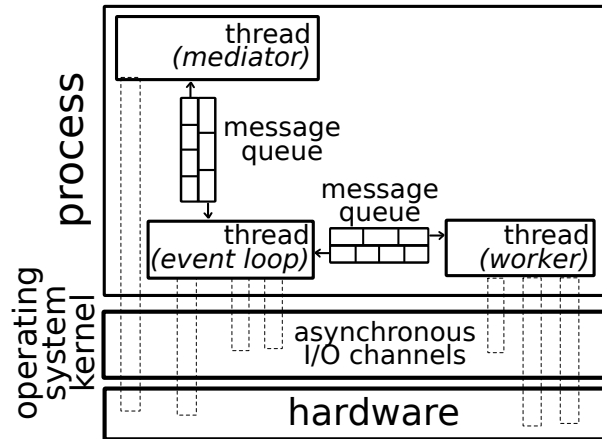


Figure 2.13: The mereological and topological meta model of a *software component*, built as one single process with several threads and message queues. The dashed rectangles represent asynchronous I/O channels, offered by the operating system and possibly requiring also hardware components such as network cards, or Analog/Digital convertors.

Since all threads reside in the same process, their mutual communication can be realised by *message queues* on the memory shared by all threads. The “event polling” for such message queues is simple and efficient, because it can be done synchronously. The Figure shows the generic case with two message queues between each two threads, to support interaction in both directions.

There is *maximum one* mediator thread, *minimum one* event loop thread¹⁴, and *zero or more* worker threads. Within the process, the threads interact via message queues; they (must) use asynchronous I/O, provided by the operating system only (e.g., Unix pipes), or the OS together with the hardware (e.g., EtherCat sockets).

Figure 2.13 does not show the **external** interfaces of the **software component**. This typically correspond to one or more **Ports** being connected to a subset of the asynchronous I/O and message queues of the component’s internals.

2.8.9 Software pattern realisations

The event loop can rightfully be called a “software pattern”, since 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.

¹⁴Hence, none of them is really the “main” event loop.

- 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**.

2.8.10 Software anti-pattern realisations

Many **middleware frameworks** in robotics/cyber-physical systems do not offer *event loops* as design and development primitives, but force their users into a hidden choice. For example, all robotic middlewares suffer from this [anti-pattern](#): ROS, Orocos, MOOS, etc.; some of their major too hard-coded policies are: dedicating one whole I/O channel to each *Port* instead of multiplexing several of them; no internal multi-threading supported as first-class design primitive; lock-in into one single programming language; and the lack of explicit provision for mediation and ownership.

2.9 Solver pattern: from declarative specification to imperative algorithm

Computations must be realised, eventually, via [algorithms](#), but sometimes the human developers do not have to program an algorithm explicitly since a software tool can be used **to generate** it from a [declarative](#) specification; that is, by specifying the *constraints* that have to be satisfied by the functions and data, but not the selection and ordering of them. This Section presents the **mereo-topological** model of such tools, which are often called [solvers](#). The generic advantage of a declarative approach is its **composability**: their key mechanisms, “constraints” and “objective functions”, lend themselves naturally to composition, while the [imperative control flow](#) of algorithms does not.

Typically, the *implementation* of software algorithms is distributed as a library with **compile-time** type checking, and its documentation is often reduced to an *Application Programming Interface* (API) description. This approach has served the robotics community for some decades, but the growing complexity of the robot systems and the higher demands with respect to application flexibility have made clear that the approach does not support developers (i) **to compose, at development time** various algorithms from the various levels of

the motion, perception and world modelling stacks, (ii) **to configure runtime interactions** with other functionalities, or (iii) **to deploy** the same algorithm with its optimal configuration settings for the large variety of software component frameworks and operating system process capabilities. All of these advanced functionalities require some form of formal models, and tooling to support the reasoning behind the required **model-to-“X” transformations**.

Moreover, during the realisation of a concrete software library, the function developer is often forced to take design choices and assumptions which later on prevent composability and reusability of the library in a different application than the one in the developers’ original focus. A typical example of these choices is the *information hiding* which often is the non-intended side effect of *object encapsulation* in object-oriented programming: in some applications it is desired to hide or protect certain data, but in the primary robotics use case of *integrated* planning, perception, control and world modelling, this has become a major show-stopper towards predictable and composable software systems. For example, for *kinematic chain* solvers, one of the most composition-limiting factors in existing API-based libraries is the fact that users of the libraries do not have access to how the “solver sweeps” over the kinematic chain have been implemented, and that they can not add attachment points to the chains for other purposes such as collision detection or visual servoing; hence, the same sweeps computations must be redone several times, within subsequent method calls, which leads to loss of efficiency, and, more importantly, to the risk of losing (computational) “state” consistency.

2.9.1 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 correspond to the scheduling of a particular **action**, and a termination condition checks whether the constructed set of actions (that is, the resulting imperative algorithm) is “good enough” to stop the solver.

2.9.2 Policy: task-based hybrid constrained optimization

The assumption behind many *task-based* approaches in robotics is that every robotic task can be modelled as a **hybrid constrained optimization problem**. Its general formalization is as follows:

task state in the task domain	$X \in \mathcal{D}$
desired state set	$\{X_d\}$
robot state in the resource domain	$q \in \mathcal{Q}$
objective function	$\min_q f(X)$
equality constraints	$g(X) = 0$
inequality constraints	$h(X) \leq 0$
tolerances	$d(X, X_d) \leq A$
solver	algorithm computes q
monitors (Boolean functions of X)	decide on switching

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:

- *parameter values* for f , X ,...
- concrete solver and monitor *models* and *implementations*.

2.9.3 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. 2.9), and hence they come close to exploiting all features of the presented algorithmic meta model.

2.9.4 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;)

2.9.5 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 strictly satisfied.
- **optimal**: the *deviation* from the constraint relation is optimized according to a *cost function*.

Satisfying versus **optimizing** solvers: the former uses the iterations towards the most optimal solution until the current intermediate solution is already “good enough”, that is, within a particular, identified, set of the constraints. The latter form of solver only returns a solution when this solution is the optimal one, or when the solution can not be computed for some reason.

2.9.6 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.

2.9.7 Policy: tolerances

- numerical accuracy of the constraint satisfaction;
- number of iterations to find solution.

2.10 Finite state machine to coordinate activity modes

This Section presents the meta model of the [Finite State Machine](#) (FSM), used to represent that activities (be it processes, systems, controllers, tasks,...) must often **realise different behaviours, one at a time**. Each of the possible [behaviourial states](#) is called a **mode**, and each mode represents one particular *set of constraints* that determines when and why an activity switches between behaviours. The description of the meta model maximizes the *separation of mechanism and policy*, and of *structure and behaviour*. This is not an obvious ambition, because in the long history of state machines various versions of their structural and behavioural models have been “standardized”, each featuring monolithic and domain-centric couplings between all these aspects. For example, [Harel statecharts](#), [Mealy machines](#) or [Moore machines](#), [19, 85]. The FSM meta model presented in this Section provides a modularization with which all these variants can be *composed*, by means of context-specific configuration relation.

2.10.1 Structure & behaviour: state, transition, event reaction table

The **structural** part of the FSM meta model, Fig. 2.14, represents each individual behaviour by a **state**, with **transitions** between states to represent switches in the behaviour. There are no constraints on how many transitions can exist between two states, and a state is allowed to have a transition to itself. The **state** and **transition** models are extremely simple; the **transition** model depends on the **state** model, but not vice versa.

The semantics of an **event** is that of a [Boolean variable](#), that represents whether or not “*something has happened*”. The role of an FSM is **to react** to events, and its **behavioural** model represents (i) the logical conditions under which the FSM **transforms** (ii) a set of **events** into (iii) a **transition** to another state, and (iv) the *creation* of another set of

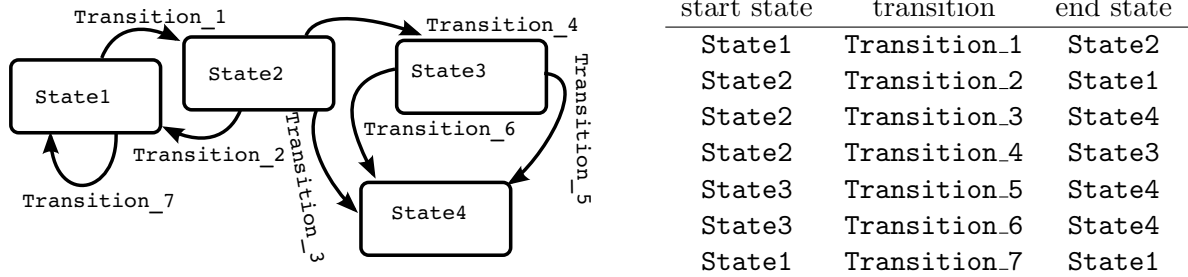


Figure 2.14: An example of the **structural** (mereo-topological) part of a *Finite State Machine* model, with its graphical representation on the left, and its tabular form on the right. In a property graph representation of this model, the nodes are **States** and the relations are **Transitions**; the FSM in itself is a higher-order relation: it composes a particular set of **States** and **Transitions**.

events. The model of this transformation behaviour is a logical mapping, the so-called **event reaction table**.¹⁵ (Figure 2.15 has an example.) Hence, an **event reaction table** is a table of **Boolean expressions**, whose truth value is computed via **propositional logic**. The processed events can be generated by the “outside” world, or by itself; the generated events can be reacted to, also by the “outside” world or by itself.

Boolean expression of events reacted to	resulting transition	events fired
$e_1 \vee e_3$	Transition_1	
e_2	Transition_2	E_2
$e_3 \wedge e_1$	Transition_3	E_1, E_3
e_4	Transition_4	E_4
$e_1 \wedge e_3$	Transition_5	

Figure 2.15: An example of an *event reaction table* that models one possible *behaviour* of the FSM in Fig. 2.14. When the Boolean condition evaluates to **true**, the corresponding **transition** is executed in the model, *and* a possibly empty set of **events** is fired. The events are not part of the structural model in Fig. 2.14, because they are the coupling with the *system* context in which the FSM is embedded.

2.10.2 Higher-order FSM model: pre, per and post conditions

An event reaction table represents the *first-order* behaviour of state transition. But FSM designers typically have knowledge about the system that can be represented as *higher-order* relations on state transitions. These relations are *constraints* on an FSM state that model the **intended reasons** to make a transition. Three types exist

- *pre-conditions*: the constraints between system variables that have to be satisfied *before* the FSM state is *entered*.

¹⁵This term *event reaction table* is not standardized outside of the scope of this document.

- *per-conditions*: the constraints between system variables that have to be satisfied *while* the system is *in* the FSM state.
- *post-conditions*: the constraints between system variables that must be satisfied *after* the FSM state is *left*.

The set of constraints in the three types can overlap. They are not events, because what matters is their truth value at the time of evaluation, not the timing of when these truth values change over time. When they are available to the system control software during runtime, they serve as **assertions** to be monitored, so one can (i) *check* whether a computed transition is justified, and (ii) *explain why* it is, or it is not.

The transition rules of an FSM are “hard coded” in order to make reaction fast. But “to explain why” a transition must take place requires higher-order modelling. More in particular, the models of pre, per and post conditions that have to be satisfied for each state, plus the relations between which of these conditions depend on each other between states. It is not necessary that a transition can only take place if the post-conditions of the first state are the pre-conditions of the next state; for example: the first state activity stopped because a time-out expired.

2.10.3 Mechanism of event handling: event queue, event processing, event monitoring, event loop

In the context of cyber-physical systems, finite state machines are used for the **coordination of activities**, and therefore the structural and behavioural models of the previous Section must be **executed**. Indeed, the models in the previous Sections are **passive abstract data types** that represent the *state of the coordination* of several *coordinated activities* by the *coordinating activity*. All activities interact by means of shared **events**, which they *fire* and/or *react to*. The meta model of that **(re)active** execution is a special case of **stream handling**, with the following extra **event handling** entities and relations, in addition to the above-mentioned **state**, **transition** and **event**:

- **event_queue**: the *structural relation* that represents the **streams** of **events** that the FSM has received and must still process, or that the FSM has fired and must still send out.
- **event_processing**: the *behavioural 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.
- **event_monitoring**: a particular instance of **event_processing** that monitors the FSM as an activity in its own right. 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 must 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**¹⁶ 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**:
 - **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, since 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.

2.10.4 Mechanism: activity Coordination behaviour via monitors and callbacks

Finite State Machines are the mechanism behind the *coordination* of several *activities*, and the latter must provide the following two types of *computations* to make use of that mechanism:

- **monitor**: a function that **observes** the behaviour of an activity, and *fires* an **event** when that behaviour reaches pre-configured boundaries. In other words, their execution is the *cause* of the **event_processing**.

A typical usage is to support the decision making about when it *may* be time to let an activity switch its behaviour. The decision logic itself is encoded in the **event reaction table** of the coordinating FSM.

- **callback**: a function that the **event_loop** executes in lieu of the *coordinated* activities that have **registered** these functions as clients for the **event_processing**. In other words, their execution is a *result* of the **event_processing**.

A typical usage is to realise reconfiguration of an activity, required by a state change in the FSM.

¹⁶Note 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”.

2.10.5 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`.

2.10.6 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.

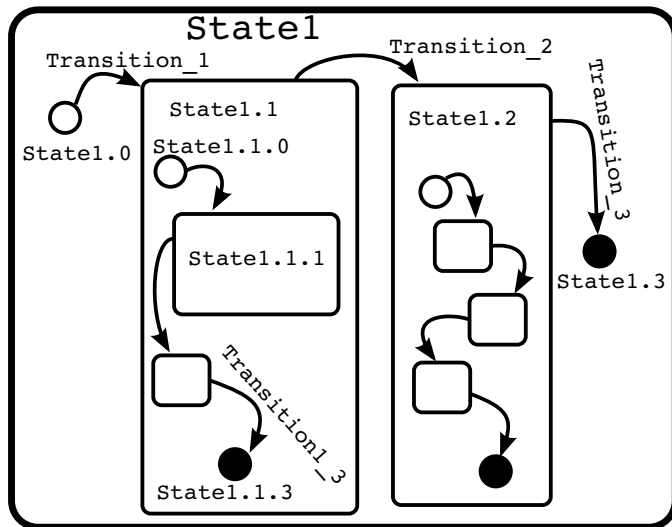


Figure 2.16: An example of a hierarchical *Finite State Machine*. The open and filled circles represent `initial_state` and `final_state`, respectively.

2.10.7 Policy: hierarchical state machine

The structural relation of **hierarchy**, as depicted in Fig. 2.16, is commonly used for finite state machines. It changes the structural and behavioural semantics of the “flat” state machine as follows:

- **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**.
- **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. 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.

The major design motivations to choose for hierarchical state machines are the following:

- **reduction of 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.

- **just-in-time creation, or lazy evaluation, of state machines from 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 Best Practice: Life Cycle State Machine for single activity deployment

Almost no component or system can provide its “services” to other components, or to users of the system, without making use itself of the services of multiple “resources”. These “resources” can be services of other components or sub-systems, but also physical and/or infrastructural resources such as energy, CPUs, memory, communication bandwidth, etc., and “computational” resources such as **interfaces** or **algorithms**. Hence, in practice it is exceptional that the services of a component can be provided, or reconfigured, instantaneously.

The design pattern of the *Life Cycle State Machine* (LCSM) in Fig. 2.17 has been used since decades, in several equivalent incarnations and names,¹⁷ to coordinate the availability of “internal” resources for the provision of “external” capabilities, and to let the capabilities be provided only *after* they have been fully configured. The semantics of the different states is as follows:

- **deploying:** the system is working on finding and configuring all the resources it needs before it can offer its capabilities to others. During this super state, the system should not be visible to other systems, since the latter can not yet (or not anymore) interact usefully with this system.
 - **creating:** the software resources are created, with which the system does the bookkeeping of its own behaviour.
 - **deleting:** the above-mentioned software resources are cleanly removed.
 - **configuring resources:** the system is configuring the service resources it requires for its own operation.
- **active:** the system is visible to other systems to engage in interactions.
 - **configuring capabilities:** the system is not providing its services, since it must first (re)configure itself to provide a particular configuration of its services.
 - **ready:** the system is ready to provide its services to other systems.
 - * **pausing:** the provision of the system’s services is put on hold, but can resume immediately. This state is necessary, for example, for the composition with **Traffic Light** activity coordination.
 - * **running:** the system’s services are provided.

¹⁷Some life cycle state machines use names for their states that represent the *reason why* a state has been reached, and not *what activity* the system is performing while being in a state. For example, names appear like “configured”, “error”, “started”, etc. These name choices restrict composability by introducing implicit assumptions of “hierarchy” between transitions.

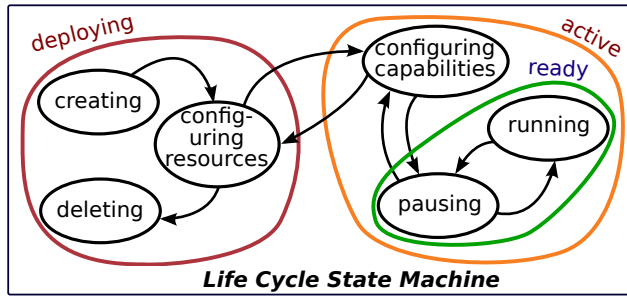


Figure 2.17: The Life Cycle State Machine meta model coordinates the configuration of the resources required for a certain activity before that activity’s capabilities can be offered as a service to third parties.

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 yet active.

Of course, the LCSM meta model **conforms-to** the meta model of *finite state machines*, which itself **conforms-to** that of *automata theory*. The LCSM meta model adds *mereological* semantics (the names of the configuration states in a LCSM), and *topological* semantics (which transitions make sense, and which are the hierarchically contained levels of abstraction that the LCSM is coordinating).

2.10.9 Best Practice: FSM for behavioural state, Flag for operational status

(TODO: a **flag** is an “memorized event”, or “level-triggered signal”: it represents the operational status of an activity, while an FSM state represents what behaviour that activity is executing; that is, what composition of control, perception, world modelling, decision making,... is activated at a particular moment in the life time of an activity. Another way of looking at flags is as shortcut for a logical condition of a composition of events: if that composition occurs “often enough” it makes senses to remember its outcome in a flag variable instead of recomputing it every time it is needed. An extra advantage is that a flag variable lends itself very well to be represented by an atomic variable, which improves multi-thread safety.)

2.10.10 Best Practice: Flag, Traffic Light, Petri Net, for inter-activity coordination

Any application of some complexity must have several activities whose only purpose is to coordinate *some actions* of *multiple activities*, because these activities can only add value to the application when their actions can be realised “at the same time”, in a particular order, or with serialised access to one and the same shared resource.

The **simplest version** is a **flag**, to indicate that a shared resource can be used by **activity1** but not by **activity2**. A flag is often the *memory* (or **level-triggered** event) of a series of events, each signaling that the **flag** can be set or unset, and the *latest event wins*.

Shared resources with more than two activities, can apply the model of **traffic lights** on a traffic junction, in which the coordinating activity generates one **flag** (“traffic light”) for each of the activities. That **flag** has a color code with the following semantics:

- **red**: the activity *can not* access the shared resource.
- **green**: the activity *can* access the shared resource.
- **orange** (optionally): the shared resource is “soon” going to become (un)available for access.

The coordinating activity fires events, to indicate a change in one of the traffic lights. And each of the coordinated activities fires similar events, that are interpreted by the coordinating activity as follows:

- **red**: the coordinated activity *is not ready* to access the shared resource.
- **green**: the coordinating activity *is ready* to access the shared resource.
- **orange** (optionally): the coordinating activity “soon” going to become (not) ready to access the shared resource.

The coordinating activity reacts to the coordinated activities’ events, and decides which **flag** to give to which activity. Examples of the **traffic light** coordination are:

- the operation of manufacturing work cells in a manufacturing line, where the “line” is the resource all workcells have to access in an orderly fashion.
- the access of multiple robot arms to the same material or tool feeder, without “fighting” for those resources.
- the deployment and runtime management of several threads in one process, which have to share several resources: RAM memory for their code and data blocks, communication buffer memory, and CPU time.

There exists a “mirror” version of the semantics above, **to synchronize** activities that must start doing something together at the same time; for example, a set of devices next to a conveyor belt have to start a particular behaviour together with the start of the conveyor belt. The semantics of the events fired from the coordinating activity to all of the coordinated activities is:

- **red**: none of the coordinated activities *is allowed to start* its behaviour.
- **green**: each of the coordinated activities *must start* its behaviour.
- **orange** (optionally): each of the coordinated activities *must prepare itself to start* its behaviour, because the synchronisation is going to be started “soon”.

The semantics of the events fired from a coordinated activity to the coordinating activity is:

- **red**: the coordinated activity *can not start* its behaviour.
- **green**: the coordinated activity *has started* its behaviour.

- **orange** (optionally): the coordinated activity is *soon ready to start* its behaviour.

The **Life Cycle State Machine** of Fig. 2.17 uses the same color code, for its super-states:

- **red**: the state of the LCSM is **deploying**; the activity *can not* provide its capabilities/services to others.
- **orange**: the state of the LCSM is **active** and *can switch “soon”* to the **ready** state (that is, either **running** or **pausing**).
- **green**: the activity is **running** or **pausing**, and *can switch* between both states *instantaneously*.

The most **complex version** of multi-activity coordination (with a widely accepted semantics) uses **protocols** instead of traffic lights, that is, the activities must exchange several messages before they can agree on a coordinated action. For example, a *prepare to transition* state that waits for a confirmation that “the other side” has indeed made its transition before committing to the intended transition itself. This last decision can be “undone”, until the commitment event has been processed. **two-phase commit protocol**. The **Petri net** meta model is often used to represent such coordination protocols.

2.10.11 Bad and Good Practices in FSM usage

- *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 the FSM, and interconnect them with an event communication; and design in extra states and logic rules in the FSM such that the behaviour is robust against the undeterministic computation and communication timing of the monitoring events.
- *bad*: to add the “guard” (monitoring) computation to a transition, since that makes the transition take an 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`.

2.10.12 Transition systems for the runtime creation of FSMs

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 `sec:higher-order` 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* [27, 28].)

2.11 Data, uncertainty and information

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: since 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.

2.11.1 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 [36, 43]. 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 [36], 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** $I(M_1 \text{ AND } M_2|M_1 \text{ AND } C) = I(M_2|C)$

2.11.2 Mechanism: Bayes' rule for optimal transformation of data into information

(TODO: [92].)

2.11.3 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. [46, 59])

2.11.4 Policy: hypothesis tree for semi-optimal information processing

(TODO: [21, 68].)

2.11.5 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**.

2.12 Formal representation languages: mature standards

Meta models for some of the representations discussed in this Chapter have already been developed (and to some extent also formalized) over the years, and several of those 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.

2.12.1 QUDT and UCUM

The *quantity-unit-dimension-type* meta model has already been formalized several times, for example in the **QUDT** initiative [86]. 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**(uantity) 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.

Next to QUDT, promoted by the **W3C**, a similar model exists, **UCUM** (the *Unified Code for Units of Measure*), which is promoted by the **Eclipse** eco-system.

2.12.2 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 2.18.

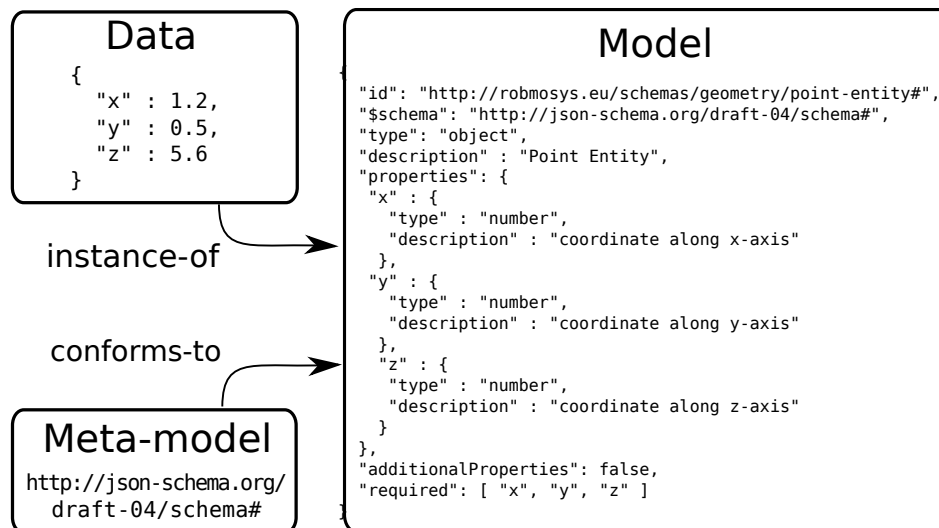


Figure 2.18: A valid data instance of a JSON-Schema Model representing three coordinates. The schema includes few constraints on the data structure, such as the values required for the validation of the JSON document. Moreover, the schema **conforms-to** a specific meta-model of JSON-Schema (draft-04).

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 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 2.18 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.

2.12.3 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.)

2.12.4 RDF1.1

RDF:

(TODO: relation to JSON-LD: minor difference in flexibility *to name* graphs; lack of keywords to represent `Semantic_ID` natively.)

2.12.5 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.

2.12.6 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.

2.12.7 FlatBuffers, Protocol Buffers

[FlatBuffers](#) and [Protocol Buffers](#) are more recent but well-supported alternatives. Their designs have been optimized for efficient runtime data processing and messaging¹⁸ and *self-description*, but not really for knowledge representation and reasoning.

2.12.8 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.

¹⁸For example, by the [Apache Arrow](#) project.

2.12.9 DFDL

DFDL (Data Format Description Language, “Daffodil”): (TODO: XML, less developed for scientific abstract data types like multi-dimensional arrays;)

2.12.10 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*).

2.12.11 Differential geometry

Differential geometry (e.g., [16]) 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.

Chapter 3

Meta models for the geometry of rigid bodies and polygonal worlds

The focus of this Chapter is on the **polygonal** models, because they are a universal base for (i) any other “smoother” representation of geometry, and (ii) any composition of the three top-level robotic “services”, namely manipulation by grippers; reaching by arms; navigating by wheels, propellers or legs. The simplest geometric models of all the mentioned entities are polygonal (“**stick figures**” and “**boxes**”), as is any base map of a world representation (“**areas**” with “**semantic tag**” points attached to them). Adding **geometric chain** relations (graph, or sequence, of geometrical primitive and relations, with (not necessarily rigid) constraints between entities), **kinematic** relations (rigid joints connecting rigid bodies) and **dynamic** relations (inertia, damping, elasticity) to the meta models is straightforward: each of these extensions is **composed** into a polygonal geometric base model, via “semantic tag” **attachment points** with **relations** with the semantic tag properties of already existing other parts of the geometric model.

3.1 The meta meta models

Robotic devices (and most other cyber-physical devices as well) take up space in the real world, and they have to coordinate their own motions with those of other devices, objects and humans. Irrespective of whether they are part of the system or live outside the system, as “disturbances”. Hence, models must be available to let the devices exchange information about how each of them is going to move in the near future, about which device gets spatial access to some areas, and about how the coordination of such motions and accesses must be realised. This, in turn, requires the devices to be able to reason about the space they occupy themselves, the space other “devices” occupy, and how these evolve over time. For robots, the (probably) most important knowledge is geometrical: how do they steer their links, connected by their joints, to reconfigure their own pose, and to move around in the world as an indivisible device. This Chapter introduces meta models for the most common geometrical entities and relations in robotics, in the 2D Euclidean **plane** and the 3D Euclidean

space, with the 1D space of the “(Euclidean) **line**” as a useful boundary case:¹

- the **point**, **line** and **line segment** (being the compositions of two points), and **polygon** (being the composition of several line segments). The absence of infinitely extended geometrical entities like lines or planes is intentional: they fit well with abstract mathematics (“just” adding some constraints on the extension of the mathematical concepts), but are not at all necessary for modelling any type of robotic system. (Or for perceiving, controlling, monitoring, . . . them.)
- the specific-because-omnipresent composition of all of the above primitives into a **rigid body**: a rigid body model adds (i) **constant relative distance constraints** between all of its parts, and (ii) **semantic tags** attached to the points, segments and polygons, to serve as “unique ID pointers” to which other composition models can refer to.
- their associated entities and relations of **(relative) position** and **direction**,
- and their time derivatives of **velocity** and **acceleration**.
- their associated relation of **(polygonal) shape**.
- the composite relation of **geometric chain** (more precisely, “geometric graph”), that represents a (partial) order on the relative positions and/or orientations of geometric primitives, without any mechanical motion constraints between them, and possibly with an evolution over time.

Position, velocity and acceleration composed together form the parts in the mereological meta model of what this document calls **motion**. There is no motivation to consider position, velocity and acceleration as separate relations on geometric primitives, because any object that moves or does not move has *always* a *state* of motion with all three motion components. The meta models of many of these entities and relations have already been consolidated in mature mathematical formalisations. These are not described in this document, but referred to as *meta meta models*.

