Composable and explainable cyber-physical systems-of-systems: meta models and best practices for resilient and explainable holonic architectures. (With a special focus on robotic systems)

Herman Bruyninckx
(KU Leuven & TU Eindhoven)
with contributions and ideas[1] by Enea Scioni, Nico Hübel, Filip Reniers
(KU Leuven, Belgium)
Christian Schlegel, Dennis Stampfer, Alex Lotz (HSU Ulm, Germany)
and René van de Molengraft (TU Eindhoven, the Netherlands)

2019-02-13

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

Executive summary

Cyber-physical systems consist of multiple sub-**systems** that **interact** with each other, **exchanging matter**, **energy** and **data**, but also increasingly more **information** and **knowledge**, formally represented as “data”. The ambition of this document is to find the **least amount** of **formally encoded knowledge models** that are needed (i) **to represent** all possible systems, and (ii) **to control** them in **predictable**, **resilient** and **explanaible** ways. This document is inspired by the **best practices** to be found in many successful and resilient societies and organisations created by humanity, and the vast amount of **knowledge** in physics, mathematics, computer science and systems-and-control theory. **Robotic systems** are the primary application target; the advocated way of designing robotic systems is to consider that the simplest system to design consists of *multiple* robots, each executing *multiple* tasks at the same time, with all their actions being realised in *multiplel asynchronous* software activities.

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

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

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

The size and the scale of integration of robotic systems grow beyond what single developers or develpment teams, can comprehend, create, validate and maintain. Contrary to what has happened in the ICT-driven applications that could be built on top of "the Web" platform, no *"giant companies"*[1] have been created that have the funding and man power to realise such consolidation single-handedly, *and* to impose it on the rest of the world in a monopolistic way. The cyber-physical domains do not have such monopolistic giants, so, it is up to the community to put its act and hands together, and to create the **digital platforms** for the domains. That platform must be functionally effective and efficient, **commercially fair and exploitable**, and offered to the world as a set of extremely **composable** modules. Those modules' further development and maintenance can be shared in the open, while their focused exploitation can support the creation of innovation-driven (non-giant) companies. These composable modules are not only software libraries and components, but also models and documentation, as well as tools and *best practice* architectural patterns.

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

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

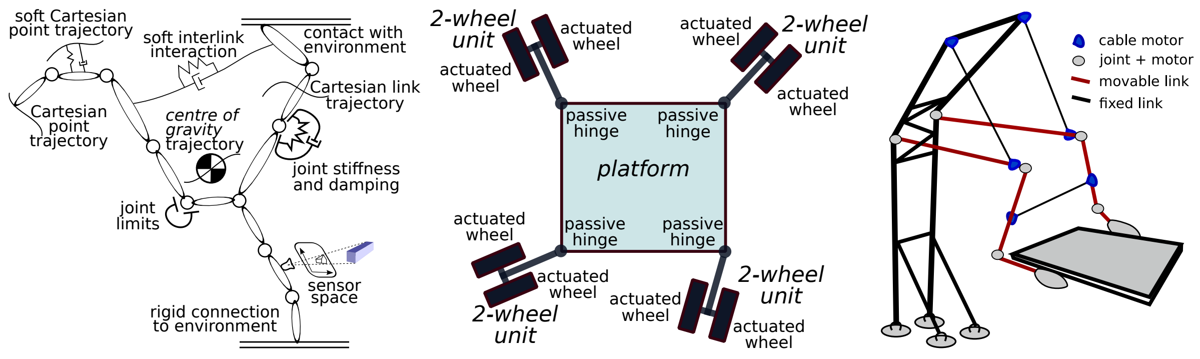
A "component" is any type of computer-readable formal representation of the **composition** of smaller parts into a bigger whole. This can be as abstract as the composition of knowledge relations into a "knowledge base", and as concrete as deploying executable software code into a process on an operating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

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

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

[1]The so-called GAFA giants: Google, Amazon, Facebook and Apple.

certify components and systems.


soft Cartesian point trajectory
soft interlink interaction
contact with environment
Cartesian point trajectory
centre of gravity trajectory
Cartesian link trajectory
joint stiffness and damping
joint limits
sensor space
rigid connection to environment
2-wheel unit
actuated wheel
passive hinge
platform
cable motor
joint + motor
movable link
fixed link

Dual-arm manipulator   Eight-wheel mobile robot   Cable-driven robot

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

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

The core methodology to link models to executed actions is **hybrid constrained optimization**. The first step in the methodology is to use the above-mentioned models to construct a *description* of the to-be-executed activities by means of (i) **objective functions** to optimize and (ii) **constraints** to satisfy. Objective functions and constraints are of three complementary types: **symbolic** constraint satisfaction (that is, reasoning on the "knowledge relations" that populate the application's **context**), **continuous** constrained optimization (in "metric" domains like time, space, force, energy,...), and **discrete** optimization (that is, the "scheduling" of which combination of specifications to solve under which conditions). The second step of the methodology is *to solve* the problem with as outcome the actuation set-points to apply to the system. Very few application contexts *require* that the executed action is indeed the most optimal that exists, and are happy with a **satisfactory** solution [51]. The solution need also not be completely computed before the system is allowed to start acting, since any *feasible* solution that is available can already be used to get the system started; not in the least because only *actions* (and not optimizations) can help the system controller to assess how well it is realising its objectives in the real world. In addition, there is typically enough time *during the system's operation* to run further iterations of the solver towards more optimal/satisficing outcomes.

With the above-mentioned core models and solver methodology, the core of **system design** is then to combine them and create **stable subsystems** (sometimes refered to as "**holons**") [30]: a subsystem is called "stable" if it provides a "good enough" trade-off between (i) the **quality of the services** it delivers to the application, (ii) the **use of resources** it requires to provide those services, and (iii) its **robustness** against the disturbances that the application's context will bring to the system.

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

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

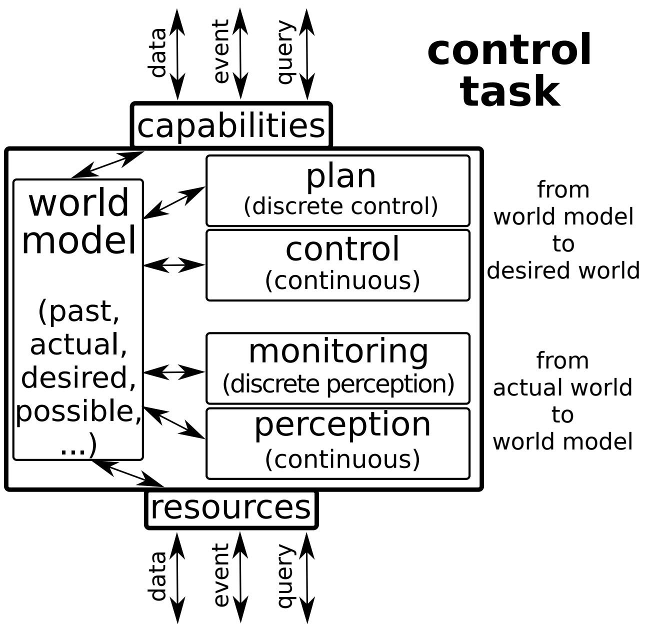


Figure 2: This figure sketches the composition of the different parts needed to realise a "task". The arrows and the rectangles represent the compostion primitives of *interconnection* and *containment*. Note that the figure is *not* representing a software component, but it represents the *structure*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

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

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

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

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

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

Contents

Chapter 1

Foundations of knowledge-driven engineering: meta modelling

Any **formal representation** of **knowledge** consists of **models** built on the axiomatic foundation of **entity** and **relation**: an entity represents "stuff", "things", "primitives", "atoms",. . . , and a relation represents a dependency between properties of entities [11]. The **meaning** of a model must/can be formally represented by relating the model to one or more **meta models**: the latter contain the representation of all the relations that constructs in the model must satisfy in order to be "well formed" and "meaningful". The relation between a model and its meta models is a relative concept: every meta model is a model in itself and hence has its own set of meta models; in common practice, one limits the terminology to the triple *model–meta model–meta meta model*. A **knowledge system** for a particular domain is a set of models and their meta models that describe "meaning" in that domain. **Reasoning** in such a knowledge system is done in two complementary ways: the graphs that make up a knowledge system are **queried** via **graph matching** or **graph traversal**, with the latter supporting the **higher-order reasoning** required for **explainable** robotic systems.

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

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

1.1 Models for science and engineering

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

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

Both scientists and engineers know that a model *is not* the reality, but just a *representation* of their artificial and subjective selection of those parts of reality which they consider relevant. They both also know that such relevance is not an absolute property, but one that depends on the *context*. That context determines that their modelling stops with a particular selection of axioms or facts, that are not modelled in further detail but are assumed to be **grounded**. In science, such grounding consists of (references to) other (possibly not yet formalised) models and axioms, and in engineering it consists of software, common knowledge and facts. The last resort of that grounding is the human mind: eventually, it will be humans who give the validation stamp to the quality of a **model**, or of the **software** that implements models and the **tooling** that transforms models.

A core ambition of a modelling activity is to represent **context** and **composition**, in such an explicit way that one can add new models to already existing ones without changing the meaning of the latter, or its formalisation. When developing computer-controlled machines, there is an obvious extra core ambition: to create **software** libraries of **digital twins**, that implement (or "ground") the models of the entities and their interactions.

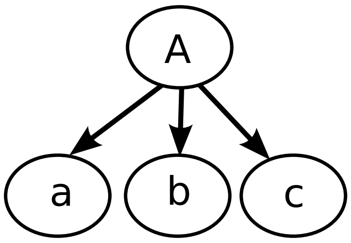


Figure 1.1: Directed graph, with anonymous edges between named nodes.

1.2 Knowledge modelling with property graph representations

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

havioural model of a knowledge graph represents how the knowledge can be exploited (activated, reasoned with,. . .) by selecting the parts of the knowledge graph required to answer "**queries**."

Mereology, holonomy and meronomy are all very related terms, that can be used interchangeably, to denote the symbolic relations that humans understand to exist between "parts" and "wholes". This document uses the term "mereologic" to represent this symbolic relation whenever *models* are considered, and the term "holonic" when holarchy *architectures* are considered (as alternative to the hierarchical, homoarchical or heterarchical designs. The latter interpretation finds its origins in the seminal works by Arthur Koestler and Herbert Simon. Its applications in robotics and manufacturing appeared in the 1990s [56, 58].

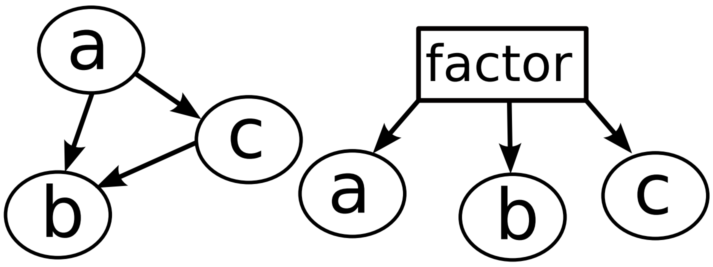


Figure 1.2: "Plain' graph on the left, and hypergraph, or factor graph, on the right. The normal graph can only represent a relation between *two* nodes, while the hypergraph can represent *n-ary* relations between *more than two* nodes.

1.2.1 Mechanism: property graph for entities, relations and constraints

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

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

A **property graph** [3] extends the hypergraph model with a **third type of node** (or edge), the **property node**, to represent the **properties** of an entity node or a factor node (Fig. 1.3). Again, this addition is a cheap way to increase the semantic richness of a graph even further, without adding any structural complexity; it does add a structural *constraint*, in that a property node should only be connected to one single other (entity or relation) node.

1.2.2 First-order relations

This document adopts the **axiomatic** basis to represent **first-order knowledge models** with **property graphs**: **nodes** represent **entities**, **edges** represent **relations**, and both have **properties**. Properties represent information like name, identity, type, the data structures that store the parameters that define the entity's "behaviour", provenance, Dublin Core

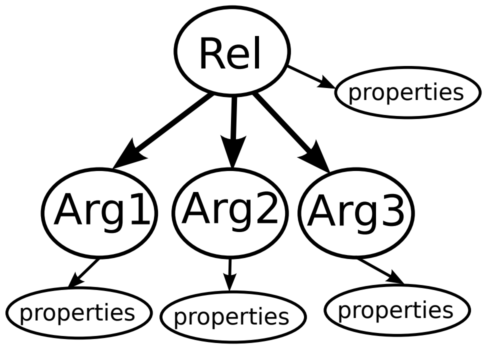


Figure 1.3: Property graph, with an edge linking every named node to a node containing key-value pairs (or any abstract data type for that matter), representing the properties of the named nodes. The property graph is a *directed graph* representation of the **mereological** model of a relation with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

metadata, etc. This step from normal graphs to factor graphs allows richer semantic expressivity, without the need to add any new structural primitive, and with just a small extra complexity in the bookkeeping of the type of node.

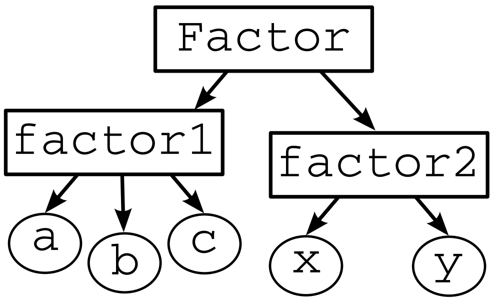


Figure 1.4: Higher-order knowledge model.

1.2.3 Higher-order relations

The second **axiomatic** basis is that all application contexts of realistic complexity need to represent not only first-order but also **higher-order** relations (Fig. 1.4), to express relations between the properties of already modelled relations *with* an explanation of the information and thematic structures in that relation. For example, the concept of a singularity in the configuration of the kinematic chain of a robot relates the values of the chain's joint positions with the chain's geometrical properties; and there is even a yet higher-order relation with the requirements of the task the robot is executing, since not every *geometric singularity* is also a *task singularity*; for example, pushing a load with fully stretched arms can be a good approach to reduce the amount of force needed in the muscles/motors, while it *is* a geometrically singular configuration.

Types of higher-order relations

- transitive, converse, identity, associative, commutative, symmetric, asymmetric, reflexive or equivalent relations.
- constraint (Fig. 1.5) which are relations that put limits on the values of some properties in some entities connected by another relation.
- tolerance: the intervals within which the values of a constraint relation must fall.
- context: the "background" knowledge that influences the property values of entities in relations (Fig. 1.6). For example, that the gains in a motion control feedback loop depend on the safety requirements of the application in which the motion control is being used.
- S-expression gives structure to mathematical relations.
- level of measurement: nominal (types), ordinal, interval, and ratio.
- taxonomy: a tree-structured relation on entities.
- directed acyclic graph (DAG): a graph-structured relation on entities with particular ordering constraints.
- causal relation, causal chain.
- dependency graph:

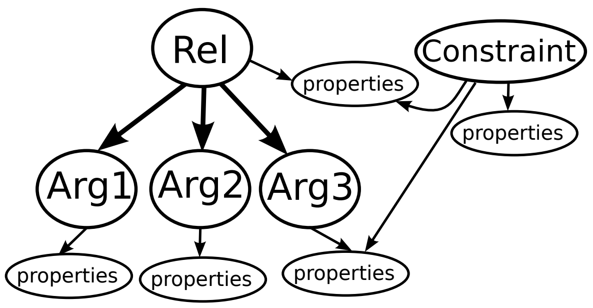


Figure 1.5: A *directed acyclic graph* representation to model a **constraint** between the properties of a relation and one of its arguments. The constraint is a (higher-order) relation in itself, with its own properties.

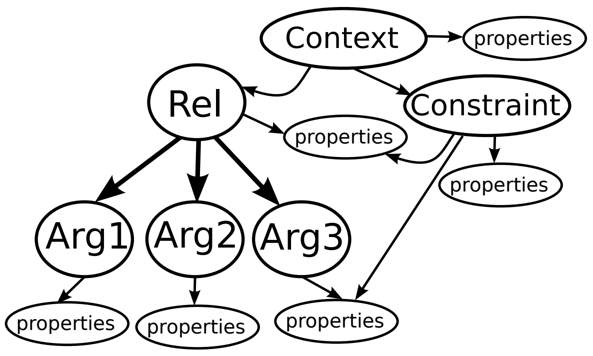


Figure 1.6: *Directed acyclic graph* representation that adds a **context** relation to the model of Fig. 1.5. Such a context can influence the meaning/interpretation of all the relations and constraints within the context, and is yet again another (higher-order) relation in itself, with its own properties.

1.2.4 Policy: `semantic_ID` meta data

Higher-order relations imply the need for so-called **reification** in the modelling, which means that *relations* become *entities* themselves. So, among other application-specific properties, they have their own *unique identifier* to refer to, so that they can be used as arguments in other relationships. In summary, the computer representations of *all* the "digital twins" and their first-order and higher-order interconnections can be realised with the same mechanism of the *property graph*; this document suggests to adopt the *policy* to give nodes (and hence also edges) the following **semantic ID** abstract data type:

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

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

The motivations behind adding semantic meta data to models in a systematic way are that (i) the topological structure "model"–"meta model"–"meta meta model" *is* domain knowledge that is worth representing explicitly in itself, and (ii) sooner or later, any model will have to be connected to new information, and any system will become part of an even larger system. At that moment, the new bigger context will have to be able to refer to already existing models and "reason" about them, to any level of detail, *and* without having to change anything to the already existing representations. The semantic richness of a knowledge graph increases significantly every time such **higher-order knowledge**, or any new **abstraction**, are added to the graph. These are just other set of nodes and edges, that represent "knowledge about the knowledge". For example: the **reasons why** relations between entities exist, or in what ways an abstraction can be turned into a concrete instantiation.

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

1.2.5 Mechanism for reasoning: property graph traversal

A property graph is the **structural** representation of a knowledge model. The **behavioural** part consists of **graph matching** and **graph traversal**. These are activities to query the graph for answers. Graph matching gives a template graph as input, and outputs (the set of) matching sub-graphs in the knowledge graph; the input in graph traversal is a data structure ("programme") that encodes at which node to start the query answering, and in which order to follow edges and nodes further in the graph to find the answer. The simplest form of graph traversal is the one that workds on tree structures; the popular serialization formats such as XML and JSON[2] have tree traversal query language, e.g., XPath (and its superset XQuery) and GraphQL.

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

The concept of traversal can be illustrated by means of the simple example in Fig. 1.6: in order to find out which constraints should be put on the values of the `properties` node of the `Arg3` node in the relation `Rel`, one can follow the arrows from the `Rel` node to the `Constraint` node and its `properties` node. Note that the direction of the arrows reflect *meaning* in a relation in a model, but these arrow directions do not constrain the traversal through the model's graph that is stored in the graph database; the latter always add two links for each directed arrow in a model, to allow to travel in any direction between the nodes that the arrow connects.

1.2.6 Policy: constraint, dependency and causality graphs

Some important instances of the property graph meta model are the **constraint graph**, the **dependency graph** (Sec. 1.15), and the **causality graph**. They all formalize constraints that exist between the properties of nodes or relations; but a dependency graph is a special case of a constraint graph, because "dependency" is a semantically richer type of "constraint", and a causality graph is a specific type of dependency, in that it represents *cause-and-effect* dependencies.

1.2.7 Best practices: quasi non-existent

In contract to *first-order reasoning*, the literature and the state of the practice on higher-order knowledge graphs and graph traversals is scarce, e.g., [3, 17, 47], and practically useful software tools are not known to the authors. Hence, there are not enough use cases out there to identify common policies, let alone bad or good practices.

1.2.8 Storage and reasoning in property graph databases

The **storage** of *directed graph* representations as in Fig. 1.3 is realised by **graph databases**, that provide **implementations** of property graphs [3]. More in particular, they directly support the mereological and topological aspects of *entity-relation models*: they keep track of which entities are connected to which other ones, and of the properties of each and every node; and they support the specification and execution of queries via graph matching and/or graph traversal.

Of course, this interlinking of models via property graphs must stop somewhere; in the case of (robotics) software systems, this "grounding" takes place when a piece of concrete soft-

[2] An insightful and concise comparison between XML and JSON can be found here.

ware is composed with the set of models that reflects the software's behaviour, and a human expert has validated that this implementation is correctly realising what is represented in the models. (The software artefacts themselves are *not* stored in the graph database, but only their *meta data*.) It is a responsibility of the community in a particular domain to decide what grounding that domain will expect, for what kind of purposes. For example, formal verification expectations are a lot lower for ROS-based educational robotic systems than for ESA's planetary rovers and manipulators; hence, also the accepted level of grounding will be more stringent in the latter case.

Various types of **reasoning** are needed on the models of (software) systems, to serve various complementary purposes: code configuration and generation; model validation (*"does the system specifications conform to the application's requirements?"*), verification (*"does the system implementation conform to its specifications?"*), or certification (*"is there an official organisation that confirms that your system implementation is validated and verified?"*); dialogues with human users and between different computer systems; etc. If all models are stored as directed property graphs in a graph database, reasoning is realised by means of the graph matching/traversal query language of the graph database. Some examples of tools that support such graph traversals are Gremlin, SPARQL, or Cypher.

1.2.9 Host languages to store, exchange and query models

Only very few formal languages are designed to support the exchange of models between graph databases and other software agents. EXPRESS is a modelling language with already a mature history and industry-backing, to represent product data, institutionalized in the ISO Standard STEP; more recent modelling languages were born in the context of "the Web", namely RDF and JSON-LD, that have built-in support to represent *named directed graphs*, via their keywords for **context** and **unique identifiers**. The `@context` allows a model to point to an "external" model, in a purely symbolical way; the `@id` allows models (external as well as internal) to symbolically point towards any entity in a model. Together, these simple semantic additions facilitate *composition* of models with very low coupling, the testing of *conformance to meta models*, and the representation of *higher-order relations*.

XML is probably the most popular "host language", with an ecosystem of tools, developers and users that is an order of magnitude larger than those of JSON-LD and RDF, but it is designed to represent only *trees* (implicitly, via the containment constraints on XML tags) and not graphs. There are XML-based extensions such as Xlink that provide the *mechanism* for cross-linking, but not the *semantics* of "context" and of "entity IDs".

Here is a possible encoding in JSON-LD of the simple model in Eq. (1.1):

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

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

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

    "TypeArgument1": "IRI-of-MetaModel-for-Argument1-Entities",
    "TypeArgument2": "IRI-of-MetaModel-for-Argument2-Entities",
    "TypeArgument3": "IRI-of-MetaModel-for-Argument3-Entities",
  },
  "@id": "ID-Relation-abcxyz",
  "@type": ["Relation, "Entity","MyTernaryRelation"],
  "RelationName": "MyRelation",
  "RelationType": "MyTernaryRelationType",
  "RelationNoA": "3",
  "generatedAt": "2017-06-22T10:30"
  "@graph":
    [
      {
      "@id": "ID-XYZ-Argument1",
      "@type": "TypeArgument1",
      "RelationRole": "MyTernaryRelationRole1",
      "EntityPropertyStructure": [{key, value},... ]
      },
      {
      "@id": "ID-XYZ-Argument2",
      "@type": "TypeArgument2",
      "RelationRole": "MyTernaryRelationRole2",
      "EntityPropertyStructure": [{key, value},... ]
      },
      {
      "@id": "ID-XYZ-Argument3",
      "@type": "TypeArgument3",
      "RelationRole": "MyTernaryRelationRole3",
      "EntityPropertyStructure": [{key, value},... ]
      }
    ]
}
```

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

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

1.3 Mereology level of abstraction: `has-a`

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

$$\texttt{Relation\_x (Entity\_1, Entity\_2, Entity\_3),} \tag{1.1}$$

with a lot of background knowledge. The simplest interpretation (often called the "highest" level of abstraction), is that of its **mereology**, which just represents the **parts** that make up the model, without any additional structure or behaviour. In this case, the parts are `Relation_x`, `Entity_1`, `Entity_2` and `Entity_3`. In other words, a mereological model consists of the **set**, or **collection**, of the `Relations` and `Entities` that are relevant for the modelled system. Remark that the **context** (Fig. 1.6) to interpret the meaning of the relation is not specified explicitly; at the mereological level, such a context is just another, larger, mereological set, often called the universe or the domain of discourse of a model, and it represents all the entities and relations that should be considered together before one can hope to interpret the meaning of the model unambiguously.

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

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

and one single **entity**:

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

For example, the "model" in Eq. (1.1) has eight instances of the `has-a` relation:

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

Figure 1.3 is one (of the many possible) graphical representations of the mereology of Eq. (1.1). Figure 1.5 extends the model with a *constraint* on the relation; the constraint brings in "loops" in the representation. The further extension with a *context* is shown in Fig. 1.6. The suggested approach of model **composition** has the advantage that the directed graphical models have only *acyclic* loops; this **pattern** of **cycle-free** composition is a best practice that is sometimes called the dependency inversion principle, and that this document tries to follow as often as possible.

1.4 Topology level of abstraction: `contains`, `connects`

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

