RobMoSys

H2020-ICT-732410

RobMoSys

Composable Models and Software for Robotics Systems

Deliverable D3.1:

First 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:	D3.1
Document title:	First motion, perception and world-model stacks specifications
Version:	1.1
Delivery date:	4 July, 2017
Nature:	Report (R)
Dissemination level:	Public (PU)
Editor:	Enea Scioni (KUL), Herman Bruyninckx (KUL)
Authors:	Enea Scioni (KUL), Nico Hübel (KUL), Herman Bruyninckx (KUL), Alex Lotz (HSU), Dennis Stampfer (HSU), Christian Schlegen (HSU)
Reviewer:	Alex Lotz (HSU)

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

Executive Summary

This deliverable provides the first contribution on the *modelling* of the so-called *motion*, *perception* and *world model stacks*, since these are foundations of any *digital platform* for robotic systems.

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 grounding into software implementations, and processes of validation and certification of components and systems. Two core aspects are: the omnipresence of the *Block-Port-Connector* meta meta model to represent *all* structural relationships and compositions, and the introduction of a formal model to represent the data structures, functions and control flow schedules of algorithms.

The second part of the modelling brings in robotics-specific material, more specifically about motion and perception. Both share the same formalisation of their mathematical, numerical and digital representations, as well as the connection to meta data for physical dimensions and units. The motion stack core consist of models of geometrical entities, relations and constraints, from points and lines upto kinematic chains with shape, inertia, actuators and sensors. The major "behavioural" functionality to be modelled explicitly is that of the *hybrid dynamics solvers* that is the generic basis of all instantaneous motion and force transformations between the joint space and the Cartesian space of all kinematic chains. Similarly, perception is composed of models of sensors, sensing features, data association to object features, and constraints imposed by the task, the environment and the object properties; *message passing* is the solver which plays a similar foundational role in adding behaviour to the models in perception as the hybrid dynamics algorithm does for motion. The world model stack is a knowledge-based system, backbone of the information infrastructure around the motion and perception stacks.

Contents

1. Introduction

"Motion" and "Perception" are two essential functionalities in any robotic system, and both have a rich and mature history. In other words, there is more than enough understanding about what constitutes a solid basis of theory and algorithms to include in a *digital platform* for robotics. For the motion part, this basis is given by the generic algorithm to solve the "hybrid" instantaneous kinematics and dynamics relationships that hold between the forces and the motions on ideal, lumped-parameter, kinematic chains. For the perception part, a similar role is played by Bayesian graphical model templates that relate raw sensor data to the task-centred features of objects involved in the robots' tasks.

This Deliverable makes concrete suggestions about how to turn those 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. Because the ambition of the Robmosys project is to (help) build the *platform* only, a lot of attention was given to creating the "right" modularity, the "right" levels of detail, and the "right" separation of concerns, and the "right" approach towards *composability*, such that the development of models, tools, implementations and applications becomes methodological, transparant and scalable.

1.1 Goal of First Call

This deliverable contains the drafts of the Motion Stack and the Perception Stack as examples for Tier 2 domain-specific models within the RobMoSys structure. These drafts will be subject to changes and refinements over the course of the project, not in the least because achieving the envisaged "right" set of models is probably a never ending process of community-driven improvements. The latest versions can always be found on the RobMoSys Wiki. Constructive feedback from the community is welcome and encouraged and can be subject of projects in the first call.

For the first call we consider the following subjects for project proposals:

- **(Meta-)Models**. Creating or refining Tier 2 domain-specific models. This includes refinements of the presented motion or perception stack as well as proposals for neighbouring domains such as control, estimation, planning, visualisation, data acquisiation and management, or decision making. This also includes work on how to encode these formal models (e.g. with domain-independent host languages such as JSON-LS, OWL, XML, etc.), so that they can be used in tooling prototypes. Finally, multiple user-friendly *domain-specific languages* are sollicited, to facilitate the specification of the usage of the models to a particular type of users, in a particular type of application domain.

- **Software Modules**. Building new modules, or refactoring already existing software modules, such that they conform to the RobMoSys modeling approach. This means that these modules have to be composable with all assumptions being explicated by modelling. Verification of of composability is required, e.g. by reusing the same module(s) within two different demonstrations by just reconfiguring them (according to their models).

  Note that "software modules" will be, both, *algorithms* provided as standalone libraries, and *components* that come ready to be deployed in a component framework.

- **Tools**. Tools should make use of the meta-models and models to help building systems. This means that they offer (one or more) specific views to the different roles[1] foreseen by the project. While some of the meta-models of the motion stack are mature enough such that tooling is possible for them up to the application level, that is, various motion *control* modes, this is more difficult for the perception stack due to the enormous variety in applications, sensors and estimators.

Please note that **components** have to be delivered together with their corresponding **models** that have to conform to the RobMoSys meta-models; not conforming to these models needs to be motivated, meaning that a project outcome could lead to a refinement of the meta-models. Concrete examples of tooling to make effective use of the models are also expected, to make sure that suggested models and implementations have a complexity trade-off that warrants the cost of developers and users to go through the inevitable learning curve.

1.2 Generic modelling foundations

This Section is a brief summary of the modelling concepts, and of the various relevant "levels of abstraction", that this document inherits from the broader RobMoSys context. (More complete and updated versions of the following paragraphs can always be found on the RobMoSys Wiki pages.) For the *kinematic chain* purposes of this document, (only) the five levels of abstraction described in Sec. 1.2.2 are relevant, together with an explicit *partial ordering* on these five levels. But before we go there, the following Section summarizes the axiomatic role played in the RobMoSys modelling approach by the concepts of *Entity* (representing "stuff", "things", "primitives", "atoms",...) and *Relation* (representing dependencies between Entities).

1.2.1 Entities, Relations and Property graphs

Modelling boils down to making an artificial language to represent, in a formal way, the properties of real-world entities for which one needs "digital twins", together with the relationships that govern the interaction between the entities. For all but the most simple models, the result is a set of *graphs*, where the nodes represent entities, the edges represent relations, and both have a set of properties (like name, identity, type, provenance, etc.). Very often, one also needs to include *constraints* in the language, to express some limits on the values of the properties of entities and relations; hence, one needs *higher-order* models (or reification), by which relations become entities themselves, to become arguments in other relationships or constraints. The result of computer representations of all the "digital twins" and their interconnections will be a so-called graph database, which provide semantic query interfaces.
This Section introduces the mereological and topological (Sec. 1.2.2) representations of the *Entity* and *Relation* concept that underly all modelling. The generic and seemingly obvious textual representation of a relations looks like this:

```
Relation ( Argument1, Argument2, Argument3 ).
```
(1.1)

Each `Argument` has a specific `Role` in the `Relation`. For example, a kinematic chain is a relationship representing motion constraints between rigid body links and (typically) one-dimensional

[1]http://robmosys.eu/wiki/general_principles:ecosystem:roles

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.
Equation (1.1) is a so-called *mereological* model, since the only thing it models is that a relation is a "whole" `has-a` argument as a "part", three times. And the *type* of the `has-a` is that of "being an argument with a particular role." Mereological models might seem overly simplified and obvious, but they have already a very important role to play in large-scale modelling efforts like RobMoSys: 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. And that effort is exacty what this document is kickstarting, for the robotics sub-domains of *motion* and *perception*; the document uses the terms "stack" to represent the RobMoSys project's ambition to find, together with the community, an agreement about how to structure all the required modelling, from the "bottom" up, until there is enough to serve as a **platform** for all robotics applications to build upon.

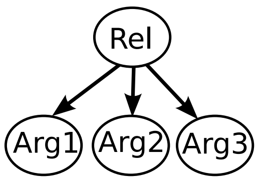


Figure 1.1: A *directed graph* representation to model relations.

Figure 1.1 depicts the *directed graph* model to represent a `Relation`. Such *directed graphs* are commonly supported by *graph database* sofware *to implement property graphs* [3], that is, a graph with property data structures attached to both node and edge.
At this **mereology** level (Sec. 1.2.2), *queries* are possible about what arguments are used in what relations, which argument are correlated, etc. Such queries can be answered by *graph traversal* tools, like Gremlin or SPARQL.
JSON-LD, RDF or XML, are choices of *host languages*, which are supported by rich ecosystems of tools, developers and users. The examples below use JSON-LD because it has (for the time being) a somewhat better support for our purposes, that is, representing *named directed graphs*, with built-in keywords for *unique identifiers*, *conformance to meta models*, *possibility to represent higher-order relations*, and *inter-linking of model files*. In order to be used in such a context of higher-order and interlinked models, the simple representation of Equation (1.1) needs more technical details to be modelled in JSON-LD:

```
{
  "@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#i
           ] } }
     ] }
}
```

The **topological** level of representation (Sec. 1.2.2) introduces a formal representation of the *connectivity* **structure**. This formal representation extends the generic Block-Port-Connector (BPC) meta model[2]:

- **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 much get an extra modelling primitive, the **Docks**: each Port must have exactly one *inside* Dock and one *outside* Dock, and both have a *Connector* between them. The constraints on both ends of this Connector are just(?) type compatibilities.

1.2.2 Mereology, topology, geometry, dynamics

A first reason to introduce explicit structure is to provide, to *human* developers, an explicit *context* to their modelling efforts and discussions, because experience has shown that such context can prevent most "religious wars" that tend to arise between developers, about whether or not a particular feature or property has to be included in a particular model. For example, at the *geometrical* level of abstraction, models of a "Rigid body" must somehow represent the fact that the *distance* between any two points on the body must remain constant. Only at the *dynamical* level, the *weight* of the body, and *forces* on the body, become relevant concepts; also at this level only, *deformations* (i.e., *non*-constant distances between points on the body) are introduced, together with the concepts of *force* and *elasticity*.
A second reason to bring structure in abstraction levels is to support *efficient automatic reasoning* in the software tools that developers will want to use to build robotic software systems: the higher level of abstraction that can be used in the reasoning that is required to answer "queries", the more efficiently the query answering can be implemented. For example, only the topological properties

[2]see also Deliverable D2.2

of a kinematic chain are needed to answer questions about *connectivity* between links, or about the *kinematic family* that a particular chain belongs to (e.g., serial or parallel robots); the bigger geometric models only come into play when the *magnitudes* of motions must be computed.

1. Mereology: this is the abstraction level where only the "whole" and its "parts" are modelled. The relevant relations are `has-a`, `collections`

2. Manifolds & Objects: all the "wholes" and "parts" in a particular context most often are further detailed in two complementary ways: as *"discrete"* things ("objects") or as *"continuous"* things ("manifolds"). Note that the same mereological Entity or Relation can have more detailed models that have both discrete and continuous parts. For example, a kinematic chain has a continuous motion space, but also a discrete set of actuators and sensors.

3. Topology: for both the manifold and object types of Entities and Relations, there is a need to introduce extra concepts with topological meaning, to represent "neighbourhood", more in particular the `contains` and `connects` relations. Each type has its own extra topological relations.

3.1. Spatial topology: `near-to`, `left-of`, `on-top-of`,...

3.2. Object topology: `block`, `port`, `connector`, and `dock`; or the *taxonomy* relationships of `hypernyms/hyponyms` of objects and verbs.

4. Geometry: this is the next level of modelling abstraction (for the manifold type of Entities and Relations only!), that introduces more concrete semantic types of Entities and Relationships. Also within the geometry context, sub-categories with commonly accepted names and meaning can be identified.

4.1. Affine geometry: `point`, `line`, `hyperplane`; `intersect`, `parallel`, `ratio`.

4.2. Metric geometry: `rigid-body`, `shape`, `orientation`, `pose`, `angle`, `distance`, `orthogonal`, `displacement`,...

5. Dynamics: the last level of modelling abstraction brings in the interactions between "motion" and "force".

5.0. 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`.

5.1. Mechanics: `mass`, `force`, `elasticity`, `damping`, `gravity`, `momentum`, `potential-energy`, `kinetic-energy`,...

5.2. Electrics: `resistor`, `inductance`, `capacitance`,...

1.2.3 Mechanism (structure & behaviour) versus policy

Because RobMoSys is a *platform* project, it has a high emphasis to separate the "mechanism" aspects from the "policy" aspects:

- *mechanism*: **what** does a piece of functionality do? Irrespective of the application in which it is used. Often, "mechanism" is subdivided into:
  - *structure*: how are the parts of the model/software connected together?
  - *behaviour*: what "state changes" does each part realise?
- *policy*: **how** can that piece of functionality be used to serve a particular application? 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 of an application is optimized towards the particular application context..

1.2.4 Grounding, validation, certification

Note that none of the models above is a complete formal representation of *all* relevant knowledge. mechanism or policy: they *refer* to other models that provide that "context". Of course, this interlinking of models must stop somewhere; in the case of robotics software systems, this "grounding" takes place when a piece of concrete software is composed with a set of models, and a human expert has validated that this implementation is correctly realising what is represented in the models. It is a community responsibility to decide what grounding it will expect, for what kind of purposes. For example, formal verification expectations are a lot lower for ROS-based educational robotics systems than for ESA's planetary rovers and manipulators; hence, also the accepted level of grounding will be more stringent in the latter case.
From a modelling point of view, validation, verification, certification etc. are *processes* that do something with models and software, and produce a *metadata* model that documents the process. This is, in principle, fully covered by *composition* of models.

2. Motion Stack Overview

The RobMoSys project has two major ambitions: to create a (effective, efficient and fair) *digital platform* for robotics, and to do so in a *composable* way. Hence, this chapter focuses on the well-understood basis of how to make robot kinematic chains "move", with a structural methodology and a feature set that can cover the motion needs in all mainstream applications, cut in pieces that are small enough (but not smaller!) to make sure that no specific set of applications is (implicitly) served better than other sets of applications, because some of the structure, "magic numbers", functions, or terminology are biased towards the former. The result might be, to some, a surprisingly large set of small models (Fig. 2.1), with a lot of composition requirements before anything close to useful functionality can be realised. In other words, the focus is on *reusability* of the platform creation results; the *usability* must, hence, be realised by adding *tools* and *domain-specific languages* that target specific developers, users and/or application domains. These design ambitions pose a difficult exercise to trade-off *flexibility* and *generality* on the one hand, versus *user friendliness* and *efficiency* on the other hand. Constructively critical suggestions from the community, to improve this trade-off, are sollicited throughout the duration of the RobMoSys project.

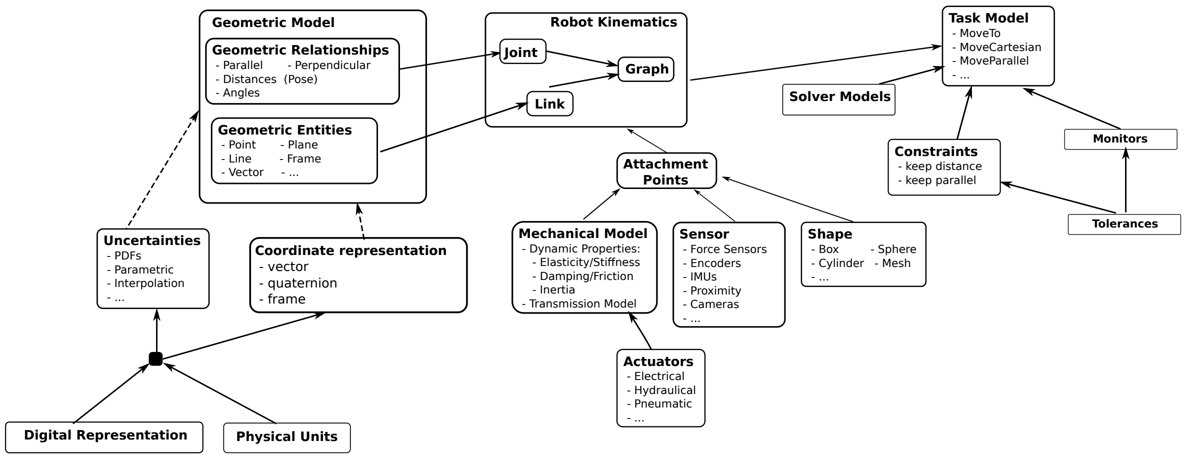


Figure 2.1: Overview of the model structure of the "motion stack". Each of the arrows represents a *composition* of complementary aspects of eventual software implementations.

Figure 2.1 gives an overview of the various models that are to be composed in any given robot application. Although there are already dozens of sub-models in the Figure, it is still incomplete: it focuses on the modelling of **kinematic chains**, on the implementations of the functionalities that come with such kinematic chains, and on the role that kinematic chains play in the specification of robotics tasks.

At the highest level of abstraction, the entities, constraints and relations connected to kinematic chains are: rigid *links* whose relative motions are constrained by *joints*, driven by *actuators* and measured by *(proprio) sensors*; a kinematic chain is an instantaneous *mechanical* energy transformation relationship between joint space and Cartesian space, which can be *redundant*, *underactuated* and/or *singular*.