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. 3.5.

3.1.1 semantic.IDs for geometry in 1D, 2D and 3D

Section 1.2.5 describes the generic foundations of semantic meta data, namely the usage of IDs as “symbolic pointers” between models and meta models. This Section adds geometry-specific specialisations, and explaining that model in human-centered form is the subject of this Chapter. The suggested MMID is

$$\{\text{MMID} : [\text{"Geometry"}, \text{"3D"}, \text{"Euclidean"}, \text{"Polygon"}] \}, \quad (3.1)$$

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

¹The generic term *space* will be used to denote any line, plane or space.

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. 3.1.2); the last tag identifies a the *most specific* geometric entity or relation. 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. 3.4.1) is the core meta model, representing a **manifold**² of zero-dimensional (“0D”) **Points**, and with simple composition rules to build all other entities. The 3D meta model (Sec. 3.4.2) 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.

3.1.2 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** 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) **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

²A *manifold* is **continuously mappable** on R^n , with R the *real line*. Hence, R^n is always one of the meta meta models of any geometric meta model in this document.

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.

3.1.3 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 [45, 52, 53]; 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 [52].

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.

3.1.4 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**, [16, 31]. 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”³, respectively, [67, 70, 91].

3.1.5 Qualitative spatial relations

(TODO: connects, contains, borders, intersects, on_top_of, Left_of, behind,..., from own or others’ point of view; in addition to mereological has-a **whole-part** relation. Qualitative **Spatio-temporal reasoning**, **spatial relations** between regions in space, e.g. **Region Connection Calculus**, **DE-9IM**, or *Cross Calculi* [12, 48].)

3.2 Taxonomy of geometric meta models

This document introduces the **partially ordered hierarchical structure** of Fig. 9.1 as a modelling axiom (a so-called **taxonomy**) for the (mostly geometric parts of) **motion/perception/world modelling stacks** in robotics. 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*⁴ 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. 1.3.1, 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*”

³This is not a commonly used term in the English-language scientific literature. Gauss [32] introduced the concept with the German term “*Zwang*”.

⁴The taxonomy includes **dynamics** too, which is covered by the mathematical theory of **differential geometry**.

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. 1.3.2 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.
- **tessellation** (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.

- 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.

3.3 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 3.1–3.2.

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 [11, 24], and these semantic relations have the **dependency structure** of Fig. 3.1. 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 meta data 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.

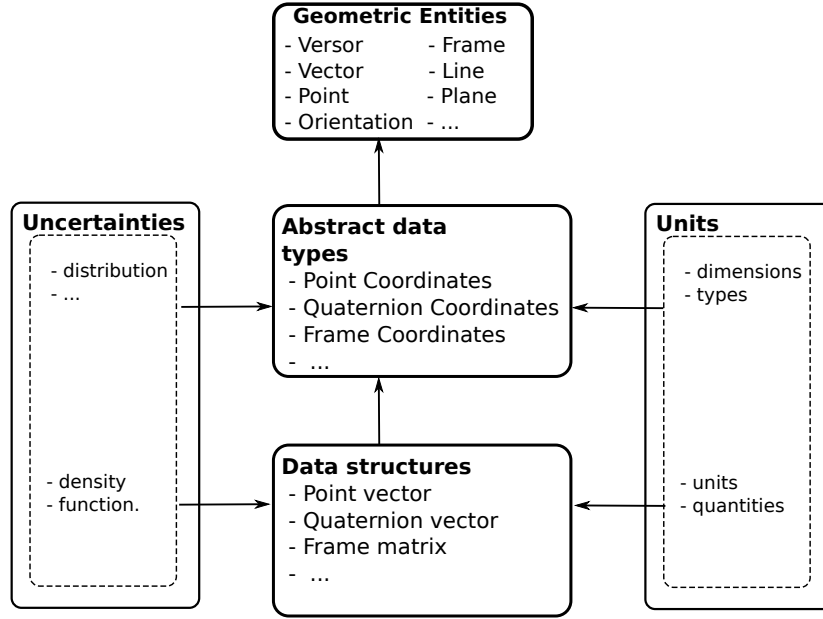


Figure 3.1: The mereo-topological model of the representations of geometric entities and relations, that is, the container entity for “mathematical” (i.e., *geometrical*), “abstract data type” and “data structure” representations. An arrow represents *composition* of complementary aspects in that numerical representation. Models of physical units *must* be composed always; models of uncertainties *can* be composed, when needed.

3.4 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 3.4.1–3.4.4), (ii) abstract data types for symbolic model validation and model-to-model transformations (Sec. 3.4.6), and (iii) data structures to carry **coordinates** (Sec. 3.4.8). 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 [sec:meta-model-geometric-chain](#)] *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](#).⁵

3.4.1 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

$$\{ \text{Point} : \text{aPoint} \}, \quad (3.2)$$

to identify a model of a **Point**. With all [Semantic_ID information](#), the full version of such a **Point relation** is depicted in Fig. 3.2. The following Sections will only use the short version.

```
{
  [ { MMID: ["Geometry","E2"] },
    { MID: "Point" },
    { ID: "Point-E2-xy34s" }
    // a unique identity code
  ],
  { Arguments: [ {PointName: "aPoint"} ] }
}
```

Figure 3.2: Mereological model of a **Point**, with [Semantic_ID](#) meta data.

Vector — This **ordered** list of two **Point** entities adds three constraints: (i) the *ordering* that gives an **orientation** to the **Vector**, (i) 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:

$$\{ \text{Vector} : [\{ \text{start} : \text{Point-E2-xy34s} \}, \{ \text{end} : \text{Point-E2-567j3} \}] \}. \quad (3.3)$$

⁵With clearly identified advantages with respect to the robotics *de facto* standard [URDF](#): all possible kinematic chains can be represented and not just tree structures with one-dimensional revolute or prismatic joints; any other meta model can be *composed* with the kinematic chain model without requiring that other meta model being visible or even known; in particular, any type of actuator and control algorithm, including [multi-articular](#) configurations.

The model above uses ID strings that refer to **Point** models as in Fig. 3.2; 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.

Orientation⁶ 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.

⁶**Right_handed** and **Left_handed** are often used synonyms.

Frame is an entity that only has meaning in the *Euclidean* context, because it adds two **metric constraints**⁷ 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. 3.4.8), 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:

$$\{ \text{Polygon} : [\{\text{start} : \text{aPoint-ID}\}, \text{bPoint-ID}, \dots, \text{zPoint-ID}, \{\text{end} : \text{aPoint-ID}\}] \}. \quad (3.9)$$

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:

$$\begin{aligned} \{ \text{Polygon} : [\{\text{start} : \text{aPoint-ID}\}, \text{bPoint-ID}, \dots, \text{zPoint-ID}, \{\text{end} : \text{aPoint-ID}\}], \quad (3.10) \\ \{\text{interior} : \text{iPoint-ID}\} \\ \}. \end{aligned}$$

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:

$$\{ \text{Polygon_pair} : [\text{P1} : [\{\text{start} : \text{aPoint}\}, \text{bPoint}, \dots, \{\text{end} : \text{aPoint}\}], \quad (3.11)$$

$$\text{P2} : [\{\text{start} : \text{Apoint}\}, \text{BPoint}, \dots, \{\text{end} : \text{Apoint}\}] \quad (3.12)$$

$$], \quad (3.13)$$

$$\{\text{interior} : \text{iPoint}\} \quad (3.14)$$

$$\}. \quad (3.15)$$

⁷These constraints are the same in 2D and 3D.

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**.

3.4.2 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 **Polyline**s. The notation is illustrated with the simple example of the **Polyhedron** in Fig. 3.3:

$$\{ \text{Polyhedron} : [\text{Pol1} : [\{\text{start} : \text{Pnt2-ID}\}, \text{Pnt3-ID}, \text{Pnt4-ID}, \{\text{end} : \text{Pnt2-ID}\}], \quad (3.16)$$

$$\text{Pol2} : [\{\text{start} : \text{Pnt1-ID}\}, \text{Pnt4-ID}, \text{Pnt3-ID}, \{\text{end} : \text{Pnt1-ID}\}], \quad (3.17)$$

$$\text{Pol3} : [\{\text{start} : \text{Pnt1-ID}\}, \text{Pnt3-ID}, \text{Pnt2-ID}, \{\text{end} : \text{Pnt1-ID}\}], \quad (3.18)$$

$$\text{Pol4} : [\{\text{start} : \text{Pnt1-ID}\}, \text{Pnt4-ID}, \text{Pnt2-ID}, \{\text{end} : \text{Pnt1-ID}\}] \quad (3.19)$$

$$] \}. \quad (3.20)$$

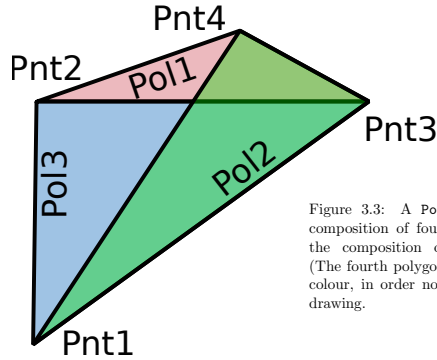


Figure 3.3: A Polyhedron as the composition of four Polygons, each the composition of three Points. (The fourth polygon is not shown in colour, in order not to overload the drawing.

The model above could have been composed differently: some of the four Polygons could 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**.

3.4.3 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:⁸

$$\{ \text{Position} : [\{\text{of} : \text{fPoint-ID}\}, \{\text{with_respect_to} : \text{ePoint}\}] \}, \quad (3.28)$$

that is, the position of point **fPoint** *from* point **ePoint**. The notation of the **Displacement** is a very simple extension to the **Vector**:

$$\{ \text{Displacement} : \{ \quad \quad \quad (3.29) \\ \quad \quad \quad \{ \text{Point} : \text{Point-ID} \}, \\ \quad \quad \quad [\{\text{start} : \text{sPoint-ID}\}, \{\text{end} : \text{ePoint-ID}\}], \\ \quad \quad \quad \}. \}$$

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.

3.4.4 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**.

⁸In other words, both **Position** relations are constrained by a higher-order relation, that of being each others’ inverse relations.

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), [18], who formulated⁹ the following theorem: the most general **Displacement** for a rigid body is a **screw motion**, [5, 7, 14, 35], i.e., there exists a line in space (called the **screw axis**, [7, 41, 69], 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, [45, 52, 53]; 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

$$\{ \text{Position} : [\{\text{of} : \text{fFrame-ID}\}, \{\text{with_respect_to} : \text{gFrame-ID}\}] \}. \quad (3.33)$$

⁹The notion of twist axis was probably already discovered many years before Chasles (the earliest reference seems to be the Italian Giulio Mozzi (1763), [17, 33, 57]) but he normally gets the credit.

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:

$$\{ \text{Orientation} : [\{\text{of} : \{\text{Rigid_body} : \text{aBody-ID}\}\}, \quad (3.34) \\ \{\text{with_respect_to} : \{\text{Rigid_body} : \text{bBody-ID}\}\}] \\ \}.$$

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:

$$\{ \text{Pose} : [\{\text{of} : \{\text{Rigid_body} : \text{aBody-ID}\}\}, \quad (3.35) \\ \{\text{with_respect_to} : \{\text{Rigid_body} : \text{bBody-ID}\}\}] \\ \}.$$

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.

3.4.5 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 `sec:geom-relations-abstract-data-type` **Coordinates** values are introduces in the `sec:meta-model-geometric-chain` Section on geometric chains.

Examples of such mereo-topological relations are depicted in Fig. 3.5: a distance between a **Point** `pPoint-ID2` and a **Line** `Line1` can be interpreted as the *shortest* distance between `pPoint-ID2` and another point lying on the **Line**, or as the length of the projection of `pPoint-ID2` 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, [42]?; harmonize symbols of points and lines with previous subsections in this Chapter.)

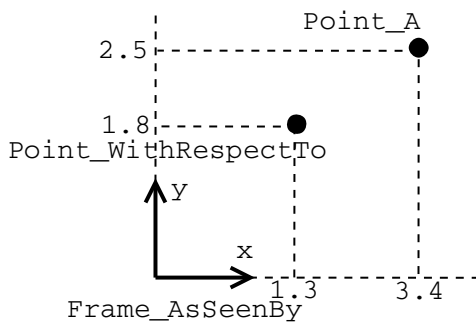


Figure 3.4: The three geometrical entities involved in a generic **Coordinates** abstract data type (in a 2D case): the **Point** (“**Point_A**”) whose coordinates are being represented; the entity **with_respect_to** which the **Position** coordinates are represented (in this case, this is also a **Point**, 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**.

3.4.6 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. 3.4 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**.

```
{ 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**.

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.
- 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**.

```

{ 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.

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. 1.2.3) and `semantic_ID` meta data (Sec. 1.2.5), 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` meta data. 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” meta data that is the same in different arguments, to the

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.)

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.

3.4.7 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 meta data 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**.

3.4.8 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. 3.4, 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,¹⁰ or (vi) whether several coordinate representations share the same units and reference frame.

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. 3.9 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:

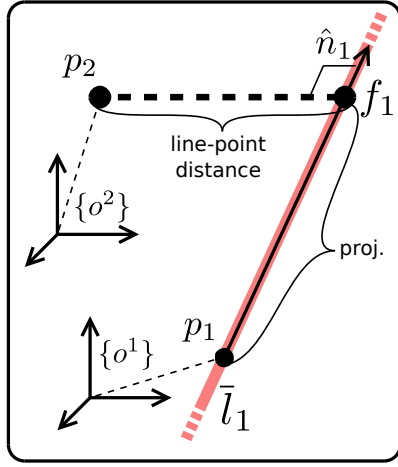
- a **Point** has no (need for a) coordinate representation; a unique symbolic **Semantic_ID** suffices.
- coordinates are needed in the *quantitative* representation of the **Position** of a **Point**. The coordinates represent the relative position of the **Point** to other geometric primitives.
- 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.)

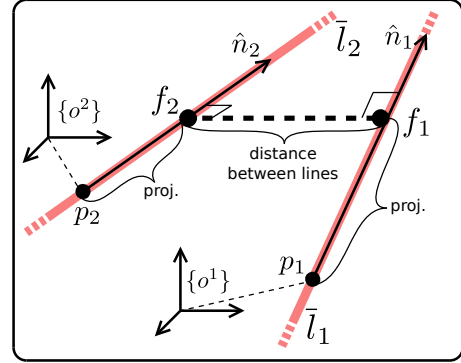
3.4.9 Constraint relations on Coordinates — Geometric_chain

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

¹⁰The **TopoJSON** standard uses this approach.



(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 3.5: Graphical representations of the five possible relations between a point and a line 3.5a, and between two lines 3.5b.

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	

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. 3.6) by (i) observing how it moves with respect to the table, and (i) remembering where the cup was on the table previously.

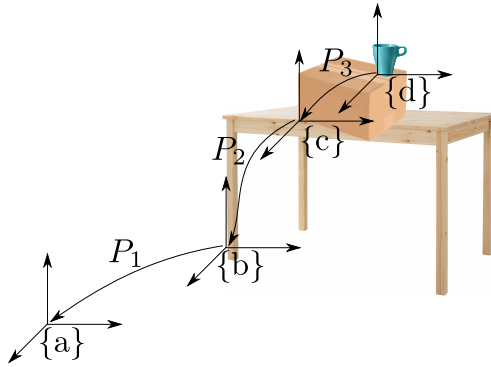


Figure 3.6: A `Geometric_chain` representing the relative `Poses` of a cup, a block and a table, where the constraints between them are of the `on-top-of` type.

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

Figure 3.6 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. 3.6). 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.

3.5 Meta models of mechanical dynamics: composing force and motion

The previous Section considered only 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?

3.5.1 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 Poincot (1777–1859), [64], 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 3.6 shows an example of a **Force** representation using the **screw_axis** model.

(TODO: complete formal semantics, with examples. Including transformation and mapping operators.)

3.5.2 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 [23], Lagrange [49], Hamilton [38], Gauss [32], Jourdain [44] or de Maupertuis [25]. 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.

3.5.3 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 example¹¹ of such a meta model, with its foundations built on the flows of energy via [Block-Port-Connector](#) mechanism.

3.5.4 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:

- **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**”, [22].
- 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](#).

¹¹Unfortunately, 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.

- **co-tangent space** $\overline{TM}(p, f)$ of **linear forms** f at **Position** \mathbf{p} that **map Velocities** v in the tangent space at p into a real scalar value w :

$$f : TM(p) \rightarrow \mathbb{R} : v \mapsto f(v) = \langle f, v \rangle = w. \quad (3.38)$$

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 :

$$\mathcal{M} : TM(p) \rightarrow \overline{TM}(p) : v \mapsto \mathcal{M}(v) = m. \quad (3.39)$$

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 :

$$\mathcal{M} : TM(p) \rightarrow \mathbb{R} : v \mapsto \frac{1}{2} \langle \mathcal{M}(v), v \rangle = w. \quad (3.40)$$

The resulting energy is the **Kinetic energy** stored in the moving mass. It can serve as a **metric** on TM [16].

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* [32], or *acceleration energy* [83].

- **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} .

$$\mathcal{D} : TM(p) \rightarrow \overline{TM}(p) : v \mapsto \mathcal{D}(v) = f_d. \quad (3.41)$$

Hence, the pairing of the friction force f_d with the velocity v which causes the friction maps into a real scalar w :

$$\mathcal{D} : TM(p) \rightarrow \mathbb{R} : v \mapsto \mathcal{D}(v) = \langle f_d, v \rangle = w. \quad (3.42)$$

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} .

$$\mathcal{K} : M(p) \times M(p') \rightarrow \overline{TM}(p) : (p, p') \mapsto \mathcal{K}(p, p') = f_e. \quad (3.43)$$

Hence, the pairing of the elastic force f_e with the displacement (p, p') which causes it maps into a real scalar w :

$$\mathcal{K} : M(p_1, p_2) \rightarrow \mathbb{R} : \text{Displacement}(p, p') \mapsto \mathcal{K}(p, p') = w. \quad (3.44)$$

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:

$$f = M a, \quad (3.45)$$

$$f = D v, \quad (3.46)$$

$$f = K \Delta p. \quad (3.47)$$

```
{ 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 3.6: Example of a **Coordinates** model of the **Force** on a **Frame "b"** with respect to a **Frame "r"**, and using the **screw** representation.

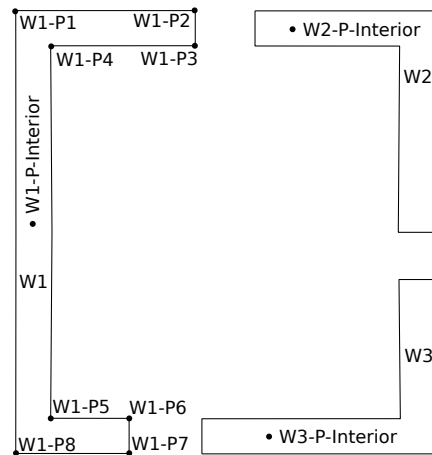
3.6 Composition relations: Map as set of geometric entities

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 3.4.1–3.4.2. 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 3.7–3.8 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. 3.7.

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.



3.7 Composition relations: Semantic_map, World_model

A **Semantic_map** is a **Map** composed with **semantic tags** (or “annotations”, or “meta data”) 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 `sec:metameta-task` Task meta model, a **Semantic_map** becomes a **World_model**. Figures 3.9–3.10 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**.)

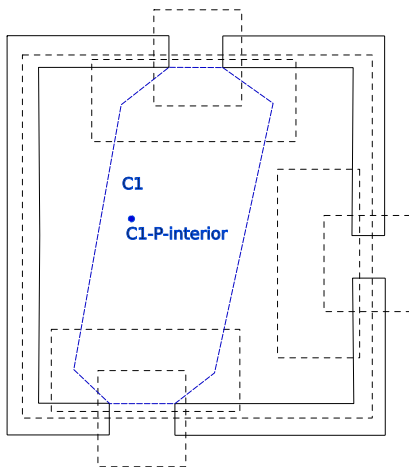


Figure 3.9: The Map of Fig. 3.8 is extended with one particular extra *area* that receives the semantic meaning of a **corridor**. That is, the semantic tag carries an **intention** of the purpose of that area.

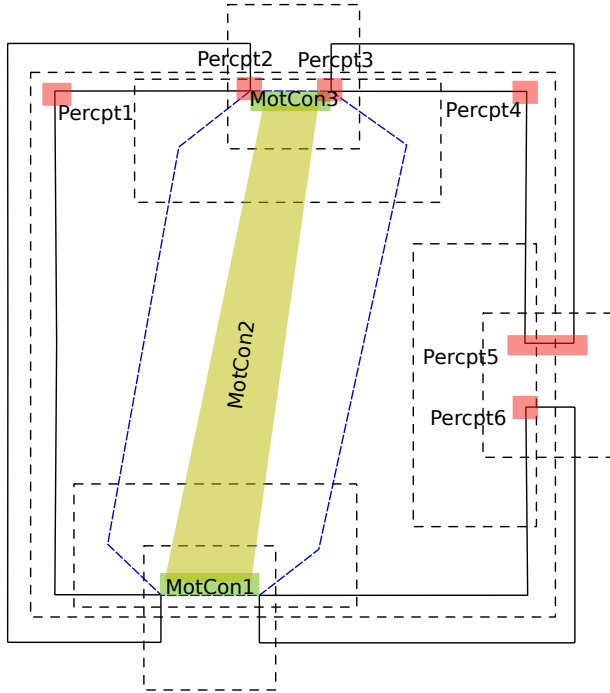


Figure 3.10: Map of the **motion** and **perception** areas that are relevant for a particular Task in the Map of Fig. 3.9. In order to follow that (virtual) “corridor”, the robot software should take into account the light-green “control” constraints, and the light-red areas that are best fits for the robot’s “perception” capabilities.

3.8 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. 3.11. 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**.

3.8.1 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;

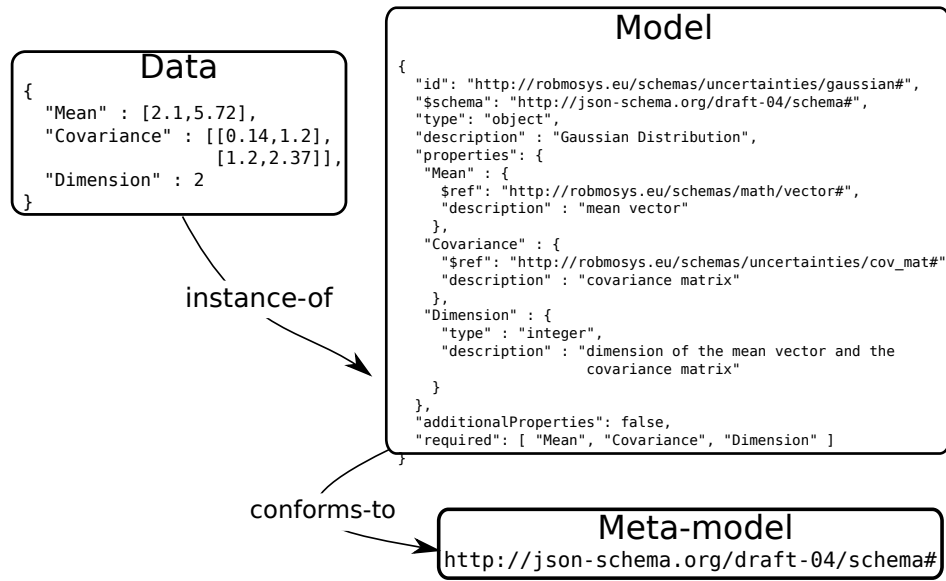


Figure 3.11: A valid data instance of a **JSON-Schema** model representing a Gaussian distribution. The schema is a composition of other schemas (for vectors and covariance matrices) and includes a few constraints on the data structure, such as the values required for the validation of the JSON document. Moreover, the schema **conforms-to** a specific meta-model of **JSON-Schema**.

- **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.

3.8.2 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.

3.9 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 2.12 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 3.12; a concrete example is discussed in Figure 2.18.

The following Sections explain the popular **Coordinates** representation approaches in robotics.

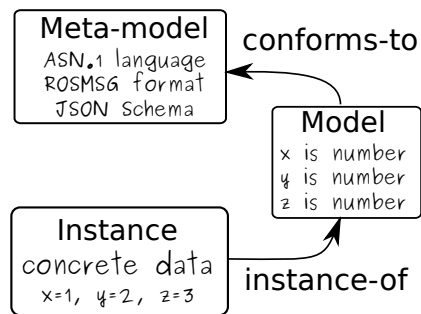


Figure 3.12: Digital representation models: a data structure in software is an **instance-of** a digital data representation model, which is a formal description that **conforms-to** a meta-model. A concrete example is shown in Fig. 2.18.

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 (meta-data), 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.

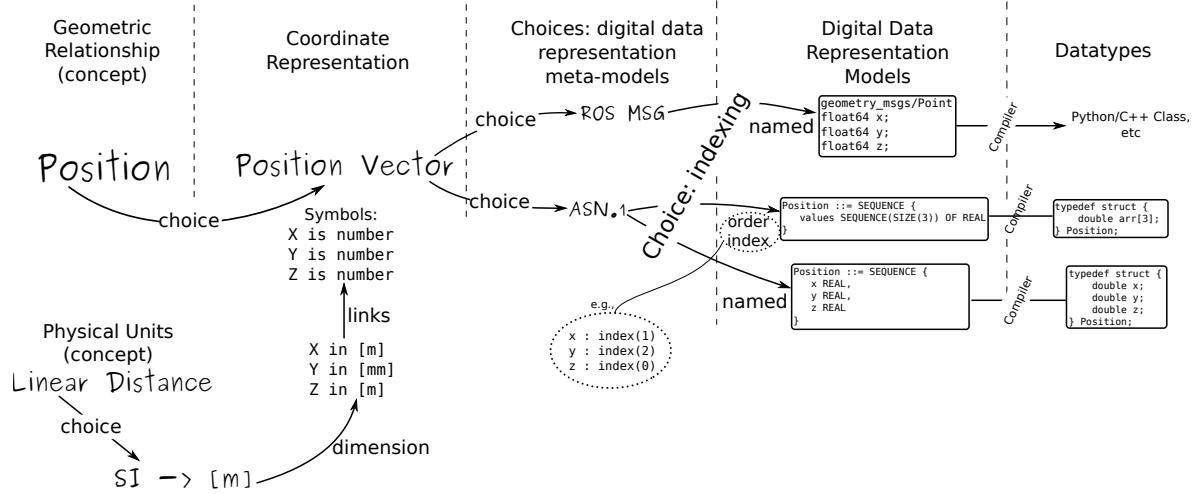


Figure 3.13: From geometric relations to digital data representation: choices for the grounding of the concept of *Position*.

3.9.1 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 constraints over a numerical value, nor specific padding and alignment information. Moreover, there is no built-in enumeration values, which is usually solved with few workarounds¹². However, default values assignment is possible in the ROS message models. ROS messages are self-describing by

¹²An `UInt8` type with unique default value assigned for each enumeration item.

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 mechanism¹³).

3.9.2 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.

3.9.3 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*¹⁴ (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.

¹³It is possible to have other representations over ROS messages, e.g., JSON documents, by using a simple `std_msgs/String` message.

¹⁴see <http://www.cs.wustl.edu/~schmidt/ACE.html>

Chapter 4

Meta models for a kinematic chain and its instantaneous motion

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):

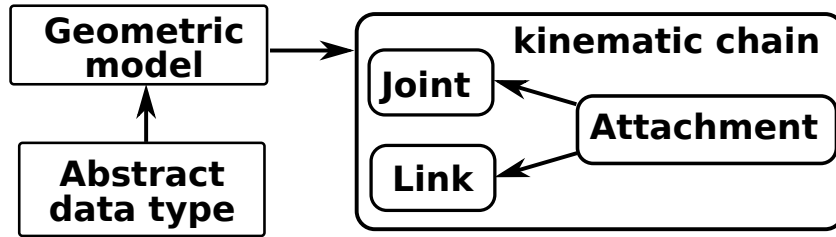


Figure 4.1: A `Kinematic_chain` model is a graph of `Link` and `Joint` entities, which are themselves defined as compositions of *geometric* entities (Sec. 3.4, Fig. 9.1) and their *abstract data type* representations. Each link can have one or more `Attachments`, to allow extensions by means of *model 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

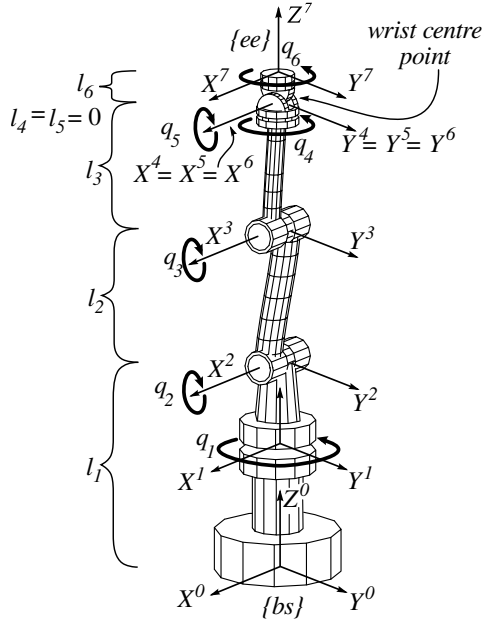


Figure 4.2: The most common kinematic design of *industrial manipulator arms*, using only revolute joints as bi-directional motion constraint between seven rigid bodies. This particular geometric configuration is a *kinematic family* parameterized by the symbols in the drawing. What is not symbolically 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**:

$$\text{Position (Link_1, Link2)} = f(q), \quad (4.1)$$

$$\text{Velocity (Link_1, Link2)} = g(q, \dot{q}), \quad (4.2)$$

$$\text{Acceleration (Link_1, Link2)} = h(q, \dot{q}, \ddot{q}), \quad (4.3)$$

- **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).

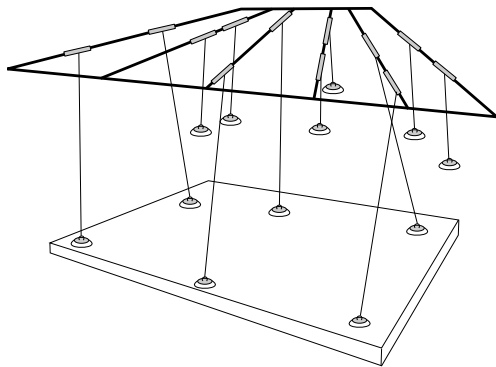


Figure 4.3: A *cable support* as uni-directional motion constraint between two rigid bodies. The mechanical framework on the top contains motors to control the length of the cables, as well as their position on the horizontal guides. The load is depicted at the bottom of the drawing, with unspecified “attachment tools” between cables and load.

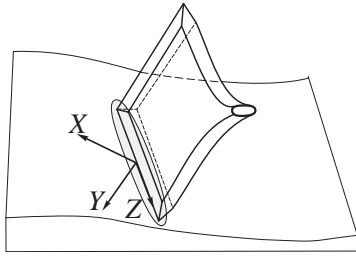


Figure 4.4: An *edge-surface* contact as uni-directional motion constraint between two rigid bodies. This model represents (part of) the geometrical abstraction of real-world motion constraints such as wheels or skates.

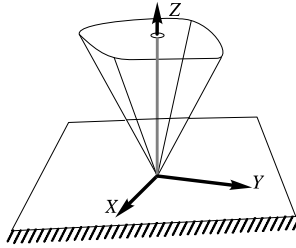


Figure 4.5: A *vertex-surface* contact as uni-directional motion constraint between two rigid bodies.

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. 3.4.
- 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.

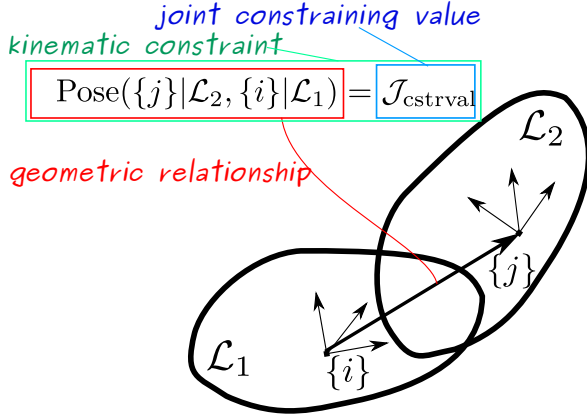


Figure 4.6: A generic model of the mathematical expression for the simplest `Kinematic_chain`, namely one with two `Links` and one `Joint`. The `Joint` has an unspecified type, but it has a `state` representation. That means that the relative `Pose` relation can be expressed as a *bijection* between 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. 3.4.7. 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. 4.4.