```
Relation_x ( Argument_1 = Entity_1, Argument_2 = Entity_2, Argument_3 = Entity_3 ).
```
(1.2)

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

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

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

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

  and with two **entities**:

  - `container`: the entity that "owns" the `contains` relations with all the entities "inside".
  - `connector`: the entity that "owns" the `connects` relations with all the connected entities.

1.5 Role of mereo-topological models

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

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

1.6 Role of `conforms-to` and `is-a` relations

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

1.6.1 `conforms-to` hierarchy on relations

A **meta model** (or *schema*) provides the entities, relations, and constraints with which to decide whether a particular model is **(syntactically) well-formed** and **(semantically) meaningful**. In other words, a meta model is a model of a language in which to write models; each such model must satisfy the `conforms-to` relation (rather, *constraint*) with respect to each of its meta models, that is, all constructs that are being used in the model satisfy the constraints on the relations that are made explicit in a meta model, but only *for as far as* the constructs in the model indeed use entities that are defined in a meta models. It is important to stress that it is *not* required that *all* constructs in a model must conform to the constraints of the meta models, because this "'underconstraining redundancy" (also known as the **open-world assumption**) is key to allow composition with any new model extensions and new corresponding meta models (*if* these do not contradict constraints introduced by earlier meta models!).

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

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

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

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

1.6.2 `is-a` hierarchy on properties

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

1.6.3 Best practice: four levels in `is-a` and `conforms-to` hiearchies

No model of the real world, or of an engineered system to control the world, can or must cover all possible aspects of that world or of its engineered instrumentation. The *selection* of the aspects of the real world that are included in a model defines the *abstraction*: any use of the model, in whatever context, will have "to make abstraction from" the non-modelled aspects. The other, complementary, way to reduce the correspondence between a model and the real-world entities that it represents, works by reducing the **level of resolution** in the model. For example, one can put a building on a map just by adding a tag with its name attached to a particular map coordinate, or one can draw a polygon on the map of the outline of the building, or one can add a 3D CAD model. In none of these three approaches, the building is abstracted away, because it can be part of modelling relations; what *is* abstracted away by the just-mentioned geometrical models of the building are its real-world properties such as material usage, function, energy consumption, etc.

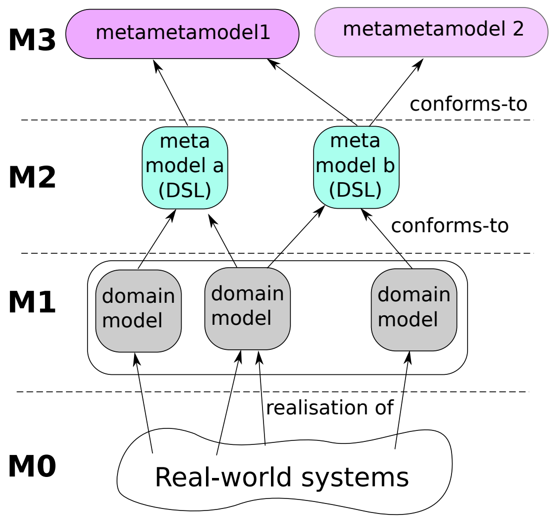


Figure 1.7: The partial order in meta models. A meta model language is often also called a Domain-Specific Language, or DSL. The mnemonics of the `M0`–`M3` abbreviations in the modelling order relation is that the `M` stands for "model", and the number represents the number of `M`'s.

1.7 Meta (meta) models and DSLs

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

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

A meta model language is often also called a **Domain-Specific Language**, or DSL (Fig. 1.7, and several domains are cooperating on making their own set of DSLs or schemas); the meta meta model languages are (typically) independent of a particular application domain, for example: mathematical models, or XML Schema. The *property graph* is this document's main meta meta model for formal modelling of entities and relations. and the mereological and topological representations introduced in earlier Sections of this document are at the meta model level. Later Chapters will apply such (meta) meta modelling to the context of models for the domain of robotics: property graphs, mereology and topology are all meta meta models for (robotics) domain-specific languages for kinematic chains, motion control, geometry, dynamical systems, physical units, etc.

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

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

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

1.8 Properties and attributes

Many modellers and developers have problems distinguishing the semantic differences between the "attributes" and the "properties" of their models and software. In the context of the Entity-Relation meta meta model (Sec. 1.2.1), the disctinction is trivial and it reflects well the etymology of the terms:

- **properties** of an Entity are the "values" that that Entity "possesses" or "owns", and without which the Entity has not meaning.

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

- **attributes** are the "values" that are given to an Entity by a Relation that it is involved in. More in particular, attributes are properties of the *role* of an Entity in the Relation.

  For example, the color of a physical object in a camera image depends on the interplay between its surface texture, the properties of its surface paint, the lighting conditions in the environment, and the properties of the camera. Or the current that flows through the physical object depends on the properties of the electrical circuit it is made part of.

In this document, there is no need for the term "attributes", because of the fundamental role played by the Entity-Relation duality.

1.9 Mechanism and policy

One of the major pragmatic problems to compose components into systems is that components are often provided with hard-coded *configuration* choices (Sec. 2.5). The reason most often being that the component was developed with just one particular application context in mind, and for which the chosen configuration was (hopefully) "optimal" and/or "obvious". So, another **mereological** aspect of component modelling to improve **composability** is to separate the description of a component's "mechanism" from the description of the "policy" with which that mechanism is used:

- **mechanism**: **what** does a component (algorithm, process, agent, piece of functionality,...) **do**, irrespective of the application in which it is used? Often, mechanism is subdivided into it **topological** sub-parts of structure and behaviour:
  - *structure*: how are the parts of the model/software connected together?
  - *behaviour*: what discrete and continuous "state changes" does each part realise?
- **policy**: **how** can the structure and behaviour of the component be **configured**, to adapt its functionality to the particular application it is used in? In its simplest form, a policy just configures some "magic number" parameters in the model/code of a library or component system; in more complicated forms, the whole architecture and interfaces of an application are optimized towards the particular application context.

1.9.1 Policy: frameworks, middleware, solvers

Two major instantiations of the coupling between *mechanism* and *policy*, in the domain of software engineering, are frameworks and middleware: they provide software libraries that

optimize the *"usability"* of the software in particular application contexts, by making already all of the policy choices that are relevant in that application context. The advantage is that the developers only have to make choices about the behaviour of their applications. The disadvantage is that these choices are most often hidden inside the framework, and hence **compromise the *"reusability"*** and composability of the framework/middleware, if one or more of the policy choices are not optimal (or even feasible) within a somewhat different application context.

1.9.2 Bad practice: interpreting attributes as properties

The concepts of "policy" and "attributes" are often encountered together, because the latter are, by definition, *always* the result of a policy decision. Here are some all too common examples (hence the name *bad practice*) where an attribute was set by a component designer *as if* it were a property of that component:

- a *priority* is *not* a *property* of a *thread*, but an *attribute* given by the *process* that composes the thread together with other activities.
- a control gain is *not* a *property* of a *controller*, but an *attribute* given by the *task* that needs the controller to improve a particular *task-dependent* performance metric.
- *resource allocation* is not to be done *in* a *Computation*, but *for* the component that requires the resource to help realise a *task-dependent* functionality.
- *colour* is not a property of an *object*, but of the *relation* that connects the material properties (texture, paint,. . .) of that object with the *lighting conditions* and the *visual perception properties* of a camera.
- *the* shape is not a property of a link in a kinematic chain, since various applications will require other shape representations for the same link; for example, to make fast computations with only a first-order accuracy, or to add a specific mesh for collision deformation simulation, etc.

1.10 Declarative and imperative

Models consist (Sec. 1.2.1) of *relations* between *entities*, with *constraints* between *properties* and *attributes* of entities and relations. Some of these relations or constraints represent **structure** (Sec. 1.4) between a set of other entities, relations, constraints, properties or attributes. For example:

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

In general, such set of structural relations can be represented by a graph in itself, a so-called dependency graph (Sec. 1.15). There are two major complementary ways to turn the information represented by such a dependency model into "actions":

- imperative: the ordering structure is determined explicitly, at design time. So, at run time, the determined "recipe" is executed, as is.

- declarative: the dependency graph is available at run time, together with (i) a solver program that *computes* the “optimal” ordering (by solving constraint satisfaction or constrained optimization problems), and (ii) a dispatcher program that executes the “actions” in the right **context**.

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

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

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

1.11 Composition and inheritance

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

Some not-so-good practices in this context:

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

1.12 Hierarchical and serial ordering: `scope`

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

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

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

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

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

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

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

- every topological relation *implies* a mereological one: it does not make sense to reason about interactions between entities if these entities have not been created and identified.

- every coordinate representation *implies* a geometric relation: one does not need to look at the exact numbers or data structure in the coordinate representation of frames to detect whether their type or their physical units are compatible or not.

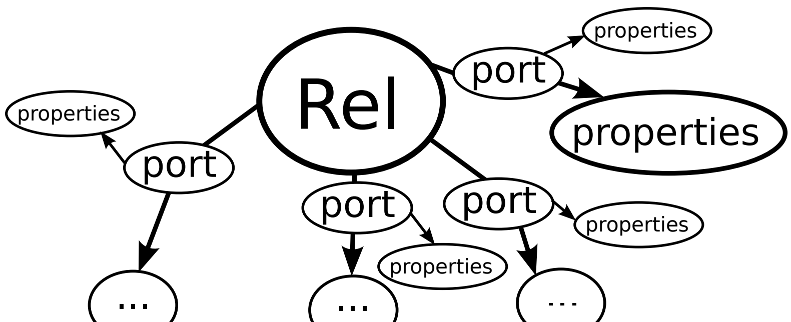


Figure 1.8: A directed graph representation of the **topological** model of the relation in Eq. (1.1). The arrows represent `connects` relations, which add extra structural information compared to the mereological model, namely the connection with a specific "port" entity that describes the *role* of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

1.13 Structural composition: `block-port-connector`

This formal representation of structural composition extends the generic Block-Port-Connector (BPC) meta meta model, [49]:

- **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.

1.14 Design patterns

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

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

The major top-level categories of patterns (in software, system development, modelling,. . .) are (i) the **structural** patterns, and (ii) the **behavioural** patterns. These are, not coincidentally, also two major categories of design aspects that appear often in this document. Since the latter's focus is on *model-driven engineering*, the above-mentioned key mereological aspects of patterns (solution, force, context) will be modelled explicitly, as well as the relations and constraints that connect them. A good pattern model balances the **declarative** and **imperative** (or "procedural") aspects of the description:

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

1.15 Dependency graphs

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

Taxonomy of increasingly more constraining dependencies: connection → relation → constraint → dependency → causality.

1.15.1 Mechanism: Directed Acyclic Graphs for partial ordering

1.15.2 Policy: temporal order, hierarchy, causality

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

Chapter 2

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

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

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

- **composability**: this is the extent to which a **component** makes all of its "5Cs" (see Sec. 2.5) **separately configurable**, to increase the opportunities **to reuse** the component in any kind of system.

- **compositionality**: this property of a **system** reflects the **predictability** of the behaviour of that system, hence making the system more easy **to use** as a composition of "components", as soon as the interconnections and the individual behaviours of the composing components are known.

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

The presented material is at the **meta meta model** level, and limited to only the **mereo-topological** parts[1] of **domain-specific** knowledge. The **behavioural levels of abstrac-**

[1]That is, the compsition of the mereological and topological parts.

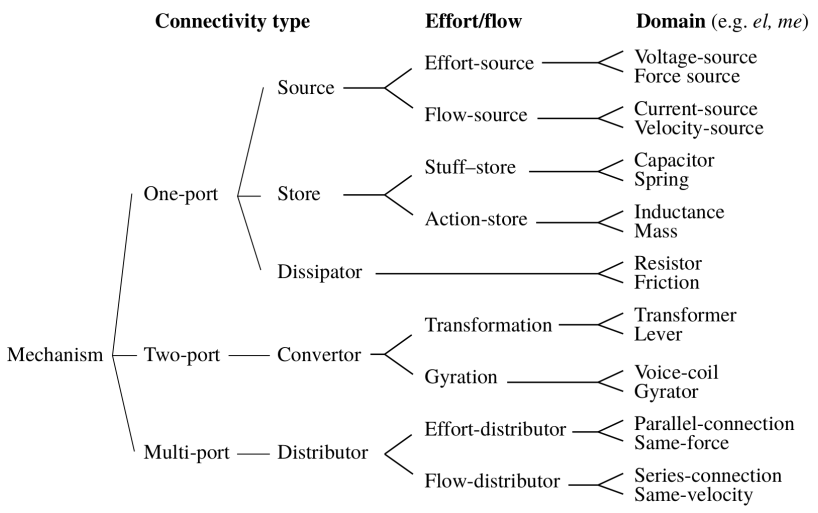


FIGURE 8. The taxonomy of physical mechanisms. The properties discriminating between the classes after branching are printed above the branch points. The classes on the right give some examples of mechanisms in the electrical and mechanical domain.

Figure 2.1: The topological relations between the various mereological top-level entities, according to [7].

tion will be added in Chapters 3 and following, where "robotics" will start to appear as a specific subset of cyber-physical systems.

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

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

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

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

2.2 Taxonomy of scientific theories

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

For the purposes of this document, the mathematical and information theoretic domains are highly relevant. Especially the former is a very mature domain, that comes with the expected structure; for example, allowing to differentiate between the geometric, algebraic and analytical parts of mathematical models. It is beyond the (current) scope of this document to formalize the taxonomy of scientific disciplines, but the structural relations will show up in many places, often in still too implicit ways.

(TODO: tree structure such as mathematics.geometry.Euclidean.SE(3), mathematics.geometry.differentialGeo
mathematics.analysis.ordinaryDifferentialEquation, mathematic.geometry.Euclidean.E(3).distance,. . .)

2.3 Levels of representation: mathematics, symbolic, software, digital and electronic data types

This document models cyber-physical systems that use computer control to realise desired relations between the behaviours of several interacting **physical** entities. The modelling requires **formal representation** of the entities and relations for, both, the physical and the informational parts of the cyber-physical system. A first **hierarchical structure**[2] that relates all formal representations is that of the following **levels of abstraction**:[3]

1. **mathematical representation**: the axiomatic definitions of relations between entities that respect particular **invariants** under **model transformations**. These representations are written down in **symbolic** form, meant to be produced and consumed by humans only.

   For example, the mathematical properties of *rigid body motion* are represented by the SE(3) group and its differential-geometric properties, such as pull-backs of tangent spaces and multi-linear forms, *exponentiation* or *logarithm*, or its lack of a bi-invariant

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

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

metric to measure the "magnitude" of a motion. Or, the mathematical properties of *uncertainty* are represented by Bayesian information theory and statistics.

These representations become useful to engineers as soon as the systems they have to develop cover so many different applications and domains that the **reuse** and **integration** of the same models, tools and software becomes effective or even necessary, and that they have to be implemented, documented and learned only once.

2. **abstract data type**: if one wants computers **to reason** with the properties of entities and the relations between them, one needs symbolic forms of knowledge-graphs. Such "reasoning" can take many forms: to formalise algorithms into computational models; to check formally the validity of a model; to transform models into code; to decide which are the right "magic numbers" in algorithms, etc.

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

3. **data structure**: the next step in bringing the models closer to executable software is to give the abstract data types an explicit *software representation*, that is, a particular **programming language** is chosen, and hence the corresponding representation of the *data structures* with which **to compute**. At this level of abstraction, also the **quantitative value** of each property is added, and its a corresponding physical **unit**; for example, meter, inch, or millimeter for *length*, Newton for *force*; second or hour for *time*.

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

   For example, the positions of a point in space can be stored as 32-bit IEEE floats, or communicated by JSON numerals.

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

2.4 Levels of representation: mature standards

Meta models for all these representations have already been developed (and to some extent also formalized) over the years, and several of those have matured into world-wide, vendor-independent standards. With the exception of QUDT, the formalizations have been developed for human developers only, and not for higher-order reasoning by computers.

2.4.1 QUDT

The *quantity-unit-dimension-type* meta model has already been formalized several times, for example in the **QUDT** initiative [62]. This ontology is nicely composable with the levels of abstraction hierarchy:

- mathematical and abstract data type representations: the **T**(ype) and **D**(imension) parts in QUDT are *linked* as semantic tags to each "type" of thing that one wants to represent. The *type* in QUDT is the same as the type in the mathematical and abstract data type representations; for example, the distance between two points in space, or Maxwell's equations. The *dimension* in QUDT adds annotations to (parts of) types such as `length`, `time`, or `voltage`.

- data types and digital representations: as soon as one makes a concrete choice of how to represent things concretely on computers, one automatically introduces the **Q**(uantity) and the (physical)**U**(nit) parts of QUDT.

- `T` and `D` always come together at the same level of abstraction, as do `Q` and `U`.

2.4.2 Differential geometry

Differential geometry (e.g., [9]) is the mathematical theory (including an unambiguous terminology and notation) of the geometry and dynamics of robotic systems (and other energy transforming cyber-physical systems), starting with the *manifold* of all positions, and the tangent and co-tangent spaces representing the velocities and forces, in the form of, respectively, the *tangent vectors* at a point on that manifold, and the *co-tangent vectors* (or *1-forms*) over each tangent space. The concept of a *jet* fits well to the geometrical combo of position, velocity and acceleration of the same point or rigid body.

2.4.3 HDF5

HDF5 and ASN.1 are internationally standardized formats, with a maturity similar to QUDT, offering meta model, models and reference implementations for all sorts of *abstract data type* representations and transformations.

2.4.4 FlatBuffers, Protocol Buffers

FlatBuffers and Protocol Buffers are more recent but well-supported alternatives; their designs have been optimized for runtime use, not for knowledge-based processing or *self-description*.

2.4.5 BLAS

BLAS and LAPACK are other mature ecosystems of models, tools and software, to provide the *linear algebra* aspects of representations and operations in geometry.

2.4.6 DFDL

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

2.5 The 5C's: composition of behavioural roles

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

The art of system engineering hence consists of finding the "right" granularity in component behaviours, and the "right" architecture of their interaction structure, such that complicated but realiable digital control systems can be created, offering predictable capability performance even though various components can come from **different, competing vendors**. A necessary condition for the latter expectation is that all models (of structure, behaviour and interaction) are built with neutral, open and formalized DSLs (Sec. 1.7).

The simple **5C**[4] meta meta model [42, 59] that any system design should conform to, contains the following **mereological** entities and relations:

1. **Computation**: the "**processing**" entities that realise the "**continuous**" behaviour of the component, that is, to transform the (streams of) input **data** into (streams of) output data and events.

2. **Coordination**: the "**logic**" entities that realise the component's "**discrete**" behaviour, that is, to process incoming **events** so as to decide whether or not the component has to change its Computation behaviour.

3. **Configuration**: the "**logistic**" entities that turn such behaviour-changing decisions into practice, taking into account the **constraints** imposed by the **reources** that the component is using. Indeed, most often a component can not change its behaviour from one execution time to the next, because the component relies on specific behaviours of other hardware and software components. Hence, configuration requests often trickle down to other components, and there is *asynchronous state* involved in the process of realising a (re)configuration.

4. **Communication**: the "**interaction**" entities that **exchange** data, events and models between **concurrently or asynchronously executing** components, respecting specific constraints on consistency, ordering, memory usage, timing, etc.

5. **Composition**: the "architectural" relations that link a given collection of components together into one or more "super" components (or "sub-systems", or "systems"), in such a way that all of the above-mentioned "Cs" are provided again to the "next" level of component/system composition.

[4]The 5C model of this Section has its acronym in common with that of [32]; while there is a thematic overlap, the meaning of the latter 5C model corresponds more to the "levels of abstraction" discussed in Sec. 1.3 and following.

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

2.5.1 Bad practices

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

(TODO: declarative, so only "don't do" relations can be given; no Configuration in Computation; no Computations in Coordination;)

2.6 Functions: algorithmic composition of computational behaviour

This Section provides the **mereological** and **topological** parts of a modelling language to describe algorithms as "composite functions", giving each of the traditional [63] parts of *data structure*, *(pure) function* and *control flow*. The description is independent of how the algorithm is deployed in different components, of its implementation mechanisms, and of the programming language used. The major added value in this meta model lies in the *separation of concerns* of representing data, functions and control flow *explicitly* (and hence *separately*), *including* the **dependency constraints** between them. Especially the *control flow* and *constraint* parts are seldom available as first-class citizens of a meta model; both are essential for solver algorithms with a large amount of runtime adaptability.

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

A mereo-topological representation of an algorithm is as follows: *"Starting from an initial state and initial input, the instructions describe a computation that, when executed, proceeds through a finite number of well-defined successive states, eventually producing output and terminating at a final ending state."* The mereological entities are "states" (i.e., data structures) and "instructions" (i.e., functions, or computations), and the relation is: "execution" (that is, a function links input data structures to output data structures); the topological relation is "progression through successive states" (i.e., the order in the *control flow*, or the "schedule", of the execution).

2.6.1 Mechanism: `Function`, `Data`, `Schedule`, and `Algorithm`

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

- `D-block`: the **Entity** to represent **data (structures)**, via a *data model* that describes what are valid structural compositions of data.
- `F-block`: the **Relation** that represent a **function**, that is, the computational element that has `D-blocks` as its arguments, with some of them having the role of "inputs", others of "outputs", and some have both roles. both roles).

  An `F-block` should be a **pure function**, that is, without only *explicitly visible* side-effects: the function will change the value of some of its data arguments, and nothing else.
- `S-block`: the **Relation** that represents the **scheduling** constraints (or *control flow*) of a collection of `F-blocks`, that is, their appropriate execution order.
- `A-block`: an **algorithm** is a composition **Relation** (or an "architecture") of all of the above, together with a `collection` of `dependency graphs` that represent **Constraint** relations; a dependency graph is a `collection` data structure (hence, a `D-block` in itself!) that contains the UIDs of the (i) composing blocks, (ii) the dependency graph relations, and (iii) its own semantic meta data.

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

- **causality**: a function that changes data values brings in a cause-and-effect relation, between the data "before" and "after" the application of the function. Since many functions and data structures represent the physical world, this algorithmic causality might or might not correspond to physical causality; this knowledge in itself brings in another relation.
- **function invariant**: the effect of the function on the data values can be modelled *imperatively* or *declaratively* (Sec. 1.10): the imperative form is that in which the function is, in itself, a composition of functions and a schedule; the declarative form is a **relation** that holds between the data values "before" and "after" the application of the function.
- **data consistency**: different functions can change different parts of the same data structure, and their function invariants must, *together*, satisfy some **relations** that *must* hold between all parts.

Together, all the constraints in an algorithm for a **constraint graph**. For example, the data structures that represent the "motion state" of a robotic kinematic chain only have consistent meaning *after* the full "inverse dynamics" algorithm has been executed.

All the above-mentioned entities and relations are *composable* (thus defining *higher-order relationships*) with some straigthforward composition constraints:

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

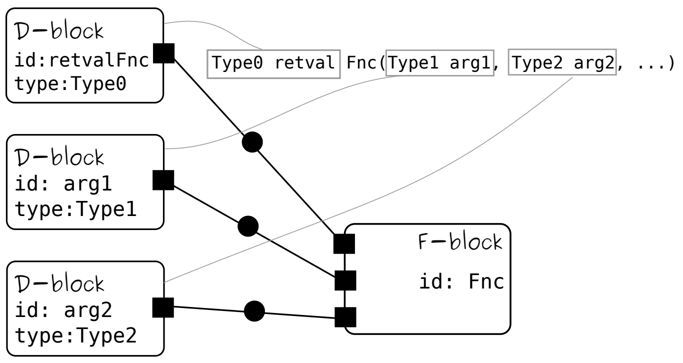


Figure 2.2: A function prototype and its graphical representation as a `F-block` connected to a set of `D-blocks`.

Other commonly used names for the `F-block` entity are: *(composite) operator*, *action*, or *actor*. `F-blocks` can easily be mapped to a *function prototype* ("signature") in any specific procedural or functional programming language. Arguments (input and output values) of the function are *ports* (in *Block-Port-Connector* terms) connected to `D-blocks`; Figure 2.2 shows an example. The structural part of the meta model `conforms-to` the Block-Port-Connector meta model (BPC): the domain is the description of an algorithm, and it is obtained by specialising the entity `block` to `F-block`, `D-block` and `S-block`, and introducing domain specific constraints (and meaning) to the `connects` relation. As an example, `ports` represent the arguments of a function, and they are typed; that is they can be connected to a `D-block` under the constraint that the digital data representation model of the `D-block` and the port are compatible. The `connector` that hosts a data access constraint has an extra attribute which indicates if the access to the data represented by the `D-block` is read-only, write-only or both (i.e., if the argument is input or output of a modeled function). Since multiple `F-blocks` can share a data access constraint to a `D-block`, the latter influences the execution order of the `F-blocks`: the execution of an `F-block` that has write access to a `D-block` prevents other `F-blocks` from being executed if they are also connected to the same `D-block`. Therefore, the data access constraint is a *declarative* form to define concurrency properties of the modeled algorithm.

2.6.2 Policy: closure

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

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

2.6.3 Policy: deployment in BPC component

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

- *encapsulation*, *information hiding*, *security*. The representation of these various forms of data *"protection"* is possible by dedicated constraints on the accessibility of the `D-blocks`. This is best done by composition of the algorithm's `A-block` with a BPC model in which some `Ports` are `connected` to selected `D-blocks`, `F-blocks` and/or `S-blocks`, and those `connect` relations get attributes that model the kind of "protection" to be composed in.

  In other words, the *closure* of the algorithm is not defined by the functional developer, at design-time, but by the component supplier, who provides the port-based view on the algorithm that fits best to the concrete context in which its functionality is to be used.

- *run time adaptability*: there is no hard technical constraint that prevents the just-mentioned port-based *views* to be adapted at runtime.

- *resource management*: the execution of an algorithm makes use of two resources of the computer hardware, namely its *memory* and its *CPU*. Because the meta model contains explicit and complete information about the memory requirements (size as well as access constraints) of *D-blocks* and the execution sequences of *F-blocks*, an application developer can provide tooling to deal with possible exhaustion of those resources; e.g., the various management policies for data buffers or stacks, and for iterators or execution schedules.

2.6.4 Policy: application programming interface

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

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

2.6.5 Policy: dataflow

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

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

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

2.6.6 Policy: functional programming

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

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

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

2.6.7 Policy: pipe line

The **pipe line** pattern is the simplest form of composition, and it can be considered as a boundary case for, both, dataflow programming and functional programming, since there is just a *linear* dependency between data buffers and function invocations.

2.7 Activities: composition of behaviour to control resources

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

- `function`: the data, function and control flow entities and relations of Sec. 2.6, and which form the foundation of all **computations** in software components.
- `algorithm`: the **composition** of several `functions` and `function` schedules which share data in a fully **synchronous** way. That is, the functions are scheduled in a **serialized** way, and all data is available exclusively to any function in any schedule.
- `program`: the **composition** of several `algorithms`, some of which may be executed concurrently. That is, care is to be taken that one function does not write data at the same time that another one is reading that data, since this **asynchronous** execution of functions can introduce inconsistencies in the data they share amongst themselves.
- `activity`: the **composition** of a `program` with the **ownership** of a particular **resource**. *Ownership* means that:
  - the values of the properties of the resource's model can only be changed by functions in the activity.
  - other activities that need access to the resource must do so via **queryies** to the owning activity.
  - *"the"* state of a resources exists only in its owning activity, and other activities just have *copies* of that data, which can be mutually inconsistent.

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

2.7.1 Mechanism (meta model): event loop computations

The *computational architecture* that fits betst to the `activity` meta model is that of the event loop, that is, the combination of synchronous and asynchronous *communications* and *computations*. The event loop pattern is so important in itself that is is explained in more details in Sec. 2.11.

2.7.2 Mechanism (operating system): thread, process, container, cloud

The `activity` implements computations, in the broad sense of the word, independent of how those computations are executed. An `activity` is **deployed** by an operating system to get access to the hardware resources that it must share with other `activities`, and for which the operating system is the *owning* activity. This sharing is realised in the following ways:

- thread: the thread[5] is the smallest operating system mechanism to make sure that the accesses (by the composition of all `activities` that it is runnning) to the CPU(s) and the RAM always satisfy the access constraints specified by each individual program.

[5]More detailed introductions can be found here and here.

A fiber is a special type of thread, in that the operating system uses cooperative multitasking to determine when to execute which fiber, while threads are scheduled via preemptive multitasking.

A coroutine is another related concept, providing cooperative multitasking, but this time organised via a programming language runtime (e.g., Lua or Go) and not the operating system.

- process: the **composition** of the execution of several threads, and coordinates the stacks of all its threads, to make sure that each program executes correctly irrespective of the number of times its execution has been preempted.
- container: has a similar **composition** role as processes for threads, but now between operating systems and the computational hardware (which continues below in this hierarchy list).
- cloud: **composes** several computers, and coordinates their internet communication interactions.

These entities will not be detailed further in this document, because there is no specific connection between those entiities and the properties of robotic and cyber-physical systems, *at the level of knowledge relations*. The entities and relations in this Section, together with those of the following Section, have already received significant meta modelling attention, for example in the AADL standard and supporting software and tools. (Unfortunately, AADL violates almost all "best practices" advocated in this document, such as: separation of mechanism and policy, inversion of control, or composability.)

2.7.3 Mechanism (hardware): core, system-on-a-chip, computer, cloud

Even further away from the properties of robotic and cyber-physical systems is the *computer hardware*, which has the following parts: `CPU`, `memory`, `bus`, `I/O`, and `network`. The CPU part comes in some variants on most modern hardware, depending on the increasing degree of sharing memory ("caches" and "RAM") and communication hardware:

- core: one CPU with some local "cache" memory that it completely under its own control.
- `processor`: a collection of `cores` that share some caches, and some buses to the RAM memory.
- System on a Chip (or *"computer"*): composes several cores and peripheral hardware on one single integrated circuit, and coordinates their access to hardware shared communication busses.

2.7.4 Policy: sequential, concurrent and distributed execution

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

- *restricted* by *constraints* on (i) access order between data structures, and (ii) execution order between functions.

- *relaxed* by *assumptions* that (i) an explicitly specified part of the data structures are not changed by "the outside world", and (ii) an explicitly specified part of the functions have no side effects.

Activities compose algorithms, in two complementary ways:

- *serialize* them: multiple algorithms can be executed in the same program, and the program designer must make choices about the order in which the various algorithms are executed.

- *iterate* them: it is a very common use case in cyber-physical and robotic systems to executed an algorithm or program repeatedly over time, at more or less fixed time intervals, in infinite or finite loops.

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

- the algorithm designer must minimize the amount of constraints on the order of data access by functions in the algorithm, while still guaranteeing the correctness of the intended behaviour.

- the program designers must take into account all constraints of data access and function execution order that come with the algorithms they have to compose, and choise serialization and iteration policies that respect them.

While algorithm and program designers take decisions about *concurrency*, the designers of the deployment of programs in threads, processes, cores, SoCs and clouds must take **composable** decisions about:

- **parallelization** of their functions over several computational cores connected by CPU busses and caches. The policy choices are determined by optimising which executions can run at the same time and still share some data.

- **distribution** over several computers connected with an local or wide area network. The policy choices are determined by optimising which executions can be offloaded to different computers, without compromising the performance of the data exchange between programs.

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

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

2.7.5 Policy: work flow

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

2.7.6 Mechanism: Conflict-Free Replicated Data Type (CRDT)

(TODO: different activities can concurrently change data structures, and the changes are merged and distributed automatically without the need for a central server which "owns" the data structure; only some data structures have the CRDT property.)

2.7.7 Policy: immutable data type

(TODO: concurrency becomes a lot easier to make predictable if any data that is created new will never have to change anymore, because *reading* of data can never lead to inconsistencies; explain where this particular type of CRDT makes sense in system architectures, and when (not) to use it. Aspects of *garbage collection* and *compaction*.)

2.8 Interfaces: composition of behaviour to control interaction between activities

Physical activities interact by exchanging energy; information activities interact by exchanging `messages` with each other, so each of them has *to produce* and *to react to* ("to consume") such messages. The software representations of activities have **to control the behaviour** of the resources they own, but, depending on their role in the system, they must also influence each others' behaviour by interaction. Such interaction is realised via interfaces, and this Section explains, at the **mereo-topological** level, how the producer/consumer communication pattern is composed with the activity pattern, to lead to a composite pattern, the `Stream`, with which **to control the interactions** between activities.

2.8.1 Policy: exchange of data, events, and models

A first aspect of modelling the *behaviour* of an interface is the modelling of the *semantics* of the *data structures* that are exchanged in the interaction between two or more activities. The following three semantic types cover most use cases:

- data structures: activities must be able to influence the **data** with which each other's internal **functions** work.
- events: activities must be able to influence each other's **control flow**. This typically happens via sending events, that is, minimal data topics whose only semantics is that "something has happened".
- queries and dialogues with models: activities must be able **to discover**, **to configure** and **to update** ach other's **functions** at **runtime**, that is, to find out the current or possible behaviour of the other one, and to provide new "computations" and new "data types" to each other. This typically happens via a **query protocol**, that composes a particularly ordered series of events and data.

Of course, just from the point of view of the act of communication, *events* and *models* are just special cases of *data*.

2.8.2 Mechanism (software): `buffer`, `queue`, `socket`

The fundamental property of the mechanism behind interfaces is that one program is given access to data structures in another program. Many policies exist about how that access is realised. But *the* mechanism that is used in all policies is that of the pointer, that is, the address in the RAM memory where the data structure is stored. (In practice, this mechanism is often burried inside libraries and/or the operating system.) There are three ways the pointer mechanism is being used:

- **address pointer**: both programs share the same address space as the interface data structure, and can use the address pointer directly in their own code, via a `buffer` they have both access to.

  The *advantage* of the shared memory mechanism is that the data need not be copied.

  The *disadvantage* is that it imposes a *data access constraint* to all programs that share the data structure: the execution of the functions that use the data structure should be interleaved, in one way or another.

- **message queue**: this mechanism also works between programs in the same address space, but both programs copy the data structure to and from its own private address space to a buffer in the operating system, the so-called message queue, with `write` and `read` operations. These functions hide the pointer-based access inside a library; so, the data access constraint is still there.

  The *advantage* of a message queue is that the developers of the reading and writing programs need not bother about the data access constraint.

  The *disadvantage* is that the execution of each program has side effects: higher memory usage because of the data copying, and non-deterministic pre-emption of their execution, because the "locking" of the buffer pointers is happening implicitly.

- **socket**: when programs do not share directly accessible memory,[6] the message queue mechanism is extended even further: the "local" message queue is not really one single queue, but the data that is written to it is sent to another computer or process via a communication channel, where it is copied to the local message queue of the program there. A common name for the data access with this mechanism is `send` and `receive`.

  The *advantage* is the same as for message queues (implicit satisfaction of the data access constraint. For so-called embarrassingly parallel applications, it *might* be that the overall time to do computations is reduced, since more cores can be used at the same time and the relative costs of communicating the data structures is dwarfed by the time needed for computations on the data.

  The *disadvantage* is that the communication *overhead cost* is increased: the data must be copied several times; communication takes longer the "further apart" the hardware cores that execute the communicating processes; the channel between both processes must be kept alive in a socket. A second disadvantage is that *"the" state* of a data structure does not exist, because of the many copies that are very difficult to keep consistent, all the time.

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

The operating system on a computer is the natural place to embed all of the above-mentioned mechanisms, exactly *because* it is "just" mechanism that applications can and should compose with their specific policies. A major aim for "mechanism" developments at the platform level is to strive for (i) standardization, (ii) performance, (iii) resilience, and (iv) security. Realising these competing goals together requires a huge effort, which is another reason to share this effort by all stakeholders of the platform. A key example illustrating this context in the case of operating systems is the D-bus inter-process communication project; or the WebRTC project in the context of multi-media streaming over "the Internet". In robotics, the ROS project tries to achieve the same role, but it has not yet reached industry-grade maturity in all of the four design goals mentioned above.

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

The choice of communication pattern (or "communication protocol") that activities use **to order** their interactions is an important policy in the interface meta model. The *forces* that drive the pattern trade-off in different directions are: number and anonymity of participants and channels; longevity of the interface; being able to support "dialogues"; level of mediation between several channels and of the traffic inside one channel. Two major versions have matured over the last 50 years:

Publish-Subscribe	**Producer-Consumer**
topic centred	message centred
one session per topic type	one session per Producer-Consumer pair. Various topic types can be arbitrarily marshalled per message
multiple consumers	single Consumer
one-way: send-and-forget	two-way: backpressure and polling stream
anonymous peers	peers know each other's identity
topic distribution outsourced to broker	peers responsible for message delivery to each other
Quality of Service by brokerage middleware	Quality of Service by application

The **Request-Reply** pattern is the simplest "hybrid" between both, and hence merits to be identified as a third major pattern.

Communication patterns are extensively dealt with in general-purpose ICT domains, and they are hence also very mature and varied. Detailed discussions are beyond the scope of this document, and the interested reader is refered to the literature, such as the ZeroMQ documentation, for the differences between the various patterns.

2.8.4 Mechanism: Producer-Consumer interaction via stream handling

The Producer-Consumer pattern has become prevalent in "the Web", and the particular mechanism of the `stream` has emerged as its reference realisation. And at the time of writing, the WHATWG stream standard is reaching maturity as the meta model for all implementations.

The **mereological** parts of the `stream` meta model are: `sink`, `source`, `chunk`, `backpressure`, `buffer`, `producer`, `consumer`, `readable_stream` and `writable_stream`. Also many established internet protocols are special cases of the `Stream` meta model, for example: SCTP at the application layer, RTSP at the transport layer. The **topological** parts are rather obvious: a source is connected to a Producer, that is connected to a buffer (the `stream_queue`), that is connected to a Consumer, that is connected to a sink. The **behavioural** part is that of the **(re)active** execution of the following **stream handling** by Producer and Consumer:

- `stream_queue`: the Producer fills the queue buffer, and the Consumer empties it.
- `stream_processing`: one `stream_queue` can have multiple Consumers, each having read access to all the elements in the buffer. The typical application is that every Consumer processes the elements in one or more stream buffers, and is the Producer for one or more output streams itself.
- `stream_monitoring`: both Producer and Consumer control the overflow/underflow of the shared stream queue buffer together, via events to indicate potential overflow and underflow. These are the `backpressure` (*"please, stop sending for a while"*) and `polling` events (*"please, send some more chunks"*) . The Producer uses them to inform the source, and the Consumer uses them to inform the Producer.

Software realisations of this meta model are the event loop (Sec. 2.11) and the *ringbuffer* (Sec. 11.1.5).

2.8.5 Policy: heartbeat, watchdog

(TODO: to let one activity inform other activities about its liveliness, and the freshness of data; resets values in abstract data type at particular frequencies and in an asynchronous way, so that an activity that must share them with other activities is informed about whether the other activities have been providing their data "fast enough".)

2.8.6 Bad practices

- to assume that messages will be delivered *exactly once.*
- to use only the *publish-subscribe* communication policy to exchange events, data and models: events require most often a *broadcasting* policy, and queries about models require one-on-one *dialogues*, and neither are done well best with pub-sub.
- to neglect the CAP and PACELC theorems, that state that the three following networking assumptions can be satisfied at the same time: Consistency, Availability, and Partition tolerance, and that Consistency and Latency are also always contradicting requirements.
- to neglect the fallacies of distributed computing, *even* between processes on the same computer.
- to communicate *state* information back and forth between distributed components, while it would be possible to deploy all the components' functionalities into the same process and store the shared state in a shared data structure.

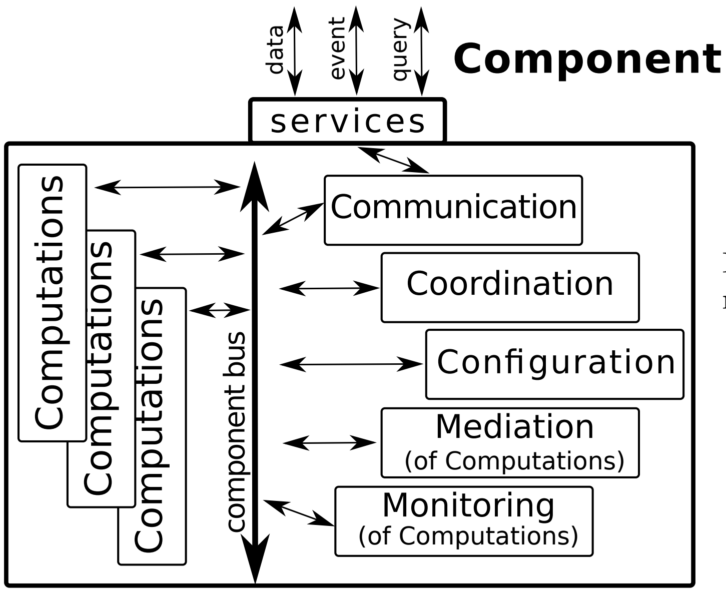


Figure 2.3: The mereo-topological model of the *Component* primitive.

2.9 Components: composition patterns for software activities

Every digital platform consists of a (potentially large) number of components, providing "*services*" to each other. This Section provides a meta model of the architecture of such components, designed for this distributed context, with **dynamically changing** runtime lifecycles of, both, the servicing components and the application being served. The design allows:

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

2.9.1 Mechanism: "5Cs"

The parts[7] of the presented meta model (Fig. 2.3) have the following **responsibilities** (or *"roles"*):

- **4Cs**: *Coordination*, *Configuration*, *Communication*, *Computation*. Section 2.5 has already explained what these are. Typically, *multiple* Computations can be deployed in the same Component, but each of the other three roles must be a singleton.
- **Component bus**: while the role of the *Communication* component is to realise the exchange of data, events and/or models with other *Components* (software and hardware),

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

the *Component*'s internal exchange of information can be organized differently as one or more software busses.

- **Monitoring**: each monitoring component is a Computational component by itself, but it has a "higher-order" role, in that:
  - it exchanges data with one or more of the computational components that are responsible for the *services* that the *Component* must provide,
  - to compute the **Quality of Service** (QoS) with which these computations are realizing their *expected* service behaviour, and
  - **to fire an event** when particular QoS thresholds are being reached.
- **Mediation**: each mediation is a Computational component by itself, but it has a "higher-order" role complementary to the *Monitors*:
  - it exchanges data with the *Component*'s *Configuration*,
  - because it has the extra **application-specific** knowledge about how various *Computations* are to be **traded-off** when the QoS of the *Component* goes beyond its configured thresholds, and
  - then to take the **decision** to trigger the *Configuration* component **to reconfigure** some of the *Components* to react according to the application trade-offs configured in itself.

There can be multiple *Monitoring* and/or *Mediator* components in each *Component* model, the same *Monitor* and/or *Mediator* can get data from several *Computations*, and the same *Computation* can provide data to several *Monitors* and/or *Mediators*.

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

- the concrete models of systems can be **pre-processed before configuration/deployment** to optimize, for example, the amount of *Communication*, *Configuration* and/or *Coordination* components that are needed in a (sub)system.
- the architecture of a deployed (sub)system can be **adapted at runtime**, *if* (i) the models of all *Components* are available, and (ii) the *Configuration* component can deal with online *queries* for re-composotion.
- the model is an excellent (because structured and explicit) **documentation** for human developers to discuss the system design, and its trade-offs.

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

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

- **separation of roles** in the digital platform]platform: this has been explained in Sec. 2.5.

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

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

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

The *Component* meta model has a handful of different roles, but Fig. 2.3 already hints at the fact that the amount of models and code that will eventually have to be used for all components is is, by far, concentrated in the *Computations*. The other component can have very little content, but that content is the one where vendors can make the difference between the *generic* service implementations and their unqique selling point and commercial added value.

2.9.4 Policy: Coordination, Orchestration, Choreography

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

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

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

Section 2.9 presents the *platform-centric* meta model for *Components*, whose ambition is to create composable software *services* (irrespective of the application domain the services are designed for) that are to be deployed in operating system processes, on top of hardware *resources*; Sections 2.6, 2.7, 2.8 and 2.11 present how to interface *functionalities* in *event loops*, executed in *activities*, and deployed in threads and processes to provide the services accessible at the ports of software components. All of the above are *generic*, i.e., application and domain independent, and this Section presents that last missing domain/application-specific piece, namely the **mereo-topology** of how an **application** must represent entities and relations of its domain by means of the **task meta model** (Fig. 2.4). That is, to model:

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

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

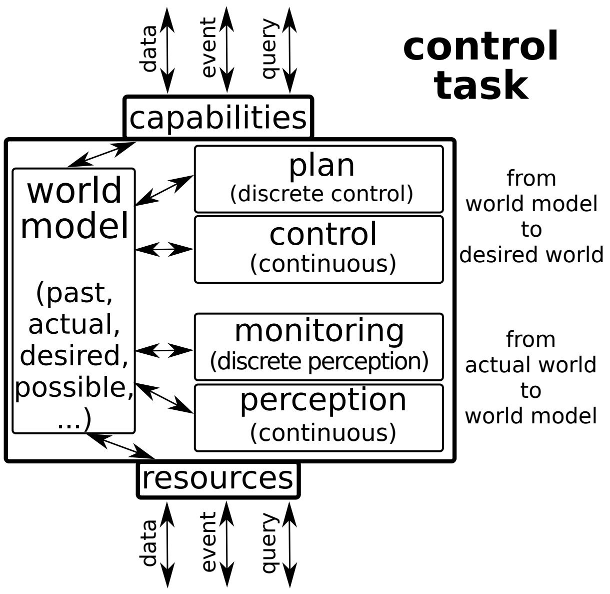


Figure 2.4: The mereo-topological model of the *Task* primitive. Note that the drawing does *not* represent a software component, but a *model of the composition* of the parts needed in a Task, and of where interactions between these parts occur. (Of course, the software components that will, eventually, implement Task executions, will be *structured* along the Task model structure as much as possible.)

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

Figure 2.4 shows the mereo-topological graph of the Task meta model. The **mereological** part of that Task meta model has the following types of **entities** (and nothing more!):

- `resource`: the *constraints* of what can be provided by the resources that are necessary for the execution of the modelled task. For example, mechanical strength, energy availability, computational and communication hardware properties, etc., together with the *quality of service* metrics which represent how well the resources are being used.

- `capability`: the *constraints* of what the task execution can deliver to users, together with the *quality of service* metrics which represent how well those capabilities are being provided.

- `world-model`: all the **information** that **needs to be shared** by all the other models in a `Task`. In other words, it contains the extra information that represents the *context* in which these other models are used; e.g., the information about how the "magic numbers" in all these models are connected to each other for each specific set of capability requirements and resource constraints, and how these are related to information about the *physical* world.

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

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

- `plan`: the model of which "motions" (or "actions", in general) to be executed in which sequence, and under which conditions. The `plan` is the **discrete** version of the **control**, that is, the one for which the actions are triggered by *events* via which the `task` tries to go from the actual model of the world to a desired world model.

- `control`: the model of how to realise the plan, in the actual world, and with the actual task requirements and resource constraints. This model contains the **continuous** aspects of **control**, that is, the one for which the actions are triggered by *data* streams.

- `monitor`: the model of the relations on world model parameters with which to check whether the *actual* `task` behaviour corresponds sufficiently enough to the *expected* behaviour. This is the **discrete** version of **perception**, that is, the one that triggers *events* in the `task` behaviour.

- `perception`: the model of how sensors can provide the *actual* information needed to create and/or update the **continuous** parameters in the world model.

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

- **separation of concerns via world model**: this constraint on the allowed connections between the mereological parts in a `Task` meta model, is a **major design axiom**, because it implies that all interfacing between `plan`, `control`, `monitor` and `perception` functionalities takes place only indirectly, via interactions with only the `world model`.

In other words, the world model is the place to store all the **state**, that is, the information that one must remember from the past, to act in the present, and to predict the future.

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

- **data, event and query** interaction models, to support the information interaction required for, respectively, inter-agent streaming, asynchronous reactive broadcasting, and discovery and configuration of inter-agent cooperative behaviour. (See Sec. 2.8 for more details.)

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

Most applications have to support multiple `tasks`, to multiple users, and with dynamically changing requirements for both. Hence, the task primitive mechanism must allow various *composition policies*; Figure 2.5 is an illustration of one such composition over multiple of the levels of abstraction discussed in this document, and the next Section explains the trade-offs available in composition.

2.10.2 Task meta model as semantic database

For any somewhat realistic application, composite task models as in Fig. 2.5 will contain several hundreds to several thousands of entities, relations and constraints, so it is appropriate to call it a "**semantic database**" (Sec. 1.2). The added value with respect to a normal database is:

- the parameters in the "data" are linked together with relations that have semantic meaning, via the higher-order interconnections between them.
- queries on the database can (hence) use semantic terms reflecting the **intention**, **causality** or **dependency** that holds between query arguments. In other words, one can get explanations about *why* the query yields the results it does, or about *the context* in which the answer holds.
- as a further result of the semantic contents of the database links, knowledge can be exploited also in the computation of the query answer, because **graph traversal** becomes possible instead of the less efficient but more general *graph matching*. The latter is the default "solver" for relational database, while the former becomes more and more standard in graph databases.

In the context of cyber-physical systems, and robotics, this means that the **coupling** between **control** and **perception** can be adapted to the context provided by (i) the expected capabilities, (ii) the available resources, and (iii) the past, actual, and expected state of the environment.

For example, a robot controller can switch its perception algorithms depending on the knowledge it has about which features in the environment fit best to, both, the available

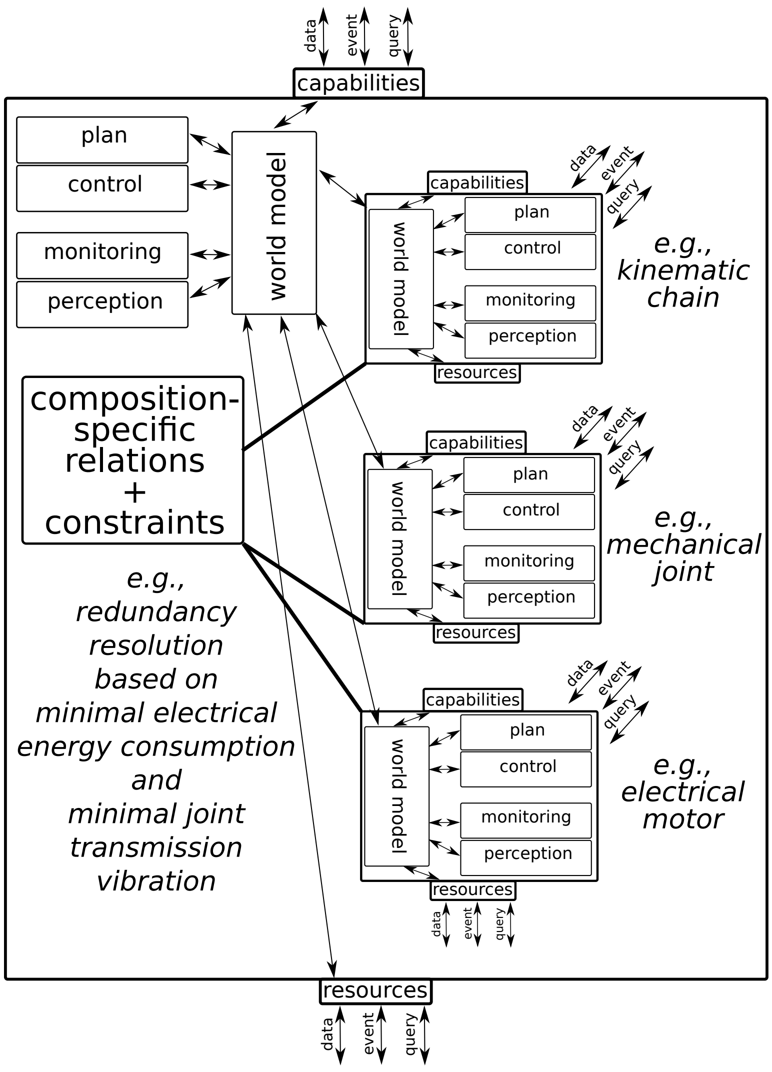

data
event
query
capabilities
plan
control
monitoring
perception
world model
composition-specific relations + constraints
e.g., redundancy resolution based on minimal electrical energy consumption and minimal joint transmission vibration
e.g., kinematic chain
e.g., mechanical joint
e.g., electrical motor
resources

Figure 2.5: Composite `task`, over multiple levels of task specification, linked to three of the most common *levels of abstraction* of a robot's kinematic chain. The `task` *composition pattern* is repeated at the composite level, adding the **knowledge** about which additions to make at the composite level, and which interconnections (constraints, new relations) to add with the parts that are already present in the composed `task` models.
Again, note that the figure represents the composition of *knowledge relations*, and not that of software components.

sensors and sensor processing software, and the required feedback and monitoring in the motion control loops. More concretely, when driving through a corridor in a hospital or office building, the robot can actively search for the "semantic tags" that have been put in the building with the explicit purpose of guiding its users towards the various destinations; a similar situation holds for all outdoor navigation tasks where traffic signs and signalling are available, such as in car and truck driving, or plane and helicopter take-off and landing.

2.10.3 Policy: coupling via shared world model

World models are *designed* to decouple all "internal" activities in Tasks, so the most deterministic way to integrate several Tasks is by sharing and coordinating selected parts of the "composite" and "component" world models. One end of the sharing spectrum is realised by a blackboard architecture: all activities that have to share world model information, read and write it on the "**same "map"**", so activities can see everything from each other. The other end of the sharing spectrum has no shared world model at all: all activities have their own internal world model, and exchange world model information via **communication**, one-on-one and on a *need to know* basis. The "forces" that determine where to position an application in this spectrum are: communication cost; model consistency; robustness against loss of interaction; specialisation of component functionalities.

2.10.4 Policy: continuous, discrete and symbolic Task models

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

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

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

2.10.5 Policy: hybrid constrained optimization problem specification (HCOP)

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

2.10.6 Policy: declarative solvers and imperative algorithms

Computations that use constraint-based solvers are **designed** to be more deterministically composable than ones with only imperative algorithms.
(TODO: a solver generates a *plan* at runtime, with a more limited horizon (in time, space, resources, capabilities) than the *Task* in whose context it works. The different policies in this respect model which solver and which horizon to choose, under which contextual situation.)

2.10.7 Policy: sharing data, events, models

Communication infrastructure is **designed** to be shared by many processes and/or computers, so a lot can be gained by deploying several task-level "client-to-server" communication channels onto the same infrastructure channel, and to do the same with the data, event and/or query messages that are to be exchanged through these channels. "Queries" are, in fact, complete *models* that are being exchanged between components, allowing for higher levels of declarative interactions. For example, the success of the "Web" is based on the fact that full HTML models are communicated, which in itself composes other web standard formats, such as SVG or JPEG; even when a receiver can not "render" the full model, the composition semantics is clear enough (i) to allow local decision making about what to render or not, and (ii) to communicate back a "status report" to the sender explaining which parts of the sent model gave problems.

2.10.8 Policy: mission, service, skill, function

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

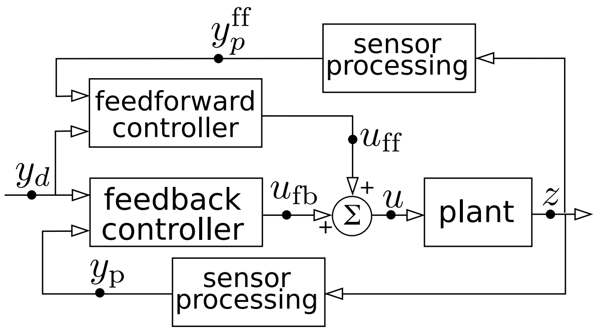


Figure 2.6: Feedback and feedforward control both contribute to the actuation signal u that is given to the plant. And they both can work with *processed* versions (y_p and $y_p^{\rm ff}$) of the current state z of the plant.

2.10.9 Vertical and horizontal composition

The Task meta model is a major part of the modelling methodology of this document. This Section explains how its design makes it a candidate for composition, both "horizontally" and "vertically".

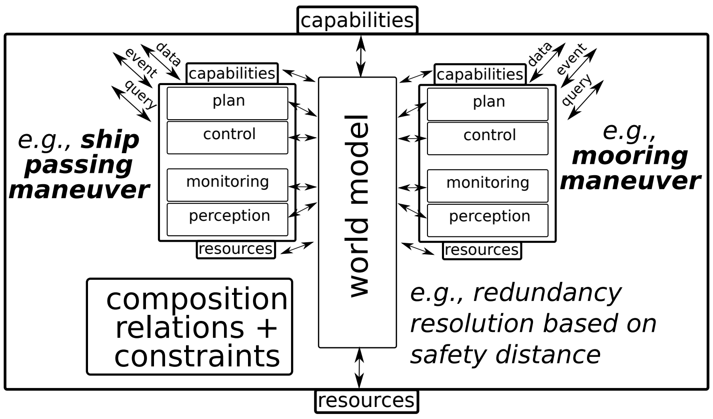


Figure 2.7: Horizontal composition of Task models, in an inlans waterways shipping context.

Horizontal composition — Task activities **share** *at least* a part of the world model, but in most cases also parts of perception, control and monitoring, Fig. 2.7. This results in:

- access to **shared resources** (such as the world model) must be coordinated between activities.
- if (the execution of) the plan, control, perception and monitoring models is done by multiple activities, their internal "state machines" must be adapted to guarantee this coordination.
- constraints and objective functions are *fused* in some parts of the Task specification's *plan*.

- tolerances might be adapted to the specific context of a specific composition, and hence the monitors that are connected to checking the tolerance violations.

Overall, the same HCOP-based control methodology applies as for the composed Tasks individually, and in many cases "all" that has to be changed are the *Coordination* and *Configuration* parts (Sec. 2.5) in the system's architecture.

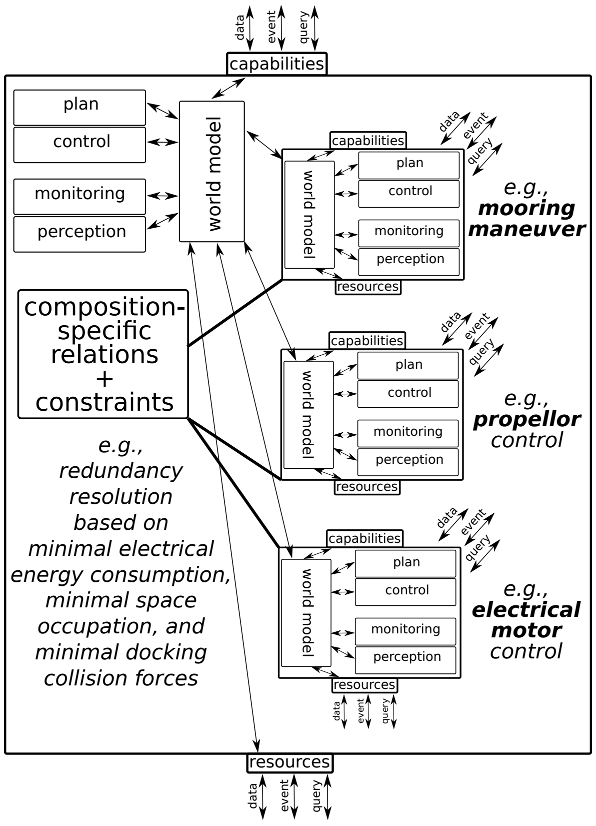

data
event
query
capabilities
plan
control
monitoring
perception
world model
composition-specific relations + constraints
e.g., redundancy resolution based on minimal electrical energy consumption, minimal space occupation, and minimal docking collision forces
e.g., mooring maneuver
e.g., propellor control
e.g., electrical motor control
resources

Figure 2.8: Vertical composition of Task models.

Vertical composition — Tasks **add** to each other's models, Fig. 2.8:

- *higher level* adds constraints and objective functions to *lower level*'s COP.
- and vice versa.
- there is a need to introduce *extra* constraints and objective functions because of the *coupling*.
- hence, also extra tolerances and monitors are *needed*.

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

The following terminology is sometimes used to refer to particular vertical composition levels:

- **strategic**: decisions about what investments in resources are needed to create profit.

- **operational**: decisions about which existing resources to deploy to create required capabilities.
- **supervision**: decision about accepting the performace of provided capabilities, or to adapt them.
- **coordination**: execution of capabilities, monitoring, and reconfiguration.
- **control**: continuous time execution control.

2.11 Event loop pattern: composition of asynchronous computational behaviour

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

2.11.1 The role of the event loop

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

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

- **data consistency in algorithms**. Many *Computations* rely on guarantees that the data they work with is changed only in deterministic ways, under their own full control. For example, there is a lot of *state* in task planning, perception, or discrete and continuous control algorithms. Hence, access of the computational functions to the data must take place in a *synchronous* way: the order in which the function operators are programmed is also the order of the changes to the data they manipulate, *and* there is no other function changing the data "behind their back".

- **localising the synchronization of the *side effect-full* data exchange**. Side effects inevitbly take place, via (hardware and software) mechanisms like interrupt handlers, mutexes, condition variables or "lock-free" and "wait-free" buffers, etc. The pattern's solution is *to copy* the data from/into asynchronous sources into/from a "thread-local" storage to which only the event loop thread has access. The above-mentioned mechanisms are explicitly recognizable in the programme code, so that it at least *possible* to identify the areas in the code where side-effects can not be avoided.

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

- **event handling latencies**. There are almost no robotic applications in which asynchronous I/O is not present, because lots of sensors and actuators have to be interfaced, and processes must communicate. The event loop pattern provides application developers with one callback object per I/O channel, and has a mechanism (i) to select which I/O channels to deal with at any particular iteration through its "loop", and (ii) to configure how long it wants to wait on the channel.

- **prioritization of event handling**. The above-mentioned selection of I/O channels is done with priority queues, for which the configuration is again under the explicit control of the application developers.

- **off-loading of asynchronous processing**. The application developers can configure a thread pool of "*worker threads*", in addition to the "*main thread*", to let each long-running event handler be dealt with in a separate "worker", and to make the results accessible to the main thread via a message queue.

2.11.2 "5C"-based programme template for an event loop

A **mereo-topological**[8] version of the event loop, using only component-centric 5C entities, is given in the following pseudo code:

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

[8]This model represents *behaviour*, as a mereo-toplogical composition of 5C entities, which represent *types* of behaviour themselves.

