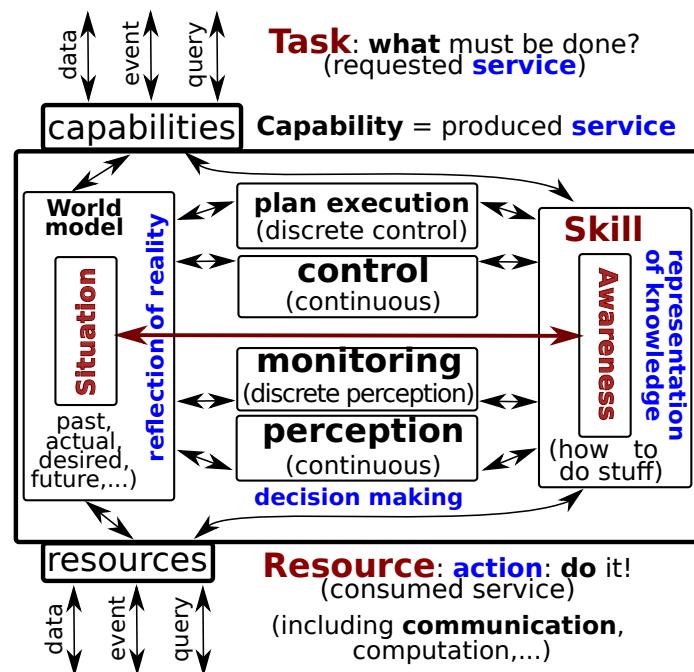


Building blocks for complicated and situational aware robotic and cyber-physical systems



Herman Bruyninckx

KU Leuven, Belgium
TU Eindhoven, the Netherlands
Flanders Make, Belgium

2022-04-22

Abstract

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

Everyone wants to build **complicated** computer-controlled systems by “just connecting” off-the-shelf components. This document targets the **architects and developers** of these systems, and has the ambition to provide them with the **science that explains**:

- how to design and implement **components**,
- in such a way that they are **composable** into any type of *system*.
- how to design and implement such a **system**,
- in such a way that the result has the **compositionality** property, that is, it is **simple** to understand, to use, to configure and to adapt, for the **application domain experts** with whom the system is **co-designed**.

In addition, application users expect ever increasing levels of **runtime adaptation** of their systems, including **runtime self-composition** (and de-composition) of systems into (temporarily active) systems-of-systems, and of their **resilience** against whatever disturbances that can occur.

A simple though not easy guided voyage over the seven Cs...

The first design objective of the system development **approach** presented in this document is **to separate mechanism from policy**: there are only *a couple of dozen* different **mechanisms** that component and system developers have to understand. This set of mechanisms represents the **fundamentally different** ways of **how “things” work**. Of course, there are orders of magnitude more ways of **how to use “things”** than there are mechanisms that are active inside these “things”.

The second design objective of this document is **to introduce two complementary types of behaviour**:

- **behaviour of components:** **Computation and Communication**, that is, to describe, design and generate **behaviour**, by *structuring algorithms* and their **data exchange**.

Although behaviour is in “cyber space”, the information that it processes is coupled, to **physical exchange of material and energy**, in various ways and with various expectations in efficiency and safety.

- **behaviour of systems:** **Coordination and Configuration**, that is, to describe, design and generate runtime **adaptations** to component behaviours, by **structuring**

the decision about (i) **which algorithms** to execute, (ii) **when**, (iii) with **which parameters**, and (iv) **via which channels** these algorithms must interact.

This adaptation of behaviour deals with **information only**, hence its impact on physical material and energy flows is always indirect, via the above-mentioned Computation and Communication behaviour.

The third design objective of this document is to extend the four above-mentioned *component-centric* C's with three *system-centric* C's:

- **Composition:** the **architecture** that put components together into a system.
- **Composability:** the **policies** that improve the efficiency and effectiveness with which a component can be put into an architecture.
- **Compositionality:** the **predictability** of the behaviour of a system based on the knowledge of the behaviour of its components.

This document's hypothesis about systems of systems design and development is that these **seven C's** must **always** be visible in every design, at any level of abstraction, any resolution and any spatial extension. The positive side-effect of the approach is its **simplicity**: it makes the **conceptual** differences between "component" and "system" disappear completely. The cost that the approach brings is that even the smallest "component" is **not easy**: it requires a non-trivial amount of design and development **efforts**.

Acknowledgements

The author remains responsible for erroneous information and claims in this document, but is very grateful for the contributions by:

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

Sven Schneider (Hochschule Bonn-Rhein-Sieg, Germany),

[Christian Schlegel](#),¹ Dennis Stampfer, Alex Lotz (HSU Ulm, Germany)

[René van de Molengraft](#),² Cesar Lopez Martinez, Wouter Houtman, Hao Liang Chen, Bob Hendrikx, Jordy Senden (TU Eindhoven, the Netherlands)

[Hans Vangheluwe](#) (Universiteit Antwerpen, Belgium; McGill University, Canada; Flanders Make, Belgium).

The author releases the document under the [Attribution-ShareAlike 4.0 International \(CC BY-SA 4.0\)](#) license. This document is a proud part of the [RobMoSys](#) ecosystem, stimulated by the European Commission under its *Horizon 2020* programme (2017–2020, grant agreement No 732410). It is an equally proud [advocate](#) of the [Wikipedia](#) ecosystem, which makes the [state of the art](#) in science and technology more accessible than the [most ambitious initiatives in academic publishing](#). Links to the [Wikipedia](#) or [Wiktionary](#) are laid out [like this](#). And internal links in this document [like this](#).

¹Christian is to be credited for the creation and maturation of many, many of the concepts, models, patterns and best practices in this book. He is also the “founding father” of, and driving force behind, the RobMoSys ecosystem.

²René is the co-creator of our *situation aware*, hence *lazy* robot task specification and control paradigm.

Summary

Holonic skill architectures for the Task-Situation-Resource paradigm

Knowledge representation, patterns, and best practices for resilient holonic data, information, and software architectures for the continuous, discrete and symbolic task specification, configuration and control of robotic and cyber-physical systems

This document introduces a **paradigm** to **model**, **design**, **implement** and **deploy** small and large scale **engineered systems**, more in particular, **cyber-physical systems**,³ and, even more specifically, **robotic systems**. Such applications are **complicated systems of components** (that reflect reality,⁴ and control that reality via computers) so that they are **composable** (their behaviour is predictable in any system they are part of), such that they can be **composed** (structurally and behaviourally) into **systems** that are **compositional** (their behaviour is predictable when their components are). The composition operation must be **fractal**, or “**holonic**” (any *composed* system becomes a *composable* system in itself, with no end towards the “top” or the “bottom” of composition) and **non-hierarchical** (a component at the “top” of a composition provides the **situation** in which *all* components “inside” the composition communicate with each other, and configure their behaviours).

The inspiration behind this design paradigm is obvious: over the last milleniums, **human society** has developed many such holonic systems, with high degrees of **robustness** against disturbances, and **resilience** to recover from disruptions and disasters. For example in its **spatial** organisation (room, house, street, area, village, city, region, country,...), its **economical** organisation (employee, team, branch, company, holding,...), or its **political** organisation (supporter, member, candidate, representative, party, democracy,...).

Engineered systems are developed to add economic value to society. That means that each of them has a set of **tasks** that it is expected to execute. This document introduces the **Task-Situation-Resource** model of interacting **skills**. A skill **composes** the **perception**, **control** and **decision making** (at continuous, discrete and symbolic levels) that a system requires:

- to realise a **task** with a given set of **resources**;
- to exhibit the **situation awareness** that is required **to adapt** to

³For example, **electrical motors** and **grids**, **ships**, **airplanes**, **looms**, **wind turbines**, **chemical** and **nuclear plants**, etc. The major difference with “purely digital” ICT platforms (e.g., distributed financial databases, social media applications, e-commerce platforms) is that cyber-physical systems directly impact (that is, “control”) the real physical world. This brings in a lot of extra constraints on its ICT components, more in particular the need for real-time feedback loops, which include physical components that come with a dynamical behaviour that has not been designed by the system engineers.

⁴This document aims to respect the history of systems design, by giving preference to seminal nomenclature. One example is the consistent use of *component* instead of the latest hype’s **buzzword** of **digital twin**.

- the **environment** in which the task is to be executed;
- the **affordances** of objects in that environment;
- the (cooperative and collaborative interaction) **sessions** via which it currently has behavioural couplings with other skills, possibly in other systems;
- and to be driven by the **sense-making** required to make **explainable and trustworthy** decisions.

The more a skill can coordinate actions and interactions over multiple **orders of magnitude in time and space**, the more value it adds to its system.

The document introduces a **couple of hundreds of concepts**, to answer a complementary set (the **5P's**) of modelling, design, implementation, deployment and configuration decisions:

- **principles, or mechanisms:** how does a *concept work*, how can it be realised, and under which names is the same mechanism known in different domains?
- **policies:** what *trade-offs* can be made in using a particular mechanism, and what are the (dis)advantages of each trade-off choice?
- (best) **practices:** in which *application-specific* situations and use cases have particular combinations of mechanisms and policies proven to work, and why?
- **patterns:** some of these *application-driven* best practices turn out to be applicable to *many* application domains, and hence this document gives them **first-class citizenship** as fundamental design models.

The **mediator** is the most important pattern in this document.

- **paradigm:**⁵ this document makes a *particular selection* of mechanisms, policies, best practices and patterns, and the result is **meant to get known** as a consistent, constructive and yet **conflict of interest-free** approach towards designing and implementing complicated systems that can adapt to the situations they have to operate in.

Each of the concepts in this document's paradigm is **simple** to comprehend, as are the ways to compose several of them together, and to connect them to software implementations. However, the *variation* with which compositions can be made, and the *scale* of the composed systems that can result from such compositions, give rise to design and implementations efforts that are anything but **easy**.

It is difficult to be more vague and unprecise than in the buzzword-loaden paragraphs above. The concepts of *knowledge*, *model*, *component*, *system*, or *design*, are so general and broad, that they can (and need) not be defined strictly or exhaustively, [1, 146]. Anyway, they all have very few identifiable properties that are present all the time. For a *component*, these **meta** properties are: *computation*, *communication*, *configuration* and *coordination*. For a *system* there is just one: *composition*. *Design* adds one more: *trade-off*. All of these latter three concepts are *by humans, and compositions of models serve as the **representation** of knowledge.*

A major quality measurement that knowledge representation and system composition share is their *compositionality*: the extent to which a system's *behaviour* is *predictable* when (i) the behaviour of its sub-systems is known, and (ii) (the *structure and behaviour* of) the interactions between the sub-systems are known. In addition, that predictability must be preserved even when (i) the number and complexity of the interactions between sub-systems

⁵Note that *paradigm* is used in the **singular** form, because there is only one in this document, albeit it a very large one. That **singleton** is the container of lots and lots of the other four *P's* in this document.

increases, (ii) each interaction introduces its own type of **uncertainty**, **volatility**, **ambiguity**, or **inconsistency**, and (iii) any interaction can make one sub-system disturb the behaviour of the other sub-systems involved in the interaction.

This document focuses on reaching compositionality for **engineered systems**—such as cyber-physical and robotics systems—whose purpose it is to **interact** with the **physical world**, and **to control** the behaviour of that interaction. Therefore, this document provides **formal models and methods** to represent **behaviours**,⁶ their **structure** and their **interactions**, at three complementary **levels of abstraction**: **continuous**, **discrete**, and **symbolic**.

Models themselves come in three complementary **levels of abstraction**, too:

- **model**: to represent “something” in the physical world of nature or engineering, or in abstract world of concepts.
- **meta-model**: to represent the **well-formedness constraints** that a model must satisfy in order to be acceptable.
- **meta-meta-model**: to represent the overlaps between meta-models, [98, 159] so that **model-to-model transformations** can be automated, and the **composition** of models can be reasoned about.

A major, if not *the* major design driver behind “good modelling” is to find the “right” balance between:

- **decoupling**: put only those entities and relations in a model that “*really belong together*”. In other words, for which **it makes no sense to make smaller models**, because any usage of those smaller models would always be using the larger model in its entirety too. The quotation marks above reflect the fact that there are, to the best of the authors’ knowledge, no *first principles* from which to derive what the granularity of a model should be.

Examples of **well-decoupled** knowledge representations are (or, rather, could be): the **taxonomies** of **traffic signs**, **road junctions**, and **vehicles in traffic**.

- **coupling** (“composition”): decoupled pieces of knowledge seldom add much value in themselves, and neither does their random combination. So, the “art” of knowledge representation is on finding the “right” relations between two or more of the above-mentioned decoupled models. Also here, there are no *first principles* to start from.

The example of **well-coupled** pieces of knowledge, in the context of the taxonomy examples above, is the model of the **traffic code**: it couples the mentioned taxonomies with (the knowledge to select the “right”) **driving skills**.

Robotics and cyber-physical systems seldom work in isolation from **interactions** with the world around them. These interactions can be are undesired **disturbances** from the “outside world”, to which a system’s behaviour must be **robust** and even (**gracefully**) **fault tolerant**. In addition, a system must **coordinate** the (desired!) interaction between two or more of its own loosely coupled sub-systems, to allow them to form a (possibly time-varying) **coalition**. The purpose of such a coalition is that:

- they **cooperate** to let each of them realise its own **tasks** while **sharing resources** with the other sub-systems. In doing so, each member of the coalition also helps other members to achieve (part of) their goals.

⁶Or *function*.

- they **collaborate** to realise a **shared task**, with their system as beneficiary.

Non-cooperative coordination is often also required, because *other* systems act **competitively**, **adversarially**, or even **hostilely** towards the task goals and resource needs of the system.

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

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

System architectures come in two complementary versions: **information** and **software** architectures. This document advocates to focus the system design efforts on the information architecture, because this is where composability and compositionality are made, or broken. And because a (formalized) information architecture is also the best possible **documentation** for a software architecture, the design and implementation efforts for the latter are drastically reduced. In the same vein, the document advocates (i) to focus on the **reuse** and **consolidation** of information architectures (and even their **standardisation** in the case of data and communication protocols), and (ii) not to be afraid to replace existing software implementations or architectures with alternative or better ones.

Systems and **systems theory** apply to social, economical, cultural contexts, and many more. In the societal context, composability and compositionality of systems have, over the centuries, become a primary design property,⁷ because society needs solutions that are **robust** and even **resilient** against a large number of disturbances. In the “engineering” contexts however, most focus has been on **modularity**, that is, to make components that can be reused in different system compositions. However, modularity has become too much of a goal in itself, to the detriment of (i) composability, and even worse, of (ii) compositionality.

⁷Unfortunately, most politicians have forgotten *why* some systems are composable or compositional, and create new systems that fail miserably, run far over budget, are not predictable, hence also not manageable, etc.

Contents

Abstract	2
Acknowledgements	4
Summary	5
1 Foundations of knowledge-based systems: ((meta) meta) modelling	22
1.1 Models to represent knowledge in science and engineering	23
1.1.1 Science reflects reality — Engineering reflects decision making	24
1.1.2 Levels of abstraction: scope of relevant entities and relations	24
1.1.3 Three dimensions of modelling: abstraction, scale, and complexity	26
1.1.4 Levels of modelling formalization	26
1.1.5 Levels of decision making: context of intentions	27
1.1.6 The mother of all association hierarchies: the DIKW pyramid	28
1.1.7 Explainability: decisions based on explainable relations and facts	28
1.1.8 Mechanism (“what it does”) and policy (“how to use it”)	29
1.1.9 The various meanings of “meta”	30
1.1.10 Paradigm, pattern, best practice	30
1.1.11 Simple, though not easy — Simplicity versus simplisticness	31
1.2 Knowledge representation: from entities to meaning	32
1.3 Knowledge representation: structure	33
1.3.1 Mechanism: set, list, tree, graph — Connecting entities in structures	33
1.3.2 Textual representations — Ordered lists and trees	34
1.3.3 Representation standards for structure: JSON-LD and RDF	35
1.3.4 Bad practice: implicit structural order implied by lexical or graphical order	38
1.3.5 Policy: graph languages with less semantics	38
1.3.6 Pattern: structural constraints on collections as array, dictionary	39
1.3.7 Policy: RDF graph, factor graph	39
1.4 Knowledge representation: relation	41
1.4.1 Mechanism: property graphs	41
1.4.2 Policy: first-order and higher-order relations	41
1.4.3 Policy: properties and attributes	43
1.4.4 Reification — Best practice of <i>attributes are properties of relations</i>	44
1.5 Knowledge representation: meaning	44
1.5.1 Best practice: M0–M3 structure to represent meaning	44
1.5.2 Policy: higher-order relations as context	47
1.5.3 Further examples of higher-order relations	47
1.5.4 Observation: “higher-order” is not necessarily “more abstract”	49
1.5.5 Facts	49
1.5.6 Declarative and imperative models of behaviour	49
1.5.7 Hierarchical and serial ordering: scope	50

1.5.8	Magic number: attribute in a higher-order causality relation	51
1.5.9	Don't care: attribute in a higher-order causality relation	51
1.6	Core relations: <code>conforms-to</code> and <code>is-a</code>	51
1.6.1	The <code>conforms-to</code> relation	52
1.6.2	The <code>is-a</code> relation	52
1.6.3	Composition and inheritance	52
1.6.4	Best practice: four types of <code>is-a</code> and <code>conforms-to</code> hierarchies	53
1.7	Mereo-topology: most abstract representation level	53
1.7.1	Mereology: <code>has-a</code>	54
1.7.2	Topology: <code>connects</code> , <code>contains</code>	54
1.7.3	Role of mereo-topological models	55
1.8	Patterns in knowledge representations	56
1.8.1	Pattern: metadata of a model — <code>Semantic_ID</code>	56
1.8.2	Pattern: entities-relation-constraints-tolerances	57
1.8.3	Pattern: taxonomy and frame as knowledge components	58
1.8.4	Pattern: behavioural constraints on collections as queue, stack, and stream	59
1.8.5	Mechanism: querying & reasoning via graph matching & traversal	59
1.8.6	Mechanism: <code>Block-Port-Connector</code>	60
1.8.7	<code>Blocks as Components</code>	61
1.8.8	Policy on <code>block-port-connector</code> mechanism: data sheets	62
1.8.9	Storage and reasoning in property graph databases	62
1.8.10	Policy: constraint, dependency and causality graphs	63
1.8.11	Bad practice: modelling a dependency with a <code>has-a</code> relation	63
1.8.12	Pattern: composition of relations as fact, model, instantiation	64
1.9	Textual representations of knowledge	65
1.9.1	Textual representation of meaning — Labelled and named graphs	65
1.9.2	Textual representation of meaning — Context and composition	65
1.9.3	Textual representation of meaning — Type and <code>conforms-to</code>	66
1.9.4	Textual representation of constraints	68
1.9.5	Policy: domain-specific language (DSL)	68
2	Meta models for behaviour: activities, their interaction and coordination	69
2.1	Cyber-physical systems: interaction of matter, energy, information & data	70
2.1.1	Physics: coupling of space, time, spectrum, matter and energy	71
2.1.2	Cyber: decoupling of continuous, discrete and symbolic models	71
2.1.3	State of a system	73
2.1.4	State representation: ordinal, categorical, continuous, discrete, event	73
2.1.5	World representation: open world, closed world, world model	74
2.1.6	Behaviour follows structure: list, tree, graph	74
2.1.7	Best practice: cope with the complexity of the open world	74
2.2	Data: structuring information for computations	75
2.2.1	Mathematical representation	75
2.2.2	Abstract data type — Semantic representation	75
2.2.3	Data structure — Symbolic representation	76
2.2.4	Digital structure — <code>Data representation</code>	76
2.2.5	Electronic realisation — <code>Physical representation</code>	77
2.2.6	Symbolically linked data: metadata, database, query, index	77
2.3	Algorithm: synchronous composition of functions	78
2.3.1	Mereo-topological model — Schedule and Dispatch	79
2.3.2	Mechanism: <code>data</code> , <code>function</code> , <code>control_flow</code> , and <code>algorithm</code>	80
2.3.3	Block-Port-Connector model for blocks	80
2.3.4	Data access constraints	81
2.3.5	Directed Acyclic Graph: declarative model of control flow dependencies	82