```

{ 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. 4.2.
- 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-putput 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 model for the mechanical dynamics of a kinematic chain

This Section extends the `geometrical Coordinates` meta model by adding the `mechanical` entities and relations. These represent 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`).

4.2.1 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. 3.5.1, 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.

4.2.2 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.)

4.2.3 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.

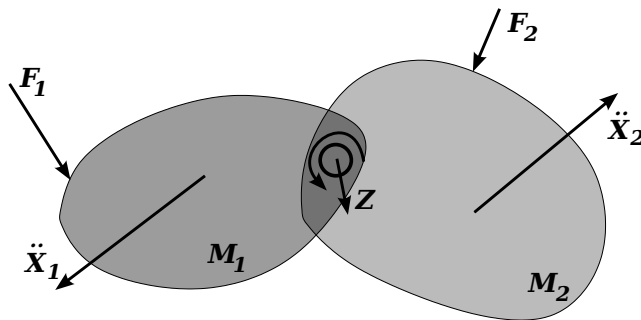


Figure 4.7: The entities involved in the dynamic behaviour of a (revolute) joint constraint. The spatial direction (Z), force (F) and acceleration (\ddot{X}) have *six* degrees of freedom; the mass (M) is a 6×6 relation between them.

4.2.4 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):

$$\mathbf{F}_1 = \mathbf{M}_1^a \ddot{\mathbf{X}}_1, \quad (4.4)$$

$$\text{with } \mathbf{M}_1^a = \mathbf{M}_1 + \underbrace{\left(\mathbf{I} - \mathbf{Z} (\mathbf{Z}^T \mathbf{M}_2 \mathbf{Z})^{-1} \mathbf{Z}^T \right) \mathbf{M}_2}_{\mathbf{P}}. \quad (4.5)$$

\mathbf{M}_1^a the so-called *articulated body inertia* [29], 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 $\mathbf{P}\mathbf{P} = \mathbf{P}$.

4.3 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.

4.3.1 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** [32, 65, 83, 84]. 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.

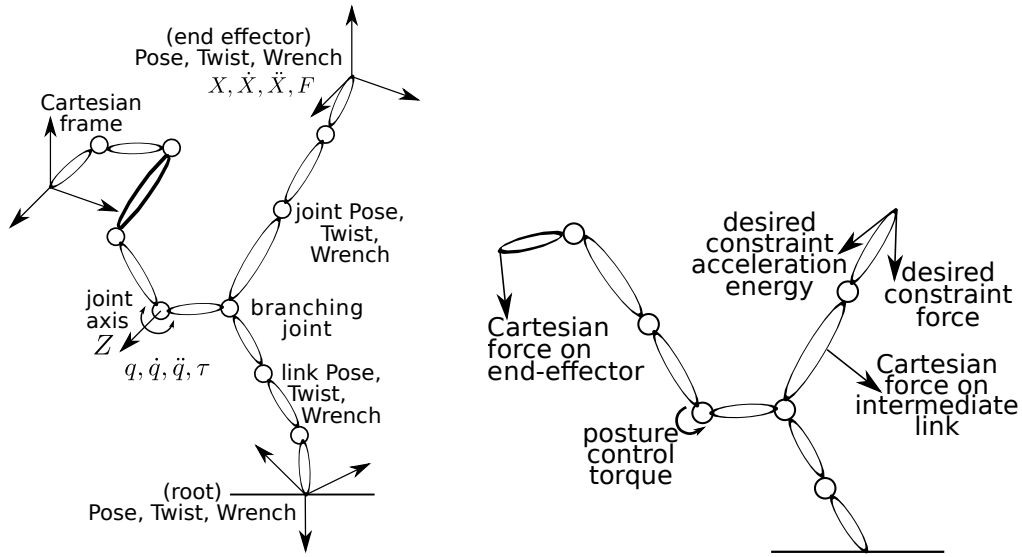


Figure 4.8: The entities in a geometric and mechano-dynamical model of a kinematic chain. On the left, all the type of variables that make up the *state* of the chain. On the right, the three possible types of *drivers* for the instantaneous motion (*change of state*) of a chain: (i) a torque generated 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*.

4.3.2 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.

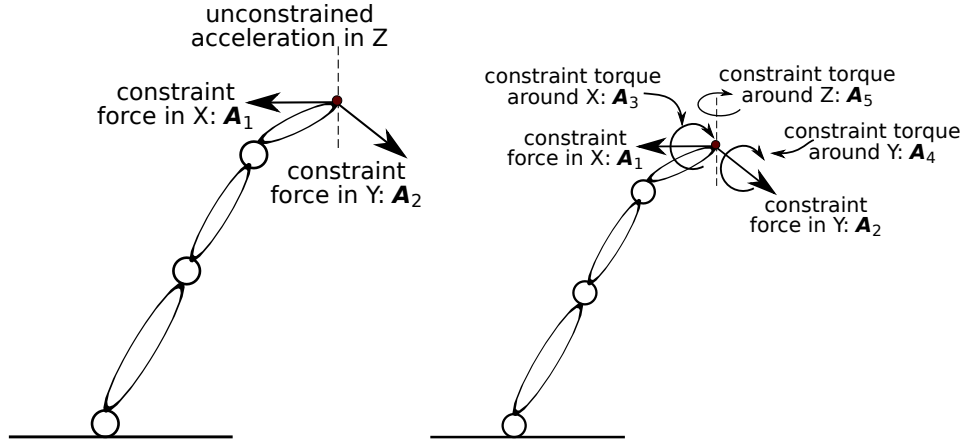


Figure 4.9: Two examples of artificial motion constraint specification via acceleration constraints. On the left: the last **Link** of the chain is constrained to have one of its **Points** move on a vertical line (irrespective of the orientation of the **Link** with respect to that line), by introducing 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 4.6 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* [9].

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 4.2 gives an example of such a mereo-topological specification model. The extension with *abstract data type* models is done in Sec. 4.6.

4.3.3 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 4.2: 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** [29]. 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`.

4.4 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`.

4.4.1 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`.*

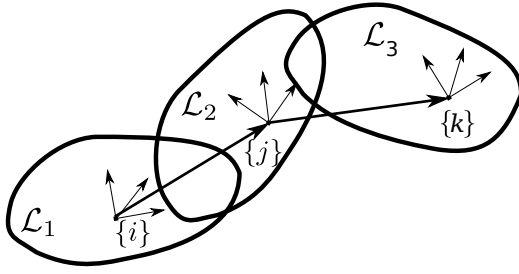


Figure 4.10: Serial composition of kinematic chains.

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. 4.2.4.

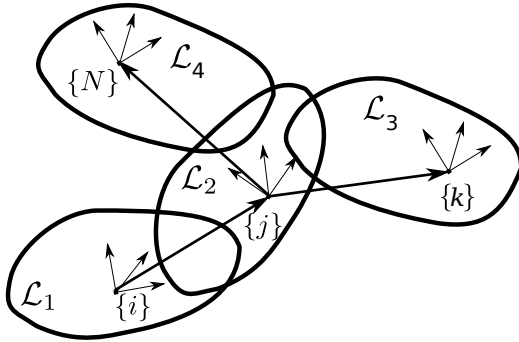


Figure 4.11: Branch composition of kinematic chains. The composite chain has three branches, and **Link** \mathcal{L}_2 is the **branch_link**.

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

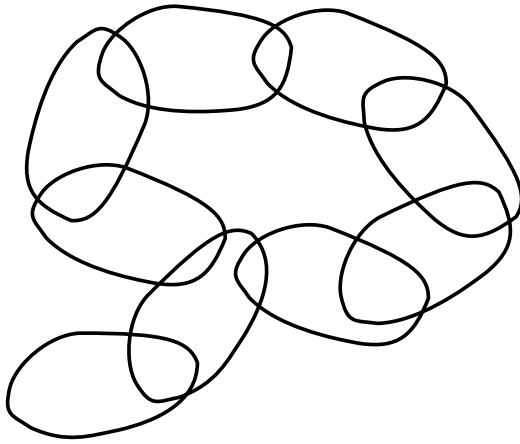


Figure 4.12: Loop composition of kinematic chains.

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. 4.4.2), 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.

4.4.2 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.

4.4.3 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.

4.4.4 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.

4.5 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*.

4.5.1 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.

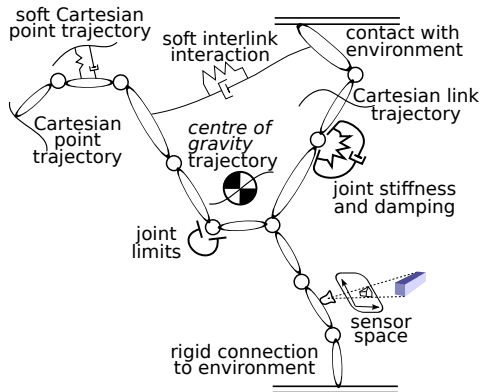


Figure 4.13: Sketch of the kinematic chain model of a dual-arm manipulator, with all(?) possible motion constraints on links and joints.

4.5.2 Mobile platform chains

- DifferentialDrive
- Holonomic:
- EightWheelDrive: (Fig. 4.14)

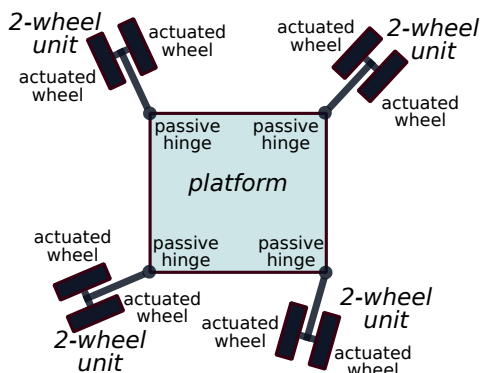


Figure 4.14: Sketch of the kinematic chain model of an over-actuated mobile robot: eight actuated wheels, connected in so-called 2WD (“two-wheel drive”) pairs, that in turn are connected to a rigid platform via passive revolute joints with a caster offset. The design drivers behind this platform are (i) passive backdrivability in all configurations, and (ii) holonomic motion behaviour in all configurations.

4.5.3 Parallel chains

- **Delta:**
- **Stewart-Gough:**

(Fig. 4.15):

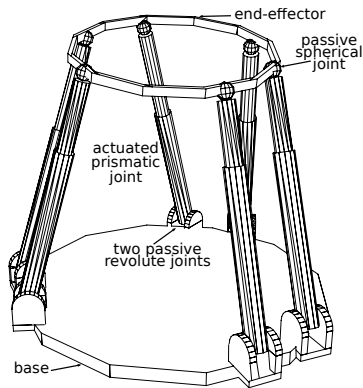


Figure 4.15: Sketch of the kinematic chain model of a parallel kinematic chain, with six legs each with six 1D joints.

4.5.4 Multirotor chains

4.5.5 Hybrid chains

featuring one or more “loops” in the chain’s topology.

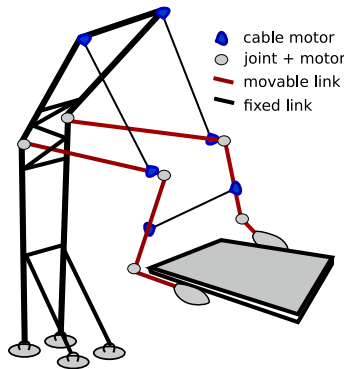


Figure 4.16: Sketch of the kinematic chain model of a cable-driven robot, with a “hybrid” topological structure.

4.5.6 Cable-driven chains

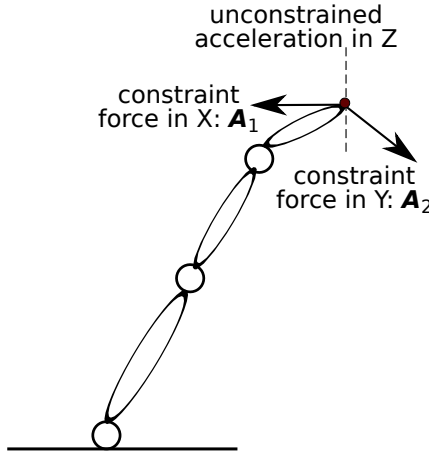
Fig. 4.16,

4.6 Solver for hybrid kinematics and dynamics

Section 4.2 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.

4.6.1 Mechanism–I: motion drivers

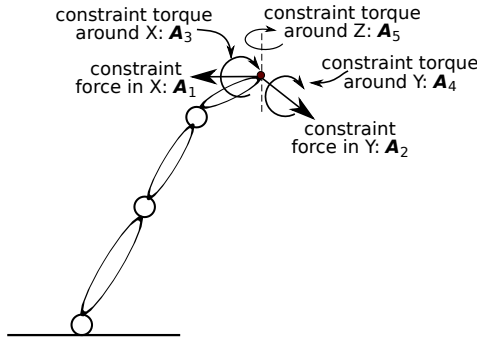
Section 4.3 introduced the mereo-topological models for the three different “motion drivers” with which to specify the instantaneous motion of a kinematic chain: joint torques, Cartesian forces, and Cartesian acceleration energy constraints. Acceleration energy is represented as a product of acceleration 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:



$$\mathbf{A} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}. \quad (4.6)$$

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:



$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}. \quad (4.7)$$

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:

$$\mathbf{A} = \mathbf{1}_{6 \times 6}, \quad \mathbf{b} = \mathbf{b}_d. \quad (4.8)$$

4.6.2 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 computations² that provide the “inverse dynamics” of a serial robot, bolted to the ground, with N joints, one external force, and one acceleration constraint.

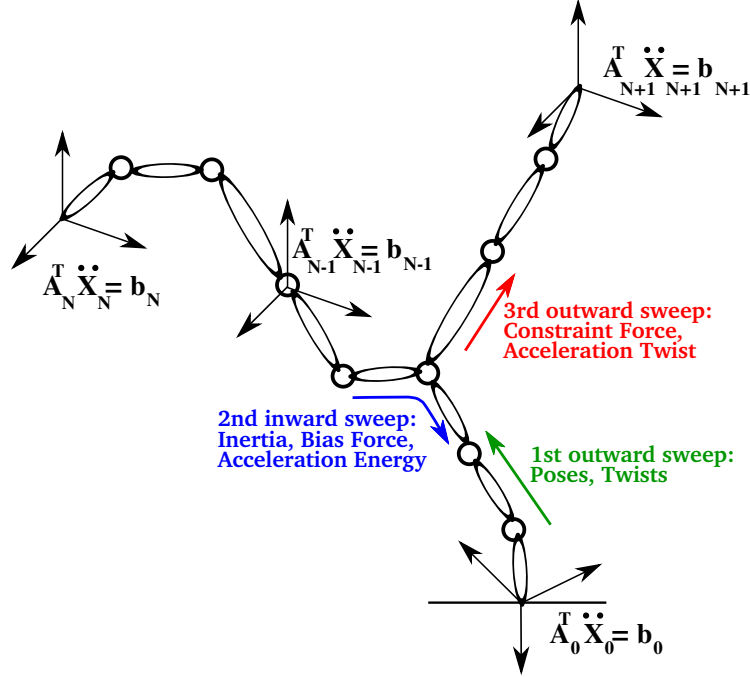


Figure 4.17: A hybrid dynamics solver uses the topological structure of the kinematic chain to schedule all the behavioural functions required to answer a particular query.

The *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.

²The bookkeeping of the indices is not yet fully consistent in the presented pseudo-code...

Algorithm 1: Hybrid dynamics solver, according to Popov-Vereshchagin [84]

```

begin
  // outward sweep, to compute the motion state:
  for  $i \leftarrow 0$  to  $N - 1$  do
     ${}^{p_{i+1}}_p \mathbf{T} = {}^{d_i}_p \mathbf{T} {}^{p_{i+1}}_{d_i} \mathbf{T}(q_i)$  ;
     $\boldsymbol{\omega}_{i+1} = \boldsymbol{\omega}_i + \dot{q}_{i+1} \mathbf{Z}_{i+1}$  ;
     $\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{r}^{i+1,i} \times \boldsymbol{\omega}_i$  ;
     $\ddot{\mathbf{X}}_{b,i+1} = \begin{pmatrix} \dot{q}_{i+1} \boldsymbol{\omega}_i \times \mathbf{Z}_{i+1} \\ \boldsymbol{\omega}_i \times (\mathbf{r}^{i+1,i} \times \boldsymbol{\omega}_i) \end{pmatrix}$  ;
  // inward sweep, to compute the force and acceleration factorization:
  for  $i \leftarrow (N - 1)$  to  $0$  do
     $\mathbf{P}_{i+1} = \mathbf{1} - \mathbf{M}_{i+1}(\mathbf{Z}_{i+1}^T \mathbf{M}_{i+1}^a \mathbf{Z}_{i+1})^{-1} \mathbf{Z}_{i+1}^T$  ;
     $\mathbf{M}_i^a = \mathbf{M}_i + \mathbf{P}_{i+1} \mathbf{M}_{i+1}$  ;
     $\mathbf{F}_i = \mathbf{P}_{i+1} \mathbf{F}_{i+1} - \mathbf{M}_{i+1}^a \mathbf{Z}_{i+1}(\mathbf{Z}_{i+1}^T \mathbf{M}_{i+1}^a \mathbf{Z}_{i+1})^{-1} \tau_{i+1} + \mathbf{F}_i^b + \mathbf{F}_i^e$  ;
     $\mathbf{A}_i = \mathbf{P}_{i+1} \mathbf{A}_{i+1}$  ;
     $\beta_i = \beta_{i+1} + \mathbf{A}_{i+1}^T \left\{ \ddot{\mathbf{X}}_{i+1} + \mathbf{Z}_i D_i^{-1} \left( \tau_{i+1} - \mathbf{Z}_i^T (\mathbf{F}_{i+1} + \mathbf{M}_{i+1}^a \ddot{\mathbf{X}}_{i+1}) \right) \right\}$  ;
    with  $D_i = \mathbf{Z}_i^T \mathbf{M}_i^a \mathbf{Z}_i$ , and  $\beta_N = 0$  ;
     $\mathbf{Z}_i = \mathbf{Z}_{i+1} - \mathbf{A}_{i+1}^T \mathbf{Z}_i D_i^{-1} \mathbf{Z}_i^T \mathbf{A}_{i+1}$ ,  $\mathbf{Z}_N = 0$  ;
  // Lagrange multipliers of acceleration constraint forces:
   $\mathbf{Z}_0 \boldsymbol{\nu} = \mathbf{b}_N - \mathbf{A}_0^T \ddot{\mathbf{X}}_0 - \beta_0$  ;
  // outward sweep to compute joint torques and link accelerations:
  for  $i \leftarrow 1$  to  $N$  do
     $\ddot{q}_i = (\mathbf{Z}_{i-1}^T \mathbf{M}_i^a \mathbf{Z}_{i-1})^{-1} \left\{ \tau_i - \mathbf{Z}_{i-1}^T (\mathbf{F}_i + \mathbf{M}_i^a \ddot{\mathbf{X}}_{i-1} + \mathbf{A}_i \boldsymbol{\nu}) \right\}$  ;
     $\ddot{\mathbf{X}}_i = \ddot{\mathbf{X}}_{i-1} + \ddot{q}_i \mathbf{Z}_i + \ddot{\mathbf{X}}_{b,i}$  ;

```

- 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.
- 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.

4.6.3 Policy: scheduling options in the third sweep

- when solving for the Lagrange multipliers $\boldsymbol{\nu}$, 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 $\boldsymbol{\nu}$), 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.

4.6.4 Policy: free-floating base

(TODO)

4.6.5 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)

4.6.6 Policy: deployment in an event loop

The discussions around Algorithm 1 in Sec. 4.6 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)

4.6.7 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.)

4.6.8 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. 4.18.

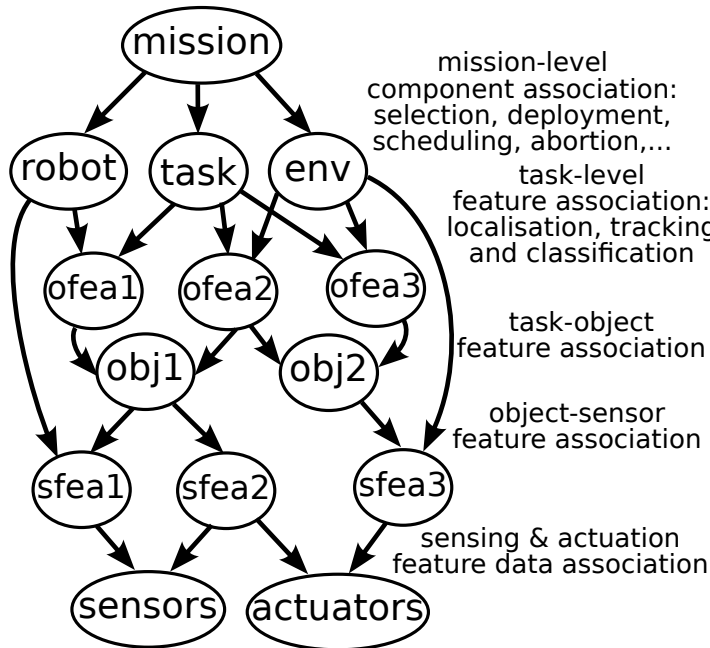


Figure 4.18: The mereo-topological model of the entities and relations needed to model **motion-perception fusion**. The model conforms to the meta models of, both, “Bayesian networks” and **Tasks**.

For example, a visual servoing task (depicted as one of the many types of task specifications in Fig. 5.20), 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.

4.6.9 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.)

4.6.10 Dynamics of serial kinematic chain

(TODO: queries for forward, inverse, hybrid transformations; motion, force and solving sweeps.)

4.6.11 Dynamics of branched kinematic chain

(TODO: queries for forward, inverse, hybrid transformations; motion, force and solving sweeps.)

4.6.12 Dynamics of kinematic chain with a loop

(TODO: queries for forward, inverse, hybrid transformations; motion, force and solving sweeps.)

Chapter 5

Meta models for robotic tasks

This Chapter extends the generic **task** meta model of Sec. 2.7 (repeated below for the convenience of summarizing its essentials) with the **natural** and **artificial** relations that appear between entities in **actions** executed by **robots**. The “task” represents **what** has to be done and under which constraints, while the “action” represents **how** a task is realised while satisfying the task constraints. In other words, a task model provides the **property graph** with the **higher-order relations** that give the **context** of a robot’s actions, and the **execution** of the actions requires **reasoning** in that context, via **graph traversals**. This Chapter provides several structures to guide both the context modelling and the reasoning.

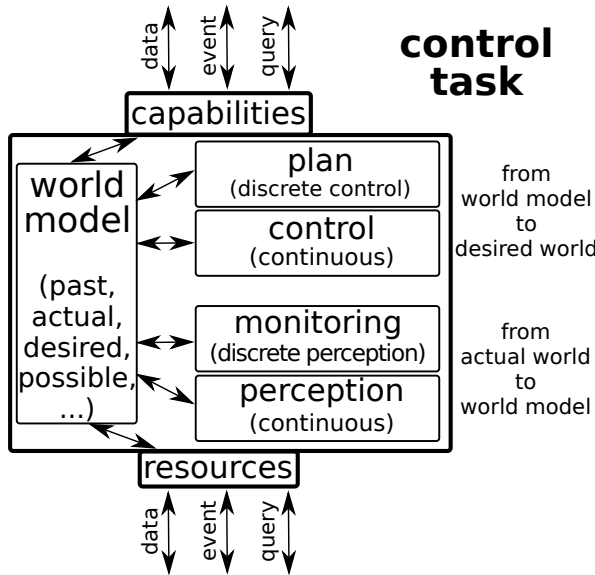


Figure 5.1: Figure 2.8, repeated for convenience. At the end of the day, there is only one thing that makes a robot really move, and that is its continuous-space *control*. All other parts in the Task meta model are “just” there to provide the right context in which to set the control parameters (setpoints, gains, system model, ...). The paradigmatic foundation of the presented task meta model is that *all* couplings between these parts take place via the entities and relations in the representation of how the world looks like, including the relations that exist between different entities in the world. In other words, *the* important knowledge relations in the figure are the ones that are *not* there, that is, relations between only two parts of the mereological entities on the right-hand side of the schema.

5.1 Simplest task: proprioceptive guarded motion

Figures 5.2–5.2 show examples of robot task specifications, in the form of so-called **guarded motions**. This is a robot programming approach that goes already a long way back in the

history of robotics [39, 56, 88], because it is the simplest way to realise the full task meta model:

- the robot’s model *is* the world model.
- the plan has only one action.
- the control has a constant specification.
- perception and monitoring use only state variables from inside the robot controller.

The “guard” part of the term refers to the monitoring: the motion goes on until a particular relation between observed state variables is reached.