Developers of robotics applications have to add a lot more concrete details to the just-mentioned abstract concepts, and the Figure structures the dependencies in the various models that are

involved: geometry, mathematical and digital representations, and physical units are necessary *data structure* whose semantics must be made 100% clear for all developers and users of the *functions* ("solvers") that implement the abstract properties of kinematic chains. The generic solver is that of the so-called *hybrid dynamics*: it computes the instantaneous joint torques required to generate the motion of the kinematic chain that is the result of a set of given Cartesian forces on some of the links, given "posture control" torques on some of the joints, and motion constraints on joints and/or links.
Important parts that are not in the Figure (but will be added to the RobMoSys project later on) are:

- **task specification**: any application requires the kinematic chain to generate motion of task-specific tools or sensors.
- **control**: specified motion must be realised via control.
- **estimation**: the desired motion of the kinematic chain is often specified relative to (the motion of) objects in the environment, whose position and motion must be estimated at runtime.
- **planning**: the motion, shape and workspace properties of the kinematic chain are important constraints to take into account in the planning of the tasks in a particular application.

3. Motion Stack Models

This chapter proposes a set of explicit models, meta-models and meta-meta models as fundamental requirements for any *composable*, *usable* and *re-usable* motion and perception stack implementations. The purpose of those models is to describe—eventually in an exhaustive and formal way—any software entity (e.g., a functionality, a component, a task specification, etc), such that motion and perception algorithms can be re-used, composed, verified and validated by means of design-time and run-time tools, used by human developers and, eventually, robot systems themselves. While the latter ambition is not for the immediate future, the effort of creating formal models is expected to be a primary contribution to the improvement of human-centered software documentation as well.
The RobMoSys project promotes the usage of explicit model instances to describe kinematic chains, physical objects, algorithms (and their properties) that conforms to a set of specific meta-models. The aim is not to propose *a single* meta-model that tries to cover all the aspects of a software implementation; instead, the aim is to propose a *composable meta-meta-model*, that is a meta-meta-model that describes all the complementary aspects of a software implementation by means of composition of several, dedicated meta-meta-models. The result is expected to become a *platform* that covers all generic aspects of robotics applications, and has the structure and the tools to allow composition of of platform-level domain-specific meta-models and software with new application-specific models and software , as long as they conform to one or more of the meta-meta-models that compose the motion and perception stack meta-meta-model.

For example, a company introduces a new actuator, kinematic chain design, or sensor processing algorithm, and the platform tools and models help that company to identify, first, exactly those properties and functionalities of its innovation that are *really* innovative and unsupported. After that, the platform supports the software development by the company in such a way that it can focus on those innovative aspects only. Of course, existing robotics software frameworks already provide a preliminary version of such platforms, but they lack in (i) formal models that can be used in tools to support the process, and (ii) sufficiently fine-grained granularity of the models and the tools to allow very customaized composition, down to the level of generating new composite compile-time algorithms (instead of "just" composing compiled software components).

The concrete models improve the overall composability and re-usability in different ways, also depending on the application and the design phases of the application itself. In general, at least the following approaches exist:

- **developer discipline** - models are used as a formal documentation, and all the development is performed manually, *in-code*, by means of previous agreements among the developers that cover a different role in the RobMoSys ecosystem (see deliverable D2.1[1]);
- **design and deployment time tools** - models are used by *offline* tools that help the developers to deliver error-free applications;
- **runtime features** - the software components provide functionalities to dynamically adapt themselves and compose with new or replaced components, at runtime. In this scenario, models are shared among the components, describing their properties and enabling runtime

[1] see http://robmosys.eu/wiki/general_principles:ecosystem:roles

introspection and (re)configuration, hence facilitating the adaptability and composability of the overall application, at runtime and by the robots themselves.

Further hints related to specific implementation details will be provided in this deliverable and during the overall duration of the project (by means of deliverables D3.2, D3.3 and D3.4).

The following Sections of this Chapter provide detailed descriptions of various sub-sets of the overall modelling picture, as sketched in Figure 2.1.

3.1 Numerical Entity Model

The numerical entity model is a composition of models that ground *concepts* with their *numerical representation* in a machine-readable form. In the context of the motion and perception stack, special attention is given to the grounding of *geometric* concepts since they are the backbone of any robotic application.

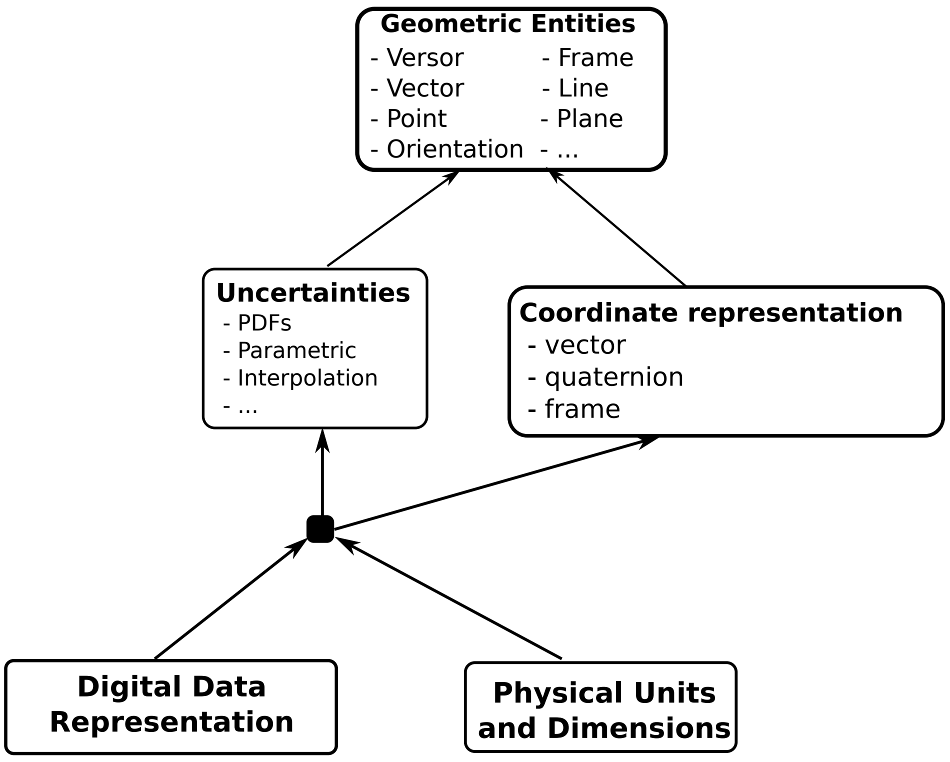


Figure 3.1: The grounding of geometric entities models by means of composition of models. Each of the arrows represents a *composition* of complementary aspects of eventual software implementations.

The grounding of a geometric entity as a numerical entity is expressed in Figure 3.1, as the composition of the following models:

- a **Coordinate Representation** model defines the mathematical representation of a geometric entity;

- a **Digital Data Representation** model specifies how numerical values are hold and managed in-memory (as part of algorithms), or in a communication object (as part of software component architectures);
- a **Physical Units and Dimensions** model to attach to the numerical values;
- a **Uncertainty** model to attach to the description of a geometric entity.

While a digital data representation is present in any implementation, the physical units and coordinate representation models are often forgotten or taken as implicit choice in the implementation itself. The RobMoSys approach considers the usage of explicit models as a necessary condition to achieve better composability and compatibility among functional software elements, especially when (i) the latter are to be developed by distributed and independently operating teams, and (ii) the end product must pass validation or certification processes, again by independent third parties.

3.2 Digital Data Representation

Definition A digital data representation model is a machine readable description about how a certain piece of information is represented and manipulated digitally, with the purpose of sharing and storing that data in all its variants (in memory, marshalling/serialising, etc).

Details Commonly, a digital data representation model is defined as a *datatype*, often constrained by the programming language used in a specific implementation. Examples are built-in types as *integer* values, *floating-points* values, *string* and their *composition*, e.g., data structures and unions in the **C** language. A digital data representation model may include specific hardware-dependent information, such as byte order (i.e. endianness), numerical precision over the representation of a number (e.g., number of bits of a floating-point value, cf. IEEE 754, Q format number over fixed-point values), bit alignment and padding information (e.g., C-structs). The level of expressivity of the precision type also depends on the language that implements a certain functionality. For instance, the **C** language is more versatile than the **JavaScript** language, 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*.
A digital data representation model may provide *constraints* over the single datatype or field, e.g., an integer value bounded within a defined range. However, a digital data representation model do not provide constraints over multiple fields, since that constraint typically depends on the *semantics* that a specific structure represents. As an example, a digital data representation of a versor (also called normalised vector) defined in a three-dimensional space can be described by means of three floating-point values constrained by having an unitary euclidean norm. Since a digital data representation model does not provide any semantics about the *meaning* of the manipulated data, the data itself should be augmented with meta-data information, providing a specific field in the data structure that holds a reference to its semantic model, or enconding (part of) the meta-data in the data structure itself for runtime reflection property. This concepts is further explained in Section 3.4, while further examples are given in Section 3.2.1.

Composability requirements *A digital data representation model is a necessary but not sufficient condition to guarantee composability and re-usability of an existing software component.* A digital data representation shows up naturally in any implementation of a software functionality, whether such an entity is a well-defined software component, an API or another interface type. Since the choice over one specific digital data representation influences the implementation of an application, the existance of an explicit digital data representation model must be considered a key-element towards composability of systems-of-systems.
The digital data representation concerns all the roles defined in the RobMoSys ecosystem, among which:

- the **Function Developer** must consider which digital data representation to use in the type signature of the developed function, class member, or abstract interface. This is particularly true for implementations based on statically typed languages;
- the **Component Supplier** must provide a digital data representation to inform about the types handled by a specific component;
- the **System Builder** should rely on predefined digital data representation chosen by domain experts, and he/she must take care that digital data representation between components are compatible, providing a transformation of the data among different models if necessary (i.e., datatype conversion).

3.2.1 Digital Representation (meta-)Models and Examples

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. In fact, 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 relationships between the data, its digital data representation model, and the meta-model used to define a specific data represention holds among the different alternatives, as depicted in Figure 3.2; a concrete example is discussed in Figure 3.3.
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;

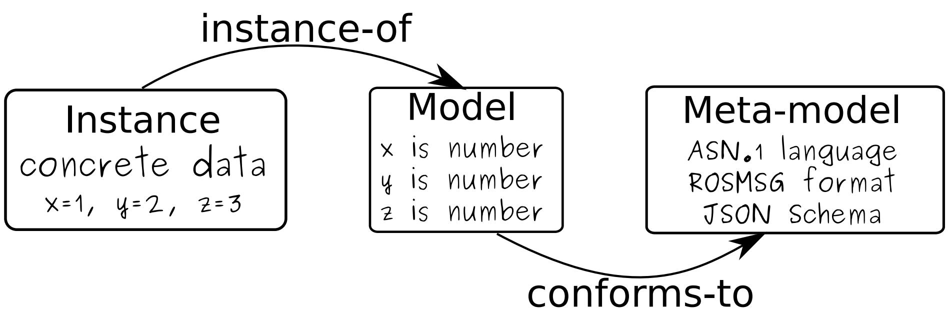


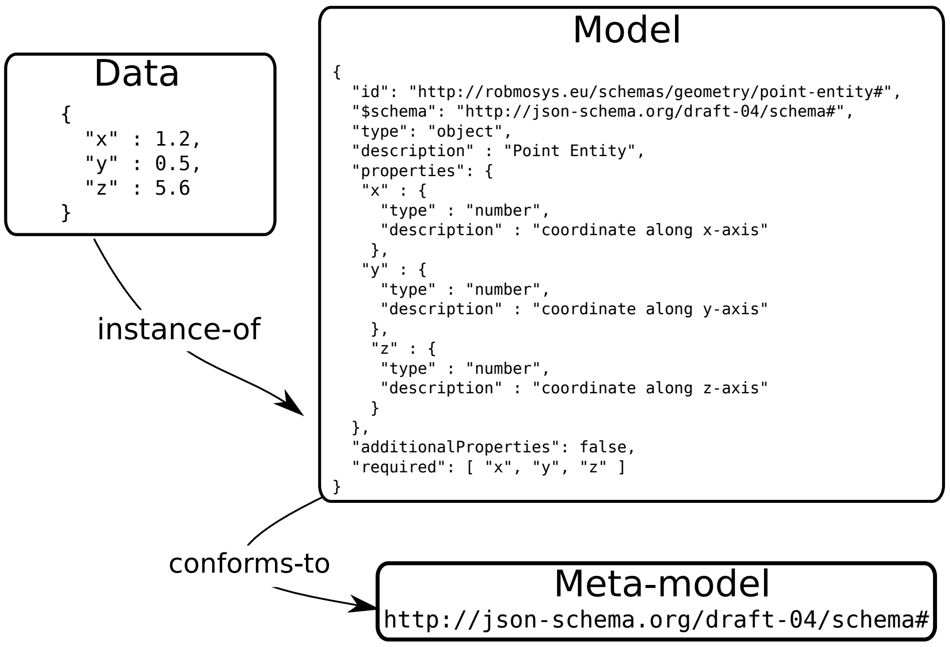
Figure 3.2: Digital Representation Models: the concrete data is instance of a digital data representation model, which is a description conforming to a meta-model. A concrete example is shown in Fig. 3.3.

- **language interoperability** (also called neutrality): the capability of a model of being language-indipendent; this requires a specific compiler to generate code-specific form of the digital data representation model;
- **self-describing**: optional capability of injecting the model in the data instance itself (metadata), or at least a reference to it; this enables reflection and run-time features.

The follow is a non-exhaustive list of those models, with a special attention to existing models used in Robotics.

Abstract Syntax Notation One (ASN.1)
ASN.1 is a IDL to define data structures in a standard, code-agnostic form, enabling the expressivity required to realise efficient cross-platform serialisation and deserialisation. 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 telecomunication 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 exists, targeting to different host programming languages, including C/C++, Python and Ada.

JSON/JSON-Schema
The JSON-Schema [1] is a schema to formally describe elements and constraints over a JavaScript Object Notation (JSON) document [14]. 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 3.3.
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



```
{
  "x" : 1.2,
  "y" : 0.5,
  "z" : 5.6
}
```

```
{
  "id": "http://robmosys.eu/schemas/geometry/point-entity#",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "description" : "Point Entity",
  "properties": {
   "x" : {
     "type" : "number",
     "description" : "coordinate along x-axis"
    },
   "y" : {
     "type" : "number",
     "description" : "coordinate along y-axis"
    },
    "z" : {
     "type" : "number",
     "description" : "coordinate along z-axis"
    }
  },
  "additionalProperties": false,
  "required": [ "x", "y", "z" ]
}
```

```
http://json-schema.org/draft-04/schema#
```

Figure 3.3: 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).

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 primivites 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 3.3 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.

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*).

Hierarchical Data Format (HDF5)

HDF5 [28] is a data model (and relative reference implementation) designed for large, distributed datasets and efficient I/O storage. Mainly adopted for scientific data storage and analysis, HDF5 implementation allows to select, filter and collect specific information from the distributed data storage thanks to its model. In details, a HDF5 model can be distributed as attachment of the data, specifying its provenance as well. Currently HDF5 is a technology not widely used in the Robotics domain, but its model meets those requirements of composability needed for both storage systems and in-memory datasets.

Google Protocol Buffers

Google protocol buffers are another digital data representation meta-model dedicated for serialising structured data. The features that distiguish the Google protocol buffers from other solutions are: (i) efficient encoding mechanism to have a small impact in terms of memory required (or message size); (ii) an optional mechanism to include self-describing messages, that is the digital data representation model can be packed together with the data (instance of the model). Moreover, the availability of dedicated compilers allows the support to a large variety of host programming languages, while the extensibility is in line with other alternatives.

ROS Messages

ROS message is a digital data representation meta-model developed for the ROS framework, aimed to describe structural data for serialisation and deserialisation within the ROS communication protocol, namely ROS topics, services and actions. Precisely, the (non formalised) meta-model is strongly coupled with the chosen ROS communication pattern:

- ROS messages (`msg` format) for streamed publish/subscribe ROS topics;
- ROS services (`srv` format) for blocking request/reply over ROS;
- ROS actions (`action` format) for ROS action pattern.

The need of having a different data model for each communication mechanism provided reduces the degree of composability of the overall system, enforcing the component supplier to a premature choice. However, it is possible to compose message specifications from existing ones, such as services and action models are built starting from messages models. The expressivity of a ROS message description is limited with respect to other alternatives (e.g., ASN.1, JSON-Schema): it allows to specify different types for numerical representations, (e.g., `Float32`, `Float64` for floating-point values) but there is no support for constrains over a numerical value, nor specific padding and alignment information. Moreover, there is no built-in enumeration values, which is usually solved with few workarounds[2]. However, default values assignment is possible in the ROS message models. ROS messages are self-describing by means of a generated ID (MD5Sum) based on a naming convention schema of the message name definition and a namespace (package of origin). Language-neutrality is provided by the several compilers available within the ROS framework. However, there is no efficient encoding mechanism applied, reducing the compilation process to a mere generation of handler classes for the host programming language. Despite the technical shortcomings of the ROS messages, they are the likely most used digital data representation model in the robotics domain, due to the large diffusion of the ROS framework (which does not allow another data representation mechanism[3]).