```
                // decide which ones to react to.
  configure()   // some events imply reconfiguration of computations.

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

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

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

The sequence of functions in the code above is not a hard constraint, and the exact selection and serialization order of these functions in an event loop is to be configured by the application developer. Each application context indeed comes with a set of dependencies between the order in which computations can be executed, and those *declarative* constraints are to be "solved" (in principle, every time the event loop is triggered) to create a *procedural* **schedule** (Sec. 2.6) that encodes the *bookkeeping structure* of the algorithmic **control flow** in the event loop.

The power of the event loop pattern, as described in the code above, is that it *stimulates* developers to separate the 5C concerns in an extremely simple way. That way has also proven[9] *to facilitate* (but not to guarantee) composability and compositionality in complex distributed systems. This composition of multiple event loops into one single larger event loop expects that all algorithms inside the event loops can be created by means of coroutines, and not of preemptive multi-tasking. One necessary, but not sufficient, condition to satisfy this constraint is that all functions in the algorithms have no side effects, and access only data that is local to the event loop. Often, that data is stored in the buffers[10] needed for the asynchronous I/O in the `communicate()` parts of the event loop.

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

Section 2.11.2 gave a mereo-topological representation about how the event loop pattern orders the *behaviour* it must realise for all the activities that it serves. This Section introduces another **mereo-topological** model for event loops, this time explaining the entities and relations —both, *computational resource*-centric and *algorithm*-centric— that together form the mechanism each event loop is built from:

- `callback`: a composition of a data structure and a function. It is the responsibility of the `context` to guarantee that the data is in a consistent state whenever the function is called.
- `source`: the composition of a `callback` with the meta data required for *mediation* within (a context inside) an event loop, to allow "optimal service" to be given to all

[9]For example, many web servers and web browsers work with an event loop architecture.
[10]For example, for UDP message passing, or TCP/IP byte streams.

callbacks. Some mediation relations are: priorities between the execution order of callbacks, or the information needed to decide whether some callbacks can/should not be executed any more.

- `event`: this data structure is filled in by the operating system (and hardware) and read by the event loop to find out whether or not, and what, data is available from asynchronous sources. Also the synchronously executing algorithms can fire events, but their handling can be realised completely without support from the operating system.
- `context`: the composition of `source`s that adds two relations to the meta model: (i) the sources over which to do the mediation are clearly identified in relation that the event loop execution must take into account, and (ii) all the data and constraints needed to execute all the functions in the sources must be accessible in a local data structure owned by the event loop, so that the callback handling can take place in a thread-safe way.
- `poll`: the function that a `context` is executing to read the information about what data an asynchronous I/O source has available for reading, or has sent out to "somewhere else".
- `prepare`, `check`, `dispatch`, `finalize`: these four functions are (possibly) executed in each iteration through an event loop, in that order,[11] for each individual I/O `source`:
  - `prepare`: do `source`-specific configurations to make it ready for `poll`ing, if needed.
  - `check`: read some "register" data in a `source`'s local data structure, to find out whether it is ready to be `poll`ed in this ongoing run through the event loop.
  - `dispatch`: actually execute the `poll` and corresponding `callback` function.
  - `finalize`: do `source`-specific configurations to "clean up" a `poll`ed `source`, if needed.

  Not all functions need to be defined for every `source`, and when they exist they need not necessarily be called every loop iteration. These decisions are part of the *policies* of the event loop *mechanism*.
- `loop`: this is a data structure representing (i) the *order* in which the above-mentioned functions are executed, on their respective `source`s, and (ii) the maximum *time* that it wants to be blocked on a `poll` before it pre-empts that the `poll` and continues with the rest of the functions in the loop.
- `message-queue`: when some `poll`ing has been off-loaded to a dedicated worker thread, the `loo` need not read the original `source`'s registers (that part of the job is being done by the worker), but it "polls" the message queue that is filled by the worker with the data it polled. In contract to asynchronous I/O, the message queue is a local data structure, which is in principle always ready to read from (possibly with empty data as a result).
- `attach-source`: connect a particular `source` to one single `context`.
- `attach-context`: connect a particular `context` to one single `thread`.

[11]Functions from several sources can be taken together, respecting the same overall order.

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

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

2.11.5 Policy: priorities

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

2.11.6 Policy: mediation

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

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

2.11.7 Policy: context deployment in threads

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

2.11.8 Policy: integration into software components

Figure 2.9 shows an *architecture* to implement the **internals** of a software component, as one single *process* with three types of *threads*:

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

Since all threads reside in the same process, their mutual communication can be realised by *message queues* on the memory shared by all threads. The "event polling" for such message queues is simple and efficient, because it can be done synchronously. The Figure shows the

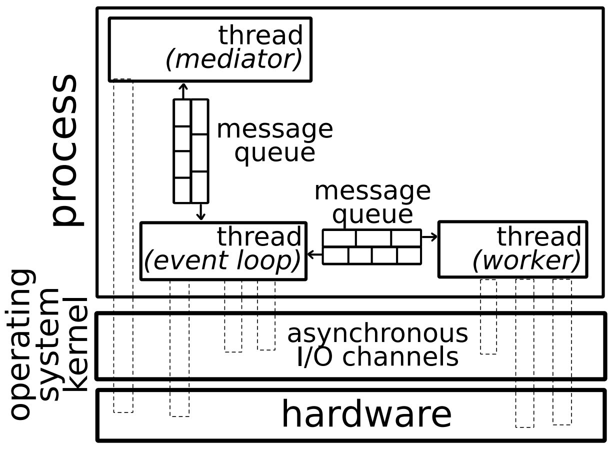


Figure 2.9: The mereological and topological meta model of a *software component*, built as one single process with several threads and message queues. The dashed rectangles represent asynchronous I/O channels, offered by the operating system and possibly requiring also hardware components such as network cards, or Analog/Digital convertors.

generic case with two message queues between each two threads, to support interaction in both directions.

There is *maximum one* mediator thread, *minimum one* event loop thread[12], and *zero or more* worker threads. Within the process, the threads interact via message queues; they (must) use asynchronous I/O, provided by the operating system only (e.g., Unix pipes), or the OS together with the hardware (e.g., EtherCat sockets).

Figure 2.9 does not show the **external** interfaces of the software component. This typically correspond to one or more **Ports** being connected to a subset of the asynchronous I/O and message queues of the component's internals.

2.11.9 Software pattern realisations

The event loop can rightfully be called a "software pattern", since there exist already various realisations, with mature and large-scale application track records; here are some of the largest realisations, with industry-friendly open source licenses:

- libuv event loop, powering the Node.js real-time web applications platform, and with the V8 Javascript engine.
- Java, JVM, Disruptor event loop
- GLib Main Event Loop:

  The FSM states of the loop.

  The event loop loops over callback `sources`, each providing `prepare`, `check`, `dispatch` and `finalize` functions; there three default tyeps: timers and file descriptors (the OS does the waiting and the loop polls for ready events) and idle ones, that are always ready to be dispatched (= run):

  The loop can also deal with also "child thread" signals: CTRL-C etc.

[12]Hence, none of them is really the "main" event loop.

- SystemD event loop

  States: states

- ZeroMQ based event loop. It has a fixed attachment between sources, context, and thread.

  ZeroMQ offers `inproc` messages queue, which fit in the event loop context to support inter-thread communication, between the main `loop` and its `workers`.

2.11.10 Software anti-pattern realisations

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

2.12 Solver pattern: from declarative specification to imperative algorithm

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

Typically, the *implementation* of software algorithms is distributed as a library with **compile-time** type checking, and its documentation is often reduced to an *Application Programming Interface* (API) description. This approach has served the robotics community for some decades, but the growing complexity of the robot systems and the higher demands with respect to application flexibility have made clear that the approach does not support developers (i) **to compose, at development time** various algorithms from the various levels of the motion, perception and world modelling stacks, (ii) **to configure runtime interactions** with other functionalities, or (iii) **to deploy** the same algorithm with its optimal configuration settings for the large variety of software component frameworks and operating system process capabilities. All of these advanced functionalities require some form of formal models, and tooling to support the reasoning behind the required **model-to-"X" transformations**.

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

2.12.1 Mechanism: relation, constraint, dependency graph, spanning tree, action, solver sweep

A typical solver has the following **mereo-topological** entities :

- `domain`: the set of all "states" of the problem under study.
- `optimization-relation` and `constraint`: these relations represent which state combinations are, respectively, desired or not allowed.
- `dependency-graph`: the graph that represents dependencies between several `optimization-relations` and `constraints`. For example, there can be an *order* in inequality constraints, or an hierarchy in optimization functions.
- `spanning-tree`: one particular way of ordering the dependencies in a tree structure. Most solvers spend time on creating such a tree structure, to have an efficient way of structuring their computations.
- `action`: any combination of allowed data and function on the `domain`. The goal of the solver is to find a control flow on such `actions` that brings the system from an initial state to a state that satisfies the `optimization-relations` and `constraints`.
- `solver-sweep`: a solver algorithm typically "sweeps" one or more times over the `spanning-tree`, where each node crossing correspond to the scheduling of a particular `action`, and a termination condition checks whether the constructed set of actions (that is, the resulting imperative algorithm) is "good enough" to stop the solver.

2.12.2 Policy: task-based hybrid constrained optimization

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

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

For each particular **domain**, one must fill in the *types* for f, X, q, etc., as well as a particular *type* of solver. For each particular **application** in that domain, one has to fill in:

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

2.12.3 Policy: dynamic programming

Dynamic programming is, often, the most difficult algorithm design pattern, since it *exploits* the knowledge about which intermediately computed data structures should be given the status of "state", because they will be reused at a later time. Obviously, that knowledge is very domain and application dependent, so few generic insights exist. With one major exception: the physical world satisfies many *conservation principles* (e.g., for energy or momentum), so any intermediate result that represents a conserved property is a natural candidate to become a status variable in the algorithm.

The design *forces* are: to maximize runtime adaptability, in dependencies between data, functions and schedules.

This is a broad family of "declaratively specified" algorithms (or *solvers*, Sec. 2.12), and hence they come close to exploiting all features of the presented algorithmic meta model.

2.12.4 Policy: static spanning tree pyramid

(TODO: all memory statically allocated, functional dependencies modelled as spanning trees, schedules modelled as spanning tree over spanning trees; configuring, preparing, dispatching;)

2.12.5 Policy: feasible and optimal solutions

Any dependency relation can be "hard" or "soft", specifying whether one desires to find feasible or optimal solutions:

- **feasible**: each constraint relation must be strictly satisfied.
- **optimal**: the *deviation* from the constraint relation is optimized according to a *cost function*.

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

2.12.6 Policy: sweep scheduling

The schedule of the solver's computations can be determined statically or dynamically:

- **static**: the spanning tree is computed once, and deployed as a static dispatch data structure.
- **dynamic**: the spanning tree can be (re)computed at runtime, allowing for dynamic reconfigurations of the solver specification.

2.12.7 Policy: tolerances

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

2.13 Mediator pattern: resource access trade-off between activities

Tasks rely on ("physical" as well as "cyber") resources being "sufficiently" available, to realise a set of ("cyber") capabilities "sufficiently well". In real-world contexts, the *quality of service* can seldom be perfect, because **trade-offs** must be made between cost, availability and quality of resources. Since such trade-offs have been challenges in all engineered systems since the beginning of humankind, several insights have been gained over the years about how to support this in software, and the result is the mediator pattern.

2.13.1 Mechanism: Quality of Service & resource-capability dependency

(TODO: define QoS metrics of resource and of capability, monitor QoS, dependency model between component behaviour and QoS, solver)

2.13.2 Policy: resource throttling

(TODO: how to choose specific trade-offs, and specific QoS)

2.13.3 Software pattern: configuration of mediated tasks by mediator in shared context

The mediator can rightfully be called a "software pattern", since there exist already various realisations, with mature and large-scale application track records; here are some of the largest realisations in the domain of ICT infrastructure:

- bandwidth throttling.
- CPU.
- process throttling.
- garbage collection.
- memory pool.

- thread pool.

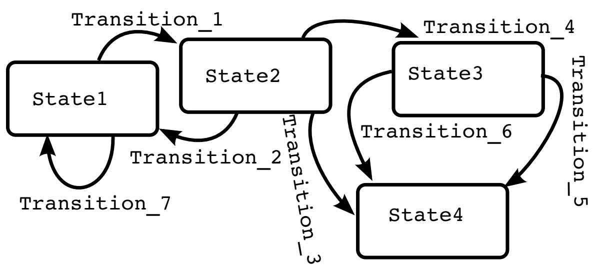
(TODO: *isolation* of *runtime* (that is, all the runtime's data are copied per "isolation", such as the heap, calling stack and garbage collection) with a different *context* for each *application activity* (that is, the application's "shared" and "global" data are copied for each of a number of concurrent activities); e.g., `V8:Isolate` and `V8:Context` in the Chrome V8 runtime engine. Note that the isolation and context pattern is in itself *not sufficient* for *thread-safe* execution!)

2.14 Finite state machine for Coordination: mediating the discrete behaviour of activities

This Section presents the meta model of the Finite State Machine (FSM), used to represent that activities (be it processes, systems, controllers, tasks,...) must often **realise different behaviours**, one at a time, and with particular constraints on when and why to switch between these behaviours. The description of the meta model maximizes the *separation* of *mechanism and policy*, and of *structure and behaviour*. This is not an obvious ambition, because in the long history of state machines various versions of their structural and behavioural models have been "standardized" with monolithic, domain-centric couplings between all these aspect. For example, Harel statecharts, Mealy machines or Moore machines, [10, 61]. The FSM meta model presented in this Section provides a modularization with which all of these variants can be *composed*, with context-specific configurations.

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

The **structural** part of the FSM meta model, Fig. 2.10, represents each individual behaviour by a **state**, with **transitions** between states to represent switches in the behaviour. There are no constraints on how many transitions can exist between two states, and a state is allowed to have a transition to itself. The `state` and `transition` models are extremely simple; the `transition` model depends on the `state` model, but not vice versa.



start state	transition	end state
State1	Transition_1	State2
State2	Transition_2	State1
State2	Transition_3	State4
State2	Transition_4	State3
State3	Transition_5	State4
State3	Transition_6	State4
State1	Transition_7	State1

Figure 2.10: An example of the **structural** (mereo-topological) part of a *Finite State Machine* model, with its graphical representation on the left, and its tabular form on the right. In a property graph representation of this model, the nodes are `States` and the relations are `Transitions`; the FSM in itself is a higher-order relation: it composes a particular set of `States` and `Transitions`.

The **behavioural** part represents the logical conditions under which the FSM meta model transforms **events** (that an FSM receives from "the outside", or that it generates itself) into (i) state `transitions`, and (ii) `events` to send (to "the outside", or to itself). The behavioural model is the so-called **event reaction table**[13]; Fig. 2.11 has an example. An `event` has meaning outside of the context of finite state machines too, so the FSM meta model `conforms-to` the `event` meta model. In the latter meta model, the mathematical semantics of an `event` is that of a Boolean variable: the event "*has happened*", or it has not. Hence, also the `event reaction table` model is simply a table of Boolean expressions, whose truth value is computed via propositional logic.

Boolean expression of events reacted to	resulting transition	events fired
`e_1` ∨ `e_3`	`Transition_1`	
`e_2`	`Transition_2`	`E_2`
`e_3` ∧ `e_1`	`Transition_3`	`E_1`, `E_3`
`e_4`	`Transition_4`	`E_4`
`e_1` ∧ `e_3`	`Transition_5`	

Figure 2.11: An example of an *event reaction table* that models one possible *behaviour* of the FSM in Fig. 2.10. When the Boolean condition evaluates to `true`, the corresponding `transition` is executed in the model, *and* a possibly empty set of `events` is fired. The events are not part of the structural model in Fig. 2.10, because they are the coupling with the *system* context in which the FSM is embedded.

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

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

- `event_queue`: the *structural relation* that represents the streams of `events` that the FSM has received and must still process, or that the FSM has fired and must still send out.
- `event_processing`: the *behaviourial relation* between the input `event_queue` stream and (i) the output `event_queue` stream, and (ii) the `transitions` stream. Boolean algebra is the mathematics of the algorithm that *computes* the behavioural relation.
- `event_monitoring`: a particular instance of `event_processing` that monitors the FSM as an activity in its own right. Typical monitor functions are:
  - detecting whether the FSM is "running in cycles". For example, the same nominal Task execution plan is always interrupted by the same non-nominal situation and

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

the same error recovery plan that leads to the same Task FSM state that the Task plan execution started from.

  - event tracing: an event must be fired each time a particular sequence ("trace") of events has occured. For example, `activity_1` sends an event after `activity_2` and `activity_2` have fired an event.

This monitoring is one of the *causes* that fires, from the inside, events to which the FSM can react to.

The computations needed for event queueing, processing and monitoring are realised by an `event_loop` that "wakes up" at configured moments in time to execute the computations. This execution requires **computational state**[14] of its own, which adds extra parameters to the FSM meta model:

- *life cycle* parameters:
  - `current_state`: the ID of the current state of the FSM.
  - `initial_state`: the FSM `state` in which the FSM starts after its event processing activity has transitioned from its `deploying` state to the `active` state (Sec. 2.14.7).
  - `final_state`(s): one or more FSM `states` in which the FSM ends its `active` state and transitions back to its `deploying` state.
  - `state_history`: the stream of state IDs that the FSM execution has moved through.
- extensions to the externally visible `event reaction table`:
  - `onEntry`, `onExit`: these events are added to the table for each `transition`, so execution behaviour can be triggered every time the `state` at the start of the `transition` is *exited* and the `state` at the end of the `transition` is *entered*. It is a best practice to use two events, since this allows the behaviour to be "owned" by the separate `states`, and not by the `transition`.
  - `onTrigger`: the `event_loop` triggers the `current_state` at regular intervals in time, and execution behaviour can be connected to such triggers by adding a `transition` from that `state` to itself.

2.14.3 Mechanism: activity Coordination behaviour via monitors and callbacks

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

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

[14]Note the fundamental differences between "state" as a representation of type of behaviour, and "state" as the values of the parameters in the algorithms that implement the behaviour. It is a pity that no widely accepted nomenclature exists to separate these two meanings of the word "state".

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

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

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

2.14.4 Policy: selection, priority, deletion

At any moment in time, the `event_queue` of an FSM contains zero or more events that it is expected to react to. And the `event_processing` relations in an event reaction table are *declarative*. Hence, extra choices must be made to determine the actual execution behaviour of the event handling. For example:

- which internal events and transitions to include in the model.
- which callback functions to attach to which events, and to which activity. This is the trade-off between
  1. *communicating the event and keeping the computation local.* The `event` is not handled in the event loop of the Coordinating state machine, but communicated to the coordinated activity; the latter then computes the callback in its own computational context.
  2. *communicating the computation to a non-local context.* The `event` is handled in the event loop of the Coordinating state machine, so the computation of the callback takes place in the event loop of the coordination activity.
- how many events to take from the event queue in one single event condition evaluation step.
- priorities on the selection of events from the queue.
- the rules to decide when to remove which events from `event_queues`.

2.14.5 Constraints on event handling meta model

The **constraints** below are (possible) assumptions to make in an FSM meta model. It is necessary to make these assumptions formally explicit, when one needs to subject an FSM model to formal verification (*"does the system implementation conform to its specifications?"*), and validation (*"does the system specifications conform to the application's requirements?"*):

- a modelled behaviour is in one, and only one, state at each moment in time.
- an FSM represents the behavioural state of only one activity.
- every state and every transition has a unique ID.
- state transitions take no time.

- event processing takes no time.
- event firing takes no time.
- representation of all parts of the model takes no space in computer memory.
- event queue bookkeeping takes no time.
- the management of the event processing takes no time.

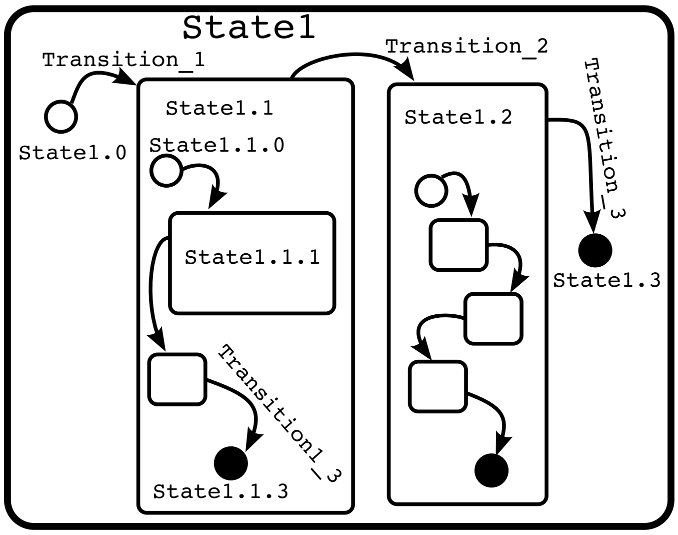


Figure 2.12: An example of a hierarchical *Finite State Machine*. The open and filled circles represent `initial_state` and `final_state`, respectively.

2.14.6 Policy: hierarchical state machine

The structural relation of **hierarchy**, as depicted in Fig. 2.12, is commonly used for finite state machines. It changes the structural and behavioural semantics of the "flat" state machine as follows:

- **containment relation**: a `state` can be `contained` in another "super" `state`.
- **containment tree constraints**: a `state` can only be `contained` in one single "super" `state`, and these containment constraints can only form a **tree**.
- **shared transitions**: a `transition` from a super `state` to another `state` represents the set of `transitions` from **all** of the internal `states` to the same other `state`.
- **non-shared event conditions**: each internal `state` **can** have a different event condition for the above-mentioned shared `transition`.
- `initial_state`, `final_state`: when the overall state machine `transitions` *into* the super `state`, **one** internal `state` **must** be selected as the `final_state` of that `transition`. One internal `state` **can** be selected as `final_state`, which means that an internal `transition` into this `final_state` **automatically** gives rise to a `transition` away from the super `state`, *if* that super `state` has only one possible `transition`.

- there **can** be a policy **to remember** the internal `state` that the FSM is in when a `transition` takes place out of the super `state`, so that this so-called **history** `state` will be selected as the `initialstate` for the next `transition` into the super `state`.

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

- **reduction of state explosion**: one can "hide" all states below a certain level in the containment tree, hence reducing the (apparent) complexity of the number of states and transitions.

- **just-in-time creation, or lazy evaluation, of state machines from model**. The reason can be to reduce the effects of state explosion on the **runtime memory consumption**, but the policy also allows for **runtime adaptation** of the discrete behaviour of a system by the *just in time* creation of new state machines via **runtime reasoning** on the basis of (declarative) behavioural models in the application.

2.14.7 Best Practice: Life Cycle State Machine for single activity deployment

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

The design pattern of the *Life Cycle State Machine* (LCSM) in Fig. 2.13 has been used since decades, in several equivalent incarnations and names,[15] to coordinate the availability of "internal" resources for the provision of "external" capabilities, and to let the capabilities be provided only *after* they have been fully configured. The semantics of the different states is as follows:

- **deploying**: the system is working on finding and configuring all the resources it needs before it can offer its capabilities to others. During this super state, the system should not be visible to other systems, since the latter can not yet (or not anymore) interact usefully with this system.

  - **creating**: the software resources are created, with which the system does the bookkeeping of its own behaviour.
  - **deleting**: the above-mentioned software resources are cleanly removed.
  - **configuring resources**: the system is configuring the service resources it requires for its own operation.

- **active**: the system is visible to other systems to engage in interactions.

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

- **configuring capabilities**: the system is not providing its services, since it must first (re)configure itself to provide a particular configuration of its services.
- **ready**: the system is ready to provide its services to other systems.
  - **pausing**: the provision of the system's services is put on hold, but can resume immediately. This state is necessary, for example, for the composition with Traffic Light activity coordination.
  - **running**: the system's services are provided.

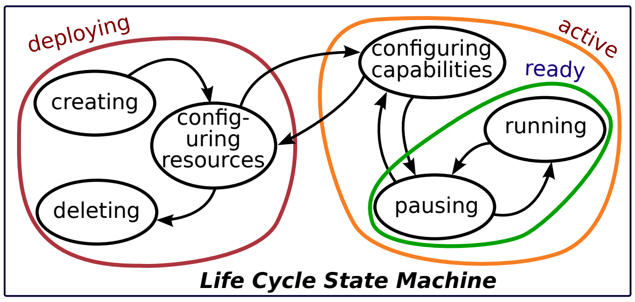


Figure 2.13: The Life Cycle State Machine meta model coordinates the configuration of the resources required for a certain activity before that activity's capabilities can be offered as a service to third parties.

An example is the service to control the motion of a robot: in its deployment superstate, it must not only create and configure the data structures and functions that it needs for its motion control service, but it must wait till other services have become active (e.g., a kinematics computation service, and the input/output device drivers to the robot's actuators and encoders). It can provide several types of motion control services, like force, impedance, velocity or position control; sometimes it will have to pause its own service, because the end user Task system is itself not yet active.

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

2.14.8 Best Practices: Flag, Traffic Light, Petri Net, for inter-activity coordination

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

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

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

- ***red***: the activity *can not* access the shared resource.

- *green*: the activity *can* access the shared resource.
- *orange* (optionally): the shared resource is “soon” going to become (un)available for access.

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

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

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

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

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

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

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

- *red*: the coordinated activity *can not start* its behaviour.
- *green*: the coordinated activity *has started* its behaviour.
- *orange* (optionally): the coordinated activity is *soon ready to start* its behaviour.

The Life Cycle State Machine of Fig. 2.13 uses the same color code, for its super-states:

- *red*: the state of the LCSM is `deploying`; the activity *can not* provide its capabilities/services to others.

- *orange*: the state of the LCSM is `active` and *can switch "soon"* to the `ready` state (that is, either `running` or `pausing`).
- *green*: the activity is `running` or `pausing`, and *can switch* between both states *instantaneously*.

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

2.14.9 Bad and Good Practices in FSM usage

- *bad*: to mix the Computations for (i) the *Boolean logic* computations to decide about making transitions, and (ii) the *continuous* computations needed *to monitor* activities and to generate their events. The former are much easier to make deterministic in time than the latter.

  good: to put such a monitoring computation into another activity than the FSM, and interconnect them with an event communication; and design in extra states and logic rules in the FSM such that the behaviour is robust against the undeterministic computation and communication timing of the monitoring events.

- *bad*: to add the "guard" (monitoring) computation to a transition, since that makes the transition take an non-deterministic time to be realised.

  good: to add no computations whatsoever to transitions, but only to states. In this way, the only time taken by a transition is the time needed to adapt the FSM data structure by changing the pointer to the `current_state`.

2.15 Data, uncertainty and information

Using formalized models to represent one's knowledge about the state of the world, and about the relations that exist between the time evolutions of interacting entities in the world, can help in formulating the "right" task control problem. However, defining the model is only half of the story: since every model contains **parameters**, one has to estimate the "real" value of these parameters, based on (i) the measurement data from sensors, and (ii) the relations that link these measurement data to the model parameters.

2.15.1 Mechanism: Bayesian probability axioms

Bayesian probability theory is a scientific paradigm for **information processing**. Its **axiomatic** (hence, fully **declarative**) foundations have been laid in the 1960s [25, 27]. These *axioms for plausible Bayesian inference* are:

I Degrees of plausibility are represented by real numbers.

II Qualitative correspondence with common sense.

III If a conclusion can be reasoned out in more than one way, then every possible way must lead to the same result.

IV Always take into account all of the evidence one has.

V Always represent equivalent states of knowledge by equivalent plausibility assignments.

They result in the well-known mathematics of statistics, with **random variables** and **probability density functions** (PDF) as major entities, and the **chain rule** and **Bayes' rule** as major relations. How to measure the information contents in a PDF was also explained axiomatically [25], resulting in the primary role of **logarithms** as the natural **measures of information**. These axiomatic foundations are as follows:

I $I(M{:}E \text{ AND } F|C) = f\{I(M{:}E|C), I(M{:}F|E \text{ AND } C)\}$

II $I(M{:}E \text{ AND } M|C) = I(M|C)$

III $I(M{:}E|C)$ is a strictly increasing function of its arguments

IV $I(M_1 \text{ AND } M_2{:}M_1|C) = I(M_1{:}M_1|C)$ if M_1 and M_2 are mutually irrelevant pieces of information.

V $I(M_1 \text{ AND } M_2|M_1 \text{ AND } C) = I(M_2|C)$

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

(TODO: [65].)

2.15.3 Policy: belief propagation

(TODO: explain how the structure of a Bayesian graphical model provides a declarative way to solve the model. Junction tree, message passing. [29, 38])

2.15.4 Policy: hypothesis tree for semi-optimal information processing

(TODO: [12, 44].)

2.15.5 Policy: mutual entropy to measure change in information

One of the major **choices** within the large family of logarithmic functions was the following:

$$H(p,q) = -\int p(x) \ln\left(\frac{p(x)}{q(x)}\right) dx,$$

where both $p(x)$ and $q(x)$ must be *strictly positive*. The function $H(p,q)$ is **asymmetric**, hence it is not a distance function, or metric. The reason why it is a "major" choice is that it focuses on the **relative** change in information between two probability density functions, instead of aiming for an absolute measure, which does not make much sense. $H(p,q)$ is known under several names: *mutual entropy*, *mutual information*, or Kullback-Leiber divergence.

Chapter 3

Meta models for rigid body geometry and polygonal world models

Three meta models take center stage in the mathematical representation of geometric entities and relations, in two- and three-dimensional abstractions of the real world: (i) **points** in **Euclidean space**s, (ii) the *composition* of points into **line segments**, and of line segments into **polygons**, and (iii) the *position* and *orientation* of a **rigid body**. The meta models also include representations for (i) **motion** relations (that is, the **changes over time** for all of the above entities and relations, but also their composition in **geometric chains**), (ii) **forces** applied to rigid bodies, and (iii) the **impedance** relations between motion and force, namely **inertia**, **damping** and **elasticity**. Together, the can model how to drive, to store and/or to dissipate the **energy** corresponding to rigid body motions.

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

3.1 The meta meta models

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

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

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

The **geometric chain** is the most complex relation in this Chapter. The **composition** of rigid bodies with **joints** as the motion constraints between them, are the topic of the Chapter on **kinematic chains**. The complementary **composition** of rigid bodies with **electro-mechanical dynamics** as complementary motion constraints are is already treated in this Chapter's Sec. 3.4.

3.1.1 `semantic_ID` for geometry meta model

Section 1.2.4 describes the generic foundations of semantic meta data, namely the usage of IDs as "symbolic pointers" between models and meta models. This Section adds geometry-specific specialisations, and explaining that model in human-centered form is the subject of this Chapter. The suggested `MMID` is

$$\texttt{Geometry::2D-3D::Point-Segment-Polygon-Body-Graph}. \tag{3.1}$$

3.1.2 Euclidean space for points

The **Euclidean spaces** in two and three dimensions, with (shorthand) `MMID`s "$E(2)$" and "$E(3)$", represent position and translational motion of points. Hence, they are meta models for motion, and for the mechanical dynamics of point masses under Newton's laws of motion. The Euclidean space has **no natural origin**, because any point in the space serves equally well as reference. This implies that "position" can never be a *property* of the representation of a geometric *entity*, but only a property of a *relative position relation* between geometric primivites. Relations between two `Point`s are:

- *topological*: `are-equal`. This relation holds for all mathematical spaces refered to in this Section.
- *metric*: Euclidean spaces are endowed with a natural metric, that **quantifies** the `distance` relation between two `Point`s. The metric is referred to as "Euclidean" metric.

One of its best-known relation is the triangle inequality. The relations `are-orthogonal` (of lines) and `speed` (of translational velocities, accelerations and forces) are consequences of the `distance` relation.

All of these entities and relations belong to the realm of the *mathematics* meta meta model. The rest of this Chapter provides more semantic models to let a computer-controlled system start computing/reasoning with these entities and relations.

3.1.3 Non-Euclidean space for lines

The composition of points into lines brings an extra **constraint** relation between the points, namely collinearity. Two major types of geometry meta models represent the semantics of relations between lines:

- affine geometry: the main constraint relations are parallelism, barycentre, and convexity.

  In a robotics context, this means that an affine description of points and lines keeps the relative order between them; for example, an affine geometric model that has a street between two other streets, keeps that street in the middle, no matter what affine transformation is applied to the model.
- projective geometry assumes less constraint relations than affine geometry, just incidence and cross-ratio.

Topological relations: `are-equal`, `are-parallel` (affine only), `intersect`, `have-ratio`. There is one semantic relation that is *not* part of the meta models, namely *distance*.

3.1.4 Non-Euclidean space for rigid bodies

When points and lines are being connected together to form "rigid bodies" in the Euclidean spaces E(2) and E(3), there emerges a new relation of orientation. All of the above-mentioned mathematical entities and relations remain meaningful; nevertheless, some extra semantics must be introduces, because of the dependency between *translation* and *orientation*. The following two spaces add the relevant semantic entities and relations:

- the **Special Orthogonal group and algebra** in two and three dimensions ("$SO(2)$", "$so(2)$" and "$SO(3)$", "$so(3)$") represent the semantics of "pure" orientations. This space has a well-defined "distance" metric between any two orientations (the smallest angle over which to reorient the first orientation to make it `equal-to` the second orientation. But there is no "unambiguous zero" rotation (because rotating a body over 360 degrees brings it back to its original orientation).

- the **Special Euclidean group and algebra** in two and three dimensions ("$SE(2)$", "$se(2)$" and "$SE(3)$", "$se(3)$") represent the semantics of the **coupling** between **translations and orientations**. One of the major constraints on this space is the **lack of a bi-invariant metric** relation [28, 33, 34]; for rigid bodies this means that:

  - one must always introduce a weight function (that is, a new semantic relation) between translation and orientation.
  - the "normal" Euclidean metric has no physical sense,
  - "orthogonality" between velocities and forces has no physical sense either [33].

  Examples of physically meaningful weight/metric functions in mechanical systems are inertia and elasticity/stiffness. The get that semantic status because their physical meaning leads to a metric that is "invariant" under any change of formal representation of the relations involved. More concretely, the kinetic energy of a moving body is independent of the physical units chosen to compute that energy, and independent of the reference frame in which one computes that energy.

3.1.5 Differential geometry

The concepts of manifold, group, metric are defined in the mathematical **meta meta models** of **differential geometry** and **group theory**. That higher-order knowledge provides useful (and already highly formalized) constraint information. For example. the relation that velocities satisfy all properties of the tangent space to the manifold of rigid body poses, accelerations live in the **second-order tangent space**, and forces live in the co-tangent space, [9, 20]. The latter means that forces can play the role of linear forms over the motion spaces, mapping position displacements, velocities and accelerations into real scalar numbers, with physical interpretation of energy; more in particular, work, power and "acceleration energy"[1], respectively, [43, 45, 64].

[1]This is not a commonly used term in the English-language scientific literature. Gauss [21] introduced the concept with the German term *"Zwang"*.

3.1.6 Polygon spaces for maps

Meta models for the more "pragmatic" (read: "computable") geometric entities of **line segments** and **(multi) polygons** have matured in Geographical Information Systems (GIS) contexts for map making. One concrete example is the GeoJSON standard.

3.2 Taxonomy of geometric meta models

The partially ordered hierarchical structure of Fig. 9.1 is introduced as a modelling axiom (a so-called taxonomy) for the (mostly geometric parts of) **motion/perception/world modelling stacks** in robotics. This partially ordered modelling structure serves as **structural backbone** of reasoning, querying and model transformations. The lower levels of abstraction (with the lowest numbers in the descriptions below) represent the (almost) domain-independent aspects of mereology (the *set* of "things" in the model) and topology (the *connections* between "things"); the gradually more "domain"-specific levels are motivated by specific sets of use cases in robotic world modelling, motion and perception. As far as the *geometry*[2] parts of the taxonomy are concerned, this document introduces the following partially ordered set of levels of abstraction, using the *kinematic chain* as the running example to ilustrate the kind of knowledge representation required at each level:

1. **Mereology**: this abstraction meta meta level is introduced in Sec. 1.3, and models the "whole" and its "parts", with just `has-a` as relevant relation and `collection` as relevant entity.

   In the mereology of the geometrical meta model, a kinematic chain `has-a collection` of `links` and `joints`, and it `has-a workspace`, that is, a part of the physical world that it can occupy.

1.1 **Objects** & 1.2 **Manifolds**: the "wholes" and "parts" in a robotics context are further classified in (the top level of) a taxonomy with two complementary branches: *"discrete"* things ("objects") and *"continuous"* things (the "manifolds" of the configuration spaces of the objects). One particular mereological entity or relation has typically discrete as well as continuous parts; for example, a kinematic chain has a continuous motion space, but also a discrete set of actuators and sensors; which by themselves again have continuous state spaces.

   A kinematic chain `has-a` its `links` and `joints` as `objects`, and its `workspace` as `manifold`.

2. `Contains-Connects` **Topology**: this abstraction level introduced in Sec. 1.4 models "neighbourhood", more in particular the `contains` and `connects` relations, in, both, discrete and continuous spaces. There are also topological relations in *taxonomies*, for which hierarchical `hypernym/hyponym` relations have been identified in linguistics (not only for objects, but also for verbs).

   A kinematic chain `connects` its `links` to its `joints`, forming the kinematic graph; most common topological instantiations are: *serial*, *tree* or *parallel*.

[2]The taxonomy includes **dynamics** too, which is covered mathematically by the theory of **differential geometry**.

2.1 **Spatial topology**: in the *continuous* real-world space around us, the following relations specialise the `connects-contains` relations:

- **neighbourhood** relations like `near-to`, `left-of`, `on-top-of`, `inside-of`, etc.
- **tesselation** (or "**tiling**") representations of spatial **coverage**.

Both are defined in the (two- as well as three-dimensional) Euclidean, affine, and projective spaces. Such spatial topological relations apply to kinematic chains relative to objects in the environment, for example, when its end effector comes `above` a table, or `inside-of` a box, or when a mobile platform covers contiguous areas in the world.

2.2 **Object topology**: within the *continuous* real-world space around us, objects can be physically connected to each other, and their `connects` relations have `block`, `port`, `connector`, and `dock` parts.

`Serial` kinematic chains have "arm" and "hand" object topologies, which are more concrete (i.e., behaviour-rich) than a "manipulator", which is more concrete than "actor".

3. **Geometry**: this is the essential next level of modelling abstraction, for the *manifold* type of entities and relations, and a large taxonomy of geometrical meta models has been defined in the mathematics literature already, of which the affine, projective and metric versions are most relevant to robotics.

A kinematic chain `has-a` lot of `points` attached to its `links`, and whose geometrical `position` and `velocity` in space are of interest in the task for which the kinematic chain is used.

3.1 **Affine geometry**: this introduces non-metric geometrical entities, such as `point`, `line`, and `hyperplane`, and relations such as `intersect`, `parallel`, and `ratio`.

Many serial kinematic chains have some `parallel` joint angle axes.

3.2 **Dimensionless metric geometry**: many use cases do not need *absolute* distances or angles, but rather *relative* ones. In other words, the absolute **scale** of the geometrical entities is not used, but the *ratios* of lengths or angles (which are physically dimensionless) are the relevant information in a task specification. For example, when driving through an office corridor, the robot perception system can track the relative (changes in) areas and the directions of wall, door, ceiling or floor surfaces, without having to know their absolute sizes. Indeed, staying at the "center" of a corridor, driving towards its "end", or moving twice as far as the nearest door, are all relative (or "qualitative") motion specifications.

A serial kinematic chain often has the "zero configuration" of its joint angles in the `middle` of their physical motion range, which is geometrically well-defined without the need to use absolute numbers. Similarly, a specification to move a joint "away from" its mechanical limits is a meaningful, metrically dimensionless, motion specification.

3.3 **Metric geometry**: as soon as one introduces a **metric** (or "distance function"), one can start talking about entities such as `rigid-body`, `shape`, `orientation`, `pose`, `angle`, and relations like `distance`, `orthogonal`, `displacement`,...

A kinematic chain transforms metric speeds at its `actuators` to metric spatial velocities of its `links`.

4. **Dynamics**: this modelling level brings in the interactions between "effort" and "flow" in the exchange and transformation of "energy". For mechanical systems, that means force and motion; for electrical systems, that means current and voltage; etc. In general, these effort-flow relations are called **impedances**.

   A kinematic chain transforms mechanical energy between its `joints` and its `links`.

4.0. **Differential geometry**: this is the domain-independent representation of physical systems, that is, all features that are shared between the mechanical domain, hydraulic domain, electrical domain, thermal domain, etc. The most fundamental concepts are: `tangent-space`, `linear-form`, `vector field`, and `metric`.

   A kinematic chain transforms electrical energy at its `actuators` to mechanical energy at its `links`. Each of the latter's motion properties have the mathematical properties of the *Special Euclidean group* SE(3) and its Lie algebra se(3).

4.1. **Mechanics**: there are just three fundamental types of mechanical interaction: `stiffness`, `damping` and `inertia` linking `force` to, respectively, `position`, `velocity` and `acceleration`. Other relevant entities and relations are: `mass`, `elasticity`, `gravity`, `momentum`, `potential-energy`, and `kinetic-energy`.

   All of the above mechanical entities and relations are present in kinematic chains.

4.2. **Electro-magnetics**: the interactions between `current` and `voltage`, namely `resistor`, `inductance`, `capacitance`, `reluctance`, `back-emf`, `flux`,...

   These entities and relations are present in a kinematic chain with electrical actuators.

Later Sections and Chapters will provide more detailed descriptions of many of the mathematical/physical concepts above, and (software) engineering extensions will be added, for example, to model data structures, coordinates and physical dimensions.

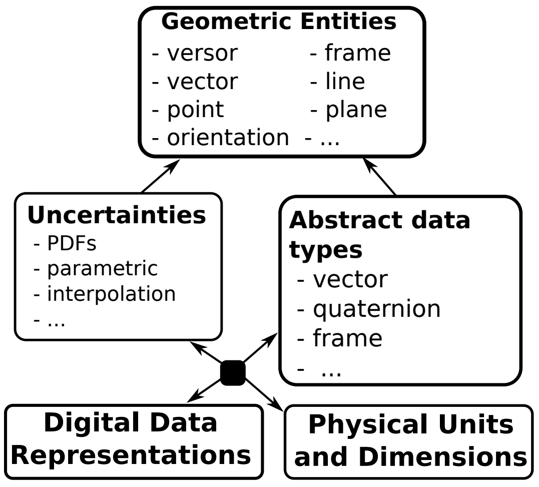


Figure 3.1: The mereo-topological model of the representations of geometric entities and relations, that is, the container entity for "mathematical" (i.e., *geometrical*), "abstract data type" and "digital" representations. An arrow represents *composition* of complementary aspects in that numerical representation, and the "black square" arrow is the generic case of an arrow representing an *n-ary* composition.

3.3 Point-centric meta model for geometry in robotics

This Section presents the meta model for **points**, their composition into **line segments** and higher-order composition into **rigid bodies**. The modelling uses the `Point` as its basic *entity*, and the various compositions of `Points` as the its *relations*. This particular approach is motivated by:

- the desire to keep the number of "basic" *abstract data types* small.
- the fact that most sensors in robotics *measure* points, and not lines, planes or bodies.
- the fact that for points the concept of *uncertainty* is uniquely defined, and not for lines, planes or bodies.
- the desire to provide all possible *"attachments"* needed in the meta model of *kinematic chains* of Chap. 4.