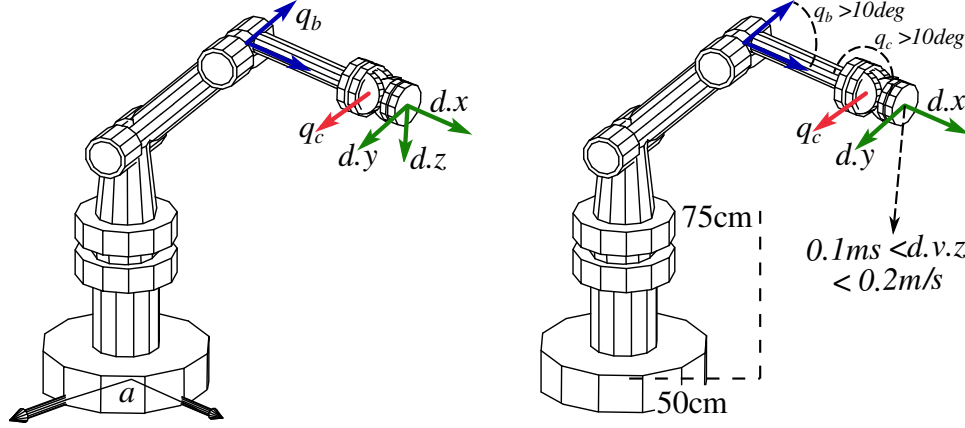


Figure 5.2: Example of a proprioceptive guarded motion. The model of the robot’s `Kinematic_chain` is extended with the `Attachments a, b, c` and `d`, via which the task is formally specified as a set of desired/monitored parameter ranges.

The following specification example makes these statement more concrete for the robot manipulator example of Fig. 5.2. (The example does not make a statement about which concrete *syntax* to use for the guarded motion [domain-specific language](#).) Ideally, *all* “magic numbers” in the specification above come from contextual relations; in the example above, three such relations are used: `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]
```

The above-mentioned specifications 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.

Figure 5.2 sketches another specification example, this time for a mobile robot moving inside of a room:

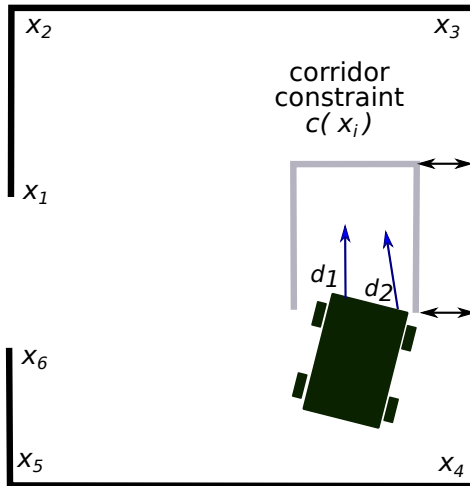


Figure 5.3: Guarded motion specification to make a mobile robot drive within a “corridor” that is defined with respect to the wall of a room. The specification uses the IDs of relevant points in the world model: d_1 and d_2 are the points on the robot where (virtual) actuation forces can be applied by the control system; x_1, \dots, x_6 are the corner points of the room.

```
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)
```

5.2 Action, actor, actant (object), activity, agent

This Section explains the relations between terms that occur often in this document.

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 and various levels of resolution, 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 action 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 the 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.¹

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 **software process** that implements the execution of an action, with “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.

5.3 Natural hierarchies in task models

The design of task models is a creative process, with a lot of possibilities for new compositions and the trade-offs that they bring. On the other hand, there are some natural, hierarchical constraints that have to be satisfied in the task design.

5.3.1 Hierarchy in 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 global navigation towards targets over the earth, driven by wheels, legs, propellers,...
- **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**: the “arms” or “manipulator” navigate towards targets in local 6D space.
- **grasp**: the “gripper” or “hands” to manipulate objects by *force closure*, *form closure*, or *non-prehensile* grasps. (TODO: references.)
- **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

5.3.2 Hierarchy in world model scope

This Section elaborates on the **robot-centric** part of composite task models, that is, those in which the **motion capabilities** of **only the kinematic chain** play the central role. (Later Sections add the possibility to extend these kinematic chain motion tasks with models of other robot action parts, e.g., perception actions to monitor the motion execution, or to update the world model information.) Robot motion capabilities grow in complexity according to the

¹The oldest(?) reference that explicitly introduces such natural hierarchy of **increasing order of intelligence with decreasing order of precision** is [71].

complexity of the plans the robots have to execute,² and another very useful **hierarchical structure** is represented by the following semantic tags:

- **move**: this represents the plans that can be realised by only the robot’s **instantaneous hybrid dynamics** (Sec. 4.2). In other words, the kinematic chain model of the robot *is* the world model, and its behaviour is that of an ideal mechanical energy transforming system.

For example, the instantaneous motion under the influence of a pushing force at the end-effector of 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.

- **moveGuarded**: one adds **monitoring** to a **move** plan, that is, some algorithms around the **proprio-sensing** of the robot determine when to stop the motion that goes on with a constant instantaneous specification.

Two typical use cases exist: (i) 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 (ii) the motion stops when a contact transition is detected via using current, force, tactile or IMU sensors mounted at various attachment points on the kinematic chain. Hence, *world model*, *perception* and *object affordances* models are restricted to what is directly attached to the robot’s kinematic chain.

- **moveTo**: one adds **extero-sensing perception** to a **move** or **moveGuarded** plan, so that the sensors localise and track **object** features in the environment, and adapt the **move/moveGuarded** properties accordingly.

The *world model* contains features of robot-external objects, with their robot-centric perception affordances, for example, *visual servoing*.

- **moveConstrained**: one adds **contact** perception to **moveTo**; these “contacts” can be physical, but they can also just exist as artificial constraints in the world model, to guide the robot’s motion.

For example, the robot is expected to slide a tool over a table, maintaining an interaction that is safe for the robot, the environment, and the tool.

When more than one robot is being composed into a motion task, the following “system of systems” coordination models are added:

- **moveCoordinated**: **one** sub-system (a robot or not) provides all other robots or non-robot sub-systems, online, with (i) individual motion specifications, and (ii) the events for the coordinated execution.

For example: dual-arm tasks performed by an *ABB Yumi*, two *KUKA iiwa*’s, a *PR2*,...

- **moveOrchestrated**: all motion subsystems have already the motion specifications **on-board**, and only the coordination events must be communicated.

²The plan complexity has, obviously, impact on the complexity of the other mereological parts of a task: control, perception, and world modelling.

For example: a robotic manufacturing cell, where all robots get the assembly programs from the cell supervisory system, together with the events to trigger their execution and (re)configuration.

- **moveChoreographed**: 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.

The modelling suggestions above are only *mereological*, hence a lot more detailed models must be provided, topological, geometrical, dynamical, etc. Section 5.4 provides a start of these modelling efforts.

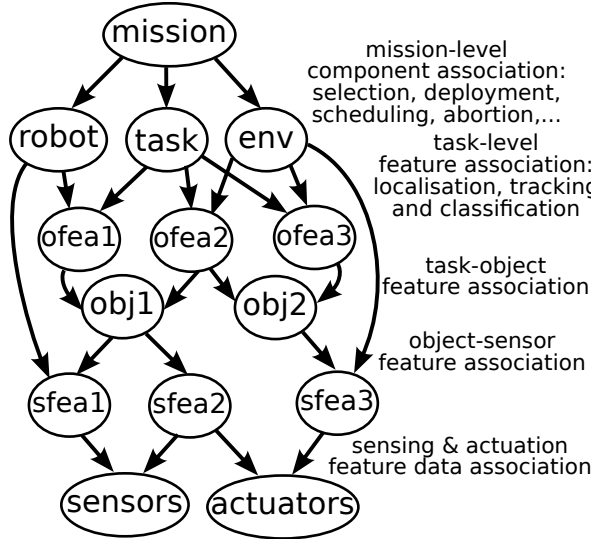


Figure 5.4: The natural hierarchical dependencies between components in the generic perception graph for robotic systems.

5.3.3 Hierarchy in perception graph

(TODO: explain Fig. 5.4.)

5.3.4 Hierarchy in cascaded control levels

The **natural hierarchy** in robotic motion stacks implies a hierarchy of **cascaded control loops**:

- **power inverter**, with typical **time constant** of 1/100.000th of a second.
- **electrical motor**: transforms electrical power into **mechanical torque**, via mechanisms such as **field oriented control**, with typical **time constant** of 1/10.000th of a second.

- **mechanical acceleration:** the generated torque results in acceleration of the attached mass; the time scales required in robotic applications lie around 1/1000th of a second and up.
- **mechanical velocity:** integrating 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](#).

5.3.5 Hierarchy in energy transformations

This Section combines all the semantics of the physical world introduced in the previous Sections, in a mereo-topological **hierarchical structural model** of motion stacks, from the “bottom” of **energy sources** up to the **coordinated motion** of several robotic systems. As always, the purpose of introducing an explicit hierarchy model is to provide a fundamental structure that can be exploited in the decision about how many formal models one uses to describe a domain, and about how they can be composed. For the sake of concreteness, the example of a *battery-driven mobile robots* (e.g, Fig. 5.5) is used to illustrate the structure:

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 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. 4.6.
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. 5.5), 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.

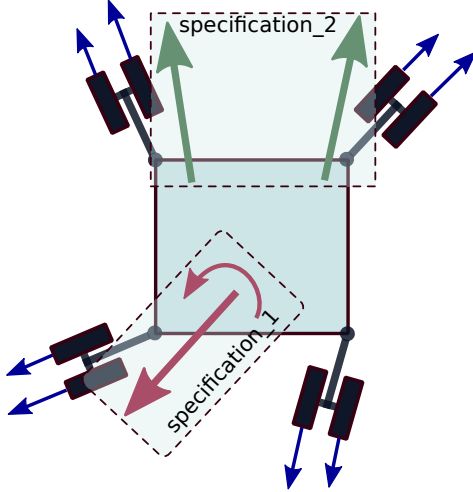


Figure 5.5: Transmissions of forces between actuated wheels (blue traction vectors along wheels’ rolling direction) and the (red) force and moment on the platform the wheels are attached to. (The green duo of forces represent an alternative way to specify desired/actual inputs to the robot platform.) This relationship model is composable because the combined effect of all actuator forces are physically realised by simple vector addition.

5.4 Mechanism: task specification as constrained optimization problem

A *composable* way to formulate a task specification is by means of a [constrained optimisation problem](#) that is formulated by the following collection of entities, relations and constraints, e.g. [8, 11, 26, 50, 60, 61, 74]:

- **configuration space**, of all the parameters in the models of the hierarchy in Sec. 5.3.5, e.g., the joint space parameters of a robot and its actuators, q , and Cartesian space parameters X ;
- **desired configuration** (X_d, q_d), which are 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)**, that is, relations $f(X, X_d, q)$ on these parameters that the task execution is expected to minimise, e.g., the delay in “progress” of the task execution,

the energy consumption of the robot, the distance to obstacles, or the closeness to a target pose;

- **constraints**, that is, 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, e.g., joint limits, or singular configurations in a kinematic chain.
- **tolerances** $d(X, X_d, q, q_d) \leq A$ describe how well the constraints have to be satisfied, or how far the objective functions have to be optimized.

The mathematical formalisation of a constrained optimization problem follows the template of Sec. 7.10, repeated here for convenience:

task state & domain	$X \in \mathcal{D}$
robot/actuator state & domain	$q \in \mathcal{Q}$
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	decide on switching

Representing a Task in this way gives 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**. One then needs a **solver** to generate (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 [1, 55] (which are online, reactive but they may suffer of local minima problems) and for plan-based solvers [50] (which are (typically but not necessarily) offline, and less reactive since they explore 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.

5.4.1 Policy: specify as objective function or as constraint

One should be careful about what to use as objective functions to optimize, for several reasons:

- functions like *time*, *energy consumption*, or *safety*, are *derived* quantities, whose values can not directly and uniquely be influenced by the actuator signals q . Hence, it’s often better to select them as inequality constraints, that monitors must follow during the

task execution to allow the plan to switch to another control approach when the realised time, energy, etc., falls outside of the prescribed 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.
- while it is mathematically easy to specify a *multi-objective optimization*, via a weighted combination of various objective functions, this *requires* a choice of weighing factors, but that is 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.

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 their expected ranges.

5.4.2 Policy: integrating multiple levels of abstraction in one task

Most real-world applications must combine several levels of abstraction (Sec. 5.3.4) in their control and the above-mentioned best practice is then applied in this multi-level control context in the following way: a “higher” control level influences a “lower” level (and vice versa) preferably via the addition of constraints and tolerances, but *not* via extra terms in the objective function.

For example, an electrical motor influences the mechanical joint motion control via motor 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 components’ behaviours with high predictability, except for (i) the exact time on which the controller will react, and (ii) the exact sequence of controller states that the system will evolve through.

5.4.3 Policy: event loops for task execution

(TODO: fastest loop: feedback control and its control-centric perception; second loop: preparations for next feedback control; third loop: perception, with its world modelling monitors and updates; fourth loop: task FSM.)

5.4.4 Policy: monitoring for hybrid constrained optimization

In general, task models require **monitoring** functionalities to determine, at runtime, whether the execution of the task specification is progressing correctly, that is, whether *all* the relations and constraints in the composed task model are satisfied within tolerance. Such online

monitoring allows to react to modeled conditions, both *desired* (i.e., the task execution obtained the desired outcome) and *undesired* (e.g., foreseen cases of non-nominal execution), and then to modify the behaviour of the robot. At the same time, the introduction of monitoring functionalities makes the Task specification into a **hybrid constrained optimization** problem, since the *plan* now needs a finite state machine to switch between different constrained optimizations, when reacting to the monitoring events.

The **moveGuarded** example in Fig. 5.6, applicable to a six degrees of freedom serial robot arm equipped with a force sensor, is probably the simplest model of a task specification that is composed with an online monitoring specification; the example specifies a *nominal* termination condition, which fires a *task accomplished successfully* event as soon as the condition is met.