2.3.6	Policy: closure in a context	83
2.3.7	Policy: application programming interface	83
2.4	Event loop: synchronous composition of computations with asynchronous data	83
2.4.1	Event loop: event part and loop part	84
2.4.2	Synchronous, asynchronous, sequential, concurrent, parallel	85
2.4.3	Mechanism: 4Cs-based event loop template	86
2.4.4	Composition of event loops	87
2.5	Activity: asynchronous composition of algorithms and their interactions	87
2.5.1	Activities are responsible for time-based behaviour	88
2.5.2	Mereo-topological meta model	88
2.5.3	Asynchronicity implies stateful activity interfaces	89
2.5.4	Port in algorithm (service) versus Port in library (API)	90
2.5.5	Continuous data, discrete event, logic flag, symbolic query	90
2.5.6	Best practice: resource ownership	91
2.6	Finite State Machine: behaviour coordination in one algorithm or activity	92
2.6.1	Mechanism Part 1: state to model coordinated behaviour	92
2.6.2	Mechanism Part 2: events to model coordination changes	93
2.6.3	Mechanism Part 3: transition and event reaction table to model coordination behaviour	94
2.6.4	Canonical meta model of a Finite State Machine	94
2.6.5	Policy: event composition	95
2.6.6	Policy: event distribution and conversion — Event queue	96
2.6.7	Policy: hierarchical states	96
2.6.8	Policy: Life Cycle State Machine of activity or resource	97
2.6.9	Policy: don't care, history, timeout	99
2.6.10	Policy: selection, priority, deletion	99
2.6.11	Implicit constraints on event handling meta model	99
2.6.12	Runtime creation of FSMs via transition systems	100
2.7	Flags and bitfield protocols for peer-to-peer coordination	100
2.7.1	Abstract data type: flags	101
2.7.2	Mechanism: protocol array with atomic iterator	102
2.7.3	Best practice: coordination mediator	103
2.7.4	Policy: big endian versus little endian flag arrays	103
2.7.5	Stop-and-go coordination — Barrier	103
2.7.6	Two-phase commit coordination	104
2.7.7	Data borrowing coordination	104
2.7.8	Higher-order flags: active, timeout, highwater-lowwater	104
2.7.9	Race condition: Time of check to time of use	105
2.8	Petri Net: algorithm/activity coordination via mediator	105
2.8.1	Examples of multi-algorithm coordination	106
2.8.2	Role of algorithm coordination in systems-of-systems	107
2.8.3	Mechanism: place, token, transition	107
2.8.4	Representation: marking reaction table	109
2.8.5	Pattern: Petri Net with one flag array per coordinated interaction	109
2.8.6	Policy: flag-to-place-to-flag naming transformations	110
2.8.7	Flag arrays as linear Petri Nets	111
2.8.8	Stop-and-go coordination — Barrier	111
2.8.9	Any-of and All-of coordination	112
2.8.10	All-to-go, one-to-stop coordination	112
2.8.11	Fork-and-join coordination	113
2.8.12	Par-Seq-Join coordination	113
2.8.13	Transaction coordination: multiple peers with synchronous access to data	114
2.8.14	Policy: hierarchical Petri Net	115

2.8.15	Policy: multi-token places with cardinality	116
2.8.16	Policy: coloured Petri Net for typed coordination decision making	116
2.8.17	Policy: Net-wide constraints	116
2.8.18	Promises, Await-Async patterns modelled as Petri Nets	117
2.8.19	Best practices in Petri Nets	117
2.9	Stream: peer-to-peer interaction via asynchronous exchange of data	118
2.9.1	Streams in the real world	119
2.9.2	Producer-Consumer protocols	120
2.9.3	Canonical meta model of a Stream: producer-buffer-consumer	121
2.9.4	Policy: flags for producer-consumer self-coordination	123
2.9.5	Policy: index for producer-consumer self-coordination	123
2.9.6	Policy: Petri Net for request and commit coordination	124
2.9.7	Policy: request-less data chunks	124
2.9.8	Mechanism: composition of streams — Stream-to-Stream	124
2.9.9	Mechanism: composition of streams — Adding transformer peers	125
2.9.10	Best practice: Life Cycle State Machine of a Stream — Session	125
2.9.11	Policy: lossless, missing data, messages, bags	125
2.9.12	Policy: multiple producer—multiple consumer stream	126
2.9.13	Pattern: composition of data and metadata streams	127
2.9.14	Pattern: Submission-Completion streams — Task queues	127
2.9.15	Pattern: journaled CRUD interfaces	128
2.9.16	Policy: Send-and-Forget, Publish-Subscribe, Request-Reply	130
2.10	Roles in Coordination and Configuration	131
2.10.1	Petri Nets, Finite State Machines, flags and protocol arrays	131
2.10.2	Events and streams	132
2.10.3	Declarative and imperative behaviour models	133
2.10.4	FSM as boundary case of single-token Petri Net	133
2.10.5	Policy: higher-order model for pre/per/post conditions	133
2.10.6	Good and bad practices	134
2.10.7	Design similarities and differences	134
3	Meta models for point and polygon geometry	135
3.1	Geometric spaces and their spatial relations	135
3.1.1	Semantic IDs for geometry in 1D, 2D and 3D	137
3.1.2	Geometric relations in projective, affine and Euclidean spaces	138
3.1.3	Non-Euclidean space for rigid bodies	138
3.1.4	Projections in geographical coordinate system	139
3.1.5	Qualitative spatial relations — Symbolic geometric chains	139
3.1.6	Taxonomy of geometric meta models	140
3.1.7	Levels of representation in geometry	143
3.2	Mechanism: point, polyline and polygon models	143
3.2.1	Point entity and its composition relations	145
3.2.2	Extra composition relations in 3D	148
3.2.3	Position and Motion relations of a Point	149
3.2.4	Position and Motion relations of a Rigid_body	150
3.2.5	Constraint relations on mereo-topological entities	153
3.2.6	Abstract data types for Coordinates of position and motion	154
3.2.7	Metadata for coordinates	157
3.2.8	Operators on coordinates	158
3.3	Coordinates — Uncertainty	158
3.3.1	Standards for coordinates	159
3.3.2	Uncertainty in geometric entities and relations	161
3.3.3	Covariance of a Frame has no meaning	161

3.4	Compositions relations — Chains and maps	162
3.4.1	Geometric chain: constraint relations on geometric entities	162
3.4.2	Kinematic chain: engineered mechanical constraints	163
3.4.3	Map: collection of geometric entities	163
3.4.4	Semantic map: map with meaning	164
3.5	Differential geometry as meta meta model — Manifold, (co)tangent space, linear forms	164
4	Meta models for a kinematic chain and its instantaneous dynamics	167
4.1	Kinematic chain	167
4.1.1	Mereo-topology	168
4.1.2	Types	168
4.1.3	Coordinates and behavioural state of a chain's motion	170
4.1.4	Geometrical operations — Forward, inverse and hybrid kinematics	171
4.1.5	Inconsistency, redundancy, singularity	173
4.2	Rigid body dynamics — Composition of force and motion	173
4.2.1	Abstract data types and data structures for dynamics	173
4.2.2	Motion as result of constrained optimization — Gauss' Principle	174
4.2.3	Coordinates for dynamics state	174
4.2.4	Coordinates and behavioural state for impedance	175
4.2.5	Geometrical operations revisited — The role of “virtual dynamics”	175
4.2.6	Dynamic relations under a 5D motion constraint	176
4.2.7	Energy relations — Bond Graphs	177
4.3	Kinematic chain dynamics — Instantaneous motion in a dynamic chain	178
4.3.1	Gauss' Principle of Least Constraint	178
4.3.2	Specification of instantaneous motion	179
4.3.3	Operations: forward, inverse and hybrid dynamics transformations	180
4.4	Composition and decomposition of kinematic chains	181
4.4.1	Composition — Serial, tree, graph	181
4.4.2	Decomposition — Spanning tree	183
4.4.3	Policy: iteration via sweeps	183
4.4.4	Policy: input-output causality assignment	183
4.5	Taxonomy of kinematic chain families	184
4.5.1	Serial chains	184
4.5.2	Mobile platform chains	184
4.5.3	Parallel chains	186
4.5.4	Multirotor chains	186
4.5.5	Hybrid chains	186
4.5.6	Cable-driven chains	186
4.6	Taxonomy of motion capabilities	186
4.7	Solver meta model for hybrid kinematics and dynamics	187
4.7.1	Mechanism-I: motion drivers	188
4.7.2	Mechanism-II: procedural sweeps	189
4.7.3	Policy: scheduling options in the third sweep	191
4.7.4	Policy: free-floating base	191
4.7.5	Policy: tasks in the mechanical domain	191
4.7.6	Policy: serial kinematic chain	192
4.7.7	Policy: branched kinematic chain	192
4.7.8	Policy: kinematic chain with a loop	192

5 Meta models for dynamic world models, semantic maps and situational awareness	193
5.1 From simple geometrical entities to world models	194
5.2 Mechanism: Semantic map, Situation, Level of Detail	195
5.2.1 Semantic map	196
5.2.2 Situation	199
5.2.3 Level of Detail, Level of Development (LOD)	199
5.3 Spatial association hierarchies	199
5.3.1 Location association hierarchies	200
5.3.2 Spatio-temporal association hierarchies — Coverage	201
5.3.3 Association hierarchy of actions in a situation	201
5.3.4 Association hierarchy in navigation	201
5.3.5 Association hierarchy in manipulation	202
5.4 Horizontal and vertical composition of situations	202
5.4.1 Mini-meso-macro motion situations	202
5.5 World model activities	203
5.5.1 Blackboard	203
5.5.2 Semantic database	203
5.6 Stable states in robotic applications	204
5.7 Best practices	204
6 Meta models for tasks	206
6.1 Task models — What, why, and how well?	207
6.1.1 Task model association hierarchy	207
6.1.2 Specification model	208
6.1.3 Quality of progress model	208
6.1.4 Situation model	208
6.1.5 Coordination model	208
6.1.6 Driver model	209
6.1.7 Capability model	209
6.2 Hierarchies in resources imply hierarchies in task models	209
6.2.1 Spatial hierarchies — Location	210
6.2.2 Execution resource hierarchies — Detail	210
6.2.3 Strategic, tactical and operational task levels	211
6.2.4 Perception capability hierarchy — Proprio-, extero- and carto-ception	212
6.2.5 Motion capability hierarchy — From power to semantic waypoints	213
6.3 Single-level task specification: instantaneous motion of one robot	214
6.3.1 Partial force and acceleration specifications in multiple reference points	214
6.3.2 Partial velocity specifications in multiple reference points	217
6.3.3 Reactive task specification — I. Guarded motion	217
6.3.4 Reactive task specification — II. Modulated motion	218
6.3.5 Reactive task specification — III. Adaptive motion	218
6.3.6 Reactive task specification — IV. Progress monitoring	218
6.3.7 Textual model of guarded motion specifications	219
6.4 Two-level specification: guarded execution of an order	220
6.5 Context aware tasks: Task-Situation-Resource and Skill	221
6.5.1 Mereo-topological model	221
6.5.2 Task execution activities: Skill and World model	222
6.5.3 Reality reflection: capability, resource, world model, perception, monitoring . .	223
6.5.4 Decision making: plan, control, knowledge-based reasoning	224
6.5.5 Interactions: world model, skill, perception and control activities	224
6.5.6 Affordances: knowledge replaces computations	225
6.5.7 Lazy skills	225

6	Composition of motions	
6.6	Horizontal task composition	226
6.6.1	More tasks — Multi-tasking	226
6.6.2	Different tasks, shared world — Co-existence	226
6.6.3	Same tasks, multiple robots — Distribution	226
6.6.4	Shared control between robot and human	227
6.7	Vertical composition: resources become capabilities	227
6.7.1	Navigation	229
6.7.2	Mobile manipulation	231
6.7.3	Dual-arm manipulation	232
6.7.4	Eye-hand coordinated manipulation	233
6.7.5	Best practice: composable specifications	234
6.7.6	Policy: redundancy dependencies in guarded motion compositions	235
6.8	Guarded motions involving physical contacts	235
6.8.1	Guarded motion: <code>move-to-contact</code>	235
6.8.2	Guarded motion: <code>Align edge to surface</code>	236
6.8.3	Guarded motion: <code>Align surface to surface</code>	236
6.8.4	Guarded motion: <code>Align block in corner</code>	236
6.8.5	Guarded motion: <code>Track vertex over surface</code>	237
6.9	Trajectory generation versus trajectory specification	237
6.10	Mechanism: specification as constrained optimization, satisfaction and reasoning	238
6.10.1	Specification in the continuous domain: hybrid constrained optimisation	239
6.10.2	Specification in the discrete domain: constraint satisfaction	241
6.10.3	Specification in the symbolic domain: knowledge-based reasoning	241
6.10.4	Policy: task requirement as objective function or as constraint	241
6.10.5	Policy: dimensionality reduction through task constraints	242
6.11	Composition of guarded motions — Situations	242
6.11.1	Situation: composition of specification–world model–coordination	242
6.11.2	Situations versus templates	244
6.12	Task specification, data sheet, contract, responsibility and commitment	245
6.13	Taxonomy of action, actor, actant, activity, agent	245
6.14	Bad practices in task specification	246
6.15	Use case: semantic indoor navigation (revisited)	247
6.15.1	Geometric world models with semantic tags for control & perception	247
6.15.2	Specification of an activity to realise a task specification	250
6.15.3	Example task plan: dock to cart in next junction left	252
6.15.4	Example task plan: escape from a room	254
6.15.5	Platform Independent/Specific Model, Domain-Specific Language	258
6.15.6	Task ontology and its standardization	258
6.15.7	Semantic mobile robot motion primitives	258
6.15.8	Semantic robot arm motion primitives	259
7	Meta models for continuous and discrete control	260
7.1	Mereology: feedback, feedforward, predictive, adaptive, preview, cognitive	261
7.1.1	Policy: adaptive and preview control	262
7.1.2	Policy: cognitive control	263
7.2	Mechanism: linear control — PID and pole placement	263
7.2.1	Linear control properties	263
7.2.2	The role of PID and linear control in robotics	264
7.2.3	Situations where linear control is insufficient	265
7.3	Information associations in control: mechanics and tasks	266
7.3.1	Mechanical domain control hierarchy	266
7.3.2	Task domain control hierarchy	266
7.4	Mechanism: state and system dynamics relations	267

7.4.1	Composition	269
7.4.2	Time-triggered and event-triggered control schedules	270
7.4.3	Cascaded control loops	270
7.4.4	Composition constraints	272
7.5	Mechanism: optimal control specification	273
7.5.1	Policy: “good enough”, or satisficing control	274
7.5.2	From PID and pole placement to optimal control	274
7.5.3	PID alternatives: coping with its limitations	275
7.5.4	ABAG control (Adaptive Bias, Adaptive Gain)	276
7.5.5	Policy: model-reference adaptive control (MRAC)	277
7.5.6	Policy: model-predictive control (MPC)	277
7.6	Mechanism: setpoint, trajectory, path and tube inputs	278
7.6.1	Policy: composition of optimal control and tube inputs	278
7.6.2	Policy: control progress objective or constraint	279
7.7	Mechanism: asynchronous distributed control	279
7.8	Mechanism: behaviour tree for semi-optimal control	280
8	Meta models for perception and its integration in tasks	282
8.1	Information associations in perception	283
8.1.1	Pre-processing	283
8.1.2	Pre-processed sensor data association to sensor feature	283
8.1.3	Sensor feature association to object sensor features	284
8.1.4	Object sensor feature association to object feature	284
8.1.5	Object feature association to object association	284
8.1.6	Association to task, environment and robot context	284
8.1.7	Association of a mission with its tasks, environments and robots	285
8.2	Mereo-topological meta model: the natural hierarchy in robotic perception	285
8.3	Mereo-topological meta model: natural hierarchy in world representation	286
8.4	Policy: (data) association	286
8.5	Mechanism: goodness of fit, similarity, template matching	286
8.6	Perception example: robots driving in traffic	287
8.6.1	Escape from a room	287
8.6.2	Ego-motion estimation with accelerometer, gyro and encoder	288
8.6.3	Ego-motion estimation with visual point and region features	288
8.7	Probability: composition of data and uncertainty	288
8.7.1	Mechanism: Bayesian probability axioms	289
8.7.2	Mechanism: Bayes’ rule for optimal transformation of data into information .	292
8.7.3	Policy: belief propagation	292
8.7.4	Policy: hypothesis tree for semi-optimal information processing	292
8.7.5	Policy: mutual entropy to measure change in information	292
8.8	Geometrical semantics in perception	292
8.9	Dynamical semantics in perception	292
8.10	Policy: tracking, localisation, map building	292
8.11	Mechanism of information update: Bayes’ rule	293
8.12	Mechanism of perception solver: message passing over junction trees	294
8.13	Policy: dynamic Bayesian network	294
8.14	Policy: Factor Graphs	296
8.15	Mechanism: composition of point and region features	296
8.16	Policy: feature pre-processing	296
8.17	Policy: deployment in event loop	296

9 Architectural patterns	297
9.1 5C component: to compose interaction heterarchy with coordination hierarchy	297
9.1.1 5C's: composition of 4C behavioural roles into one component	298
9.1.2 Component = activity with 5C-based architecture	301
9.1.3 Types of state: state, status (flag), and their changes (event)	302
9.1.4 Computation: monitoring, coordinating, mediation, scheduling, dispatching . .	303
9.1.5 Coordination: coordinated, orchestrated, choreographed	304
9.1.6 Communication: property & attribute, parameter & stream	304
9.1.7 Architecture of Finite State Machines in an activity	305
9.1.8 Best practice: separation of mechanism and policy — System composition . .	305
9.1.9 Policy: vendors add value in Configuration, Coordination, Composition . . .	306
9.1.10 Good and bad practices in deployment of coordination	306
9.1.11 Bad practice: interpreting attributes as properties	307
9.1.12 Vertical and horizontal composition	307
9.1.13 Lifecycle of compositions	307
9.1.14 Block-Port-Connector for components, activities and functions	308
9.2 Event handling association hierarchy for Configuration and Coordination — Document Object Model	308
9.2.1 Mechanism: DOM model structures the decision making	309
9.2.2 Policy: decision making in a context	312
9.2.3 Policy: syntax, custom tag, paths, diffs and multi-tree	312
9.2.4 Cascading Style Sheets (CSS) represent top-down parameter configuration . .	313
9.2.5 Event bubbling represents bottom-up reactivity	313
9.2.6 Petri Net coordination for event bubbling	314
9.2.7 Event capturing represents top-down reactivity	315
9.2.8 Petri Net coordination for event capturing	315
9.2.9 Components are not <i>in</i> a DOM, but their behaviour is <i>represented</i> by it	316
9.2.10 Graph connections as constraints between DOM model trees	317
9.2.11 DOM models for activities, algorithms, functions, data	317
9.2.12 DOM model of Block-Port-Connector relation	320
9.2.13 DOM model for relation-constraint-tolerance models	320
9.2.14 DOM model for a guarded motion specification	321
9.3 Activity to execute Configuration and Coordination — Mediator	323
9.3.1 The mediator pattern in human society	324
9.3.2 Meta model of a mediator	325
9.3.3 Mediator for a data exchange stream	326
9.3.4 Mediator as a DOM-coordinating DOM	327
9.3.5 Mediator for events, flags, protocols, Petri Nets, FSMs	327
9.3.6 Mediator architecture to coordinate interacting activities	329
9.3.7 Mediator components in a system's life phases	330
9.3.8 Policy: extend a mediator into a broker	331
9.3.9 Mechanism: reification of a Connector and its Ports	332
10 Information architectures for skill-based execution systems	333
10.1 Methodology to design information architectures	334
10.1.1 Analysis and synthesis — From requirements to specifications	334
10.1.2 Policy: ownership, loose coupling, semantics	335
10.2 Situation-aware skill architecture	335
10.2.1 Task execution via constrained optimization solving	336
10.2.2 Situational aware architecture in the “edge”	338
10.2.3 Sense-Plan-Act, Three-Layered, Behaviour, Subsumption	341
10.2.4 Task-Situation-Resource versus Sense-Plan-Act	341
10.3 Execution system architectures	341