RTT/Orocos typekits

The RTT/Orocos typekits 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 representaiton model if a dedicated tool is supplied. For example, tools that generate a typekit starting from a ROS message definition exists.

[2]An `UInt8` type with unique default value assigned for each enumeration item.

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

SmartSoft Communication Object DSL
The SmartSoft framework provides a specific DSL based on the Eclipse Xtext to describe a digital data representation for the definition of data structures. This DSL allows 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*[4] (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.

3.2.2 Why digital data representation models are important?

Any software implementation makes use of a digital data representation, often implicitly defined as a language-dependent data structure used in the application. An *explicit*, *language-independent model* of a digital data representation enables better *composability* and *re-usability* of any software entity (e.g., functions, components, kinematic models, task specifications, etc) by means of the following approaches:

- **developer discipline:** digital data representation models serve as documentation for developers (mainly system builder and component suppliers[5]). The developers agree upon one (or more) digital data representation to be used in an application. Composability is possible but tedious and error-prone: for any change in the application (e.g., replacing a component with another providing similar functionalities), developers must check the used digital data representation *manually*, along with the required glue-code in case that a data conversion is needed;
- **design-time tooling:** digital data representation models can be used effectively to perform many tasks otherwise obtained by developer discipline, among which datatype compatibility checks and glue-code generation for datatype conversions. This approach is less error-prone than an approach driven by developer discipline, and it enables automatic unit testing, so it is a step towards software certification of an application. Moreover, the tools may allow different forms of *optimisation*, as a compromise between the resources required (e.g., required memory and computational power) and efficiency with respect to the concrete use of the data (e.g., different models can be used for communication[6], storage, or for computational purposes). However, optimisations and choices taken at design-time (or deployment-time) are *static*, thus they limit further composability at runtime.
- **runtime negotation:** at runtime, two (or more) software entities (i.e., component instances) initialise a negotation phase to decide which digital data representation to use. This approach represents the ultimate interpretation of flexibility and composability, sometimes despite the resource requirements necessary for the negotation phase. This approach should be adopted if the application requires composability at runtime, and in those cases where a design decision could not have been taken upfront at design-time. Moreover, this scenario implies that both software entities uses the same protocol/middleware for the negotation phase.

[4] see http://www.cs.wustl.edu/~schmidt/ACE.html
[5] see http://robmosys.eu/wiki/general_principles:ecosystem:roles
[6]see http://robmosys.eu/wiki/modeling:metamodels:commobject

3.3 Physical Units and Dimensions Model

The digital data representation (see Section 3.2) expresses only how a set of numerical values are managed, in-memory, but still no semantics representation is attached to the values. A physical units and dimensions model is a first of those models that give semantic information over plain data. The correct usage of this models enables the following features, which are necessary composability conditions for any motion and perception stack:

- **unit compatibility**, which consist in checking the compatibility over the data shared between functional components, if possible, to perform the necessary conversions that guarantee compatibility;
- **dimensional analysis** over the declared inputs and outputs of a functional component model.

The RobMoSys project promotes the use of ontologies to describe physical units and dimensions. In particular, the RobMoSys consortium suggests as primary reference the *QUDT* ontology [31] by NASA; however, equivalent ontologies are also accepted.

3.3.1 Role of the physical units and dimensions

This model can assume different roles, depending on the development approach:

- **developer discipline**: there is an agreement between the component suppliers, and for each datatype/communication object exists an informative documentation about the physical units and dimensions adopted. To reduce complexity, it is common practice to use the same units and dimensions where applicable in the whole application. However, this makes the system highly not composable, since each functional component must be manually validated before being used. Sometimes this requires extra development efforts to realise the necessary conversions.
- **tooling**: it is possible to perform checks over connected components automatically, and generate glue-code for conversions, if applicable. Moreover, this is a requirement towards code certifications and reduces the possibility to introduce a bug in the underlying implementation.
- **runtime adaptation**, possible if the digital data representation includes meta-data about the physical units and dimension used. In this scenario, the component implements several conversion strategies to adapt the incoming/outcoming data to its internal functionalities.

As a final remark, an explicit physical units and dimensions model enables a sort of *numerical localisation*, analogous to *language localisation* features (e.g., i18n), most of the time available in a software product.

3.4 Coordinate Representation

Coordinate representations express how a geometric concept is mathematically represented, assigning to a concept a symbol value, a vector or a matrix with a semantic value. In the contex of a motion and perception stack, the basic geometric concepts are those *relationships* that describe a motion over physical bodies, such as: position, orientation, pose, linear and angular velocities, torques, forces, etc. However, there is no unique coordinate reprentation that describes the same geometric semantics. This implies that the (sometimes) arbitrary choices over the coordinate representation can be source of uncompatibility and composability issues over a complex system of systems. Moreover, the choice over the coordinate representation can include additional constraints on the semantics of geometric relation itself; this is the case for *rotation matrixes* and *homogeneous transformation matrixes* [15].

Table 3.1 resumes common choices of coordinate representation indexed by the coordinate semantics that they express. Those geometric relationships, backbones of the motion and perception stack, are further formalised and discussed in Section 3.7 and [15].

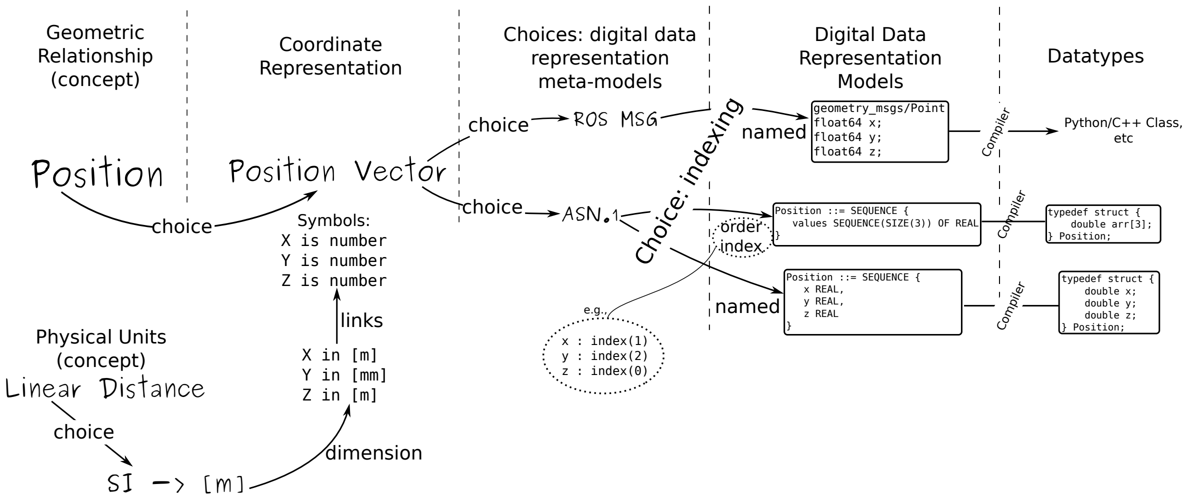
Coordinate Semantics	**Coordinate Representation**
Position	Position vector
Orientation	Euler-axis angle Rotation vector Rotation matrix Euler angles RPY-angles Quaternions
Pose	Homogeneous transformation matrix Finite displacement twist Screw axis
Linear Velocity	Linear velocity vector
Angular Velocity	Angluar velocity vector Rotation matrix time derivative Euler angle rates RPY angle rates Quaternion rates
Twist	Homogeneous transformation matrix time derivative six-dimensional vector twist Pose twist Screw twist Body-fixed twist Instantaneous screw axis

Table 3.1: A collection of commonly used coordinate representations associated to relative coordinate semantics.

3.5 Coordinate Representation, digital data representation and physical Units

Similarly to the relationship between coordinate semantics and coordinate representation, a coordinate representation has no unique digital data representation, indipendently from the meta-model used to specify the digital data representation model. This variability is related on the *ordering* of the values of the coordinate representation, which is decoupled from how the data access is performed, e.g., by means of a *named* key or by means of an *anonymous* indexing.

Moreover, the concept of the geometric relationship is associated to one (or more) physical unit representation (e.g, SI or imperial system), and each single value must be interpreted accordingly to the associated dimension.



```
geometry_msgs/Point
float64 x;
float64 y;
float64 z;
```

```
Position ::= SEQUENCE {
   values SEQUENCE(SIZE(3)) OF REAL
}
```

```
Position ::= SEQUENCE {
   x REAL,
   y REAL,
   z REAL
}
```

```
typedef struct {
    double arr[3];
} Position;
```

```
typedef struct {
    double x;
    double y;
    double z;
} Position;
```

Figure 3.4: From geometric relationship to digital data representation: choices for the grounding of the concept of *Position*.

The grounding of a geometric concept as *Position* is a sum of multiple choices, as shown in Figure 3.4:

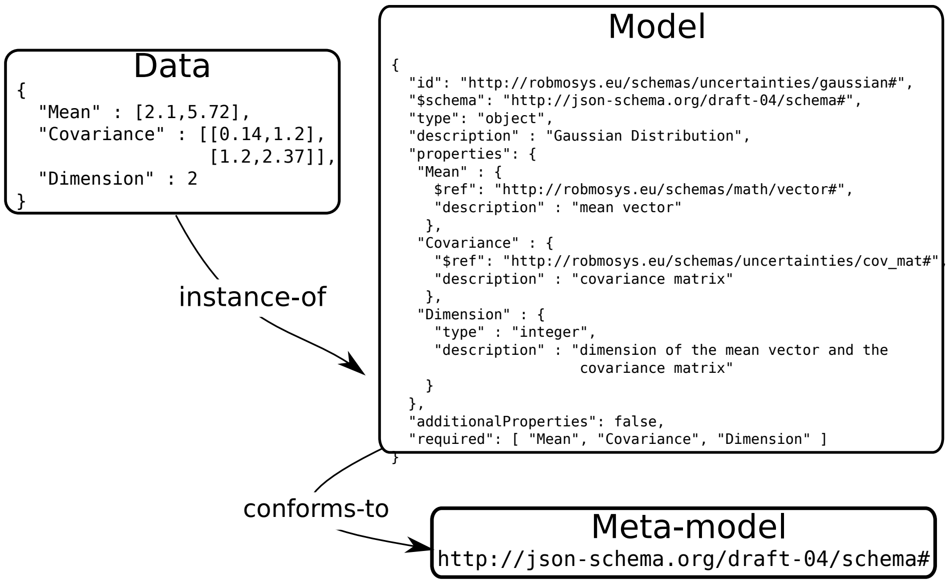
- **which coordinate representation to use, and its symbolic definition?** For *position* relationship, the choice is unique and it is a *position vector*, defined as 3 numerical symbols x y and z. In general, this choice is not unique, as shown in Table 3.1;
- **which system measurement to adopt, and which unit to associate with?** A *position* covers the concept of *linear distance*, which is measured in meters if the SI is adopted; this choice is not unique;
- **which dimensions the numerical values represent?** In general, this choice is made by convenience of the specific application, for example: (i) an application can be indoor or outdoor; (ii) precision required (by the task), or (iii) provided by the measurements (combination of sensing and estimation);
- **which digital data representation meta-model to use?** This choice opens up on different possibilities, and it is a convenient choice which depends on the framework/middleware in which the functional component is targeted;
- **Choice on indexing: how to index a particular data?** e.g, by named key, ordered values, etc;
- **Digital data representation: how the data is managed?**

All the above are concrete questions that a component supplier, system builder and application developer answers, implicitly or explicitly, when developing a new functionality, or deploying an existing component in an application-dependent architecture. Solving those by discipline for any change in the application is thus tedious and cumbersome, and it is one of the main reasons that the RobMoSys consorium promotes model-driven engineering to enable a systematic and error-free solutions.

3.6 Models of Uncertainties

The purpose of an uncertainty model is to describe the variability over the numerical entity representation of a geometric entity. Similar to the geometric entities it provides semantic structures (e.g. probability distributions and their constraints), which are then grounded in an numerical representation composed from a digital data representation and a chosen physical unit and dimensions.

An example would be a Gaussian distribution, which can be numerically represented by its mean vector and its covariance matrix (see Figure 3.5). Please note, that this model is also a composition of mathematical models (vector and matrix) that are defined in other JSON Schemas. Additional constraints, like that the covariance matrix must be symmetric, are not modelled in this example. 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.



```
{
  "Mean" : [2.1,5.72],
  "Covariance" : [[0.14,1.2],
                  [1.2,2.37]],
  "Dimension" : 2
}
```

```
{
  "id": "http://robmosys.eu/schemas/uncertainties/gaussian#",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "description" : "Gaussian Distribution",
  "properties": {
   "Mean" : {
     $ref": "http://robmosys.eu/schemas/math/vector#",
     "description" : "mean vector"
    },
   "Covariance" : {
     "$ref": "http://robmosys.eu/schemas/uncertainties/cov_mat#",
     "description" : "covariance matrix"
    },
   "Dimension" : {
     "type" : "integer",
     "description" : "dimension of the mean vector and the
                      covariance matrix"
    }
  },
  "additionalProperties": false,
  "required": [ "Mean", "Covariance", "Dimension" ]
}
```

```
http://json-schema.org/draft-04/schema#
```

Figure 3.5: 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 (draft-04).

Sources of uncertainties can be, but are not limited to:

- **uncertainties by construction**, which are those uncertainties that represent approximations over the description of the geometric entity attached to it. This is the case for mechanical constraints, such as the coupling tolerance in a joint;

- **sensor noise**, which is a property of the sensor but can be influenced by other factors such as environmental condition or robot motions;

- **process noise**, which represents the uncertainty in the modelling of a behaviour.

3.7 Geometric Semantics of Relations between Rigid Bodies

Physical bodies and their relationships assume a primary role within the motion and perception stack, defining the semantics of geometric relations such as relative position, orientation, pose, *etc*. Those are used in many models, among which:

- **Kinematic Chain Models**, where *links* are physical rigid bodies having a *mechanical* motion constraint between each other (i.e., *the joints*) whose ideal properties are well represented by geometric relations;
- **Task Specifications**, where geometric relations are primary ways to express the intention of the robots' motion as a set of *artificial* motion constraints. A typical example is a positioning behaviour, i.e., a task defined as a to-be-obtained relative position between two objects in the environment;
- in the **World Models**, where geometric relations add *metric* information to the *topological* information of connectivity. For example, a topological description such as "the ball is contained inside the box" can be grounded in sensor processing algorithms as a geometric relation of type *position*, constraining the position of the ball within the box dimensions.

It is important that the models of geometric entities and their relations add semantic interpretation to their *metric* information. This means that geometric relations must also be expressed in a symbolic form, with links to (one of the many possible) numerical entity models whenever a *numerical evaluation* of the expressed metric is required. To this end, it is relevant to define the semantics of commonly used geometric relationships, thus removing any ambiguity sources related to a different coordinate representation or implicit definitions.
The overview presented in this section updates the work presented in [15], which proposes a standardisation for expressing the semantics of geometric relations, limited to rigid bodies. Moreover, the definitions of geometric primitives are extended considering the work presented in [12]. Further details about the overview presented in this section can be found in [15, 12].

3.7.1 Geometric Primitives

The following primitives and their notions are defined by means of *axioms*, which are valid in the context of *Euclidean geometry* (three-dimensional Euclidean space). For the sake of clarity, a nomenclature and conventions are proposed, but they are not unique; other equivalent solutions are in fact possible.

Spatial Point

A spatial point refers to an element which belongs to an Euclidean space. Points are adimensional, and as such they have no geometric or physical properties, such as a volume, shape, area, length, and weight. A three-dimensional point is represented mathematically by means of three free parameters, typically represented numerically by an ordered triplet of scalars (x, y, z), and this is called its "coordinate representation". For the sake of clarity, the symbols $\boldsymbol{a}, \boldsymbol{b}, \ldots$ are used to indicate spatial points.

Vector

A vector is a geometric primitive that connects a point $source$ to a point $target$. A vector has *(i)* a magnitude (distance between the points), *(ii)* and a direction (from $source$ to $target$).

There is no unique coordinate representation of a vector. For example, a vector can be represented by its $source$ point, magnitude and direction (e.g., a versor), without the need of specifying its $target$ point; in the same vein, the $source$ point can be omitted if the $target$ point is given.

Versor
A versor (or *normalised vector*) represents a direction. In a three-dimensional Euclidean space, a possible coordinate representation is an ordered tripled of scalars (x, y, z) such that their Euclidean norm is unitary (i.e., $||\cdot|| = 1$). A suggested symbol to represent a versor is $\widehat{\boldsymbol{n}}$.

Orientation Frame
An orientation frame represents an orientation by means of three orthonormal vectors which denote the frame's X-axis X, Y-axis Y and Z-axis Z. Multiple coordinate representations exists. For sale of clarity, the symobols $[a], [b], \ldots$ are used to denote orientation frames.

Displacement Frame
A displacement frame is a *composite* primitive that represents an orientation and position, by means of an orientation frame and a point; the latter acts as orientation's frame origin). For the sake of clarity, the symbols $\{a\}, \{b\}, \ldots$ indicate displacement frames.