```

move compliantly {
  with task frame directions
  xt: velocity 0 mm/sec
  yt: velocity 0 mm/sec
  zt: velocity v_des mm/sec
  axt: velocity 0 rad/sec
  ayt: velocity 0 rad/sec
  azt: velocity 0 rad/sec
} until zt force < -f_max N

```

Figure 5.6: Example of a guarded-motion task definition [15].

However, the task specification in Figure 5.6 does not specify:

- **non-nominal** conditions, which enable to react to non-nominal situations. In general, at least one non-nominal termination condition should be indicated, and it is denoted as a maximum deviation over the expected execution time of a motion task. Moreover, non-nominal conditions are not only used to evaluate a possible failure *after* the end of a motion, but also during the execution of the motion itself (i.e., *continuous monitoring*, and not only *discrete monitoring*);
- **tolerances**, both on the condition of nominal and non-nominal task execution.

In literature, there are few task specifications that include a monitor specification as a primitive (e.g., [1]). A composable robotic modelling approach requires a task specification modelling language that includes all kinds of monitoring, and this inevitably leads to a **graph of relations** around the nominal specification model. Some initial modelling efforts can be found in [74].

5.4.5 Policy: iterating feasible solutions during task execution

If one uses a constrained optimization formulation for a task, it is seldom mandatory that the computation of the optimum is effectively finished before the robot can start to act. Indeed, any **feasible solution** can be used, and the iterations towards a better solution can be spread over subsequent control time instances.

(TODO: examples.)

5.5 Bad practices in Task specification

(TODO: instantaneous reactive control, e.g., via potential fields; weighting of translation and rotation; weighting 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.)

5.6 Task examples: indoor and outdoor robot driving

The traffic system, or “driving” in general, is used throughout the Chapter as a familiar example of a formal representation of motions (i.e., driving in indoor corridors and rooms (Fig. 5.7), or in outdoor traffic lanes and parking lots) and of the perception required during those motions (i.e., recognizing building features or traffic signs, localizing them in their spatial context, and inferring their influence on the robot’s current action). Traffic lanes and signs are semantic tags in the world, that influence (i.e., constrain, as well as optimize) the driving behaviour of AGVs, cars, bikes and pedestrians, individual as well as groups. They model “motion” in a fully **declarative** way, because they do not model *how* a traffic user should move, but rather which relations the actual motion should satisfy. The real-world implementations of traffic signs are *designed to be perceivable* by human drivers in (almost) all weather conditions, and the areas they cover in the world are *designed* to fit to the *control bandwidths* that can be safely expected from (almost) 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.

The main purpose of this Section is to explain how *engineered systems* can make use of the rather abstract task meta model of Fig. 5.23 by concrete of examples of motion and perception models for a mobile robot driving around in the “traffic” of an office-like indoor environment. The first step in that direction is to add the **geometric** level of abstraction to the mereo-topological level of abstraction in the earlier task meta model. More in particular, the world model will now be filled with (models of) the shape of the environment in the neighbourhood of the robot, the location and shape of the traffic areas, as well as the shape and motion of the robot’s **kinematic chain** itself. 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; more in particular, 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.

The task specifications given in this Section will illustrate that the *world model* is indeed the modelling primitive that couples all of the other task aspects, of control (discrete plan, and continuous control) and perception (discrete monitoring, and continuous perception), to realise particular motion capabilities, and using particular resources.

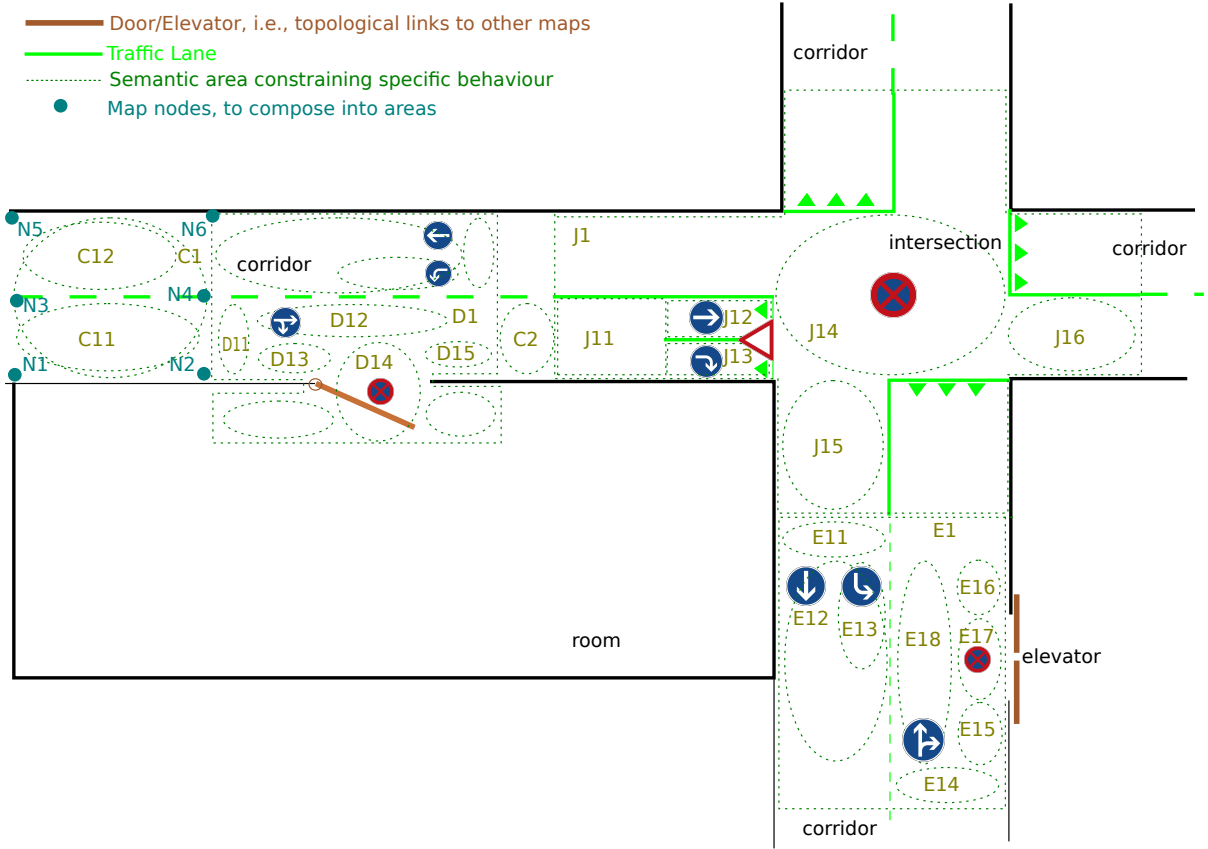


Figure 5.7: *World model* of an indoor “two-corridor-with-intersection” area: the solid black lines represent walls, and the red lines represent doors, while all other map entities are the **semantic tags** of the traffic signs and markings. These tags model the *motion constraints* that every robot must respect when it drives through the area, but they do not give information about how exactly the robot should move its wheels.

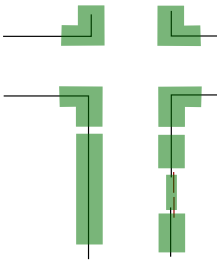


Figure 5.8: Semantic world model, featuring perception tags, for a laser range finder with sensor processing capabilities that can detect straight wall segments and rectangular corridor corners.

5.6.1 Geometric world models for control & perception integration

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, since they have to steer clear from collisions with these “hard” world landmarks. Some (possibly different) landmarks also serve the robots’ **perception**; for example, Fig. 5.8 represents the **perception** tags for a simple laser range finder type of sensor.

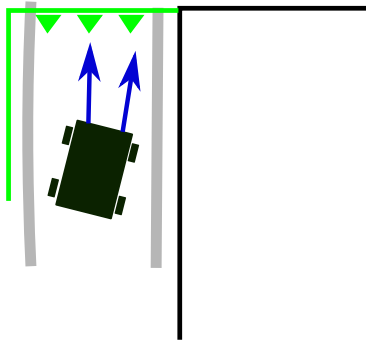


Figure 5.9: Semantic world model, extended with a motion **control** specification, in the form of a *tube* (the shaded gray area that represents the constraints of the control) and *motion drivers* (the blue arrows that represent (artificial) forces that generate the **instantaneous motion** of the robot).

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.

The **control** model can be as simple as the two blue arrows in Fig. 5.9, that represent two driving forces attached to the kinematic chain of the robot, and whose direction is determined by some of the target tags in the world model. These artificial forces are the inputs to a motion controller, that transforms them to actual actuating torques on the motors. The control algorithm in itself is simple and constant, and all of the time and context dependent model information is embedded in the world model.

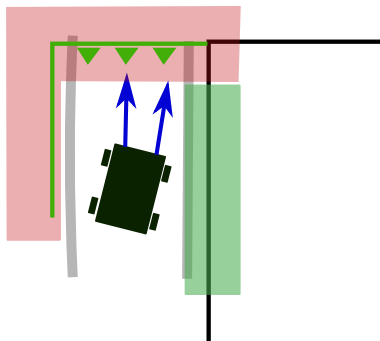


Figure 5.10: Semantic world model of Fig. 5.9, extended with the **local perception tags**: the green area focuses the perception of tracking a “wall” feature, at a resolution high enough to support the motion control; the red area focuses the monitoring on the detection of “any” feature in the direction of motion to react to, at a resolution low enough to react in time.

The same approach holds for the **perception**: Fig. 5.10 extends the world model with **semantic tags** for the **perception** part in the robot task model. Based on that information in the world model, the robot controller can *focus* its perception on just two areas: (i) the wall

to be followed in the motion, and (ii) the nearby “rest” of the environment in the direction of motion, to be monitored for the presence of “obstacles”. Detecting them signals the 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, since 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.

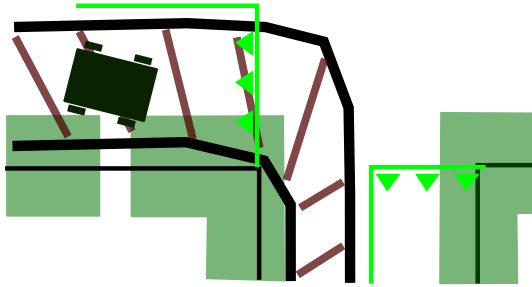


Figure 5.11: Semantic world model, with a task horizon that covers the sequential composition of two or more sub-tasks.

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 Fig. 5.10, but now one with a bend around the next expected corner. Again, nothing changes in the perception, control and monitoring functionalities, since all extensions are added to the world model, and to the plan.

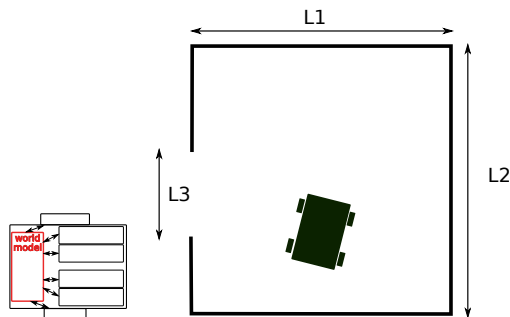


Figure 5.12: The **task** of the robot is to drive out of a rectangular room with a hole in one of its walls. The size and position of room and hole are unknown. The small figure on the left refers to the generic task model of Fig. 2.8, and indicates its parts that are involved; in this case, only the *world* is being modelled.

5.6.2 Example task plan: escape from a room

This Section provides the next step in the concrete design of the different **task** parts of the previous Section, for the **capability** shown in Fig. 5.12: *escape from a room*. One of the simplest possible versions of a **plan** 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.

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.

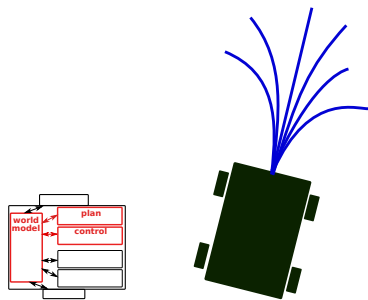


Figure 5.13: A possible *control* strategy for the escape-from-room capability in Fig. 5.14: to select from between a discrete set of pre-computed open loop trajectories, each corresponding to one particular time-invariant input at the actuators.

The **control** part of the task can as simple as making a selection between various “open loop” motion trajectories, Fig. 5.13:

- 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.

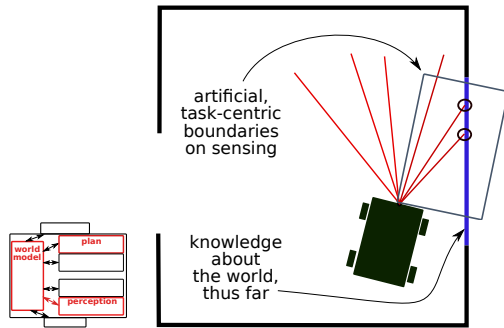


Figure 5.14: A possible *perception* strategy for the escape-from-room capability in Fig. 5.14.

The **perception** part does not need all the data provided by the distance sensor, since it could use the following algorithm (Fig. 5.14):

- select a *region of interest* (the grey box in Fig. 5.14) 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.

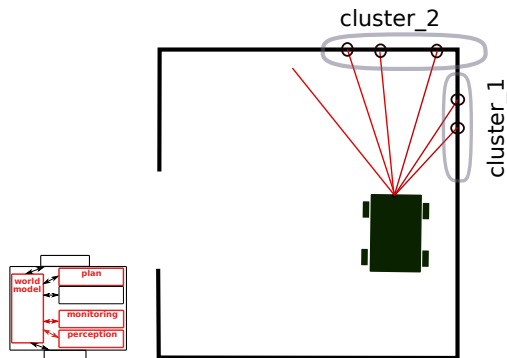


Figure 5.15: The *monitoring* strategy for the escape-from-room capability in Fig. 5.14 must follow the *wall on the right* (which is needed for the motion control) and look out for a *wall in front*.

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*.
3. the leftmost rays can be discarded, because they are outside of the scope of the *plan*, which reduces the computational load.
4. *all* measurements *could* be *neglected* until needed again, based on the *planned* speed of the motion.

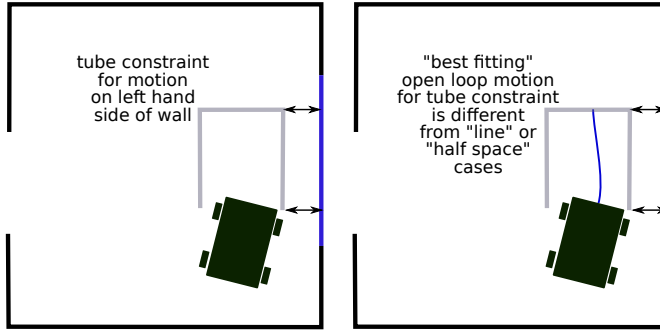


Figure 5.16: Left: a possible *motion specification*, in which a “tube” constrains the allowed motions, but does not command an explicit motion trajectory. Right: a corresponding *control* choice.

The **motion specification** needed in the **control** can be as simple as the “tubes” in Fig. 5.16:

- 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. 5.13 that fits best, according to a task-specific metric. **Alternatives** for the **control** design are depicted in Fig. 5.17.

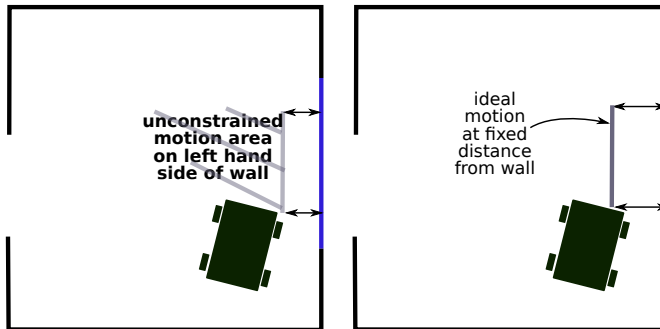


Figure 5.17: Two alternative controller approaches for the motion specification in Fig. 5.16. Left: a open half space, to the left of the wall. Right: a line trajectory at a specified distance from the wall.

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. 5.18. 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.

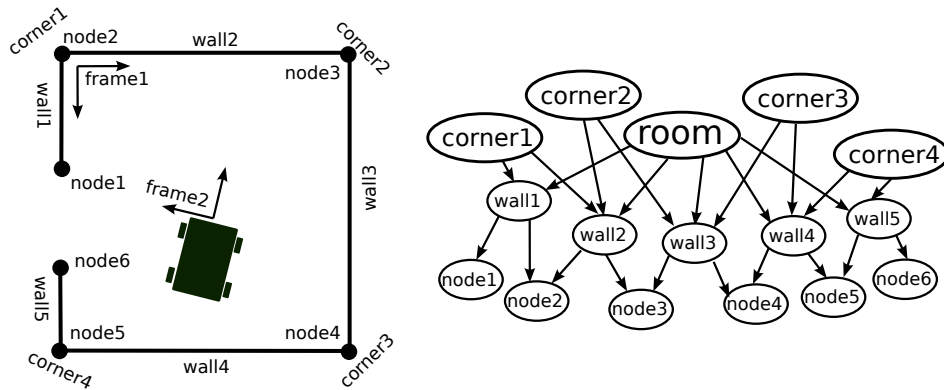


Figure 5.18: Right: topology of *room* model entities (“data structures”) and relations (**has-a**, and **connects**). Left: geometrical properties of all entities. The robot’s pose in the room can be represented numerically by position coordinates of the frame attached to the robot with respect to the frame attached to the room.

5.7 Semantic task specification: domain-specific languages

The following Sections give 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.

5.7.1 Task ontology

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.

5.7.2 The importance of task ontology standardization

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

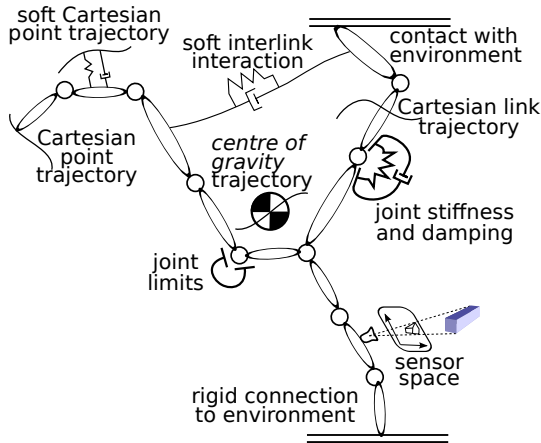


Figure 5.20: Various types of **artificial** motion constraints between (some of) the natural entities in Fig. 5.19. The figure uses a sketch of an “arm” robot, but similar constraint types apply to “mobile” robots too.

Fig. 5.20. The *coupling* of the motion capabilities of the kinematics chain, joints and actuators, to objects in the environment, is realised by adding *model attachment points* to the kinematic chain models: in the latter model, one just has to foresee that “something” can be coupled in, at any later time, and of any particular type. So, none of the kinematic chain model parameters should *depend* in any particular way on the coupled entities, and no information about the coupling relations or constraints should end up in models of kinematic chains. This design approach is an example of the best practice of the “[dependency inversion principle](#)”.

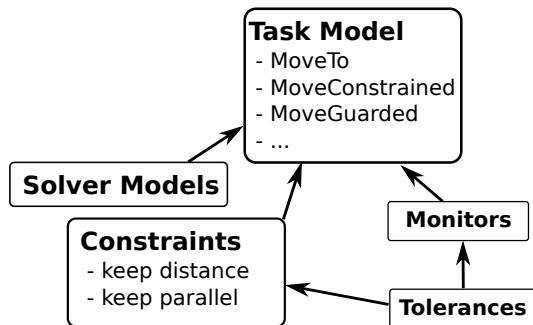


Figure 5.21: Overview of the models involved in the composition of a motion task. (The arrows represent **is-part-of** relations, i.e., inverse **has-a** relations.)

5.7.5 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 [34] 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,...).

5.7.6 Semantic robot arm motion primitives

The mobile robot example in the previous Sections is conceptually the easiest to grasp, since 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 sematic motion primitives can be defined. Figure 5.22 sketches some simple examples, with two levels of resolution in representing the mentioned configuration spaces.

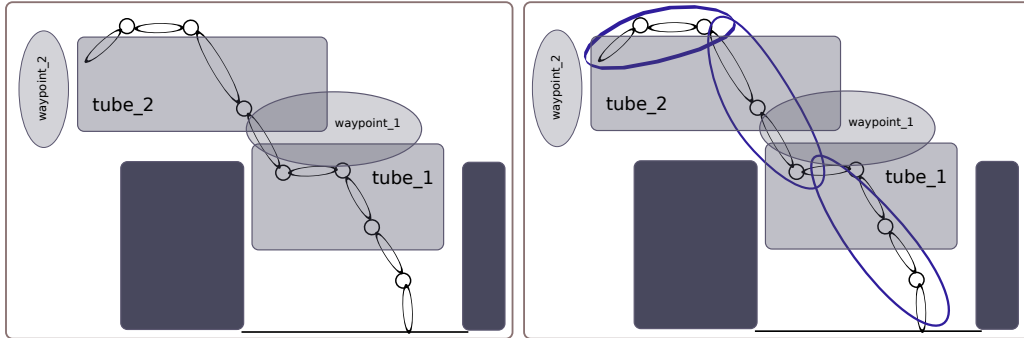


Figure 5.22: Model of a “high” (left) and a “low” (right, in blue) kinematic resolution motion plan for a serial robot arm during a sequence of tubular motion between obstacles.

5.8 Vertical and horizontal composition of Tasks

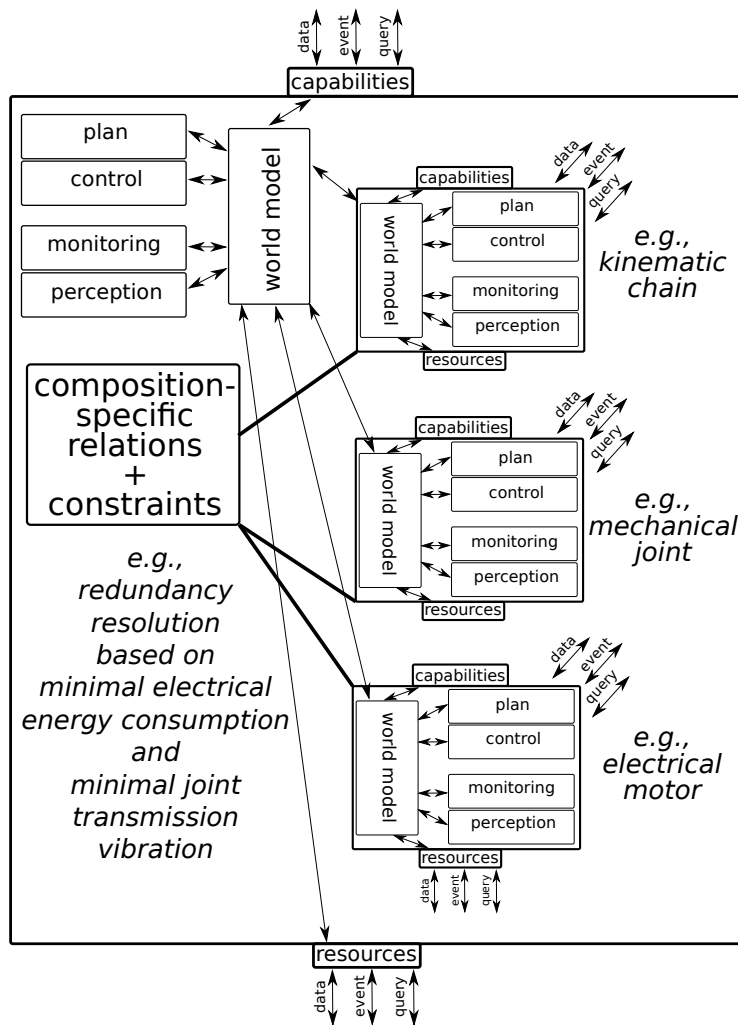


Figure 5.23: Composite task, interconnecting one or more world-models with one or more plan, control, monitoring and perception models, over multiple *levels of abstraction* of a robot's kinematic chain. The extra relations and constraints in the task *configure* some of the many "magic numbers" in the modules that exist in its scope, because they link the motion capabilities of the robots, the task expectations, and the constraints imposed by the resources allocated in the execution of the task.

Chapter 6

Meta models for world modelling and its integration in tasks

World models are an essential part of task models. . .

Chapter 7

Meta models for control of continuous time behaviour

Control is the part in a system’s **task model** that is responsible for realising the **continuous-domain**¹ behaviour of the system, in order to bring the “world” from its *actual* state to its *desired* state, as specified in the **plan**. This Section introduces the **mereo-topological** meta model of **control diagrams**, and some of the natural structures and “best practices” in the control domain.

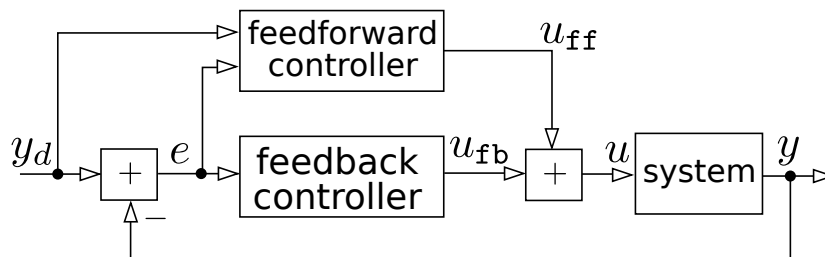


Figure 7.1: Feedback and feedforward control loops: simple version.

7.1 Mereology of control behaviour: feedback, feedforward, and adaptation

Figure 7.1 shows the (almost) simplest case of a control system, that is, with only feedback and feedforward contributions. Figure 7.2 shows the (almost) complete case of a control system, with the following contributions to the actuator input signal:

- **feedback**: this is a function that converts the desired and actual “state of the world” into an actuator signal to reduce that “error”. Feedback functions typically are parameterized, in order to be configurable for different applications and domains; e.g. to adapt the **gains** and **errors** to the control problem at hand.

¹“Space” or “effort” versus time, for example. **Discrete** control comes back in other Sections, like Sec. 2.10.

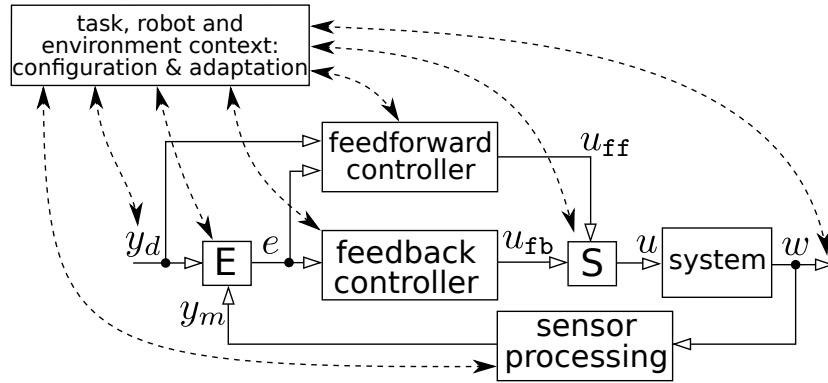


Figure 7.2: Feedback and feedforward control loops: extensive version. With the *setpoint error* as the main input to the controller computations. This is a special, “local” case of the generic controller of Fig. 2.10, since it considers only the *instantaneous error* between desired and actual state value y .

- **feedforward**: this is a (typically parameterized) function that generates an actuator signal based on the current “state of the world” and on a **model of how the system is expected to react** to actuator signals.

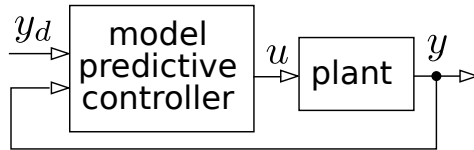


Figure 7.3: Predictive controller.

- **predictive**: the actuation signal is computed as the “best” value resulting from a simulation of how the plant behaviour is expected to be over a certain time horizon into the future, when the control inputs are varied over a specified domain, Fig. 7.3.
- **adaptation**: this is a (typically parameterized) function that converts the *observed history* of the control signal and plant state into an adaptation of one or more **parameters in the models** of the control functions, of feedback and/or feedforward. For example, it adapts the gain in the feedback control. So, it does not change the **structure** of the control behaviour models, only the **behaviour** itself.

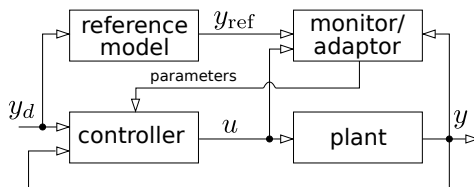


Figure 7.4: Model-reference adaptive controller.

- **preview**: this *does change the structure of the control* behaviour models, because it adds a model of (part of) the control in the **next** task in the sequence of task specifications, as long as that addition is not expected to compromise the ongoing control performance beyond a specified tolerance. For example, when moving a robot hand towards a door handle, one can already start controlling (from a certain distance

to the door) the opening of the hand as well as the orientation with respect to the door, such that the hand is already better aligned when it is going to have to open the door in the next sub-task.

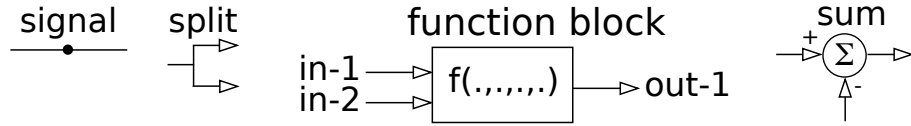


Figure 7.5: The four entities in the controller meta model that conform to the **algorithm** meta model: the **data** blocks **signal** and **split**, and the **function** blocks **function-block** and **sum** block.

7.2 Mechanism: state and system dynamics relations

Any controller (or **control diagram**) is the composition of only four entities (Fig. 7.5) that conform-to the **algorithm** meta meta model:

- **signal**: the **data** that flows between two **function** blocks; its numerical representation 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 **signals** are common to all controllers, so the meta model adds them as first-class modelling primitives:

- **setpoint**: the entry point of a controller, which is not connected at its input side, at least not inside the scope of the current control diagram. Its meaning in the context of an application is that this signal gives the *desired* value of the **plant** state.
- **measurement**: the other entry point of a **control-diagram**, which provides state information of the **plant**, possibly after some processing of the raw data from the sensors that are physically connected to the **plant**.
- **error**: the “difference” between **setpoint** and **measurement**, whose “magnitude” is a measure for the quality of the controller.²

²“Difference” and “magnitude” are written with quotation marks, since it is not always the case that the state space of the controlled system is a vector space (where “subtraction” is well defined) and/or has an invariant metric (via which “magnitude” is well defined).

- **actuation**: the exit point of a **control-diagram**, which provides information for the actuators that can change the **state** of the **plant**.
- **state**: a selection of signals somewhere in a control diagram, that make sense to the application that used the controller, when their values are taken together, at the same execution instant of the control diagram.
- **sample-instant**: the **signal** that represents the time of one particular execution of the **control-diagram**.
- **sample-period** (and its inverse, the **sample-frequency**): the **signal** that represents time between two subsequent executions of the **control-diagram**.
- **event**: a **signal** that represents the fact that “something” has happened during the execution of the **control-diagram**.

The following **function** blocks are so common that the meta model adds them as modelling primitives too:

- **plant**: 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** that convert physical values into digital **signals**.
- **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**.
- **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 **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 **function** block that takes a **state** of the **control-diagram** as input, and computes an **event** for the application software that uses the controller. The event itself is not further used in the **control-diagram** itself, but it can give rise to a change in one or more parts in the **control-diagram**.
- **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 **control-diagram** is the model of the function that takes the **output state** of the **plant** as one of its arguments, and the *desired state* of the **plant** as its **setpoint** argument, and computes the

input that should be applied to the **plant** to make it evolve towards the desired **state**. In general, a **control-diagram** model is a **cyclic graph**, because of the presence of feedback loops.

- delay, or buffer:

In order “to run” a **control-diagram**, all its function blocks must be serialized into an **execution spanning tree**, or unrolled control loop. Extra constraint in case of cascaded loops:...

Properties of the execution are:

- latency:
- jitter:
- loop-time:

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.

7.3 Policy: model-reference adaptive control (MRAC)

One particular **policy** of adaptive control has been given the name **Model Reference Adaptive Control** (MRAC); its computation uses a model of the desired plant behaviour.

7.4 Policy: model-predictive control (MPC)

One particular **policy** of predictive control has been given the name **Model Predictive Control** (MPC); its computation uses a cost function that includes samples along the *full* predicted trajectory.

7.5 Mechanism: setpoint, trajectory, path and tube control inputs

A second ordering in control approaches is according to the type of the **input** that the controller is expected to accept:

- **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 an instantaneous value of the **plant state**, a trajectory of desired **plant state** values at multiple sample times over a certain horizon is used in the control computations. This mechanism brings more *constraints* than one single setpoint; the latter is a boundary case of the former.
- **path**: another mechanism is to use a **path** instead of a trajectory, which is less constraining since the **time is not imposed**, i.e., the **state** is *constrained* to follow the geometry of the path in state space, but not any timing along that path.
- **tube**: this is an even less constrained control specification, in that the controller is expected to keep the plant state inside a “tube”, or “region”, in the **state** space, and no extra constraints are attached to the concrete trajectory that the controller generates within the tube boundaries.

The design choice about what is the *input* to a controller defines, implicitly, also the meaning of the term **(control) error**.

7.6 Policy: control progress objective

For setpoint and trajectory control, the *objective* of the controller is the same: to reduce the **error** to zero. But *path* and *tube* constraints are not complete enough specification to determine the behaviour of the controller. That behaviour also depends on the **progress objective** that is specified for the controller. There are an infinite number of ways in which such a progress objective can be chosen, so this becomes a *policy* decision.

7.7 Policy: PID, sliding mode, gain scheduling and ABAG

A third ordering in control approaches considers the *choice* of how the “error” is taken into account in the **feedback/feedforward** parts, without using any specific knowledge about the dynamics of the controlled system:

- only a **setpoint** error is used in the **feedback**, with **PID** control as typical representative.
- an **error area** is used to **select** the control **feedforward**, with **sliding mode** as typical representative. Similarly, an area in the **configuration space** of the controlled system is used to select the **feedback** part, as in **gain scheduling**.

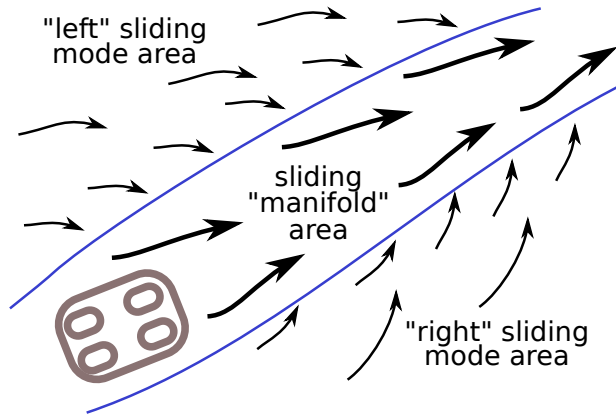


Figure 7.6: The concept of *sliding mode control*: the state space of the robot is divided into areas, each with a particular *a priori* determined control approach. In the strictest version of the concept, the middle area is the specified *trajectory*.

The control designer determines, at design time, the areas in which the plant error can be, and for which a different feedback-feedforward control action is prepared off-line, Fig. 7.6.

- the **trend** in the error is used to **adapt** the control **feedforward**, with **ABAG** control [30] as a representative.

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 *feedback* part of the ABAG control algorithm is in the fact that it reacts to the error, and the *feedforward* part comes from the fact that the **waveform** of the *bias* is determined beforehand; in addition, both *bias* and *gain* are **limited** to chosen maximal values, *before* they act on the control signal and not afterwards, as in the case of the **anti-windup policy** in a PID controller.

The *waveform* of a PID controller is not known in advance since it is completely determined by the error, while the waveforms are determined by the control designer for the ABAG and sliding mode cases. Hence, the latter have more predictable behaviour (and hence **stability**).

The “I” in a PID controller, and the “B” in an ABAG controller have a similar intention: to compute the signal that the controlled plant needs for a **steady state** evolution. For example, the force to compensate for gravity, or for friction. The contribution from the integral term is proportional to both the *magnitude* of the error and the *duration* of the error, which is a lot more difficult to predict than the adaptation behaviour of the fixed-waveform “B” term. Not in the least since the building-up and reduction of the I term depends on the concrete error signal, and not on decisions introduced by the control designer.

The traditional policy for a PID controller is to be used as *one single algorithm*, irrespective of the error. The traditional policy for a sliding mode controller is to be used as a *hybrid algorithm*, in the sense that it identifies different areas in the error state space and selects an algorithm on that basis. The traditional policy in the ABAG controller is to allow *adaptation* of its B and G parameters. Of course, there is no fundamental reason why these different

policies can not be used all together. In the context of control for system-of-systems, the trend is obviously even stronger, since practice shows that **every controller must be hybrid and adaptive**:

- *hybrid*: the computation of the control action requires different modes (or states, or regimes,...) because (i) the *output* must be limited (no actuator has infinite effort resources, no application tolerates infinite control inputs, etc.), and (ii) the *inputs* must be thresholded (no sensor has the same accuracy over the whole dynamic range of the plant, so some “too high” or “too low” values must be discarded).

Of course, many particular *tasks* will introduce extra reasons to limit or threshold control signals.

- *adaptive*: no plant has ideal linear system dynamics behaviour, so control designers will introduce different dynamical regions for the plant, and design controllers for each of them separately. (The choice of which regions to identify and select is often not a property of the plant itself, but rather a design trade-off between task requirements and resource capabilities.) The simplest form of adaptation is to select different parameters for the same control algorithm, in the different plant dynamics regions.

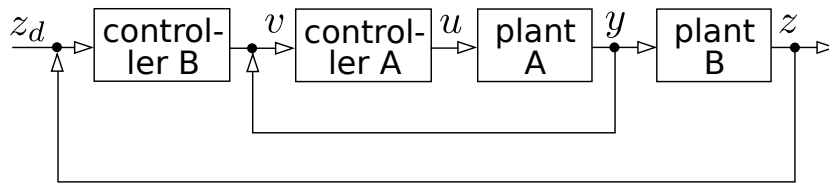


Figure 7.7: Cascaded control loops.

7.8 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. 7.7: 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.)

7.9 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 **fieldbusses** 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*.

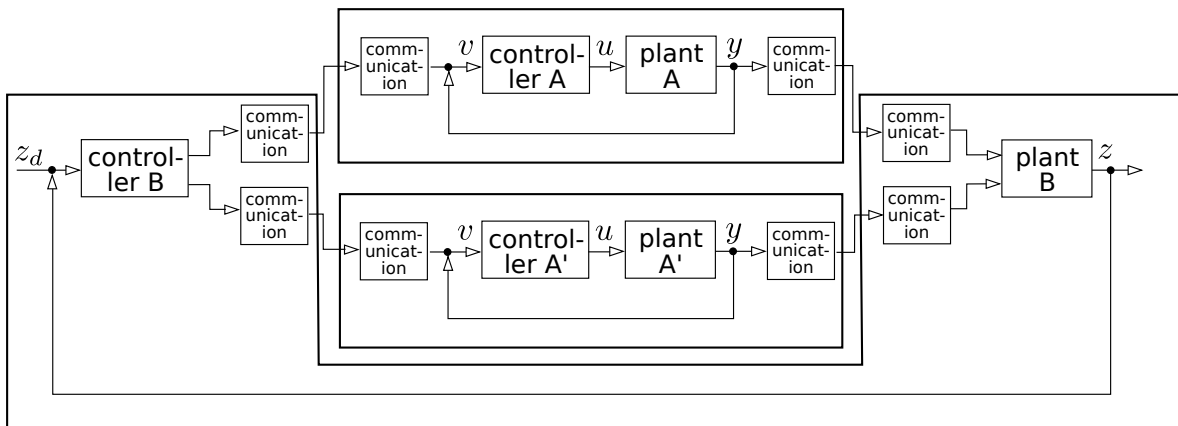


Figure 7.8: Many modern systems must rely on *communication* between sub-systems, in order to realise cascaded control loops.

So, such control loops involve *communication* between control computing processes, Fig. 7.8. 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.

7.10 Policy: optimal control

The control mechanism in Sec. 7.5, setpoint, trajectory, path and tube control, still allows multiply ways of *how* the control input is realised. **Constrained optimization** has become a popular approach, because there are almost always contextual pieces of information that can help to formulate the input as the optimum of some objective function and a set of constraints. Here is the generic version of a formalisation of a constrained optimization problem from which a control setpoint could be computed:

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

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.)

7.11 Policy: 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.

7.12 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. 7.9) composes the **task**, **algorithm**, **Finite State Machine**, control diagram, and **event loop** meta models, and adds specific **policies** (i.e., model configurations):

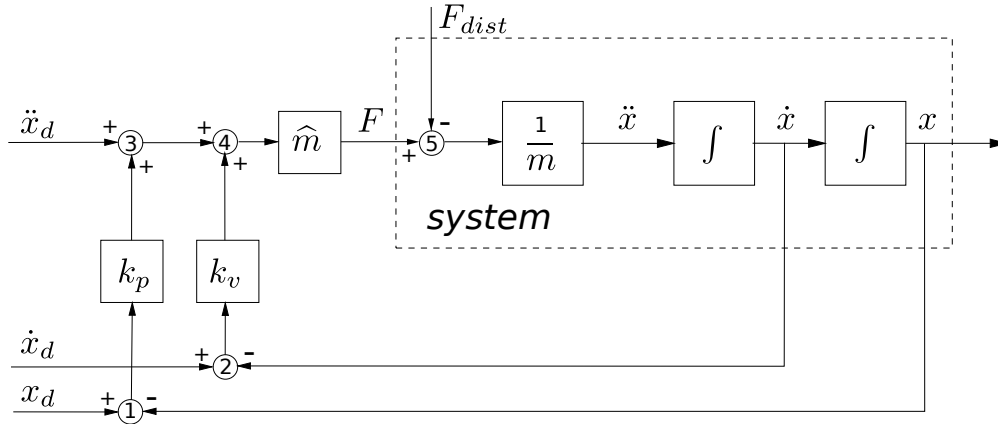


Figure 7.9: A one-dimensional position controller, generating the actuating force F from the desired position x_d , velocity \dot{x}_d and acceleration \ddot{x}_d , via nested velocity and position feedback loops with gains k_v and k_p . The “world model” of the controller consists of an estimate \hat{m} of the moved mass.

- control diagrams as in Fig. 7.9 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 “summaters” 1, ..., 5 in Fig. 7.9), 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. 7.9): 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, **realtime** requirements of the application have an impact on the model of the event loop.

7.12.1 Example: one-dimensional position control

For example, the event loop of the one-dimensional position controller depicted in Fig. 7.9 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 busses](#), 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.)

7.12.2 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.

7.12.3 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
}
```

7.12.4 Policy: realtime activities via the “multi-thread” software pattern

Many robotics and cyber-physical systems contain one or more activities whose execution must be **predictable** (“**deterministic**”, “**realtime**”) 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 realtime applications must configure the interrupt capabilities appropriately. The common configuration options are: to inhibit (“mask”) some interrupts before a realtime activity is launched, to inhibit interrupts on the set of cores that share cache memories with the realtime core, or to assign only the realtime relevant interrupts to the CPU core on which the realtime activity is running.

The **good practice** solution, Fig. 7.10, 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 **realtime** 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 realtime 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 realtime thread with the dataflow it needs, (ii) getting the realtime 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).)

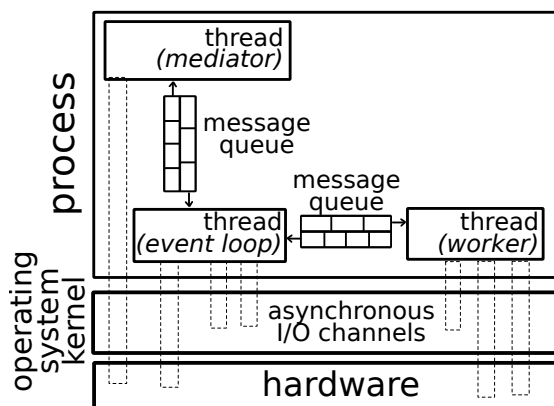


Figure 7.10: Multi-threaded process architecture for the concurrent and parallel execution of activities. The “message queues depicted in the figure represent any type of fast and local inter-thread communication; e.g., lockfree buffers, circular buffers, etc.