10.3.1	Manipulation execution architecture	341
10.3.2	Manufacturing execution architecture	341
10.4	Best practices in activity–stream architectures	342
10.4.1	Interact via requests instead of via commands	342
10.4.2	One Life Cycle State machine per activity	342
10.4.3	Every entity and relation has one explicitly identified owner activity	343
10.4.4	Explicit causality in data access policy	343
10.4.5	Every task is a shared resource	343
10.5	Minimal latency data exchange — Double and triple buffers	344
10.6	Task queue and worker pool — Configuration and Coordination for task execution	347
10.7	Coordination and Configuration for runtime creation activities — Factories	349
10.7.1	Discovery, connection, session, configuration, flow control	349
11	Software architectures for components	350
11.1	Single-process 5Cs component: hierarchy of threads, activities and algorithms	351
11.1.1	Process: main thread owns interactions with operating system	352
11.1.2	Thread: executing an application via activity schedules	357
11.1.3	Activity: coordinating the execution of algorithmic functionalities	359
11.1.4	Algorithm: executing the functionalities	360
11.2	Single-process component: policies and best practices	360
11.2.1	Process with realtime behaviour: synchronous thread	360
11.2.2	Event loop design trade-offs	361
11.2.3	Application mediation activity	362
11.2.4	Support of operating systems	362
11.2.5	Mediation on shared resources	362
11.2.6	Event loop with ring buffer and scatter-gather I/O	363
11.2.7	Best practice: multi-rate time series streams	363
11.3	Mechanism: stream as a ring buffer byte array	364
11.3.1	Data structures: array, port, index	365
11.3.2	Mechanism: contiguous data for producer and for consumer	365
11.3.3	Algorithms	366
11.3.4	Life Cycle State Machine	367
11.3.5	Protocol flags for producer-consumer-owner coordination	368
11.3.6	Policy: late binding of data chunk type	368
11.3.7	Policy: composition of data and metadata	368
11.3.8	Policy: time series	369
11.3.9	Policy: event handling synchronisation	370
11.3.10	Policy: composition of stream and memory pool	370
11.3.11	Policy: <code>heartBeat/watchDog</code> mediation for “lazy” stream	370
11.3.12	Standards and software projects supporting stream channels	371
11.4	Mechanism: stream based on a finite buffer	371
11.4.1	Mechanism: producer–consumer ring buffer	371
11.4.2	Policy: ring buffer without a mediator	373
11.4.3	Policy: status flags for activity and backpressure	373
11.4.4	Policy: ping-pong buffer — Synchronization with data payload	374
11.4.5	Policy: triple buffer — Wait-free and last-update only	375
11.4.6	Policy: heterogeneous data chunks — Message passing	376
11.4.7	DOM model for hierarchy in dependencies between activities	377
11.4.8	DOM model for hierarchy in dependencies between heterarchical streams	377
11.5	Architecture to compose task queues, workers, event loops, and solvers	377
11.5.1	Pattern: composition of “task queue” and “library API” patterns	378
11.5.2	Policy: frameworks, middleware, solvers	379
11.5.3	Composition of stream with object pool	379

11.5.4 Pattern: schedule-based event handling in an event loop	379
11.5.5 Pattern: splitting a solver into scheduling and dispatching	380
11.6 Best and bad practices	381
11.6.1 Best practices	381
11.6.2 Composable configuration in compilation, linking, launching and execution	382
11.6.3 Framework plug-ins (bad) versus library composition (good)	382
11.6.4 Bad practices in robustness	383
11.6.5 Bad practices in resource locking	384
11.6.6 Bad practices in communication	384
11.7 Mechanisms: close to the hardware	385
11.7.1 Mechanism: two-way mapping between symbolic IDs and memory addresses	385
11.7.2 Mechanism: memory barriers — Asynchronicity in compilers & CPUs	385
11.7.3 Atomic and lockfree operators	386
11.7.4 Conflict-Free Replicated Data Type (CRDT)	387
11.7.5 Immutable data type	387
11.7.6 Bitfields for flags, FSMs and Petri Nets	387
11.7.7 Buffer, queue, socket	388
11.7.8 Async/await	390
11.7.9 Iterator	390
11.7.10 Maybe	390
12 Software architectures for systems	391
12.1 Architectures for activity deployment	391
12.1.1 Hardware hierarchy: core, system-on-a-chip, computer, edge, cloud	391
12.1.2 Software hierarchy: runtime, thread, process, shell, container, cloud	392
12.2 Systems with 5Cs architecture	394
12.2.1 Single 5Cs process: activities with shared memory interactions	394
12.2.2 Single 5Cs process with multiple-rate activity interactions	395
12.2.3 Multiple 5Cs processes with IPC interactions on one CPU	395
12.2.4 Device: multiple 5Cs processes on multiple CPUs	396
12.2.5 Edge: multiple devices with application-controlled cooperation	396
12.2.6 Cloud: multiple applications with server-controlled cooperation	396
12.3 Modelling languages: mature implementations & tools	396
12.3.1 QUDT and UCUM	397
12.3.2 JSON and JSON-Schema	397
12.3.3 JSON-LD	398
12.3.4 RDF1.1	398
12.3.5 Abstract Syntax Notation One (ASN.1)	399
12.3.6 Hierarchical Data Format — HDF5	399
12.3.7 FlatBuffers, Protocol Buffers, Apache Arrow	399
12.3.8 Common Trace Format — CTF	399
12.3.9 BLAS, LAPACK	399
12.3.10 DFDL	400
12.3.11 XML Schemas	400
12.3.12 PROV-O provenance ontology	400
12.3.13 Functional Mockup Interface (FMI)	400
13 Information and software architectures for robotic systems	401
13.1 Form factors for robotic systems — Pod, flock, fleet	401
13.2 Pod architecture for single-process/single-device motion control	403
13.2.1 Simplest pod architecture: proprio-ceptive control	405
13.2.2 Mechanism: hybrid event control	406
13.2.3 Mechanism: cascaded control loops and their DOM models	407

13.2.4	Pod architecture for extero-ceptive control	407
13.2.5	Pod architecture for carto-ceptive control	407
13.2.6	Policy: trade-offs between throughput and latency	409
13.2.7	Policy: real-time activities via the “multi-thread” software pattern	409
13.3	Pod architecture for Task-Situation-Resource skills	411
13.4	Flock architecture for multi-device task execution	411
13.4.1	Flock architectures	411
13.5	DOM models for geometry in robotics	411
13.5.1	DOM model for polygonal shapes	412
13.5.2	DOM model for coordinates	413
13.5.3	DOM model for geometric and kinematic chains	413
13.5.4	DOM model for shape and inertia kinematic chain links	416
13.5.5	DOM model for actuator and transmission in kinematic chain	417
13.6	DOM model for numeric, symbolic and semantic maps of world, building, and floors	418
13.6.1	DOM model for semantic localisation and navigation	419
13.6.2	DOM model for motion tasks	419
13.6.3	DOM model for instantaneous motion specification in kinematic chains	419
13.7	Architecture for a highly overactuated mobile robot	419
13.7.1	The eight-wheel drive mobile platform	420
13.7.2	Topological navigation and metric motion tasks	421
13.7.3	Platform navigation and motion modes	423
13.7.4	2WD force transmission	424
13.7.5	2WD efficiency of force transmission	425
13.7.6	Platform pushing is (locally) stable, pulling is unstable	427
13.7.7	Platform force distribution over all 2WD units	428
13.7.8	Mechanism: declarative platform motion specification via tubes	433
13.7.9	Policy: 2WD control modes, task progress, energy, slippage	433
13.7.10	Policy: hybrid platform control	434
13.7.11	Navigation: topological motion specification and control	435
13.7.12	Information architecture	435
13.8	Common software representations in robotics	436
13.8.1	ROS Messages	436
13.8.2	RTT/Orcos typekits	437
13.8.3	SmartSoft Communication Object DSL	437
14	Holonic system architectures	438
14.1	System design process	438
14.1.1	Design phases	439
14.1.2	Natural hierarchies for decision-making in systems	439
14.1.3	Hierarchy in knowledge versus hierarchy in architecture	440
14.1.4	Architecture: holonic responsibility structure	441
14.1.5	The 7Cs: 5C components, plus their Composability and Compositionality	442
14.1.6	Policy: component model as documentation, specification, or contract	443
14.1.7	System-level properties of architectures	443
14.2	Holon architecture for heterarchical decision making in task execution	444
14.2.1	Resilient, stable, robust sub-systems	444
14.2.2	Dependencies in architectures: hierarchy, heterarchy and holarchy	445
14.2.3	Holonic practices to use DOM models: advice, but don’t command	445
14.2.4	Explainability: causal driver for robustness, resilience, anti-fragility	446
14.2.5	Best practice: be conservative in what you send, be liberal in what you accept	446
14.3	Autonomy as explainable decision making	446
14.3.1	Endsley’s five levels of system autonomy	447
14.3.2	Sheridan’s ten levels of system autonomy	447

14.3.3	Explanation levels for autonomous decision making	447
14.3.4	Role of meta models in horizontal and vertical task composition	448
14.3.5	System safety as explainable decision making	449
14.3.6	Best practice: safety PLC	449
14.3.7	Best practice: active safety behaviour	449
14.3.8	System security as explainable decision making	449
14.4	Data sheets: meta data for digital resources	449
14.4.1	Data architecture	450
14.4.2	Function architecture	450
14.4.3	Coordination architecture	450
14.4.4	Information architecture	450
14.4.5	Software architecture	451
14.4.6	Hardware architecture	452
14.4.7	Digital platform	452

Chapter 1

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

This document's **formal representations of knowledge** (also referred to as "**models**") are built on the **axiomatic foundation** of (i) **entity** and (ii) **relation**: an **entity** represents "stuff", "things", "primitives", "atoms", ..., (for example, *wheel* and *seat*) and a **relation** represents a dependency between one or more properties of the entities that are connected by that relation [30] (for example, a *car* has *wheels* and *seats*, and a *bike* too). The composition of entities and relations that "belong together" is called a *property graph*, or *knowledge graph*.

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

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

The relation between a model and its meta model(s) is a relative concept: every meta model is a model in itself and hence has¹ its own set of meta models. Common practice limits the terminology to the triple (i) **model**, (ii) **meta model**, and (iii) **meta meta model**.

A **knowledge system** (*knowledge base*, *knowledge graph*) for a particular **domain** is a set of models and their meta models that describe "meaning" in that domain. **Ontology** is, in this document, a synonym for "knowledge system", in, both, its **information science** and **philosophical** incarnations. **Reasoning** in such a knowledge system is done via **queries**, that are **solved** via **graph matching** or **graph traversal**. A query is a partially complete model in a domain, and the query solver finds the missing parts by following the "right" relations in the knowledge graph.

A robotic or cyber-physical system has an **explainable** control system, if the

¹Or, ideally, *should* have, because it takes a lot of human effort to create such meta models.

latter can do the **higher-order reasoning** required to identify and communicate the **causal chains** in the graph relations that represent the knowledge on which the controller design is based. For example, an autonomously driving car that is involved in an accident must be able to motivate all the decisions it made in the situation that led to the accident.

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

- to find the **entities** and **relations** to include in the system’s modelling, and which ones to neglect.
- to find the right **abstraction**: which details must be represented in the **properties** of the models’ entities and relations, and how are **levels** of abstraction connected to each other.
- to design the **structures** that **connect** levels, and the **behaviours** that these structures relate to each other at different levels.
- to bring the right **levels in scope** of an offline design, or in an online decision making **activity**, at the right moment.
- how to create the **software** behind the knowledge bases, so that computers can **reason** on the available models.
- to find the right “**host**” **languages**, **to store and communicate** the available knowledge representations.
- how to reach the **standardization** required for **multi-vendor interoperability**.

1.1 Models to represent knowledge in science and engineering

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

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

A few examples of relations are:

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

The seminal contributions by [Herbert Stachowiak](#) [140] define a *model* as a formal representation that provides:

- **mapping**: it represents *something* in the real world, be it of natural or artificial nature.
- **reduction**: it represents only those parts of that “*something*” that are relevant for the application at hand.

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

- **pragmatism:** the model must serve some practical purpose(s), like documentation, analysis, simulation, design, . . .

1.1.1 Science reflects reality — Engineering reflects decision making

Scientists strive for models that allow **to analyse reality**. Engineers strive for models that allow **to design artefacts**, that **make decisions of how to change reality** in a **controllable and predictable way**. Engineering models typically make use of scientific models; the inverse is less commonplace. This document’s major ambition is to represent the human knowledge about the real and the artificial in formal models, in such a way that these models are not only used to help the *engineers* design and implement machines, but that the same models also feed those same *machines* with the information **to reason** about how they (should, could) **act** in the real world.

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 of the model, but one that depends on the **context** in which the model is to be used. In addition, that context also determines that the human-driven modelling stops with a particular *selection* of **axioms** or **facts**, that are not modelled in further detail but are assumed to be **grounded** in reality. In science, such grounding consists of (references to) other (possibly not yet formalised) models and axioms, and engineers add *software* to the mix of groundings. For both scientists and engineers, 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. This document represents explicitly the concepts of **context** and **composition** of models, in such a structured way that one can *add* new models to already existing ones *without changing* the structure and the meaning of the latter. When developing computer-controlled machines, there is an obvious extra core ambition: to create **software** libraries of **digital twins**, that implement (or “ground”) the models of the entities and their interactions.

1.1.2 Levels of abstraction: scope of relevant entities and relations

Every model is an **abstraction** of the real and engineered world: only (what the system designers consider to be) the *relevant parts* of how the real world behaves is covered by a model’s entities and relations. So, system designers must *actively choose* the “right” **scope** of these relevant parts, or, in other words, the right **levels of abstraction** that fit to the system’s purposes, [2, 50, 51, 154]. This implies two complementary choices:

- **abstractions:** to choose the “right” set of entities, together with the relations that connect properties in two or more entities.
- **levels:** to choose a *hierarchy* in the relations between the models, The purpose is to facilitate a methodological decision about whether or not to include particular information about the real world, in particular contexts.

Hypernyms and **hyponyms** are the names given to these *relative* abstraction levels. “Abstraction” implies that one makes an equally explicit choice about what *not* to take into account, as about what do to take into account. A pragmatic, minimal expectation of a “right” model is that:

- the relations do not contradict each other.

- the model does not have sub-models that are not connected to other sub-models by any relation.

Higher or *lower* level of abstraction are not absolute properties of a model, because that label only make sense *between* two models.

A **common interpretation** of *higher* and *lower* is that of the **is-a** relation, or *hierarchy in representation of behaviour detail*. That is, *model A* is considered to be of a *higher abstraction* than *model B*, if the latter **contains all** of the entities and relations of A, **plus**:

- *new entities* that are connected to entities already present in A, by the already present relations of A.
- *new relations* that connect the already present entities of A to the new entities of B.
- *new entities and relations* that were not already in A.

Here are some examples of this common interpretation:

- *high level*: a *car* has *four wheels* and a *steering wheel*.
- *medium level*: the *car's* two *front wheels* are *connected* to an *internal combustion engine*, and the steering wheel is of the *drive-by-wire* type.
- *low level*: the details of the type of engine and *electronic control unit* are added.

This document's **interpretation** of *higher* and *lower* is that of *hierarchy in representation of behaviour scope*. That is, a *model A* is considered to be of a *higher abstraction* than *model B*, if *model A* represents behaviour that "happens" at a larger scale in space and time than the behaviour in *model B*, while *being aware* of the existence of entities of the type of *model B*. Examples of this interpretation are:

- at a "high" level of abstraction, a *company* is represented by entities like *CEO*, *Board of Directors*, *plants*, and *products*. At a "medium" level of abstraction, the entities are *manufacturing lines*, *AGV logistic systems*, and *training divisions*. While a "low" level of abstraction deals with entities like *software versions of PLC controllers*, *energy levels of batteries*, and *number of screws in storage*.
- at a "high" level of abstraction, a *traffic system* is represented by entities like *cities*, *highways*, and *train stations*. At a "medium" level of abstraction, the entities are *houses*, *cross roads*, and *bridges*. While a "low" level of abstraction deals with entities like *density of traffic in a street*, *type of traffic lights on a particular crossing*, and *number of parallel lanes on a road*.

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

- the **association hierarchies** of knowledge and **information**.
- the **mereological and topological** relations, to represent "parts" and their "interconnections". For example, the **containment** hierarchy.
- the relation between **hyponyms** and **hyperonyms**, as the essence of *is-a-(sub/super)-type-of* hierarchy, or **subsumption**.
- **modelling hierarchy**.
- **mathematical representation** [139].
- **physical detail** (Fig. 2.1).
- coverage and extension in time and **geometrical space**.
- **tasks**: *what* has to be done is more abstract than *how* it is being done; several levels are common.
- **architectures**: they reflect the **engineering choices** about *how* all parts of a complicated system should be connected together, and *what* their interactions should be.

- software: an interface is a higher **abstraction** than any (of its possibly many) **implementations**.

1.1.3 Three dimensions of modelling: abstraction, scale, and complexity

System modellers must make decisions about **the³** three complementary modelling dimensions:

- **abstraction** (or **analogy**): entities and relations that reflect (part of) reality, independent of the other two dimensions.

For example, at all scales in space and time, the abstractions of **action**, **actor**, **actant**, **activity**, and **agent** hold; or of *free space*, *corridor*, *motion constraint* and *traffic flow*. Indeed, it makes sense to introduce these concepts when modelling traffic of agents: inside or around an elevator, in a corridor, in a building, between buildings, in a neighbourhood,... up to the scale of intercontinental ship and airplane navigation.

Similar **skills**, information associations and decision makings hold in analogous **situations**; while the concrete control and perception algorithms may differ drastically.

- **scale** (or **size**): the physical, geometrical or temporal magnitude of a model's entities and relations.

For example, robots acting (i) at *millimeter* scale inside a human body, (ii) at *centimeter* scale in bin picking applications, or (iii) at *meter* scale in indoor logistics navigation.

- **complexity** (or **coupling**): the ratio of the number of relations in a model to the number of entities in the model.