Line
A (straight, one-dimensional) line is another well-known composite geometric concept with axiomatic connotations. The ambiguity not only lies in the implicit assumption of the chosen coordinate representation, but also on the geometric primitives chosen to define a line. For example, a couple of points define uniquely a line, as well as a point defined as "origin" and a versor that indicates the direction where the line lies. In the latter case, an implicit information is added (i.e., the line is directed), which can be an handful representation in certain applications (e.g., to express constraints for a task specification, see Section 3.11). As a coordinate representation, a line defined in the Euclidean space can be expressed by means of the parameters of a system of parametric equations (three equations in a three-dimensional Euclidean space, or two constrained equations), but also by a non-minimal choice of five free parameters (this is the case if the line is defined by a origin point and a direction); other equivalent solutions are possible. A suggested symbol to represent a line is $\bar{\boldsymbol{l}}$.

Plane
A plane is a geometric concept that indicates a flat surface described as two-dimensional geometric element. In fact, a plane can be interpreted as two-dimensional version of a line, and as such, the observations reported about the line applies to a plane as well. A possible description can be formulated as composition of a point (origin) and a versor; the latter denotes a *normal* (and not a direction, as reported for the line). A suggested symbol to represent a plane is $\mathfrak{B}$.

Rigid Body The term "body" denotes a solid, physical object, which is formally represented by the following (equivalent) relations:

- deformations are neglected, that is, the distance between any of its points remains invariant over time;
- the *pose* relation between any of the points *fixed* to the body is constant.

The latter statement used the concept of *pose* relation, which is modelled later in Section 3.7.2. For the sake of clarity, the symbols $\mathcal{A}, \mathcal{B}, \ldots$ are used to denote rigid bodies.

All the geometric primitives above are *relative* concepts, that is they are not fully defined without a definition of a *geometric reference*. For example, a point is not completely defined without a displacement frame used as a reference (also called coordinate frame); in this case, the symbol $\boldsymbol{p}_{\{w\}}$ indicates that a point $\boldsymbol{p}$ is defined upon the coordinate frame $\{w\}$. Moreover, a body can serve as geometric reference as well, that is a point can be arbitrarily attached to a specific body instance; in this case, the symbol $\boldsymbol{p}|\mathcal{A}$ denotes that the point p is attached to the body $\mathcal{A}$.

3.7.2 Geometric Relations

This section resumes some of the most useful geometric relations that are possible to express by means of the geometric primitives listed in Section 3.7.1. In particular, geometric relations between rigid bodies are reported in Figure 3.6 (excerpt from [15]), while other "body-free" relationships are extracted from [12], including higher order relationships and metric definitions such as linear and angular relative distances between line-point and versor-plane primitives (see Section 3.7.2.) The following description helps to provide a semantic meaning of these relationships, and to propose a symbolic representation.

Position
A *position* expresses a relative metric (*distance*) between two points, and this denoted with $\underline{\text{Position }(\boldsymbol{e}, \boldsymbol{f})}$, where $\boldsymbol{e}$ and $\boldsymbol{f}$ are point primitives; position relationship is *directed*, that is, the previous notation is semantically equivalent of "position of point $\boldsymbol{e}$ *from* point $\boldsymbol{f}$". If those two points are attached to different rigid bodies, namely $\mathcal{C}$ and $\mathcal{D}$, then the relative position between the rigid bodies can be expressed by means of the attached points, and this is denoted with $\underline{\text{Position }(\boldsymbol{e}|\mathcal{C}, \boldsymbol{f}|\mathcal{D})}$. Moreover, an orientation frame $[r]$, instantaneously fixed to a *reference body* (an arbitrary choice between the two bodies $\mathcal{C}$ and $\mathcal{D}$) is required, such that the coordinates can be expressed with respect to a *coordinate frame*; this is denoted as $\underline{\text{PositionCoord }(\boldsymbol{e}|\mathcal{C}, \boldsymbol{f}|\mathcal{D}, [r])}$.

Orientation
Similarly to the position, an *orientation* is a relative metric between two orientation frames, e.g., $[a]$ and $[b]$, and it is expessed as $\underline{\text{Orientation}([a], [b])}$. When each of the orientation frames is attached to a different rigid body, namely $\mathcal{C}$ and $\mathcal{D}$, the relationship also expresses a relative orientation between the two rigid bodies, and it is denoted with $\underline{\text{Orientation }([a]|\mathcal{C}, [b]|\mathcal{D})}$; instead $\underline{\text{OrientationCoord }(\boldsymbol{e}|\mathcal{C}, \boldsymbol{f}|\mathcal{D}, [r])}$ expresses explicitly the chosen orientation frame $[r]$ to express the coordinates in the coordinate frame.

Pose
A *pose* is a geometric relationship that expresses a *composite* metric between two displacement frames, e.g., $\{g\}$ and $\{h\}$, or between an equivalent couple of pairs of position and orientation primitives, e.g., $(\boldsymbol{e}, [a])$ and $(\boldsymbol{f}, [b])$; this is denoted with $\underline{\text{Pose}(\{g\}, \{h\})}$ or $\underline{\text{Pose}((\boldsymbol{e}, [a]), (\boldsymbol{f}, [b]))}$. If those primitives are attached to different rigid bodies $\mathcal{C}$ and $\mathcal{D}$, the pose relationship between the two rigid bodies is expressed as $\underline{\text{Pose }(\{g\}|\mathcal{C}, \{h\}|\mathcal{D})}$ or $\underline{\text{Pose }((\boldsymbol{e}, [a])|\mathcal{C}, (\boldsymbol{f}, [b])|\mathcal{D})}$. The choice over a coordinate representation imposes an explicit notation of the orientation frame [r] used to express the coordinates in the coordinate frame, that is $\underline{\text{Pose }(\{g\}|\mathcal{C}, \{h\}|\mathcal{D}, [r])}$ or $\underline{\text{Pose }((\boldsymbol{e}, [a])|\mathcal{C}, (\boldsymbol{f}, [b])|\mathcal{D}, [r])}$.

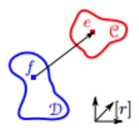
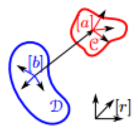
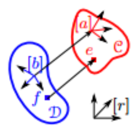
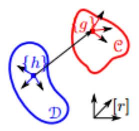
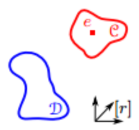
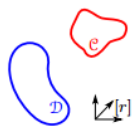
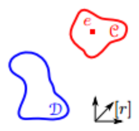
Geometric Relation	(Coordinate) semantics	Geometric primitives	Graphical representation
Position	Position $(e\|\mathcal{C}, f\|\mathcal{D})$ PositionCoord $(e\|\mathcal{C}, f\|\mathcal{D}, [r])$	point e body $\mathcal{C}$ reference point f reference body $\mathcal{D}$ coordinate frame $[r]$	
Orientation	Orientation $([a]\|\mathcal{C}, [b]\|\mathcal{D})$ OrientationCoord $([a]\|\mathcal{C}, [b]\|\mathcal{D}, [r])$	orientation frame $[a]$ body $\mathcal{C}$ reference orientation frame $[b]$ reference body $\mathcal{D}$ coordinate frame $[r]$	
Pose	Pose $((e, [a])\|\mathcal{C}, (f, [b])\|\mathcal{D})$ PoseCoord $((e, [a])\|\mathcal{C}, (f, [b])\|\mathcal{D}, [r])$	point e orientation frame $[a]$ body $\mathcal{C}$ reference point f reference orientation frame $[b]$ reference body $\mathcal{D}$ coordinate frame $[r]$	
	Pose $(\{g\}\|\mathcal{C}, \{h\}\|\mathcal{D})$ PoseCoord $(\{g\}\|\mathcal{C}, \{h\}\|\mathcal{D}, [r])$	frame $\{g\}$ body $\mathcal{C}$ frame $\{h\}$ reference body $\mathcal{D}$ coordinate frame $[r]$	
Linear velocity	LinearVelocity $(e\|\mathcal{C}, \mathcal{D})$ LinearVelocityCoord $(e\|\mathcal{C}, \mathcal{D}, [r])$	point e body $\mathcal{C}$ reference body $\mathcal{D}$ coordinate frame $[r]$	
Angular velocity	AngularVelocity $(\mathcal{C}, \mathcal{D})$ AngularVelocityCoord $(\mathcal{C}, \mathcal{D}, [r])$	body $\mathcal{C}$ reference body $\mathcal{D}$ coordinate frame $[r]$	
Twist	Twist $(e\|\mathcal{C}, \mathcal{D})$ TwistCoord $(e\|\mathcal{C}, \mathcal{D}, [r])$	point e body $\mathcal{C}$ reference body $\mathcal{D}$ coordinate frame $[r]$	

Figure 3.6: Minimal semantics and coordinate semantics (expressed in coordinate frame $[r]$) between the two rigid bodies $\mathcal{C}$ and $\mathcal{D}$, including the minimal but complete set of geometric primitives for position, orientation, pose, twist, linear and angular velocity, and their graphical representation. This is an excerpt from [15].

Linear Velocity

A *linear velocity* is a relationship expressed between two point primivites, that is $\underline{\text{LinearVelocity}(\boldsymbol{e}, \boldsymbol{f})}$. In case that the points are fixed to different rigid bodies $\mathcal{C}$ and $\mathcal{D}$, then the velocity between the two rigid bodies is denoted as $\underline{\text{LinearVelocity}(\boldsymbol{e}|\mathcal{C}, \mathcal{D})}$; in this case, the specific point $\boldsymbol{f}$ fixed to $\mathcal{D}$ (i.e., the reference body) is not indicated since any choice of the point primitive is arbitrary and equivalent, as long as it is fixed to $\mathcal{D}$.

Angular Velocity

An *angular velocity* is a relationship between two orientation frames, expressing their relative

Table 3.2: Summary of linear distance relationships between point and line primitives.

	point	line
point	point-point distance	
line	line-point distance projection of point on line	distance btw lines projection (p1-f1) projection (p2-f2)
plane	point-plane distance	

Table 3.3: Summary of angular distances relationships between versor and plane primitives.

	versor	plane
versor	angle btw versors	
plane	incident angle	angle btw planes

angular velocity, that is AngularVelocity($[a], [b]$). Semantically, the angular velocity between two rigid bodies is simply indicated as AngularVelocity($\mathcal{C}, \mathcal{D}$), since it is invariant from the choice of the orientation frames fixed to each rigid body. The notation AngularVelocityCoord($\mathcal{C}, \mathcal{D}, [r]$) indicates the choice over the coordinate orientation $[r]$ to express the coordinates.

Twist

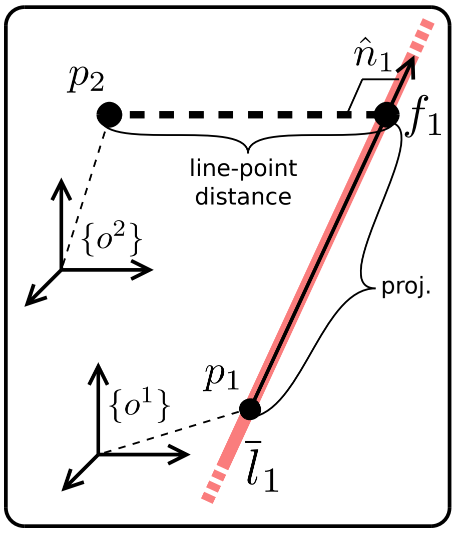
Similarly to the pose relationship, a *twist* relationship combines the semantics of the linear and angular velocities. For example, a twist between two rigid bodies is denoted as Twist($\boldsymbol{e}|\mathcal{C}, \mathcal{D}$), and it combines a LinearVelocity($\boldsymbol{e}|\mathcal{C}, \mathcal{D}$) and an AngularVelocity($\mathcal{C}, \mathcal{D}$). The notation Twist($\boldsymbol{e}|\mathcal{C}, \mathcal{D}, [r]$) indicates the choice over the coordinate orientation $[r]$ to express the coordinates.

Forces, Torques and Wrenches

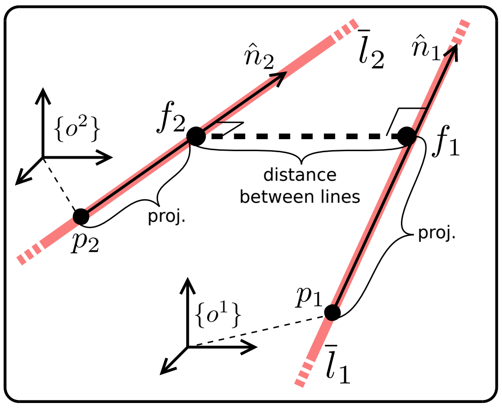
From screw theory and algebra of pairs of vectors, there is an analogy between twists, which are composed by linear and angular velocities and wrenches, which are composed by force and torque vectors. This analogy reflects directly their semantics and the relative notation, that is: Force($\boldsymbol{e}|\mathcal{C}, \mathcal{D}$), Torque($\mathcal{C}, \mathcal{D}$), and Wrench($\boldsymbol{e}|\mathcal{C}, \mathcal{D}$).

Linear and Angular Distances

Another useful geometric relationship is the concept of *distance*, which is a *higher-order relationship* among the primitives/relationships described above. In fact, the concept of distance implies a definition of a *metric* that holds within the Euclidean space; while (linear) distances between points and (angular) distances between orientation frames are uniquely defined, this is not the case between poses. Moreover, there might be ambiguities related to different interpretations of a distance relationship between certain primitives. An example is depicted in Figure 3.7a: a distance between a point $\boldsymbol{p}_2$ and a line $\bar{\boldsymbol{l}}_1$ can be interpreted as *shortest* distance between $\boldsymbol{p}_2$ and another point lying on $\bar{\boldsymbol{l}}_1$, or as length of the projection of $\boldsymbol{p}_2$ on $\bar{\boldsymbol{l}}_1$. To remove any ambiguity, Table 3.2 and Table 3.3 resume the different interpretations of linear distances and angular distances, respectively, considering some geometric primitives enumerated in Section 3.7.1; these relationships are also shown in Figure 3.7.



(a) Line 1 is expressed in $\{o^1\}$, Point 2 in $\{o^2\}$.

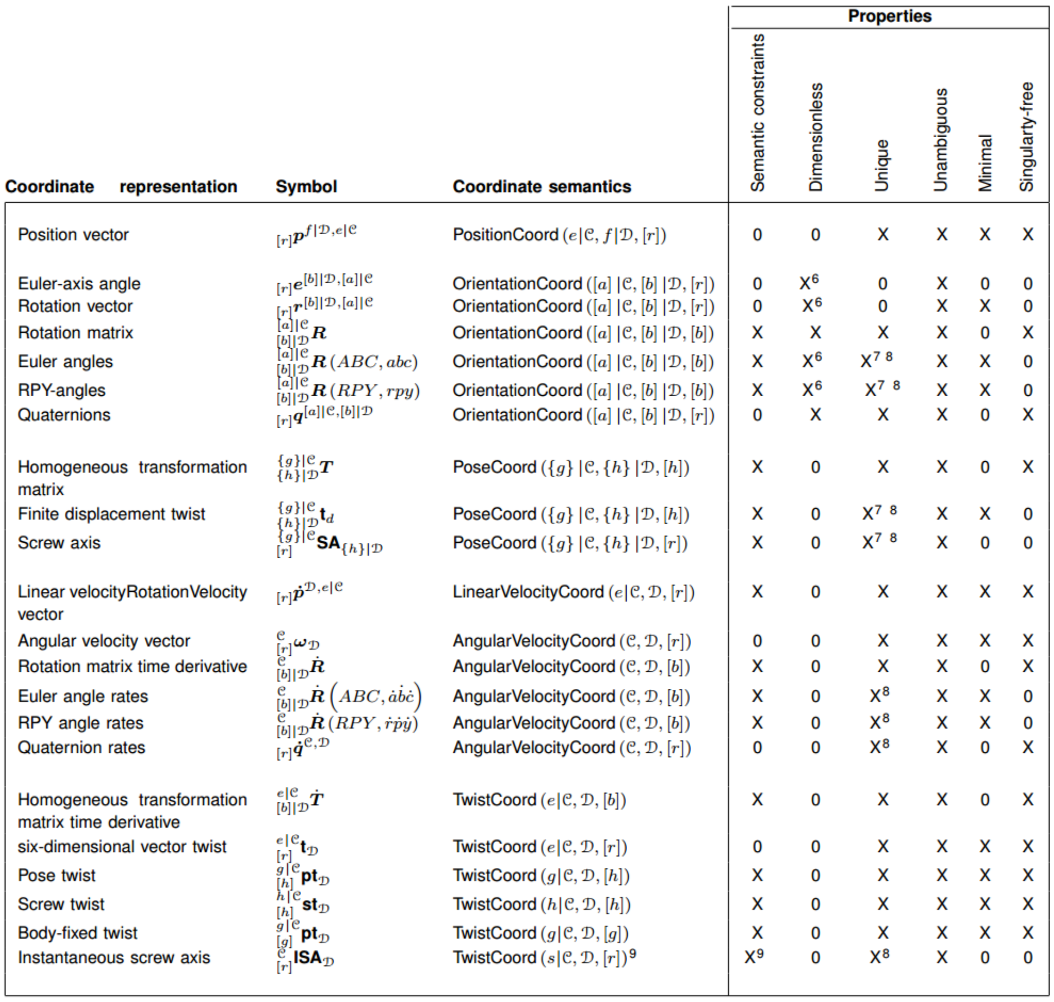


(b) Lines 1 and 2 are expressed in $\{o^1\}$ and $\{o^2\}$, respectively.