The knowledge in the combination of the meta model for geometry and that for kinematic chains is sufficiently mature and complete to allow their formal representation to start a community-driven process towards a vendor-neutral **open standard**.[3]

Meta models of geometric entities and their relations add semantic meaning ("information") to the digital quantities ("data") that they (or rather, their software instantiations) work with [6, 15]. These semantic relations have the following **dependency structure**, Fig. 3.1:

- each **geometric entity** can be represented by (*"composed with"*) multiple **mathematical** models. For example, a `LineSegment` is the set of all `Points` that lie on the `Line` between a `start Point` and an `end Point`.
- each of which can be represented with multiple **abstract data type** models. For example, a `LineSegment` is an `array` with two members, with one member having a meta data tag of `start` and the other member a meta tag of `end`.
- each of which can be represented with multiple **digital representation** models. For example, each member in the `LineSegment` has a `Coordinate` with three `floats`.
- all of them must be composed with a model of the relevant **physical units**. For example, the above-mentioned **floats** are of the type `lenght` and a `unit` of `meter`.

3.3.1 Geometric entities — the `Point` and its compositions

The `Point` representations (and their *compositions* in *unordered and ordered sets*) are to be interpreted in the *context* of the *Euclidean geometry* in three- or two-dimensional space, but the non-rigid-body versions also hold in affine or projective spaces. The "`tele-type font`" is used to typographically idenitfy the entities and relations that must end up in the meta model.

Point — A point is the *zero dimensional* element of the space **manifold**, so it has no properties of volume, shape, area, length, or weight. None of this knowledge must be represented explicitly; just referring to the mathematical meta meta models suffices. In other words, the mathematical representation of a `Point` is just a **symbol**, because that is what is needed to

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

represent uniquely the **identity** of a `Point` in the space. This document chooses the notation $\boldsymbol{a}, \boldsymbol{b}, \ldots$, to identify `Points`.

The obvious *abstract data type* representation is an ordered triplet (x, y, z) of scalars on the *real line*; that meta model is composed with the *mereological* meta model of the previous paragraph. Another composition is to add values to the triplet; these coordinates only make sense **relatively** to another point, the **origin** of the space, and their numerical values make only sense relatively to an (orthogonal, and oriented) reference **frame** at that origin.

Point pair — An `unordered` set with only **two** `Points`. It is worth introducing such a "trivial" composition as a separate first-class entity in the taxonomy, since many of the other entities in the taxonomy are extensions of the `Point pair` with some extra constraints or other higher-order relations. When one adds the `order` relation to the set of `Points`, one of these `Points` gets the *attribute* of `start` and the other that of `end`.

Grid — A set of two or more `Points` in space. For example, to represent the granularity with which the location of a robot must be known in an application. The `grid` representation can be **composed** with a choice of **ordering relation** in the set: the distances between the points are constrained to certain values; each of the points gets a numerical ID that represents a choice of *coverage* of the space; etc.

Line segment — Or "finite line", is the set of points on a line between the two `Points` of a `Point pair`. `Length` (or `distance` between the two `Points`) is a property that the line segment adds to the `Point pair`.

Polyline — A polyline is a *ordered set* of `Line segments`, with the **constraints** that (i) the `start` point of one line segment is the `end` point of the previous `Line segment` in the ordered set, (ii) each `Point` belongs to exactly two `Line segments`, except for the `start` point of the first `Line segment` and the `end` point of the last `Line segment`. `Length` is a **property** of the `Polyline`, which is the sum of the `Length` of each of the `Line segments`. The **direction** of the `Polyline` is an extra **relation**.

Polygon — This is a polyline with the extra **constraint relation** that the two not yet connected start and end `Points` must now coincide. `Area` is a **property** of the polygon; *orientation* is an **attribute**.

Vector — A vector is a `Line segment` with a `direction`, in the sense that:

- it **connects** the `Point` in the `Line segment` that `has` the semantic tag `start` to the `Point` with the tag `target`.
- it has a **property** of `magnitude`, namely the Euclidean distance between the two points.

Common abstract data type representations are (i) the composition of `start` point representation, its `magnitude` (or `length`) positive real scalar, and its `direction` coordinates, or (ii) an **ordered** combination of two point coordinates.

Different semantic versions of vectors exist, depending on whether the `start` point is *constrained* to be on a given line (**Line vector**), constrained to have its origin at a *fixed* point (**Point vector**), or remains unconstrained, or *free* (**Free vector**). Examples in the mechanical domain of instantiations of these different versions are, respectively, a moment of force acting on a rigid body, the position of a point in space with respect to another point, and a linear force acting on a rigid body.

Versor — A versor (or **direction vector**, **unit vector**, or **normalised vector**) is a **vector** that represents just a direction, without meaning of the spatial position of its `start` point. The meta model needs just one point coordinate set because the *policy* (that is, the **composition constraint**) is that the `start` point of the direction vector *coincides* with the (arbitrarily chosen) `origin` point of the space. In a three-dimensional Euclidean space, a possible coordinate representation is an ordered triple of scalars (x, y, z) with unit Euclidean norm (i.e., $|| \cdot || = 1$). This document represents a versor with the hat symbol, $\widehat{\boldsymbol{n}}$.

Vector field — The selection of one `Vector` at each `Point` in space. For example, to represent the linear velocity of the point on a moving body that instantaneously coincides with that point in space.

Simplex — The 3D spatial primitive representing all `Points` that lie within the boundaries (that is, an extra **constraint relation**) of the `Polygons` formed by the **composition** of **three** `Line segments` on non-coplanar `Vectors`. `Volume` is a **property** of the `Simplex`; `orientation` is an **attribute**.

Polyhedron — This is the generalisation of the `Simplex` to any spatial shape, that is, connecting more than three `Line segments`. `Volume` remains a valid *property*, but `orientation` not.

Orientation (Frame) — An orientation, or orientation frame, represents an orientation as a `Simplex` whose sides are **orthonormal unit** `Vectors` that share the same `start Point`. The most common *policies* are:

- the `Vectors` are given an implicit **order**, with semantic names of X-axis, Y-axis and Z-axis.
- the mentioned order reflects the binary orientation choice between **right handedness** or **left handedness**.

This document uses the symbols $[a], [b], \ldots$ to denote `Orientation frames`. Multiple abstact data type representations exists for their coordinates.

Rigid Body — The term "body" most often denotes a solid, physical object, but in the geometrical context it is a `Polyhedron` with the **constraint relations** that (i) it is non-planar, and (ii) the distance between any of its points remains invariant over time. A `Simplex` or an `Orientation frame` are the simplest representations of a `Rigid body`, when the **constraint** is added that all `Points` on the `Simplex` or `Orientation frame` are rigidly fixed to the `Rigid body`. This semantically more constrained geometric entity is commonly also known under the short name of a **Frame**.

This document suggests the symbol $\mathcal{A}, \mathcal{B}, \ldots$ to denote rigid bodies.

Line — A (straight, one-dimensional) line is, in addition to the `Point`, another geometric entity with axiomatic foundations.

The *abstract data types* for lines come in different flavours, determined by how one *chooses* **to compose** the abstract data types of *points* together:

- an (un)ordered `Point pair` defines an (un)directed `line`.
- a `Point` and a `Versor` (see below) define a (directed) `line`.
- the intersection of two `Planes` (see below) defines an undirected `line`.

This document uses the symbol $\bar{\boldsymbol{l}}$ to represent a *line*.

Plane — A plane is a third axiomatic primitive of geometry, that represents a flat surface described as a two-dimensional subspace of the world.

Like for the plane, many abstract data type representations exist, depending on how many points, versors and/or lines one composes in the representation:

- an (un)ordered composition of a set of three `Points` defines an (un)directed `Plane`.
- a point and two versors define a directed (or, oriented, see below) `Plane`.
- the composition of two *intersecting* `Lines` defines an undirected `plane`.
- a `Simplex` represents a directed `Plane`: two `Line segments` determine the `Plane` and the *right-hand rule* with the third `Line segment` determines the direction/orientation.

This document suggests the symbol $\mathfrak{P}$ to represent a plane.

Half space — This contains all the points in space which lie on one side of the *supporting plane*. In this way, that plane *orients* the space in "left" and "right", where the terminilogy is an *arbitrary convention*. Such orientation can be represented by one single representative point in the "right" half space; or by the above-mentioned ordered couple of direction vectors; or by a `Simplex` in which two `Line segments` determine the `Plane` and the `end Point` of the third lies in the `Half space`.

3.3.2 Geometric relations — `Position` and instantaneous `Motion`

This Section extends the mereological taxonomy of the geometric *entities* with the relations of **motion** of rigid bodies with respect to each other, Fig. 3.2, including linear and angular **distances** between `line-point` and `versor-plane` entities. The `Motion` can be considered in two ways:

- **locally**, **passively** and **instantaneously**: **time derivatives** of `Position` give rise to `Velocity` and `Acceleration`.
- **finite distance**, **actively**, and **time-independent** as a `Displacement`.

In a mathematical representations, there is no difference between `Position` and `Displacement`, so both terms can be used as synonyms. The mathematical (more in particular, differential geometric) relation between `Velocity` and `Displacement`/`Position` is the exponential map and its inverse, the logarithmic map.

Euclidean space has **no absolute origin**, which is why mathematicians call it an homogeneous space, and why (the instances of) all geometric entities above can only be *relative*, with respect to other instances. Some entities are *arbitrarily* chosen as *reference* for the others. For example, a the `Position` of a `Point` is not completely defined if its coordinates are not interpreted with respect to “world” reference frame.

This document uses the symbol $\boldsymbol{p}_{\{w\}}$ to indicate that a point $\boldsymbol{p}$ is defined in the reference frame $\{w\}$. Obviously, a *rigid body* can serve as geometric reference too; in this case, the symbol $\boldsymbol{p}|\mathcal{A}$ denotes that the point p is attached to the body $\mathcal{A}$.

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 entities. A position relation is *directed*, that is, the previous notation is semantically equivalent to “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 relation 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.

Displacement Frame — A displacement frame is a *composite* entity that represents an orientation and position, by means of an Orientation and a Position, the latter acting as the orientation's frame origin. This document uses the symbol $\{a\}, \{b\}, \ldots$ to denote displacement frames.

Pose — A *pose* is the **composite** geometric relation between two displacement frames, e.g., $\{g\}$ and $\{h\}$, or between an equivalent couple of pairs of position and orientation entities, 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 entities are attached to different rigid bodies $\mathcal{C}$ and $\mathcal{D}$, the pose relation 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 $\underline{\text{Pose }((\boldsymbol{e}, [a])|\mathcal{C}, (\boldsymbol{f}, [b])|\mathcal{D}, [r])}$.

Linear velocity — A *linear velocity* is a relation 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 entity is arbitrary and equivalent, as long as it is fixed to $\mathcal{D}$.

Angular Velocity — An *angular velocity* is a relation between two orientation frames, ex-

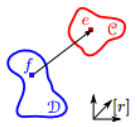
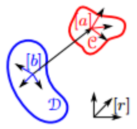
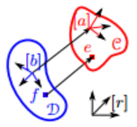
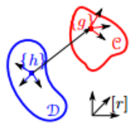
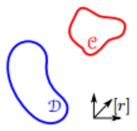
Geometric Relation	(Coordinate) semantics	Geometric primitives	Graphical representation
Position	Position $(e\|\mathcal{C}, f\|\mathcal{D})$ PositionCoord $(e\|\mathcal{C}, f\|\mathcal{D}, [r])$	point e body $\mathcal{C}$ reference point f reference body $\mathcal{D}$ coordinate frame $[r]$	
Orientation	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 3.2: 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 entities for position, orientation, pose, twist, linear and angular velocity, and their graphical representation. This is an excerpt from [15].

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

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

Acceleration twist, or **Rigid body acceleration** — At the level of the second-order time derivative, things become a little bit more complicated, since there are two different types of time derivative of a Twist:

- **Eulerian** (or **ordinary**) time derivative: one looks at the matter that flows under one particular fixed point in space, and reports on the change of the pose or velocity observed over time. So, the `acceleration` is the difference between the `velocities` of *two different* particles, at the *same place* in space but at *different instants in time.*

- **Lagrangian** (or **material**) time derivative: one follows one particular point fixed on a rigid body, reports on the change of its pose or velocity over time. So, the `acceleration` is the difference between the `velocities` of the *same particle* at *two different instances in time*, and hence also at two *different places* in space.

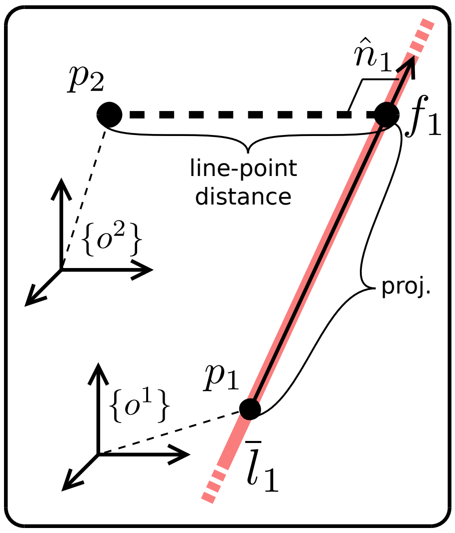
To derive `velocity` from changes in `pose`, the Eulerian time derivative equals the Lagrangian time derivative; they are also equal for `acceleration` from *rest*, i.e., zero initial `velocity`. Different domains use these two different definitions, most often without explicit information, which results in non-composability problems in robotics systems that must integrate both domains.

(TODO: references needed.)

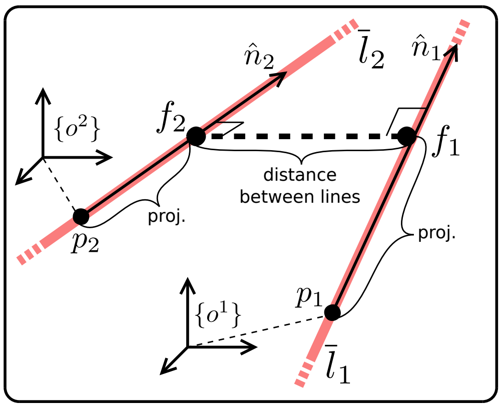
Motion — This document uses the term “motion” as an aggregate of all position, velocity and acceleration entities and relations.

Forces, Torques and Wrenches — These physical concepts are *not* geometrical, because they are the link between the *extra* concepts of **mass**, **damping** and **stiffness**, and the above-described **motion** concepts. Part of their semantics can be given a geometrical *abstract data type*, because there is an analogy between twists (composed of linear and angular velocities) and wrenches (composed of force and torque vectors). This geometric part is: $\underline{\text{Force}(\boldsymbol{e}|\mathcal{C},\mathcal{D})}$, $\underline{\text{Torque}(\mathcal{C},\mathcal{D})}$, and $\underline{\text{Wrench}(\boldsymbol{e}|\mathcal{C},\mathcal{D})}$.

Linear and Angular Distances — Another useful geometric relation is the concept of *distance*, which is a *higher-order relation* among the entities and relations 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 relations between certain entities. An example is depicted in Figure 3.3a: 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 3.1 and Table 3.2 resume the different interpretations of linear distances and angular distances, respectively, considering some geometric entities enumerated in Section 3.3.1; these relations are also shown in Figure 3.3.



(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 3.3: Graphical representations of the five possible relations between a point and a line 3.3a, and between two lines 3.3b.

Table 3.1: Summary of linear distance relations between point and line entities.

	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	

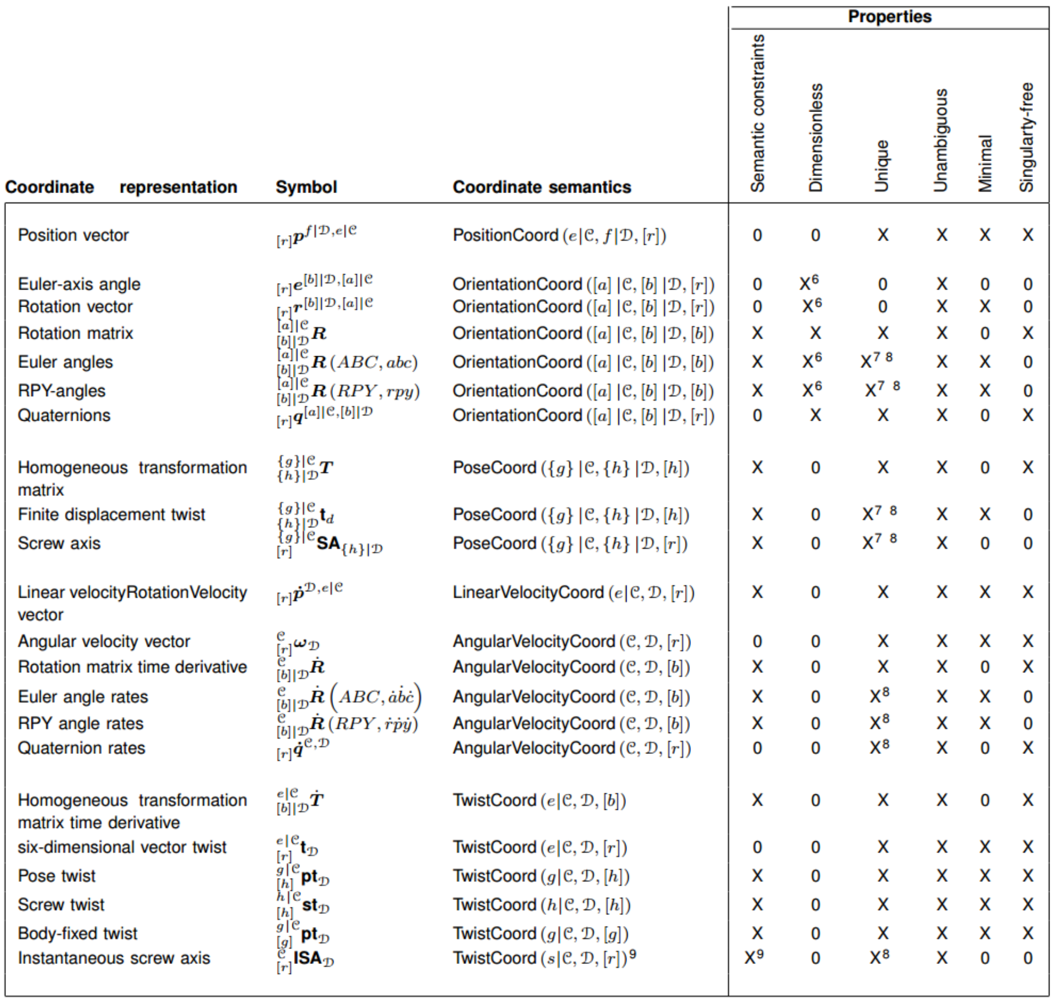
The choice of a coordinate representation is not unique for expressing a geometric concept. Moreover, this choice often implies hidden assumptions or ambiguities. To this end, Figure 3.4 resumes most of the commonly used coordinate representations and their properties, such as uniqueness of the representation, ambiguity, introduction of singularity representation, and minimality.

The **composition** of `motion` relations has the following behaviour:

- `pose` compositions form a *multiplicative group*.
- `velocity` compositions form an *additive group*, also without natural metric, but with a natural origin of "zero velocity".

Table 3.2: Summary of angular distances relations between versor and plane entities.

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



Coordinate representation	Symbol	Coordinate semantics	Semantic constraints	Dimensionless	Unique	Unambiguous	Minimal	Singularity-free
			Properties					
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	X6	0	X	0	0
Rotation vector	${}_{[r]}\boldsymbol{r}^{[b]\|\mathcal{D},[a]\|\mathcal{C}}$	OrientationCoord $([a]\|\mathcal{C}, [b]\|\mathcal{D}, [r])$	0	X6	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	X6	X7 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	X6	X7 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	X7 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	X7 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	X8	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	X8	X	X	0
Quaternion rates	${}_{[r]}\dot{\boldsymbol{q}}^{\mathcal{C},\mathcal{D}}$	AngularVelocityCoord $(\mathcal{C}, \mathcal{D}, [r])$	0	0	X8	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	X9	0	X8	X	0	0

Figure 3.4: Some commonly used abstract data type representations and properties, linked to their type semantics. This is an excert from [15], and it expands Table 3.3.

- `acceleration` composes *additively* but it depends nonlinearly on `velocity`, so it is not really a group operation, just a manifold.

3.3.3 Map and World Model

A `Map` is a set (**composition** relation) of instantiations of all the geometric entities and relations of the previous two Sections.

A `worldModel` is a `Map` **composed** with **semantic tags** (or "annotations", or "meta data") for some of the `Map`'s entities or relations. The composition relations in world models are trivial and mereological, and not restricted to just the geometric primitives introduced above. The obvious extensions are that of geometric chains and kinematic chains, but also tags that refer to the parts in the Task meta model (`plan`, `control`, `perception` and `monitoring`).

3.3.4 Meta model of geometric chains

Robotic systems must keep track of the relative position and motions of several entities, and these often have a **chain relation**: there is a sequential order of some kind in which poses and their time derivatives are to be computed. For example, a cup can be placed on a table, that in itself is standing on the floor, which itself is located in a particular room. This situation represents the common case where the motions of several (rigid) bodies have **constraints** that are due to **natural** causes, such as gravity and the stiffness of the bodies' materials. The next chapter describes the case of **engineered** motion constraints, in the form of "robots" or "kinematic chains"; note the subtle difference in the terminology of "geometric" and "kinematic" chains.

(TODO: more concrete representations and discussion.)

3.4 Meta model of rigid-body dynamics

The previous Section considered only the geometrical behaviour of geometric primitives, but this Section adds the main **dynamics** behaviour relations, namely the procedural **Newton-Euler equations** and the declarative **d'Alembert-Lagrange-Hamilton-Gauss relations** [14, 21]. The major modelling step is the composition of the inertial effects of two rigid bodies connected by a frictionless and rigid one-degree of freedom motion constraint.

(TODO: a lot more details.)

3.5 Software meta model

The software model is a composition of models that ground **mathematico-physical concepts**[4] (entities and/or relations) with a **formal representation** in a machine-readable form, that is, in a **programming language**. For example, a `length` gets a *positive real number* as its software representation, and a `free-vector` gets an *array of three real numbers* as its software representation. In the context of the motion and perception stack, special attention is given to the grounding of **geometrical** concepts since they are the backbone of any robotic application. The **grounding** of a geometric entity as a numerical entity is depicted in Fig. 3.1, as the **composition** of the following models:

- a **abstract data type** representation defines the numerical representation of a physical (mathematical, geometrical,...) entity; for example, a spatial moment can be represented by a free vector; or by a couple of two line vectors of equal length but opposite direction, and on parallel lines.

- a **digital data representation** model specifies how numerical values are stored in memory (as part of algorithms), or in a communication object (as part of software component architectures); for example, a free vector's digital data representation van be an array of three 32-bit float numbers.

[4] "Physical concept" can also mean "mathematical concept" (such as a tensor, or an integral), or "geometrical concept" (which is a specific set of mathematical concepts, such as planes, or cubes, or free vectors.

- a **physical units and dimensions** model to attach to the numerical values; for example, the spatial moment has the property that its *dimensions* are `force` times `length`, and it can be given the attribute that the chosen *units* are `Newton-meter`.

- an **uncertainty** model to attach a "measure of information" about how much one knows about the various possible numerical values of the represented entity.

While a digital data representation is explicitly present in any implementation, the physical units and coordinate representation models are often forgotten or taken as implicit assumptions in the implementation. Nevertheless, *all three* are needed in order to avoid any ambiguity in the interpretation of the numerical values in the digital data representation. (*Uncertainty* is not needed in all application contexts.) A knowledge-driven approach to robotics mandates the usage of all these explicit models, in one single composition, to achieve composability and compatibility (structurally and semantically) among functional software elements. This is especially required 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.

3.5.1 Digital data representation

A *digital data representation* 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/ and serialising, etc.).

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 concept is further explained in Sec. 3.7, while further examples are given in Sec. 3.5.2.

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 existence of an explicit digital data representation model must be considered a key-element towards composability of systems-of-systems.

3.5.2 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 relations 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 3.5; a concrete example is discussed in Figure 3.6.

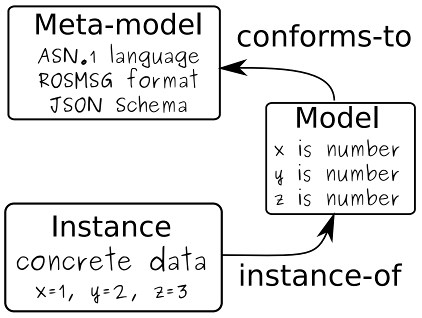


Figure 3.5: Digital representation models: a data structure in software is an `instance-of` a digital data representation model, which is a formal description that `conforms-to` a meta-model. A concrete example is shown in Fig. 3.6.

To evaluate the positive impact of a digital data representation meta-model (and its underlying tools) as bones of a composable software solution, the following aspects must be evaluated:

- **expressivity** of a meta-model to describe different properties over the data, including:
  - basic, built-in data types available;
  - possibility to indicate *constraints* on the data structure;
  - customisation over the memory model to store the data (instance of the model);
- **validation**: availability of a formal schema of the digital data model, meta-model and tools to validate both data instance and model schema;

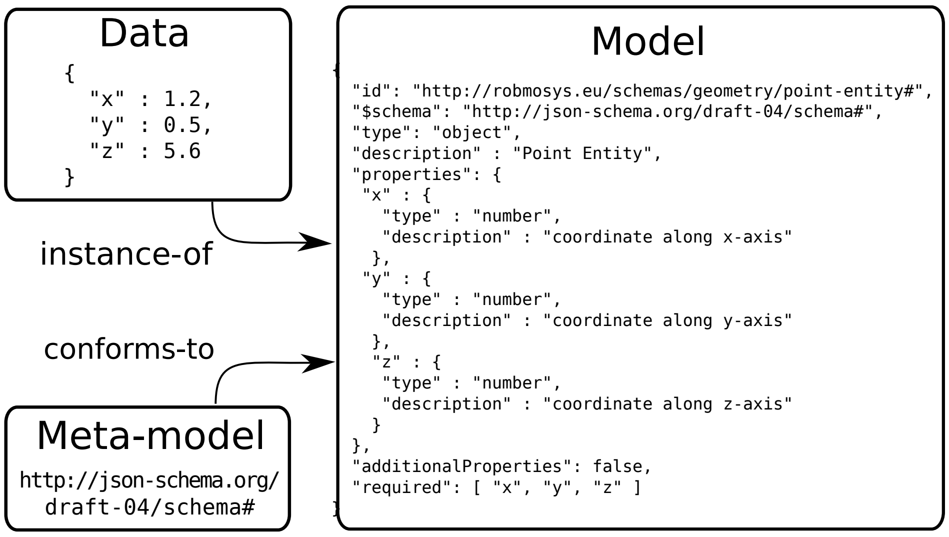
- **extensibility**: the possibility to extend (by composition) the expressivity level of a digital data representation model;
- **language interoperability** (also called neutrality): the capability of a model of being language-indipendent; this requires a specific compiler to generate code-specific form of the digital data representation model;
- **self-describing**: optional capability of injecting the model in the data instance itself (meta-data), 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 [13]. 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 3.6.

JSON-Schema is considered a composable approach, since (i) JSON supports associative array (only strings are accepted as keys) and (ii) JSON-Schema supports JSON Pointers (RFC 6901) to reference (part of) other JSON documents, but also objects within the document itself. This allows to compose a schema specification from existing ones, and to refer only to some specific definitions. JSON-Schema is used in web-technologies and it is very flexible in terms of requirements needed to be integrated in an application. However, it is verbose with respect to other alternatives, as well as not efficient in terms of number of bytes exchanged with respect to the informative content of a message. In fact, JSON-Schema does not provide native 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 3.6 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.



```
{
  "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" ]
```

Figure 3.6: 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).

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 [53] 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.

FlatBuffers

(TODO: binary format for serialization, storage and communication, with a model in JSON, and with direct access to any subset of the composite abstract data type. Not self-describing, but its model can be composed into the data as a string.)

Google Protocol Buffers

Google protocol buffers are another digital data representation meta-model dedicated to serialising structured data, without direct access but with an efficient encoding mechanism to have a small impact in terms of memory or message size. It has the option to include a self-description, that is, the digital data representation model can be packed together with an instance of its data 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[5]. 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[6]).

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 representation model if a dedicated tool is supplied. For example, tools that generate a typekit starting from a ROS message definition exists.

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

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

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

middleware or programming language and provides grounding (through code generation) into different communication middlewares, including CORBA IDL, the message-based *Adaptive Communication Environment*[7] (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.

3.5.3 The role of digital data representation models

The digital data representation concerns all the roles defined in the modelling 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).

Any software implementation makes use of a digital data representation, often implicitly defined as built-in language structures. 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. 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, 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.

[7] see http://www.cs.wustl.edu/~schmidt/ACE.html

- **runtime negotiation:** at runtime, two (or more) software entities (i.e., component instances) initialise a negotiation 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 negotiation 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 negotiation phase.

3.6 Physical units and dimensions

The digital data representation (see Section 3.5.1) 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.

In the context of a model-driven approach to robotics, it is obvious to promote the use of ontologies to describe physical units and dimensions. Some suggestions are the QUDT ontology [62] of the W3C, or UCUM (the *Unified Code for Units of Measure*) supported by the Eclipse eco-system.

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,[8] that are often available in software products.

3.7 Abstract data types

Abstract data types express how geometric entities and relations are mathematically represented, assigning to each concept a symbol value, a vector or a matrix with unique semantic meaning. In the context of a motion and perception stack, the basic geometric concepts are those *relations* that describe the *relative* "motion" of rigid bodies, such as: position, orientation, pose, linear and angular velocities, torques, forces, etc. However, there is seldom a unique coordinate representation for a given geometric concept, so a surprisingly large set of choices must be made to get a complete numerical representation. In practice, such a large *freedom of choice* is often the source of incompatibility, and of lack of composability, of models and implementations created by independent development teams, for the simple reason that not all choices are being made explicit, especially not in the formal representations.

Table 3.3 summarizes common choices of coordinate representations ordered by the geometrical concept they express. Those geometrical relations are further formalised and discussed in Sec. 3.3.

Geometrical concept	**Abstract data type**
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	Angular velocity vector Rotation matrix time derivative Euler angle rates RPY angle rates Quaternion rates
Twist Rigid body velocity	Homogeneous transformation matrix time derivative six-dimensional vector twist Pose twist Screw twist Body-fixed twist Instantaneous screw axis

Table 3.3: Commonly used coordinate representations for geometrical concepts.

A coordinate representation has no unique digital data representation (independently of the meta-model used to specify the digital data representation model), because of the allowed *choice* on (i) the *ordering* of the values of the coordinate representation, and (ii) the physical

[8]Often abbreviated as "i18n", for *internationalisation*, that is, the initial "i" and the final "n" with 18 other letters in between.

units. Another choice is about how the data *access* is performed, e.g., via *named keys* or via *anonymous* indexing.

Let's take the example of the grounding of the geometric concept of a *Position*, as the composition of multiple choices, Fig. 3.7:

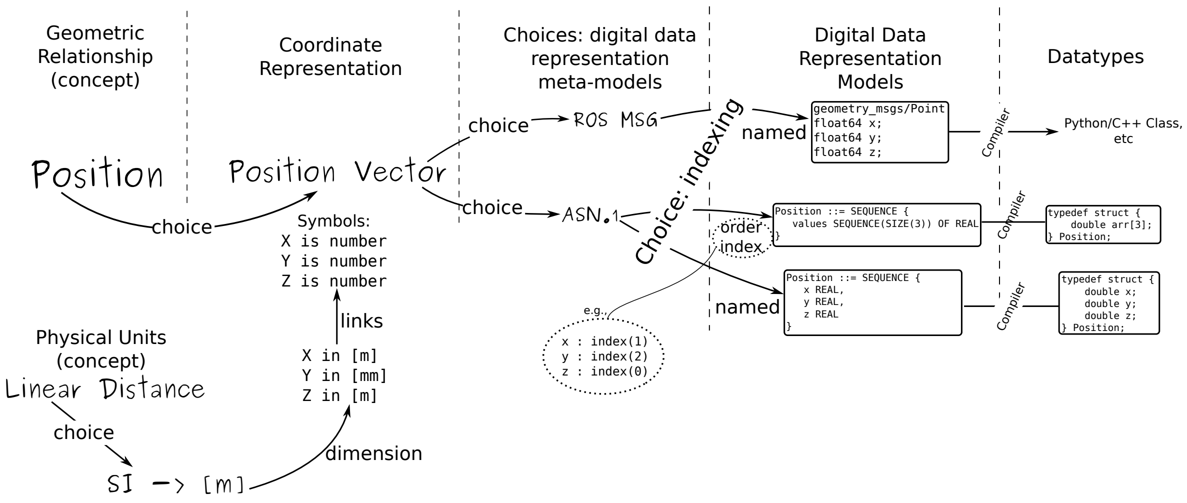
- **which coordinate representation to use, and its symbolic definition?** For *position* relation, the choice is unique and it is a *position vector*, defined as 3 numerical symbols x y and z. In general, this choice is not unique, as shown in Table 3.3;
- **which system measurement to adopt, and which 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 is the data stored and communicated?**

All the above are concrete questions that a component supplier, system builder and application developer must answer, 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 to promote model-driven engineering to enable a systematic and error-free solutions.

3.8 Uncertainty in geometric entities and relations

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

An example is to represent the uncertainty on the position of a point in Euclidean space with a Gaussian probability distribution (also called "normal distribution"), which can be numerically represented by its mean vector and its covariance matrix, Fig. 3.8. This model is also a composition of mathematical models (vector and matrix); additional constraints, like that the covariance matrix must be symmetric, are not modelled, for now. Similarly, a Gaussian mixture model can be created by composing an array of the presented Gaussian models with the constraint used for scaling them appropriately.



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

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

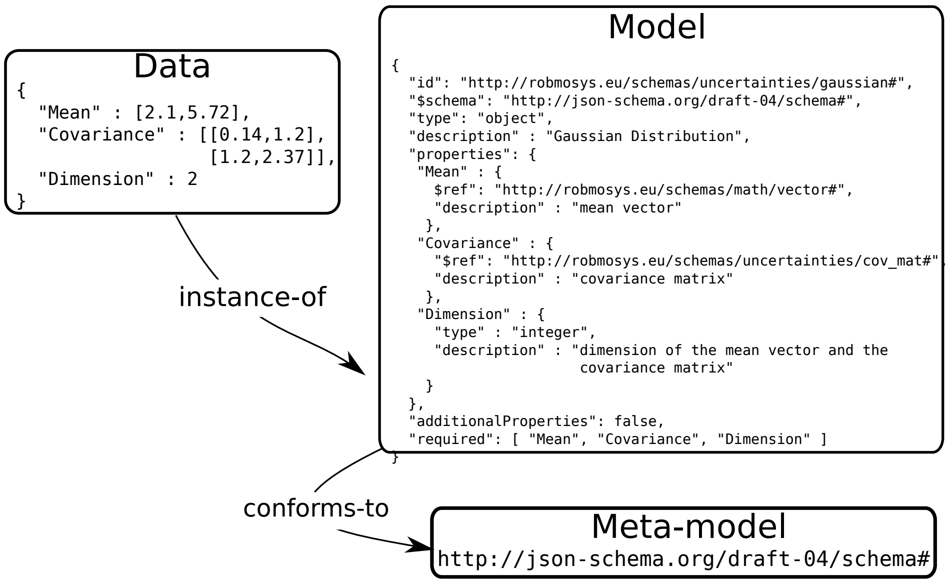
```
Position ::= SEQUENCE {
   x REAL,
   y REAL,
   z REAL
}
```

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

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

Figure 3.7: From geometric relations to digital data representation: choices for the grounding of the concept of *Position*.

There is little confusion, or choice, in how to represent uncertainties on position. But the situation is a lot more complex for most of the entities which consist of compositions of two or more points, such as line segments, lines, areas, frames,... For example, representing the uncertainties on a line segment by means of Gaussian uncertainties on its two end points is *different* from representing it by means of a Gaussian uncertainty on one point together with an uncertain direction vector.



```
{
  "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" ]
}
```

Figure 3.8: 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.
- **categorical confusion**, which represents the uncertainty in knowing to which category a particular entity belongs.

3.8.1 Covariance of a frame has no physical meaning

An important **mathematical fact** is that SE(3) does not have a bi-invariant metric; in engineering terms this means that it makes no sense to talk about the "distance" between two frames, and, hence, also not about the covariance matrix on rigid body motion or force: there is *always* a weighing factor needed to balance the physical dimensions of the translational and angular parts, and this weighing factor is *always* **arbitrary**.

A representation of the uncertainty on a frame, *with* consistent physical meaning, consists of *choosing* **three points** on the frame (e.g., its origin and the end-points of two of the three Cartesian unit vectors), and adding a position covariance matrix to the coordinate representations of all of them. This results in a *non-minimal* representation of the uncertainty; the constraints that have to be added reflect the facts that the two vector must have *unit length* and are *orthogonal.* Of course, this representation *also* introduces an arbitrary weighing between translation and orientation, albeit in an indirect way, via the choice of which points to select in the representation.

Chapter 4

Meta models for kinematic chains and their instantaneous motions

One way to describe "motion of a robot" is as the instantaneous transformation of energy between the robot's **motors** (in the so-called "**joint space**") and its various **end effectors** (in the so-called "**Cartesian space**"), using the **mechanical structure** of the robot's **kinematic chain** as the energy transformer. That mechanical structure adds **motion constraint** relations to the rigid bodies it connects together. This Chapter introduces the models for the instantaneous transformations, and also makes the link with the task meta model: (i) it identifies the actuators as the *resources* and the motions as the *capabilities* of a kinematic chain, and (ii) it provides the modelling entities ("attachments") and relations ("motion drivers" of joint and Cartesian forces, and of Cartesian acceleration constraints) *to specify* any type of instantaneous motion that is physically allowed by the kinematic chain.

Section 3.3 already introduced all the geometric entities and relations required to model a kinematic chain, so this Chapter adds the structural and behavioural entities and relations that makes a "geometric chain" into a "kinematic chain".

4.1 Mereo-topological meta model

The **mereo-topological** meta model of a kinematic chain has four parts: (i) a `collection` of `links`, (ii) mechanically `connected` by means of `joints`, (iii) with zero or more `attachment` points on each `link`, (iv) where the mechanical motion constraints are `connected` to. So, a formal model of a kinematic chain is a `graph` with the following nodes and edges:

- `link`: a node representing a mechanical **rigid body**.
- `attachment`: a node representing a geometric (so, non-mechanical) entity (e.g., point, orientation frame, displacement frames, etc.) `connected` to a `link`, to serve as an argument in all sorts of **coupling relations** in which the `link` can be involved, e.g., (models of) inertia, shape, sensors, tools, joint damping and elasticity, motion ranges, control actions, etc. It is the core model primitive for **composition** with all other relevant domain models.

- `joint`: an edge representing the **constraint relation** of the mechanical motion constraint between two `links`, with the `attachment` points on the `links` as arguments of the constraint relation.

  The **type** of the motion constraint is an inherent part of the semantic meta data of the `joint` model. One-dimensional revolute and prismatic joints are very common as bi-directional motion constraints, but also uni-directional constraints such as cables or surface contacts are types provided in the mereology part of the meta model.

Of course, in this document's context of higher-order modelling, the latter edge relation becomes a node in a more encompassing property graph that represents the whole kinematic chain. As with the `joints`, it makes sense to include in the mereological part of the meta model an *enumeration* of types of kinematic chains, called `kinematicFamilies` (Sec. 4.5).

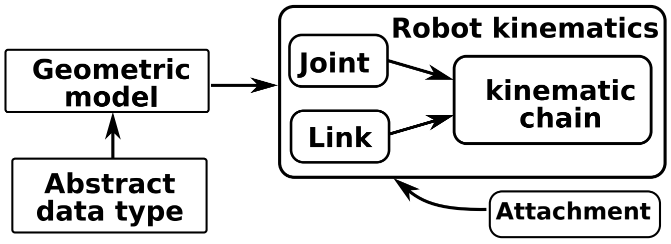


Figure 4.1: A robot kinematic model is defined as a graph of *link* and *joint* entities, which are themselves defined as compositions of *geometric* entities (Sec. 3.3, Fig. 9.1) and their *abstract data type* representations (Sec. 3.5). Each link can have one or more *attachments*, to allow extensions by means of *model composition*, such as, geometric shape, dynamical properties, sensors, actuators, or handles for task specifications. An arrow represents a `is-part-of` relation, which is the *inverse* of the `has-a` relation.

4.2 Geometric meta model

This Section adds **geometrical** entities and relations to the *mereo-topological* ones of the previous Section, to model the **motion behaviour** of a kinematic chain, that is, the **constraints** that the chain adds to the relative `motion` (`Position`, `Velocity` and `Acceleration`) of the individual `links`:

- it makes the *constraint* that a `link` must be a rigid body[1] explicit, by constraining the *distance* between a selection of `Points` on the `rigidBody` to a given value. Being tagged as `rigidBody` in this way is a *symbolic* constraint on the *type* of entities that can serve as a `link`. The model of what a `rigid body` means is already part of the geometric meta model of Chap. 3, namely that the *distance* between all points in the body remains always the same, but the meta model prescribes that an explicitly identified subset of all points must be provided. The distance constraint need often not be modelled explicitly as a `distance` constraint, since it suffices to give the *coordinates* of the `Points` together with the symbolic constraint relations of "fixed distance" between these `Points`.

[1]Flexible links are not treated in this document; however, the mereo-topological modeling still applies, since rigidity is a geometric concept.

- an `attachment` get the geometric relation that is it `connected` at a particular `position` on a `link`. While the model of such an attachment position is already part of the geometric meta model of Chap. 3, the added relation in the context of the kinematic chain meta model is that the representation of motion constraints *requires* particular *types* of the attachments.

- a `joint` is the mechanical coupling (or "**motion constraint**") between two `attachments` on two[2] `links`, and the geometric meta model of the kinematic chain adds a higher-order relation over the geometric relations between the `connected` set of `attachment` points. The arguments in the mathematical expression of motion constraints are the positions of the `attachment` entities on the `links`.

- the `position` between two `links` is the "Cartesian space" part of the just-mentioned mathematical relation. It represents which geometric operation must be applied to the geometric models of the `attachment`(s) on the first `link`'s rigid body, to make it coincide with the `attachment`(s) on the second link. Examples of such geometric references are: a `frame`; a combination of four non co-planar points; a combination of three non co-planar lines; a combination of a point and two intersecting lines not co-planar with the point.

- `velocity` and `acceleration` are the **time derivative** relations of the `position` relation. The constraint between the `position`, `velocity` and `acceleration` of one `rigid body` is, in differential geometric terms, represented by the concept of a **jet**.

- `motion` is the **composition** (in the least constrained form of an aggregation or of an unordered set) for `position`, `velocity` and/or `acceleration`.

For motion constraints that are most common in robotic kinematic chains, that is, `revolute` and `prismatic` joints, the extra constraint implies that one needs only one single scalar, the `joint-position`, to represent the relative pose of the links connected by the joint.