For example, the layout of a logistics center with (i) all pallets placed on the floor, (ii) with the pallets placed in racks where mobile robots can drive in between, or (iii) with the “pallets” are boxes that are placed in a dense 3D grid, where mobile robots drive over the top, being able to make 90 degrees turns, and getting material from the boxes by lowering their grippers via a cable.

An *increase* in the magnitude of one or more of these modelling dimensions *may* lead to a *decrease* in the magnitude of the other(s). For example:

- the design of an assembly cell with only one robot arm can be more complex than the design of one with two or more arms: because the latter *requires* coordination of the motions of all arms, it makes sense to introduce as coordination relation the allocation of *discrete* areas in the cell's workspace. And this type of coordination is a much simpler problem to solve than the continuous time and space coordination of the same two arms.
- from a certain density of traffic on, cars form platoons, and that simplifies the situation awareness perception requirements of all cars except the first one in the platoon.

1.1.4 Levels of modelling formalization

This document has models in all **levels of modelling** formalization:⁴

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

³I welcome insights about other sets of *fundamental* system modelling dimensions.

⁴As identified in the context of the **RobMoSys** ecosystem.

domain (including less experienced juniors) about the relevant entities and relations in that domain.

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

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

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

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

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

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

5. **autonomy:**⁵ these models can serve as an “artificial language” for run-time dialogues between machines. The formalization and its automatic checking has become (i) efficient enough to be used at runtime by the robots *themselves*, and (ii) rich enough so that the machines can set up added-value cooperations *themselves* via dialogues, or can configure, adapt and explain their own behaviour, with only minor human interaction. This class of modelling supports autonomous *self-configuration*, -adaptation, and -explanation, at run-time. Later sections in this document explain the many **degrees of autonomy** that one can distinguish in engineered systems.

1.1.5 Levels of decision making: context of intentions

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

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

⁵“Autonomy” is not the most appropriate term, because its **most popular interpretation** misses the **resilience** aspect: it requires a lot more than “just” autonomy for a robot to make the decisions that it can not make progress on its own any longer, *and* that it must engage in interactions with other robots. That “much more” requires unprecedented levels of “situational awareness” and “empathy”.

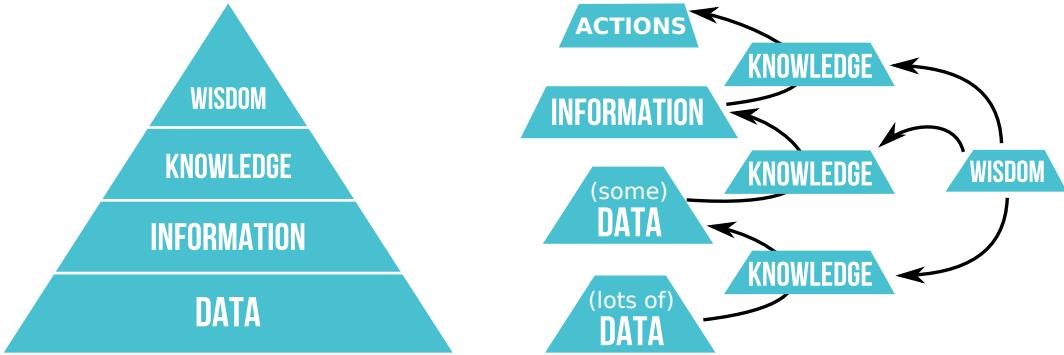


Figure 1.1: The *Data-Information-Knowledge-Wisdom* pyramid (left), and its application in *engineered systems* (right).

1.1.6 The mother of all association hierarchies: the DIKW pyramid

Figure 1.1 sketches the **essential association hierarchy** of all knowledge-based systems, namely the *Data-Information-Knowledge-Wisdom* pyramid. This document focuses on *engineered systems*, so the right-hand side version of this DIKW pyramid fits better: any engineered system in operation decides about its *actions* on the basis of *information* that it derives from the *data* that it has access to at runtime, or that has been programmed in advance. The human developers have put their *knowledge* into the system, on the basis of their *wisdom* about the system, its tasks, its resources and the environments in which it has to operate. Many types of *association hierarchies* are introduced in later Chapters of this document, and each of them has its own version of the DIKW pyramid's components: data, information, knowledge and wisdom. This document hypothesizes that:

- all components, except wisdom, will sooner or later be integrated in software.
- currently, most engineered systems only succeed in turning massive amounts of data into the much smaller number of data labels, with which the actions of the system are **hard-coded** by the system developers.

1.1.7 Explainability: decisions based on explainable relations and facts

Many decisions in *current* cyber-physical and robotic systems are made on the basis of:

- *stored facts*. For example, a robot moves from “A” to “B” *because* that decision is made in its **task specification**. Or it adapts its driving speed *because* it is driving through an area where there is a speed limit.
- *optimization formulations* of the problems that the system is designed to solve. For example, the trajectory that the robot follows to move from “A” to “B”, or to adapt its speed, are *computed* from a **constrained optimization** formulation that links the robot’s task specification and the traffic signs in the environment to the dynamics of the robot’s motion.

The appropriate use of these sources of information is almost exclusively taken care of by the *human developers*, because they have the *insights and wisdom* in the *cause-and-effect* relations that are relevant for the software implementation of the system controller that these human developers program by hand. Engineered systems can only scale—in size, reliability, liability and **resilience**—if more and more of its decisions will not be *programmed* in the

controller, but that controller can also make decision as the result of reasoning about the *symbolic representations* of all of the two above-mentioned other pieces of *system knowledge*. This approach fits very well in a *knowledge-driven* context, because (i) the reason why each of the relations or facts is being used can be explained whenever such an explanation is needed, and (ii) when using a relation or fact in a decision to act, the *execution* of the resulting action automatically gets one or monitors to check—before or during the execution of the action—whether the assumptions behind the validity of the decision are still satisfied.

For the meaning of the terms “explainability” and “decision making”, this document adheres to the following “age-old” definitions of Simon and Newell:

- **explainability** [104, 138]:

“To explain a phenomenon means to show how it inevitably results from the actions and interactions of precisely specified mechanisms that are in some sense “simpler” than the phenomenon itself.”

...

“For complex phenomena there may be, and usually are, several levels of explanation; we do not explain the phenomena at once in terms of the simplest mechanisms, but reduce them to these simplest mechanisms through several stages of explanation.”

...

“Every flea has its little fleas, and the scientist’s view accepts no level of explanation as “ultimate”. In summary, the study and explanation of complex human behavior is to proceed as follows: Behavior is to be explained by specifying programs that will, in fact, produce the behavior. These programs consist of systems of elementary information processes.”

- **rational decision making** [134, 135] gives priority to **satisfactory** solutions, and not *optimal* solutions:

“Evidently, organisms adapt well enough to “satisfice”; they do not, in general, “optimize”.”

This document’s ambition around “explainability” is to go a step further than the commonly used interpretation of that term in *Artificial Intelligence*: the *system itself* should provide explanations, and not its *human developers*. In addition, the explanations should provide causality relations that go further than what constrained-optimization solvers can generate directly from the system’s task specifications.

1.1.8 Mechanism (“what it does”) and policy (“how to use it”)

This document advocates to separate the modelling of representing how something works, or what that something does, from the representation of how to use it “best” in particular application contexts. The motivation behind this ambition is that “best” is always dependent on the *context* in which a piece of knowledge is applied.

The separation *mechanism vs policy* is only relevant in engineering, because science does not expect humans to make application-centric and context-dependent decisions.

1.1.9 The various meanings of “meta”

The [natural language meaning](#) of the adjective “meta” denotes something that is *more comprehensive*, of *higher order*, or *reflective on itself* or *on others*. This document attaches the following meanings to the term “meta”:

- *meta model*: model of the constraints a model must satisfy. Synonym of [meta language](#), or [schema](#).
- *meta meta model*: a model that **associates** entities and relations in two or more meta models. It is needed to allow [model-to-model transformations](#). For example:
 - to represent the fact that the term [is-a](#) in model A has the same meaning as the term [sub-class-of](#) in model B, and as [instance-of](#) in model C.
 - the above-mentioned equivalence of the two commonly used terms *meta model* and *schema*.
 - the [equivalent textual representations](#) of tree structures in [LISP](#), [XML](#), and [JSON](#).
- *metadata*: information about the interpretation and [provenance](#) of the fields in a data structure.
- being of a *higher* level of **abstraction**. The more appropriate terms here are [hyperonyms](#) (more abstract; less concrete). and [hyponyms](#) (less abstract; more concrete). Of course, the terms are the attributes of *relations* between entities, not of entities themselves. For example, [to act](#) is a hypernym of [to walk](#), and [to walk](#) is a hyponym of [to act](#); the latter is itself a hypernym of, both, [to-speedwalk](#), and [to-scramble](#).

1.1.10 Paradigm, pattern, best practice

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

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

A (design) [pattern](#) is **knowledge** about **how to solve** a recurring design problem, by means of a parameterized solution approach. In other words, it describes a [generic, reusable](#)

solution to a commonly occurring problem within a given **context**.⁶ Major reasons why a design can be called a “pattern” are:

- it has been used in multiple real-world applications. In other words, it **has proven “to work”**.
- the design description explicitly refers to the various “**forces**”, that can pull the design into several (foreseen) directions. In other words,
- the design description explicitly discusses the **trade-offs** between choosing which forces to apply to a specific context, and to what extent.
- the solution is **documented** following a structure that helps to discuss and select the **design trade-offs**: the pattern identifies explicitly what the influence is of each of the “forces” that drive the design in particular directions. In other words, a pattern has a knowledge graph behind it, so that reasoning about the design becomes possible.

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 *the* two categories of design aspects that appear everywhere in this document. Since the document’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.

A **best practice** is a **factual** observation that a particular approach and/or solution has been recognized to work well **in particular contexts**. The typical consequence is that a best practice is rather *monolithic*: it is to be applied as-is, without much leeway to optimize any trade-offs in the solution, or with hooks foreseen to adapt to the specific context of the application. for the simple reason that there is no identified relation between a set of design decisions and the resulting system performance. In other words, they are valuable *starting points* to consider in a design, but should never be a design target in themselves.

1.1.11 Simple, though not easy — Simplicity versus simplisticness

Formalisations of human knowledge come with varying levels of **complexity**, **comprehensibility**, and **understanding**. This document uses the term *simple* to refer to the *effort to understand*, and the term *easy* to refer to the *effort to implement*. The document also advocates to keep both aspects of system development clearly separated: the efforts to understand need be done only once, but the efforts to implement are to be repeated over and over again.

Two different terms derived from “simple” are worth introducing too: **simplicity** refers to the (positive) ambition to remove everything except what *matters*; **simplisticness** refers to the (negative) outcome that so much has been removed from a model that what remains is ineffective to reach the purpose of the modelling, namely to use the models as the foundations

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

for the perception, control and decision making that the application requires, with a specified and guaranteed quality.

1.2 Knowledge representation: from entities to meaning

Knowledge is the [interconnection](#) between [data, information, and meaning](#), [2]. Common names for knowledge [organization systems](#) are: [knowledge base](#), [knowledge graph](#), [semantic database](#), [thesaurus](#), [lexicon](#), [taxonomy](#), or [ontology](#). This document assigns no differentiating meaning to any of these names, except for [taxonomy](#) because the latter's *tree structure* offers good computational performance. (Formal) [model](#) is used as a generic term to refer to any formal representation of knowledge. At the highest [level of abstraction](#), such representation of knowledge has four parts:

- **entity** (“node”, “edge”, “term”,...): this is the [paradigmatic primitive](#) component in any knowledge representation. This document does not define *entity* any further than just any “thing” about which one has knowledge that needs to be formalised in a model. For example, a *robot* is an *entity* in a model of the automation infrastructure in a company.
- **structure**, with as commonly used alternative terms [grammar](#), [syntax](#), [lexical form](#),... Structure adds *connectivity* to *entities* of knowledge. This document considers the *graph* as the [first-class citizen](#) in the representation of connectivity structures. The *primitive* structural component in any knowledge representation is the [abstract data type](#) (“[key-value pairs](#)”, “[data structure](#)”) connected to an *entity* and representing that entity's [properties](#).
- **relation**. A [relation adds properties](#) to a *structure*, that is, the [key-value pairs](#) that “say something” about which properties of the entities in the structure are connected, and what *constraints* between their values are introduced by the relation. The *primitive* relation component in any knowledge representation is the [property graph](#). It is the [first-class citizen workhorse](#) of this document.
- **meaning** connects the properties of a relation to “actions” in the real world, that is, it *makes models executable*. **Behaviour** is a very special type of meaning, especially in this document's focus on cyber-physical systems because these “do things” and “have behaviour”.

Meaning has two complementary interpretations in this document:

- **explanation**: is built [on top of](#) the relation part, by linking (properties of entities in [relations to other relations](#), to “explain” their “role” in a particular “context”. That is, relations are added that explain *why* a particular relation has particular arguments and particular connections. For example, it is a [fact](#) that the robot system in Fig. 13.6 can not move its wheels in any direction at any time, and *the reasons why* are:
 - any two-wheel unit has only two actuated degrees of freedom, and motion in any direction in a plane has three degrees of freedom.
 - the platform introduces extra [kinematic motion constraints](#) between all wheel units.
- Commonly used more or less equivalent terms for “explainable meaning” are: [semantics](#), [pragmatics](#), [semiotics](#), [word sense](#), [lexical sense](#), [context](#),...
- **grounding**, or **embodiment**: some properties in a relation are connected to parameters in [software](#) components, that [make decisions](#) in a cyber-physical system by relating the “cyber” properties to “physical” properties in the real world.

For example, the controller of a robot decides about the robot's actual motion based on (i) sensor information about where “things” *are* in the world, and (ii) how the robot is *expected* to move around or to these “things”.

The mechanism to connect the “cyber” and “physical” parts is the **fluent**.⁷ This between software and models is, ideally speaking, bi-directional: software depends on models to make the correct decisions, but models must explicitly be designed to allow to be used in such a way by software.

For both versions, knowledge representers face the same challenge: **symbol grounding**. And for all types of knowledge representation, **formal** (that is, *computer readable* and *textual*) representation is a necessary condition for computer-based **processing** of the encoded knowledge. That processing has two complementary parts:

- **query formulation:** the formal representation of a “question” that one wants to ask to a knowledge base.
- **query solving:** the software that provides the formal representation of the “answer” to the query.

1.3 Knowledge representation: structure

The knowledge encoded in the *structure* part of the formalisation of the knowledge one has about the *entities* in the *domain of discourse* of interest, is limited to represent **which entities are connected to which other entities**. (The *reason why* that connection exists, is part of the *meaning* in the knowledge representation.)

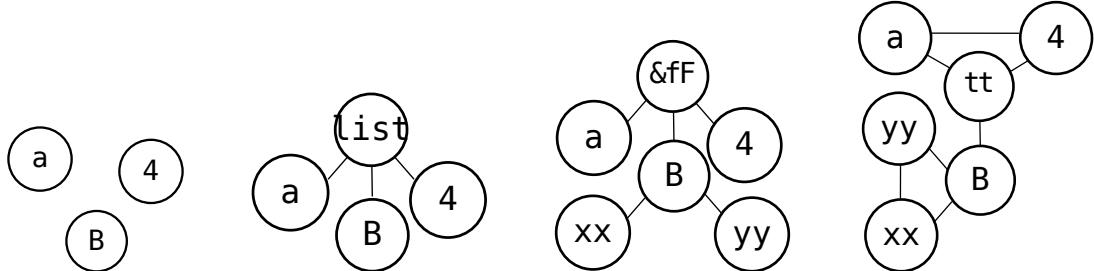


Figure 1.2: The core building blocks of knowledge representation, in their graphical form: *set*, *list*, *tree* and *graph* of *entities*.

1.3.1 Mechanism: set, list, tree, graph — Connecting entities in structures

This section introduces the following list of common structure representations (Fig. 1.2):

- **set** (“*bagcontainer*”) of *entities*, and this collection can be represented as an entity by itself. For example, a company owns a set of robots.
- **list:** an *ordered set*. For example, the order of the robots in an assembly line. The properties of the particular way of ordering the entities in the list is part of the knowledge, too.

⁷The traditional meaning of *fluent* in Artificial Intelligence is limited to *logical* parameters, but it can simply be extended no *continuous* parameters in space, time, energy, etc.

- **tree**: a list of two entities (“ordered pair”, or “2-tuple”) (E, C) , with E being a set of entities, and C being a set of *connections* between two of the entities in E . A **path** is a list of connections. The *whole set* of connections C has this **constraint** that any entity in E is connected to any other entity by (at most) **one path**.

The notation (E, C) introduced above is not the one most often used in mathematics. That latter context uses (V, E) , where “ V ” stands for “vertices” (our “entities”), and “ E ” stands for “edges” (our “connections”). Another popular terminology is “node” and “arcs”.

- **ordered tree** (“plane tree”, “directed tree”): a tree (E, C) in which every connection in C has as property a *list* of two entities, namely the first entity in the list is the **start** entity of the connection, and the second entity is the **end** entity. An alternative terminology is **parent** for the former, and **child** for the latter.

An equivalent definition of an ordered tree is that it is a *list* of entities and each of these entities can itself be a *list*. Here, the order of entities in the tree is inherited from the order of the entities in the lists.

An example of a tree is the organisation of a manufacturing plant: it has several lines, each line has several workcells, and every workcell can have several robots.

- **polytree**: a generalization of a tree, in that any node can have more than one *parent*, Fig. 1.3. This definition makes only sense if the polytree is **ordered**: the edges between nodes have a **direction**.

An example of a polytree is a **bill of materials** of a robotic assembly line: each robot can be the combination (via “multiple inheritance”) of several “types” of automation, e.g., a serial robot, equipped with a camera and a force sensor, etc.

- **graph**: a generalization of a tree, with:
 - *the same definition* of an ordered pair (E, C) , with E being a set of entities, and C being a set of *connections* between two of the entities in E .
 - *a different constraint* in that any node can be connected to another node via more than one path. Equivalently, graphs can have paths that *loop* from an entity to itself, while trees do not.

An example of a graph is the layout of a manufacturing plant: there are multiple roads that one can take to travel from one robot to another one.

- **directed graph**: any *connection* of edges in the graph is *ordered*.
- **directed acyclic graph** (DAG), Fig. 1.3: a directed graph with a special structure, so that more than one path can exist between two nodes, but the direction of the edges in the tree is such that one will never encounter a loop if one follows the paths along the arc directions only.

An example of a DAG is the **dependency structure** of actions in a task, for example, the assembly of a kitchen cabinet.

In the context of this document, *ordered trees* and *DAGs* play an important role, because their structural connection relations yield a formal representation of **partial order** between the entities in the tree or DAG.

1.3.2 Textual representations — Ordered lists and trees

Three major “standard” textual representations have been created over the years for the structures in the previous sections: **LISP**, **XML**, and **JSON**. Here is how these **host languages** represent an **entity**, a **list** and a **tree**, Fig. 1.2:

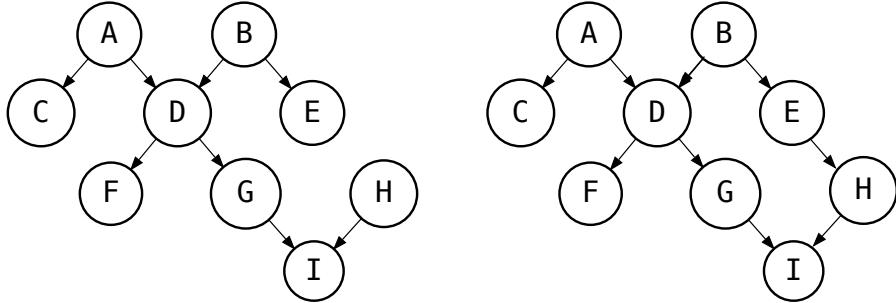


Figure 1.3: Polytree and directed acyclic graph (DAG).