Figure 3.7: Graphical representations of the five possible relations between a point and a line 3.7a, and between two lines 3.7b.

3.7.3 Coordinate Representations for Geometric Semantics

As reported in Section 3.4, the choice of a coordinate representation is not unique for expressing a geometric concept. Moreover, this choice often implies hidden assumptions or ambiguities. To this end, Figure 3.8 resumes most of commonly used coordinate representations and their properties, such as uniqueness of the representation, ambiguity, introduction of singularity representation, minimality and more. Further details can be found in [15].



Coordinate representation	Symbol	Coordinate semantics	Properties					
			Semantic constraints	Dimensionless	Unique	Unambiguous	Minimal	Singularity-free
Position vector	${}_{[r]}\boldsymbol{p}^{f\|\mathcal{D},e\|\mathcal{C}}$	PositionCoord $(e\|\mathcal{C}, f\|\mathcal{D}, [r])$	0	0	X	X	X	X
Euler-axis angle	${}_{[r]}\boldsymbol{e}^{[b]\|\mathcal{D},[a]\|\mathcal{C}}$	OrientationCoord $([a]\|\mathcal{C}, [b]\|\mathcal{D}, [r])$	0	X[6]	0	X	0	0
Rotation vector	${}_{[r]}\boldsymbol{r}^{[b]\|\mathcal{D},[a]\|\mathcal{C}}$	OrientationCoord $([a]\|\mathcal{C}, [b]\|\mathcal{D}, [r])$	0	X[6]	0	X	X	0
Rotation matrix	${}^{[a]\|\mathcal{C}}_{[b]\|\mathcal{D}}\boldsymbol{R}$	OrientationCoord $([a]\|\mathcal{C}, [b]\|\mathcal{D}, [b])$	X	X	X	X	0	X
Euler angles	${}^{[a]\|\mathcal{C}}_{[b]\|\mathcal{D}}\boldsymbol{R}\,(ABC, abc)$	OrientationCoord $([a]\|\mathcal{C}, [b]\|\mathcal{D}, [b])$	X	X[6]	X[7 8]	X	X	0
RPY-angles	${}^{[a]\|\mathcal{C}}_{[b]\|\mathcal{D}}\boldsymbol{R}\,(RPY, rpy)$	OrientationCoord $([a]\|\mathcal{C}, [b]\|\mathcal{D}, [b])$	X	X[6]	X[7 8]	X	X	0
Quaternions	${}_{[r]}\boldsymbol{q}^{[a]\|\mathcal{C},[b]\|\mathcal{D}}$	OrientationCoord $([a]\|\mathcal{C}, [b]\|\mathcal{D}, [r])$	0	X	X	X	0	X
Homogeneous transformation matrix	${}^{\{g\}\|\mathcal{C}}_{\{h\}\|\mathcal{D}}\boldsymbol{T}$	PoseCoord $(\{g\}\|\mathcal{C}, \{h\}\|\mathcal{D}, [h])$	X	0	X	X	0	X
Finite displacement twist	${}^{\{g\}\|\mathcal{C}}_{\{h\}\|\mathcal{D}}\mathbf{t}_d$	PoseCoord $(\{g\}\|\mathcal{C}, \{h\}\|\mathcal{D}, [h])$	X	0	X[7 8]	X	X	0
Screw axis	${}^{\{g\}\|\mathcal{C}}_{[r]}\mathbf{SA}_{\{h\}\|\mathcal{D}}$	PoseCoord $(\{g\}\|\mathcal{C}, \{h\}\|\mathcal{D}, [r])$	X	0	X[7 8]	X	0	0
Linear velocityRotationVelocity vector	${}_{[r]}\dot{\boldsymbol{p}}^{\mathcal{D},e\|\mathcal{C}}$	LinearVelocityCoord $(e\|\mathcal{C}, \mathcal{D}, [r])$	X	0	X	X	X	X
Angular velocity vector	${}^{\mathcal{C}}_{[r]}\boldsymbol{\omega}_{\mathcal{D}}$	AngularVelocityCoord $(\mathcal{C}, \mathcal{D}, [r])$	0	0	X	X	X	X
Rotation matrix time derivative	${}^{\mathcal{C}}_{[b]\|\mathcal{D}}\dot{\boldsymbol{R}}$	AngularVelocityCoord $(\mathcal{C}, \mathcal{D}, [b])$	X	0	X	X	0	X
Euler angle rates	${}^{\mathcal{C}}_{[b]\|\mathcal{D}}\dot{\boldsymbol{R}}\left(ABC, \dot{a}\dot{b}\dot{c}\right)$	AngularVelocityCoord $(\mathcal{C}, \mathcal{D}, [b])$	X	0	X[8]	X	X	0
RPY angle rates	${}^{\mathcal{C}}_{[b]\|\mathcal{D}}\dot{\boldsymbol{R}}\,(RPY, \dot{r}\dot{p}\dot{y})$	AngularVelocityCoord $(\mathcal{C}, \mathcal{D}, [b])$	X	0	X[8]	X	X	0
Quaternion rates	${}_{[r]}\dot{\boldsymbol{q}}^{\mathcal{C},\mathcal{D}}$	AngularVelocityCoord $(\mathcal{C}, \mathcal{D}, [r])$	0	0	X[8]	X	0	X
Homogeneous transformation matrix time derivative	${}^{e\|\mathcal{C}}_{[b]\|\mathcal{D}}\dot{\boldsymbol{T}}$	TwistCoord $(e\|\mathcal{C}, \mathcal{D}, [b])$	X	0	X	X	0	X
six-dimensional vector twist	${}^{e\|\mathcal{C}}_{[r]}\mathbf{t}_{\mathcal{D}}$	TwistCoord $(e\|\mathcal{C}, \mathcal{D}, [r])$	0	0	X	X	X	X
Pose twist	${}^{g\|\mathcal{C}}_{[h]}\mathbf{pt}_{\mathcal{D}}$	TwistCoord $(g\|\mathcal{C}, \mathcal{D}, [h])$	X	0	X	X	X	X
Screw twist	${}^{h\|\mathcal{C}}_{[h]}\mathbf{st}_{\mathcal{D}}$	TwistCoord $(h\|\mathcal{C}, \mathcal{D}, [h])$	X	0	X	X	X	X
Body-fixed twist	${}^{g\|\mathcal{C}}_{[g]}\mathbf{pt}_{\mathcal{D}}$	TwistCoord $(g\|\mathcal{C}, \mathcal{D}, [g])$	X	0	X	X	X	X
Instantaneous screw axis	${}^{\mathcal{C}}_{[r]}\mathbf{ISA}_{\mathcal{D}}$	TwistCoord $(s\|\mathcal{C}, \mathcal{D}, [r])$[9]	X[9]	0	X[8]	X	0	0

Figure 3.8: Some commonly used coordinate representations and properties, linked to their coordinate semantics. This is an excert from [15], and it expands Table 3.1.

3.8 Kinematic Modeling

Section 3.7 introduced the geometric elements and their semantic value required to model a kinematic chain, which is topic of this section.

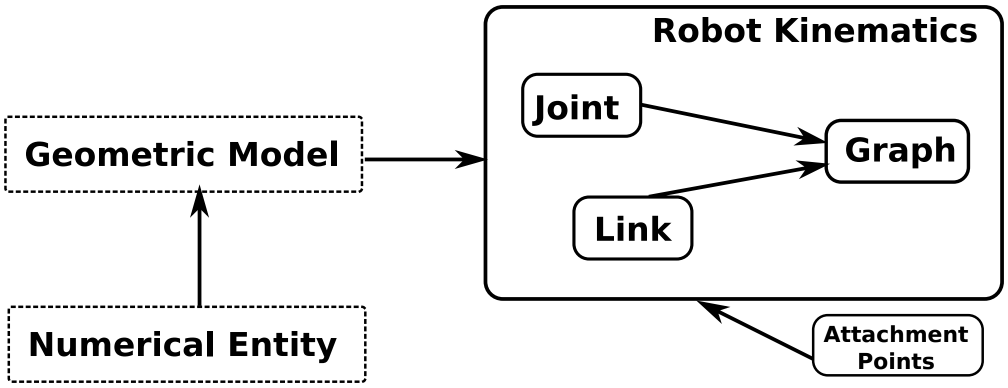


Figure 3.9: A kinematic model is defined as a graph of link and joint elements, which are themselves defined as composition of a geometric semantic model (see Section 3.7) and its numerical description (see Section 3.1). On the link, attachment points can be defined to allow extensions by model composition, such as, for example, geometric shape, dynamical properties, sensors, actuators, or handles for task specifications.

A kinematic chain can be modeled as a set of physical links mechanically connected between each other; in detail:

- a **link** is a physical rigid body[7];
- any geometric primitive (e.g., points, orientation frames, displacement frames, etc) fixed to a link can be used as an **attachment point**, that is used to define geometric relationships, but also to extend by composition the kinematic model with other domain-specific data;
- the mechanical coupling is a higher-order concept (that is, a *relationship of relationships*), more in particular it consists of one or more geometric *constraints* defined over the geometrical relation between pairs of links, and expressed by means of attachment points on those links. The set of these higher-order relations is typically called a **joint**.

The main geometric relationship exploited to define a joint is a *pose* expression, followed by linear and angular velocity relationships (depending on the type of joint described). To each of these relationships, a *joint constraint value* ($\mathcal{J}_{\text{cstrval}}$) is applied. The coordinate representation of the constraining value must be compatible with the coordinate representation of the geometric relationship, and it can be either a constant or another expression; typically, this expression is a *function* of one or more variables which identifies the *state* of the joint (e.g., joint position (q), joint velocities ($\dot{q}$), and so on).
The role of an attachment point is not only to aid the definition of geometric relationship, but also to “connect” to other domain-specific models which characterise and enrich the kinematic models, such as *(i)* a shape model (useful for collision detection and avoidance algorithms, and for visualisation purposes); *(ii)* a mechanical model; *(iii)* a model of a sensor physically attached to a link.

[7]Flexible links are not treated in this document; however, this modeling description still applies.

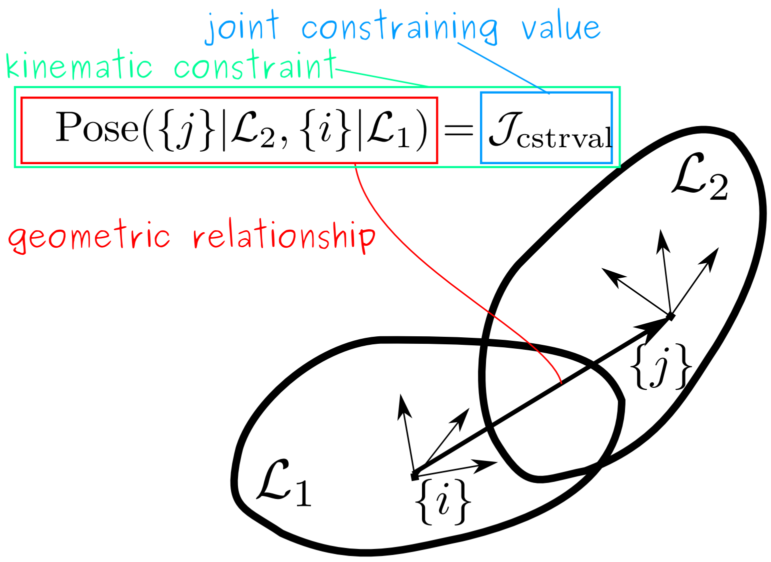


Figure 3.10: A kinematic model example. The displacement frames $\{i\}$ and $\{j\}$ are fixed to the links $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively. A joint is defined as a single kinematic constraint, imposing a constraining value $\mathcal{J}_{\mathrm{cstrval}}$ to the pose relationship defined between $\{i\}$ and $\{j\}$.

Figure 3.10 shows a kinematic model example, where the links $\mathcal{L}_1$ and $\mathcal{L}_2$ are mechanically constrained by a joint defined by means of a single kinematic constraint (i.e., pose between the links). In this example, the joint is defined only by a single, pose-based kinematic constraint. However, it is possible *(i)* to define a joint by means of a single velocity-based (or another type of) kinematic constraint; *(ii)* to define a joint as a combination of kinematic constraints, including inequality constraints; for instance *(iii)* constraint inequalities based on linear or angular velocity relationships can express the boundaries of the minimum and maximum velocity achievable by the joint, and *(iv)* constraint inequalities based on pose relationships can express the limited working range of a revolute or linear joint.
Obviously, a equivalent descriptions of the model depicted in Figure 3.10 exist; for example, replacing the kinematic constraint based on a pose relationship with couple of constraints on position and orientation relationships (decomposition) describes the same the mechanical constraint.

3.8.1 Describing a kinematic model with the Block-Port-Connector meta-model

The kinematic model previously defined fully conforms to the *Block-Port-Connector* meta-model[8] (BPC). In fact, the proposed kinematic model adds *domain-specific* knowledge to a structural description (topology) of a kinematic chain (link-joint-link pattern). The derivation of the kinematic model from its structural description is described in Figure 3.11:

- a mechanical link conforms to a `block` entity in the BPC meta-model;
- a displacement frame is attached to a rigid body (e.g., $\{j\}|\mathcal{L}_2$), conforming to a `port` entity in the BPC meta-model; this modeling choice is motivated since the frames (e.g., $\{i\}$, $\{j\}$) are used to express a (geometric) relationship between links (e.g., $\mathcal{L}_1, \mathcal{L}_2$);

[8] see http://robmosys.eu/wiki/modeling:principles:block-port-connector

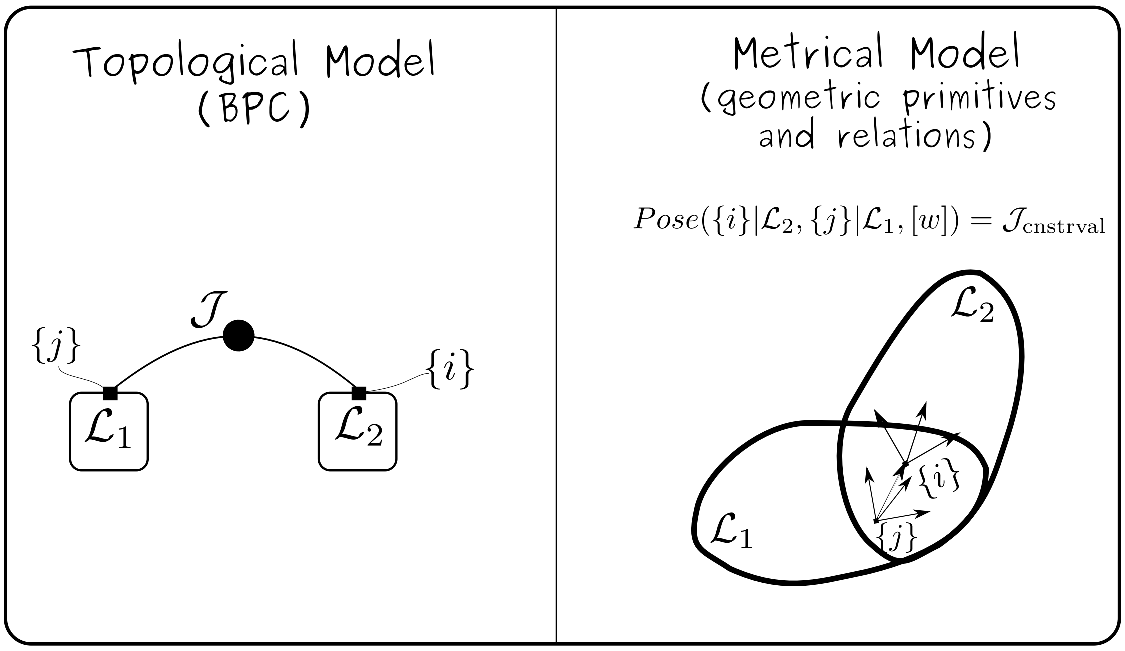


Figure 3.11: A topological model and a metrical model of a two link robot. The topological model conforms to the BPC meta-model, while the metrical model introduces the geometric primitives and relations that allows to define a metric for the computation over the kinematic chain.

- any other attachment point is structurally defined as a `port`, since it allows to establish a connection between a domain-specific entity (represented as a `block`) with another;
- a joint expresses a set of kinematic constraints, and it is represented as a `connector` entity of the BPC meta-model. Precisely, a joint is a *relationships of relationships*, that is, it constrains geometric relationships previously defined. The `connector` hosts the properties of the joint relationship, among which*(i)* a joint type (e.g., prismatic, revolute, etc), *(ii)* a least one geometric expression (e.g., position, orientation, pose, angular velocities, etc), *(iii)* one or more values (or its symbolic representation) that constraint *(iv)* further attachments to refine, thus linking with other domain-specific models (e.g., transmission model, actuator model, etc)

The `connector` (the joint) in Figure 3.11 `connects` only two `blocks` (the links) by means of their `ports` (the attached frames); in general, it is possible to connect more than two `blocks` (links) to the same `connector` (joint).
Moreover, the choice of modeling a joint as a `connector` entity in the BPC model is related to the (arbitrary) *level of abstraction* that the joint describes; in this case, "simply" a set of kinematic constraints. In the case there is the need of express more details of a joint, nothing prevents to "extend" the structural description of a joint as a full `block` BPC entity, allowing to model its internals.