The realtime 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 sometimes called the **hard** realtime thread, and the other ones the **soft** realtime thread. These adjectives have no absolute meaning, and the major **best practice** design requirement is that there can be **only one hard realtime 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 realtime 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 any dependency on the software of an operating system.

7.12.5 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. 2.8 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. 2.8 will most probably get 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 model and software 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 **Bayesian information theory** as the 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.

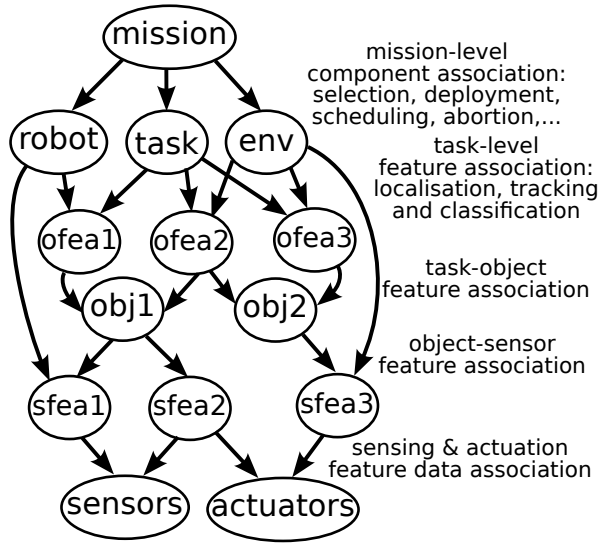


Figure 8.1: The meta model of the perception stack. The semantics of the graphical notation is that of “Bayesian networks”, hence (i) the arrow does *not* indicate information flow but rather the causality order of all arguments in the conditional probability represented by a connection, and (ii) n-ary relations are not modelled explicitly by the structural properties alone.

8.1 Mereotopological meta model: the natural hierarchy in robotic perception

The perception stack model has *structural* parts and *behavioural* parts. The structural part uses *hypergraphs* 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.

8.2 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.

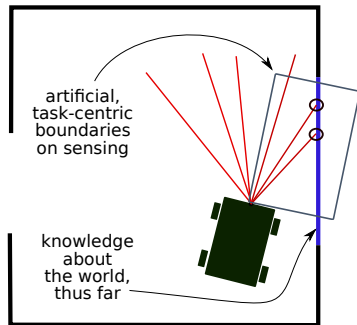


Figure 8.2: Application example of the *escape from the room* task (Sec. 5.6.2) to illustrate various levels in the perception stack hierarchy of Fig. 8.1.

8.3 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.

8.3.1 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 Figure 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.

8.3.2 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.)

8.3.3 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.)

8.4 Mechanism: Bayesian information theory

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!

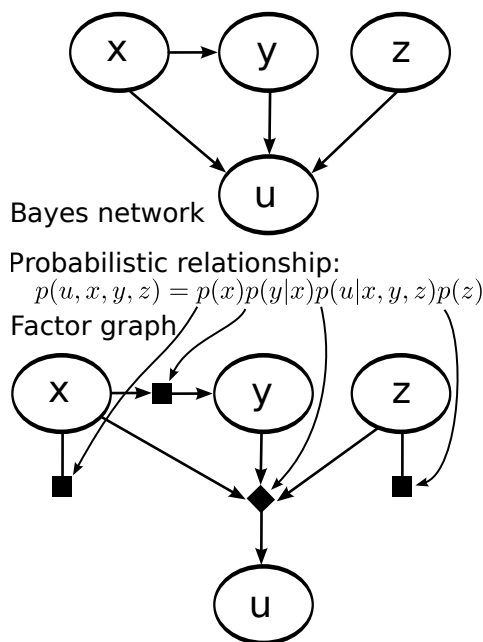


Figure 8.3: Example of a probabilistic model, in Bayesian network form (top), as a factored conditional probability density function (middle), and as a factor graph (bottom).

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](#).
- same *real-world* system can have various graphical *models*.

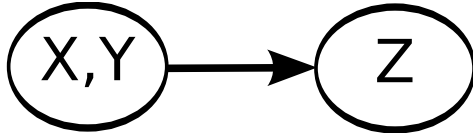


Figure 8.4: Simplest Bayesian network.

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*!

Mean μ , Covariance \mathbf{P} :

$$\mu = \int x p(x) dx, \quad \mathbf{P} = \int (x - \mu)(x - \mu)^T p(x) dx$$

(μ is vector) (\mathbf{P} is matrix)

Advantages of Gaussian PDF representations:

- only two parameters needed (per dimension of the domain).
- information processing is (often) analytically possible.

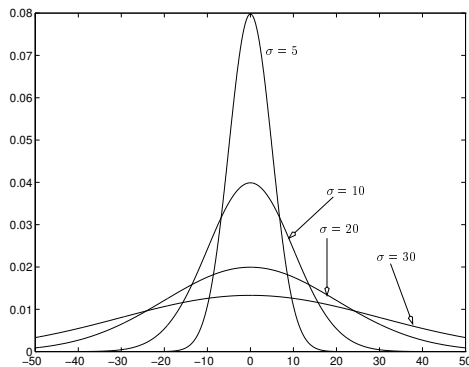


Figure 8.5: Simplest PDF: Gaussian (or **normal**) PDF, with **mean** $\mu = 0$ and **variance** $\sigma = 5, 10, 20, 30$.

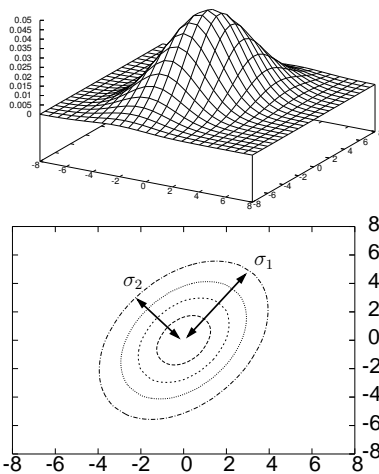


Figure 8.6: 2D Gaussian.

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 Geometrical semantics in perception

8.6 Dynamical semantics in perception

8.7 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.

8.8 Mechanism of information update: Bayes' rule

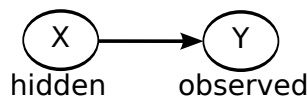
The essential role of Bayes' rule: "*Inverse probability*":

$$p(x \text{ and } y|H) = p(y \text{ and } x|H)$$

(product rule) \Downarrow (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:

$$\boxed{\begin{aligned} p(\text{Model params}|\text{Data}, H) \\ = \frac{p(\text{Data}|\text{Model params}, H)}{p(\text{Data}|H)} p(\text{Model params}|H). \end{aligned}}$$

$$\text{"Posterior"} = \underbrace{\frac{\text{Conditional data likelihood}}{\text{Data Likelihood}}}_{\text{"Likelihood"}} \times \text{Prior.}"$$

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.

8.9 Mechanism of perception solver: message passing over junction trees

The *message passing algorithm in factor graphs* [10] plays a similar role in perception as the *hybrid dynamics solver* of Sec. 4.6 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. 4.6), 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.)

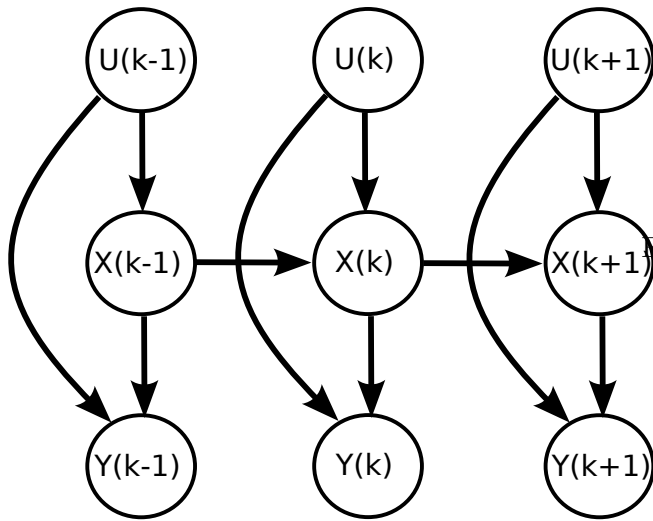


Figure 8.7: Dynamic Bayesian network.

8.10 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”

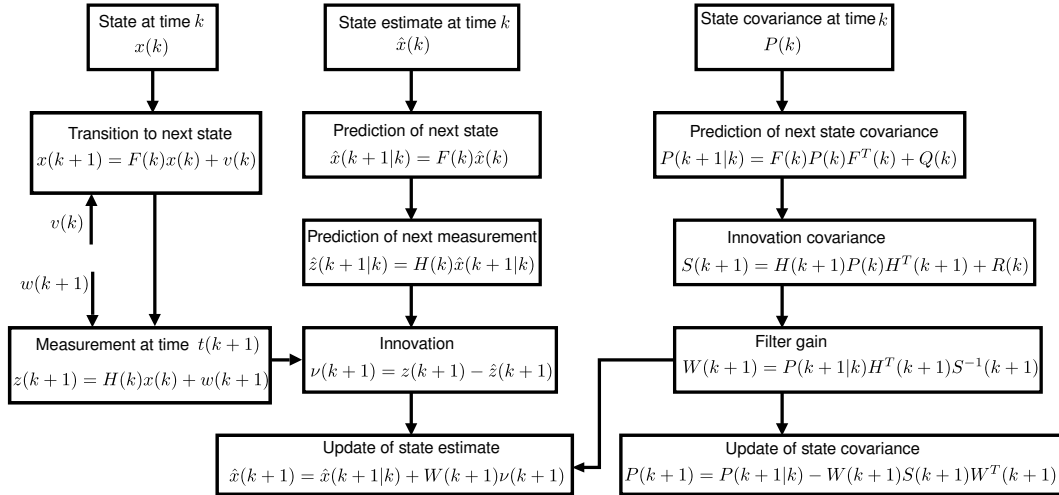


Figure 8.8: Computational schema of the Kalman Filter.

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.

8.11 Policy: Factor Graphs

8.12 Mechanism: composition of point and region features

8.13 Policy: feature pre-processing

8.14 Policy: deployment in event loop

Chapter 9

Meta models for holonic, resilient & explainable system architectures

Designers of cyber-physical systems create **information architectures** of **activities**, and **hardware architectures** of **devices**, and interconnect both with a **software architecture** of **components**, in which all of the above engage in **peer-to-peer** interactions of different kinds.

A **resilient**¹ system has a top-level software architecture that consists of only so-called **holons**, (or *agents*, or *digital twins*), that is, components/sub-systems that (i) remain operational under *any* (lack of) interaction with peer systems (that can come and go dynamically), (ii) can always decide for themselves when and how to switch to what kind of **graceful degradation** behavioural state(s), while (iii) **explaining** their decisions to whatever **peer** holon that is interested in knowing because, (iv) it needs to build up **trust** in their mutual interactions.

In addition to this **holarchic** peer-to-peer behaviour, **performance**, **robustness** and **correctness** of software components (at all levels of composition) are mainstream (but imprecise) metrics to evaluate the quality of a system.

It is (probably) useless to try and be precise about “the definition” of a system, because what counts are the *design patterns* and *best practices* that are available to develop, deploy and maintain systems and their compositions. Seminal work in **holonic** system design started in the 1960s and matured around the beginning of this century [40, 47, 62, 76, 80, 78]. The starting point in holonic design is to design a system in such a way that it is ready to be composed into a larger **system-of-system**, sooner or later, *and* that it can **decide for itself** *that, when, why, where* and *how* it becomes part of a larger system, or break away from it.

The major responsibilities of the system architects (compared to component developers) are:

- to provide a design that can predict **system-level Quality of Service** for those specific requirements that only show up at the system level: **autonomy**, **safety**, **security**, or **resilience**.

¹This document uses the terms **stable** or **robust** as synonyms for *resilient*.

- to realise resilience against **software erosion**: this is not a physical phenomenon, because the software does not actually decay, but rather a social phenomenon. Indeed, software projects can suffer from a lack of developers to remain responsive to the changing environment in which the software must work.
- to have an *information architecture* that **does not depend on specific choices** made in the *software and hardware architectures* that happened to be the one available the first time the information architecture was implemented. In particular, the *configuration values* should be changeable for each actual runtime that is deployed from the software architecture.

Individual components can never *guarantee* these requirements, but can easily *undermine* them by not being able to adapt their own behaviour to the overall behaviour expected at the system level.

The system design process has major phases: (i) to identify which **tasks** must be realised by which *activities*, (ii) to identify which *resources* are **owned** by which of these activities, (iii) to define how the *state* of these resources can be *shared* with other *activities*, (iv) to create the software *event loops* to coordinate the execution of the algorithms that provide the *behaviour* of activities and their interactions, and (v) to map these event loops onto an appropriate number of *threads* to exploit the performance of the cores and the networks provided by the hardware architecture.

This document designs a holon around one or more instances of the **mediator pattern**, because that allows to guarantee **single point of decision making** over each of the “resources” the holon is responsible for. Each holon is able to base that decision making on the formal reasoning it can perform with the **knowledge relations** it has about its **internal** behaviour and about how its **externally** visible behaviour interacts with other holons. These are (necessary but not sufficient) conditions to realise the system’s *predictability*² under interactions with any type of **external** system.

9.1 Robustness, resilience and explainability as system design drivers

The literature has introduced the concepts of *resilience* and *holarchy* [47, 76] to explain *natural* and *social* systems, and compositions of them into system-of-systems; 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.*

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

²But not necessarily its *performance* or *robustness*.

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.

(TODO: robustness: explainable reaction to known disturbances; resilience: explainable reaction to unknown disturbances, or rather, to disturbances that one can *detect* but not *classify*.)

9.2 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](#).)

9.2.1 Information architecture

The [software architecture](#) design often gets the lion share of the system developers’ attention, but the *step change* that this document wants to realise is to give that central role to what it calls the **information architecture**. (This Section gives an overview, and Chap. 10 provides a more in-depth discussion.) This shift in focus strengthens the role in engineering systems of (i) formally represented *knowledge*, and (ii) *information depending* on knowledge as *the* method to provide *meaning* to *data*. An information architecture consists, in general, of [models](#) for (i) **entities** to include in the system, (ii) **relations** between entities that must be taken into account, (iii) **constraints** on such relations to be satisfied, and, especially, (iv) **policies**, or trade-off choices, to be made every time a new coupling is introduced in the system. The research hypothesis behind this approach is that such a formalized network of semantically connected models is the best possible *specification* for the design of the software and hardware architectures that must *realise* the system. The following types of information models are introduced, as well as the couplings between them:

- **domain models:** the property graphs of the various pieces of domain knowledge the application builds upon; for example, the relevant physical units and mathematical solvers; the rules of traffic; etc.
- **application models:** the property graphs representing the knowledge needed about the application: task requirements, social and technical constraints, Quality of Service, introspection and self-diagnosis models, etc.
- **provenance models:** information about how data, entities and relations have been created, who has *ownership* of them and has the responsibility to make decisions about what [to do](#) with them, including providing them to other “organisations”, etc. Two established meta models for provenance are [Dublin Core](#) metadata, and the [PROV](#) models from the [World Wide Web Consortium](#).

- **abstract data types:** from all of the above, select those entities and relations that the application has to create abstract data types for. This includes models for the data structures, communication messages, operators, etc.
- **activities**, with [event loops](#) to serve all [5C](#) responsibilities, including managing [concurrency](#) of algorithms such that state changes in abstract data types remain consistent.
- **mediators:** each *shared resource* (communication channel, task, space, algorithm, world model, CPU, robot hardware,...) needs a mediator activity to centralize the configuration and coordination decision making about that resource. In some contexts, also all [CRUD](#) operations (Create, Read, Update, Delete) on the shared resource are run through the mediator.
- **solvers:** the algorithms that will realise all the activities behaviour need to be modeled.
- **contexts:** a major design effort in making an information architecture is to make sure that every part in it (activity, solver, function,...) is always active within the correct *context*, that is, the “state” of the system that it is not responsible for itself but relies on to be correctly filled in by other parts of the system.
- **Life Cycle State Machines:** each activity needs its own LCSM to manage the resources it uses, and each task and each event loop need a separate LCSM too. One of the more difficult design challenges is to make the life cycle of the *contexts* explicit for the whole system, and for the whole duration of the system’s life.

The result of all these interconnected models is a property graph in itself, representing the dependencies between all of the above. This document’s [working hypothesis](#) is that multiple levels of abstraction, high coupling, or complex multi-disciplinarity can not be avoided, but should be dealt with head-on. The way to make this happen is (i) by making knowledge relations explicit, (ii) available for runtime reasoning in the robot’s control software, and (iii) with identified natural [causality](#) constraints between relations. These steps result in an *information architecture* that gives structure to the dependency complexity, and hence also to the reasoning needed to let the control software explain the robot’s behaviour in the context of the tasks assigned to it, the capabilities it is expected to provide to others, and the resources it relies on.

9.2.2 Software architecture

An [information architecture](#)’s property graph adds constraints on the data and control flows in any software architecture that *implements* and *deploys* the processes that realise the activities represented in the information architecture. The software architects must make decisions about the following implementation aspects:

- **data structures:**
- **functions:**
- **schedules:**
- **threads:**

- **processes:** provide shared memory to threads and system calls to functions. This infrastructure touches the operating system, hence must be configured at that level. For example, control groups in Linux.
- **device and process interrupts:**
- **communications:**

For many of those, *software patterns* exist. This Section gives an overview, and Chap. 11 provides a more in-depth discussion.

9.2.3 Hardware architecture

The hardware architects' role in this document is focused on how to design the information and software models of all hardware that the system must bring under computer control; that is, *deploying* software onto hardware, taking the decisions about how to execute the software architecture on computational hardware, communication busses, I/O to peripherals, and storage mediums.

(TODO: patterns for device drivers, with special attention to *interrupt handlers*, and how to realise the *LCSM* and *Traffic Light* policies with them.)

9.2.4 Digital platform

This Section is about *exploiting* the software, that is, taking the decision about which **digital interaction standards** to use to let a well-identified set of **end-users** work with the software in ways that fit “naturally” to the traditions and semantics of their application domain. A software/hardware combo deserves the name “platform” only when it allows 100% *multi-vendor interoperability* via 100% *open standards* (for data, events and models). Typically, a platform is a software architecture that comes with a lot of tools and standard formats to make working with the software “easier”.

This document is built on the hypothesis that **successful meta modelling is the key to platform success**. So, it is mainly concerned about the information architectural aspects, and tries to be complete in it, for the domain of robotics. The structuring relations that create most added value to a platform are: containment hierarchies, mediators, event loops, Coordination and Configuration; hence, these are also the parts of knowledge for which it makes most sense to introduce intellectual property protections. Fortunately, the amount of such knowledge models is (often deceptively) small, *and* the introduced composition models are optimized to keep these parts easily separated from the “mainstream” parts.

(TODO: add explanations of platform services such as *provisioning*, *configuration management* and monitoring of physical and virtual servers.)

9.2.5 Meta models for robotic stacks

In the more specific context of robotic systems, the modelling efforts described in this document result, together, in a **large set of single-focus models**, structured in so-called “stacks” for robotic systems. A stack is an architectural structure of functionalities, starting at the

“bottom”³ with sensors, motors and mechanics, builds up towards instrumented and actuated kinematic chains, and further to sensor-based tasks executed by loosely and opportunistically coupled multi-robot systems-of-systems. However, there are also “horizontal” directions to these stacks, because some (meta) meta models are relevant for entities and relations at more than one level of the “vertical” stack direction. And **geometry** is one of the most prominent such “horizontal” meta meta models, together with most other branches of mathematics (analysis, logic, numerical linear algebra, etc.), and with *software patterns*.

The **most mature** stack (and also the most “pure robotics” one) is the “(motion) control stack”, because it has the least amount of *open world* assumptions inside: the robot *is* the world, for the largest part of the stack, and electro-mechanics make up the majority of the meta models for computer-controlled motion of mechanical devices. The good news is that, by now, it is indeed well known how to make kinematic chains move themselves (and the sensors and tools connected to them) with prescribed behaviour, relying on the physically limited variety in motion control modes (current, torque, impedance, velocity, force and position) that can cover the task needs (Chap. 5) of all mainstream applications. The bad news is that, even after 50 years, there is still no standardization in place that covers most of the motion stack; this lack of standardization starts already with the representation of geometry.

The other “stacks” that are relevant for the domain of robotics are the ones it shares with other domains: perception stack, world modelling stack, task specification stack. The robotic motion stack already integrates parts of those stacks: even for its own motion, a robot needs some non-trivial amounts of perception, world modelling and task specification. This interdependency is a clear indication that the different stacks should not be developed in isolation, and that their **composability** is a primary design and development concern.

The mereo-topological overview of the modelling of “stacks” (Fig. 9.1) is a set of *loosely coupled* models, where the driving focus of the loose coupling is in the models’ **reusability** (or composability, flexibility, or freedom of choice); its **usability** (or user friendliness, or freedom from choice) can then be realised by adding *tools* and *domain-specific languages* that target specific developers, users and/or application domains.

The **kinematic chain** is the central entity within the motion stack picture, and, at the highest level of abstraction, the entities, relations and constraints connected to kinematic chains are: (*rigid body*) *links* whose relative motions are constrained by *joints*, driven by *actuators* and measured by (*proprio*) *sensors*; behaviourally speaking, a kinematic chain is an instantaneous *mechanical* energy transformer between joint space and Cartesian space, and that relation can be *redundant*, *underactuated* and/or *singular*, all at the same time even.

Developers of robotic application software have to add concrete implementations to the just-mentioned concepts, and the Figure structures the dependencies in the various models that are involved: mathematics of geometry and dynamics; coordinate and digital representations; and physical units. These are indispensable *data structures* whose semantics must be made 100% clear (and **eventually standardized**) for all developers and users of the *functions* and *solvers* that implement the behavioural properties of kinematic chains.

9.2.6 “Sense-Plan-Act”, “Three-Level”, “Subsumption”

Three-level [3]: Planning and Executive (or Decisional), and Functional (or Control).

³The bottom is a relative concept: sensors, mechanics and actuators are only the bottom in the rigid body **abstraction scope** of this document, but not when opens the scope to electronics, electromagnetic and thermal fields, etc.

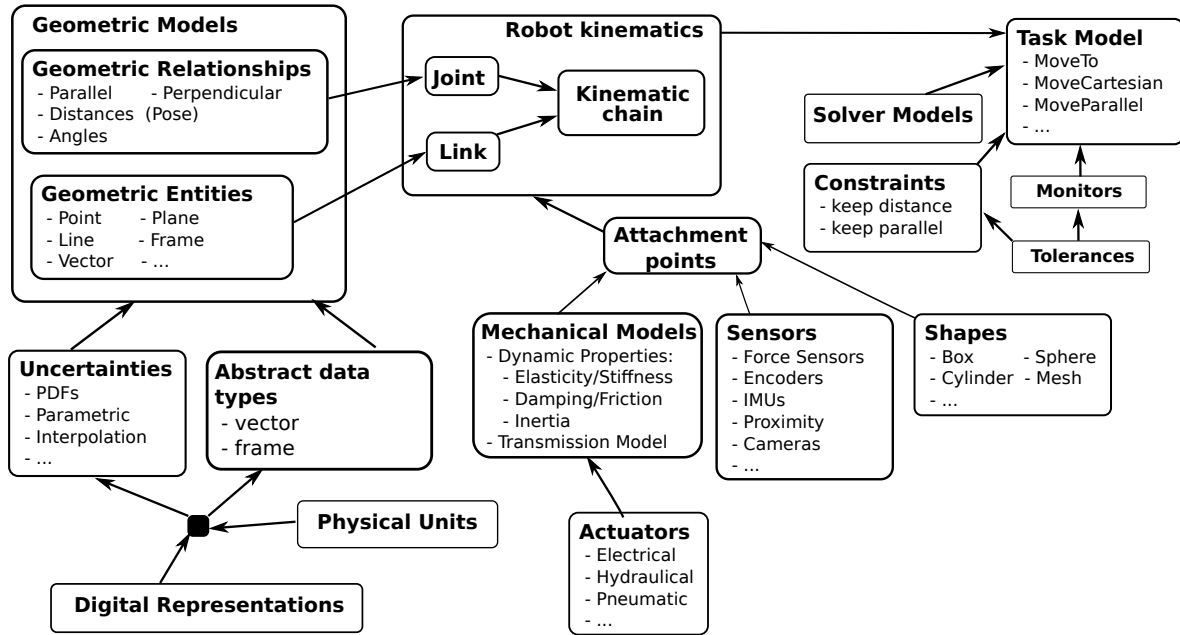


Figure 9.1: The mereo-topological overview of the “motion stack” in which a majority of the depicted blocks makes use of geometrical entities and relations. An arrow represents *composition* of complementary aspects of the system, represented in separated *models*, and eventually implemented in separated *software components*; the “black square” arrow represents an *n-ary* composition dependency.

9.2.7 Role of middleware

Three-level: Planning and Executive (or Decisional), and Functional (or Control).

9.3 Autonomy and 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.

9.3.1 Sheridan’s ten levels of system autonomy

One of the most popular definitions is Sheridan’s **10 levels of autonomy** [58], Table 9.1.

9.3.2 Explanation levels for autonomous decision making

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. The granularity of the levels is designed to allow

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 [58].

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.

9.3.3 Links with horizontal and vertical integration

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 (Sec. 2.7.9). 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.

(TODO: examples.)

9.4 Safety

(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**”.)

9.4.1 Best practice: safety PLC

9.5 Security

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

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 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 [54].

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 is the design layer between (i) the application (which provides (models of) the task requirements), and (ii) the [software](#) and [hardware](#) architectures (which provide the digital and material platforms to realise the information architecture). The responsibility of the information architecture is to provide semantic [completeness](#) and [correctness](#) of the whole application.

The information architects use this Chapter's structured ways **to compose** a system's building blocks into an **information architecture**. In other words, they [design the models](#) about **how** the behaviour and interaction of activities and streams realise the system's **tasks** with the available **resources**. This document advocates the following order in the design process:

1. to decide which **abstract data types** (**world model**, **plan**, **control**, **perception** and **monitoring**) to create and to share.
2. to decide which **solvers** to create (to let **functions** operate on the abstract data types) and to assign to which **event loops**.
3. to decide which event loops to assign to which **behavioural states** in which **activities**, and which **events** cause the behavioural **state switches**.
4. to decide which **streams** to create and to connect to which solvers and which activities.

As a cross-cutting responsibility to all above-mentioned design decisions, extra decisions are made about:

- **ownership** and **mediation** of the information: which task owns (which instantiations of) which information in which model, and which activity owns resources, and mediates them.
- **loose coupling** of data, solvers and activities.
- **meaning** of the information: what documentation, offline tooling and online activities must be introduced to guarantee that information is interpreted in the unambiguously correct way, by human developers as well as by activities everywhere in the system.

All of the above-mentioned design decisions introduce complex *n-ary dependency relations*. So, information architecture models are **property graphs** in themselves.

The **task** is the **causal** trigger of any system design:¹ the task provides the **purpose**, or **intention**, of the system behaviour, and almost comes already with a (possibly only partial or abstract) **plan** of *how* to realise that behaviour. The **activity** and the **producer-consumer stream** are the core building blocks for the system architect to realise tasks. Stream-based **interactions** between activities takes place via so-called **CRUD** operations (Create, Read, Update, Delete) on the **abstract data types** and **data structures** that represent models, events and data. Good system architects (at the levels of information, software *and* hardware) design **coordination protocols** on these CRUD operations that are robust against an identified set of disturbances to the system behaviour.

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 span several levels of abstraction and require many scientific and engineering disciplines to be integrated.
- most software projects start *bottom-up*, around some algorithms and middleware that “works”, and hence result in the proverbial “**law of the instrument**”.

This Chapter tries to remedy this situation, and introduces best practices and compositional patterns to help system developers to cope with the resulting intricate dependencies between activities. Figures 10.1–10.2 sketch some **vertical and horizontal** task integration challenges that must be tackled, for the typical application use case of a two-wheel driven mobile platform, Fig. 10.3.

An information architecture is still independent of concrete choices of programming languages, communication middleware, operating systems, communication protocols, or development tools. Chater. 11 adds extra software-centric models and design structures to the information architectures described in this Chapter.

10.1 Running example: two-wheel driven mobile robot

Even the traditional and simple use case of a mobile robot with two actuated wheels, Fig. 10.3, brings a large system design complexity, as soon as one needs to consider all control and system-level performance aspects, from the battery and actuation energy management, to the long-term resilient operability as a “Mobility as a Service” platform.

This Section describes the constraints that the hardware imposes on the control system (Sec. 10.1.1), and some representative task requirement descriptions (Sec. 10.1.2).

¹Often, hardware architecture decisions bring in hard causalities too.