- **LISP:**

```
entity: (a)
list: (list (a B 4))
tree: (tree (a (tree B (list xx yy) ) 4))
```

- **XML:**

```
entity: <a>...</a>
list: <list><a>...</a> <B>...</B> <4>,,,</4></list>
tree: <list><a> <list><B> <list><xx> <yy></list> </list> <4></list>
The string “list” is an arbitrarily chosen (that is, not standardized) tag name.
```

- **JSON:**

```
entity: {a}
list: { {a}, {B}, {4} }
tree: {a, {B, {xx yy}}, 4}
```

This section considers only the *structure* provided by the language syntax, not the *meaning*. Syntactically, the list- and tree-structuring tokens in these three languages are equivalent: `(...)`, `{...}`, `<>...</>`. In other words, what all these three languages provide is **serialisation** of a list or tree structure. That serialisation introduces *constraints* on the **structures** in two *implicit* ways:

- **lists have order:** the **left-to-right** convention of **Indo-European languages**, and especially of the technology dominating English language, introduces a **sequence ordering**: entities **lexically** more on the left are ordered “before” entities more on the right.

- **trees have hierarchy:** the recursive embedding, or **containment** structure, of the syntax keywords, such as

```
{ ... {...} ... }
```

introduces a **hierarchy** amongst the entities in the tree: the deeper entities are contained in the serialisation, the “lower” they are in the hierarchy of the tree.

1.3.3 Representation standards for structure: JSON-LD and RDF

The above-described *implicit* representations of structure have some disadvantages:

- the order and hierarchy implied by the syntax of the representation may not correspond to order and hierarchy in the real-world relations that are being represented. For example, the difference between a list (collection of entities with order) and a set (collection of entities without order) disappears.

- from the “outside”, one can not refer to a part in the list or tree. Hence, their [composability](#) is limited. For example, to make a list which is the composition of three other lists, one has to copy the contents of these three lists in a new list. The same information is then stored in more than one place, which is an error prone situation when that information must be updated or removed.
- only a parent node can “point” to its children nodes. Hence, a graph can not be represented, because a graph has at least one node with at least two other nodes that “point” to it.

The following graph modelling languages provide solutions to these problems, and they were born in the context of the [semantic web](#): [RDF](#) and, especially, [JSON-LD](#),⁸ have [built-in support](#) to represent [named \(directed\) graphs](#). Here is a possible encoding in JSON-LD (of the overlapping parts) of the graphs in Fig. 1.3, using the built-in primitives [@graph](#), [@type](#), and [@id](#), and our own (non-standardized!)⁹ tags [incoming-arcs](#) and [outgoing-arcs](#):

```
{
  "@id": "my-graph-xyz",
  "@graph": [
    {
      "@id": "A",
      "outgoing-arcs" : [ { { "@id": "C"}, {"@type": "directed"} }, { "@id": "D"} ] ,
      "incoming-arcs" : [ { ... } ] ,
    },
    {
      "@id": "B",
      "outgoing-arcs" : [ { "@id": "D"}, { "@id": "E"} ] ,
      "incoming-arcs" : [ ] ,
    },
    {
      "@id": "C",
      "outgoing-arcs" : [ ],
      "incoming-arcs" : [ { "@id": "A"} ] ,
    },
    ...
    {
      "@id": "I",
      "outgoing-arcs" : [ ],
      "incoming-arcs" : [ { "@id": "G"}, { "@id": "H"} ] }
  ]
}
```

The [@id](#) and [@type](#) keywords are the [semantic stepchanges](#) in JSON-LD compared to JSON or XML, because:

- the [@id](#) allows a statement in one model **to point symbolically** towards any entity in any other model.
- the [@type](#) allows **to point symbolically** towards a model that has all the information about the meaning of the node with this [@type](#). In other words, [@type](#) is the [@id](#) of the node’s [meta model](#).

The textual example above does not correspond completely to the graphical sketch in Fig. 1.3: only the arc between A and C got the property that its [@type](#) is [directed](#), while all arcs in the figure are directed. Better equivalence can be reached by either of these two approaches:

- adding the ["@type": "directed"](#) property to all *arcs*.
- adding the ["@type": "directed"](#) property to the *graph*, so that it applies to all arcs inside:

⁸Therefore, this document adopts JSON-LD as its preferred host language. But it does not introduce any dependency on that specific choice.

⁹These tags are not part of the JSON-LD standard, so one should not expect that the consistency of incoming and outgoing arcs is guaranteed automatically. In other words, it is the responsibility of the modeller to realise the consistency.

```
{
  "@id": "my-graph-xyz",
  "@type": "directed",
  "@graph": [
    {
      "@id": "A",
      "outgoing-arcs" : [ {"@id": "C"}, {"@id": "D"} ],
      ...
    }
  ]
}
```

The non-standard tags introduced above (`outgoing-arcs` and `incoming-arcs`) can strictly speaking be avoided:

- any `outgoing-arc` is faithfully represented *implicitly* by *containment*:

```
{
  "@id": "my-graph-xyz",
  "@graph": [
    {
      "@id": "A",
      "outgoing-arcs" : [ {"@id": "C"}, {"@id": "D"} ],
      ...
    }
  ]
}
```

is equivalent to this more implicit version:

```
{
  "@id": "my-graph-xyz",
  "@graph": [
    {
      "@id": "A",
      [ {"@id": "C"}, {"@id": "D"} ],
      ...
    }
  ]
}
```

- the information in the `incoming-arcs` can be discovered by *reasoning* over the graph: whenever an outgoing arc from X to Y is modelled, the inverse incoming arc from Y to X is deduced.

The equivalence above pertains to the *structure* only. The `@type` discussion already pertains to the *meaning* of nodes. Anyway, this document advocates *not* to rely on implicit meaning; this implies that a `@type` property is expected for every node.

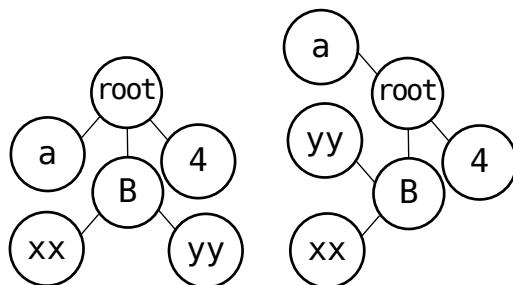


Figure 1.4: Left: labeling one entity in a tree as `root` gives an *order* to the tree. Right: *graphical* re-ordering of the tree does not change *structural* order.

1.3.4 Bad practice: implicit structural order implied by lexical or graphical order

As described before, using **textual** representations of the structure in “graphs” inherits structuring assumptions about order and hierarchy from the lexical structure of text. That structure inheritance happens *implicitly* too often, and implicit relations are a fundamental problem for good knowledge representation. Hence, two keywords have been introduced in JSON-LD to make the *ordering of entities* in a **JSON-LD array** explicit: `@list` and `@set`.

A similar “inherited structure” presents itself with the **graphical** representations of “graphs”: the left-hand side of Fig. 1.4 shows a tree as we often see it used in (not so good) “modelling” contexts:

- *hierarchy* is suggested by the graphical top-down layout of the tree.
- *order* is suggested by the graphical left-right layout of the tree.

The first implicit structure can be made explicit by indicating which of the entities is the **root** of the tree. This indication of being **root** is not **property** of the entity, but an **attribute** of that entity, given to it by an explicit root-indicating relation.

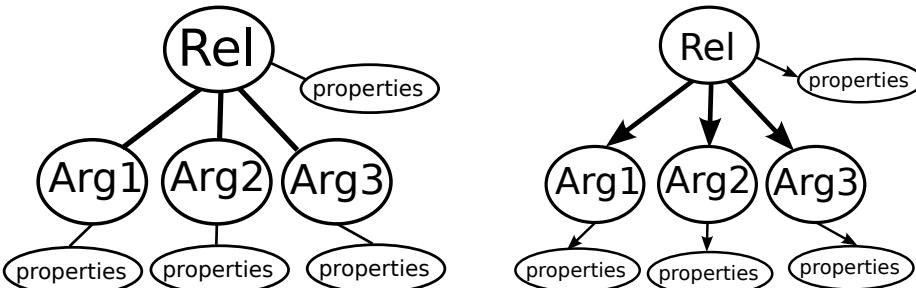


Figure 1.5: A relation represented as undirected or directed graphs. The latter’s arrows never(?) add information that can not be represented in the property nodes.

The “direction” of edges often represents a **meaning**, whose graphical representation (the “arrow”) is **implicit**, that is, only clear between people that share the same application domain background knowledge. Figure 1.5 is a graphical sketch of a tree (in this case, a **relation**), using *undirected* arrows (left), and *directed* arrows (right), as is commonly done in literature. However, the arrow does not convey any particular extra meaning in addition to what is represented in the properties. Indeed, for all cases where the arrow has some meaning (like for indicating *order* of arguments in a relation), that meaning can be represented in a key-value pair property.

Figure 1.6 is another graphical sketch of two (undirected) trees. This document adds no meaning at all to the difference in the *graphical ordering* of the nodes in the trees, because it advocates to add order in lists, trees and graphs as *explicit relations*.

1.3.5 Policy: graph languages with less semantics

The JSON-LD ecosystem is the most recent branch of the (not too) many standardization efforts for “graph” languages. Historically speaking, **XML** is probably the most popular “host language”, due to the fact that it has the olderst ecosystem of tools, developers and users,

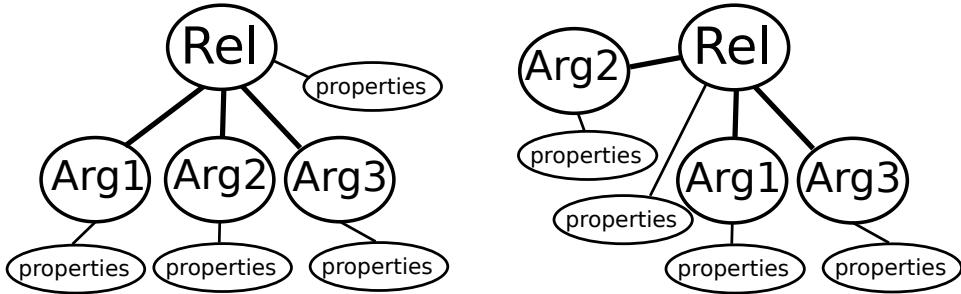


Figure 1.6: The same relation represented by two undirected graphs, but with a different “ordering” in the graphical representation. This document adds no meaning at all to this order.

that is, still, an order of magnitude larger than those of JSON-LD or RDF. XML-based extensions for “graphs” have been developed, such as [Xlink](#); it provides the *mechanism* for cross-linking that is necessary in a graph, but it does not provide the *semantics* of “context” and of “entity IDs”. [EXPRESS](#) is another modelling language with a mature history (but a decaying industry-backing), to represent product data, institutionalized in the [ISO Standard STEP](#).

1.3.6 Pattern: structural constraints on collections as array, dictionary

The following compositions of collections are of universal importance:

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

1.3.7 Policy: RDF graph, factor graph

This Section introduces some common types of relation graphs. They are all just special cases of the property graph representation primitive, and are sometimes used as “standard” representation in particular application domains.

An [RDF¹⁰](#) graph (or **RDF triple**, or **linked data graph**) is a **subject-predicate-object** relation (Fig. 1.7):

- the first node represents the *subject*.
- the second node represents the *object*.
- the directed arrow represents the *predicate*.

¹⁰“RDF” stands for [Resource Description Framework](#).

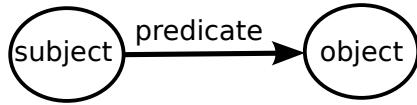


Figure 1.7: An **RDF** graph is a property graph with *triples*, that is, two nodes and one relation edge: one *subject* node, one *predicate* edge, and one *object* node.

Figure 1.8 is the straightforward property graph equivalent of the traditional “labelled” RDF graph representation of Fig. 1.7. The labelled arrow is by **reified** into a relation node.

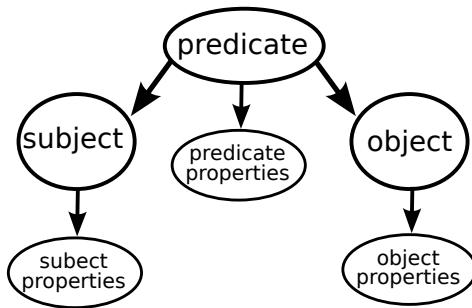


Figure 1.8: The property graph equivalent of the RDF triple of Fig. 1.7.

A database with RDF triples is called a **triplestore**; if the database adds the “name” of the graph in which the triple is stored—and hence adds a fourth component to the representation—one calls it a **quadstore**, or a *named graph*. The introduction of that “name” (or, “**identifier**”) is essential in the creation of **linked data** representations: it allows one entity or relation **to refer to** other entities or relations. **RDF graphs** have their origin in the domain of **linked data**, and serve first and foremost the *human* modeller. The property graph originates from the domain of **graph databases** and their *software support*.

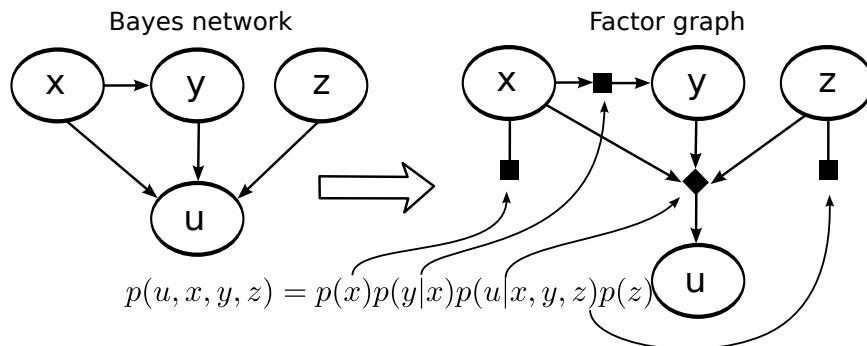


Figure 1.9: Factor graph, as generalisation of a Bayesian network.

Factor graphs find their origin in (Bayesian) information processing [80], removing a structural restriction of traditional **Bayesian networks**:

- *any* joint probability distribution is **factored** in smaller parts, and not just in *conditional* probability distributions.
- each node can only be connected with an arrow to one other node, Fig. 1.9.

Many properties in a factor graph *are numerical* values that represent *probability*, while RDF and property graphs *can consist of symbolic* values only.

1.4 Knowledge representation: relation

This Section explains the knowledge representation step from a **structure** to a *structure-with-properties*, that is, a *relation*.

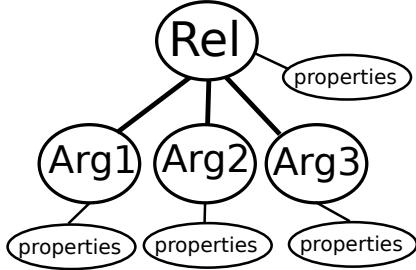


Figure 1.10: A relation represented as an (undirected) graph, with **property** nodes attached to the **relation** node, as well as to each of the relation's **argument** nodes.

1.4.1 Mechanism: property graphs

This Section introduces the (undirected) graph representation of a generic **relation** between **entities**. A relation adds **properties** (or, **data types**) to the **entities** involved in the relation. Not surprisingly, that combination is called a **property graph**, or **entity-relation graph** (Fig. 1.10):

- the **relation** as a whole is represented by the root node **Rel**.
- the **arguments** **Arg1**, **Arg2** and **Arg3** in the **relation** **Rel** are each represented by a node, with a connection (“arc”, “edge”) between the **relation** node and each **argument** node.
- there is an arc between each node (*and arguments*) and its own **property** node.
- *properties* represent information like name, identity, type, the *role* of the **argument** in the **relation**, the data structures that store the parameters that define the entity’s “behaviour”, **provenance**, **Dublin Core** metadata, etc.
- the *combination* of a node and its **property** node(s) is called an **entity**.

Properties are represented as **key-value pairs**, or any other **abstract data type** that can represent properties of “something”, and more in particular the *meaning* of the node it is attached to. The keys are often called the node’s **semantic tags**.

Traditional graphs represent a relation by *one single edge* between *two nodes*, that is, a **binary** relation. But one of the main reasons for the popularity of property graphs, is that they can represent **n-ary** relations between *more than two nodes*. In graph theory, the latter *structure* is sometimes given the name of **hypergraph**; and the *reason why* n-ary relations are necessary goes by the name of **reification**. In the context of Fig. 1.10, reification means that the node **Rel** is explicitly introduced to contain all information of the n-ary relation that could not be represented without it.

1.4.2 Policy: first-order and higher-order relations

A **first-order** relation has only *one layer* of relation-argument connections, as in Fig. 1.10. Here are some examples of first-order relations:

- “ $a = b$ ”, equivalently represented as “ $=(a,b)$ ”: the **relation** is *equality*, with two **arguments**, **a** and **b**. The **properties** of the **a** argument could be that it is a *natural number*, and of the **b** argument could be that it is a *real number*. The **properties** of the **relation** could be **pointers** to the mathematical theories of “equality”, between different *number systems*.
- relations between labels on maps, with the [OpenStreetMap](#) and [Open Geospatial Consortium](#) ecosystems providing many of these [\(geo\)spatial relations](#); [here](#) is a list of such relations. This [concrete example](#) of the [river Ourthe](#) shows the key-value pairs in the **properties** of the relation “Ourthe”:

name	Ourthe
name:ru	Yrt
waterway	river

- [spatial relations](#), that each have two regions as **arguments**. For example, [boundary](#), [interior](#), [contains](#), [disjoint](#), etc.

First-order relations have already a long history in engineering, computer science and artificial intelligence. A major example is the [S-expression](#). It is the basis of [context-free languages](#), with programming languages as major representatives. But also [reasoning](#)-capable languages, like [first-order logic](#) (logical statements composed via [Boolean operators](#)), [Prolog](#) (parameterized first-order logic relations), and [Lisp](#); or [query](#) languages like [SPARQL](#). A specific major type of S-expression is an [algebraic](#) relation; for example, [Newton's law](#) expresses the linear relation $f = ma$ between force f , inertia m and acceleration a .

Higher-order relations are formed from **composition** of first-order relations. A higher-order relation is a **relation about a relation**, that is, a relation that has other relations as its entities, or “arguments”, Fig. 1.11, [65, 111, 128]. Here are some examples of higher-order relations:

- [second-order logic](#) adds [quantifiers](#), such as in [linguistics](#) or in [logic](#): [for-all](#) (\forall) and [there-exists](#) (\exists). For example: *all Belgians are Europeans*, expressed more formally as: $\forall a : a \in \{\text{Belgians}\} \rightarrow a \in \{\text{Europeans}\}$.
- [transitive](#) relations: for example, if one person is the [descendant of](#) another person, and the latter is the [descendant](#) of a third person, the first person is also the [descendant of](#) the third person.
- [inverse](#) relations: for example, [ancestor-of](#) and [descendant-of](#). One person can be the [child](#) of another person, and the latter is then the former's [parent](#).
- other types of such relations are the [transitive](#), [converse](#), [identity](#), [associative](#), [commutative](#), [symmetric](#), [asymmetric](#), [reflexive](#) or [equivalent](#) relations.
- [constraint](#) (Fig. 1.14) which are relations that put limits on the values of some properties in some entities connected by another relation. For example, the relation that expresses that one object is *closer* than another object.
- [tolerance](#): the intervals within which the values of a constraint relation must fall. For example, a robot can be tasked to approach objects *not closer* than one-tenth of its size.
- the modelling pattern [entities-relation-constraints-tolerances](#) itself is a common form of a higher-order relation, in that no engineering system can be made to execute robustly unless many parts of its behaviour are continuously checked to remain “within tolerance”.