Figure 4.2 depicts a joint constraint relationship, with a `joint-position` *value* $\mathcal{J}_{\text{cstrval}}$; the dimensionality of $\mathcal{J}_{\text{cstrval}}$ is a number between "0" (rigidly connected) and "6" (not constrained at all). The coordinate representation of the constraining value must be compatible with the coordinate representation of the geometric relationship, and it can either be a constant or another expression. For revolute and prismatic joints, this expression is a *function* of just one variable (often denoted by the scalar q) that identifies the *state* of the joint, e.g., `joint-position` q, `joint-velocity` $\dot{q}$, and so on. More complex joints do not have clearly defined `joint-positions`, e.g., the knee and shoulder joints in human bodies, or the unilateral motion constraints of contacts or cables (Fig. 4.9); in this case, the state of the constraint is the relevant concept, and not the state of the joint position.

Figure 4.2 shows a kinematic model example, where the links $\mathcal{L}_1$ and $\mathcal{L}_2$ are mechanically constrained by a joint (whose type and geometry are not made explicit in the Figure). The joint is defined by means of a single kinematic constraint (i.e., a relative pose between the links). In this example, the joint is modelled by one single, pose-based kinematic constraint. However, it is common practice (i) to define a joint by means of a single velocity-level kinematic constraint, for **"non-holonomic"** motion constraints such as wheels or skates; (ii) to

[2]Or more than two attachments on more than two links, e.g., human joints like the shoulder have tendons connecting multiple bones to multiple muscles.

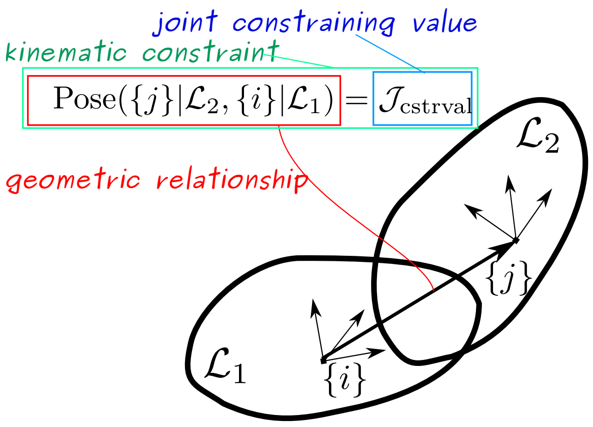


Figure 4.2: The simplest kinematic model example, with two links and one joint (of unspecified type). The frames $\{i\}$ and $\{j\}$ serve as *attachment points*, fixed to the *links* $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively. The *joint* between $\mathcal{L}_1$ and $\mathcal{L}_2$ is a kinematic constraint that imposes a constraining value (set) $\mathcal{J}_{\text{cstrval}}$ to the pose relationship defined between $\{i\}$ and $\{j\}$.

define a joint as a combination of kinematic constraints, including inequality constraints; for instance (iii) **higher pair** motion constraint inequalities based on linear or angular velocity relationships can express the boundaries of the minimum and maximum achievable velocities; and (iv) **joint limit** constraint inequalities based on pose relationships can express the limited working range of a revolute or linear joint.

4.3 Motion composition meta model

Section 3.3.2 described motion composition for *unconstrained* rigid bodies, hence outside of the context of kinematic chains, where the `poition`, `velocity` and `acceleration` of a `rigid body` do not depend on those of other `rigid bodies`. This Section adds the **composition relations** for the `motion` meta model that are unique to the kinematic chain context:

1. `serial-compose`: the `positions`, `velocities` and `accelerations` of *all* `links` are found from (i) the **unconstrained** "Cartesian space" `positions`, `velocities` and `accelerations` of *individual* `links`, and (ii) the "joint space" **constraints** at joint position, velocity and acceleration level. **Physically**, the motion constraint can only be modelled at the level of the full mechanical dynamics of the links and joints; many applications are only interested in the *kinematic* versions.

   Figure 4.3 sketches the geometric primitives involved in such a serial composition. The core composition relation is the one that connects *two links* over *one motion constraint*; all other serial compositions are *sequences* of the same core composition.

2. `branch-compose`: this is the composition of `serial-compose` with the extra structural constraint that there is one `link` from which three (or more) serial connections branch out.

3. `loop-compose`: this is the composition of `branch-compose` with the extra structural constraint that the branching `link` is connected twice to the same `link`, at the latter's two ends.

The numbering choice reflects the dependency structure: `loop-compose` depends on `branch-compose`, which depends on `serial-compose`. Of course, higher-order compositions of these composition relations are allowed too (for example, loops within loops, like one finds in the musculo-skeletal structure of animals and humans), and the connections between two `links` in a `serial-compose` can be a `joint` of any type, including one that has in itself the structure of a parallel chain as in "snake or "elephant trunk" designs. Figures 4.6–4.9 give examples in which most of the above compositions appear.

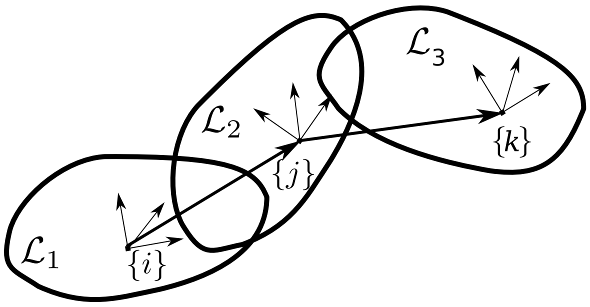


Figure 4.3: Serial composition of `motion` relations of kinematic chains.

4.3.1 Policy: input-output causality assignment

The meta models represent *relations* between `motion` entities, which are the **a-causal** *mechanism* describing how the properties of the various enities in the motion are related. Most applications require causal relations, or *input-output functions* as they are called more often, with the following **arguments**:

- a *model* of the kinematic chain;
- the *list* of geometric primitives which are *given* as inputs; and
- the *list* of geometric primitives which must be *computed* as outputs.

One single a-causal relation gives rise to many conforming input-output functions, one for each combination of input and output choices.

4.3.2 Inconsistency, redundancy, singularity

Because kinematic chain relations are **non-linear functions** of their constituent geometric entities, some input-putput functions can be:

- **inconsistent**: the requested output can not be computed with the given inputs and the given kinematic chain model, because, for example, the chain has two loops when one is expected, or it has one loop when none is expected, etc.
- **redundant**: more than one output is consistent with the same input.
- **singular**: the computations to find the outputs are numerically ill-conditioned, because the physical energy transformation between Cartesian and joint space reaches an extremum.

While these insights are already part of the meta models, it is really only in *software* implementations that they pose challenges: simplistic implementations will crash because the problems often occur only in specific configurations of a kinematic chain, or with specific input-output combinations.

4.4 Dynamics meta model

This Section extends the geometrical meta model by adding the *mechano-dynamical* entities and relations connected to the **mass** of the bodies involved, and to the forces acting on the bodies, directly or via **springs** or **dampers** between the bodies. The sources of the forces can be externally generated forces on the links (including gravity) but also internally generated forces or torques at the joints via actuators and transmissions.

From a structural point of view, a meta model of mass, spring and damper is easily composed with the already described geometric models: just use an attachment point to connect a model of the rigid body mass to a `link`, and use two attachment points on different links to connect with the elasticity and damping relations. 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.

It seems logical to decouple the geometric and dynamic meta models of kinematic chains, because the introduction of masses, springs and dampers can be composed onto the geometric models via `attachment` primitives. But the dynamic properties occur already at the geometric level, be it in disguise: the above-mentioned challenges of `redundancy` and `singularity` can only be solved by introducing "weighing" or "cost" functions, that measure the magnitude of the problems; the mathematical formulation of these functions reveals that they *must* have the physical dimensions of mass or elasticity. Hence, *the* **behavioural** meta model for kinematic chains will **couple the geometrical and (mechanical) dynamical entities and relations** together, all the time.

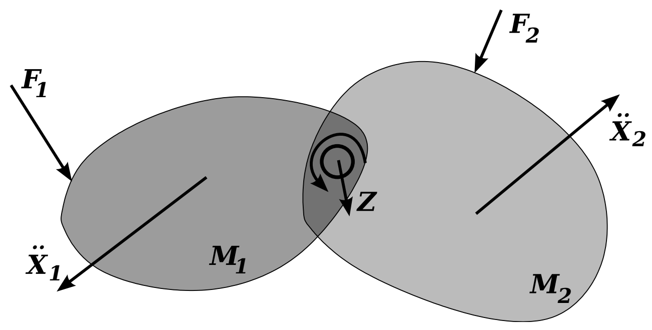


Figure 4.4: The entities involved in the dynamic behaviour of a (revolute) joint constraint. The spatial direction ($\boldsymbol{Z}$), force ($\boldsymbol{F}$) and acceleration ($\ddot{\boldsymbol{X}}$) have *six* degrees of freedom; the mass ($\boldsymbol{M}$) is a 6×6 relation between them.

4.4.1 Dynamics of a one-joint kinematic chain

Figure 4.4 sketches the dynamical behavioural relations for the simplest possible `serial-compose` case of a kinematic chain: (i) *one* single `joint` between two rigid body `links`, and (ii) the `joint` has only *one* degree of freedom. The Figure shows the example of an *unactuated* revolute joint, but the modelling for other types of joints use the same type of entities, relations, and constraints.

The forces $\boldsymbol{F}_1$ and $\boldsymbol{F}_2$ on both links are connected to their accelerations $\ddot{\boldsymbol{X}}_1$ and $\ddot{\boldsymbol{X}}_2$, their inertias $\boldsymbol{M}_1$ and $\boldsymbol{M}_2$, and to the versor $\boldsymbol{Z}$ representing the joint axis, by the following dynamical constraint relation (expressed as six-vector *mathematical* representations, and not *coordinate* representations):

$$\boldsymbol{F}_1 = \boldsymbol{M}_1^a \ddot{\boldsymbol{X}}_1, \tag{4.1}$$

$$\text{with} \quad \boldsymbol{M}_1^a = \boldsymbol{M}_1 + \left(\boldsymbol{I} - \boldsymbol{Z}\left(\boldsymbol{Z}^T\boldsymbol{M}_2\boldsymbol{Z}\right)^{-1}\boldsymbol{Z}^T\right)\boldsymbol{M}_2. \tag{4.2}$$

$\boldsymbol{M}_1^a$ 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. (The matrix in front of $\boldsymbol{M}_2$ satisfies indeed the conditions for being a projection matrix.)

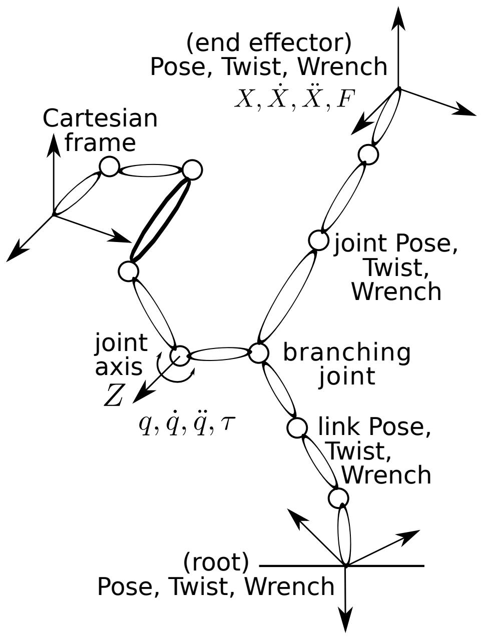


Figure 4.5: The entities in a geometric and mechano-dynamical model of a kinematic chain.

4.4.2 Dynamics of serial kinematic chain

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

4.4.3 Dynamics of branched kinematic chain

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

4.4.4 Dynamics of kinematic chain with a loop

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

4.5 Taxonomy of kinematic chain families

From a modelling point of view, it makes a lot of sense to introduce "families" of models, as a simple mereological higher order model for **classification**, because the kinematic chain structure of most robots falls within one such category. The major added value of the `kinematic family` relation is to collect the constraints that are shared between all members of the family, such as: the number of joints, the type of the joints, the singular configurations, and the abstract data types representing joint space and Cartesian space state. The most common top-level members of these taxonomies are:

4.5.1 Serial chains

Serial kinematic chains, or "arms" (Fig. 4.6):

- **Serial-321**: traditional industrial robots
- **Serial-3-X**: many "cobots" like Universal Robot, or Mabi.
- **Serial-313**: KUKA iiwa, ABB YuMI,...
- **SCARA**

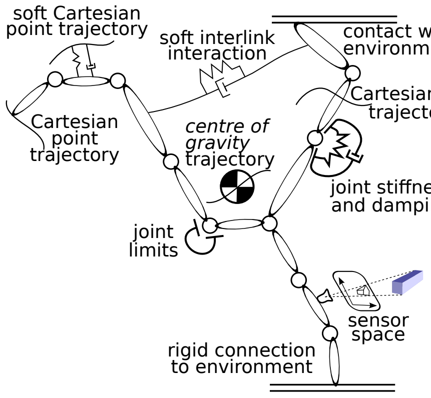

soft Cartesian point trajectory
soft interlink interaction
contact with environment
Cartesian link trajectory
Cartesian point trajectory
centre of gravity trajectory
joint stiffness and damping
joint limits
sensor space
rigid connection to environment

Figure 4.6: Sketch of the kinematic chain model of a dual-arm manipulator, with all(?) possible `motion` constraints on `links` and `joints`.

4.5.2 Mobile platforms

- **DifferentialDrive**
- **Holonomic**:
- **EightWheelDrive**: (Fig. 4.7)

4.5.3 Parallel chains

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

(Fig. 4.8):

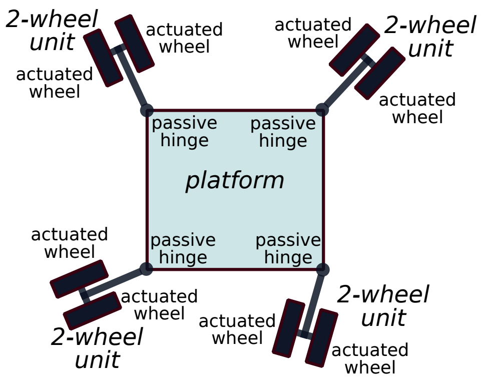


Figure 4.7: Sketch of the kinematic chain model of an over-actuated mobile robot: eight actuated wheels, connected in so-called 2WD (*"two-wheel drive"*) pairs, that in turn are connected to a rigid platform via passive revolute joints with a caster offset. The design drivers behind this platform are (i) passive backdrivability in all configurations, and (ii) holonomic motion behaviour in all configurations.

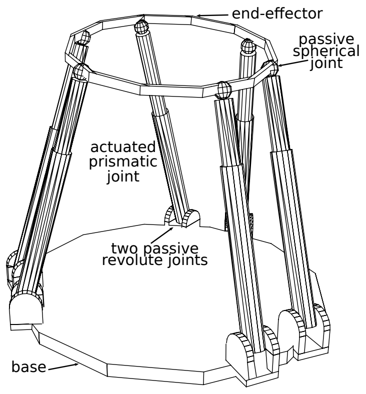


Figure 4.8: Sketch of the kinematic chain model of a parallel kinematic chain, with six legs each with six 1D joints.

4.5.4 Multirotor

4.5.5 Hybrid chains

featuring one or more "loops" in the chain's topology.

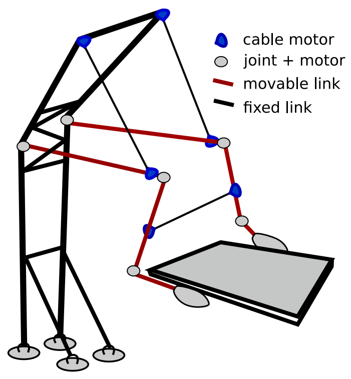


Figure 4.9: Sketch of the kinematic chain model of a cable-driven robot, with a "hybrid" topological structure.

4.5.6 Cable-driven chains

Fig. 4.9,

4.6 Taxonomy of motion capabilities

Most existing robot systems have intended functionalities that are the **composition** of two or more of the following:

- **mobility**: to take care of the global navigation towards targets over the earth, driven by wheels, legs, propellers,...
- **reach**: the "arms" or "manipulator" navigate towards targets in local 6D space.
- **grasp**: the "gripper" or "hands" to manipulate objects by *force closure*, *form closure*, or *non-prehensile* grasps. (TODO: references.)
- **touch**: to sense and manipulate by *form features* such as nails, finger tips, or whiskers.

The robotics domain has (informally) introduced *semantic tags* like "mobile manipulators", "humanoids", "gantries", or "welders", as specific compositions in particular application domains.

(TODO: describe `motion` queries that go beyond the traditional frame-based "forward" and "inverse" capabilities, i.e., with all other geometric primitives as motion specification inputs and/or outputs. For example: the inverse velocity kinematics with three points of the end-effector being kept parallel to a plane in the environment, or to move with one point kept moving along a line, with a minimal and maximal speed of progress, etc. Is there also a "taxonomy" to be found in these motion specifications, e.g., based on the taxonomy in the geometric primitives?)

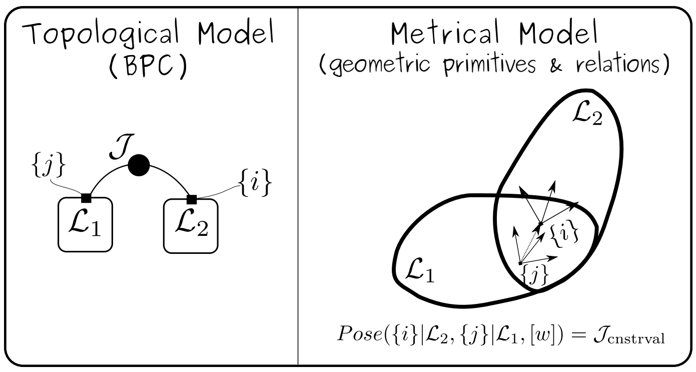


Figure 4.10: A topological model and a metrical model of a two link robot. The topological model conforms to the BPC meta meta model, while the metrical model introduces the geometric primitives and relations that allows to define a metric for the computation over the kinematic chain.

4.7 Conformance to Block-Port-Connector

The mereo-topological part of a kinematic chain model conforms to the *Block-Port-Connector* meta meta model (BPC), in an obvious way:

- a `link` is a `block`.
- an `attachment` point is a `port`.
- a `joint` is a `connector`.

The other levels of abstraction of the kinematic chain model, starting with its geometry, add further domain-specific extensions, also conforming to the BPC meta model (Fig. 4.10):

- a `displacement frame` is attached to a rigid body (e.g., $\{j\}|\mathcal{L}_2$), via the `attachment` point `port` entity. The `connector` between the frames $\{i\}$ and $\{j\}$) expresses a geometric relationship between links $\mathcal{L}_1 and \mathcal{L}_2$).

- a `joint` expresses a set of kinematic constraints, and hence is a `connector`. It 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 to generate the joint *torque*, etc)

The joint in Figure 4.10 `connects` only two links, via their `ports`. In general, it is possible to connect more than two links to the same joint, but this structural pattern is not commonly used by robot designers; it does appear in parallel kinematic chains, in "shoulder" and "knee" joints, etc.

4.8 Physical hierarchy in the motion stack

This Section combines all the semantics of the physical world introduced in the previous Sections, in a generic **hierarchical structural model** of motion stacks, from the "bottom" of **energy sources** up to the **coordinated motion** of several robotic systems. The **purpose** of making an explicit hierarchy model is twofold:

- to get **community agreement** (and eventually community-driven **standards**) about the collection of entities and relations that must be modelled, how they are best collected in sub-structures, and how these sub-structure models are constrained and interconnected themselves.

- to allow model, code and tool developers to delineate the **scope** of their efforts explicitly, hence "signing a contract" about which structures, behaviours, representations and software their tools must provide. For example, it is possible to have a small scope of only the kinematic transformations in a software component, but also a large scope where the energy level in a battery can influence the motion control constraints in an assembly task executed by a mobile manipulator.

This Section only *sketches* the overall *structure*, and does not provide all the concrete formulas in formal models. Even for the simplest robotic system, such complete level of formality involves many hundreds of entities and relations, and an order of magnitude more mathematical, numerical and digital representations. The robotics community should deliver, eventually, a large amount of these, in clear formal ways, and with concrete tools and software implementations; it is indeed to be expected that only a "complete" ecosystem of models, tools and software will suffice to create adoption in the (industrial) community…

For the sake of concreteness, the semantics of *battery-driven mobile robots* has been used in presenting the structure below, but that structure of relevant entities, relations and constraints holds for all robotic and actuator systems:

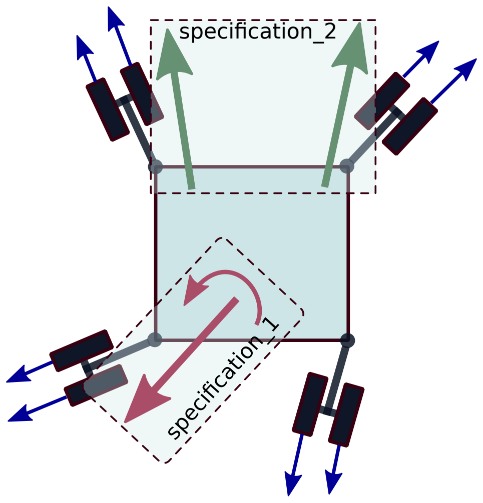


Figure 4.11: Transmissions of forces between actuated wheels (blue traction vectors along wheels' rolling direction) and the (red) force and moment on the platform the wheels are attached to. (The green duo of forces represent an alternative way to specify desired/actual inputs to the robot platform.) This relationship model is composable because the combined effect of all actuator forces are physically realised by simple vector addition.

1. **battery** as an electrical **energy source**: it can provide electrical energy (current and voltage) via well-known impedance relations, and with constraints on maximum and minimum power, voltage levels, temperature dynamics, etc.

2. **energy transformation** between AC or DC **electrical energy** into AC energy for (a)synchronous motors: the battery energy is transformed into mechanical energy via impedance relations of multi-phase, multi-pole motor models, and with torque/speed/efficiency constraint curves.

3. **energy transformation** via a **mechanical transmission** from the electrical motor to the mechanical joint: the joint "consumes" not only the electrically generated torque for its own motion, but the dynamics of the whole chain are coupled in. A transmission can, itself, introduce extra (mechanical, thermal,…) dynamics, for example, via friction and elasticity, heat generation,…

4. **energy transformation** between the **joints and kinematic chain** to produce **Cartesian space motion** of end effectors (and other link attachment points): these are the "hybrid dynamics" introduced in Sec. 4.9.

5. **task specification** relations between **Cartesian attachment points on a robot and motion targets in the environment**: *only* when the robot is in contact with objects in the environment, the interactions are dominated by mechanical relations of the same type as those of the kinematic chain, but contact-less "interactions" are often *specified* as *artificial* constraints of the same type as the physical constraints.

6. **task specification** relations between **multiple moving robots**: again, these coordinated motions are not impacted by physical relations, but only via *artificial* ones.

The model semantics speak about "energy transformation", because that is the more declarative and non-instantaneous way of expressing the relations, instead of the imperative and instantaneous terms "force" and "velocity". (This is also the difference in approach represented by the (equivalent!) Newton-Euler and Euler-Lagrance theories of dynamics.) Instantaneously, the energy is indeed transmitted via *forces* (Fig. 4.11), and this is a very important fact that *must* be represented faithfully in all models (and hence also software). Indeed, different sources of force can be *added* together instantaneously, which is a major instantiation of the **composability** ambition of the modelling efforts. Position-based relations (positions and poses and their time derivatives), however, are *not* composable consistently in an additive way.

4.8.1 Dynamic meta models for actuators

The models introduced in this Chapter have a **mechanical** focus, hence built on the *joint force or torque* entity, without any modelling of where the mechanical forces come from. Those force generators are grouped under the abstract semantic tag of **actuators**, with more specific families depending on the source of the energy that is transmitted into mechanical force: *electrical* actuators are most popular in robotic systems, followed by pneumatic and hydraulic actuators, and various others. The entities relevant to for example, electrical actuators are:

- brushless DC motor, surface mounted permanent magnet synchronous motor, permanent magnet sinusoidally wounded motor, asynchronous induction motor, switched reluctance motor, synchronous reluctance motor , synchronous reluctance internal permanent magnet motor, stepper motor,...
- stator, rotor, pole pairs, windings, 3 phases, 5 phases,...
- resistance, inductance, reluctance, eddy currents,...
- Field Oriented Control, Maximum Torque per Ampere, Maximum Power Per Ampere, minimum input KVA, efficiency-optimized,...

The relations between these entities, and constraints on these relations (e.g., energy consumption, heat generation, efficiency, power factors,...), are commonly found in the literature, but not yet available in a formal model form that can support code generation with full control flow fusion as mentioned in the previous Section.

4.8.2 Dynamic meta models for energy resources

The actuators are not the "deepest" level of physics that is relevant in robotics applications, because also the **energy resources** can have an important impact, especially in mobile or in-body medical robots. So, the following additions to the electrical motor meta models above have to be modelled:

- dynamics of power drives, such as MOSFETs, EMC, noise,...
- dynamics of battery management systems, such as energy regeneration,...
- mechanical transmissions, such as inertia, friction, elasticity, play, heat generation,...

4.9 Hybrid kinematics and dynamics solver

(TODO)

Chapter 5

Meta models for robotic tasks and their composition

(TODO...)

This Chapter extends the **task** meta model that was introduced before, to a *robotics* context. Figure 2.4 and Fig. 2.5 sketch a graphical representation of the mereo-topological level of abstraction of tasks in general; Fig. 5.1 repeats the latter figure, for convenience and with a more elaborate caption that fits in the scope of this Chapter. The robotic context is added via meta models (mereo-topological, but also geometrical and dynamical) that represent the **natural** and **artificial** relations and constraints that appear between entities in **robotic actions**. The semantic relation between "task" and "action" is that the former represents **what** has to be done and under which constraints, while the latter represents **how** a task requirement can be realised while satisfying the task constraints. In other words, a task gives **context** to actions.

5.1 Mechanism: action, actor, actant (object), activity, agent

This Section explains the relations between several often-occuring names that are used in this document.

The "**action**" noun is a **semantic hypernym** for the two nouns **motion** and **perception**, and it represents a **model of what happens in the world**. (An action has various forms of interpretation: "actual", or "desired", or "possible', or...) Action models appear in *all* robotic systems, at various levels of abstraction and various levels of resolution, and with a large variety of "performance". All natural languages have already a hierarchical structure (of so-called **hypernyms and hyponyms**[1]) in their semantics, that can be copied one-on-one to formal modelling efforts in robotics. For example, *action* is more abstract than *motion*, which is more abstract than *grasping*, which is more abstract than *pinch grasping*. Or, *seeing* is more abstract than *recognizing*, which is more abstract than *localising*, which is more abstract than *tracking*.

[1]The terminology for these relations is another name for what this document has called the `is-a` relation, and its inverse.

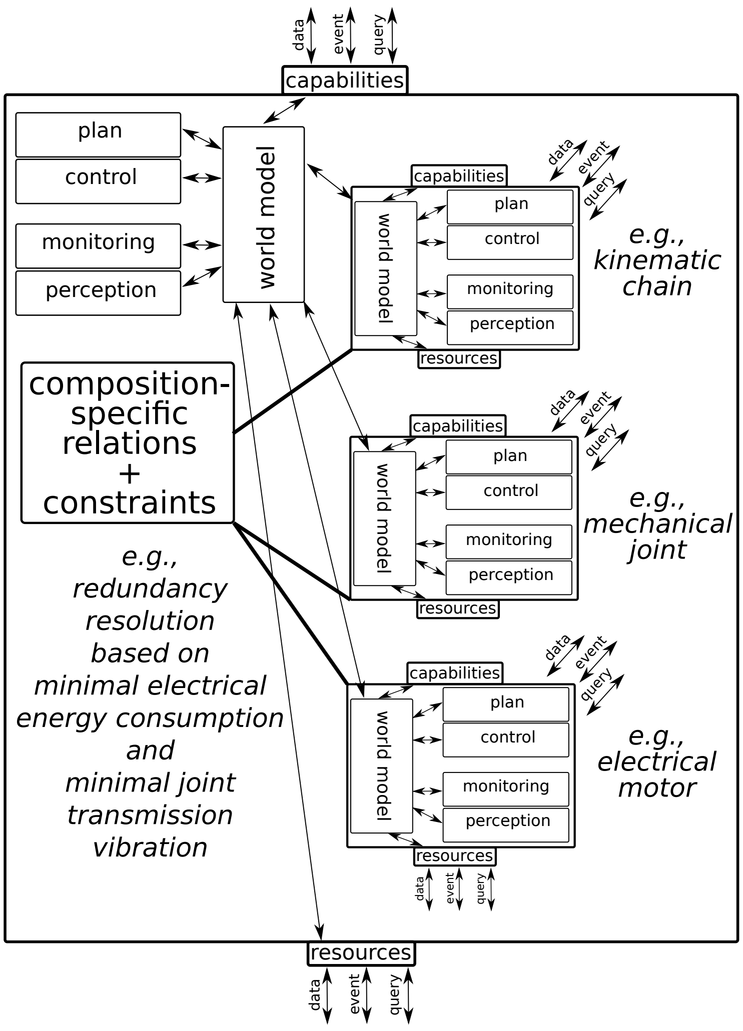

data
event
query
capabilities
plan
control
monitoring
perception
world model
composition-specific relations + constraints
e.g., redundancy resolution based on minimal electrical energy consumption and minimal joint transmission vibration
e.g., kinematic chain
e.g., mechanical joint
e.g., electrical motor
resources

Figure 5.1: Composite `task`, interconnecting one or more `world-models` with one or more `plan`, `control`, `monitoring` and `perception` models, over multiple *levels of abstraction* of a robot's kinematic chain. The extra relations and constraints in the `task` *configure* some of the many "magic numbers" in the modules that exist in its scope, because they link the motion capabilities of the robots, the task expectations, and the constraints imposed by the resources allocated in the execution of the task.

More semantic concreteness comes from a more concrete identification of the **actor** that executes the **action**, possibly using one or more **objects** (or (object) actants. Any robot can move, but only a hand can grasp (and then it grasps "something"), and a pinch grasp is performed with only the thumb and the index finger. At the same time, the **spatio-temporal scope** of each term becomes smaller if the term is attached "deeper" in the hierarchy.[2] Figure **??** represents the main actors in the robotics domain, in their most abstract form, the **robot** and the **environment**, and their **control** and **perception** relations.

The terms action, actor and object, as introduced above, represent **knowledge models** in this document. This knowledge is used in a an **activity**, which is a **software process** that implements the execution of an action, with "digital twins" for actor and object(s). Finally, the name (software) **agent** is given to the **system** of activities that belong together, as being executed by one single physical system in the real world.

[2]The oldest(?) reference that explicitly introduces such natural hiearchy of **increasing order of intelligence with decreasing order of precision** is [46].

5.2 Task ontology

Modelling all relevant task-level compositions is a huge undertaking, but will hopefully and eventually result in a (*standardized*!) collection of domain/application specific **ontologies**; this undertaking is not a main focus of this document, but all of the document's contents serves as a foundation for that undertaking. The good news in the short-term is that even the "poor" mereological top of this structure as introduced here is very useful to support discussions between human developers, not in the least to make the scope of their developments explicit.

5.2.1 The importance of task ontology standardization

Sooner or later, the **stakeholders** in the **robotics domain** must **agree on a *standardized* ontology** for all these terms, because that is the **only** way to realise a vendor-neutral digital plaform that can serve as the basis for composable innovation.[3] The second added value of creating a widely supported domain ontology with a *hypernym–hyponym hierarchical structure* is for the robot controller software to exploit: **only** when the mentioned ontological information is available, at **runtime** and in **formal representations**, one can expect robots **to reason** about their actions on the "most appropriate" level of abstraction, **to assess** whether what they are doing corresponds to what they are supposed to do in their current **task**, and **to adapt** their action plan accordingly.

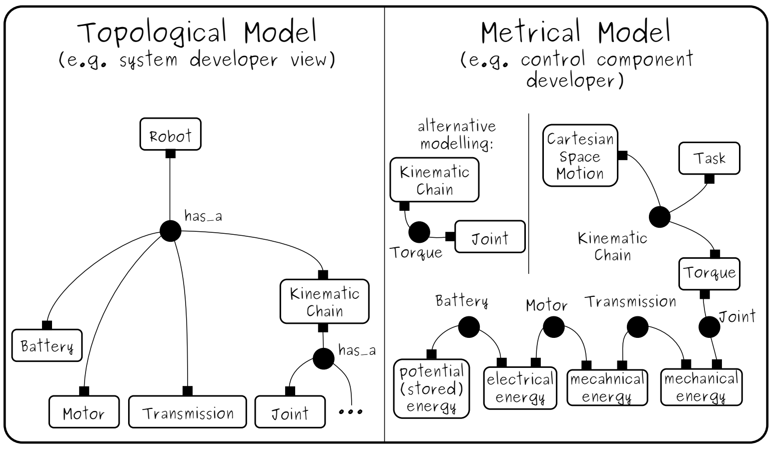


Figure 5.2: The graphs representing the *mereological* model of a kinematic chain, with the **natural** (since physical) entities of battery, motors, transmissions, kinematic chain, joints and links, and the energy-transforming relations between them.

5.2.2 Task as composition of natural and artificial constraints

The models of the instantaneous kinematics and dynamics behaviour of kinematic chains (Fig. 5.2) play an essential role in all robotic tasks, as the **natural** relations and constraints that must be satisfied in order to realise physically consistent robot motions. A similar role is played by the natural dynamics of actuators. Even together, these natural entities, relations and constraints are, however, not a *sufficient* condition to have a well-defined **task** for the robot. Indeed, the robot can move in an infinite number of ways, and still satisfy the laws of physics. So, it is the role of the `task` model to compose of the kinematic chain's natural motion properties with other **artificial** types of behavioural relations and constraints (Fig. 5.3), with

[3]After more than half a century of robotics industry, there are still no significant results in the direction of semantic standardization of the field.

the purpose of letting the robot actions realise the task requirements in a **predictable** way. "Predictable" is not the same as "unique": as is clear in the context of traffic, there is seldom a need for a task to impose one particular **motion trajectory**, but rather to allow a **motion tube** in time and space that the robot controller should move in.

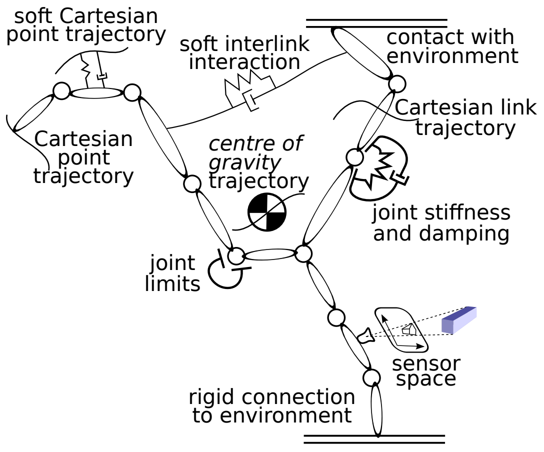

soft Cartesian point trajectory
soft interlink interaction
contact with environment
Cartesian point trajectory
centre of gravity trajectory
Cartesian link trajectory
joint stiffness and damping
joint limits
sensor space
rigid connection to environment

Figure 5.3: Various types of **artificial** motion constraints between (some of) the natural entities in Fig. 5.2. The figure uses a sketch of an "arm" robot, but similar constraint types apply to "mobile" robots too.

5.2.3 Task meta model realises the "dependency inversion principle"

Typically, it is only in the **context** provided by a **composite** task that these artificial relations and constraints are added. They couple parameters in the models of the individual tasks' **control** and **perception** (discrete as well as continuous), and of the **world models** that they share, Fig. 2.5, to extra world models, or to information required by *Configuration* and *Coordination* functionalities, or to solvers and monitors (Fig. 5.4) that compute and assess the robot's instantaneous motions from the artificial motion constraints of the task(s), Fig. 5.3. The *coupling* of the motion capabilities of the kinematics chain, joints and actuators, to objects in the environment, is realised by adding *model attachment points* to the kinematic chain models: in the latter model, one just has to foresee that "something" can be coupled in, at any later time, and of any particular type. So, none of the kinematic chain model parameters should *depend* in any particular way on the coupled entities, and no information about the coupling relations or constraints should end up in models of kinematic chains. This design approach is an example of the best practice of the "dependency inversion principle".

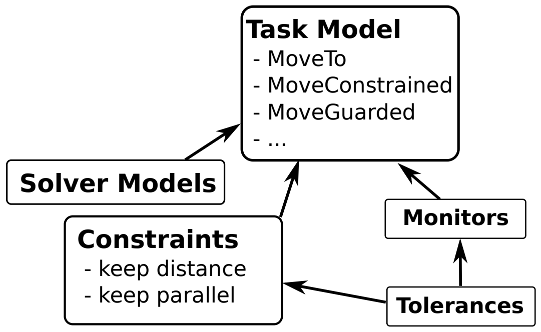


Figure 5.4: Overview of the models involved in the composition of a motion task. (The arrows represent `is-part-of` relations, i.e., inverse `has-a` relations.)

5.3 Task examples: robotic indoor and outdoor driving

The traffic system, or "driving" in general, is used throughout the Chapter as a familiar example of a formal representation of motions (i.e., driving in indoor corridors and rooms (Fig. 5.5), or in outdoor traffic lanes and parking lots) and of the perception required during those motions (i.e., recognizing building features or traffic signs, localizing them in their spatial context, and infering their influence on the robot's current action). Traffic lanes and signs are semantic tags in the world, that influence (i.e., constrain, as well as optimize) the driving behaviour of AGVs, cars, bikes and pedestrians, individual as well as groups. They model "motion" in a fully **declarative** way, because they do not model *how* a traffic user should move, but rather which relations the actual motion should satisfy. The real-world implementations of traffic signs are *designed to be perceivable* by human drivers in (almost) all weather conditions, and the areas they cover in the world are *designed* to fit to the *control bandwidths* that can be safely expected from (almost) all actors that take part in the traffic. Together, a traffic layout is an *architecture* of semantic traffic primitives in the world that (almost) guarantees that *humanly controlled systems* can drive safely and efficiently.

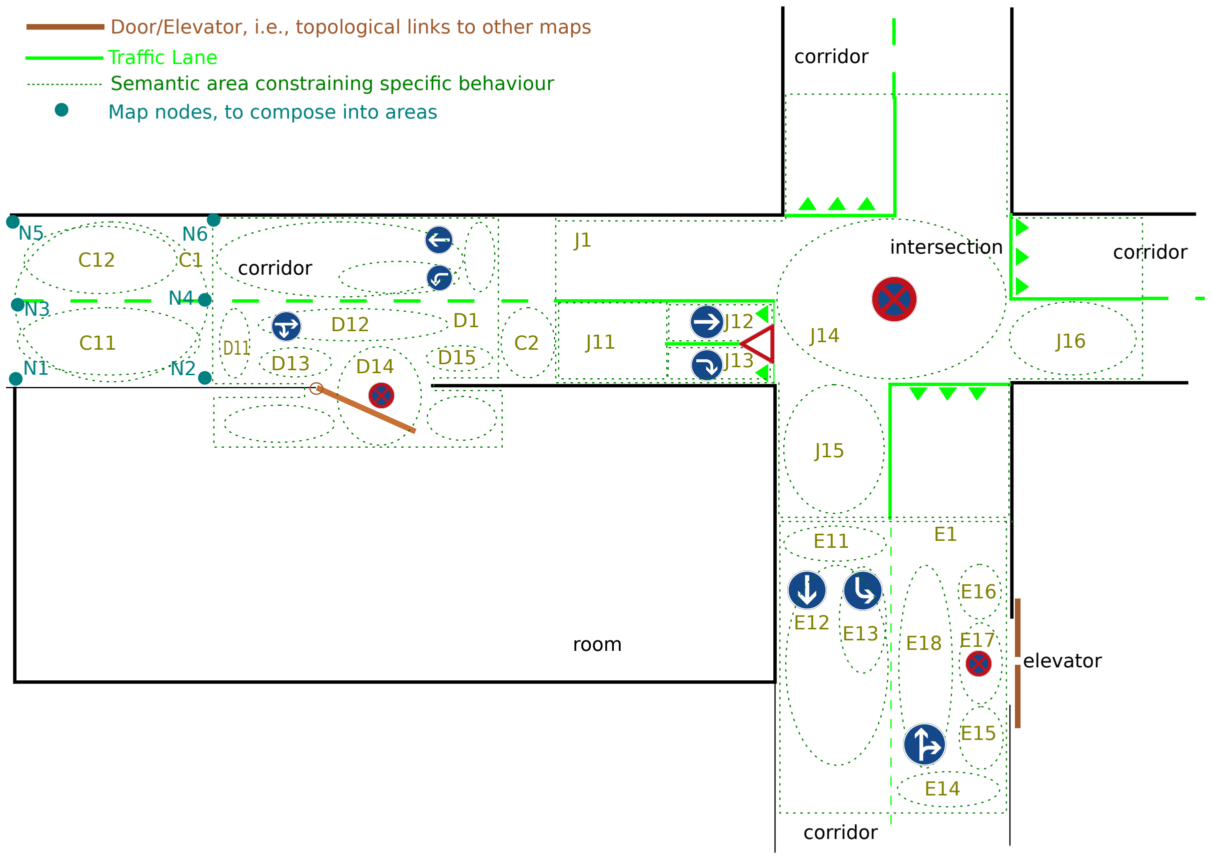


Figure 5.5: *World model* of an indoor "two-corridor-with-intersection" area: the solid black lines represent walls, and the red lines represent doors, while all other map entities are the **semantic tags** of the traffic signs and markings. These tags model the *motion constraints* that every robot must respect when it drives through the area, but they do not give information about how exactly the robot should move its wheels.

The main purpose of this Section is to explain how *engineered systems* can make use of the rather abstract task meta model of Fig. 5.1 by concrete of examples of motion and perception models for a mobile robot driving around in the "traffic" of an office-like indoor environment. The first step in that direction is to add the **geometric** level of abstraction to the mereo-topological level of abstraction in the earlier task meta model. More in particular, the world model will now be filled with (models of) the shape of the environment in the neighbourhood of the robot, the location and shape of the traffic areas, as well as the shape and motion of the robot's **kinematic chain** itself. In addition, the world model gets relations that link some of its geometric primitives to perception capabilities that are present in the robot control system; more in particular, there are some "walls" in the environment of the robot that its laser scanner sensor processing algorithm can detect and track, so that the position of the robot in the world model can be updated when new sensor information comes in.

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

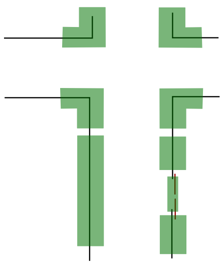


Figure 5.6: Semantic world model, featuring perception tags, for a laser range finder with sensor processing capabilities that can detect straight wall segments and rectangular corridor corners.

5.3.1 Geometric world models for control & perception integration

Figure 5.5 depicts a **world model**, with information at the *geometric level of abstraction*: it has **geometric primitives** such as **points** and planar **polygons**, and it has **semantic tags** (each attached to a geometric primitive) to model **landmarks** (i.e., *task-relevant* places in the world) that have **features** (i.e., *task-relevant* properties of a landmark used in `motion` and `perception` models). The Figure sketches an indoor area of two intersecting corridors, with doors to rooms and elevators. These doors and walls form **geometric constraints** for any `motion` that robots execute in the modelled world, since they have to steer clear from collisions with these "hard" world landmarks. Some (possibly different) landmarks also serve the robots' `perception`; for example, Fig. 5.6 represents the **perception** tags for a simple laser range finder type of sensor.