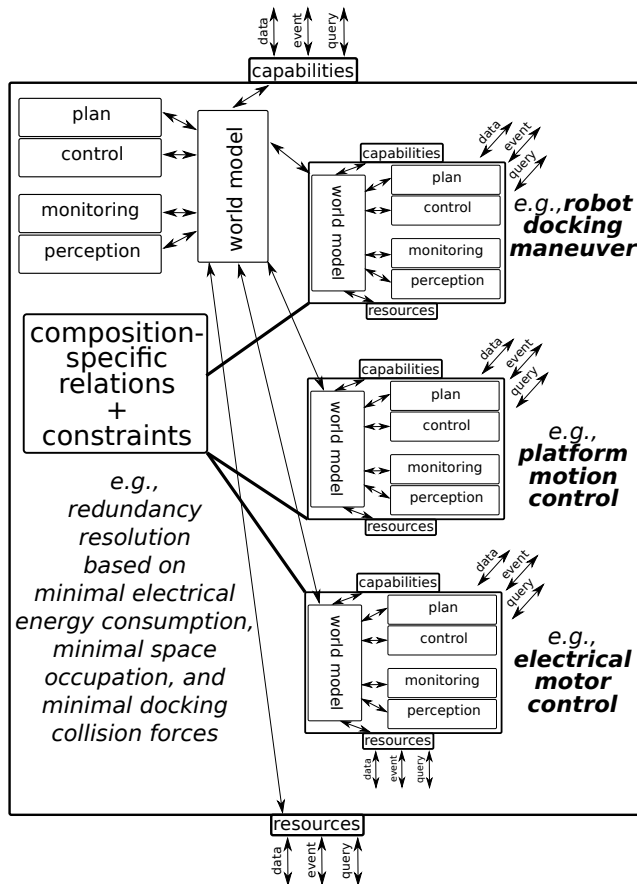


Figure 10.1: “Vertical” integration of various levels of task control in a mobile robot. The tasks in neighbouring **levels** influence each other’s behaviour via relations (constraints to satisfy, and objective functions to maximize) that connect task meta-model parameters from both levels. (Hence,) the coupling takes place via the *world model*.

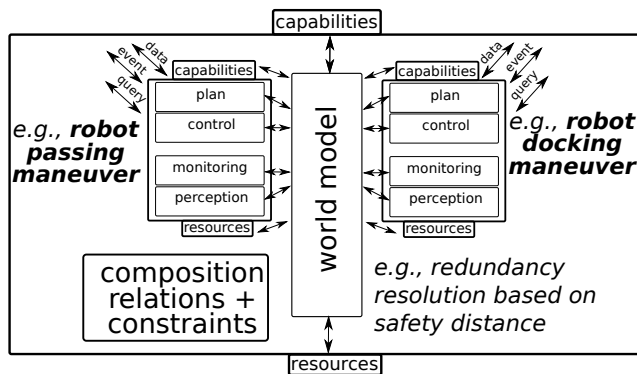


Figure 10.2: “Horizontal” integration at one level of task control in a mobile robot. Multiple **tasks** influence each other’s execution via relations that connect parameters from both task models. One major coupling mechanism is via the *world model*.

10.1.1 Hardware architecture

As the other essential causal input to the information architecture design, the following rather common hardware architecture is assumed to be present in the mobile platform:

- the motors are **brushless DC** (BLDC) electrical motors, using **field-oriented control** (FOC).
- that control is embedded in a stand-alone **servo drives**.
- those drives are interfaced to an **industrial PC**.

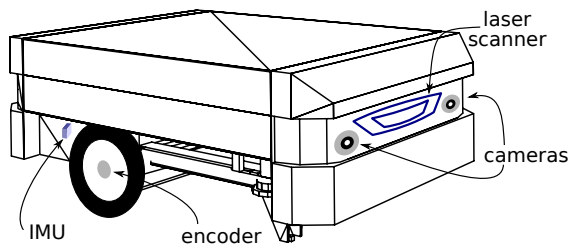


Figure 10.3: A two-wheel driven mobile robot, with a 2D laser scanner and two cameras at its front; internally, it has encoders on its wheel motors, and an *Inertial Measurement Unit* on its body. This is a popular hardware architecture, with the hardware constraint that the robot can not move in all directions at each moment in time.

- that interfacing uses a realtime communication platform, such as a [CAN](#) bus, or an protocol, e.g., [EtherCAT](#).
- the robot has [rotary encoders](#) on its motors, an [inertial measurement unit](#) (IMU) on its body, and a [2D laser scanner](#) and two [colour cameras](#) embedded in its front [bumper](#).
- the industrial PC runs a [real-time](#) version of the [Linux kernel](#).
- it has sufficient RAM and CPU cores to run all required software on-board.
- one of its processes is a [Node.js](#) server that is responsible for
 - all networking with computers elsewhere in the local network; for example, to do the task scheduling for all robots.
 - a [web browser](#)-based [graphical user interface](#) (GUI).

10.1.2 Typical tasks

Section 5.1 introduced several task examples for robots like the 2WD mobile platform, including (a sketch of) a formal task specification language for “guarded motions”, but without explanations of the control architecture. This Chapter adds the “best practice” information architectures for the 2WD use case.

10.2 Pattern: task control at three levels of abstraction

This Section gives a list of (types of) **tasks** to be performed by two-wheel robots, each “owning” one of *the three particularly relevant* levels of abstraction. There is little *scientific* motivation behind the statement that the three described levels of abstraction are “the” relevant ones; the *pragmatic* arguments are:

- 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 control loops than the one using the robot’s own sensing.

- every external information control loops *can* be deployed outside of the robot’s hardware. While control loops using the robot’s own sensors for the robot’s own motion *should* be deployed on the robot’s hardware.

The list of the three levels is as follows:

- **proprioceptive guarded motion.** This is the simplest type of task specification, because *“the robot is the world”*:
 - it uses only *perception* information that is 100% caused by the motion of the robot itself, and observed by its proprioceptive sensors (encoder, IMU and current).
 - the *world model* consists 100% of models of the robot itself: its geometry, kinematics, and the attachment points of actuators and sensors.
 - three *levels of abstraction* make sense: the actuators, the wheels, and the platform to which the wheels are connected.

Hence, also the *plan*, *control* and *monitoring* specifications are limited to that modelling context. For example, the robot starts moving straight ahead, using only itself as the reference for, both, the direction and the progress along the motion, and with a current, torque and/or velocity control activity for all wheels, *until* one the following monitoring conditions are met:

- the torque on one or more of the wheels is higher than a configured threshold.
- the measurements of IMU and encoder are not consistent.
- a configured time period has passed.

- **exteroceptive guarded motion task.** The task model adds the following *level of abstraction* to the previous task model: control specifications and monitoring “guards” are now also defined on those geometric primitives in the world model that *can be observed* by the robot’s exteroceptive sensors (camera and laser scanner).

In other words, *“the world is as large as the robot’s sensor go”*.

- **map-based motion from area A to area B .** This *level of abstraction* adds those task specifications that can make use of all information that is on *maps*.

In other words, *“the world is as large as the robot’s map”*.

For example, the sensor information can localize the robot on the map, and hence task specifications can now also use primitives that can not be observed physically or directly, but only because the maps defines their position in the world with respect to observable features. This allows tasks such as *“driving in traffic corridors”*, or *“remote opening of elevator doors”*.

Such **three-level architectures** are structured according to the **robot-centric** and **world model-centric** constraints of what the **robot** can **observe**, about **itself** and about the **world**. This is a major observation that supports this document’s strategic decision to place the world model as the unique and only centre of coupling into task metamodeling for robotics. These three levels appear naturally in robotics systems, so their [vertical composition](#) (of which Fig. 10.1 sketches a particular instantiation) provides a major use case for information architectures. The complementary core use case is the horizontal composition, sketched in Fig. 10.2: how to design an information architecture that allow robots to do [multitasking](#).

The following Sections provide more detailed descriptions of some core “control problems” in a three-level architecture. The design features three [holonic](#) activities, that share a common task execution ontology, and that do so via dedicated [streams](#).

10.2.1 Proprioceptive guarded motion holon

Current and torque control of wheel motor, power drive units, saturation, encoders for [Field-oriented Control](#), winding heating model, adaptive motor efficiency. Force actuation of wheel & guarded motion of platform; no jumps in torque setpoints, saturation, adaptive control,

- *operational modes*: in addition to the modes of the Life Cycle State Machine, the **Running** state has the following modes:
- *plan*:
- *control*:
- *perception*:
- *monitoring*:
- *world model*:
- *interactions of data, events, models*:
- *ownership, identifiers*:

10.2.2 Exteroceptive guarded motion control holon

- *operational modes*: in addition to the modes of the Life Cycle State Machine, the **Running** state has the following modes:
- *plan*:
- *control*:
- *perception*:
- *monitoring*:
- *world model*:
- *interactions of data, events, models*:
- *ownership, identifiers*:

10.2.3 Map-based guarded control holon

- *operational modes*: in addition to the modes of the Life Cycle State Machine, the **Running** state has the following modes:
- *plan*:
- *control*:
- *perception*:
- *monitoring*:
- *world model*:
- *interactions of data, events, models*:
- *ownership, identifiers*:

10.2.4 Inter-holon stream architecture

10.3 Specialisations of Producer-Consumer stream model

The meta model of **streams** can be adapted to many application context, 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.

10.3.1 Stream with 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. 10.4).

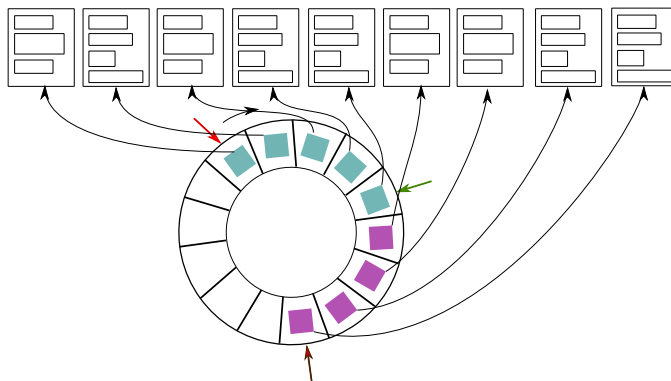


Figure 10.4: Stream with heterogeneous data chunks. The stream buffer is used only for efficient and effective ownership transformation of pointers to any type of data structure.

10.3.2 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.

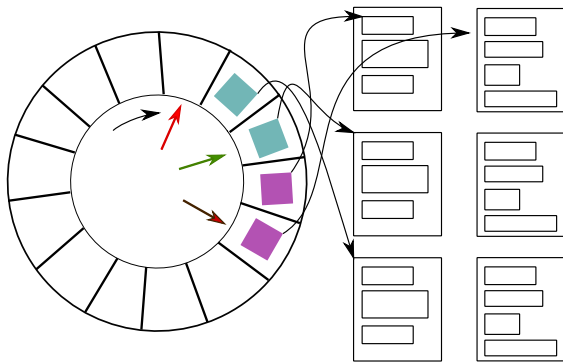


Figure 10.5: Stream with object pool. The data structures on the right are statically allocated, and they are only “borrowed” temporarily and “returned” at runtime, without being deallocated. The stream buffer is used only for efficient and effective ownership transformation of the pointers to the various available object data structures.

10.3.3 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. 10.6).

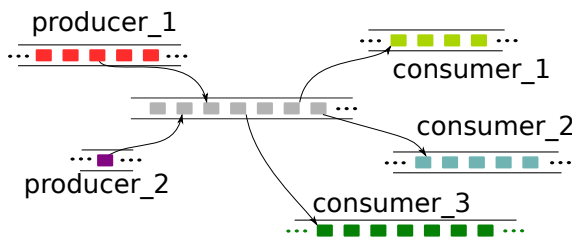


Figure 10.6: Single stream with multiple producers and consumers. Such an architecture requires explicit coordination between all producers, and another explicit coordination between all consumers, because *ownership* of each data chunk in each shared stream is not clear.

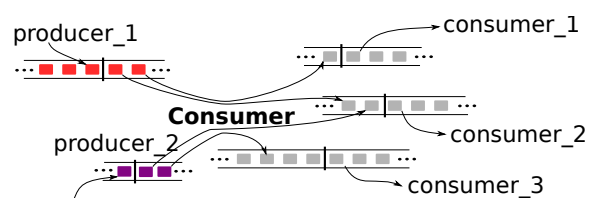


Figure 10.7: An equivalent architecture to Fig. 10.6 (*only* for the left-hand side of that Figure, covering the sharing of the “grey” stream): **one** intermediate **Consumer** activity is introduced in the architecture, and that activity is the **single** reader on the multiple producer streams, an 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 10.7 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. 10.6 are not directly produced from the grey “shared” stream in the middle, but first the “local” grey streams in Fig. 10.7 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.

10.3.4 Ownership, garbage collection, and blocking policies

(TODO: the producer of one stream should/can be the owner of multiple consumer sides of other streams it depends on. Example: realtime activity is the one that decides when data is being exchanged.)

10.4 Best practices

This Section introduces and motivates some best practices in assigning streams in information architectures.

10.4.1 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.

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 realtime 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.)

10.4.2 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.

(TODO: examples: actuator or sensor; task; state of a solver; resource allocation; etc.)

10.4.3 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 *realtime* 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-realtime thread* would do the copying, the data transfer could be interrupted by preemption of that thread.

(TODO: more examples.)

10.4.4 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.

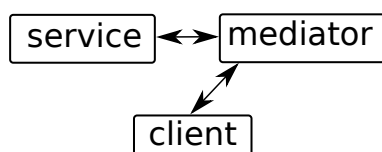


Figure 10.8: Mereo-topological model of the (peer) *mediator* pattern, to help system designers introduce *loose coupling* between “*service*” and “*client*” activities. The arrows represent *data access constraints* on the interacting activities.

10.5 Pattern: mediation in peer-to-peer activity interactions

The [structural compositions](#) shown in Figs 10.8 and 10.9 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).² 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,
- 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 is. The relevance of the separation between *coordination and configuration* on the one hand (realised by the mediator mechanism, introduced in Sec. 10.5.1) and the *computation and communication* on the other hand (realised by the peer activities themselves), increases with:

- the *number* of interacting peers (Fig. 10.9),
- the *complexity* of the interaction protocols,
- and the *statefulness* of the peers.

One of the consequences is that the mediator must provide a *Traffic Light* instance, to coordinate the multiple *Life Cycle State Machines* in all subsystem peers. The *streams* meta model already has such a coordination mechanism built-in, via its `control_flow` and `peer_activity_status_flags`. Hence, streams are very appropriate composition primitives for *loosely coupled* peer-to-peer architectures. What this Section adds to the system architecture discussion is a pattern about how the interacting peers can make best use of that mechanism.

10.5.1 Mechanism: the mediator activity

The design *driver force* in the pattern that has been created to realise the kind of interacting activities depicted in Fig. 10.8, is **to concentrate all knowledge and decision making** about the **behavioural coupling** in a so-called *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. The mediator activity checks

²The 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.

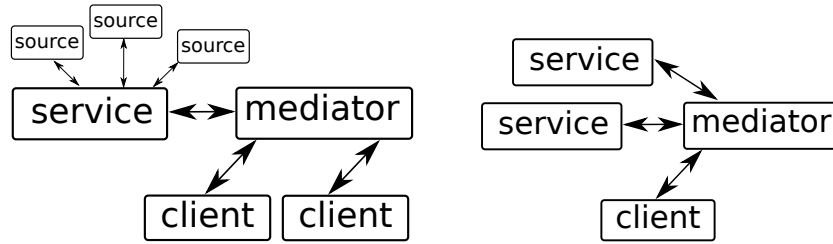


Figure 10.9: Variants of the *mediator* pattern, with multiple sources for the service, and multiple clients or services for the mediator.

whether those relations are compatible, decides to continue or not with its “*brokerage*” activity, and decides to include *glue code* where necessary.

All this model checking and adaptation takes place in the *resource configuration* step of the Life Cycle State Machines of all peers involved. This requires a protocol between mediator and peer to go through the same configuration steps at the same time.

2. which **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. 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, monitor the **quality of service** of the coupled behaviour and 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*.

10.5.2 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. 10.10).

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.)

10.5.3 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

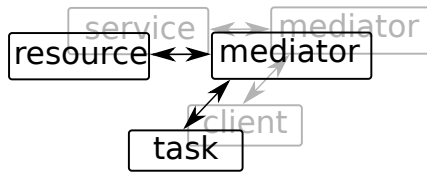


Figure 10.10: The special case of the mediator pattern applied to the interaction between *task* and *resource* activities.

“obviously” resilient subsystems. For example:

- in the context of *tasks* to organise the many interactions between humans into a loosely coupled societal hierarchy: person organise themselves in families, they form neighbourhood, making up villages, organised into a metropolitan area around a centre town, all assembled in regions and countries.
- in the context of *tasks* to provide building infrastructures: rooms are structurally connected by corridors, that form wards with some form of functional cohesion, aligned into floors, that form wings of buildings.
- 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-communicating lines; these lines, in turn, are laid out in plants; that are logistically interconnected and coordinated as a company.

Modern societal instantiations of the mediators are: the *foreman* in a team of factory workers, 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. 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 specifics 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.

10.5.4 Resilience via higher-order dependency relations

Because of the concentration of formalized knowledge about the inner workings of a system’s information architecture, mediators are *the* place to realise **resilience**. A necessary (but not sufficient) condition to achieve this ambition, is that the *human* system designers first succeed in *all* of the following:

- *to identify* the system’s capabilities and resources, and *formalize* the *hybrid constrained optimization problems* that model their interactions in a declarative way.

- *to formalize the knowledge* about the system’s internal behaviour: abstract data types and their operations; the activities and the event loops inside, to bring the system’s behaviour *live*; its operational modes and the corresponding finite state machine; Life Cycle State Machine to coordinate its resources; etc.
- *to identify* the spectrum of its own internal behaviour that can be explained by the formalised knowledge, and formalize the *boundaries* as constraints on the internal behavioural relations.
- *to identify* the spectrum of the external behaviour that it can tolerate at its interaction ports, and add the corresponding relations that constrain the internal behavioural spectrum.
- *to formalize* the knowledge that it needs *to explain* how to react “best” to external behaviour, based on reasoning with all of the above-mentioned knowledge relations.
- *to identify* how *to monitor* at runtime when the external interactions enter explainable behavioural areas, *and* how to connect the monitoring events to the internal coordination relations. *Formalize* these new higher-order knowledge relations.

The full package can only be provided in mature domains, and by senior domain experts. In addition, all knowledge relations involved are of the [higher-order](#) type, coupling all behavioural modes of all peers in the composition. Distributing decision making over multiple activities compromises explainability, so it is a *best practice* to have only **one mediator per composition** to realise that (higher-order) decision making. (This guideline is in itself an instance of another architectural pattern, the [singleton](#) pattern.)

10.6 Information architecture examples

This Section applies the material of all previous Sections to model some concrete information architectures, in the context of the [2WD mobile robot](#) example.

10.6.1 Proprioceptive motor–wheel–platform control

From the running example of the mobile robot with two actuated wheels, Sec. [10.1](#), this Section takes the [vertical composition](#) of four [levels of control](#).

(TODO: details of control behaviours in activities and their interactions via streams.)

10.6.2 Exteroceptive platform control

10.6.3 Safe interactive control over a network

This Section explains the **composability** of the advocated information architecture design by extending the controllers of Secs [10.6.1–10.6.2](#) with **shared control** functionalities: human operators can influence the motion control of the platform via a network connection, *and* receive “[kinesthetic feedback](#)” from the platform to the “master” arm they are physically interacting with.

(TODO: more details of control behaviours in activities and their interactions via streams.)

Chapter 11

Meta models for software architectures

A software architecture is the design layer between (i) the *application*-centred **information architecture** and (ii) the *resource*-centred **hardware architecture**. The responsibility of the software architecture is to provide **performance**, that is a loose composition of **efficiency**, **efficacy**, **effectiveness**, and **productivity**. The software architecture complies to the requirements and constraints of, both, information and hardware, and adds the decisions about

- which **data structures** and **solvers** to implement in which **language**.
- which **activities** to deploy in which operating system **threads**.¹
- which threads to deploy in which **processes**.
- which **producer-consumer streams** to realise with which **buffers** and **inter-process communication**.
- how to realise the **ownership** constraints of information architectures with concrete primitives in programming languages and operating systems.
- the **coordination of the access** to shared data.
- providing activities with sufficient quality of data **freshness** and **consistency**.

Because of the design driver trade-off towards performance, all of the above choices introduce **hard couplings**. The hardness of the coupling is relaxed by every option **to configure** that the software implementations introduce. This is an indicative list of configuration approaches in order of decreasing hardness: compile time of functions, deploy-time of binaries, start-up of processes, runtime of activities.

Inter-process communication has dependencies on the task information and introduces itself dependencies on the execution of activities. This results in complex *n-ary relations*, and hence software architecture models are **property graphs** in themselves.

Obvious major challenges are **to implement** the conceptually perfect information processing capabilities in the information architecture of the stream, with the imperfect resources available on computers; more in particular:

¹Or to run them on the **bare metal** of the CPU.

- 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-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 with the same names.
- resource contention, concurrency, and end-to-end latencies caused by multi-core/multi-computer execution contexts.

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.

11.1 Running example: mobile robot with two actuated wheels

Figure 10.3 depicts the platforms, its hardware architecture is described in Sec. 10.1.1. Multiple information architectures for various task contexts are explained in Sec. 10.1. This Section adds the concrete software architecture information.

(TODO: which activities to deploy in which threads; which processes to deploy these threads; which ringbuffers to use for which streams; which flags and FSM; which networks to connect computers. Special attention to distributing the world model activities.)

11.2 Mechanism: implementing streams by coupling two ring buffers

The stream meta model represents how a producer and a consumer interact with each other, asynchronously, via a stream. This Section extends the information architecture model with software-centric implementation design aspects. The core parts of that design are depicted in Fig. 11.1:

- the ring buffer² data structure, to map the conceptually infinite stream onto a finite part of the computer memory.

Two ring buffers are connected back to back, one for the producer activity in the stream, and one for the consumer activity.

- the ownership transfer operator.

Transfer of ownership on individual entries in the buffer takes place there where both ringbuffers are connected. **Three** such buffer connection points exist:

- from the producer-owned part to the consumer-owned part.

²“Circular buffer” is a commonly used alternative term.

- from the un-owned (“free”) part to the producer-owned part.
- from the consumer-owned part to the un-owned part.

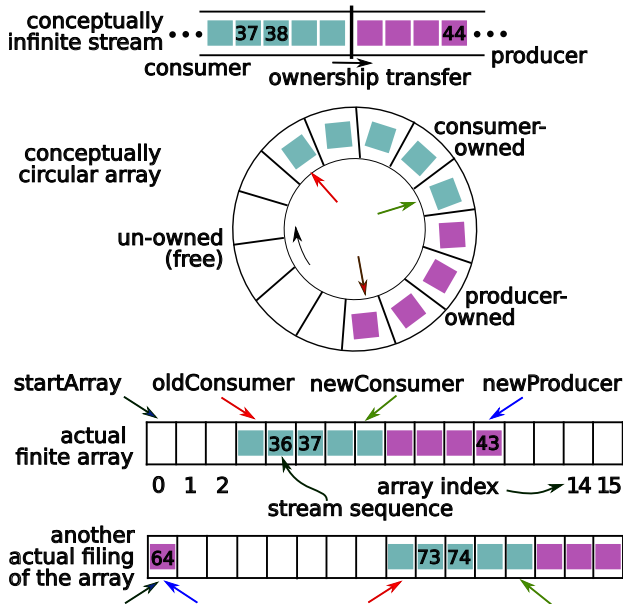


Figure 11.1: The information-architectural primitive of the **producer-consumer stream** (top), is mapped onto the conceptual software model of a *ring buffer* (center), and implemented with a finite array (bottom). The ring buffer array has three pointers: one where the producer gets data chunk ownership from the free part of the buffer, one where the ownership is transferred from producer to consumer, and one where the consumer transfers ownership back to the buffer. The finite array adds a fourth pointer, to “ground” the start of the array in actual memory. Each data entry that is added to the stream gets a new stream sequence number, counted by a perpetually incremented integer variable.

11.2.1 Software design

Various implementations of ring buffers exist (with [this](#) one introducing many of the design innovations). This Section composes (“couples”) two ring buffer instances back-to-back, one for the producer and one for the consumer, in such a way that producer and consumer activities can be **deployed** in the same thread, on different threads, in different memory-sharing processes, or on different cores, without changing any of the many software design decisions described below. Here is the set of design decisions, for the **particular but high-impact case** of a **single producer** activity and a **single consumer** activity:

- the **stream’s buffer** is a **finite array**³ of entries of the same type.
- the array is divided in **three contiguous parts**: one owned by the producer, one owned by the consumer, and the remaining un-owned part.
- producer and consumer can access all the array entries that they own in **random access** order.
- producer and consumer can **transfer ownership** of **any number** of the array entries that they own in **one operation**.

³The 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.

- the buffer becomes conceptually **circular** by constraining the successor of the last entry in the **buffer** array to be the first entry in the array.
- the array has an **arraySize** property, represented by an **integer number** n .
- 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** $[0, n]$.
- a **data_chunk** model represents the unique type of each entry in the stream. It is an **abstract data type** composed of parts from multiple **levels of representation**.
- one **integer number** in the **data_chunk** model is the **streamIndex** k , that represents a buffer entry's **sequence number** in the stream.

The sequence number is incremented perpetually, every time a new entry is produced. At a filling rate of one million entries per second, a 64-bit integer can guarantee that a **buffer overflow** will not occur for more than half a million years.⁴

- by making the **arraySize** n a power of 2, the **arrayIndex** i of an entry in the stream can be computed cheaply from the latter's **streamIndex** k , via a **modulo** operation: $i = k \bmod n$.

The example in Fig.11.1 has $n = 16$, so that **streamIndex** 36 has **arrayIndex** 4, and **streamIndex** 64 has **arrayIndex** 0.

- the array is positioned somewhere in the **RAM memory** of a process, and that address is pointed to by the **startArray** integer.
- the array needs **three pointers**, each being an **arrayIndex**:
 - **oldConsumer**, pointing to the oldest entry in the consumer-owned part of the array. The entry pointed to is the first one the consumer will release to the un-owned part of the buffer.
 - **newConsumer**, pointing to the end of the consumer-owned part of the array. This is the newest entry for which ownership has been transferred from producer to consumer.
 - **newProducer** of the producer-owned part of the array. This is the newest entry from which the producer has obtained ownership from the un-owned part of the buffer.

The start of the producer-owned part of the array could be represented by another pointer (“**oldProducer**”), but this is always just pointing one entry further into the array than the pointer of the end of the consumer-owned part. So, it carries no extra semantic information, and has also no implementation advantages.

- the **ownership** of the just-mentioned pointers lies as follows:
 - **oldConsumer**: owned by consumer activity.
 - **newConsumer**: owned by producer activity.

⁴ $2^{64}/10^6/60/60/24/365 = 584\,942$.

- `newProducer`: owned by producer activity.
- the producer activity can **change** the `newConsumer` and `newProducer` pointers, without risking a **race condition** with the consumer activity:
 - `newConsumer`: the ownership transfer operation does not overwrite any consumer-owned variables.
 - `newProducer`: before executing the ownership transfer operation, the producer activity first checks how many un-owned buffer entries are still available, and that check can never be invalidated by the consumer activity, because the producer activity is the only one to decrease the number of un-owned buffer entries.

Similarly, the consumer activity can change the `oldConsumer` pointer independently of the producer activity.

- transferring ownership of `newConsumer` by the producer activity just involves changing pointers. **Compare-and-Swap (CAS) operations** on such 64-bit integers are supported as **atomic operators** by all operating systems and many CPU hardware architectures.
- putting each of the three pointers in its 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.
- transferring ownership of `newProducer` by the producer activity consists of two steps: (i) filling in the buffer entry with an instance of the `data_chunk` type, and (ii) changing pointers. For the same reasons as mentioned above, both operations can be done in any order, *after* a successful availability check for un-owned buffer entries.
- transferring ownership of `oldConsumer` by the consumer activity just involves changing pointers. But even if the policy is to initialize the freed entry, there is no risk for race conditions.
- the buffer has an **integer number**, the `fillRate` f , which counts the number of entries in the buffer that are owned/occupied by consumer and producer:

$$f = \text{newProducer.streamIndex} - \text{oldConsumer.streamIndex} + 1. \quad (11.1)$$

- the buffer has two integers, `highWater` and `lowWater`, to generate the **backpressure** status flags:

$$\text{highWater} = \text{newConsumer.streamIndex} - \text{oldConsumer.streamIndex}, \quad (11.2)$$

$$\text{lowWater} = \text{newProducer.streamIndex} - \text{newConsumer.streamIndex}. \quad (11.3)$$

If these numbers are higher (respectively, lower) than configured values, the relevant **status flags** are set, informing the consumer (respectively, the producer) that it is time to empty (respectively, to fill) the buffer.