Both first-order and higher-order relations are **property graphs**; the former are always very shallow *trees*, but the latter are almost always real *graphs*. This document has a strong focus on higher-order relations, because they are *the* mechanism to add **semantics** (“meaning”). That is, in many cases, higher-order relations encode the knowledge about:

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

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

1.4.3 Policy: properties and attributes

In most *entity-relation* models, the entities as well as the relations have **properties** (or **predicates**), that is, **key-value pairs** represent the *meaning* of the entity or the relation. A **higher-order** relation is a *connection* between two or more of such keys or values in two or more entities or relations.

Often, the terms **property** and **attribute** are used interchangeably, but this document makes the following semantic difference:¹¹

- **properties** of an entity are the data that that entity **possesses**, or “owns”, or that are “proper” to its existence. So, without that data the entity loses its meaning, *irrespective of the context* in which it is used.
- **attributes** are the data that are **given** to an entity, *depending on a particular context* of relations that involve that entity as an argument. More specifically, the attributes are the properties of the *role* of the entity in such a relation.

An example of a *property* is that every physical object has mass and electrical conductivity. Here are some examples of *attributes*:

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

Properties are typically unique; for example, one particular object has only one specific mass, not several. Attributes can be given multiple times to an entity, every time in another relation. For example, a car can have a *serial number* given to it by the manufacturer, another one

¹¹With the choice motivated by the *etymology* of the terms.

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

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

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

So, in principle, this document has little need to use the term “attribute”, because it advocates the systematic use of relations as **first-class citizens**, so any possible attribute of an entity is always represented as a property of a relation. In other words, the attributes that an entity would get in a relation are represented by the entity-specific properties of that relation.

During the *human process* of knowledge modelling, it is **best practice** for the knowledge modellers to always ask themselves the question whether the key-value pairs they are adding to a knowledge graph are “properties” or “attributes”: classifying them as the latter *implies* they have to make a hitherto “hidden” relation explicit. That newly explicated relation then has become an entity in itself, so that it can be used as an argument in other relations.

Hypostatic abstraction is a commonly used reification operator: it transforms a **predicate** of an entity into a property of a relation. For example:

- *this joint in a robot is very stiff* is modelled as *this robot joint has a stiffness relation, and the magnitude property in that relation is high.*
- *this map is old* is modelled as *this map has a provenance relation, and the age value property in that relation is high.*

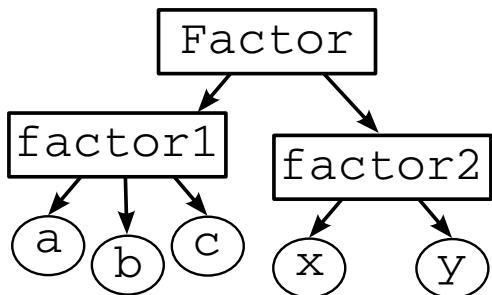


Figure 1.11: Higher-order relation: The Factor relation has two other relations, factor1 and factor2, as its entities.

1.5 Knowledge representation: meaning

This Section explains the knowledge representation step from a single relation to a *relation with other relations on top*, in other words, *meaning*.

1.5.1 Best practice: M0–M3 structure to represent meaning

Figure 1.12 gives a particular composition of higher-order relations that has many use cases, and that has gained the status of “standard” in knowledge (and formal language) represen-

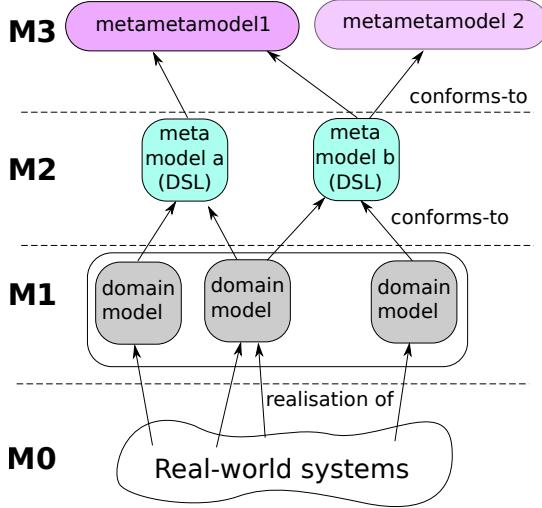


Figure 1.12: The particular (partial) order between types of models that appears in all knowledge representation projects, in one way or another. Note the essential difference of the `realisation_of` relation between the M0 and M1 levels, and the `conforms_to` relation between levels M1 and M2, and levels M2 and M3.

tation: [OMG's M0–M3 meta modelling structure](#),¹² with a more generic version introduced in [17, Fig. 7, p. 178]. The choice of representing the composition between *three* modelling levels (and *one* “level of reality”), is not arbitrary, but motivated by the following insights:

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

In this document, a model takes the form of a [property graph](#).

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

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

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

A particular model can have more than one meta model that it conforms to. For example, [Newton's second law](#) conforms to the meta models of:

- [mechanics](#). For example, the numerical value of a mass is *constrained* to be always a positive real number.

¹²The [mnemonics](#) of the M0–M3 abbreviations in the modelling order relation is that (i) the M stands for “model”, and (ii) the number represents the number of M's in the symbolic label of a level.

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

A simple example are the mappings that translate a model using JSON as its host language into a model that uses XML; the mapping is mostly concerned with transforming **curly braces** (i.e., “{” and “}”) into pairs of **angle brackets** (i.e., “<...>...<\...>”), while maintaining the *tree structure* of the encoding.

The pragmatic relevances of meta meta models are:

- that **translations** between models created in two modelling languages are only possible *if* and *where* there is such a **formally represented overlap** in the semantics of the modelling languages. That overlap is sometimes called the **canonical model**.
- as a loose synonym for **paradigm**: the essential concepts (entities as well as relations) in a domain are given a name. This **nomenclature** is then used as a constraint on the design of more concrete meta models: the *minimum* they have to do is to represent all concepts in the meta meta model.

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

This M0-M3 structure is a meta meta model, or paradigm, in itself: it represents the *belief* of designers that a particular composition of models can be considered together as a “connected”, “complete”, “consistent”, “relevant”, “essential”,... representation of a domain. In addition, the discussion in this Section is a proof that the M0-M3 structure can also be applied to any set of abstract concepts that humans use to attach meaning to their observations of that real world, or to invent meaning in imaginary worlds: models no not exist in the real world, only in the minds of people.

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

easier for practitioners in a particular application domain.

1.5.2 Policy: higher-order relations as context

This document has the following **axiom** at its core: *situational-aware explainable engineering systems are impossible without explicit modelling of higher-order relations*. The need for higher-order relations is the fact that each application comes with its own particular **context**, that is, the knowledge needed **to interpret** entities and relations used in that application, and (hence) **to configure** many of its “**magic numbers**” to fit that context.

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. Or, the earlier-mentioned relation of Newton’s law, because a full understanding of that law requires a large set of knowledge relations that are not present in the the law’s symbolic representation itself:

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

1.5.3 Further examples of higher-order relations

This Section lists some more examples of (**complicated**) higher-order knowledge relations:

- *cybernetics and robotic systems*: they not only *observe* the physical world, but provide *control* around those observations to steer the physical world towards a desired state, [58].
- *intent and configuration of tasks at various levels of control*: every robotic system has motor controllers to create motion, and sensor processorss to perceive the state of the world. Both activities are eventually driven by the “*intent*” of the task.
- the concept of a *singularity* in the mechanical configuration of the **kinematic chain** of a robot, relates the values of the chain’s joint positions with the chain’s geometrical model, and with its energy transmission capabilities. The knowledge *that* something like singularities exist is already a higher-order knowledge. The knowledge *how to assess/compute* singularities is yet a level higher in the modelling, because it relates the singularity knowledge to particular numerical solvers. Another higher level of modelling is to connect the singularity concept to loss of motion capabilities of the robot’s kinematic chain.

Hence, in case the robot’s actual motion deviates “too much” from the sensed motion, the hypothesis of being in a singular configuration is relevant to explore. But that hypothesis comes only in view of the robot controller if the latter could do the above-mentioned *cause-effect chain* reasoning, by traversing the mentioned relations from “effect” to (possible) “cause”. Finding solutions to the singularity problem re-

quires yet another higher relation, that connects the singularity concept to skills that provide alternative ways to satisfy the same task constraints by reconfiguring the kinematic chain.

- the context of the *requirements of the task* the robot is executing: not every *geometric singularity* is also a *task singularity*. For example, pushing a load with fully stretched arms can be a good approach to reduce the amount of force needed in the muscles/motors, while it *is* a geometrically singular configuration.
- *semantic maps*: collections of formally represented geometric primitives (“map”) in the “world”, with relations (“semantic tags”) on top that link some *properties* of the geometric primitives in the map to *meaning* and *behaviour* of the “agents” in an application context.

In the domain of **geovisualisation**, the term **thematic map** is used more often.

- *semantic localisation and tracking*: using knowledge relations to support the decision making about what sensor processing algorithms to use on which parts of the sensor data, and to find out where a robot is on which part of a map.
- *configuration* of constraint solver algorithms: which constraint and objective functions to use, how to initialise the solver, which monitors to add for deciding whether the solver has reached a desired result or not,...
- *configuration* of software architectures: which communication streams to use, with which mediators, which data models, which communication patterns, etc.
- the *semantics of natural languages* have a hierarchical structure of interconnected higher-order relations, the so-called hierarchical structure **hypernyms and hyponyms**.¹³ 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*. Or, “*to act*” is a more abstract verb than “*to move*”, because motion is a specific form of action that requires entities and relations of **physical objects** and their location in **space and time**. And “*to move*” is more abstract than “*to grasp*”, because grasping is motion with hands and fingers, and with the relations of **form and force closure** of the fingers around a physical object.
- **level of measurement**: nominal (types), ordinal, interval, and ratio.
- **directed acyclic graph** (DAG): a graph-structured relation on entities with particular ordering constraints.
- **causal** relation, **causal chain**.
- **dependency graph**: any relation modelling the knowledge that one particular relation can only be applied in a particular context *if* other relations are also applied. A dependency relation often comes with an *order* (partial or strict) of those applications.

(TODO: other relevant types: *theory of mind* [164]: what do agents know about what other agents know? **action languages**, **transition system**, **abstract rewriting systems**, **Kripke structure** and **semantics**, **universal algebra**, **fluents** and **terminals** [99], **graph rewriting**, **situation calculus**, **event calculus**, **constraint satisfaction problems**. Most of these are mathematical models are of little practical use, except for fluents (that link the knowledge to executable software and to real-world measurements) and constraint satisfaction problem solvers (that turn declarative action specifications into procedurally executable ones).)

¹³The terminology for these relations is another name for what this document has called the **is-a** relation, and its inverse.

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

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

1.5.5 Facts

A **fact** is a *leaf* relationship in a knowledge graph: it is not *explaining* any other relation, or *deducing* one relation from some other relations. Facts can have any “degree” of knowledge contents, such as:

- *physical* first principles, or *axioms*, like [Newton’s Laws](#).
 - *natural* first principles, such as the [Linnaean taxonomy](#).
 - *societal* first principles, or *laws*, such as the [traffic code](#).
 - *statistics*, such as [demographic](#) facts.
 - *heuristics*, such as: *where there’s smoke, there’s fire*, or *the number of hours to spend on understanding a course is proportional to the number of lectures*.
- Another category of heuristics is when one does not yet have all the facts needed to make a fully consistent, knowledge-based reasoning. For example, a robot continues its [task](#) even though it lacks the information to be sure that all the task’s pre-conditions are met, but driven by the fact that is also does not have information about the contrary. That is, there are no indications that the task is progressing incorrectly. Such a situation occurs often when localising oneself on a map, searching for the first unique “landmark” that can tell for sure where one is.
- *just plain facts*, such as [there is an abbey in Leuven](#) with the Dutch name of [Abdij van Park](#). [Maps](#) are indeed huge collections of such facts, and in themselves primary examples of property graphs with meaning.

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

1.5.6 Declarative and imperative models of behaviour

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

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

The composition of such dependencies results in a graph of relations in itself, a so-called [dependency graph](#) (Sec. 1.8.10). There are two major complementary ways to turn the information represented by an action dependency model into “actions”:

- **imperative**: the dependency constraints are “solved” explicitly, at [design time](#). So, at

run time, the system executes the resulting “**recipe**”. Which is sometimes also called the **control flow** of the system’s behaviour.

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

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

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

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

1.5.7 Hierarchical and serial ordering: scope

A key motivator for *model-driven engineering* (MDE) is the observation that the amount of software in modern (robotic) systems has grown so large that human developers can no longer 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 code, which would mean that no significant progress has been made.

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

- **hierarchical order**: or, *taxonomies*. These are **trees** of models, where each depth level models an explicitly identified set of properties, relations and constraints.
For example, topology *implies* mereology (one can not talk about two entities being connected if the two entities have not been identified), etc. Kinematic families of serial and parallel robots, with further specialisations of 6R serial chains or Stewart-Gough parallel platforms, etc.
- **serial order**: or, *dependencies*. The most useful dependency ordering has the structure of a **directed acyclic graph**, because the “direction” of an edge is a *declarative* way to model serial dependencies, that is, to bring a particular type of “order” between several **arguments in a relation**.

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 deter-

mine the order in which they are given access to physical resources, such as space or energy.

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

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

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

Most implementations of complicated systems have an abundance of *magic numbers*, that is, property values for which the **cause** of that value is not identified explicitly. In other words, it are the human developers who have chosen “a value that works”, and did not add the knowledge relations to the implementation via which the system controller itself could reason about what the “best” value would be, at any given time and in any given context. This is, obviously, an undesirable situation in this document’s context of **explainable** systems. The advice to deal with magic numbers is:

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

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

1.5.9 Don’t care: attribute in a higher-order causality relation

While a *magic number* is an attribute that is *important* to get right, for the efficient workings of the system, some numbers in the relations are not, in particular contexts. They sometimes get the explicit label **don’t care**; this label is again *not* a property of the number, but an attribute it gets in a particular context.

For example, the air pressure of the atmosphere is most often not relevant for the motion control of a robot.

1.6 Core relations: conforms-to and is-a

The **conforms-to** relation is essential and generic, for all model-driven engineering systems. The **is-a** is a very common specific case of the **conforms-to** relation, adding some strict constraints that need not be satisfied in the generic case.

1.6.1 The conforms-to relation

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 (or, rather, the **conforms-to constraint**) with respect to each of its meta models, that is, all constructs that are being used in the model satisfy the constraints on the relations that are made explicit in a meta model, but only *for as far as* the constructs in the model indeed use entities that are defined in a meta models. This **multiple conformance** property does *not* imply what is commonly called **multiple inheritance**, that is, that *all* constructs in a model must conform to the constraints of the meta models. (This is a property of the **is-a** relation.) This “not fully constraining” property allows composition with any new model and meta models, *if* these do not contradict constraints introduced by earlier meta models. This property of composition is a necessary condition for a system to deal with the **open-world assumption**.

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

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

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

1.6.2 The is-a relation

The **is-a** relation is a specific case of the **conforms-to** relation: the relation **A is-a B** requires that **A** satisfies **all** constraints that **B** satisfies, and that it has the same list of properties (but not necessarily the same values of these properties).

Its best known incarnation is probably in the form of **Liskov’s substitution principle** in **object-oriented programming** that states that **A** can replace **B** **everywhere**, without the rest of the world noticing the substitution.

(TODO: instance, object, class; inverse relation is **type-of**.)

1.6.3 Composition and inheritance

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

Some not-so-good practices in this context:

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

1.6.4 Best practice: four types of `is-a` and `conforms-to` hierarchies

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

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

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

The above-mentioned modelling hierarchies can be **composed** in non-hierarchical ways. For example: at a high level of *abstraction*, one can represent a robot as having *motors* and *links*; and it can make sense to compose that information with (i) a very low level of *granularity* (e.g., specifying the brand and type of the motors) and (ii) a medium level of *resolution* (e.g., specifying that their mass should be lower than 1kg).

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

1.7 Mereo-topology: most abstract representation level

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

1.7.1 Mereology: has-a

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

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

with a lot of background knowledge. The simplest interpretation (often referred to as the “highest” [level of abstraction](#)), is that of its [mereology](#).¹⁴ A mereological model just represents the **parts** that make up the “world”, without any additional structure or behaviour. In this case, the parts are `Relation_x`, `Entity_1`, `Entity_2` and `Entity_3`. In other words, a mereological model consists of the [set](#), or [collection](#), of the [Relations](#) and [Entities](#) that are relevant for the modelled system. The [context](#) to interpret the *meaning* of the relation is not specified explicitly. At the mereological level, such a context is just another, larger, mereological set, often called the [universe](#) or the [domain of discourse](#) of a model, and it represents all the entities and relations that should be considered together before one can hope to interpret the meaning of the model unambiguously. 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 the entity does not own the relation entities themselves.

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

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

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

Figure 1.8 is one (of the many possible) graphical representations of the mereology of Eq. (1.1). Figure 1.14 extends the model with a *constraint* on the relation, and a *tolerance* on this constraint; both bring in “loops” in the representation. The suggested approach of model **composition** has the advantage that the directed graphical models have only *acyclic* loops; this **pattern of cycle-free** composition is a [best practice](#) that is sometimes called the [dependency inversion principle](#), and that this document tries to follow as often as possible.

1.7.2 Topology: connects, contains

The [topological](#)¹⁵ version of Eq. (1.1) is as follows:

$$\text{Relation_x} (\text{Argument_1} = \text{Entity_1}, \text{Argument_2} = \text{Entity_2}, \text{Argument_3} = \text{Entity_3}). \quad (1.2)$$

¹⁴Holonomy and meronony are related terms to denote the symbolic relations that humans attach to “parts” and “wholes”.

¹⁵Strictly speaking, *topology* is the *geometric* concept of [continuity](#) of mappings between geometrical entities. Graph theory uses only the weakest, discrete, form of “continuity”, namely [connectivity](#): two nodes in a graph are [connected](#) if there is a [path](#) between them.

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

- **connection**: `Entity_i` is connected to the i th argument of the `Relation_x`, and this explicit association allows to reason about how to interpret the meaning of that specific entity in that specific role, again within, both, the inward and outward contexts.
- **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.

More formally, the **topological** view of the model in Eq. (1.1) brings two extra **relations** with respect to the **mereological** `has-a` relation:

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

and with two **entities**:

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

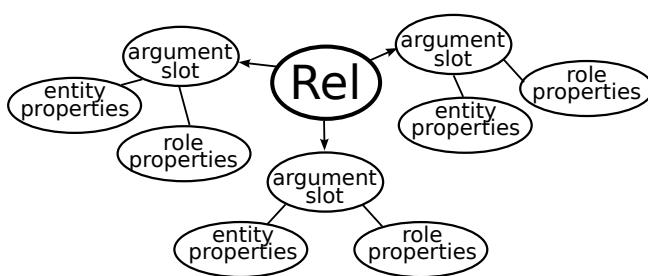


Figure 1.13: A graph representation of the **topological** model of the relation in Eq. (1.1). The arrows represent the `contains` relations, of the `relation` with respect to its `arguments`. The latter have undirected `connects` arrows to the properties of the `entities` that play a particular `role` in the `relation`.