The next step after the modelling of the world, is the modelling of the task **plan**. The simplest form of such a plan is a *finite state machine*, with in each state a choice of a model for the *control*, the *perception* and the *monitoring* that the robot is expected to realise in that state; the monitoring provides the *events* to trigger a state change in the *plan*.

Figure 5.7 sketches how to use world model landmarks to attach some essential tags used in a **plan** model: a *"tube"* is connected to some tags in the traffic model, to indicate the area within which the *control* must keep the robot, while its *perception* measures whether it is making "good enough" progress towards some other traffic tags; various *monitors* will

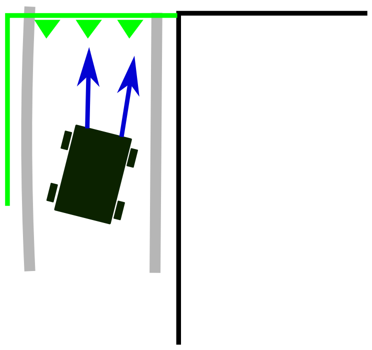


Figure 5.7: Semantic world model, extended with a motion **control** specification, in the form of a *tube* (the shaded gray area that represents the constraints of the control) and *motion drivers* (the blue arrows that represent (artificial) forces that generate the **instantaneous motion** of the robot).

follow the approach to one or more of these target tags, and signal the *plan* when the robot has "reached" one of them. More concretely, the robot should (i) not drive into the natural constraint of the wall but follow it, (ii) satisfy the artificial constraint of the traffic lane, *and* (iii) be ready to stop in time in front of the intersection.

The **control** model can be as simple as the two blue arrows in Fig. 5.7, that represent two driving forces attached to the kinematic chain of the robot, and whose direction is determined by some of the target tags in the world model. These artifical forces are the inputs to a motion controller, that transforms them to actual actuating torques on the motors. The control algorithm in itself is simple and constant, and all of the time and context dependent model information is embedded in the world model.

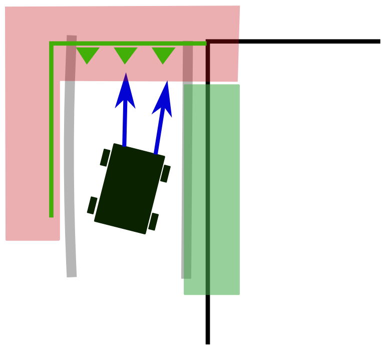


Figure 5.8: Semantic world model of Fig. 5.7, extended with the **local perception tags**: the green area focuses the perception of tracking a "wall" feature, at a resolution high enough to support the motion control; the red area focuses the monitoring on the detection of "any" feature in the direction of motion to react to, at a resolution low enough to react in time.

The same approach holds for the **perception**: Fig. 5.8 extends the world model with **semantic tags** for the `perception` part in the robot task model. Based on that information in the world model, the robot controller can *focus* its perception on just two areas: (i) the wall to be followed in the motion, and (ii) the nearby "rest" of the environment in the direction of motion, to be monitored for the presence of "obstacles". Detecting them signals the switch to another part of the `plan`. An important added value of the represented task *knowledge* is that all the sensor data that comes from beyond this local horizon need not be processed, since it does not have an impact on the currently executed parts of the `plan`. In resource-constrained applications such as robotics, it is indeed as important to know what *not* to spend effort on as it is to know what *must* be done.

A more advanced task plan can include a **preview control** model, as depicted in Fig. 5.9: one can compose two or more sequential sub-tasks "to look ahead" to the next area on the map that the robot has to drive through, and to provide more extensive natural and artificial constraints that come with this extended context. The composite task plan is again a *"tube"* as in Fig. 5.8, but now one with a bend around the next expected corner. Again, nothing

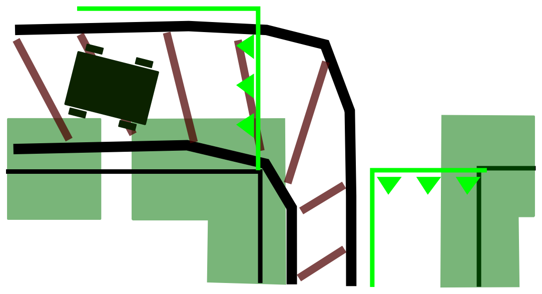


Figure 5.9: Semantic world model, with a task horizon that covers the sequential composition of two or more sub-tasks.

changes in the perception, control and monitoring functionalities, since all extensions are added to the world model, and to the plan.

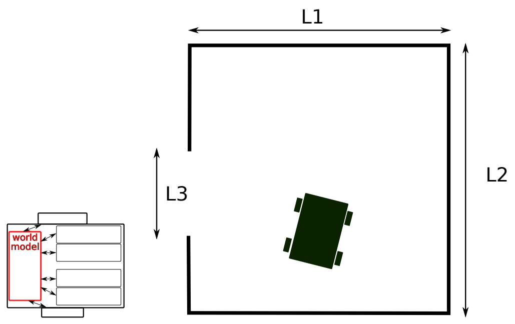


Figure 5.10: The `task` of the robot is to drive out of a rectangular room with a hole in one of its walls. The size and position of room and hole are unknown. The small figure on the left refers to the generic task model of Fig. 2.4, and indicates its parts that are involved; in this case, only the *world* is being modelled.

5.3.2 Example task plan: escape from a room

This Section provides the next step in the concrete design of the different `task` parts of the previous Section, for the **capability** shown in Fig. 5.10: *escape from a room*. One of the simplest possible versions of a **plan** has the following nominal *sequence* of **states**:

1. initialize sensors and motors, without motion control, perception or monitoring;
2. move forward till wall is detected, with the simplest possible control and monitoring;
3. move while *following* wall *on the right*, with somewhat more extensive control and monitoring, and with wall detection perception.
4. turn right at first large enough hole, with extra hole detection perception and monitoring.
5. stop.

The sequence above only represents the *nominal* plan, that is, it is not ready to cope with an execution of the robot's actions that would bring it in other states than the mentioned sequence. A **robust** plan, i.e., one that can cope also with non-nominal executions, must have monitors in all of its states, to check whether the sensor measurements still satisfy the assumptions that hold in each plan staet, within a task-specific tolerance. Typically, a robust plan requires an order of magnitude more design efforts than a nominal one.

The **resources** available to realise the task are assumed to be:

- *laser range finder*: it provides at regular intervals in time an array of rays, regularly spaced in a range of orientations, and indicating the free space within a minimum and maximum range of distances.
- *encoders*: they provide the change of the robot's wheel rotations over time, and hence an estimate of the instantaneous velocity of the robot.
- *velocity control*: it tries to realise the instantaneously specified desired velocity of the platform, via control of the corresponding wheel velocities.
- *effort value*: one scalar that represents the percentage of "full" available power used for the current motion.
- *keyboard button*: events from the keyboard of the human operator.

At **initialization**, the following knowledge is **assumed** to correspond to the real environment of the robot:

- the robot is *inside* a room.
- the room has a *rectangular* shape as in the figure, with unknown lengths of the walls.
- the room has *one door*, *wide enough* to let the robot pass through.

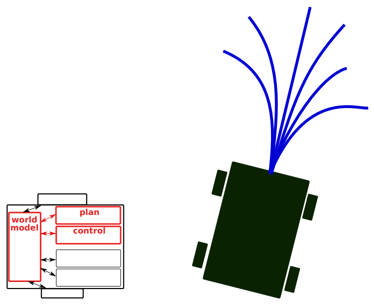


Figure 5.11: A possible *control* strategy for the escape-from-room capability in Fig. 5.12: to select from between a discrete set of pre-computed open loop trajectories, each corresponding to one particular time-invariant imput at the actuators.

The **control** part of the task can as simple as making a selection between various "open loop" motion trajectories, Fig. 5.11:

- when a set of constant speeds is applied to each wheel, the result is a set of known trajectories of the robot in the near future. These trajectories can be obtained from a model only, or can be identified on the real robot.
- the sparsity and density of these trajectories can be chosen, in a plan-directed way, to reduce the computational requirements to a level that does not allow to separate between trajectories that are closer together than the sensing resolution, or than the tolerance allowed in the task specification.
- time and space horizons can be chosen for the trajectories, again in a plan-directed way.
- the *control* action can then be as simple as *selecting* the best open loop trajectory and apply the corresponding wheel constant velocities.

The **perception** part does not need all the data provided by the distance sensor, since it could use the following algorithm (Fig. 5.12):

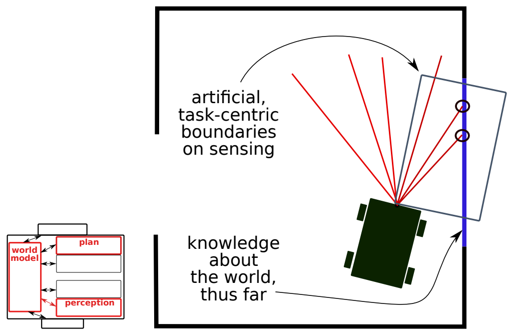

artificial, task-centric boundaries on sensing
knowledge about the world, thus far
world model
plan
perception

Figure 5.12: A possible *perception* strategy for the escape-from-room capability in Fig. 5.12.

- select a *region of interest* (the grey box in Fig. 5.12) that fits to the *plan*, because the latter is only interested in the right-hand side of the robot.
- fit a *line* through a *large enough* cluster of measurement.
- do this over a *time window* of measurements.

In other words, perception is done by means of the least-squares fitting of a line of limited length, through a clever, plan-directed selection of current and previous hits of the scanner rays with obstacles.

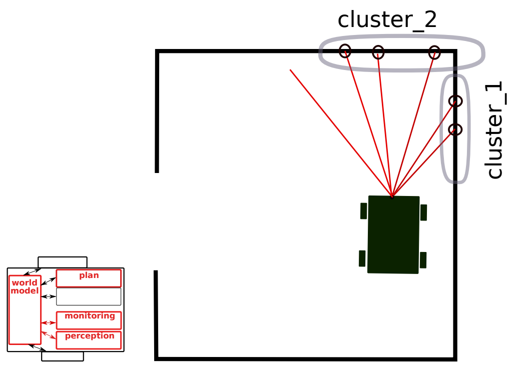


Figure 5.13: The *monitoring* strategy for the escape-from-room capability in Fig. 5.12 must follow the *wall on the right* (which is needed for the motion control) and look out for a *wall in front*.

The **monitoring** deals with finding which of the following four *hypotheses* gets most support from the sensor data:

1. one can fit a *wall* on the *right*, by looking only at a *local* horizon of measurements , as expected by the *task* context;
2. a further horizon in the *forward* direction is needed:
   (a) to monitor whether there is “something”, to react to in the *plan*;
   (b) to find another *line cluster*, orthogonal to the first one, to update the *world model* with a new *corner*.
3. the leftmost rays can be discarded, because they are outside of the scope of the *plan*, which reduces the computational load.
4. *all* measurements *could* be *neglected* until needed again, based on the *planned* speed of the motion.

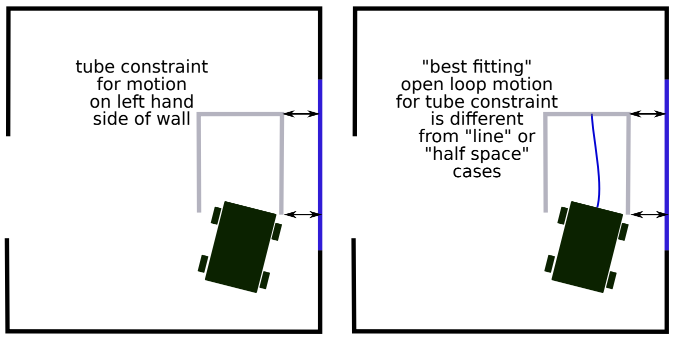


Figure 5.14: Left: a possible *motion specification*, in which a "tube" constrains the allowed motions, but does not command an explicit motion trajectory. Right: a corresponding *control* choice.

The **motion specification** needed in the **control** can be as simple as the "tubes" in Fig. 5.14:

- the robot is allowed to move anywhere inside a **tube** at some distance from the wall.
- it must make progress towards the next **waypoint** which is at the closed end of the tube.

The controller then selects one of the open loop trajectories of Fig. 5.11 that fits best, according to a task-specific metric. **Alternatives** for the **control** design are depicted in Fig. 5.15.

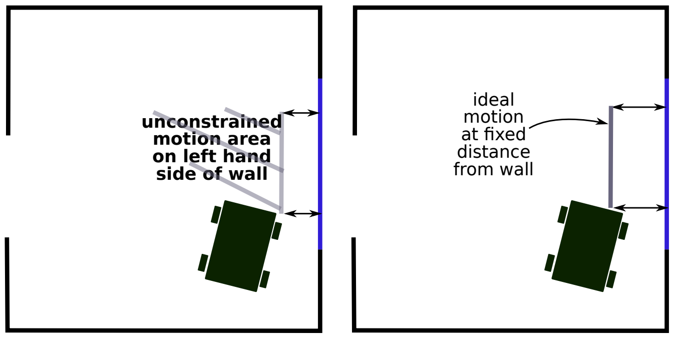


Figure 5.15: Two alternative controller approaches for the motion specification in Fig. 5.14. Left: a open half space, to the left of the wall. Right: a line trajectory at a speficied distance from the wall.

The **world model** entities and relations needed in the **plan** are as follows:

- `room` has four `corners`, one `door`, and five `walls`.
- each `wall` is represented by two `nodes`, that is, a point in the 2D plane.
- the `robot` has a `pose` with respect to the `room` features.

This gives rise to the topology of Fig. 5.16. The geometrical properties are rather straightforward:

- every `node` gets two coordinate numbers, being its position in the room's `frame1`.
- every `corner` gets a property tag representing that it is a straight angle.
- the position of the `robot` is given by the coordinates of its local frame in the room frame.

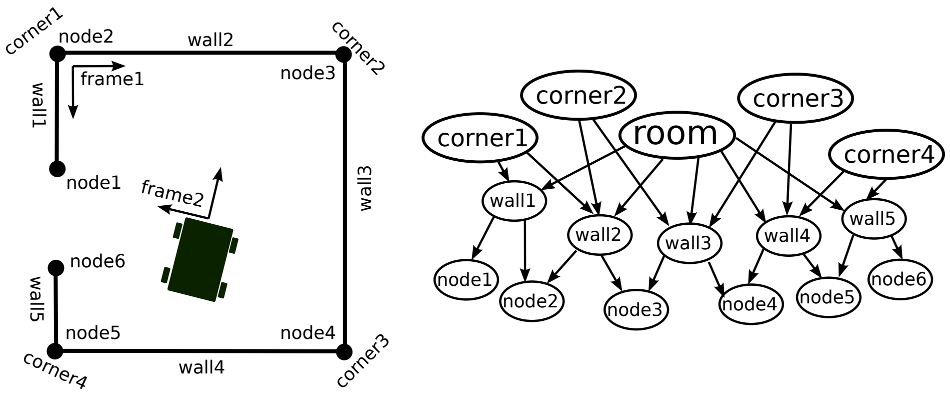


Figure 5.16: Right: topology of *room* model entities ("data structures") and relations (`has-a`, and `connects`). Left: geometrical properties of all entities. The robot's pose in the room can be represented numerically by position coordinates of the frame attached to the robot with respect to the frame attached to the room.

5.4 Policy: event loops for task execution

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

5.5 Mechanism: `motion` specification as a constrained optimization

A *composable* way to formulate a task specification is by means of a *constrained optimisation problem* that is formulates by the following collection of entities, relations and constraints, e.g. [4, 6, 16, 31, 39, 40, 48]:

- **configuration space**, of all the parameters in the models of the hierarchy in Sec. 4.8, e.g., the joint space parameters of a robot and its actuators, q, and Cartesian space parameters X;
- **desired configuration** (X_d, q_d), which are the sub-sets of the whole joint and Cartesian configuration spaces that the execution of the task should have as its outcome.
- **objective function**(s), that is, relations $f(X, X_d, q)$ on these parameters that the task execution is expected to minimise, e.g., the delay in "progress" of the task execution, the energy consumption of the robot, the distance to obstacles, or the closeness to a target pose;
- **constraints**, that is, equality relations $g(X, q) = 0$ and/or inequality relations $h(X, q) \leq 0$, on the configuration space parameters, that must be satisfied during the execution of the task, e.g., joint limits, or singular configurations in a kinematic chain.
- **tolerances** $d(X, X_d, q, q_d) \leq A$ describe how well the constraints have to be satisfied, or how far the objective functions have to be optimized.

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

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

Representing a Task in this way gives a **declarative** specification, that is, it is a model (generated off-line or online) that expresses the logic of the (desired) task execution without describing its control flow. One then needs a solver to generate (at runtime, taking the lastest sensor information into account) the **imperative** (or, "procedural") flow of control actions (actual setpoints in joint position, velocities, accelerations or torques to send to the actuators) or plans (more detailed declarative task models, with a more focused scope in time and space), required to realise the task. The above holds both for control-based approaches [2, 36] (which are online, reactive but they may suffer of local minima problems) and for plan-based solvers [31] (which are (typically but not necessarily) offline, and less reactive since they explore a bigger search space).

The term "Task" was used in the paragraphs above as a container term for each of its parts: plan, control, monitoring, perception, capabilities and resources. Often, the configuration spaces of two or more of them are taken together. A common example appears when specifying **active perception** tasks: the plan contains motions that have as sole purpose to improve the perception and monitoring aspects of a task. This is what humans do, often unconsciously, for example when double checking their location in an environment, they focus their attention to a series of important landmark, in a particular order and with a frequency of revisiting the relevant landmarks, that depends on the tolerances required for that localisation.

5.5.1 Policy: specify as objective function or as constraint

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

- functions like *time*, *energy consumption*, or *safety*, are *derived* quantities, whose values can not directly and uniquely be influenced by the actuator signals q. Hence, it's often better to select them as inequality constraints, that monitors must follow during the task execution to allow the plan to switch to another control approach when the realised time, energy, etc., falls outside of the prescribed boundaries.

- motion and effort variables are more directly influenced by the actuators, hence it is typically easier to use objective functions that combine one or more of such variables. For example: deviations from geometric paths; or predicted violations of tubular regions after a certain time horizon in the future.

- while it is mathematically easy to specify a *multi-objective optimization*, via a weighted combination of various objective functions, this *requires* a choice of weighing factors, but that is often impossible to derive from the task context in a unique or deterministic way. Hence, the weighing factors often remain very arbitrary, and not motivated by knowledge insights into the task challenges.

The resulting **best practice** is to choose **one single objective function** per state in the Task plan, and to foresee other control states to switch to whenever one or more of the other monitored (but not optimized) functions exceed their expected ranges.

5.5.2 Policy: integrating mutliple levels of abstraction in one task

Most real-world applications must combine several levels of abstraction (Sec. 5.7.3) in their control and the above-mentioned best practice is then applied in this multi-level control context in the following way: a "higher" control level influences a "lower" level (and vice versa) preferably via the addition of constraints and tolerances, but *not* via extra terms in the objective function.

For example, an electrical motor influences the mechanical joint motion control via motor heating and energy efficiency constraints; and that mechanical joint motor control influences the kinematic chain motor control via position or torque limits. The objective functions to be optimized for the motors and for the kinematic chain can be designed independently of these constraints, and it is only by the *solution* of the whole constrained optimization problem that the integration takes place. That is, the *monitoring* of the constraints gives rise to switches between several optimization problems to be solved. This approach yields a very **composable** way of sub-system integration, and the overall system behaviour emerges from the individual components' behaviours with high predictability, except for (i) the exact time on which the controller will react, and (ii) the exact sequence of controller states that the system will evolve through.

5.5.3 Policy: monitoring for hybrid constrained optimization

In general, task models require **monitoring** functionalities to determine, at runtime, whether the execution of the task specification is progressing correctly, that is, whether *all* the relations and constraints in the composed task model are satisfied within tolerance. Such online monitoring allows to react to modeled conditions, both *desired* (i.e., the task execution obtained the desired outcome) and *undesired* (e.g., foreseen cases of non-nominal execution), and then to modify the behaviour of the robot. At the same time, the introduction of monitoring functionalties makes the Task specification into a **hybrid constrained optimization** problem, since the *plan* now needs a finite state machine to switch between different constrained optimizations, when reacting to the monitoring events.

The **moveGuarded** example in Fig. 5.17, applicable to a six degrees of freedom serial robot arm equiped with a force sensor, is probably the simplest model of a task specification that is composed with an online monitoring specification; the example specifies a *nominal* termination condition, which fires a *task accomplished successfully* event as soon as the condition is met.

However, the task specification in Figure 5.17 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,

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

Figure 5.17: Example of a guarded-motion task definition [8].

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]). A composable robotic modelling approach requires a task specification modelling language that includes all kinds of monitoring, and this inevitably leads to a **graph of relations** around the nominal specification model. Some initial modelling efforts can be found in [48].

5.5.4 Policy: iterating feasible solutions during task execution

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

(TODO: examples.)

5.6 Semantic task specification: domain-specific languages

The following Sections give examples of *domain-specific languages* that bring the specification of tasks (as introduced in the previous Sections) in line with the terminology used in particular applcation or technology domains. The provided examples are far from standardized, yet.

5.6.1 Semantic mobile robot motion primitives

Abstracting a bit from the concrete examples in the previous Section, the following semantic motions could form the basis of a mobile robot's "**platform** motion stack capabilities":

- *start-to-cruise* (and its *inverse*, *cruise-to-stop*): how to get the robot start its motion and reach a "cruising" motion behaviour, when all it has to do is to drive "straight ahead" within its current lane, and that lane continues till "far" beyond the dynamic bandwidth of the robot.

- *cruise through tubular area*: the *world model* for this semantic motion has landmarks on the robot are geometrically constrained by a "tubular" area in the Cartesian space.

- *overtake obstable during cruise*: this is a composite cruising task, with lange-changing motion capabilities added. Reference [22] contains already a very worked-out formalization of this semantic motion primitive, at a mereo-topological level of abstraction that fits to that of this Chapter.
- *cruise to approach*: this is an extended version of the *cruise-to-stop* primitive, in that the "approach target" semantic tag adds extra constraints on the motion behaviour, such as optimal/expected relative motion positions and orientations, and relative motion speed profiles.
- *approach to stop*: similarly, this semantic primitive adds extra stopping behaviour, determined by properties of the approached target.
- *approach to turn right/left in tubular area*: this primitive adds extra behaviour of how to connect two *cruise through tubular area* motions, one before and one after a "crossing". The tubular area in the *world model* has specific landmarks that guide the robot to make a right (or left) turn. Examples are: traffic lane indicators on the ground; or "walls" in the built environment as well as in the natural environment (trees, bush, river,...).

5.6.2 Semantic robot arm motion primitives

The mobile robot example in the previous Sections is conceptually the easiest to grasp, since the world is mostly flat, *and* the shape of the robot is mostly constant. For arm-based robotic systems, the full 3D Cartesian space and the full nD joint space are to be taken into account, but at the mereo-topological level of abstraction, very similar sematic motion primitives can be defined. Figure 5.18 sketches some simple examples, with two levels of resolution in representing the mentioned configuration spaces.

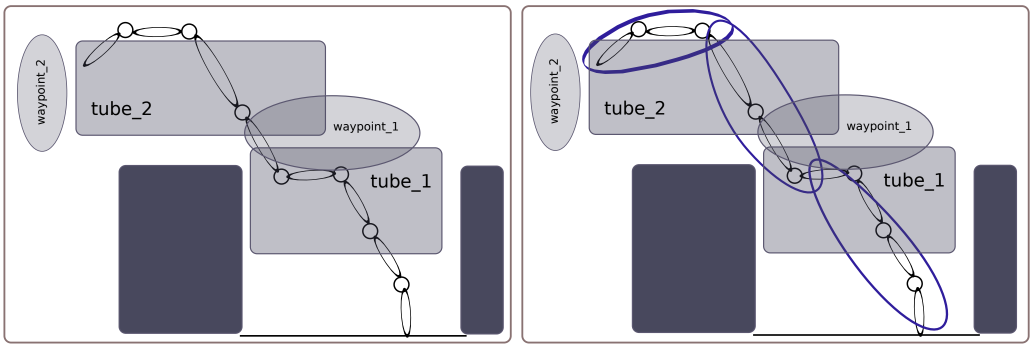


Figure 5.18: Model of a "high" (left) and a "low" (right, in blue) kinematic resolution motion plan for a serial robot arm during a sequence of tubular motion between obstacles.

5.7 Natural hierarchies in task models

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

5.7.1 Hierarchy in world model scope

This Section elaborates on the **robot-centric** part of composite task models, that is, those in which the **motion capabilities** of **only the kinematic chain** play the central role. (Later Sections add the possibility to extend these kinematic chain motion tasks with models of other robot action parts, e.g., perception actions to monitor the motion execution, or to update the world model information.) Robot motion capabilities grow in complexity according to the **complexity of the plans** the robots have to execute,[4] and another very useful **hierarchical structure** is represented by the following semantic tags:

- **move**: this represents the plans that can be realised by only the robot's **instantaneous hybrid dynamics** (Sec. 4.4). In other words, the kinematic chain model of the robot *is* the world model, and its behaviour is that of an ideal mechanical energy transforming system.

  For example, the instantaneous motion under the influence of a pushing force at the end-effector of the kinematic chain, or the instantaneous open loop motion under the influence of torques applied at the joints, possibly together with instantaneous (artificial) acceleration constraints.

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

  Two typical use cases exist: (i) the motion makes the robot reach the planned position in space, as far as this can be interpreted by the robot's own sensors, or (ii) the motion stops when a contact transition is detected via using current, force, tactile or IMU sensors mounted at various attachment points on the kinematic chain. Hence, *world model*, *perception* and *object affordances* models are restricted to what is directly attached to the robot's kinematic chain.

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

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

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

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

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

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

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

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

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

  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 but only perception.

  For example: human-aware robotic manufacturing cells, where the reactions of the robots to the presence of humans in their neighbourhood are pre-programmed (or, better, modelled), but the coordination events inside and between robot control systems are to be generated by the latter control systems themselves.

The modelling suggestions above are only *mereological*, hence a lot more detailed models must be provided, topological, geometrical, dynamical, etc. Section 5.5 provide a start of these modelling efforts.

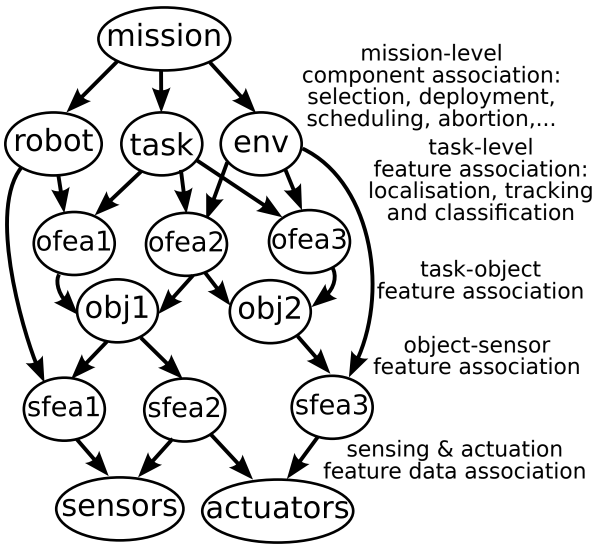

mission
robot
task
env
ofea1
ofea2
ofea3
obj1
obj2
sfea1
sfea2
sfea3
sensors
actuators
mission-level component association: selection, deployment, scheduling, abortion,...
task-level feature association: localisation, tracking and classification
task-object feature association
object-sensor feature association
sensing & actuation feature data association

Figure 5.19: The natural hierarchical dependencies between components in the generic perception graph for robotic systems.

5.7.2 Hierarchy in perception graph

(TODO: explain Fig. 5.19.)

5.7.3 Hierarchy in cascaded control levels

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

- **power inverter**, with typical time constant of 1/100.000th of a second.

- **electrical motor**: transforms electrical power into **mechanical torque**, via mechanisms such as field oriented control, with typical time constant of 1/10.000th of a second.
- **mechanical acceleration**: the generated torque results in acceleration of the attached mass; the time scales required in robotic applications lie around 1/1000th of a second and up.
- **mechanical velocity**: integrating acceleration results in velocity; again, an order of magnitude less in required time scale is order.
- **mechanical position**: further integration into position is the final step in the mechanical level of abstraction, with again an order of magnitude lower time scale.

The *battery* is not part of the cascade hierarchy because:

- its time scale is determined by the *chemical dynamics* of conversion of chemical energy into electrical energy, and this is orders of magnitude slower then the dynamics inside the electrical DC-to-AC energy conversion.
- its energy production need not follow the time scales of the control, since that is the responsibility of the DC-to-AC power inverter.

Chapter 6

Meta models for world modelling and its integration in tasks

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

Chapter 7

Meta models for control and its integration in tasks

Control realises task behaviour…

Control is one of the fundamental parts in a **task** model, responsible for realising the **continuous**-domain[1] behaviour of the system that brings the "world" from its *actual* state to its *desired* state as specified in the **plan**. This Section introduces the mereo-topological meta model of **control diagrams**, and some of the natural structures and "best practices" in the control domain.

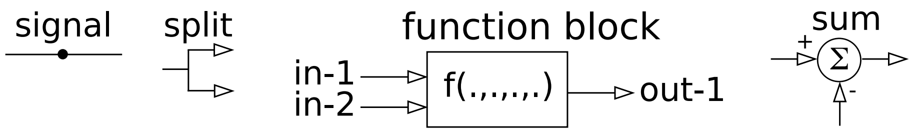


Figure 7.1: The four entities in the controller meta model that conform to the `algorithm` meta model: the `data` blocks `signal` and `split`, and the `function` blocks `function-block` and `sum` block.

7.1 Mereology of control behaviour: feedback, feedforward, predictive, adaptive and preview

Here are the five complementary sources of contributions in computing the actuator input signal:

- **feedback**: this is a function that converts the desired and actual "state of the world" into an actuator signal to reduce that "error", Fig. 2.6. Figure 7.2 shows the specific case that use the *error* as input parameters. Feedback functions typically are parameterized, in order to be configurable for different applications and domains; e.g. to adapt the `gains` and `errors` to the control problem at hand.

[1] "Space" or "effort" versus time, for example. **Discrete** control comes back in other Sections, like Sec. 2.14.

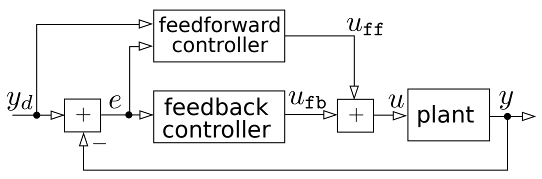


Figure 7.2: Feedback and feedforward control loops, with the *setpoint error* as the main input to the controller computations. This is a special, "local" case of the generic controller of Fig. 2.6, since it considers only the *instantaneous error* between desired and actual state value y.

- **feedforward**: this is a (typically parameterized) function that generates an actuator signal based on the current "state of the world" and on a **model of how the system is expected to react** to actuator signals, Fig. 2.6.

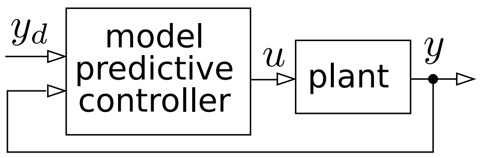


Figure 7.3: Predictive controller.

- **predictive**: the actuation signal is computed as the "best" value resulting from a simulation of how the plant behaviour is expected to be over a certain time horizon into the future, when the control inputs are varied over a specified domain, Fig. 7.3.

- **adaptation**: this is a (typically parameterized) function that converts the *observed history* of the control signal and plant state into an adaptation of one or more **parameters in the models** of the control functions, of feedback and/or feedforward. For example, it adapts the gain in the feedback control. So, it does not change the **structure** of the control behaviour models, only the **behaviour** itself.

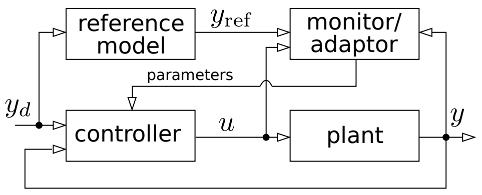


Figure 7.4: Model-reference adaptive controller.

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

7.2 Mechanism: state and system dynamics relations

Any controller (or **control diagram**) is the composition of only four entities (Fig. 7.1) that `conform-to` the algorithm meta meta model:

- `signal`: the **data** that flows between two `function` blocks; its numerical representation is a traditional *data structure*.
- signal `split`: special case of a "`signal`" that appears in quasi every control diagram, and that represents the fact that the *same* signal is used as inputs to more than one `function` block. The split has a *direction*: there is only one connection to an output of the `function` block that creates the `signal`, but the `signal` can be the input of more than one `function` block.
- `function` block: this is a *pure function*, with one or more `signals` as inputs and one or more `signals` as outputs.
- `sum` block: this is a special case of a `function` block, that appears in quasi every control diagram, and that gives as its output the sum of all its inputs, each possibly with a "-1" sign inversion.

The following `signals` are common to all controllers, so the meta model adds them as first-class modelling primitives:

- `setpoint`: the entry point of a controller, which is not connected at its input side, at least not inside the scope of the current control diagram. Its meaning in the context of an application is that this signal gives the *desired* value of the `plant` state.
- `measurement`: the other entry point of a `control-diagram`, which provides state information of the `plant`, possibly after some processing of the raw data from the sensors that are physically connected to the `plant`.
- `error`: the "difference" between `setpoint` and `measurement`, whose "magnitude" is a measure for the quality of the controller.[2]
- `actuation`: the exit point of a `control-diagram`, which provides information for the actuators that can change the `state` of the `plant`.
- `state`: a selection of signals somwhere in a control diagram, that make sense to the application that used the controller, when their values are taken together, at the same execution instant of the control diagram.
- `sample-instant`: the `signal` that represents the time of one particular execution of the `control-diagram`.
- `sample-period` (and its inverse, the `sample-frequency`): the `signal` that represents time between two subsequent executions of the `control-diagram`.
- `event`: a `signal` that represents the fact that "something" has happened during the execution of the `control-diagram`.

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

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

- `plant`: this is the part of the real world whose `state` the `control-diagram` tries to influence, and with which it interacts via `sensors` and `actuators` that convert physical values into digital `signals`.
- `feedback` loop: a `function` block that takes `setpoint` and `measurement signals` as inputs, and computes a `signal` that can be used by an `actuator`. The topology of `feedback` is a "loop" because the `plant` couples the `measurement` and the `actuation`.
- `feedforward` chain: a `function` block that takes `setpoint` signals as inputs, and uses a *mathematical model* of the `plant` dynamics to compute a `signal` that can be used by an `actuator`.
- `sensor`: a `function` block that gives `signal` values to some physical values of the `plant`.
- `actuator`: a `function` block that converts `signal` values in the controller to physical values of the `plant`.
- `adapter`: a `function` block that takes a `state` of the `control-diagram` as input, and computes a `signal` that another `function` block in the `control-diagram` can use to change the value of one or more of the parameters it uses in its computations.
- `monitor`: a `function` block that takes a `state` of the `control-diagram` as input, and computes an `event` for the application software that uses the controller. The event itself is not further used in the `control-diagram` itself, but it can give rise to a change in one or more parts in the `control-diagram`.
- `control-diagram`: this is the **composition** of all of the above, which conforms to the constraints of the controller meta model, discussed below. It has **one trigger** entry point, to execute all computations inside the diagram; to this end, every `control-diagram` contains a data structure to represent its `schedule`. In other words, the `control-diagram` is the model of the function that takes the `output state` of the `plant` as one of its arguments, and the *desired* `state` of the `plant` as its `setpoint` argument, and computes the `input` that should be applied to the `plant` to make it evolve towards the desired `state`. In general, a `control-diagram` model is a **cyclic graph**, because of the presence of feedback loops.
- `delay`, or `buffer`:

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

Properties of the execution are:

- `latency`:
- `jitter`:
- `loop-time`:

Not all **compositions** of entities in the meta model result in meaningful `control-diagrams`, since the following **constraints** must be satisfied:

- **feedback loop constraint**. The following chain *must* be present: `setpoint`, `feedback`, `actuation`, `plant`, `measurement`.

- **feedforward chain constraint**. The following chain *must* be present: `setpoint`, `feedforward`, `actuation`.

- **cascaded feedback loops constraint**. More than one `feedback` loop can be present in the same `control-diagram`, and the proper composition of an “inner” and an “outer” loop requires that the `setpoint` for the “inner” loop is an `actuation` signal of the “outer” loop.

- **adapter chain constraint**. Any `adapter` has at least one `state` as its input, and its output goes into a `feedback`, `feedforward` or `monitor` block.

- **monitor chain constraint**. Any **monitor** has at least one `state` as its input, and has no output that goes into a `feedback`, `feedforward` or `adapter` block.

7.3 Policy: model-reference adaptive control (MRAC)

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

7.4 Policy: model-predictive control (MPC)

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

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

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

- **setpoint**: only one single **instantaneous value** of the **desired** `state` of the plant is being used in the control computations. In other words, the **control horizon** is only one time instant “deep”.

- **trajectory**: instead of just an instantaneous value of the `plant state`, a trajectory of desired `plant state` values at multiple sample times over a certain horizon is used in the control computations. This mechanism brings more *constraints* than one single setpoint; the latter is a boundary case of the former.

- **path**: another mechanism is to use a **path** instead of a trajectory, which is less constraining since the **time is not imposed**, i.e., the `state` is *constrained* to follow the geometry of the path in state space, but not any timing along that path.

- **tube**: this is an even less constrained control specification, in that the controller is expected to keep the plant state inside a "tube", or "region", in the `state` space, and no extra constraints are attached to the concrete trajectory that the controller generates within the tube boundaries.

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

7.6 Policy: control progress objective

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

7.7 Policy: generic PID, sliding mode and ABAG

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

- only a **setpoint** error is used in the **feedback**, with **PID** control as typical representative.

- an **area** error is used to **select** the control **feedforward**, with **sliding mode** as typical representative.

  The control designer has, at design time, determined a number of areas in which the plant error can be, and for which a different feedback-feedforward control action is prepared off-line, Fig. 7.5.

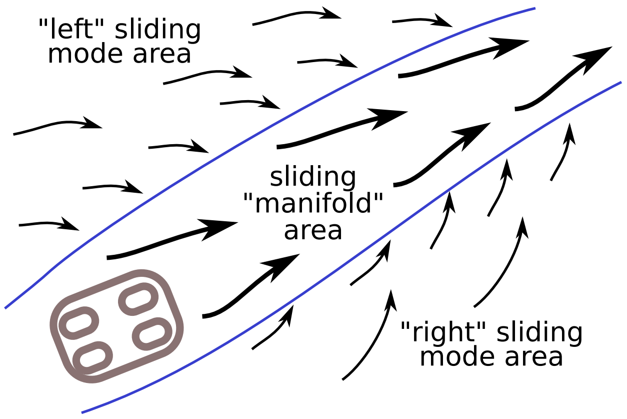


Figure 7.5: The concept of *sliding mode control*: the state space of the robot is divided into areas, each with a particular *a priori* determined control approach. In the strictest version of the concept, the middle area is the specified *trajectory*.

- the **trend** in the error is used to **adapt** the control **feedforward**, with **ABAG** control [19] as a representative.

  The control computes a **bias** (the "B" in "ABAG") that represents the control signal needed to keep the plant in its current state, and that follows the trend in the error at "slow" speed, while there is also a **gain** (the "G" in "ABAG") to react "fast" also to the direction of the error. (The "B"s in "ABAG" indicates that both parts are **adaptive**.)

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

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

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

The traditional policy for a PID controller is to be used as *one single algorithm*, irrespective of the error. The traditional policy for a sliding mode controller is to be used as a *hybdrid algorithm*, in the sense that it identifies different areas in the error state space and selects an algorithm on that basis. The traditional policy in the ABAG controller is to allow *adaptation* of its B and G parameters. Of course, there is no fundamental reason why these different policies can not be used all together. In the context of control for system-of-systems, the trend is obviously even stronger, since practice shows that **every controller must be hybrid and adaptive**:

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

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

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

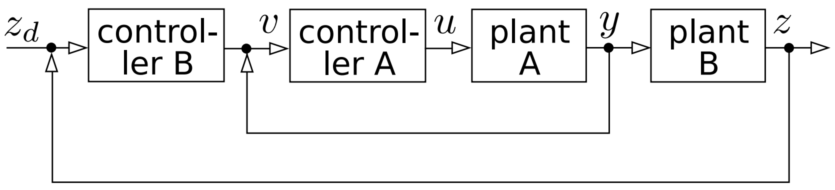


Figure 7.6: Cascaded control loops.

7.8 Mechanism: cascaded control loops

The natural hierarchy in the physical domains that are relevant in robot system control, and especially the differences in the natural time constants in these domains, leads to the **best practice** of **cascaded control loops**, Fig. 7.6: the innermost loop deals with the control of the fastest physical time scale (in particular, the DC-to-AC conversion), and the loops around it cover the next time constants in increasing order: torque, acceleration, velocity and position. In every loop, a new part of the plant dynamics must be taken into account; e.g., the inner loop sees the electrical dynamics of a motor, and the loop around it also sees the mechanical inertia.
(TODO: figures.)

7.9 Mechanism: asynchronous distributed control

The Sections above made the **assumption** of **synchronous control**:

- all computations can be done in *zero time*.
- the computations are executed at the *right time*, say in 1kHz loop.
- the *scheduling* of the execution of the computations is the same every time one computes the whole control loop.

This assumption does not hold anymore for many modern machines, like cars or robots, that have multiple fieldbusses inside, and many of the computations (e.g., sensor processing) must run on separate processes or even computers. In addition, demands are shifting towards **system-of-systems** applications, in which separate machines come together in temporary systems and must realise tasks together; for example:

- multiple **tugboats** maneuvering a tanker.
- multiple *cranes* moving same load.
- multiple *drones* transporting same load.
- getting cars into and out of a *platoon*.

So, such control loops involve *communication* between control computing processes, Fig. 7.7. Such a **distributed cascaded control** architecture introduces **asynchronicity** into the control problem:

- the closing of *feedback loops* is disturbed, because the latest state information is not available at the theoretically ideal time.

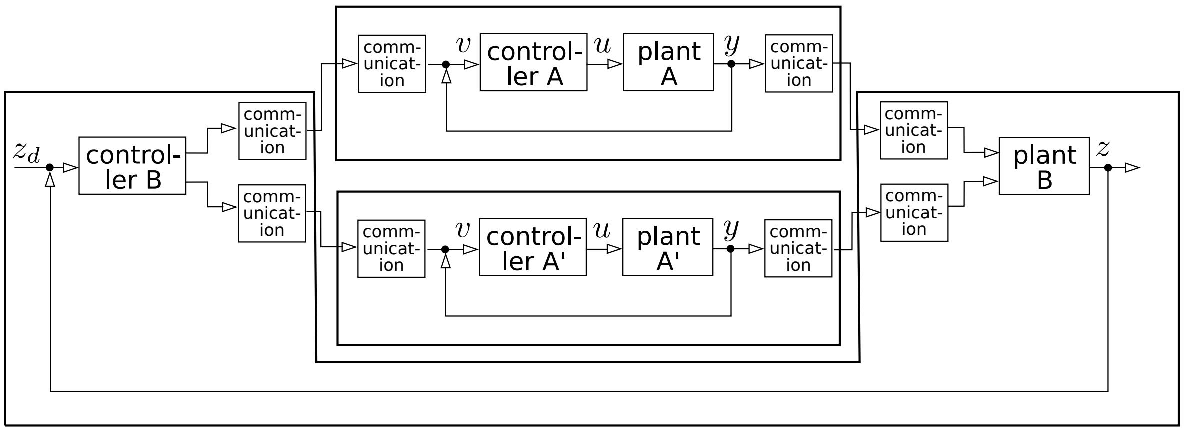


Figure 7.7: Many modern systems must rely on *communication* between sub-systems, in order to realise cascaded control loops.