- the following **data consistency constraints** must hold at all times:
 - `arraySize` should never change.
 - `startArray` should never change.

- the buffer can not be filled to more than its capacity, that is, its `fillRate` f is always smaller than or equal to the `arraySize` n :

$$f \leq n. \quad (11.4)$$

- the sequence number of the entry pointed to by `newConsumer` is always larger than or equal to the sequence number of the entry pointed to by `oldConsumer`, and smaller than or equal to the sequence number of the entry pointed to by `newProducer`:

$$\text{newConsumer.streamIndex} \geq \text{oldConsumer.streamIndex}, \quad (11.5)$$

$$\text{newConsumer.streamIndex} \leq \text{newProducer.streamIndex}. \quad (11.6)$$

The values of these entries' indices in the array need *not* satisfy such constraints.

- the following **data access constraints** must hold at all times:
 - the producer can only fill/own new entries in the buffer if the `fillRate` allows it.
 - the **producer** is the only one **to increase** the `fillRate`.
 - (hence,) it is the only one **to set** the `highWater` mark, and **to clear** the `lowWater` mark.
 - the **consumer** is the only one **to decrease** the `fillRate`.
 - (hence,) it is the only one **to clear** the `highWater` mark, and **to set** the `lowWater` mark.
- the paragraph above *seems* to suggest that (i) the **buffer** implementation *must* provide the producer and consumer activities with data structures to represent `fillRate`, `highWater`, and `lowWater`, and (ii) the producer and consumer must both have read/write access to these variables. However, *neither* of these conditions is necessary, because the producer and consumer activities can do their jobs, independently of each other in their own separate threads, and in a **race-free** way, by *computing* their own **local versions** of these variables whenever they need them. Indeed:
 - they only have *to read* the integer numbers `newProducer.streamIndex`, and `oldConsumer.streamIndex`.
 - even if the consumer (producer) happens to read an “old value” of the number owned by the producer (consumer), the result is **always conservative**. That is, the producer can decide *not to fill* an entry because its local `highWater` or `fillRate` computation indicates that is the right thing to do, although the consumer has freed one or more entries in the meantime. Similarly, the consumer can decide *not to consume* an entry because its local `fillRate` computation indicates that the buffer is empty, although the producer has filled one or more entries in the meantime.
- the buffer has two **peer activity status flags**, `consumerModus` and `producerModus`, to represent the activity modulus of the producer and the consumer activities on the stream. Their values come from the following **enumerated type**:

- **active**: the activity is producing/consuming the stream.
- **inactive**: not **active**.
- **pausing**: the activity can become **active** any time soon.
- **requesting**: the consumer is waiting for the producer to become active, or the other way around.

Enumerated types always fit in data structures that can be read and written atomically on all hardware platforms, so updating the status flags does not introduce race conditions.

- the buffer has two **control_flow status flags** representing the **backpressure** status of the buffer:
 - **consumerPull**: **active** or **inactive**.
 - **producerPush**: **active** or **inactive**.

11.2.2 Policy: composition of data and meta data streams

The software-centric additions to the information architecture description in Sec. 2.5.8 are:

- the meta data **data_chunk** data structure must be defined.
- the meta data 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 meta data stream is with the same producer and consumer activities.

The former addition is the major challenge, but most of it belongs already to the responsibility of the information architect.

11.2.3 Policy: time series stream

Time series are a very important use case of streams with meta data. Two variant of the data structure of the meta data exist:

- a very simple and **singleton** meta data structure that just represents the fact that each entry in the data stream has its own time stamp.
- the more elaborate version (Fig. 2.4, in which each data chunk in the meta data stream has fields to represent:
 - the *range* of stream sequence numbers for which the following two meta data information hold.
 - the **provenance**: how was the data created, where does it come from, etc.
 - the **time representation** for each range.

11.2.4 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. 11.2.

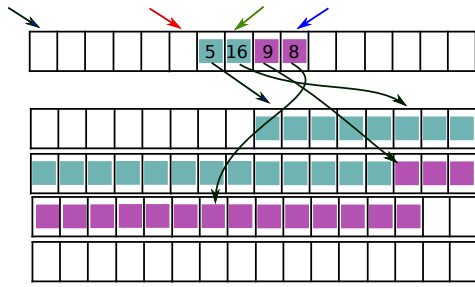


Figure 11.2: Stream buffer with memory pool. Each data chunk in the stream buffer contains two integers: the address pointer into the memory pool of the array that contains the serialized data chunk of the “real” data, and the size of that array. The stream buffer is used for the ownership transformation of these pointers from producer to consumer.

11.2.5 Policy: pipe line

The **pipe line** pattern is a simple form of composition. It can be considered as a boundary case for, both, dataflow programming and functional programming, since there is just a *linear* dependency between data buffers and function invocations. In the context of stream semantics, a pipeline is a stream in which a first consumer is the producer for a later consumer, and so on. That means that the buffer is now composed of more than two streams buffers connected back-to-back, and this preserves the very interesting ownership and efficiency properties.

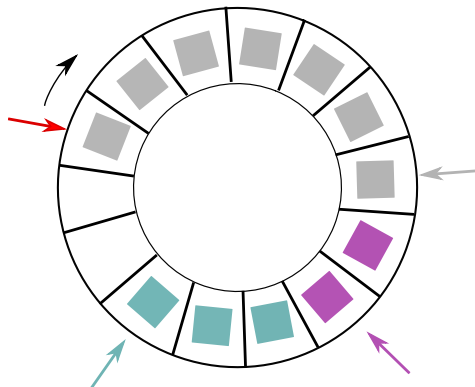


Figure 11.3: Stream ring buffer with pipeline composition. The stream buffer is partitioned over multiple owners, and ownership is transferred serially.

11.2.6 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 **flow.control** 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 time⁵ 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 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.

⁵That **timeout** is a configuration parameter.

- 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.

11.2.7 Pattern: 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. 11.2)
- an event loop (Sec. 2.8) 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 *vectored 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.

11.2.8 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*. The implications of such a stream buffer version 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.

11.2.9 Best practices

Here is a list of best practice software design insights:

- 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* [2] help to *avoid* the involvement of the operating system, such that applications have a larger impact on their own behaviour.

- the [CAP Theorem](#) and the [exactly-once semantics](#) of communication are major *constraints* for realising consistency of the exchanged data. Or, rather, once system designers are aware of the difficulty to realise all three aspects of “CAP”, they will look for architectures that are **robust** against (combinations of) each of the three disturbances.
- [garbage collection](#). Both producer and consumer can delegate to a “stream buffer activity” the decision making about what to do with a full or an empty stream buffer, to a **mediator activity** that is owned by the ring buffer. For example, in audio processing or control problems, a common policy is to overwrite 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:
 - *compaction*: the data in the buffer that has been produced but not yet consumed (or 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.
 - *clear* the whole buffer, because the *completeness* of the stream is not guaranteed 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 still owning all its side of the buffer, so it can execute the mediation actions itself.

11.2.10 Standards and software projects supporting streams

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.

11.3 Mechanism: stream with wait-free maximal freshness

The stream buffer mechanism of Sec. 11.2 has the advantages of:

- very clear and efficient ownership protocol.
- high reactivity of producer and consumer to the status of their interaction.

There are interaction use cases, however, that are not well served, such as tasks where multiple activities are interacting only sporadically, but when they do, they need a lot of information from each other and they want only the most fresh data. Major examples are where one activity needs images or results of repetitive database queries, at irregular times. With the stream buffer mechanism, these use cases result 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 pointers.
 - perpetually incrementable sequence numbers.

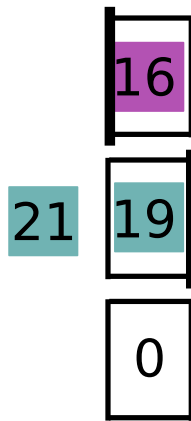


Figure 11.4: Stream with optimal freshness. Both producer and consumer can write, respectively read, the latest available version of the shared data structure. In the example, the consumer is still working on data chunk “16”, while the producer has already made data chunk “19” available for reading and 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 redundant to the sequence number being even (consumer-owned) or uneven (producer-owned); this sequence number is what is used in the implementation, and the bars just serve as human-centric visualisation.

Figure 11.4 illustrates the approach:

- 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 an `arraySize` of “1”.
- instead of letting the producer increment the sequence counter by “1” for each new data chunk, it now **increments it by “2”**, and it uses only **uneven** sequence numbers.
- when the consumer has consumed a data chunk, it puts the sequence number to “0”.
- the mechanism uses **conditional transfer of ownership**:
 - the producer can fill the slot with sequence number “0”, because the consumer will never use it, nor change its ownership.
 - when the producer has filled that slot, it atomically switches its sequence number to its next (uneven) version.
 - when the consumer wants to read a new data chunk, it checks the three slots in whatever order, increments the sequence number by one when it finds a filled one, and starts reading that slot.
 - by incrementing the sequence number, the consumer has **accepted ownership** that the producer has offered. That offer was conditional, because the producer can decide to clear a slot that it filled previously, as soon as it has a newer slot available and as long as the consumer has not taken it.

- when the consumer has finished reading the data chunk, it switches its sequence number to “0”, making it unconditionally available to the producer again.
- if it does this *before* it starts searching for a new data chunk, it will never own more than one slot.
- this means 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 number to “0”, in an atomic [Compare-and-Swap](#) (CAS) operation.

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.

11.4 Policy: event loop mediation for multiple stream buffers

The [event loop](#) is the computational work horse in software components, 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 meta data 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 [vectorized I/O](#) (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.

11.5 Mechanism (operating system): thread, process, shell, container, cloud

The **activity** implements computations, in the broad sense of the word, independent of how those computations are executed. An **activity** is [deployed](#) by an operating system to get access to the hardware resources that it must [share](#) with other **activities**, and for which the operating system is the *owning* activity. This sharing is realised in the following ways:

- [thread](#): the thread⁶ is the smallest operating system mechanism to make sure that the accesses (by the composition of all **activities** that it is running) to the CPU(s) and the [RAM](#) always satisfy the access constraints specified by each individual program.

In other words, the thread is the composition primitive that allows (or forces) functions **to share the same CPU**.

It is also the primitive to realise **runtime** (re)configuration and [introspection](#).

A [fiber](#) is a special type of thread, in that 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.

⁶More detailed introductions can be found [here](#) and [here](#).

- **process**: the **composition** of the execution of several threads, and coordinates the **stacks** of all its threads, to make sure that each program executes correctly irrespective of the number of times its execution has been **preempted**.

In other words, the process is the composition primitive that allows (or forces) functions **to share the same computer memory**.

It is also the primitive to do **compile time** configuration.

- **shell**: operating systems provide the shell primitive as a shared **context** for several processes to be active in. The same process can be started within different shells, and get another set of **environment variables** that the process uses to configure its behaviour.

In other words, the shell is the composition primitive that allows (or forces) processes **to share the same 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 allows (or forces) shells **to share the same operating system instance**.

- **cloud**: **composes** several computers, and coordinates their internet communication interactions.

These entities will not be detailed further in this document, because there is no specific connection between those entities and the properties of robotic and cyber-physical systems, *at the level of knowledge relations*. The entities and relations in this Section, together with those of the following Section, have already received significant meta modelling attention, for example in the **AADL** standard and supporting **software and tools**. (Unfortunately, AADL violates almost all “best practices” advocated in this document, such as: separation of mechanism and policy, inversion of control, or composability.)

(TODO: thread pool: set of threads waiting on ring buffers where each entry in the buffer is a task to be executed, and that fill one or more result ring buffers.)

11.6 Pattern: composition of (a)synchronous threads

Figure 11.5 sketches the **multi-threading architecture** that is common to the **deployment** of many **information architectures**’ activities and **streams** onto **real-time** control threads.

11.6.1 Software design

The design drivers behind this pattern are as follows:

1. one **main** thread.

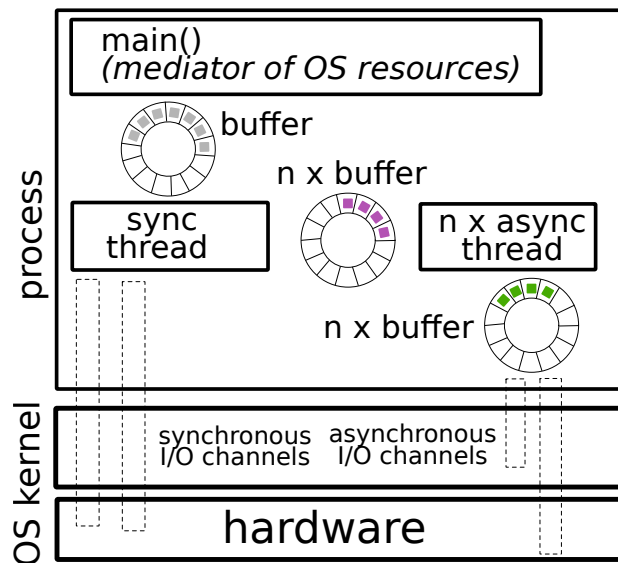


Figure 11.5: This process architecture is a pattern to let synchronous and asynchronous activities run in several threads: one *main*, one *synchronous thread*, and one or more *asynchronous threads*. All threads exchange their information via ring buffers.

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.

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 advocates to use the `main` thread exclusively to configure resources owned by the operating system:

- creating and scheduling of [threads](#).
The [POSIX threads](#) standard and implementations are a software basis to realise this role.
- [memory management](#).
The Linux operating systems offers software support in the form of, for example, [mmap](#), [cgroups](#), or [systemd](#).
- 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](#).

In other words, the `main` thread executes the activity in the `deploying` state of the application's [Life Cycle State Machine](#).

2. one synchronous thread.

This one's activity consist of the [event loop](#) of the real-time control. In other words, the core activity in the `running` state of the application's [Life Cycle State Machine](#).

The `synchronous` thread should never *block*, which can only work if the system's hardware and software are designed with this performance objective in mind:

- 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](#).
3. one or more **asynchronous** threads.

The role of each of these threads in the process is to serve as (i) a stream interface towards out-of-process activities (including device drivers for hardware, and local area networking), (ii) a coordinator for a set of other activities, or (iii) a mediator for a set of other activities. Each of the **asynchronous** threads communicates with a resource or with another activity, and is allowed to *block* while doing so. In order not to let the **synchronous** thread wait, the stream buffer between them has received ample resources from the deployment activity.

Applying it to the context of a robotics motion control stack, instances of this pattern can look as follows:

- *torque control of 2WD platform.*

11.6.2 Bad practices

- let threads set their priority and CPU affinity themselves.
- let them behave as if they are master of any interaction they are involved in, that is, let a “request” be interpreted as a “command”.
- not to use an Life Cycle State Machine for every resource they own.
- not to be unambiguous about which resources every thread owns.
- let the “event loop” be realized by the operating system. That operating system does not know anything about the application, 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.

11.6.3 Good practices

- map RAM in memory to prevent swapping out, e.g, via [mmap](#).
- by definition of the word, the “highest priority” can be given to one single thread only. Here is the [POSIX](#) way of configuring this.
- allocate a dedicated core to the **synchronous** thread. Or even one whole computer.
- [masking interrupts](#) for that core, to only the ones needed in the **synchronous** thread.
- reserving CPU, memory and I/O from operating system. For example, via [systemd](#).

11.6.4 Policy: mediation on shared resources

The mediator can rightfully be called a “software pattern”, since 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!)

11.7 Mechanism: programming language operators

11.7.1 Async/await

(TODO: [async/await](#) programming language construct to make some forms of asynchronous programming look very much like synchronous programming; discussion on when to use it and when not; examples needed.)

11.7.2 Iterator

(TODO: [iterator](#) helps to separate the structure of the data from the operations, in a declarative way.)

11.7.3 Maybe

(TODO: [Maybe](#) type, to represent not-yet-known data.)

11.7.4 Memory barrier

(TODO: [memory barrier](#), to order access to fast memory.)

11.7.5 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.

11.7.6 Bad practices with locks

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.

11.7.7 Bad practices in communication

- to assume that messages will be delivered *exactly once*.
- to use only the *publish-subscribe* communication pattern. Events often profit from a *broadcasting* policy, and queries about models require one-on-one *dialogues*; neither is done well best with pub-sub.
- 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.
- 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.

11.8 Mechanism: core, system-on-a-chip, computer, cloud

Somewhat further away from the properties of robotic and cyber-physical systems is the *computer hardware*, with 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:

- **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.

11.9 Policy: framework plug-ins versus library composition

Frameworks are one of the most popular ways towards digital platforms, for three good reasons: *code reuse*, *best practice implementations*, and *tooling*.

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 be called by the framework’s *runtime engine* at the “right” time. However, the frameworks themselves are very poorly composable with other frameworks or systems, into peer-to-peer architectures, because:

- their plug-in interface offers only one single level of composition hierarchy. So, it is for example not possible to develop a function that couples “state” at two different plug-in interfaces.
- their runtime engines typically expect that plug-ins adapt to their policies and protocols, and not the other way around. So, it is for example not possible to configure the runtime engine to share resources with non-framework activities.

In robotics, common cases of the mentioned composability gaps pertain to integration between “control”, “perception” and “monitoring” functionalities.

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 hard immutable choices.

Chapter 12

Skill architectures for the composition of Tasks

(TODO: configure information and software architectures, to let several robots with several capabilities execute several tasks at the same time, in mutual coordination and interaction.)

Chapter 13

Skill architectures for two-arm manipulator robots

Dual-arm robotic tasks exploit the full potential of the hardware only with a realtime integration of the Tasks in the application with the Tasks in each of the arms.

This Chapter presents the application context of two-arm manipulators, with each arm being a redundant serial kinematic chain. The sketch in Fig. 4.13 makes the arms branch from a common “torso”, which is itself a serial kinematic chain connected rigidly to the ground; of course, the latter connection could be to a mobile platform in itself, Chap. 14.

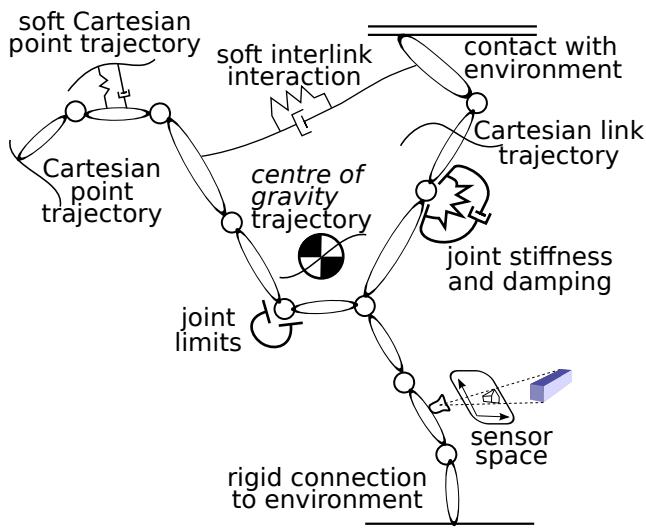


Figure 13.1: Sketch of a dual-arm manipulator, with all possible “*motion constraints and drivers*”.

Controlling the platform takes place at three levels of abstraction at the same time: (i) the local motion of both arms, (ii) the motion of all joints inside the arms, and (iii) the actuators. Hence, controllers at these three levels interact in multiple ways:
(TODO: most of the chapter...)

Chapter 14

Skill architectures for mobile robots

Chapter 15

Skill architectures for cable robots

Bibliography

- [1] 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.
- [2] AL BAHRA, S. Nonblocking algorithms and scalable multicore programming. *Communications of the ACM* 58, 7 (2013), 50–61.
- [3] 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.
- [4] ALEXANDER, C., ISHIKAWA, S., AND SILVERSTEIN, M. *A pattern language: towns, buildings, construction*. Oxford University Press, 1977.
- [5] ANGELES, J. *Rational Kinematics*. Springer, 1988.
- [6] 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.
- [7] 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.
- [8] 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).
- [9] 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.
- [10] BISHOP, C. M. *Pattern Recognition and Machine Learning*. Springer, 2006.
- [11] BORGHEAN, 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.
- [12] BORGO, 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.

- [13] BORST, P., AKKERMANS, H., AND TOP, J. Engineering ontologies. *International Journal on Human-Computer Studies* 46 (1997), 365–406.
- [14] BOTTEMA, O., AND ROTH, B. *Theoretical Kinematics*. Dover Books on Engineering. Dover Publications, Inc., Mineola, NY, 1990.
- [15] BRUYNINCKX, H., AND DE SCHUTTER, J. Specification of force-controlled actions in the “Task Frame Formalism”: A synthesis. *IEEE Transactions on Robotics and Automation* 12, 5 (1996), 581–589.
- [16] BURKE, W. L. *Applied differential geometry*. Cambridge University Press, 1992.
- [17] CECCARELLI, M. Screw axis defined by Giulio Mozzi in 1763. In *9th World Congress IFToMM* (Milano, Italy, 1995), pp. 3187–3190.
- [18] 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.
- [19] 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.
- [20] CHEN, P. P.-S. The entity-relationship model—Toward a unified view of data. *ACM Transactions on Database Systems* 1, 1 (1976), 9–36.
- [21] COX, I. J., AND LEONARD, J. J. Modeling a dynamic environment using a Bayesian multiple hypothesis approach. *Artificial Intelligence* 66 (1994), 311–344.
- [22] 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.
- [23] D’ALEMBERT, J. L. R. *Traité de Dynamique*. 1742.
- [24] 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.
- [25] DE MAUPERTUIS, P. L. M. Accord de différentes lois de la nature. In *Oeuvres, Tome IV*. Olms, Hildesheim, Germany, 1768.
- [26] DE SCHUTTER, J., DE LAET, T., RUTGEERTS, J., DECREÉ, 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.
- [27] EHRIG, H., ERMEL, C., GOLAS, U., AND HERMANN, F. *Graph and Model Transformation. General Framework and Applications*. Monographs in Theoretical Computer Science. Springer, 2015.
- [28] ENGELS, G., LEWERENTZ, C., SCHÄFER, W., SCHÜRR, A., AND WESTFECHTEL, B., Eds. *Graph Transformations and Model-Driven Engineering*. Springer, 2010.

- [29] FEATHERSTONE, R. *Rigid Body Dynamics Algorithms*. Springer, 2008.
- [30] 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* (Signapore, 2017), pp. 5203–5208.
- [31] FRANKEL, T. *The Geometry of Physics*. Cambridge University Press, Cambridge, England, 1996.
- [32] GAUSS, K. F. Über ein neues allgemeines Grundgesetz der Mechanik. *Journal für die reine und angewandte Mathematik* 4 (1829), 232–235.
- [33] 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.
- [34] 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.
- [35] GOLDSTEIN, H. *Classical mechanics*, 2nd ed. Addison-Wesley Series in Physics. Addison-Wesley, Reading, MA, 1980.
- [36] GOOD, I. J. A derivation of the probabilistic explanation of information. *Journal of the Royal Statistical Society (Series B)* 28 (1966), 578–581.
- [37] 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.
- [38] 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.
- [39] 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).
- [40] HOLVOET, T., WEYNS, D., AND VALCKENAERS, P. Delegate MAS patterns for large-scale distributed coordination and control applications.
- [41] HUNT, K. H. *Kinematic Geometry of Mechanisms*, 2nd ed. Oxford Science Publications, Oxford, England, 1990.
- [42] IMIYA, A. A metric for spatial lines. *Pattern Recognition Letters* 17 (1996), 1265–1269.
- [43] JAYNES, E. T. *Probability Theory: The Logic of Science*. Cambridge University Press, 2003.
- [44] JOURDAIN, P. E. B. Note on an analogue of Gauss’ Principle of least constraint. *Quarterly Journal of Pure and Applied Mathematics* 8L (1909).
- [45] KARGER, A., AND NOVAK, J. *Space kinematics and Lie groups*. Gordon and Breach, New York, NY, 1985.

- [46] 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.
- [47] KOESTLER, A. *The Ghost in the Machine*. 1967.
- [48] 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.
- [49] LAGRANGE, J. L. Mécanique analytique. In *Oeuvres*, J.-A. Serret, Ed. Gauthier-Villars, Paris, France, 1867.
- [50] 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.
- [51] 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.
- [52] 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.
- [53] LONČARIĆ, J. Normal forms of stiffness and compliance matrices. *IEEE Journal of Robotics and Automation* RA-3, 6 (1987), 567–572.
- [54] MADNI, A. M., AND SCOTT, J. Towards a conceptual framework for resilience engineering. *IEEE Systems Journal* 3, 2 (2009), 181–191.
- [55] MANSARD, N., AND CHAUMETTE, F. Task sequencing for sensor-based control. *IEEE Transactions on Robotics* 23, 1 (2007), 60–72.
- [56] MASON, M. T. Compliance and force control for computer controlled manipulators. *IEEE Transactions on Systems, Man, and Cybernetics* SMC-11, 6 (1981), 418–432.
- [57] MOZZI, G. *Discorso Matematico sopra il Rotamento Momentaneo dei Corpi*. Stamperia del Donato Campo, Napoli, 1763.
- [58] 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.
- [59] PEARL, J. Fusion, propagation, and structuring in belief networks. *Artificial Intelligence* 29 (1986), 241–288.
- [60] 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).

- [61] 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.
- [62] PHILIPS, J., VALCKENAERS, P., AERTBELIËN, E., VAN BELLE, JAN EN SAINT GERMAIN, B., BRUYNINCKX, H., AND VAN BRUSSEL, H. PROSA and delegate MAS in robotics. In *Holonic and Multi-Agent Systems for Manufacturing*, vol. 6867 of *Lecture Notes in Computer Science*. 2011, pp. 195–204.
- [63] 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.
- [64] POINSOT, L. Sur la composition des moments et la composition des aires. *Journal de l'Ecole Polytechnique* 6, 13 (1806), 182–205.
- [65] POPOV, E. P., VERESHCHAGIN, A. F., AND ZENKEVICH, S. L. *Manipulyatsionnye roboty: dinamika i algoritmy*. Nauka, Moscow, 1978.
- [66] RADESTOCK, M., AND EISENBACH, S. Coordination in evolving systems. In *Trends in Distributed Systems. CORBA and Beyond*. Springer-Verlag, 1996, pp. 162–176.
- [67] RAMM, E. Principles of least action and of least constraint. *Gesellschaft f. Angewandte Mathematik und Mechanik (GAMM)* 34, 2 (2011), 164–182.
- [68] REID, D. B. An algorithm for tracking multiple targets. *IEEE Transactions on Automatic Control AC-24*, 6 (December 1979), 843–854.
- [69] 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.
- [70] 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.
- [71] 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.
- [72] SCHULZ, M. Decisions and higher-order knowledge. *Noûs* 51, 3 (2017), 463–483.
- [73] SCHÜRR, A. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science*, vol. 903 of *Springer Lecture Notes in Computer Science*. 1995, pp. 151–163.
- [74] 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.

- [75] 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.
- [76] SIMON, H. *The Sciences of the Artificial*. MIT Press, 1969.
- [77] SIMON, H. A. Rational choice and the structure of the environment. *Psychological Review* 63, 2 (1956), 129–138.
- [78] VALCKENAERS, PAULAND VAN BRUSSEL, H., BRUYNINCKX, H., SAINT GERMAIN, B., VAN BELLE, J., AND PHILIPS, J. Predicting the unexpected. *Computers in Industry* 62 (2011), 623–637.
- [79] VALCKENAERS, P., BONNEVILLE, F., VAN BRUSSEL, H., BONGAERTS, L., AND WYNS, J. Results of the holonic control system benchmark at KU Leuven. In *Computer Integrated Manufacturing and Automation Technology* (1994), pp. 128–133.
- [80] VALCKENAERS, P., VAN BRUSSEL, H., HADELI, BOCHMANN, O., SAINT GERMAIN, B., AND ZAMFIRESCU, C. On the design of emergent systems: an investigation of integration and interoperability issues. *Engineering Applications of Artificial Intelligence* 16 (2003), 377–393.
- [81] VAN BRUSSEL, H. Holonic manufacturing systems. The vision matching the problem. In *First European Conference on Holonic Manufacturing Systems* (1994), pp. 1–11.
- [82] 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.
- [83] 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.
- [84] 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.
- [85] 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.
- [86] W3C. QUDT (Quantities, Units, Dimensions, and Types). <http://www.qudt.org>.
- [87] 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.
- [88] WILL, P. M., AND GROSSMAN, D. D. An experimental system for computer controlled mechanical assembly. *IEEE Transactions on Computers* 24, 9 (1975), 879–888.

- [89] WINFIELD, A. F. T. Experiments in artificial theory of mind: From safety to story-telling. *Frontiers in Robotics and AI* 5 (2018).
- [90] WIRTH, N. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.
- [91] 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.
- [92] ZELLNER, A. Optimal information processing and Bayes’s theorem. *The American Statistician* 42 (1988), 278–284.