1.7.3 Role of mereo-topological models

Mereological and topological models might seem overly simplified and obvious, but they have already a very important role to play in large-scale modelling efforts of digital 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. This scope is sometimes called the “world”, or the “**universe of discourse**”.

That effort is exactly what this document is kickstarting, for the robotics sub-domains of *motion*, *perception*, *world models* and *task specifications/plans*. For example, a kinematic chain is a relationship representing motion constraints between rigid body links and (typically) one-dimensional revolute or prismatic joints. The role of the links is to transmit mechanical energy (motion and force), while the role of the joints is to constrain or alter that transmission. Obviously, the order of the joints in the chain has an influence on the chain’s overall behaviour, and vice versa. Note that these sentences already contain a form of reasoning, such as: a robot has to have at least six joints to move its end-effector in all spatial directions; or, if

a joint is not connected to a link, directly or indirectly via other links and joints, it cannot influence that link's motion.

1.8 Patterns in knowledge representations

This Section introduces some very common patterns in knowledge representations.

1.8.1 Pattern: metadata of a model — Semantic_ID

The motivations to add **metadata** to all entities and relations in a model in a systematic way are:

- the topological structure

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

is domain knowledge. It is worth representing that domain knowledge explicitly, because then software tools can use it for semantic “graph search” purposes. More in particular, to structure the search to find where specific entities, relations and constraints are *defined*, and what *well-formed models* are.

- sooner or later, any model will have to be connected to new information, and so any system will become part of an even larger system. Again, explicit metadata helps to automate such model compositions. The simplest way to add knowledge to an existing model (which in a well-designed knowledge representation system is, by definition, also the default way) is to add new entities, relations and constraints, together with a reference to the meta-models where they are defined, without having to change anything to the already existing model.

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

A major consequence of (i) the *entity-relation* meta model for knowledge representation, and (ii) the practical usefulness of the M1, M2 and M3 modelling levels, is that this document suggests to adopt the *policy* to give every entity¹⁶ in a model the following set of two complementary *metadata*:

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

- **ID** (“model ID”): a unique identifier with which the entity can be referred to, unambiguously, as a *model* itself, even when being part of another model.
- **{MID}** (“meta model ID”): a **set** of unique identifiers that each refer to a meta model, that is, a model in which the constraints are defined that describe the well-formedness of those entities and relations that the model uses from that particular meta model. Often used alternative names for meta model are “*type*” or “*schema*”.
- **{MMID}** (“meta meta model ID”): a **set** of unique identifiers that each point to a meta meta model of this model, needed to transform the model to another formal representation, while keeping the meaning of the model unchanged.

The {MMID} is an *optional* argument, because it is often more natural to encode this information in the {MID} model. Or even not to encode it at all inside a (meta) model because

¹⁶ And via **reification** also every relation, and hence also every constraint.

it is a [higher-order](#) relation that connects different (meta) models together, and hence it introduces too much coupling when each of the connected (meta) models “knows” about the other ones it is connected to.

The purpose of the `semantic_ID` is to provide software that works with the model, with the ([symbolic](#)) information about where to navigate to, to find the semantic information about an entity in a model. Such a [semantic search](#) can even be supported *at runtime*, because storing the extra symbolic metadata information in binary versions of compiled software code does not require much space.

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

- `{outE}` (“outgoing edge UIDs”): a set of *pairs of identifiers*, where each pair identifies (i) one outgoing edge connected to the entity, and (ii) the entity with which that particular edge makes a connection. And in this context, “connection” means “the local entity is a *relation* and the “outgoing” entity is an *argument* in that relation.”
- `{inE}` (“incoming edge UIDs”): the same, but for the incoming edges. That means that the local entity is an *argument* in the relation represented by the “incoming” entity.

The purpose of these two identifier sets in the `edge_ID` is most often to serve as [indices](#). Indices are never extra knowledge in themselves, but are *derived* from the knowledge in a graph, for the sole (but useful) reason to speed up semantic search. They add value irrespective of whether the metadata they store is in a text-based “source code” form, or in compiled “binary” form.

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

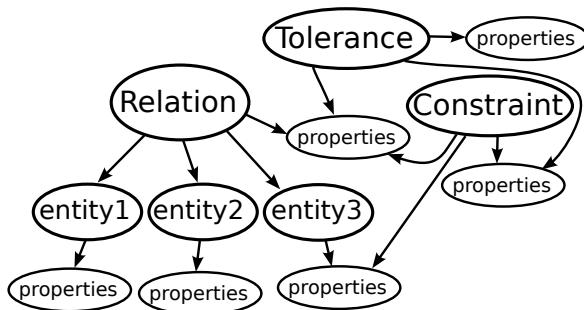


Figure 1.14: A *directed* graph representation of a higher-order model: the *constraint* between the properties of a relation and one of its arguments is a higher-order relation, with its own properties. Similarly, the *tolerance* relation is an even higher-order relation, too.

1.8.2 Pattern: entities–relation–constraints–tolerances

Figure 1.14 gives one particular **composition** of higher-order relations, which comes back in so many use cases that it merits to be called a [pattern](#):

- **one relation** connects **several entities**.
- each entity has its own *property* set.
- the relation has its own *property* set, too.

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

The figure uses a *directed* graphical representation, although that is **semantically not needed**: the arrows are just an easy way to represent the **dependency order** that comes with the pattern.

Here are some examples of knowledge representations where this pattern makes a lot of sense:

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

1.8.3 Pattern: taxonomy and frame as knowledge components

A **taxonomy** is a **tree-structured relation** on entities, [125], and it is this tree structure that allows for efficient reasoning via graph traversal. The **semantics** (or, *meaning*) of the *ancestor-descendant structure* that is built in in tree representations is that of **hyperonyms and hyponyms**:

- **hypernym**: (TODO)
- **hyponym**: (TODO)

The structure and meaning between hyperonyms and hyponyms are the **generalization** of the similar concepts in **class hierarchies in object-oriented languages**: objects are used for **encapsulation** and **information hiding**, while frames are used for explicit representation of knowledge. And the knowledge about what “encapsulation” and “information hiding” are, is just a specific special case of “abstraction” and “relation”. In knowledge representation, “hiding” is a (very) *bad practice*: the goal is *not* to make the same “thing” **reusable in multiple contexts** without having to change anything in its internal *implementation* (which has become a, or maybe even *the* the goal of object-oriented design), but rather the *opposite*: to allow reasoning to link any “internally” represented entities and relations to those in that context. And while **multiple inheritance** is a nightmare for *implementations*, it is an easy to fulfill requirement for *representations*.

The insights into what “stable” or “reusable” **components of knowledge representation** should be, existed already in the early years of Artificial Intelligence; **Marvin Minsky** called this **frames** of knowledge, [99]. The frame model adds to the knowledge represented in one or more taxonomies the extra entities and relations needed *to use* that knowledge:

- the context relations, that link knowledge to interpretations.
- the action relations that link interpretations of contexts to decision making.
- the behaviour relations that link actions to their expected outcomes, and to the disturbances that can make the action execution deviate into other outcomes.
- the graceful degradation relations that link disturbed outcomes to contingency actions to remedy the deviation.
- the fluents that link the knowledge to executable software (for example, the selection of which software to execute, and with which configuration of its ‘**magic numbers**’), and to results of other knowledge-based reasoning actions.

(TODO: more explanations and examples!)

1.8.4 Pattern: behavioural constraints on collections as queue, stack, and stream

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

- **queue** (or “**FIFO**”): entities are **added** always on one side of the list, and **removed** at the other end.
- **stack** (or “**LIFO**”): entities are **added** on one side, and **removed** from that same side.
- **stream**: a *producer* adds entities at one side, and a *consumer* removes them from the other side, but both have exclusive access (“**ownership**”) of their contiguous part of the full stream of entities.

1.8.5 Mechanism: querying & reasoning via graph matching & traversal

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

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

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

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

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

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

Graph traversal is an approach towards realising the **holy grail** of **higher-order reasoning**. The simplest form of graph traversal is the one that works on tree structures. And **tree serialization formats** such as **XML** and **JSON**¹⁷ have tree traversal query languages; e.g., **XPath** (and its superset **XQuery**), or **GraphQL**. Mainstream reasoning frameworks, like the ones built around **Prolog**, **OWL**, or **SWRL** cannot provide such higher-order operations,

¹⁷An insightful and concise comparison between XML and JSON can be found [here](#).

because their modelling remains at *first order*: it does not have the semantics to represent *relations about relations*.¹⁸

The concept of traversal can be illustrated by means of the simple example in Fig. 1.14: in order to find out which constraints should be put on the values of the **properties** node of the **entity3** node in the **Relation**, one can follow the arrows from the **Relation** node to the **Constraint** node and its **properties** node. The direction of the arrows reflect *meaning* in a relation in a model, but these arrow directions do not constrain the traversal through the graph: the **edge_ID metadata** allows to “navigate against the arrows” efficiently.

(TODO: In contrast to *first-order reasoning*, the literature and the state of the practice on higher-order knowledge graphs and graph traversals is scarce, e.g., [8, 43, 129], and software tooling support is only emerging. Hence, there are not yet enough use cases out there to identify common policies, let alone bad or good practices.)

1.8.6 Mechanism: Block-Port-Connector

This Section explains the **Block-Port-Connector** (*BPC*) meta model, [131], that has been used in various forms in software and systems engineering since decades. The BPC is a model for a **relation**, as a *composition* of several other types of relations:

- **Block**: is-a entity that **has-a** (number of) Ports, and each Port can play the role of argument in a specific BPC **relation**. When the Block entity is given the semantics of **is-an-agent**, its **has-a** relation has the extra semantics that the Block **owns** its Ports: the Block is the only agent entity to allow operations on the Ports; e.g., setting its properties, connecting or disconnecting to it, querying its status, etc.
- **Port**: each Port **represents** an **argument** in the BPC **relation**, and the **attributes** of the Port **represent**:
 - the *type* of the **argument**.
 - the *role* that the Port **argument** plays in the Block’s **relation**.
 - the *configuration* of the behaviour of the Block behind the Port that is visible to the Connector through that Port.
- **Connector**: is-a **relation** that **connects** two Port **arguments** in a concrete **instance-of** of a BPC relation. The **attributes** of the Connector **represent**:
 - the *type* of the Ports it connects. The **types** of both Ports must match.
 - the *role* that each of the Port **arguments** plays in the Connector’s behaviour **relation**.
 - the *configuration* of the behaviour of the Connector.

What is described above is only the **outer** part of the Port-centred interaction between two Blocks connected via a Connector. The **inner** view is realised by a Block’s internal part of a BPC relation. (Of course, that inner part can be again a composition of Blocks and Ports and Connectors.) The entity in the inner part **represents** the algorithm that realises the **behaviour** of the Block’s Port in the BPC relation. The traditional *graphical* way to represent a graph (as in Fig. 1.13) is with nodes and arrows between nodes. This may be good enough for *human* consumption, because we *see* what is the “*inside*” and the “*outside*” of a node. But to realise a *textual, computer-readable* representation of that link between the

¹⁸This is also a main shortcoming of one of the most prominent modelling languages in engineering, namely the **Unified Modelling Language** (UML). SWRL *does* allow *relations on relations*, but is semantically limited to *logical* rules, being based itself on the **RuleML** ecosystem.

outside and *inside*, the model of a Port must get an extra entity, the **dock**, Fig. 1.15. The *structural* part of the Port is extended with the constraint that each Port must have *exactly one inside dock* and one *outside dock*, and that both have exactly one **connector** between them. The *behavioural* part dock is extended with:

- type compatibility constraints between a dock's *inside* and *outside* parts.
- *transformation* behaviour. For example, changes in physical units, or in naming.

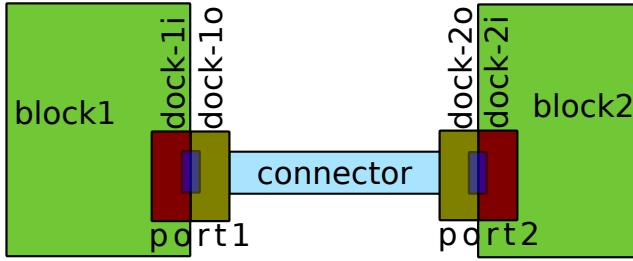


Figure 1.15: A computer-interpretable representation of a port needs an explicit modelling primitive, the **dock**, to represent its *inside* and *outside*, as seen from the perspective of the Block that *owns* the Port. There is also a Connector between the *inside* and *outside* docks of a Port.

1.8.7 Blocks as Components

In the context of cyber-physical and robotics systems, *Block-Port-Connector* is *the* meta model for the composition between **components**. “Component”, “system”, “subsystem”, or “agent”, . . . are *axiomatic* terms: they mean what the system designers want them to mean. To the best of the authors' knowledge, any possible definition always involves *circular reasoning*. This document uses the term “component” to indicate any entity that has **behaviour**. In the context of the *Block-Port-Connector* meta model, that behaviour is structured as follows:

- Block: the entity that *always* has *behaviour* inside.
- Port: a *view* (or “*interface*”) to (a part of) the behaviour in the Block. *If* the Port has behaviour, it is the just-mentioned *transformation* relation between its docks.
- Connector: *if* it has behaviour, it is the *interaction behaviour* of two Ports. For example, the *flow control* in the *stream* between a producer and a consumer.

This document's most abstract view on “Components” is the Block, *because*:

- that entity *always* has behaviour.
- Ports and Connectors *can* be behaviourless, and, at an abstract view, they can be replaced by a **connects-to** property of the Blocks involved.

For example:

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

processes have software representations of the behaviour of the task and subtasks (the *Blocks*), the information that the subtasks make available to each other (the *Ports*), and the **buffers** through which that behavioural information is exchanged (the *Connectors*).

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

(TODO: more details. Data sheet contains all information to allow usage of a port and a connection. Structural as well as behavioural information. Including constraints on the usage ranges, resource usage, dependency on other blocks or ports, etc. Hence, there will be a large amount of different data sheets, also on top of exactly the same implementation of the BPC mechanism. For example, safety setting for a mobile robot should depend on the context in which they are deployed: working in an industrial setting with trained adult operators is very different from working in a hospital where curious children can be sharing the same corridor as the robot.)

1.8.9 Storage and reasoning in property graph databases

The **storage** of *property graph* representations as in Fig. 1.8 is realised by **graph databases**, that provide **implementations** of property graphs [8]. More in particular, they support the mereological and topological aspects of *entity-relation models*: they keep track of which entities are connected to which other ones (via the `edge_ID` meta model, or a variant thereof), and of the properties of each and every node (via a **property graph** mechanism). And they support the specification and execution of queries via graph matching (always) and/or graph traversal (in still rare cases).

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

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

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

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

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

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

1.8.10 Policy: constraint, dependency and causality graphs

Many [higher-order](#) relations have the meaning of **constraints**:

- the relation represents a particular **configuration** of the properties in one or more entities or relations;
- it comes with a **semantic annotation** that represents the **type** of the constraint.

For example, that configuration could be *not allowed*, or on the contrary, it could be *intended to be realised*; similarly, it could be *advised*, *measured*, *estimated*, etc. Even for simple systems, the designers must be able to model **dependencies** between sets of constraints that is, some constraints are only to be considered *after* some other constraints have been satisfied; or they are mutually exclusive. 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; or it does not make sense to configure a *maximum* speed constraint for a robot that is lower than a *minimal* speed constraint. The following is a small “*taxonomy*” of increasingly more constraining dependencies:

connection → relation → constraint → dependency → [causality](#).

A causality dependency is the most constraining one, because the “effect” in the constraint *will* be observed when its “cause” has taken place.

There is a rich terminology to represent constraints and dependencies. For example:

- [constraint graph](#).
- [dependency graph](#).
- [causality graph](#), or **cause-effect chain**.
- **temporal ordering**: representing the constraint of (non)overlapping start and end events.
- **hierarchical ordering**: (TODO)
- **model dependency**: (TODO)
- **junction tree**: a given graph can have multiple *spanning trees*, via domain-dependent choices of how to reduce a graph-connected sub-graph into one single node.

The value of constraints, dependencies and causalities in knowledge representation is:

- they represent *what* conditions must be satisfied by any model (and hence software) that [conforms-to](#) them, but without constraining *how* the conditions need to be satisfied.
- it is trivial to *compose* them, by simple symbolic juxtaposition. It is less trivial to *guarantee* that such a composition still has [meaning](#); encoding such guarantees often requires *extra* [higher-order](#) relations.

The second item is often a consequence of the first.

1.8.11 Bad practice: modelling a dependency with a has-a relation

Many models are created within one particular application context, and then one often observes the following practice: entities in that context *always* come with a particular constraint, so that developers start assuming—implicitly but incorrectly—that the constraint is a *property* of those entities; hence, they add the constraint to the model of an entity by means of

a **has-a** relation. In practice, this leads to *tree* representations, and that has the advantage of being the simplest and most efficient knowledge graph models. The disadvantage of this practice appears when one needs to extend the scope of the knowledge models to broader and more generic application contexts, because the particular constraint does not hold anymore for all instances of the generalized entity. Here are some examples:

- *mobile robots* always have to carry their *energy source* themselves, and one must monitor how much energy is still available, to make sure that the robot can still reach a “charging station”. Hence, it is common to see a property like **has-a-battery** in mobile robot models. But that constraint does not generalize to all types of robots, because many of them are connected to the mains.
- *industrial robots* always have an *end effector*, to which tools are to be attached with which to realize the robot’s tasks. Hence, it is common to see a property like **has-end-effector** attached to the last link in a serial kinematic chain robot. But more general kinematic chain robots can have tools attached to many other links than just the “last” one.
- *weights* and *priorities* are introduced often in applications in which the robot system must execute multiple *tasks* at the same time. Hence, it is common to see a property like **has-priority** attached to the model of a task. But weights and priorities are most often the first ones to adapt every time a new task is added to the application, or removed from it.

The obvious better practice is to introduce stand-alone (“**reified**”) relations for these constraints. For example:

- an energy source relation that connects a robot to an energy source.
- a tool attachment relation that connects a tool to a particular link on a kinematic chain.
- a task dependency graph relation that connects a set of tasks to be executed now on a particular robotic system.

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

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

For example, the [Pythagorean theorem](#)

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

is an *abstraction* of all [right-angled triangles](#), and

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

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

Here is another example: [President of the European Council](#)¹⁹ is an *abstraction* of a particular [office holder](#) in the [European Union](#). It is a *fact* that [Herman Van Rompuy](#) was the first holder of that office.

¹⁹The Wikipedia article [just referred to](#) presents a nice example of a [higher-order relation](#), in the form of the [infobox](#): that is a filled-out version of a (more abstract) meta model, which is represented as a list of [key-value pairs](#) that are necessary to identify every *instantiation* of the represented relation.

1.9 Textual representations of knowledge

Drawing the graphs that represent higher-order relations is fine for human consumption, but computer processing of knowledge requires textual representations. This Section introduces relevant best practices and standards.