3.9 Composing the joint model with rigid body dynamics

The modelling up to now has been *geometrical*, and this Section adds the primary mechanical dynamical entities and relations, namely those determined by the mass of the bodies involved, and the forces acting on the bodies (external forces on the links as well as internal forces generated on the joints by means of actuators). From a structural point of view, a mass model is easily composed with the already described geometric models, using an attachment point to which an inertia matrix is connected. The mathematical and coordinate representations of force, and of the time derivates of relative pose, are all well understood. The semantic meta data that is needed for an unambiguous interpretation is completely similar to the geometric case, with only the physical dimensions and units being different.

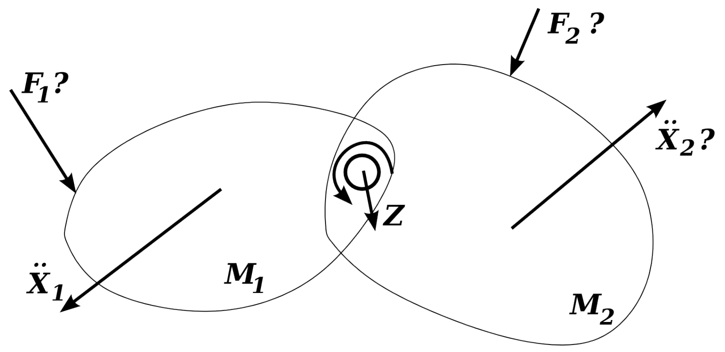


Figure 3.12: The fundamental entities involved in the dynamic behaviour of a (revolute) joint constraint.

Figure 3.12 depicts the essential dynamical behavioural relationship for a joint. In the case at hand, the simple example of an unactuated revolute joint is used, but the modelling for all types of joints is based on the same principles expressed below. The forces $\boldsymbol{F}_1, \boldsymbol{F}_2$ on both links are connected to their accelerations $\ddot{\boldsymbol{X}}_1, \ddot{\boldsymbol{X}}_2$, their inertias $\boldsymbol{M}_1, \boldsymbol{M}_2$, and to the versor $\boldsymbol{Z}$ representing the joint axis, by the following constraint relation:

$$\boldsymbol{F}_1 = \boldsymbol{M}_1^a \ddot{\boldsymbol{X}}_1, \tag{3.1}$$

$$\text{with} \quad \boldsymbol{M}_1^a = \boldsymbol{M}_1 + \left(\boldsymbol{I} - \boldsymbol{M}_2 \boldsymbol{Z} \left(\boldsymbol{Z}^T \boldsymbol{M}_2 \boldsymbol{Z}\right)^{-1} \boldsymbol{Z}^T\right) \tag{3.2}$$

$\boldsymbol{M}_1^a$ the so-called *articulated body inertia* [18], 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. $\boldsymbol{M}_1^a$ of link 1 is the sum of its own link inertia $\boldsymbol{M}_1$ and the “projected” part of the inertia of the second link.

3.10 Composing joint models into a kinematic chain model

Figure 3.13 depicts all entities that form a kinematic chain. The whole collection has a structural composition which is a systematic repetition of the single-joint motion constraint model, with some extra semantic tags that are specific to the composite entity of the *kinematic chain*: end-effector, root, branches, tree structure, etc.

3.10.1 Hybrid dynamics solvers

Figures 3.14 and 3.15 depict the entities that have to be added to the kinematic chain model for any *algorithm* (“solver”) that computes the kinematic and dynamic state of a given kinematic

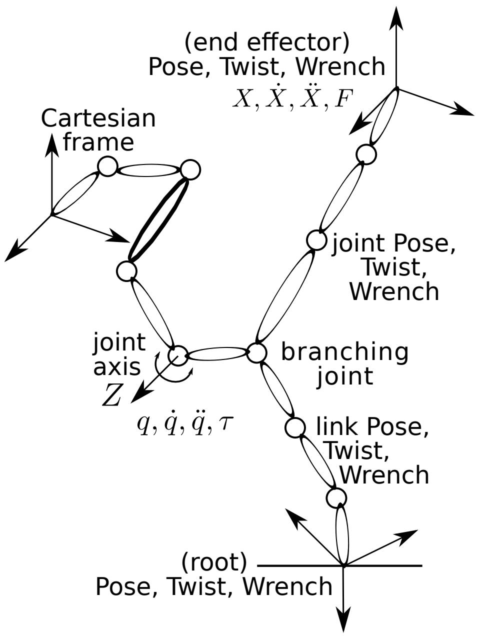


Figure 3.13: The entities that form a kinematic chain.

chain. Classical mechanics, since the time of Gauss [20] already, offered all insights and semantics for even the most advanced modern-times algorithms; all of the relevant entities have already been introduced, except for one: any link in the mechanical system can be subjected to *acceleration constraints*. These can be of *physical* nature, such as contacts or kinematic loops, or *artificial*, as part of task specifications. Note that only acceleration-level constraints have physical meaning, and the (often used) velocity-level constraints are always a composition of an acceleration-level physics with a *constraint controller* [7]. In other words, the most generic "conservation principle" in mechanics is that of Gauss, which states that the mechanical system follows the path of "least constraint" where the latter is measured in terms of the "acceleration energy"; that is the acceleration-level alternative to the better-known velocity-level "kinematic energy" and is represented as the product of acceleration and inertia. In the Figures, the matrix $\boldsymbol{A}$ represents the matrix of the acceleration basis, that is, versors that indicate the spatial directions on acceleration constraints.

All of the solver algorithms introduce, one way or another, an ordering in the functional computations that they need, which is based on the structural model of the kinematic chain. These schedules are commonly referred to as "sweeps". Figure 3.15 depicts the necessary sweeps for the generic example of a tree-structured kinematic chain, together with the entities that are being computed.

While most solver libraries offer programming interfaces that are only solving the kinematic and dynamic state entities, most robotics applications need to "fuse" those computations with perception, control and planning computations. For example, a visual servoing task (depicted as one of the many types of task specifications) in Fig. 3.17, 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. Hence, the RobMoSys models for kinematic chains and their algorthmic solvers have a high level of decoupling, in order to allow all relevant compositions with the other types of solvers mentioned above (control, estimation, etc.).

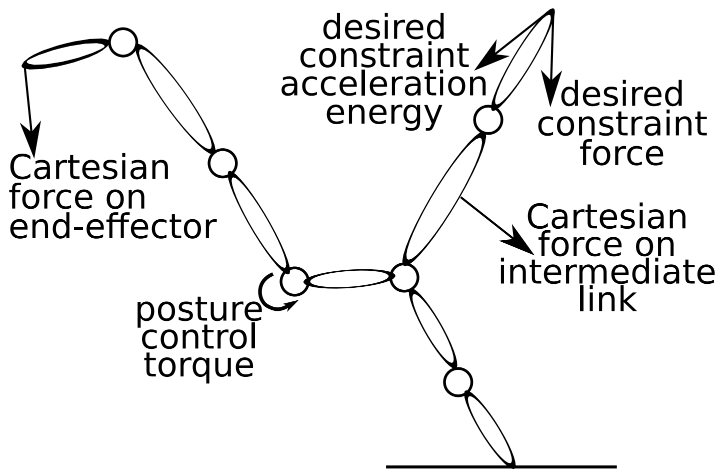

desired constraint acceleration energy
desired constraint force
Cartesian force on end-effector
Cartesian force on intermediate link
posture control torque

Figure 3.14: A hybrid dynamics solver processes the types of inputs depicted above.

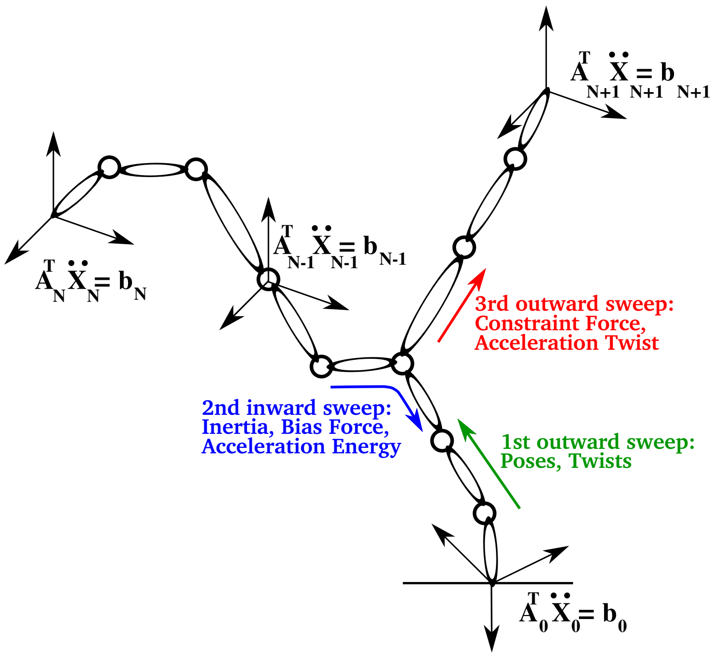

$A^T_{N+1}\ddot{X}_{N+1} = b_{N+1}$
$A^T_{N-1}\ddot{X}_{N-1} = b_{N-1}$
$A^T_N\ddot{X}_N = b_N$
3rd outward sweep: Constraint Force, Acceleration Twist
2nd inward sweep: Inertia, Bias Force, Acceleration Energy
1st outward sweep: Poses, Twists
$A^T_0\ddot{X}_0 = b_0$

Figure 3.15: A hybrid dynamics solver uses the topological structure of the kinematic chain to schedule all the behavioural functions required to answer a particular query.

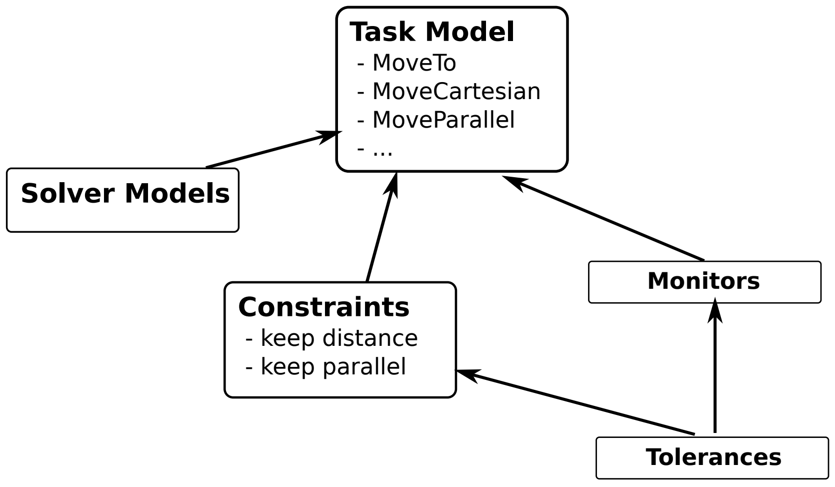


Figure 3.16: Overview of the models involved in the composition of a task specification meta-model.

3.11 Task Specification in the Motion Stack

The instantaneous kinematics and dynamics behaviour of kinematic chains play an essential role in all robotics activities, but obviously they are only one of the many necessary behavioural components. In the context of the motion stack goals of this document, the instantaneous force and motion transformations described by the hybrid dynamics solver must be composed with other types of robotics domain behaviour, such as controllers, estimators, planners, decision makers, etc. The models of such robotics domain behaviour, however, can themselves be composed of, on the one hand, domain-independent behavioural models (Figure 3.16), which are then, on the other hand, configured and specialised with robotics-specific extra semantics; Figure 3.17 sketches a variety of the latter which are *directly* coupled to the hybrid dynamics aspects of kinematics chains. That means that we can model the interaction *structure* via the *attachment point* Entities in the kinematic chains models, and that the interaction *behaviour* can be modelled in terms of link and joint forces, and of acceleration constraints; position and velocity constraints are more specific instances of the latter since they require additional constraint solving.
The *mereological* modelling of the *"motion stack" platform* is the one with the highest level of abstraction, that is, it just represents that any "whole" robot application will necessarily have "parts" from the following complementary domains:

- **Robot**: these contribute capabilities of motion control and (robot-mounted) perception, which are the focus of this document. More in particular, this domain brings **performance** constraints on what an application can expect as capabilities of robots.
- **World**: one or the other form of (distributed) "database" of what objects and activities whose presence in the world the robot application has to be aware of, and of where, at what time and how they can influence the robot's motions. More in particular, this domain adds **safety** constraints to the robot platform capabilities.
- **Task**: these represent the application requirements, and the relations to how the robot's

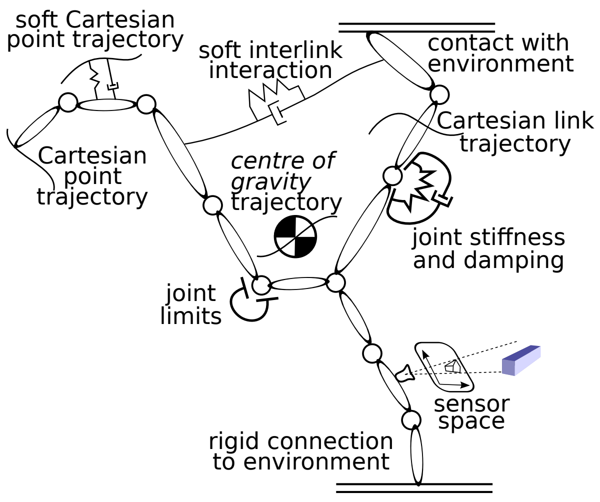

soft Cartesian point trajectory
soft interlink interaction
contact with environment
Cartesian link trajectory
Cartesian point trajectory
centre of gravity trajectory
joint stiffness and damping
joint limits
sensor space
rigid connection to environment

Figure 3.17: Various types of task specifications for which solvers can be built on top of a generic hybrid dynamics solver.

motion capabilities are expected to contribute to realising these requirements. More in particular, this domain adds in **quality** and **progress** constraints.

- **Object**: representations of the properties of shape, size, functional parts,. . . of the objects that the robot has to interact with. More in particular, this domain adds **affordance** constraints to the motion platform.

The **robot motion capabilities** stack, that is, the various kind of robot **motions**, grows in complexity as follows:

- **move**: this is just the already modelled `robot`'s *instantaneous hybrid dynamics*, hence the `task` provides "queries" to the `robot`'s hybrid dynamics solver. For example, the motion resulting from pushing at the end-effector of the kinematic chain; or the forward kinematics; etc.

- **moveGuarded**: this kind of `tasks`, adds `proprio-sensing` perception to the `robot`'s hybrid dynamics `move` capabilities by using its own sensors to determine when the ongoing motion must stop. This is typically realised by monitoring contact transition estimators using current, force, tactile or IMU sensors mounted at various attachment points on the kinematic chain.

- **moveTo**: this kind of `tasks` composes the `move` or `moveGuarded` models with `extero-sensing` perception: the sensors localise and track `object` features in the environment, and adapt the `move`/`moveGuarded` properties accordingly.

- **moveConstrained**: this kind of `tasks` composes the `moveTo` odels with `object` features in the `world`, that are not contributing to the motion's utility, but only bring in "cost" constraints, because the robot must adapt its motion to avoid or control the interactions with these `objects`.

When more than one robot is being composed into a Task, the following "system of systems" coordination models must be added to the motion stack's models:

- **moveCoordinated**: **one** motion specification sub-system provides all other motion subsystems, 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 iiwas, a PR2,...

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

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

- **moveChoreographed**: all subsystems generate coordination events themselves based on their **sensor-based observation** of the other platforms; hence no communication is needed.

  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.

We repeat that the modelling suggestions above are only mereological, hence lots and lots of more detailed models must be provided, topological, geometrical, dynamical, etc.

3.11.1 The relevance of the Task Monitoring during Execution

All the motions above-mentioned require *continuous* **monitor** capabilities to determine, at runtime, if the execution of the task specification model is progressing correctly. 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.
The **moveGuarded** example in Figure 3.18 is probably the simplest possible model of a task specification that is composed with an online monitoring; the example specifies a *nominal* termination condition, which fires a *task accomplished successfuly* 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 3.18: Example of a guarded-motion task definition from [13].

However, the task specification in Figure 3.18 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., [2]). In the RobMoSys approach, a task specification language must include a monitor specification language, which will be developed during the duration of the project. Further information on this topic can be found in [26].

3.11.2 Constraint-based Task Specifications

A way to formulate a task specification is by means of a *constrain-optimisation problem* that is defined by:

- *objective function*(s), e.g., minimise the energy consumption of the robot;
- *constraints* that must be satified during the execution of a task.