- there is now a need for *monitoring*: each subsystem must monitor how well its own control responsibilities are progressing with respect to what the overall system expects from it. It must provide this information to the other system components, and in turn must use the similar progress quality information that it receives from the other components.

- the result is the need for *mediation*: each subsystem must have a (safe, effective) reaction to the situation where the control progress, of itself or of its peers, is "not good enough".

The result is that every distributed controller becomes an **hybrid event controller**:

- each sub-controller needs a *Finite State Machine*, with different control configuration in each state.

- all sub-controllers must also send *events* to each other, *to coordinate* their FSMs.

7.10 Policy: optimal control

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

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

The **domain** fills in the *types* for f, X, q for a particular "robot", and a particular *type* of solver. The **application** then adds choices for the *parameter values* for f, X,..., and the concrete solver and monitor *implementations*.

(TODO: concrete examples.)

7.11 Policy: behaviour tree for semi-optimal control

A behaviour tree is a mathematical model, with a limited but very composable number of entities and relations, to decide what next *action* to take in a control loop. (It is a more specific version of a decision tree, focused on "control".) It trades off optimality of the quality of the solution for speed of finding a feasible solution. The knowledge encoded in a behaviour tree model is typically known (i) to be "good enough" in particular use cases, and (ii) reflects experience in how to detect the (or rather, "a") relevant use case via a series of decision making conditions that are fast to compute.

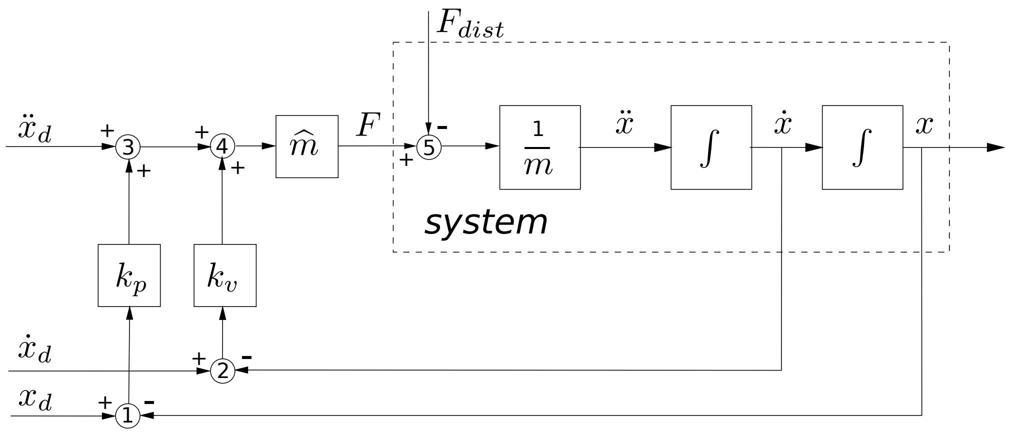


Figure 7.8: A one-dimensional position controller, generating the actuating force F from the desired position x_d, velocity $\dot{x}_d$ and acceleration $\ddot{x}_d$, via nested velocity and position feedback loops with gains k_v and k_p The "world model" of the controller consists of an estimate $\widehat{m}$ of the moved mass.

7.12 Event loops revisited: control behaviour composition

"Control" is an essential part of all robotics and cyber-physical systems, and the composition of the material introduced by all previous Sections now allows to model the meta model of controllers. In summary, a controller event loop (with a simple example depicted in Fig. 7.8) composes the task, algorithm, Finite State Machine, control diagram, and event loop meta models, and adds specific **policies** (i.e., model configurations):

- control diagrams as in Fig. 7.8 represent the dataflow model of an algorithm. The **structural model** is a **graph**, but typically unambiguous ways exist to find its **spanning tree** (typically *cutting* the diagram at the "summators" 1,. . . , 5 in Fig. 7.8), with equally unambiguous ways *to serialize* it into a **schedule**.

- dataflow buffers (i.e., the "arrows" in the control diagram) are often just "one deep", since the controller is only interested in the most recent version of measured data ("*Last Write Wins*"), such that older versions can be overwritten when new measurements arrive. However, modern controllers see an increasing use of *Model-Predictive Control* (MPC) or *Moving-Horizon Estimation* (MHE), which require data flow buffers of size $N > 1$.

- if necessary, pre-processing of measurement data takes place in the `prepare` step for each loop, and gives the result as "new measurement" to the control loop. Such pre-processing can consist of averaging operations, or curve fitting, or other types of **observers** or estimators, like the MPC or MHE approaches mentioned above.

- several nested (or cascaded) loops can exist (e.g., Fig. 7.8): the natural **causality hierarchy** is to schedule the computations of an inner loop more frequently that those of an outer loop.

- many variables computed in control loops must be **monitored**, and these computations must be integrated in the "right way" into the scheduling of all other control loop computations.

- similarly, some monitors do not only generate *events* to trigger *discrete changes* in the control loop configuration, but also *adaptation* of some *continuous parameters* in the controllers, such as feedback gains or model parameters.

- last but not least, **realtime** requirements of the application have an impact on the model of the event loop.

7.12.1 Example: one-dimensional position control

For example, the event loop of the one-dimensional position controller depicted in Fig. 7.8 specializes this generic pattern as described below. The *data blocks* are the arrows in the control diagram:

- *state* of the system $x(t)$, i.e. position of the mass m. The state changes continuously over time, but the controller only needs the most recent versions of the state measurements.

- *setpoint inputs* $x_d, \dot{x}_d, \ddot{x}_d$, e.g. desired position, velocity and acceleration of the mass.

- *measurement inputs* x and $\dot{x}$, e.g. actual position and velocity of the mass.
- *feedback gains* k_p, k_v, i.e. proportional position/velocity control gains computed, for example, via pole placement (off line) or an observer/adapter combination (on line).
- *outputs* of control is desired acceleration, reached after summation "4".
- feedback output is transformed, via *feedforward* multiplication by the *estimated* mass $\hat{m}$, into force F to system actuators.
- *disturbance* force F_{dist} applies after control, at summation "5". This is not a summation that is performed in software, because it is realised by nature, in the real world. The measurement actions (that turn real-world values into digital numbers) are not depicted explicitly.
- that real-world *system* is depicted in the Figure within the dashed rectangle. It is *modelled* to be a perfect double integrator with real mass m.

The *function blocks* are the rectangles in the control diagram, and they represent a multiplication; each circle represents a summation function block. The arrows in the diagram represent *data blocks*, but also some of the rectangles have data inside, e.g., the estimated mass rectangle. The high-level *schedule* that realises the controller's *event loop* (by triggering *function blocks*) is the following:

```
when triggered // = OS executes controller every, say, 10 milliseconds
 do {
  communicate()  // read desired position/velocity/acceleration
                 // from input data block(s)
                 // read actual position/velocity from sensors
  schedule()     // trigger function blocks, in the following order:
                 // sums 1 & 2, multiplications k_p & k_v,
                 // sums 3 &; 4, multiplication \hat{m}
  communicate()  // write computed control force to actuator data block
 }
```

It is possible that the computation of the control loop generates *events* itself. Or rather, such events are generated in *monitor* functions that are not shown explicitly in the Figure, and that the `schedule()` function adds to some data blocks in the controller. For example:

- when an *error* between desired and actual state parameters is too large.
- when the *trend* of the error is undesired, e.g., always positive.
- when the computed control force F is too large for the actuators.
- when the actual execution sample time deviates too much from the desired one.

It is possible that the computation of the control loop must react to *events* that come from the outside (and that are different from the *timer events* that most control loops rely on). For example:

- a new motion plan is started, so that some control parameters must be reset, such as the setpoints and the gains.

- the current plan is interrupted, so that the controller must bring the system to a safe stop as quickly as possible. In practice, this boils down to the controller starting a new motion plan itself.

Hence, the control loop event queue must be extended with `coordinate()` functions to react to (and/or generate) events, and `configure()` functions to realise the reconfigurations triggered by the coordination execution. The "safe stop" functionality would require the addition of extra functions blocks, hence by a new `schedule`.

Implementations of control loops used to require no asynchronous Communication, but just synchronous reading and writing from data in the memory of the computer; the Programmable Logic Controller (PLC) works like this, and while it still is *the* workhorse of the automation industry, all modern versions implement the asynchronous and *hybrid* variants. The key hardware-supported technology here is memory-mapped IO. But most modern robotic systems now have one or more field busses, such as CAN, EtherCat, or another Industrial Ethernet variant, so some asynchronous Communication parts have become necessary in the event loops. Such software architectures require at least two asynchronously running activities: the field bus device driver takes care of the communication over the network, and writes/reads messages into the data blocks that the controllers use in the their event loops.

Interrupts are another very important source of events in control systems; many modern interface devices can be configured to generate events to which the operating system will react. The application can configure the operating system to schedule a specific interrupt handler function as soon as the interrupt arrives. (The above-mentioned communications most often work in such an interrupt-driven way.)

7.12.2 Policy: hybrid event control

No realistic task can be realised by just one single control loop, and so-called **hybrid event controllers** are needed:

- the **continuous** control behaviour is realised by feedback control loops, like the one introduced above, or by a constraint optimization solver.

- some **discrete** control behaviour is added, often in the form of a Finite State Machine, where each state executes a different continuous controller, together with other continuous time and space computations, such as monitors, observers, adapters, etc. Transitions between continuous controller modes are triggered by events, generated by the actually running continuous controller itself, or by external activities.

Obviously, such hybrid controllers fit perfectly in the event loop approach, since that structures the computations, communications and configurations of the control loops, the FSMs, the event triggering and processing, with synchronous as well as asynchronous activities.

7.12.3 Policy: throughput and latency

Applications require a variety of controllers, and one of the major design trade-offs is that between optimising the controller's **scheduling** for either of the two following *Quality of Service* measures:

- **throughput**: the **more data is processed**, the better. This is important when the behavioural performance of the controller depends on the amount of information that can be extracted from the raw sensor data.
- **latency**: the **faster functions** are executed, the better. This is important when (i) the natural dynamics of the real-world system under control is "fast", and/or (ii) the control design method requires "exact" timing of the controller computations since the behavioural performance of the controller depends on it.

In many applications, Tasks have a need for both types of computations, the former typically to update their *world models*, and the latter to realise their *feedback control*. An often seen adaptation of the generic high-level control schedule first does the feedback control as fast as possible and only then spends the remaining computing cycles to world model updating:

```
when triggered
 do {
  communicate()       // read only sensor data needed for control actions
  schedule-feedback() // now do all Tasks' feedback control actions
  communicate()       // write computed control efforts to hardware
                      // read extra sensor data needed for world model updates
  schedule-updates()  // now update all Tasks' world models
  coordinate()        // only now process events that could
  configure()         // trigger reconfigurations
  communicate()       // do all remaining non-control communications
 }
```

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

Many robotics and cyber-physical systems contain one or more activities whose execution must be **predictable** ("**deterministic**", "**realtime**") with respect to the computational resources they have available:

- **time**: the execution must take place within a small tolerance of the ideal instance in time. The two key performance measure are *latency* and *jitter*.
- **memory**: the execution must respect consistency constraints on the data structures that are operated upon by the various dataflows used in the activities. A key performance measure is *mutual exclusion*, or *locking*.
- **interrupts**: interrupts can preempt most of the software citivies on a computer, so realtime applications must configure the interrupt capabilities appropriately. The common configuration options are: to inhibit ("mask") some interrupts before a realtime activity is launched, to inhibit interrupts on the set of cores that share cache memories with the realtime core, or to assign only the realtime relevant interrupts to the CPU core on which the realtime activity is running.

The **good practice** solution, Fig. 7.9, splits the event loops of these activies into multiple parts, each in a separate **thread**, and all contained within the same **process**:

- the **mediator** thread is the one that comes with the process that is deployed in the operation system, to create the other threads in the process, and to manage their *Life Cycle State Machines*:
  - *resource creation & deletion*:
  - *resource configuration*:
  - *capability configurations*:
  - *running capabilities*:
  - *pausing capabilities*:
- the **realtime** thread executes (i) all *Computations* that must be executed *immediately*, (ii) all *Communications* that are done via non-blocking memory-mapped I/O, and (iii) the *Coordination* that is triggered by the realtime event loop itself and must be dealt with immediately (e.g., deciding to switch to a fail safe control mode).
- the **workers** are other threads, each with one single responsibility, such as (i) feeding the realtime thread with the dataflow it needs, (ii) getting the realtime dataflow and distribute it to the registered clients higher up in the control stack, and (iii) getting the diagnostic information back, to allow for online or offline analysis of the control performance. (Without loss of generality, the latter can be seen as just a special case of (ii).)

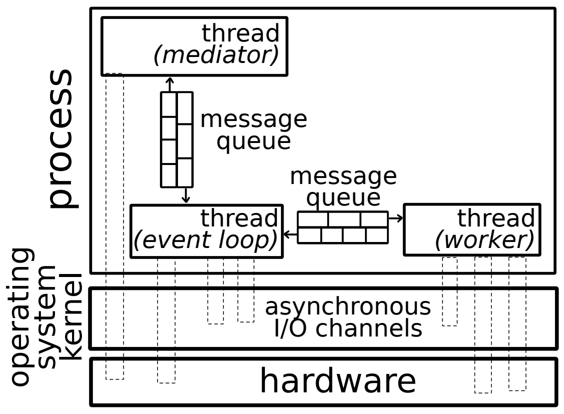


Figure 7.9: Multi-threaded process architecture for the concurrent and parallel execution of activities. The "message queues depicted in the figure represent any type of fast and local inter-thread communication; e.g., lockfree buffers, circular buffers, etc.

The realtime performance of multi-threaded design depends to a large extend on the choice of buffers between both threads; the two common policies are to use either a **locked** buffer approach (by means of a *mutex* or another locking mechanism), or a **lockfree** buffer approach.

The first thread is sometimes called the **hard** realtime thread, and the other ones the **soft** realtime thread. These adjectives have no absolute meaning, and the major **best practice** design requirement is that there can be **only one hard realtime thread on the whole computer**; and giving that thread the highest priority allowed by the operating system is just a *necessary* but *not sufficient* condition for reaching this requirement. Putting the hard realtime part on a dedicated computer system that runs no other software, is often the only really deterministic design. The state of the technology allows to make dedicated chips (e.g., FPGAs) that can even do away with any dependency on the software of an operating system.

7.12.5 Policy: event loops for Task control

The Task meta model is the core structure for the design of cyber-physical systems, so it is necessary to identify and model the impact every specific Task model has on the design of the event loops in its activities. More concretely, all 'property graph 'arrows" in Fig. 2.4 must be turned into decisions on how to exchange model data in an activity, synchronously or asynchronously. So, the generic event loop structure of Sec. 2.11 will most probably get specialisations of the generic `communicate()`, `coordinate()`, `configure()` and `compute()` functions, with explicit references to the (interactions between) the Task meta model aspects of control, monitoring, plan, world model and perception.

Chapter 8

Meta models for perception and its integration in tasks

Perception is *dual* to control, in many aspects and for all engineering systems, so that both control and perception "stacks" can be seemlessly integrated, at all levels of abstraction. However, in a robotics context the amount of perception opportunities (that is, sensors with sensor data processing activities) is huge; however, the information and software architectures for the integrated control-and-perception stacks are copies of those for the control stacks in themselves.

Previous Chapters focused on the *motion specification and control* aspects of a robotic model and software stack. Motion in a robotics context always requires various forms of *perception*: specifying and controlling motion requires access to information about how "the world looks like" at any given moment, and that requirement can only be achieved if the (task-relevant part of the) world is perceived. Examples of such close integration between motion and perception are visual or force-based tracking of the interaction between a moving robot and its environment. So, it does not make much sense to develop all stacks independently, or to deploy their software implementations in only loosely coupled components: the way how things are perceived by robots, how robots are perceived by other agents, or how robots can/should move, depends to a large extent on how the world around the robots looks like, and on what information of that world can be provided by the sensor-based perception; similarly, motion is in many cases important to help perception, especially to improve *observability* of the world model updating process; finally, the task capabilities that a system offers help to focus the perception to those sensor-processing efforts that are relevant to make progress in the task execution.

The term "stack" refers to the *hierarchical information/model structure* of all entities and relationships involved in perception. The first, **mereo-topological**, step in that direction is sketched in Fig. 8.1. This Chapter first explains that mereo-topological model in more detail, and then adds the meta models with structure and behaviour at the geometrical, dynamical, information theoretical levels of abstraction, using **Bayesian information theory** as the scientific foundation of this meta modelling. The connections are made for the task-centric integration between motion, perception and world modelling, allowing to model explicitly how task specifications can add artificial constraints on the perception behaviour.

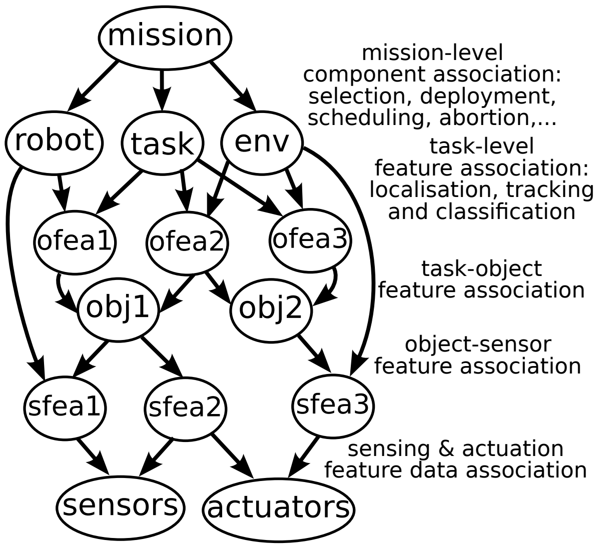

mission
robot
task
env
mission-level component association: selection, deployment, scheduling, abortion,...
task-level feature association: localisation, tracking and classification
ofea1
ofea2
ofea3
task-object feature association
obj1
obj2
object-sensor feature association
sfea1
sfea2
sfea3
sensing & actuation feature data association
sensors
actuators

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

8.1 Mereo-topological meta model: the natural hierarchy in robotic perception

The perception stack model has *structural* parts and *behavioural* parts. The structural part uses *hypergraphs* to model the fact that n-ary relations exist between entities in the stack; these structural relations conform to the Block-Port-Connector meta model. The behavioural models describe the dependencies between of the values of the properties in the connected entities, and these dependencies can be continuous, discrete, or hybrid. For robotic systems, every perception model has n-ary *relations* between *entities* of the following types:

- **sensor**: to describe the properties of the **data** generated by sensor devices.
- **actuator**: to describe the properties of the **data** to be provided to actuator devices to make them put energy into the system.
- **features**: relations between *sensor* data and *object* properties that play a role in the context of a *task*, or between *object* properties and their role in a *task* to determine how the *robot* should move, or both of the above within one single relation. "*task*" can be replaced by a composition of entities in the models of the *task*, the *robot* and the *environment* in which the previous two operate.
- **objects**: have properties that can be linked to data features, for sensing as well as actuation, and to the tasks that describe what robots have to do with them.
- a **robot** model represents the sensing and actuation capabilities and resources of robotic devices and systems.
- **environment** entities, are often relevant for parametrizing perception algorithms (e.g. camera parameters due to lighting conditions) or to select appropriate sets of sensors (e.g. during fog or rain outdoors or when encountering a dark indoor area during night or in the basement). Obviously, this part of the perception stack contains the links to world modelling; an important development within the project will be the "right" separation and composition of perception modelling and world modelling.

- the **task** plays a crucial role in constraining the selection of all other entities ranging from limiting object types that are relevant during that task to the selection of the perception features that need to be detected.
- the **mission** model makes choices of which task, robot and environment models must be used together to realise "long-living" applications.

The models of all the above entities are to be extended with proper **numerical representation**.

8.2 Policy: (data) association

Association relations represent the inherent uncertainty of the inference process that must decide which (sets of) "features" at a lower level of the perception hierarchy are "caused" by which (sets of) "properties" at a higher level. Such association relations appear (at least!) in four different complementary ways, as indicated in Fig. 8.1: between sensors and features, between features and objects, between objects and task/robot/environment, and between a mission and the tasks, robots and environments it requires. Even a small number of possible choices in every association relation results in a huge number of uncertainties in the overall system model; this complexity is most often very much underestimated by the human mind.

The term **data association** is most often reserved for the association between the sensor data and feature properties.

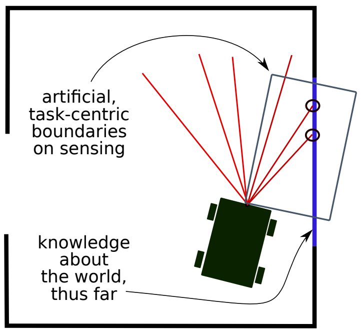

artificial, task-centric boundaries on sensing
knowledge about the world, thus far

Figure 8.2: Application example of the *escape from the room* task (Sec. 5.3.2) to illustrate various levels in the perception stack hierarchy of Fig. 8.1.

8.3 Perception example: robots driving in traffic

This section provides "running examples" for this Chapter's modelling an application conforming to the perception stack meta model.

8.3.1 Escape from a room

Assume the robot has a *laser scanner*, *encoders* on the wheels, and a *cameras* (one looking down to the floor, to use its texture for self-localisation; one looking to the ceiling, for similar purposes; and one looking forward). Part of the sensor models describes the physical units, the mathematical, numerical and digital representations of the data that the sensors produce. For the camera this is a matrix with dimensions defined by the sensors resolution property; each of the values in this matrix is a vector containing the RGB values, which are in turn chosen to be represented as integers between 0 and 255. The laser scanner has a similar representation, but with the 2D RGB image replaced by a 1D vector of depth values.

The digital representation of the camera image is used by one or more *segmentation* algorithms. For example, one based on color is configured with the size and color properties of the expected features in the room; assume that there is a wall with a round green drawing. While the system architect would only choose the type of algorithm, the system builder needs to choose a specific implementation here (e.g. in which color space to look for "green segments"). Also the grounding what "green" means in terms of regions in a color space needs to be grounded in a digital representation (potentially by linking to an ontology describing colors in various spaces); in addition, also the *environment conditions* play a role, because the perceived color depends not only on the object properties but also the lighting conditions. The output is a set of green regions, which some algorithms might use as prior knowledge for the next iteration.

To simplify the data association problem, it is assumed that only the circle with the highest probability will be used. This circle has a state which is represented as its centroid and diameter. By only looking at the numbers shown in Figure 8.2 it is difficult to say that these numbers are in image or pixel coordinates. Therefore, it is again important to point at the meta model describing the digital representation and the semantics of the data.

This centroid and diameter found in the camera image are then used as an input to a Kalman Filter (some additional pre-processing is not displayed). A Kalman Filter is a generic, "platform", algorithm that needs to be configured with a process and a measurement model, initial conditions, as well as noise parameters. These are configured from the sensor model, task model, and object model. Please note that, in contrast to the perception stack, the task is not explicitly shown in this figure since it is influencing the overall architecture and choices. A Kalman Filter requires a state to work on, which is a (dynamically changing) property of the ball. Again, its digital representation is important as is the semantical context like the frame its position is expressed in (see motion stack).

The green round feature typically has many properties that can also change with every new application. Therefore, the suggested structure allows them to be composed with the "ball" while keeping their semantic context by pointing to the models they conform to. The number of possible object properties is huge and will have to grow over time.

8.3.2 Ego-motion estimation with accelerometer, gyro and encoder

(TODO: link the proper time derivatives of the trajectory of the plan with the corresponding levels in the sensors: linear acceleration in the accelerometer, angular velocity in the gyroscope, and wheel position in the encoder. Then do a *least-squares* parameter identification for each of the sensors separately, or by weighing them all together with the uncertainty magnitude of each individual sensor source.)

8.3.3 Ego-motion estimation with visual point and region features

(TODO: point features are abundant in vision, at the detriment of regional features, that are often more difficult to compute, more dependent on the application, and on other features. Typical regional features are: entropy, geometric or tecture patterns, and spatial and temporal frequencies, often on the raw pixels but also on pre-processed pixel values.)

8.4 Mechanism: Bayesian information theory

A **model** is a set of relations between entities in a domain, and the model for **information** (or **uncertainty**) is a *Probability Density Function* (PDF) $p(X, Y, \dots)$ over the parameter space of a selection of the properties in the model:

- *discrete* PDF: parameter space has only *finite* number of possibilities.
- *continuous* PDF: parameter space is continuous.
- *hybrid* PDF: parameter space combines discrete and continuous sub-spaces.

Information representation is *subjective*, because a PDF is a multi-dimensional, single-valued function $p(X, Y, Z, \dots)$ that describes the probabilistic relationship between the variables $X, Y, Z, \dots$ **in a model** M:

$$p(X, Y, Z, \dots | M)$$

and the model M is a *chosen* representation of the *chosen* relationships (assumptions, constraints,...) between the variables $X, Y, Z, \dots$ So, the PDF represents what the *system "knows" about* the world, *not* what the world really *is*. *Engineers* have to *choose* what mathematical representations to use, for domain as well as for information!

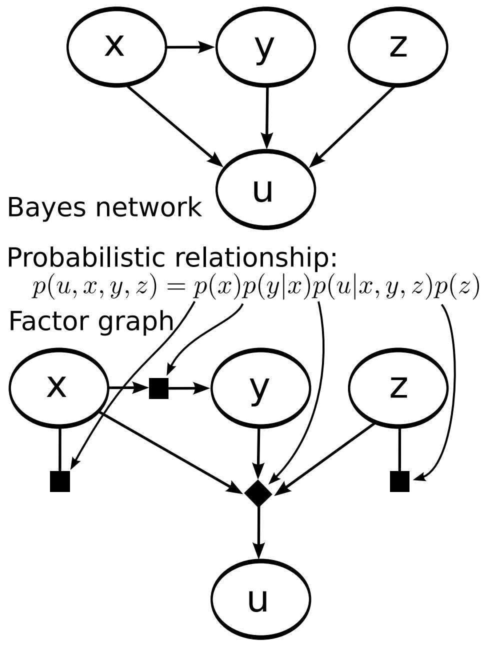

x
y
z
u
Bayes network
Probabilistic relationship:
$p(u, x, y, z) = p(x)p(y|x)p(u|x, y, z)p(z)$
Factor graph
x
y
z
u

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

Information structure = graph:

- **node** contains **variables**, with representation of their uncertainty.
- **arc** (edge, link, arrow, ...) contains **(probabilistic) relationship** between variables in connected nodes.
- terminology: Bayesian network, belief network, factor graph.
- same *real-world* system can have various graphical *models*.

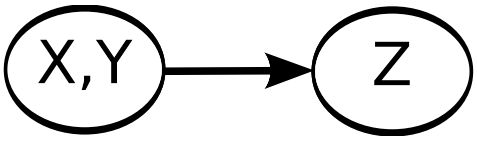


Figure 8.4: Simplest Bayesian network.

The most important arcs are the ones that *are not there*!

The simplest Bayesian network is one with just one directed arc, Fig. 8.4. The *explicit* relationship $Z = f(X, Y|\Theta)$:

- "if I know something about X and Y, so what can I now then predict about Z?"
- arrow direction: "easy" *to calculate*
- is not necessarily *physical causality*
- *factorizes* joint PDF via *conditional PDFs*:

$$p(x, y, z) = p(z|x, y)p(x, y).$$

Mathematical representation of a PDF: a single-valued, positive function $p(x)$ + density "dx" around the value x. What really counts is the **"probability mass"** ("**expected value**") over a certain domain D:

$$D = \int_D p(x)\, dx$$

Extra "property" of PDF: **integral** over complete configuration space of x = **1**:

- value "1" is arbitrary choice/convention!
- only **relative** value of probability mass is important.

Measure of (change in) information:

- mutual information, relative entropy of *two* PDFs P and Q:

$$H(P\|Q) = \int_X \log \frac{dP}{dQ}\, dP.$$

- *"how much does information $\underline{\text{change}}$ when new data becomes available?"*
- *"no information"* does not exist $\rightarrow$ always *relative*!

Mean $\boldsymbol{\mu}$, Covariance $\boldsymbol{P}$:

$$\boldsymbol{\mu} = \int \boldsymbol{x}\, p(\boldsymbol{x})\, d\boldsymbol{x}, \qquad \boldsymbol{P} = \int (\boldsymbol{x} - \boldsymbol{\mu})(\boldsymbol{x} - \boldsymbol{\mu})^T p(\boldsymbol{x})\, d\boldsymbol{x}$$

($\boldsymbol{\mu}$ is vector) ($\boldsymbol{P}$ is matrix)

Advantages of Gaussian PDF representations:

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

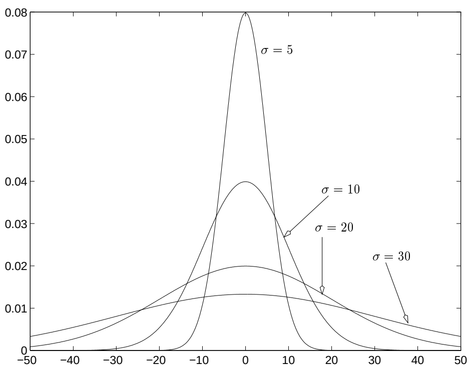


Figure 8.5: Simplest PDF: Gaussian (or **normal**) PDF, with **mean** $\mu = 0$ and **variance** $\sigma = 5, 10, 20, 30$.

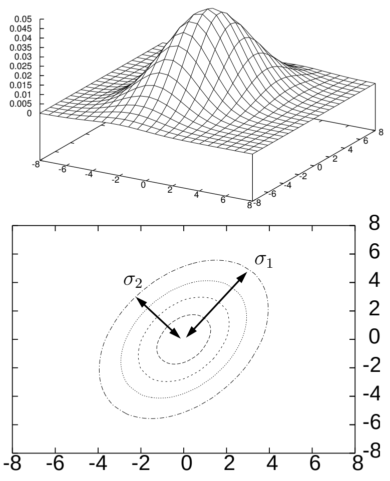


Figure 8.6: 2D Gaussian.

Disadvantages:

- mono-modal = uni-variate = only one "peak".
- extends until infinity = never zero.

Efficient **extensions**:

- sum of n Gaussians: can have up to n peaks.
- exponential PDFs: $\alpha\, h(x) \exp\{\beta\, g(x)\}$: analytically tractable.

Sample-based PDF is an **approximated PDF** by means of samples with a weight:

Operations on PDFs (e.g., Bayes' rule) reduces to operations on samples. For example, "integral" becomes "sum":

$$\int \phi(x)p(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} \phi(x^i) = \sum_{i=1}^{N} w^i \phi(x^i)$$

8.5 Geometrical semantics in perception

8.6 Dynamical semantics in perception

8.7 Policy: tracking, localisation, map building

1. **tracking**: how does an identified object's position in the world change over time?

2. **localisation**: where is the robot in the world?
   Recognition is perception of the same type as localisation, but intended to know where particular objects are in the robot's environment.

3. **map building**: what is the map of the world?

The modelling (and hence also computational) complexity increases roughly with an order of magnitude with every category of perception.

8.8 Mechanism of information update: Bayes' rule

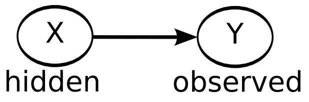
The essential role of Bayes' rule: *"Inverse probability"*:

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

$$\text{(product rule)} \Downarrow \text{(product rule)}$$

$$p(x|y,H)p(y|H) = p(y|x,H)p(x|H)$$

$$\Rightarrow \quad \boxed{p(x|y,H) = \frac{p(y|x,H)}{p(y|H)}p(x|H)}$$



Bayes' rule, for the inclusion of new data:

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

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

Data: *observed*; Model parameters: *hidden*

All factors are functions of *model parameters*, except $p(\text{Data}|H)$ = often just "*normalization* factor."

Bayes' rule: important properties:

- $p(M|D,H)$: **function** of M, D, *and* H.
- PDF on Model parameters "in" $\Rightarrow$ PDF on Model parameters "out."
- Integration of information is *multiplicative.*
- Computationally intensive for general PDFs.
- Easy for discrete PDFs and Gaussians. (And some other families of continuous PDFs.)

- p(Data|Model params): requires known table or mathematical function Data= f(Model params) to predict Data from Model.
- Likelihood is *not* a PDF.
- Optimal Information Processing and Bayes's Theorem, Arnold Zellner, *The American Statistician*, 42(4):278–280, 1988.

8.9 Mechanism of perception solver: message passing over junction trees

The *message passing algorithm in factor graphs* [5] plays a similar role in perception as the *hybrid dynamics solver* of Sec. 4.9 does for motion. For example, Kalman or Particle Filters, Bayesian networks or Factor Graphs, ARMAX or Butterworth filters, are algorithms with very similar structural properties as the hybrid dynamics solver (Sec. 4.9), as far as they pertain to the "sweeps" over tree structures, and their generation by reasoning about the structural relations (graph interconnections) and the functional constraints ("dynamic programming" solvers of constrained optimization algorithms).

(TODO: much more details and examples.)

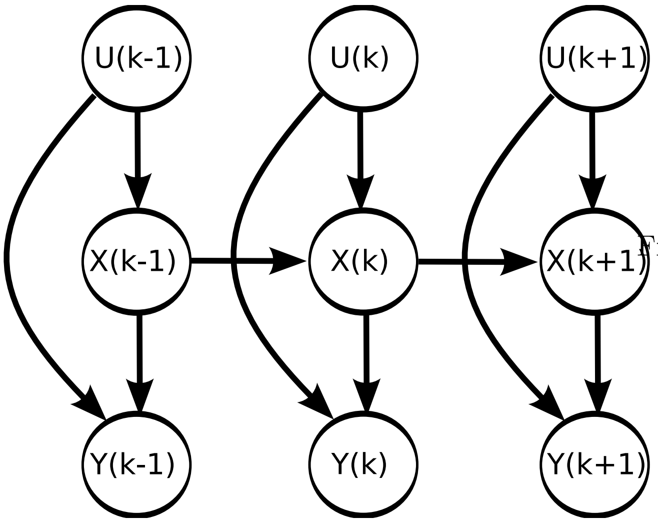


Figure 8.7: Dynamic Bayesian network.

8.10 Policy: dynamic Bayesian network

Figure 8.7 sketches a typical dynamic Bayesian network, where one of the arrows represents *evolution over time*.

Variables:

- U: control inputs
- X: state information
- Y: measurements

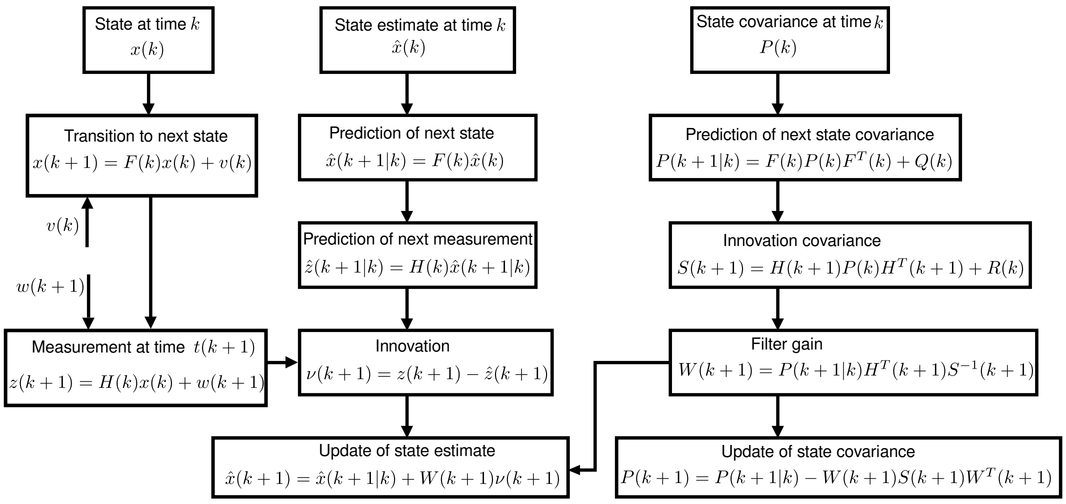
Arrows:

- motion model: $X(k+1) = f(X(k), U(k+1))$
- measurement model: $Y(k) = g(X(k), U(k))$

Multiple arrows can be represented by *one function.*
"*1st-order Markov*" = "time"-influence only *one step* deep.
A dynamic Bayesian network is the probabilistic extension of the representation of a physical control system:

$$\begin{cases} \frac{dx}{dt} = f(x, \theta, u) \\ y = g(x, \theta, u) \end{cases}$$

- x: domain values.
- t: time.
- θ: model parameters (PDF, relationships).
- u: input values.
- y: output values.
- f: state function, or "*process model*"
- g: output function, or "*measurement model*"



State at time k $x(k)$
State estimate at time k $\hat{x}(k)$
State covariance at time k $P(k)$
Transition to next state $x(k+1) = F(k)x(k) + v(k)$
Prediction of next state $\hat{x}(k+1|k) = F(k)\hat{x}(k)$
Prediction of next state covariance $P(k+1|k) = F(k)P(k)F^T(k) + Q(k)$
$v(k)$
$w(k+1)$
Prediction of next measurement $\hat{z}(k+1|k) = H(k)\hat{x}(k+1|k)$
Innovation covariance $S(k+1) = H(k+1)P(k)H^T(k+1) + R(k)$
Measurement at time $t(k+1)$ $z(k+1) = H(k)x(k) + w(k+1)$
Innovation $\nu(k+1) = z(k+1) - \hat{z}(k+1)$
Filter gain $W(k+1) = P(k+1|k)H^T(k+1)S^{-1}(k+1)$
Update of state estimate $\hat{x}(k+1) = \hat{x}(k+1|k) + W(k+1)\nu(k+1)$
Update of state covariance $P(k+1) = P(k+1|k) - W(k+1)S(k+1)W^T(k+1)$

Figure 8.8: Computational schema of the Kalman Filter.

The simplest dynamic network is the *Kalman Filter*.
Required inference: *given* Y and U, *update* X.
Assumptions: fast analytical solution possible!

- Process model: $x_{k+1} = F\, x_k + Q_k$.
- Gaussian "uncertainty" on x_k: covariance P_k.

- Gaussian “process noise”: covariance Q_k.
- Measurement model: $z_k = H\, x_k + R_k$.
- Gaussian “measurement uncertainty” on z_k: covariance R_k.

Typical application: *tracking* = adapting to *small* deviations from previous values.
Second simplest dynamic network: Particle Filter for localisation.

- functional relationships $f(\cdot), g(\cdot)$: can be *non-linear*.
- PDF *representation*: samples.
- *each sample* is sent through f and g separately, and then a new PDF is reconstructed.

So, a *numerical* solution is needed. **Typical application:** *localisation* with *large* uncertainties.

8.11 Policy: Factor Graphs

8.12 Mechanism: composition of point and region features

8.13 Policy: feature pre-processing

8.14 Policy: deployment in event loop

Chapter 9

Meta models for holon architectures for resilient & explainable system

Designers of cyber-physical systems create **information architectures** (of *activities*) and **hardware architectures** (of *devices*), and interconnect both with a **software architecture** (of *components*). This Chapter provides mereo-topological models of the architectural designs required to compose cyber-physical systems, with *peer-to-peer* interactions. The architectural design process has the following major phases: (i) to identify which **tasks** must be realised by which *activities*, (ii) to identify which *resources* are **owned** by which of these activities, (iii) to define how the *state* of these resources can be *shared* with other *activities*, (iv) to create the software *event loops* to coordinate the execution of the algorithms that provide the *behaviour* of activities and their interactions, and (v) to map these event loops onto an appropriate number of *threads* to exploit the performance of the cores and the networks provided by the hardware architecture.

Performance, *robustness* and *correctness* of the software are obvious mainstream metrics to assess software architectures. This document's (extra) focus on *knwoledge models* provides the information to let the software assess itself, and to explain and to interpret its own decisions. In turn, this the necessary (but not obviously sufficient) condition to make system architectures **resilient** and **trustworthy**. The single most important design driver to reach resilience is to make a system only of **holons**, that is, components/sub-systems that (i) remain operational under *any* (lack of) interaction with their peer components, and (ii) can always decide for themselves when and how to switch to what kind of **graceful degradation** behavioural state(s), while **explaining** their decisions to whatever peer is interested in knowing.

The role of an information architecture is to realise the *tasks* of an *application*, via *activities* that *interact* with the *physical world* (and with each other). Both tasks and physical worlds are *modelled* with *continuous*, *discrete* and *symbolic* parameter spaces, of various levels of abstraction chosen by the designers. A model of the physical world consists of a *parameterized state space*, with *constraints* on (i) the *evolution over time* of the state parameters, and (ii) the *limitation* on the values that each of these parameters is allowed to have. Both constraint types are sometimes called the *natural, or physical, dynamics*, and the *natural, or physical,* *constraints*, respectively. A model of a task (the so-called *task specification*) adds extra *artificial, or task, dynamics*, and *artificial, or task, constraints* on top of the natural ones. In other words, a task specification is a formal representation of how the application wants the physical world to evolve from its *current* state to a *desired* state.

Models alone are not enough to make changes in the physical world. So, any application requires a software architecture (of so-called *agents*, *components*, *holons*, or *digital twins*) to realise the application's information architecture, by executing *algorithms* in *threads* and *processes*, that *communicate messages* between each other to provide information about the state of the physical world. The software architecture must realise the task specification while respecting the physical world constraints. However, software can only interact with the physical world when dedicated hardware resources are added, such as analog-to-digital and digital-to-analog convertors, amplifiers, power supplies or motor controllers.

A **resilient**[1] system has a so-called holonic architecture of subsystems (or "components", or "agents"). Each holon applies the mediator pattern to guarantee **single point of decision making** over each of the "resources" it is responsible for. Each holon is able to base that decision making on the formal reasoning it can perform with the **knowledge relations** it has about its **internal** behaviour and about how its **externally** visible behaviour interacts with other holons. These are (necessary but not sufficient) conditions to realise the system's *predictability*[2] under interactions with any type of **external** system.

It is (probably) useless to try and be precise about "the definition" of a system, because what counts are the *design patterns* and *best practices* that are available to develop, deploy and maintain systems and their compositions. Seminal work in **holonic** system design started in the 1960s and matured around the beginning of this century [26, 30, 41, 50, 57, 55]. The starting point in holonic design is to design a system in such a way that it is ready to be composed into a larger **system-of-system**, sooner or later, *and* that it can **decide for itself** *that*, *when*, *why*, *where* and *how* it becomes part of a larger system, or break away from it.

This Chapter introduces three complementary parts of a system's architecture: **information** architecture, **software** architecture, and **hardware** architecture; the composition is sometimes refered to as a (business or computing) **platform**. *The* major responsibilities of the system architects (compared to component developers) are:

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

- to realise resilience against software erosion: *"[software erosion] is not a physical phenomenon: the software does not actually decay, but rather suffers from a lack of being responsive and updated with respect to the changing environment in which it resides."*

- to have an *information architecture* that does not depend on specific choices made in the *software and hardware architectures* that happened to be the one available the first time the information architecture was implemented. In particular, the *configuration values* should be changeable for each actual runtime that is deployed from the software architecture.