The role of the solver is then to generate a control action (joint position, velocities, accelerations or torques) required to realise the task. The above holds both for control-based approaches [2, 22] (which are online, reactive but they may suffer of local minima problems) and for plan-based solvers [21] (which are typically offline, and less reactive since they explore a bigger search space). Example of constraints used in a task specification are the ones imposed by a robot platform, thus constraints defined along the kinematic model (e.g., position and velocity limits, see Section 3.8). Other examples of constraints are: *i)* additional joint limits constraints, if the application requires specific limitiations, *ii)* obstacle avoidance constraints, for those objects that the robot must not interact with, *iii)* constraints that express the *desired outcome* of an *action* in the environment. The latter are the most intuitive, since they conceptually describe *what* the robot must realise. Section 3.7 introduces geometric primitives and semantic relations between rigid bodies, then used to describe a kinematic model; these constraints are physical, related by the mechanical coupling of the robot platform. Additionaly, the very same primitives can be used to describe *desired* relationships between the robot and other objects (physical and virtual), allowing to specify which operations the robot must perform. As an example, it is possible to specify: to keep a certain distance from a wall (a plane, whether it is physical or virtual); to approach a certain object by constraining its pose relationship; to constraint the robot motion to follow a desired trajectory (feedforward component of the control algorithm); and more. In the last decade, this promising approach is becoming popular in research literature; however it is still not a mainstream approach in industrial scenarios. For further reading, please refer to [21, 26, 16, 12, 5, 25, 24].

3.12 Models of Algorithms

The previous sections introduce models of necessary elements to compose a motion stack: kinematic chains, solvers, numerical entities and the semantics of geometric relationships. Additionally, a motion (and perception) stack requires models (and related implementations) of *algorithms* to enable a robot to *act*, more in particular the solvers that compute a control action from the state of the robot, the state of the environment and a given task specification.

Typically, the implementation of an algorithm in software is distributed as a library with *compile-time* type checking, and its documentation is often reduced to an *Application Programming Interface* (API) description, without much support to help developers *i)* to (statically) compose an algorithm with others (e.g., planning, control and sensing), *ii)* to support *runtime* interactions with other functionalities, *iii)* or to deploy an algorithm (e.g., in a software component). Moreover, during the realisation of a concrete software library, the function developer[9] is often enforced to take design choices and assumptions which later on could prevent composability and reusability of the library itself in a different application than the one in the developer's original focus. A typical example of these choices is *information hiding* often caused by *object encapsulation* in object-oriented programming: in some applications it is desired to hide or protect certain data, but in others this could be a main issue (this is particularly true in case of not well-designed libraries). For kinematic chain solvers, one of the most composition-limiting factor in existing API-based libraries is the fact that users of the libraries do not have access to how the "sweeps" have been implemented, and 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.

In short, *composability* and *reusability* is not only required at component level, but also at function level, and a formal, composable model to describe an algorithm implementation must be provided.

The complementarity between the component-based parts of a system's software architecture and its function parts is a key concept of the RobMoSys approach: components[10] allow the composition of *activities* and *services*, including "inter-process" *communications* of various forms, thus covering the following set of system development issues: *i)* scheduling and managing (computational) resources, *ii)* dealing with realtime operations, *iii)* enabling synchronous and asynchronous communications, *iv)* serialisation and deserialisation needed for communication. Instead, composition at the function level is all about sharing *data structures*, with explicitly formalised *data access constraints*, and about the correct *execution order* of all the *functions* that act on those data structure, and that, all together, compose a specific *algorithm*. The implementation of these algorithms ends up inside the above-mentioned components, to realise the components' internal behaviour.

A key design-time responsibility for algorithm developers is to provide a design that is as composable as allowed by the desired behaviour, with respect to *concurrency*, that is, minimizing the amount of constraints on the order of execution of sub-parts of the algorithm while still guaranteeing the correctness of the intended behaviour. This allows more freedom in (compile time or run time) configuration of the control flow ("scheduling") in the implementations that are actually deployed in components.

Note that it is not relevant to let the algorithm designers take decisions about the *parallelization* of their functions. Indeed, the amount and form of parallelism at runtime depends on the resources available in terms of activities, as well as the deployment model of the application and the division of the different functionalities into activities and components. Of course, the algorithm designers can reduce the amount of parallelism by providing design with a high amount of (often implicit!) concurrency constraints, because concurrency is a necessary condition for parallelization, but not a sufficient one.

This Section drafts a modelling language to describe the *data + function + control flow* design aspects of algorithms, independently of how the algorithm is deployed in different components,

[9] http://robmosys.eu/wiki/general_principles:ecosystem:roles
[10] http://robmosys.eu/wiki/modeling:metamodels:component

of its implementation mechanisms, and of the programming language used. The Entities and Relations in that meta model are:

- *D-block*: the Entity to represent *data*, via a concrete instance of a *digital data representation* (or data structure).
- *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" and other having the role of "outputs". (It is possible that one argument in the relation has both roles). The F-blocks should be pure functions, that is, without any side-effects.
- *S-block*: the Relation that represents the *scheduling* constraints (or *control flow*)—that is, the correct execution order—of a collection of F-blocks.

All the above-mentioned entities are *composable*: *i)* a D-block can contain D-blocks; *ii)* a F-block can contain other F-blocks and D-blocks; *iii)* a S-block can contain other S-blocks, thus defining *higher-order relationships* (or *relationships of relationships*). Obviously, F-blocks and D-blocks are *connected* to each other, because the data serves as arguments in the functions; hence, there is a need to be able to represent the *data access constraint* relations, to model the requirements that the order of execution of two or more functions must satisfy to guarantee correct behaviour of the composition. The composition "architecture" above requires its own primitive in the modelling language, and we call that the *C-block*.
F-blocks can be easily mapped to a function prototype ("signature") of a procedural or functional programming language: arguments and return value of the function are *ports* (in *Block-Port-Connector* terms) connected to D-blocks; Figure 3.19 shows an example.

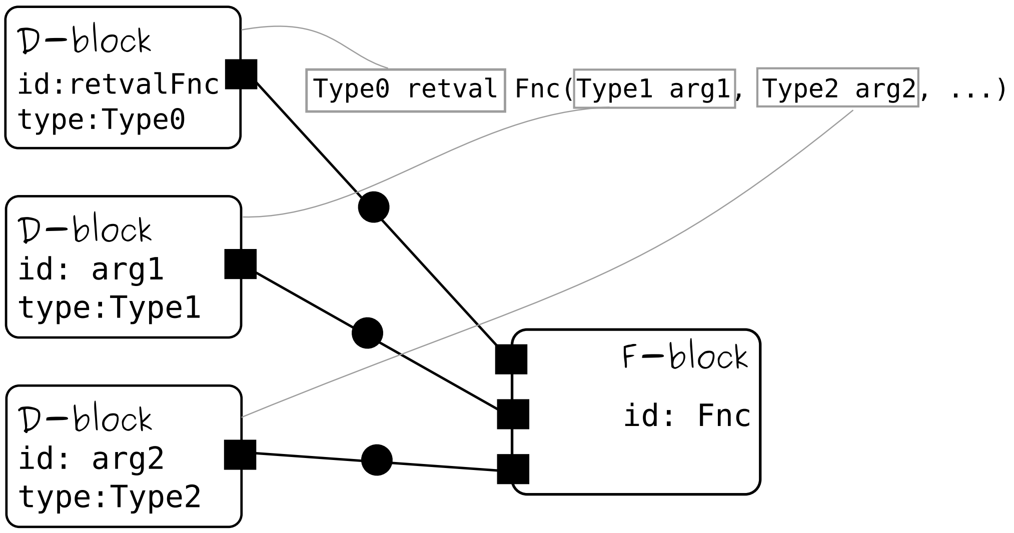


Figure 3.19: A function prototype and its graphical representation as a F-block connected to a set of D-blocks.

Not surprisingly, the above-described composition of data and functions conforms to the Block-Port-Connector meta-model (BPC): the domain is the description of an algorithm implementation, 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` entity. 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 property 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.
Another modeling principle is 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 hiding information, thus enabling better composability and *reusability* of the modelled algorithm. Finally, the representation of data *protection* is possible by dedicated constraints on the accessibility of the D-blocks. An advantage of this approach with respect to a classical API definition is that composite functions (represented by F-block) can be connected/disconnected from D-blocks even at runtime; the data represented by a D-block can be protected by not having any relationship with it, or by adding a read-only data access constraint. In short, the *closure* of the algorithm is not defined by the functional developer, at design-time, but by the component supplier, which evaluates the concrete context in which a functionality is used.
In the context of the "basic building blocks" (WP3) of the RobMoSys project, a formal language to describe algorithms (and the underlying mechanisms and tools to implement their composability) is under development, in collaboration with the RobMoSys community, starting from the principles described in this section; this language will be fully conformant to the BPC model to represent the *structural* aspects.

4. Perception and World Model Stacks

4.1 Introduction

The previous Sections focused on the "Motion Stack", and several paragraphs already made clear that, in a robotics context, motion often requires perception, and both require access to information about how "the world looks like" at any given moment. Examples of such integrated features 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: 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 about how the world around the robots looks like, and what aspects of that world can be detected by the sensors.
This Section explains the similarities that exist between modelling the motion stack and the perception stack, and where they are linked via world models. There are many such similarities, already starting with the fact that also perception has a large set of *platform* functionalities (e.g., Kalman or Particle Filters, Butterworth filters, clustering, etc.). There is also a "stack" structure, that is, a partial order of meta models, with various levels of abstraction, and various sensor data processing dependencies. There *is* an important difference between "motion" and "perception", and that is that the former contains a lot less design freedom and a lot less relevant features: there are just only that limited number of ways to make kinematic chains move, while there is an infinite amount of sensors, sensor features, object properties to estimate, interpretations of actions to support decisions, etc. So, as with the motion stack material in this document, this Section provides a draft of the platform-centred aspects of the perception/world modelling stacks. That draft is supposed to be refined during the course of the project by the consortium, not in the least by means of its interaction with the community. But it is less clear where the perception "platform" stops and the "applications" start.
The perception and world model stacks consist of structural parts and behavioural parts. A structural model describes the parts of a system, their relationships, and the constraints between these two. The perception stack conforms to the Block-Port-Connector meta-meta-model[1] and uses hierarchical hypergraphs to represent the n-ary relations between its meta-models. Behavioural models describe the functionality deployed inside of software component blocks and can be continuous, discrete, or hybrid. Knowledge models describe the n-ary Relations between Entities, and both have Property data structure parameters.

4.2 Modelling

The perception stack is an **example** of a **tier 2 domain-specific model**[2] within the RobMoSys structure. The meta-model of the perception stack can be seen in Figure 3.2. At the mereo-topological level of abstraction, It has the following Entities and Relations:

- **Sensor Models** describe sensors and their properties. The Sensor Model is itself composed of a semantic model of the sensor and its properties, as well as the numerical entities to ground the data it produces in a **numerical representation of sensor data**. This allows for compatibility checks, automated type conversion, and all the advantages discussed in the

[1] http://robmosys.eu/wiki/modeling:principles:block-port-connector
[2] http://robmosys.eu/wiki/general_principles:ecosystem:start

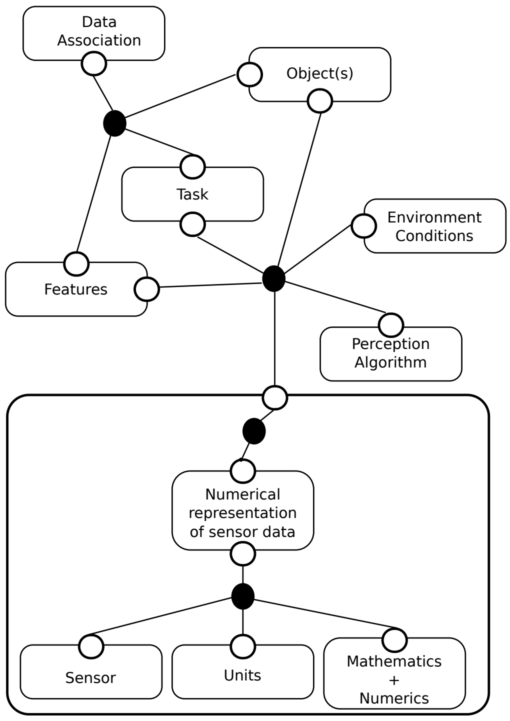


Figure 4.1: The meta-model of the perception stack.

previous chapter. Currently, this is mostly solved by developer discipline with some tooling support, but the ambition of RobMoSys is to replace most of the discipline with structured tooling.

- Models of **Perception Algorithms**, which describe the actual (set of) algorithm(s), their constraints, properties, etc. The *message passing algorithm in factor graphs* [9] plays a similar role in perception as the *hybrid dynamics solver* of Sec. 3.10.1 does for motion. Currently, most algorithms come only with a human readable manual, which makes tooling support and runtime negotiation difficult.

- **Environmental Conditions** 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.

- Models of **Objects** and their properties are typically used as an input and output. Properties of the objects require **matching sensors**, e.g. color typically requires a camera while shape can be detected also be laser scanners and other sensors. Sometimes properties are used to parametrize perception algorithms. And typically perception algorithms are used to detect a state of one or more objects.

- **Features** are extracted from sensor data and/or belong to objects. However, there are algorithms that link sensor data and objects directly without (explicitly) computing features.

- **Data Association** algorithms are required if an algorithm first extracts features from sensor data and then needs to associate sets of detected features with objects.
- 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 algorithm based on what needs to be detected during the task. Especially in this Task modelling, the links between perception, motion and world modelling occur most prominently, so that a large amount of development efforts (including community-centred discussions and iterations) will be required before an "industry-grade" result will have been achieved.

Please note that this figure holds also for a pipeline of several perception algorithms applied one after the other (see example section). RobMoSys does not suggest one large meta-model but instead aims for small, composable meta-models (in the context of tier 2 also known as Domain Specific Languages or DSLs). Concrete examples for the composition of such DSLs were given in the previous chapter.
These entities of the perception stack structure the interaction between the different roles are:

- the **Function Developer** writes the perception algorithm and defines what paramters it requires, data formants accepted, etc;
- the **Service Designer** models the services the perception algorithm provides and requires; interdependency with function developer and component supplier;
- the **Component Supplier** models bahviour fo function supplied by function developer and uses the services from the service designer to package a component;
- the **System Builder** needs to solve constraint satisfaction problem between requirements of components, task, hardware, etc;
- the **Behavior Developer** integrates perception into his task plot and also defines constraints given by the task.

The perception stack discussion provides guidelines for structuring components and applications such that they are **composable** in a larger application. Due to the **heterogeneity of perception algorithms** the perception stack is currently only modelled at a high, mereo-topological, level of abstraction. When going to the (geo)metric and dynamic levels of abstraction, multiple new meta-models need to be composed onto that generic basis, each covering certain approaches or sets of algorithms. For the perception stack one important constraint is to be able to express dependencies between sensor types and object properties as mentioned above. Similarly, constraints on the applicability of perception algorithms on given tasks, objects, or sensors need to be expressed. The following section will provide an example that conforms to the abstract perception stack meta model.