[1]This document uses the terms **stable**, **robust** or **holonic** as synonyms for *resilient*.

[2]But not necessariy its *performance*.

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

9.1 Resilience and explainability as system design drivers

The literature has introduced the concepts of *resilience* and *holarchy* [30, 50] to explain *natural* and *social* systems, and compositions of them into system-of-systems; this document redefines the mereo-topological interpretation of these concepts in the context of the architecture of engineering systems:

> *A system has a* ***resilient*** *architecture if its behaviour is* ***explainable****, and remains so under any change in the* behaviour *of any of the systems it interacts with, as well as any change in the* topology *of its interactions.*

Explainability is the first step in a hierarchy of system behavioural resilience metrics; **adaptability**, **predictability** and **dependability** are next; and **guaranteeability** is the holy grail. A working hypothesis of this document is that the explainability of an architectural design of a system is proportional to the level of **knowledge concentration** that the designer can achieve in providing one **unique mediator** component that makes the decisions about how to coordinate its own system-level behaviour with that of all of its internal subsystems, as well as with all of the external peer systems. In other words, explainability can only be achieved when *all* decisions the system makes to adapt its behaviour to its context, are made in one single place, based on one consistent combination of (offline) *knowledge relations* and (online) *world model data*. Of course, trying to apply this design driver blindly, by centralising decision making, has time and again proven not to be a good approach; the more resilient architectures have "holarchies" of resilient sub-systems, with the "right" loose coupling of their behaviours and, especially, their decision making.

9.2 Meta models for robotic stacks

In the more specific context of robotic systems, the modelling efforts described in this document result, together, in a **large set of single-focus models**, structured in so-called "stacks" for robotic systems. A stack is an architectural structure of functionalities, starting at the "bottom"[3] with sensors, motors and mechanics, builds up towards instrumented and actuated kinematic chains, and further to sensor-based tasks executed by loosely and opportunistically coupled multi-robot systems-of-systems. However, there are also "horizontal" directions to these stacks, because some (meta) meta models are relevant for entities and relations at more than one level of the "vertical" stack direction. And **geometry** is one of the most prominent such "horizontal" meta meta models, together with most other branches of mathematics (analysis, logic, numerical linear algebra, etc.), and with *software patterns*.

The **most mature** stack (and also the most "pure robotics" one) is the "(motion) control stack", because it has the least amount of open world assumptions inside: the robot *is* the world, for the largest part of the stack, and electro-mechanics make up the majority of the

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

meta models for computer-controlled motion of mechanical devices. The good news is that, by now, it is indeed well known how to make kinematic chains move themselves (and the sensors and tools connected to them) with prescribed behaviour, relying on the physically limited variety in motion control modes (current, torque, impedance, velocity, force and position) that can cover the task needs (Chap. 5) of all mainstream applications. The bad news is that, even after 50 years, there is still no standardization in place that covers most of the motion stack; this lack of standardization starts already with the representation of geometry.

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

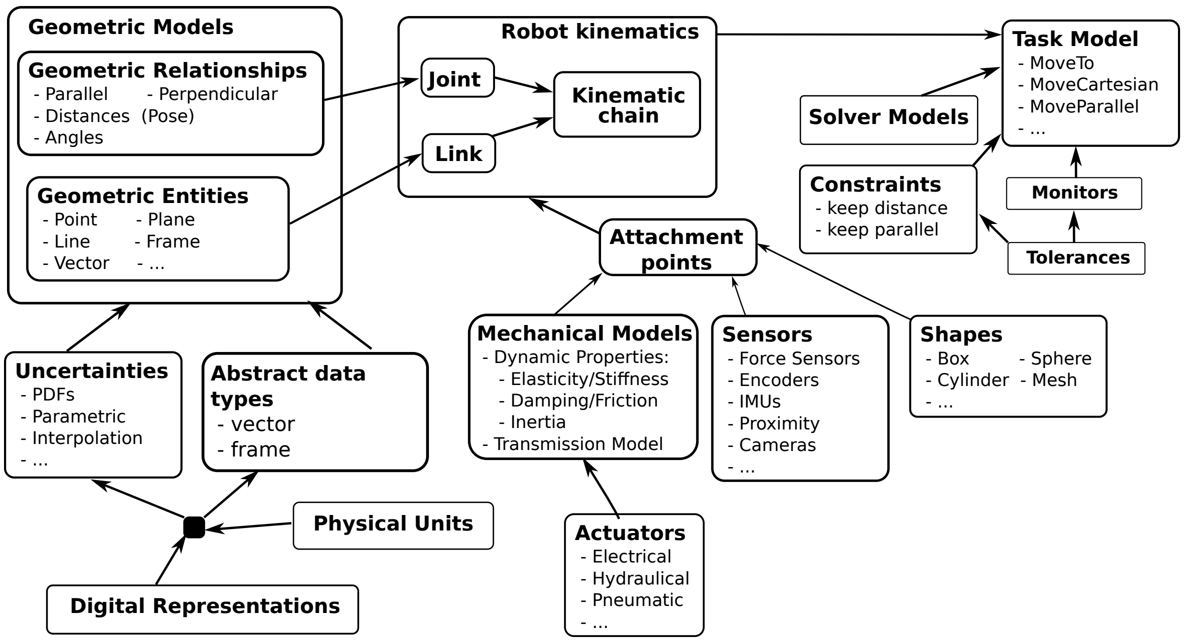


Figure 9.1: The mereo-topological overview of the "motion stack" in which a majority of the depicted blocks makes use of geometrical entities and relations. An arrow represents *composition* of complementary aspects of the system, represented in separated *models*, and eventually implemented in separated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

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

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

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

9.3 Information architecture

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

- **domain models**: the property graphs of the various pieces of domain knowledge the application builds upon; for example, the relevant physical units and mathematical solvers; the rules of traffic; etc.

- **application models**: the property graphs representing the knowledge needed about the application: task requirements, social and technical constraints, Quality of Service, introspection and self-diagnosis models, etc.

- **provenance models**: information about how data, entities and relations have been created, who has *ownership* of them and has the responsibility to make decisions about what to do with them, including providing them to other "organisations", etc. Two established meta models for provenance are Dublin Core metadata, and the PROV models from the World Wide Web Consortium.

- **abstract data types**: from all of the above, select those entities and relations that the application has to create abstract data types for. This includes models for the data structures, communication messages, operators, etc.

- **activities**, with event loops to serve all 5C responsibilities, including managing concurrency of algorithms such that state changes in abstract data types remain consistent.

- **Life Cycle State Machines**: each activity needs its own LCSM to manage the resources it uses, and each task and each event loop need a separate LCSM too.

- **mediators**: each *shared resource* (communication channel, task, space, algorithm, world model, CPU, robot hardware,. . .) needs a mediator activity to centralize the configuration and coordination decision making about that resource. In some contexts, also all *CRUD* operations (Create, Read, Update, Delete) on the shared resource are run through the mediator.
- **solvers**: the algorithms that will realise all the activities behaviour need to be modeled.

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

9.4 Software architecture

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

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

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

9.5 Hardware architecture

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

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

9.6 Digital platform

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

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

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

9.7 Autonomy and decision making

Like most other terms in this Chapter ("architecture", "safety", "system of systems",...) *autonomy* has been given several different definitions, although none of them is sufficiently constructive to be used as a refutable design driver.

9.7.1 Sheridan's ten levels of system autonomy

One of the most popular definitions is Sheridan's **10 levels of autonomy** [37], Table 9.1.

9.7.2 Explanation levels for autonomous decision making

Sheridan's scope was limited to the interaction between one *single* machine and one *single* human. Modern robotic and cyber-physical systems must extend this scope to *systems-of-systems* contexts, with *multiple* agents, multiple tasks, multiple resources, multiple vendors, multiple regulators, and multiple machines. This document introduces a definition of "autonomy levels", Table 9.2, based on the **dialogue** with which a system **explains its decisions** to other agents, human as well as artificial. The granularity of the levels is designed to allow incremental *step change* developments in autonomy, for very focused decision making challenges. Note the huge technical challenges to go from "level 4" to "level 5", and, especially,

Level	Description
10	Computer does everything autonomously, ignores human.
9	Computer informs human only when it sees fit.
8	Computer informs human only if asked.
7	Computer executes automatically, and informs human when neccessary.
6	Computer allows human restricted time to veto before starting action.
5	Computer executes suggested action if human approves.
4	Computer suggests one single alternative.
3	Computer narrows alternatives down to a few.
2	Computer offers a complete set of decision and action alternatives.
1	Computer offers no assistance, human makes all decisions & actions.

Table 9.1: Sheridan's ten levels of system autonomy [37].

from"level 6" to "level 7". These steps introduce two subsequent levels of empathy: (i) to reflect on one's own actions and put them in the context of the **user** for whom a Task is being executed, and (ii) to create and maintain also the world models of **other systems**, and to reason in their place.

9.7.3 Links with horizontal and vertical integration

A system' decision making levels of Table 9.2 are agnostic to the scale of the system, so they are also relevant for any type of horizontal and vertical composition of Tasks (Sec. 2.10.9). If such a composition is constructed with knowledge-driven hybrid constrained optimization (KHCOP), the on-line solvers of such KHCOP problems have already answers to some of the questions in Table 9.2, because decisions are made in the Coordination state machines and Task progress is monitored semantically via constraint violation checks.

(TODO: examples.)

9.8 Safety

(TODO: explain why safety is a system-level aspect, and composes parts of the HCOPs in all the current Tasks. The role of the LCSM as the model for an independent "safety PLC".)

9.9 Security

Security is a system- and application level aspect, because (i) no middleware can be trusted, and (ii) no task specification can be trusted. Hence, and in addition to the best practice in encrypting all communications, the application has to engage in *dialogues* with all parties that provide or consume data and task specifications. The dialogue consists of back-and-forth questions and answers with a large enough variety in encrypted keys to exclude man in the middle problems. In this document's broader context of knowledge-driven systems engineering, these dialogues do not impose a lot of extra overhead because a lot of messages are already

Level	Description
	One system — One task
1	What am I doing?
2	Why am I doing it?
3	How am I doing it,. . .
4	. . . and how well am I doing it,. . .
4b	. . . and how do I decide to stop doing it?
	One system — Multiple tasks
5	What could I be doing instead,. . .
5b	. . . and still be useful,. . .
5c	. . . and how do I decide to switch what I am doing?
6	What is threatening my progress,. . .
6b	. . . and how can I make myself resilient,. . .
6c	. . . and how do I decide to add a particular resilience?
	Multiple systems — Multiple tasks
7	What progress of others am I threatening,. . .
7b	. . . and how can I make myself behave better,. . .
7c	. . . and how do I decide to adapt a particular better behaviour?
8	What other machines and humans can I cooperate with,. . .
8b	. . . and how do I find out how we can coordinate our cooperation,. . .
8c	. . . and how do we decide, together, what coordination to adopt,. . .
8d	. . . and how do we monitor our coordination,. . .
8e	. . . and how do we decide that someone has cooperation problems?

Table 9.2: Systems' decision making explanation levels. They represent the various degrees to which systems can (i) perform self-diagnosis monitoring and Coordination, (ii) *explain* their autonomus decision making, and (iii) adapt in order to increase resiliency [35].

exchanged for other "non-functional" purposes, such as heartbeats, resource mediation, and task coordination.
(TODO: lot more concreteness.)

Chapter 10

Meta models for information architectures

An information architecture consists of the **models** for (i) the **entities** to include in the system, (ii) the **relations** between entities that must be taken into account, (iii) the **constraints** on such relations to be satisfied, and, especially, (iv) the **policies**, or trade-off choices, to be made for every entity and/or relation coupling in the system. **Ownership** is a key policy decision to be made by the information architects: which **Task** owns (which instantioations of) which information in which model, and which **Activity** owns (the execution of) which Tasks?

The model of an information architecture is a **property graph** in itself, representing the dependencies between all of the above, and allowing reasoning on the architectural structure and properties. With some form of]**domain-specific standardization**[1] of formal models of that architectural property graph, systems could *exchange* that information and do reasoning about it *at runtime*.

This Chapter introduces a multitude of architectural models with which **to compose the information** in the various models of the components in a system architecture, with all dependency relations explicitly formalized. Hence, they are available for automatic reasoning, in system development tools at design time, as well as by the controllers of the engineered systems at runtime.

The **Task** is (or rather, *should be*) the causal trigger of an information architecture **design**. It often remains overlooked as a system design driver, maybe because (i) the knowledge relations that link platform resources on the one hand, and Task capabilities and performance on the other hand, span several levels of abstraction and many scientific and engineering disciplines, and (ii) very few *software frameworks* support task-level models.

The **Activity** is the core model of how Tasks are to be **executed**. Making the right decisions about the many "magic numbers" in all of the system's Activities relies on a large body of knowledge. This Chapter introduces best practices and structureal patterns to help system developers to cope with the resulting intricate dependencies.

The **interactions** of Activities with the Task models they execute, take place, generically speaking, via so-called **CRUD** operations (Create, Read, Update, Delete); these operations

[1]This kind of standardization is non-existent in most domains.

must be *Configured* and *Coordinated*, such that the information in the information architecture remains (eventually) consistent.

10.1 Best practice: every Task model is a shared resource

The **model** of a Task is a property graph, that links information in the Task's control, perception, plan and monitoring parts together with information of how the world looks like, at any moment in the Task's lifetime. That means, the runtime version of the Task model represents **state** in the system. State information is only useful if it is *shared*, but updating state information in a distributed execution context implies a risk of making the information inconsistent. Hence, it is the information architects' major responsibility to think thoroughly about which Activities are allowed to operate on which parts of the runtime Task model, and how various "competing" operations are coordinated inside and between Activities. In other words, the Task model must be considered as a *shared resource*, and all relevant design patterns must be applied. The start of this design is to identify (explicitly and formally) what are the *information access dependencies* in the system. Such dependencies often go further than coordinating each operation on each shared part of the Task model, because there is often also "state" in *sequences* of the CRUD operations themselves: some sequences must be applied atomically, or not at all; some sequences can be interleaved ("pre-empted") by specific other sequences; etc..

The **execution** of a Task consumes **platform resources** in often very indirect and hidden ways, via control and perception activities, and it is almost impossible to deploy different Tasks in a system without causing interference at the resource usage level.

10.2 Best practice: every entity and relation has one owner

It is the owner who decides about the policies on which CRUD operations are allowed to act on the data of the entity or the relation.

(TODO: examples: actuator or sensor; task; state of a solver; resource allocation; etc.)

10.3 Best practice: one mediator Activity per shared resource

To give mediation a unique "owner" for all decisions about how to protect *and* optimize access to the resource.

(TODO: sharing same communication channel with various Tasks; sharing overlapping workspaces between two robots; etc.)

10.4 Best practice: one LCSM per Activity

Each Activity has its own "life", independent of any other Activity, and that "life" must be a singleton. Hence, there is one and only one Life Cycle State Machine in each Activity.

Activities are not just essential as information architectural entities to represent "life" of their own, but also to give "life" to the system: they interact with each other to realise a system's desired behaviour. Two necessary (but not sufficient) conditions to make sure that such interactions produce **predictable behaviour** are that (i) each of the interacting Activities is itself in its own internal state that is designed to support the interaction, and (ii) it can then communicate explicitly about the interaction with the other Activities it is interacting with.

(TODO: example: *EtherCat* communication can not be used in realtime before appropriate handshake is performed, and the master as well as all slaves go to the same state in the standardized EtherCat protocol state machine.)

10.5 Best practice: causality in data access policy

Activities must interact with each other to realise Tasks. And each Task brings (models of) causal dependencies between Task functions operating on Task data: a particular function should only operate on a particular abstract data type when certain conditions are met. Such causal relationships must be modelled explicitly, by relations representing so-called *data access policy* constraints.

For example, the *realtime* thread in a dual-thread Activity is the one who does the *writing* from the source of the data to any shared data structure, because it is the creator, and hence the owner, of that data. When the *non-realtime thread* would do the copying, the data transfer could be interrupted by preemption of that thread.

(TODO: more examples.)

10.6 Design Pattern: Peer-to-Peer Mediation in Activities

The extremely simple structural composition pattern[2] of Fig. 10.1 applies to the many use cases in which the *client-server* behavioural composition needs to be realised between a Service and its Client (or multiple of them), and when one wants to avoid *direct behavioural* coupling between them. That is, they do not know about each other's existence, name, software component port address, or data access policies. But, of course, they *do* have a *behavioural* coupling in the form of (i) the representation of the information they exchange, and (ii) the events through which they *coordinate* their behaviours and *configure* them. These couplings, however, depend only on the *type* of their Activities, and not on the concrete *instances*.
The design driver force in the pattern is **to concentrate** all **knowledge and decision making** about the **behavioural coupling** in a so-called mediator Activity:

1. the mediator knows the **information representation relations** of all peers involved, irrespective of whether a Peer has the role of a Service provider or of a Client consumer. The mediator Activity must check whether those relations are compatible, decide to

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

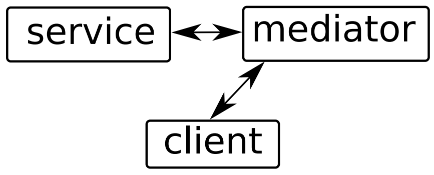


Figure 10.1: Mereo-topological model of the *Peer Mediator* pattern: many interactions between a *Service* and its *Clients* require an extra Activity to guarantee that all data access policies of all Peers (i.e., Service, and Client Activities) are met, and to make decisions how to reduce the service quality to each Client when there are not enough resources to satisfy all Client requests in full. The arrows represent such data access policy constraints between the behaviours inside the connected ctivities.

continue or not with its brokerage activity, and to include format transformation glue code when necessary.

All this model checking and adaptation takes place in the resource configuration step of the Life Cycle State Machines of all Peers involved.

2. the mediator knows which **events** the various Peers emit and/or react to, in order to change their behavioural states. It can then decide whether all Peers have events with the same semantic meaning, and can include *event forwarding* glue code when necessary.

3. the mediator knows about the **Tasks** that the various Peers have to support, together, and can add extra components to realise Service Level Agreements, or, at least, monitor the **quality of service** of the coupled behaviour and fire the appropriate events to all Peers when the Task service falls below the required threshold.

10.6.1 Resilience

From the pattern description in the previous Section, it is clear that mediators play a key role in a system's information architecture to achieve **resilience**, and therefore the *human* system designers must first succeed in *all* of the following:

- *to identify* the system's capabilities and resources, and *formalize* the *hybrid constrained optimization problems* that model their interactions in a declarative way.
- *to formalize the knowledge* about the system's internal behaviour: abstract data types and their operations; the activities and the event loops inside, to bring the system's behaviour *live*; its operational modes and the corresponding finite state machine; Life Cycle State Machine to coordinate its resources; etc.
- *to identify* the spectrum of its own internal behaviour that can be explained by the formalised knowledge, and formalize the *boundaries* as constraints on the internal behavioural relations.
- *to identify* the spectrum of the external behaviour that it can tolerate at its interaction ports, and add the corresponding relations that constrain the internal behavioural spectrum.
- *to formalize* the knowledge that it needs *to explain* how to react "best" to external behaviour, based on reasoning with all of the above-mentioned knowledge relations.

- *to identify* how *to monitor* at runtime when the external interactions enter explainable behavioural areas, *and* how to connect the monitoring events to the internal coordination relations. *Formalize* these new higher-order knowledge relations.

The full package can only be provided in mature domains, and by senior domain experts. In addition, all knowledge relations involved are of the higher-order type, coupling all behavioural modes of all partners in the composition. Decision making via multiple concurrent decision makers has difficulties with realising explainability of the decision making in a predictable way, so it is a *best practice* to have only **one mediator per composition** to realise that higher-order decision making. (This guideline is in itself an instance of another structural pattern, the **singleton** pattern.)

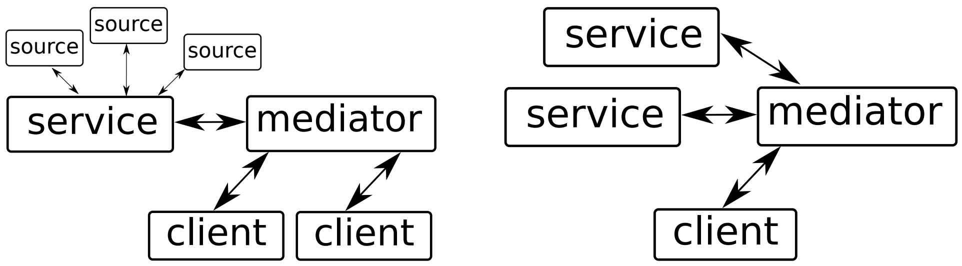


Figure 10.2: Variants of the *Peer Mediator* pattern, with multiple sources for the service, and multiple clients or services for the mediator. More and more application contexts do not have clear server and client roles anymore, so they often use the more neutral term *peer* for both service and client. Anyway, the terminology is not relevant, but the specific dependencies between the specific behaviours in all interacting peers.

As mentioned before, the **difference** with the more simple and common service **broker** pattern (which is commonly used in service-oriented architectures, e.g., pub-sub communication middleware), is that the mediator contains **a lot more knowledge** of the system's **Tasks** as well as of its **architecture**: in addition to the message brokering, it must (i) **coordinate** the activities in both server and client, and (ii) **configure** the "magic numbers" required in the coupling. The relevance of such separation between (higher-order) *Coordination and Configuration* on the one hand (embedded in the mediator) and the *Computation and Communication* on the other hand (embedded in the Peers' Activities), increases with (i) the *number* of interacting Peers (Fig. 10.2), (ii) the *complexity* of the interaction protocols, and (iii) the statefulness of the Peers. One of the consequences is that the mediator must provide a *Traffic Light* instance, to coordinate the multiple *Life Cycle State Machines* in all subsystem Peers.

10.6.2 Application of the Pattern in human society

Human society has introduced the pattern many centuries ago already, by building systems-of-systems of high complexity with societal architectures that are now considered as "obviously" resilient subsystems. For example:

- in the context of *Tasks* to organise the many interactions between humans into a loosely coupled societal hierarchy: person organise themselves in families, they form neighbourhood, making up villages, organised into a metropolitan area around a centre town, all assembled in regions and countries.

- in the context of *Tasks* to provide building infrastructures: rooms are structurallt connected by corridors, that form wards with some form of functional cohesion, aligned into floors, that form wings of buildings.
- in the context of *Tasks* to organise industry: device are interconnected with high-performance local networks into work cells, forming more flexibly and loosely communicating lines, laid out in plants, that are legally coordinated as a company.

Modern societal instantiations of the mediators are: the *foreman* in a team of factory workers, the *coach* in a team of athletes and staff, the *CEO* of a company, the *mayor* of a town, the *architect* of a large public infrastructure construction, etc. Examples where the pattern has been applied in engineering systems are:

- a motor control service for robot end-effector tasks: the mediator deals with the specificities of the kinematic chain that links the motors to the tasks, such as redundancy resolution, singularity avoidance, energy optimization, etc. Figure **??** sketches the context of mediating between the 3D motion task "client" in the platform and the "traction sources" in the eight wheels, where each two-wheel unit provides a "force service" to the platform. Section 10.6.5 explains this use case in some more detail.
- system of systems integration where the communication performance *in* each system is high, while it is low *between* systems. For example, a production line in a manufacturing plant where the different devices in each work cell can communicate via optimized channels and components developed by the same team, while the coordination between the work cells can only make use of the less mutually optimized communication and behaviour coordination of independently developed machines from different vendors.
- single-page applications, such as a graphical user interface (GUI) for a very stateful system of systems but an extremely stateless GUI.

10.6.3 Example: predictable behaviour of concurrent Activities

shared resource(s) between activities, with CRDT data; LCSM for each activity; Traffic Light for inter-activity synchronization;

(TODO: link forward to the software architecture cousin of this example, namely the realtime control thread pattern, and wait-free buffer.)

10.6.4 Example: shared world model for self-calibration

Robots should be able to identify their own kinematic chain structure for themselves: how are its sensors, joints and motors connected together.

(TODO: put IMU on robot and let robot move such that the position of sensor is found automatically. Do "SLAM" on its own kinematic chain world model.)

10.6.5 Example: motor + kinematic chain + Tasks

A robot can, in general, realise more than one Task at the same time, but it requires a mediator Activity to find out by itself which Tasks it can realise at what moment and with what quality of service, and to make decisions about which parts of which Task *not* to realise. Of course, it makes a lot of sense to let the mediator Activity dialogue with the Clients of the kinematic chain Service, to configure that decision making process.

(TODO: examples.)

Chapter 11

Meta models for software architectures

A software architecture consists of XXX. The result is that a software architecture is a property graph in itself, representing the dependencies between all of the above.

Addition to the *application-centred* information architecture: models about how the hardware works to do computations and communications, that is, distributing sofware over multiple cores, processes and computers. Focus on *streams* as the major model to let different Activities share information, under the conditions of the CAP Theorem, that is, without the hard constraint of strict data consistency and loss-less data exchange.

Design principles:

- streams ("dynamic, endless arrays") are important composite data structures because they are key for producer-consumer based, asynchronous, reactive operation.
- arrays are important data structures because of their memory locality, and the accompanying synchronous operations.
- lock-free concurrency avoids involvement of the OS kernel.
- *single-writer principle* avoids cache line contention since it allows atomic semantics. The Single Writer Principle is that for any item of data, or resource, that item of data should be owned by a single execution context for all mutations.

11.1 Best practices in single-computer stream processing

Challenge is resource contention, that is, to allow functions in multiple Activities to access data structures in a concurrent way, with predictable trade-offs between (i) data loss, (ii) latency, and (iii) configuration of multi-core performance model. Lock-free and wait-free algorithms for data sharing [?] form an essential body of knowledge, where software and hardware properties are constrained together.

11.1.1 Singleton data ownership

(TODO: all data to be *written* is always owned by one clearly identified owner.)

11.1.2 Single-computer/multiple-cores: memory barriers

(TODO: *single* producer `writes` to stream and multiple consumers `read` from stream asynchronously, on the same multi-core shared memory but without locks, only relying on memory barriers.)

11.1.3 Single-computer/multiple-cores: asynchronous stream processing

(TODO: producer `writes` to stream and multiple consumers `write` to stream asynchronously, but without locks. CAS operation on different cache line *sequences*, sugin the Disruptor circular buffer pattern. This relies on singleton ownership.)

11.1.4 Single-computer/multiple-cores: synchronous stream processing

(TODO: producer must `read` and `write` to stream without ever waiting. CAS operation on same cache line *freelist*, hence improving cache coherence.)

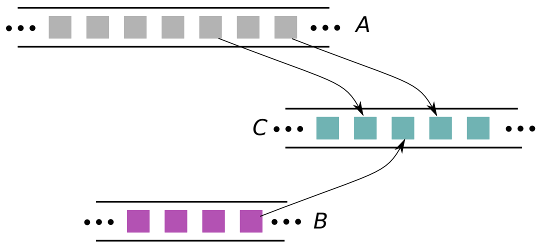


Figure 11.1: The mereo-topological model of *streams*, and *stream processing*.

11.1.5 Ring buffer — Circular buffer

(TODO: explain the "Disruptor" ringbuffer implementation; single/multiple producer, single/multiple consumer, single/multi threaded, deployment on cores.)

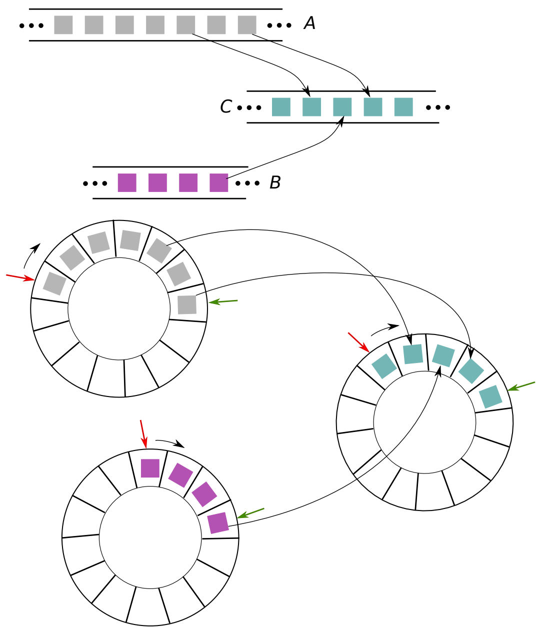


Figure 11.2: The mereo-topological model of *streams*, and *stream processing*, Fig. 11.1, together with the software model of the *ringbuffer* that conforms to that information model.

11.1.6 Policy for stream flow control: backpressure, polling, compaction, negotation, overwriting

Resilience requirement implies that the design of a buffer must be ready to deal with the case that the producer overruns the consumer(s), hence the buffer fills up. There are multiple ways to cope with this case:

- *backpressure*: the producer gets an event to slow down, as soon as the buffer has reached a certain level of filling up; this level is sometimes called the *high water* mark (or "high tide" mark). When the buffer empties below the *low water/tide mark*, the producer gets a signal that it can produce more. Similarly, consumers can get "mirror" events, namely to consume more when the buffer fills up.

- *polling*: buffer is only used on request, not by pushing onto it.

- *compaction*: the data in the buffer that has been produced but not yet consumed (or claimed by a consumer) is reduced, according to an application-specific policy rule.

- *negotiation*: the amount of data per event is reduced, so more event can fit in the same buffer. For example, WebRTC.

- *overwriting*: the extreme case of compaction is to overwrite old buffer data with new data; also according to an application-specific policy rule.

All policies can be composed.

11.1.7 Refactoring software with locks into lockfree operation

(TODO: replace every synchronous "mutex" area with an asynchronous stream.)

11.2 Multi core programming: from slow-CPU/fast-RAM to fast-cores/slow-RAMs

(TODO: a lot of software design "best practices" date from a time where memory was faster than CPUs and (hence) interrupts and context switching was needed; while in modern CPUs, the cores are much faster than the memory, hence caches are needed and CPUs can/must do polling; Yodaiken's reference.

Hence, (i) the event loop pattern becomes more and more relevant, and (ii) a system architecture for "the cloud" resembles very much the architecture for "deeply embedded" (cores, caches, busses, GPUs, FPGAs, DSPs,. . .) in that "memory" can always be influenced by other threads of execution. From *sharing memory* to *passing ownership of memory*, like Rust does; Yodaiken's reference 2.

The predictability of what one writes in code in a programming language is reduced by "optimizations" that *compilers* introduce behind the screens; Yodaiken's reference 3.

Cache lines typically come in 65 bytes, and only one CPU can read/write a cache line at the same time -¿ mutual exclusion at the level of "communicating" with the memory!)

11.3 Peer Mediation for realtime in-process computation, coordination, configuration and communication

Activities: hard and soft realtime for control and perception; diagnostics, logging, async IO; world model;

Task: provide all activities with the requested CPU, RAM, BUF, IO resources
Mediation for Configuration: specification of OS, middleware, event loops, buffers
Mediation for Coordination: LCSMs and Traffic Lights; wait free IPC; Heartbeat

11.4 Peer Mediation for interprocess communication

processes and COMM;

11.5 Software pattern: realtime event loop in dual-thread process

11.6 Policy: configuring processes and threads in operating systems

(TODO: do not set priorities and allocate memory inside threads, but let that be done via a configuration setting at the OS level; e.g., Linux' `control groups` and `systemd`.)

11.7 Best Practice: event loop schedules Activities, OS schedules resources

(TODO: the OS does not know anything about the application, so it should not be asked to schedule the application's Activities. That is the responsibility of the event loop design. because only there one can make the right decisions about when Activities can be pre-empted, and by which other Activities.)

11.8 Policy: framework plug-ins versus library composition

Frameworks are one of the most popular ways towards digital platforms, for three good reasons: *code reuse*, *best practice implementations*, and *tooling*. *Composability* in a framework is typically provided via so-called *plug-in* interfaces: the framework provides several places in its code base where developers can *register* their own functions, that will be called by the framework's *runtime engine* at the "right" time. However, frameworks themselves are very poorly composable via plug-ins themselves, since that interface offers only one single level of composition hierarchy. For example, it is not possible for developers to provide a function that couples the "state" in the framework at two different plug-in interfaces; common cases in robotics involve such couplings between the "control", "perception" and "monitoring" functionalities. The better approach is to replace the runtime engine part of the framework, and generate an application-specific one by (i) composition of functions and data from pure libraries, and (ii) generating the code for the application's runtime, starting from composable implementations of software architecture patterns. Examples of the latter are the event loops and the inter-process communication patterns, for which framework runtimes often have made hard immutable choices.

Chapter 12

Skill architecture for the composition of Tasks, capabilities and resources

(TODO: configure information and software architectures, to let several robots with several capabilities execute several tasks at the same time, in mutual coordination and interaction.)

Chapter 13

Models of two-arm manipulator robots

Dual-arm robotic tasks exploit the full potential of the hardware only with a realtime integration of the Tasks in the application with the Tasks in each of the arms.

This Chapter presents the application context of two-arm manipulators, with each arm being a redundant serial kinematic chain. The sketch in Fig. 4.6 makes the arms branch from a common "torso", which is itself a serial kinematic chain connected rigidly to the ground; of course, the latter connection could be to a mobile platform in itself, Chap. 14.

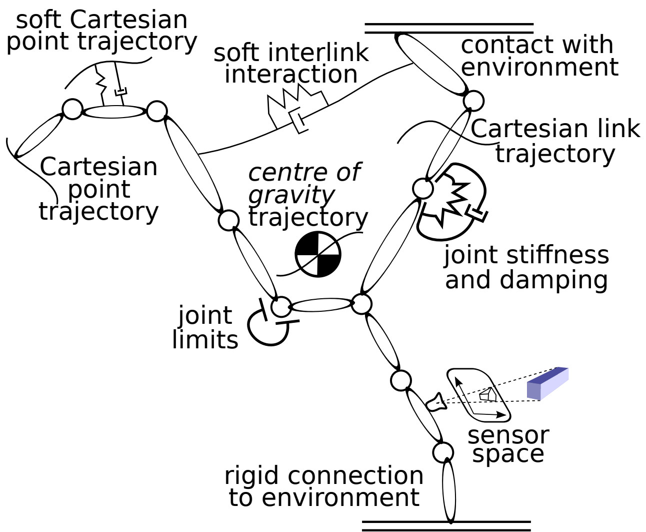

soft Cartesian point trajectory
soft interlink interaction
contact with environment
Cartesian link trajectory
Cartesian point trajectory
centre of gravity trajectory
joint stiffness and damping
joint limits
sensor space
rigid connection to environment

Figure 13.1: Sketch of a dual-arm manipulator, with all possible *"motion constraints and drivers"*.

Controlling the platform takes place at three levels of abstraction at the same time: (i) the local motion of both arms, (ii) the motion of all joints inside the arms, and (iii) the actuators.

Hence, controllers at these three levels interact in multiple ways:

(TODO: most of the chapter. . .)

Chapter 14

Models of mobile robots

Chapter 15

Models of cable robots

Bibliography

[1] JSON Schema: A media type for describing json documents. http://json-schema.org/latest/json-schema-core.html, 2018.

[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 50*, 5 (2017), 1–40.

[4] 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).

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

[6] 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.

[7] Borst, P., Akkermans, H., and Top, J. Engineering ontologies. *International Journal on Human-Computer Studies 46* (1997), 365–406.

[8] 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.

[9] Burke, W. L. *Applied differential geometry.* Cambridge University Press, 1992.

[10] Chauvel, F., and Jézéquel, J.-M. Code generation from UML models with semantic variation points. In *International Conference on Model Driven Engineering Languages and Systems* (2005), no. 3713 in Springer Lecture Notes in Computer Science, pp. 54–68.

[11] Chen, P. P.-S. The entity-relationship model—Toward a unified view of data. *ACM Transactions on Database Systems 1*, 1 (1976), 9–36.

[12] Cox, I. J., and Leonard, J. J. Modeling a dynamic environment using a Bayesian multiple hypothesis approach. *Artificial Intelligence 66* (1994), 311–344.

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

[14] D'Alembert, J. L. R. *Traité de Dynamique.* 1742.

[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] Ehrig, H., Ermel, C., Golas, U., and Hermann, F. *Graph and Model Transformation. General Framework and Applications.* Monographs in Theoretical Computer Science. Springer, 2015.

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

[19] Franchi, A., and Mallet, A. Adaptive closed-loop speed control of BLDC motors with applications to multi-rotor aerial vehicles. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Signapore, 2017), pp. 5203–5208.

[20] Frankel, T. *The Geometry of Physics.* Cambridge University Press, Cambridge, England, 1996.

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

[22] Glez-Cabrera, F. J., Álvarez Bravo, J. V., and Díaz, F. QRPC: A new qualitative model for representing motion patterns. *Expert Systems with Applications 40* (2013), 4547–4561.

[23] Goldstein, H. *Classical mechanics*, 2nd ed. Addison-Wesley Series in Physics. Addison-Wesley, Reading, MA, 1980.

[24] Golub, G., and Van Loan, C. *Matrix Computations.* The Johns Hopkins University Press, 1989.

[25] Good, I. J. A derivation of the probabilistic explanation of information. *Journal of the Royal Statistical Society (Series B) 28* (1966), 578–581.

[26] Holvoet, T., Weyns, D., and Valckenaers, P. Delegate MAS patterns for large-scale distributed coordination and control applications.

[27] Jaynes, E. T. *Probability Theory: The Logic of Science.* Cambridge University Press, 2003.

[28] Karger, A., and Novak, J. *Space kinematics and Lie groups.* Gordon and Breach, New York, NY, 1985.

[29] Kim, J. H., and Pearl, J. A computational model for combined causal and diagnostic reasoning in inference systems. In *Proceedings of the Eigth International Joint Conference on Artificial Intelligence* (Karlsruhe, Germany, 1983), pp. 190–193.

[30] Koestler, A. *The Ghost in the Machine.* 1967.

[31] 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.

[32] Lee, J., Bagheri, B., and Kao, H.-A. A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems. *Manufacturing Letters 3* (2015), 18–23.

[33] Lipkin, H., and Duffy, J. Hybrid twist and wrench control for a robotic manipulator. *Transactions of the ASME, Journal of Mechanisms, Transmissions, and Automation in Design 110* (1988), 138–144.

[34] Lončarić, J. Normal forms of stiffness and compliance matrices. *IEEE Journal of Robotics and Automation RA-3*, 6 (1987), 567–572.

[35] Madni, A. M., and Scott, J. Towards a conceptual framework for resilience engineering. *IEEE Systems Journal 3*, 2 (2009), 181–191.

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

[37] Parasuraman, R., Sheridan, T. B., and Wickens, C. D. A model for types and levels of human interaction with automation. *IEEE Transactions on Systems, Man, and Cybernetics. Part A: Systems and Humans 30*, 3 (2000), 286–297.

[38] Pearl, J. Fusion, propagation, and structuring in belief networks. *Artificial Intelligence 29* (1986), 241–288.

[39] 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).

[40] 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.

[41] Philips, J., Valckenaers, P., Aertbeliën, E., Van Belle, Jan en Saint Germain, B., Bruyninckx, H., and Van Brussel, H. PROSA and delegate MAS in robotics. In *Holonic and Multi-Agent Systems for Manufacturing*, vol. 6867 of *Lecture Notes in Computer Science.* 2011, pp. 195–204.

[42] Radestock, M., and Eisenbach, S. Coordination in evolving systems. In *Trends in Distributed Systems. CORBA and Beyond.* Springer-Verlag, 1996, pp. 162–176.

[43] Ramm, E. Principles of least action and of least constraint. *Gesellschaft f. Angewandte Mathematik und Mechanik (GAMM) 34*, 2 (2011), 164–182.

[44] Reid, D. B. An algorithm for tracking multiple targets. *IEEE Transactions on Automatic Control AC-24*, 6 (December 1979), 843–854.

[45] Saint Germain, A. d. Sur la fonction s introduite par M. Appell dans les équations de la dynamique. *Comptes Rendus de l'Académie des Sciences de Paris 130* (1900), 1174.

[46] Saridis, G. N., and Stephanou, H. E. A hierarchical approach to the control of a prosthetic arm. *IEEE Transactions on Systems, Man, and Cybernetics 7*, 6 (1977), 407–410.

[47] Schürr, A. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science*, vol. 903 of *Springer Lecture Notes in Computer Science*. 1995, pp. 151–163.

[48] 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.

[49] Scioni, E., Hübel, N., Blumenthal, S., Shakhimardanov, A., Klotzbücher, M., Garcia, H., and Bruyninckx, H. Hierarchical hypergraphs for knowledge-centric robot systems: a composable structural meta model and its domain specific language NPC4. *Journal of Software Engineering in Robotics 7*, 1 (2016), 55–74.

[50] Simon, H. *The Sciences of the Artificial.* MIT Press, 1969.

[51] Simon, H. A. Rational choice and the structure of the environment. *Psychological Review 63*, 2 (1956), 129–138.

[52] Strang, G. *Linear Algebra and its Applications*, 3rd ed. Harcourt Brace Jovanovich, San Diego, CA, 1988.

[53] The HDF Group. Hierarchical Data Format, version 5. https://portal.hdfgroup.org/display/HDF5/HDF5.

[54] Udwadia, F. E., and Phohomsiri, P. Generalized LM-inverse of a matrix augmented by a column vector. *Applied Mathematics and Computation 190*, 3 (2007), 999–1006.

[55] Valckenaers, Pauland Van Brussel, H., Bruyninckx, H., Saint Germain, B., Van Belle, J., and Philips, J. Predicting the unexpected. *Computers in Industry 62* (2011), 623–637.

[56] Valckenaers, P., Bonneville, F., Van Brussel, H., Bongaerts, L., and Wyns, J. Results of the holonic control system benchmark at KU Leuven. In *Computer Integrated Manufacturing and Automation Technology* (1994), pp. 128–133.

[57] Valckenaers, P., Van Brussel, H., Hadeli, Bochmann, O., Saint Germain, B., and Zamfirescu, C. On the design of emergent systems: an investigation of integration and interoperability issues. *Engineering Applications of Artificial Intelligence 16* (2003), 377–393.

[58] Van Brussel, H. Holonic manufacturing systems. The vision matching the problem. In *First European Conference on Holonic Manufacturing Systems* (1994), pp. 1–11.

[59] Vanthienen, D., Klotzbücher, M., and Bruyninckx, H. The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming. *Journal of Software Engineering in Robotics 5*, 1 (2014), 17–35.

[60] Vereshchagin, A. F. Modelling and control of motion of manipulational robots. *Soviet Journal of Computer and Systems Sciences 27*, 5 (1989), 29–38. Originally published in Izvestiia Akademii nauk SSSR, Tekhnicheskaya Kibernetika, No. 1, pp. 125–134, 1989.

[61] von der Beeck, M. A comparison of Statecharts variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, H. Langmaack, W.-P. de Roever, and J. Vytopil, Eds., vol. 863 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1994, pp. 128–148.

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

[63] Wirth, N. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

[64] Yunt, K. On the relation of the principle of maximum dissipation to the principles of Jourdain and Gauss for rigid body systems. *Transactions of the ASME, Journal of Computational and Nonlinear Dynamics 9* (2014), 031017–1–11.

[65] Zellner, A. Optimal information processing and Bayes's theorem. *The American Statistician 42* (1988), 278–284.