4.3 Example

This section provides an example for modelling an application conforming to the perception stack (see Figure 4.2). It will use the tracking of the 3D trajectory of a ball with with a single RGB camera.
At the bottom there is a model of the used RGB camera. Part of this model is composed with units and math/numerics to obtain the digital representations of the data that the sensor produces. For

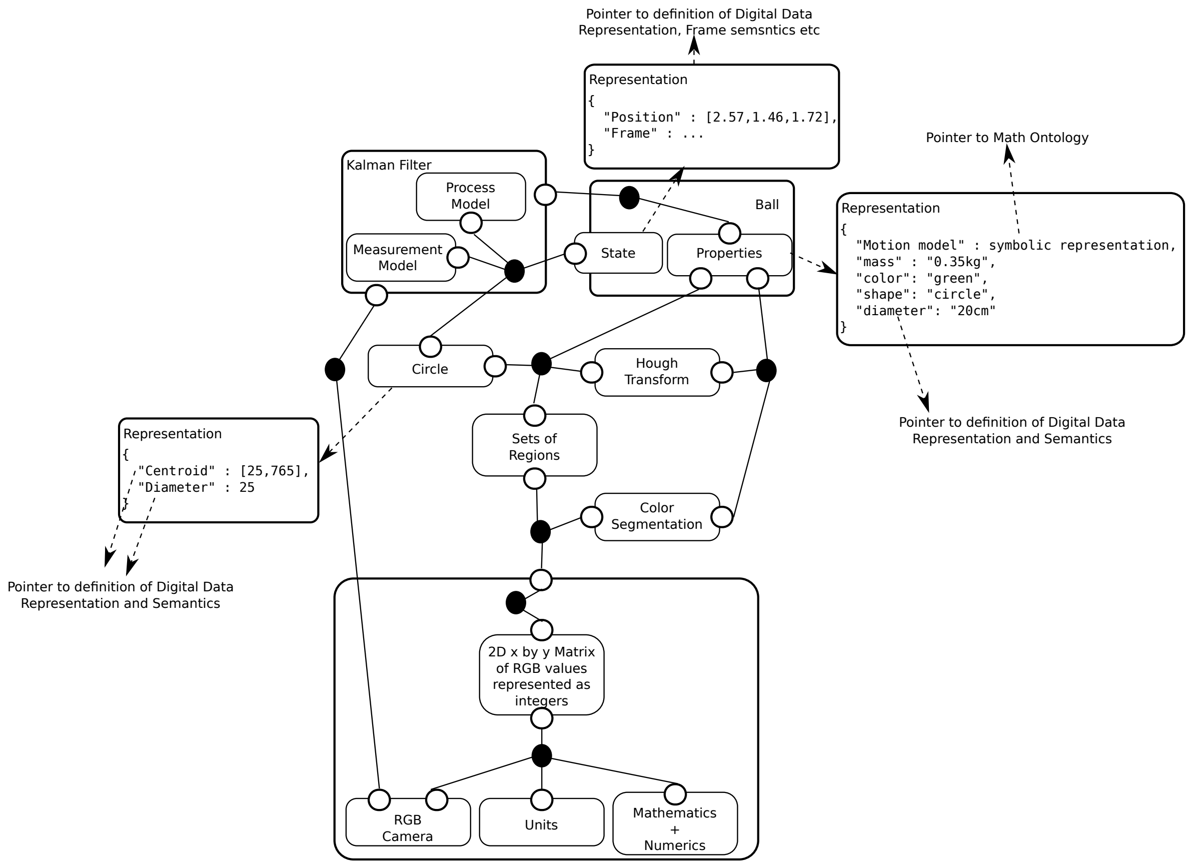


Figure 4.2: Example of the various models that must be composed to represent the functionalities of detecting and tracking a ball with an RGB camera.

this 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.
This digital representation of the camera image is used by a color segmentation algorithm. This algorithm in turn is configured with the color property of the the ball (green) and only looking for green regions. 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"). 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.
Next, a Hough transform is applied on the identified regions to find circular regions. The latter is configured from the shape property of the ball. Since "shape" can have many meanings in different contexts and can be digitally represented in various ways, it is important that it is explicated to which meta-model this property conforms in this application such that the choice of components can be constraint accordingly.
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 4.2 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 ball, an objects in general typically have 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.
While this full application is already possible today, it typically requires the discipline of the programmer and system architect for making consistent choices on implementation and configuration details. In contrast, if all mentioned models exist, tooling can be built to support the persons by linking the corresponding values together or even to support reasoning by the system itself, enabling it to react to changes or performance deficits at runtime.

4.4 World Model

This section briefly introduces the principles and the specifications for an explicit *World Model* description and its implementation, as the necessary glue between "motion" and "perception", when building any robotic application. The role of the world model is to store a concrete representation of the environment, interacting with the motion and perception stack: the perception stack updates the world model with the latest information available from sensing and estimation algorithms; the motion stack updates the robot state information, the internal state of the implemented algorithms and it makes the data previously acquired by the perception stack available to compute robot control actions, which modify the environment state, hopefully according to the task specification. While this description implicitly focuses on the *instantaneous* state of the world, the ambition is to design world models in such a way that the world modelling platform also supports *memory* and *predictions*, with various degrees of resolution and horizon in both time and space. For example, a navigation application needs to know more details about the immediate past, future, space and objects than about the more distant ones, and the platform should support continuously sliding "windows" over time, space and resolution. This technology is less mature than that of perception and, especially, than that of motion, hence the project's ambitions are adapted accordingly: the expectation is to lay a first foundation for the envisaged world modelling scope, but full community agreement, let alone "standardisation", is considered less likely than for the motion stack; the perception stack is somewhat in between with respect to expected community-wide agreement and consilidation.

Anyway, world modelling implies a lot of relations between a lot of objects, hence any implementation will result in a **knowledge-based system** of a significant complexity, not in the least because it has to be seemlessly composable/integrated with the the motion and perception stacks. The data stored in the world model should not cover all the aspects of the environment as a whole, but only those application-dependent information needed for the correct execution of the tasks. Or, expressed differently, many applications are expected to require several world models to run asynchronously, because the different sub-systems in the application have different needs. Hence, all modelling efforts should take into account the technological challenges of distributed synchronisation and (eventual) consistency. These challenges make the need for **composability** of all the modelling efforts extremely important, so the project will give preference to composability quality over feature quantity, when measuring its progress in the domain of world modelling.

In literature, domain-dependent world model representations have been investigated for specific applications: manipulation control [32], haptics [33], rescue applications [4, 23], management of robocup soccer teams [27] and more. On the top of "functional" world models, recently *semantics* and specific anchoring with the perception stack has been introduced [6, 30, 17].

Due to its importance, a world model representation can be found in any robotics applications, often implicitly within the source code (e.g., as a particular data structure to be updated by a certain process) or explicitly and a separate software component (in the form of a "process", as well as of set of functions and data structures); the RobMoSys approach advocates the latter approach for better composability, reusability and usability. In the context of world modelling, "usability" refers to the capability of inquiring different *views* on the same object(s), as will be explained later in the text.

In robotics, the most obvious examples of world modelling can be found in applications that rely on *Simultaneous Localisation and Mapping* (SLAM) functionalities. That is, they consists of i) a motion stack that implements navigation functionalities, including trajectory planning and sensor-based feature tracking; ii) a perception stack to determine the "distance" between the robot and the "static" environment, as well as the identity and value of tracked features; iii) specific data structures to store the map, i.e., the world model which is the outcome of the SLAM activities. Of course, the semantics of "distance" vary depending on the SLAM methodology and on the type of sensor used for the localisation.

Multiple **levels of abstraction** can already be identified from the example above, that is: the outcome of a SLAM process (i.e., the map of the environment) can be a two-dimensional or three dimensional representation; the chosen digital representation of the map may be a probabilistic occupancy grid model; there might be an integration between topological and metric versions of the map; etc. It is important to notice that the "right" level of abstraction choice depends on the application, and it influences the choice over the concrete data structure that represents the map (e.g., probabilistic grid, octomap, etc); obviously, multiple models having different levels of abstraction of the same entity are possible within the same application.
To this end, a RobMoSys compliant world modelling must support multiple **view models** to offer a different aspect, interpretation and/or level of abstraction, of the same entity. Examples of the latter are structured along the same lines already used to represent levels of abstraction in the motion and perception modelling:

- **mereology:** describes the `has-a` (and `collections`) relationships between entities (and the world model instance itself). All entities represented in a world model must be uniquely identified by an `uid` (by means of a IRI, URI or other forms), which is sufficient to describe the *existence* of a certain entity instance in the world model, in a symbolic form;
- **topology:** evinces the *connectivity* of different elements in the scene, e.g., connected rooms in an map environment. Topology can also be expressed symbolically by means of *spatial* descriptors, such as `left-of`, `near-to`, `on-top-op`, `inside to`, etc;
- **geometry:** including affine geometry (e.g, point, line) metric geometry (e.g., displacement, distances, dimensions). Geometric entities can be expressed by means of the geometric semantics described in Section 3.7, whether they are physical objects or virtual artifacts that describes a *feature* used in the task specification;
- **dynamics:** this view is offered to those motion and perception stack algorithms that are capable of dealing with kinematic motion models (differential geometry) and their interaction with the environment (in terms of "physical forces", deformations, etc).

These view models help to structure the selection of motion and perception algorithms in many ways:

- a reasoning algorithm, e.g., symbolic planner, can inference a concrete plan (sequence of tasks) only exploiting symbolic information and connectivity (mereology and topological view); an inference example is to trace the location of one object contained in a box, even after that the box has been moved;
- a motion planner uses geometrical property of the map, but different levels of abstraction can be used (two-dimensional map or three-dimensional map) depending on the application (mobile navigation in a pre-dominantly flat environment, or 6 DOF manipulation);
- the concrete properties of an entity modeled in the world model can be available in different phases of an application, sometimes only after that a certain perception task has been performed. For instance, in the example of Section 4.3, a task specification can indicate the presence of a ball, but its location and pose ball can still be unknown (or are not relevant to the task, yet): in this case, the entity "ball" is modeled only at the "symbolic level" (mereology). Additional topological information can be used as "first guess" of a perception algorithm: if "the ball is on top of a table" and the geometric properties of the table are

known (i.e., geometry view), the location of the ball can be roughly determined by inference reasoning. In a later phase, the visual tracking/servoing task of the ball is possible, updating the geometry view of the ball in the world model;

- in many applications, different views and levels of abstraction are needed to reduce the search space. It is the case for touch-based active sensing tasks, which combines the "act" (motion stack) with the purpose of localising (perception stack) a physical object (and its properties). For instance, in [29] an object is localised by decoupling the active sensing activity in different sub-tasks, each one working on a different configuration space of the world model;
- two or more robotic systems can share the same environment, sometimes accomplishing cooperative tasks, sometimes acting as competitors on sharing the environment resources. In this scenario, the world model is *distributed* and plays a crucial role in the success of the application, since i) the models of the robotic systems (the "agents") must be included in the world model; ii) the negotiation of the resources happens on the basis of the information available in the world model, which must be ("sufficiently") coherent for all the "agents"; iii) each agent may have different limitations in terms of available computational, memory and communication resources; iv) each agent may have a different *local* world model, managing only relevant information for its own tasks, without being aware of the needs of the overall application; v) the previous challenge is even larger in case of communication issues, e.g., in rescue scenarios where the communication interferences can be very high.

4.4.1 World Model Specifications

The implementation of the world model is a fully-fledged knowledge-based system that stores all the information required for the correct execution of a robotic application, *and* must memorize all what is relevant for later applications. The main features that a world model implementation must support are domain-specific in the context of robotics, such as:

- **responsive to "real-time" queries**, both for fast manipulation of the database (insertion/update) and data requests (inference); the real-time boundaries are given by the application itself and the latency requirements from the motion and perception stacks;
- **composability: views and levels of abstraction.** Each view must be defined by a proper meta-model supported by the world model implementation. That is, each view can provide a specific Domain Specific Language (DSL) to perform dedicated queries;
- **distributed:** a world model must be distributed and deployed in different robot platform; each local world model must not be "complete" as at whole, but focus on the concrete data exploited by the *local* motion and perception stacks, with special attention to the present limitations in each single system;
- **history and logs:** a world model should not contain only *instantaneous* information, but also previous states of the environment. This enables task execution analysis *a posteriori*, as well as the application of (online and offline) learning algorithms to improve future executions;
- **framework-agnostic:** the world model should be independent of the component framework, or of any communication middleware used to interact with it; instead it should be trivial to integrate world model functionalities within a component.

Once again, the *structural* aspects of all domain-specific models used in the the RobMoSys world model will be based on the *Block-Port-Connector* (BPC)[3] meta-model. In this way, the BPC (and the tools built around it) is used as a common (infra)structure. The features of supporting queries with multiple *views* and levels of abstraction can be defined by specialising common queries performed over a BPC model instance that represents the structural parts of a world model instance with added domain semantics. This approach is, for example, adopted also in the *OpenStreetMap* ecosystem, where the `relation` and the `way` are the two only structural primitives that are needed at the platform level.

Recently, some implementations have been provided as base for a world model stack. A primitive form of a world model is *the Transform Library* (ROS-TF) [19], which is a distributed way to share poses between different software components (deployed in different computational units). This solution is rather error-prone by design: i) there is no guarantee of consistency and coherence between the local world models, which are reconstructed on the consumer-side; ii) TF implementation mechanism cannot be decoupled from the ROS communication infrastructure; iii) there is no guarantee of uniqueness in the `uid` system adopted (plain names with scope support), thus the same pose could be published by multiple components; iv) there is no support for adding semantics to the pose, or any other additional information that could be attached to the pose; v) many applications require other geometric primitives than poses, for example points, not in the least because any representation of uncertainties on poses introduces mathematical non-invariance problems.
Other attempts to provide a world model solutions are [10, 11], (with a focus on critical rescue situations), or the *open-ease* project [8], which proposes as knowledge-based infrastructure for inference over previous task executions.

The principles introduced in this section should be considered propaedeutic only, since the first RobMoSys third-party funding focuses on the consolidation of the motion stack and perception stack as main priorities. Nevertheless, it is important to start the community discussions about the topic of world modelling, as quickly and broadly as possible, so that the descriptions and suggestions of this Section can further evolve into concrete and consolidated *models* and specifications. As always in the RobMoSys context, special attention will be paid to meta-models, and to the different *views* over the stored data. This discussion should ideally not remain the exclusive effort of the RobMoSys project, but a joint effort driven by the whole robotics community. The actual status of this evolution will be updated on the RobMoSys wiki website.

[3] http://robmosys.eu/wiki/modeling:principles:block-port-connector

Bibliography

[1] JSON Schema: A media type for describing json documents. http://json-schema.org/latest/json-schema-core.html, 2017.

[2] 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.

[3] Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., and Vrgoč, D. Foundations of modern graph query languages. *ACM Computing Surveys* (2017).

[4] Asama, H., Hada, Y., Kawabata, K., Noda, I., Takizawa, O., Meguro, J.-i., Ishikawa, K., Hashizume, T., Ohga, T., Takita, K., Hatayama, M., Matsuno, F., and Tadokoro, S. Information infrastructure for rescue systems. In *Rescue Robotics. DDT Project on Robots and Systems for Urban Search and Rescue*. Springer, 2009, pp. 57–69.

[5] 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).

[6] Bateman, J., and Farrar, S. Modelling models of robot navigation using formal spatial ontology. In *Spatial Cognition IV*, C. F. et al., Ed., vol. 3343 of *Springer Lecture Notes in Artificial Intelligence*. 2005, pp. 366–389.

[7] 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.

[8] Beetz, M., Tenorth, M., and Winkler, J. Open-EASE — A knowledge processing service for robots and robotics/ai researchers. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Seattle, USA, 2015).

[9] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, 2006.

[10] Blumenthal, S., Brieber, B., Huebel, N., Yazdani, F., Beetz, M., and Bruyninckx, H. A case study for integrating heterogeneous knowledge bases for outdoor environments. In *Proceedings of the 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Daejeon, Korea, 2016), IROS2016.

[11] Blumenthal, S., Bruyninckx, H., Nowak, W., and Prassler, E. A scene graph based shared 3D world model for robotic applications. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Karlsruhe, Germany, 2013), ICRA2013, pp. 453–460.

[12] Borghesan, 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.

[13] 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.

[14] Crockford, D. The application/json Media Type for JavaScript Object Notation (JSON). http://tools.ietf.org/html/rfc4627, 2006.

[15] 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.

[16] De Schutter, J., De Laet, T., Rutgeerts, J., Decré, W., Smits, R., Aertbeliën, E., Claes, K., and Bruyninckx, H. Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty. *The International Journal of Robotics Research 26*, 5 (2007), 433–455.

[17] Elfring, J., van den Dries, S., van de Molengraft, M. J. G., and Steinbuch, M. Semantic world modeling using probabilistic multiple hypothesis anchoring. *Robotics and Autonomous Systems 61*, 2 (2013), 95–105.

[18] Featherstone, R. *Rigid Body Dynamics Algorithms*. Springer, 2008.

[19] Foote, T. tf: the transform library. In *Proceedings of the IEEE International Conference on Technologies for Practical Robot Applications* (Woburn, Massachusetts, USA, 2013).

[20] Gauss, K. F. Uber ein neues allgemeines Grundgesatz der Mechanik. *Journal fü die reine und angewandte Mathematik 4* (1829), 232–235.

[21] 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.

[22] Mansard, N., and Chaumette, F. Task sequencing for sensor-based control. *IEEE Transactions on Robotics 23*, 1 (2007), 60–72.

[23] Noda, I., Hada, Y., Meguro, J.-i., and Shimora, H. Information sharing and integration framework among rescue robots/information systems. In *Rescue Robotics. DDT Project on Robots and Systems for Urban Search and Rescue*. Springer, 2009, pp. 145–159.

[24] 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).

[25] 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.

[26] 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.

[27] Silva, J. a., Lau, N., Neves, A. J. R., Rodrigues, J. a., and Azavedo, J. L. World modeling on an msl robotic soccer team. *Mechatronics 21* (2011), 411–422.

[28] The HDF Group. Hierarchical Data Format 5. http://hdf.ncsa.uiuc.edu/HDF5/.

[29] Tosi, N., David, O., and Bruyninckx, H. DOF decoupling task graph model: Reducing the complexity of touch-based active sensing. *Robotics 4*, 2 (2015), 141–168. Special Issue "Representations and Reasoning for Robotics".

[30] van de Molengraft, M. The RoboEarth project. http://www.roboearth.org/, 2011.

[31] W3C. QUDT (Quantities, Units, Dimensions, and Types). https://www.qudt.org.

[32] Walker, M. W., and Dionise, J. A world model based approach to manipulator control. In *Proceedings of the 1990 IEEE International Conference on Robotics and Automation* (Cincinnati, OH, 1990), ICRA90, pp. 453–460.

[33] Yoshikawa, T., and Ueda, H. Module-based architecture of world model for haptic virtual reality. In *6th International Symposium on Experimental Robotics* (Barcelona, Spain, 1997), pp. 155–166.