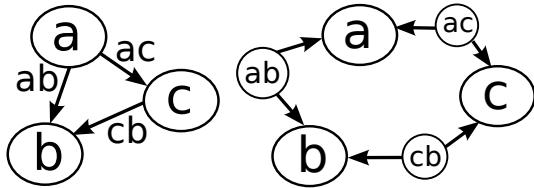


Figure 1.16: Labelled graph. Left: graphical representation for “human consumption”. Right: computer-representable version with the same contents, by **reification** of every arrow with a label, that is, by making the label a full-fledged relation of its own.

1.9.1 Textual representation of meaning — Labelled and named graphs

Figure 1.16 sketches a *labelled graph*, that is, a graph in which one or more of the arrows got annotated with a label. The **best practice** used to represent a labelled graph in a textual way is to give the labelled arrow a full relation on its own:

- the `@id` of the new `relation` is the “name” of the label. (Hence, this graph is also a *named graph*.)
- the new `relation` graph has the two original nodes as its `arguments`.
- the `properties` of the new label node encode the roles of both original nodes in the new `relation`.

Here is one of the many *possible* encodings in JSON-LD:

```
{
  "@id": "graph-abc",
  "@graph": [
    {
      "@id": "a",
      "ab": {
        "@id": "ab",
        "incoming-arc": "a",
        "outgoing-arc": "b"
      },
      "ac": {
        "@id": "ac",
        "incoming-arc": "a",
        "outgoing-arc": "c"
      }
    },
    {
      "@id": "b",
      "ab": {
        "@id": "ab",
        "incoming-arc": "a",
        "outgoing-arc": "b"
      },
      "cb": {
        "@id": "cb",
        "incoming-arc": "c",
        "outgoing-arc": "b"
      }
    },
    {
      "@id": "c",
      "ac": {
        "@id": "ac",
        "incoming-arc": "a",
        "outgoing-arc": "c"
      },
      "cb": {
        "@id": "cb",
        "incoming-arc": "c",
        "outgoing-arc": "b"
      }
    }
  ]
}
```

1.9.2 Textual representation of meaning — Context and composition

The Sections above introduced the first step to add `meaning` to the `structure` of a graph, namely by identifying particular trees in the graph as first-order relations, each with its

own **semantic label** (i.e., symbolic identifier and/or name). This Section goes one step further, adding the semantic label of **composition**: JSON-LD's `@context` keyword models the symbolic pointer from a graph model to sub-graphs in one or more “external” models. For example, the representation of an **argument** in a **relation** can be available in a separate model. And giving each “piece of knowledge” its own separate model in this way is a best practice:

- that approach allows *to represent* that knowledge only once.
- that knowledge model's `@id` allows *to refer* to that piece of knowledge at very low cost.

Here is an example model for Eq. (1.1):

```
{
  "@context": {
    "Entity_1": "IRI-of-Model-for-an-Entity/entity",
    "Entity_2": "IRI-of-Model-for-another-Entity/entity",
    "Entity_3": "IRI-of-Model-for-yet-another-Entity/entity",
  },
  "@id": "Relation_X",
  "@graph": [
    {
      "@id": "Entity_1" },
      {"@id": "Entity_2" },
      {"@id": "Entity_3" }
    ]
}
```

This model uses **shorthand names**, encoded in its `@context` sub-graph, to refer to specific entities in other graph representations, in a space-efficient way. The **IRI** part of the symbolic names refers to the concept of an **Internationalized Resource Identifier**; this is a standard way to create unique symbolic pointers, worldwide. The better known **URL** (Uniform Resource Locator) is a special case of IRI.

1.9.3 Textual representation of meaning — Type and conforms-to

JSON-LD's `@type` keyword models the symbolic pointer from a graph model to its **meta models**, that is, the models that describe the constraints that the graph model has to conform to.

For example, the fact that an **arguments** has a certain role in a **relation** implies that its properties must satisfy certain constraints, imposed by that **relation**.

Here is the example of the model in Eq. (1.1), that refers (in its `@context`) to a large number of other graph representations, each referred to by a **shorthand name** in this representation:

```
{
  "@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}, ... ]
}
]
}

```

Most nodes in the graph above have a `@type`, which is nothing more than a symbolic pointer to another model, somewhere, that represents “*what it means*” to be of that type.

1.9.4 Textual representation of constraints

In the JSON-LD ecosystem, the standardized formal languages [ShEx](#) and [SHACL](#) have been developed to model **constraints** on models.²⁰ The example below is a ShEx model of the constraint of 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-length-function/length"  
  },  
  "@id": "ID-RelationConstraint-u3u4d8e",  
  { "@context": "http://www.w3.org/ns/shex.jsonld",  
    "type": "Schema",  
    "shapes": [  
      { "id": "MyTernaryRelation",  
        "type": "Shape",  
        "expression": {  
          { "type": "TripleConstraint",  
            "predicate": "RelationNoA",  
            "value": { "type": "NodeConstraint",  
              "datatype": "http://www.w3.org/2001/XMLSchema#int",  
              "value" : "{length(MyTernaryRelation)}" }  
          } } ]  
    ] }  
}
```

(TODO: example misses a lot of information still.)

1.9.5 Policy: domain-specific language (DSL)

The textual tree representation languages introduced above (especially XML and JSON) are very popular *to exchange models between software tools*, because:

- they are worldwide accepted **standards**.
- (hence) they have a **rich eco-system of tooling** that is fully independent of the application domain in which they are used. More in particular, [parsers](#), [message queues](#), [message passing](#), [indexing](#), and [search](#).
- the best practice to map more complex data structures (in casu graphs and multi-trees) onto a tree model, is **mature and well-known**.

Most [Domain-Specific Languages](#) (DSL) use XML or JSON as their domain-independent *host language*.

²⁰At the time of writing this document, it is not yet clear which of both will get most traction in the community.

Chapter 2

Meta models for behaviour: activities, their interaction and coordination

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

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

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

Application developers design the composition of all of the above into [components](#), optimizing the trade-offs in the following three design aspects:

- **separation of concerns** into the five major categories (the “5C’s”, Sec. 9.1.1): *Composition* of a system via *Coordination* of *Computation* (“activity”) and *Communication* (“interaction”), with all of the above allowing *Configuration* of their data structures and functionalities.
- **composability**: the extent to which a **component** makes all of its “5Cs” **separately configurable**, to increase the opportunities **to reuse** that component in any kind of system.
- **compositionality**: the extent to which the behaviour of a **system** can be **predicted** based on the knowledge of the individual behaviours of the com-

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

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

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

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

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

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

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

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

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

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

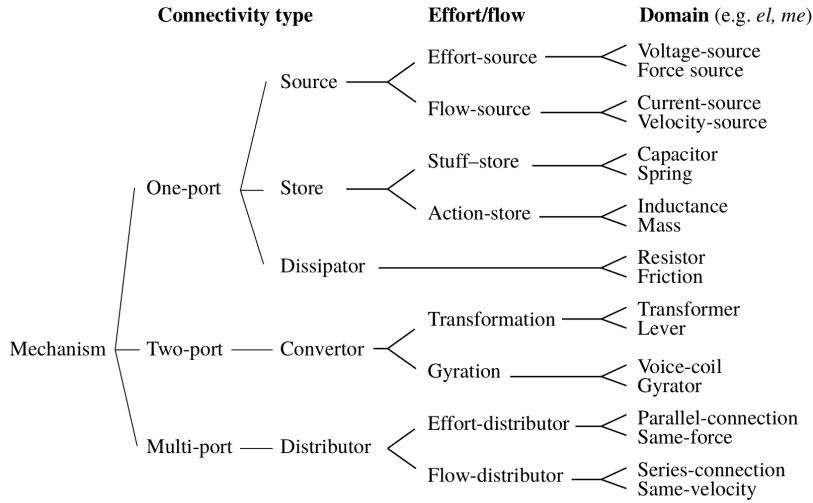


FIGURE 8. The taxonomy of physical mechanisms. The properties discriminating between the classes after branching are printed above the branch points. The classes on the right give some examples of mechanisms in the electrical and mechanical domain.

Figure 2.1: The topological relations between the various mereological top-level entities, according to [22].

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

Human civilizations have spent tremendous efforts on the scientific (“mathematical”) modelling of the **physical** (or “**continuous**”, or “**hardware**”) parts. Powerful **scientific paradigms** have been created, which support most of the technological innovations of the human race. Figure 2.1 gives a summary of one of the most successful of such paradigms, that of *engineering ontologies* within a **bond graph** context. The engineering of the **cyber** [162] (or, “**discrete**”, or “**software**”) parts has a much shorter history, and a lot less completeness, concensus and harmony has been achieved in the domain of **software engineering**.

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

(TODO:)

2.1.2 Cyber: decoupling of continuous, discrete and symbolic models

One fundamental aspect of this document’s system modelling paradigm is that **all** formal representations contain³ entities and relations of **all** of the following three complementary types, for **every** concept they represent:

³Or rather, “*should contain*”, because almost no robotic systems already satisfy this hypothesis.

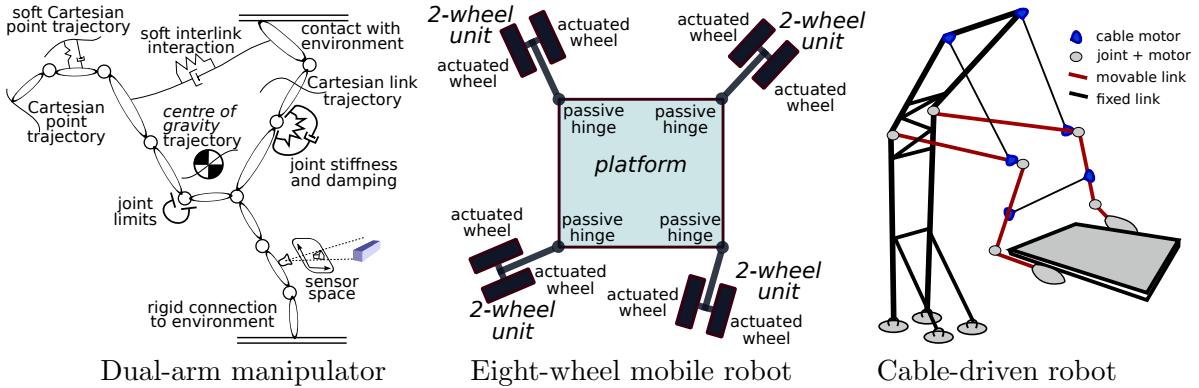


Figure 2.2: Sketches of various “advanced” robotic systems whose modelling is covered in this document.

- **continuous:** the relations are continuous functions of parameters in the system’s entities, such as the time, space, force, effort, cost,... aspects of a system.
For example, the [kinematic and dynamic equations](#) of a robot’s mechanical structure. Or the mathematics of a [PID controller](#).
- **discrete:** the relations are discrete functions of the systems’ entity parameters, that is, they represent the conditions when, why and how to select the “right” continuous system models. This often called *supervisory* or *coordination* control [95].
For example, the logical decision making functions in [Petri Nets](#) or [Finite State Machines](#).
- **symbolic:** the relations encode (not directly measurable) dependencies between continuous and discrete models, such as:
 - insights that [explain why](#) to configure some [magic numbers](#) in the continuous and discrete models, and with which values. The foundation of the explanation is **physical causality**.
For example, the maximum torque of an actuator can be limited *because* the required currents could overheat the wires, based on a thermal model of the motor.
 - representation of the **intention** of a task. The foundation here are **application-centric requirements**.
For example, when making a left turn, a car driver may first move the car a bit more to the right of the lane, in order to later be able to make a smoother turn. Or, a dual arm robot hands over an object from the right hand to the left hand, because it will need the latter in a later part of the task.
 - representation of the **context** of a task. The foundation here are (not necessarily causal) **dependencies between sub-parts**.
For example, the above-mentioned maximum motor torque could be lowered when a robot gets near a human, or when it enters a specific zone in a factory, in order to reduce the risk of damage in case of a collision. Or, a dual-arm robot may carry a load closer to its body, and move closer to its statically most stable posture, whenever that load is very fragile, risky, and/or expensive.

There is a clear hierarchical structure in these three types of relations, that helps to keep their formalisation efforts decoupled from the efforts of eventually coupling them together in specific system designs:

- symbolic relations are **higher-order** relations on discrete and continuous entities and relations. But a symbolic relation can also be higher-order relation on other symbolic relations; for example, the relations that represent *motor vehicles* are of a higher-order than those that represent *electrical bikes*.
- discrete relations are **higher-order** relations on continuous entities and relations. But possibly also on other discrete relations; for example, the **Life Cycle State Machine** of a robot system links together the LCSMs (of many) of its resources, because the robot can not be considered ready to be active if not all of the resources that it needs have reached their active status.
- continuous relations can be **higher-order** relations on other continuous entities and relations. For example, **integrability** is a continuous relation that represents a particular property of mathematical functions. And most of the *association relations* in **situation hierarchies**, **perception hierarchies** and **control hierarchies** are continuous relations; such as, the geometrical relations between parts of a visually recognizable pattern.

The “perception” and “control” at these three complementary levels of representation are often called, respectively, (i) (knowledge-based) **reasoning**, (ii) **supervisory** or **discrete event** control, and (iii) **continuous** control.

2.1.3 State of a system

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

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

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

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

2.1.5 World representation: open world, closed world, world model

A major challenge in knowledge representation of [reality](#) and the [real word](#), is how to give the so-called **open-world assumption** a place in the *policy* of making models. The origins of this policy date back to unknown depths of human history, because it represents the following situation:

- any model of the world contains [statements](#) that can either be “true” or “false”.
- any system (engineered, philosophical, ontological,...) that uses such models encounters situations in which it does not have all the information to evaluate the [truth value](#) of some of these statements.
- the system *designers* have to make a decision about what to do with these statements in such situations:
 - any statement that can not be proven to be “true” is considered “false”.
 - any statement that can not be proven to be “false” is considered “true”.
 - any statement that can not be proven to be “false” or “true” is [semantically tagged](#) as “don’t-know”, or as [don’t-care](#).

This document uses the term **world model** for any representation of the world that has explicitly made the above-mentioned choice, and that choice is part of the formal model.

2.1.6 Behaviour follows structure: list, tree, graph

This document’s paradigm is about finding engineering solutions to structure the interactions between components that each have their own behaviour, in such a way that the composition of these individual behaviours yields a system behaviour⁴ whose properties can be deduced from (i) the properties of the individual behaviours, and (ii) the properties of the structure via which they interact. There exist only a **shallow hierarchy** of structure types to represent behavioural interactions:

- **list**: any component interacts at most with one “neighbour” on its “right” and one on its “left”.
- **tree**: any component’s behaviour is *influenced* by only one “parent” component, and it influences itself only a finite number of “child” components.
[Context-free](#) is an adjective that is sometimes used to refer to this situation.
- **graph**: any two components in a system can (but need not!) influence each others’ behaviour.
[Context-sensitive](#) is an adjective that is sometimes used to refer to this situation.

2.1.7 Best practice: cope with the complexity of the open world

- *make closed-world assumptions explicit and monitor them*:
- *add spanning and junction trees*:
- *make any model composable*:
- *allow “the world” to be as small or large as is relevant*:

⁴Some examples from established domains in society where some form of “behaviour follows structure” have made it into design guidelines are: [Function-Behaviour-Structure ontology](#), and [Form follows function](#).

2.2 Data: structuring information for computations

One of the foundations of modelling is the representation of “**data**”: the symbols and numerical values that encode the *properties and attributes* of the entities and relations that make up the **information** part of a cyber-physical system. One (but not the only) **hierarchical structure** that relates formal representations of information is that of the **levels of abstraction**.⁵ The hierarchy comes from the observation that each of these levels is *a* (but not *the*) meta model of the level below, and has the level above as one of its own meta models. The following Sections describe this document’s five levels of abstraction; together with a “cross-sectional” data representation requirement, such as *metadata and queries*.

2.2.1 Mathematical representation

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

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

2.2.2 Abstract data type — Semantic representation

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

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

One needs computers **to reason** with the symbolic representations of entities and the relations between them, but also **to compute** with the numerical values of properties. Such reasoning and computation can take many forms: to formalise algorithms into computational models; to check formally the validity of a model; to transform models into code; to decide which are the right “**magic numbers**” in algorithms, etc.

⁵At least, for *computer-controlled engineering* systems, because there the needs to compute and to communicate between machines are essential. Sciences and humanities often stop with only the two top-most levels of abstraction, the first one for inter-human communication, the second one for computerised tools like **computer algebra systems**.

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.

Running example: a time series of temperature measurements from a weather station.

2.2.3 Data structure — Symbolic representation

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

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

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

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

Running example: an array in the symbolic schema of [Apache Arrow](#).

2.2.4 Digital structure — Data representation

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

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

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

Running example: a C language array with a C language data **structure** representing a weather measurement.

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

2.2.5 Electronic realisation — Physical representation

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

Running example: a buffer with bytes.

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

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

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

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

Higher-order data — Metadata

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

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

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

Data base: symbolically linked data

(TODO:)

Query: constraint between linked data

(TODO:)

Index: database of common queries

(TODO:)

2.3 Algorithm: synchronous composition of functions

This Section provides a **mereological** and **topological** model of an **algorithm**, as the *composition* of the three traditional parts of **programming in the small**, [38, 165]: **data structure**, **(pure) function** and **control flow**. In most application contexts, this composition must take into account a set of **dependency constraints** between some of the three mentioned algorithm parts. These dependency constraints can come from complementary sources:

- the *application context*. For example, the choices for **magic numbers** inside algorithms (e.g., the gains in a **PID control** function) have an impact on *when* the control flow will make decisions about which functions to execute (e.g., to switch to another controller function because the error in the currently running PID controller has become “too large”).
- the *asynchronicity context*. For example, if algorithms are executed in separate **activities**, one needs extra mechanisms to make sure that they are aware of when they provide data to each other. **Streams** are one such family of mechanisms.
- the *implementation context*, that is, how an algorithm is implemented in a programming language, and then compiled and deployed on a computer and its operating system. Examples here are constraints in the control flows to prevent **race conditions** between functions executed in **concurrently** running **threads**.

Although not strictly necessary for the explanations in this Chapter, the descriptions in this Section already conform to the “**5Cs**” meta model, that is introduced later in the document, to classify functions in types that have a semantic meaning in the context of **system architectures**. The “Composition” is realized via components, which are a container for the other four “C’s”. The algorithms in “Coordination” are typically just **Boolean functions**, that represent all *decision making* in a system; the “Communication” algorithms are typically **protocol stacks**; “Configurators” execute **configuration scripts**, possibly after **parsing** one or more **configuration files**. What ends up in the “Computations” parts is the rich variety of algorithms that belong to a particular application domain. System architects make two essential decisions when designing algorithms to encode the system’s computational behaviour:

- the *control flow* of an algorithm **coordinates** the *computations* that take place in functions, by making two choices: (i) the *order* in which the algorithm must execute its functions, and (ii) the *selection* of which functions to execute at a given time.
For example, a **Kalman Filter** has “prediction” and “correction” parts that must be executed in the right order, but that need not both be executed together all the time.
- **dependency injection** to **configure** a function’s *parameters*. That is, to give them a value that is appropriate for that function’s use in a particular application. This configuration of function arguments is realised via functions, too, of course, and these functions must end up somewhere in the algorithm’s control flow, too.

An example of dependency injection is the setting of the control gains in each of the possibly many **PID controller** functions in the system. There are three complementary aspects to such functions:

- the *function* is always the same series of computations.
- the gain *parameters* of the function are configured (by the user of the control algorithm) in the beginning, and maybe at sparse times during the runtime of the algorithm.
- the *values* of the function’s input and output *arguments* change every time the function is executed.

The *configuration* parameters are *properties* of the *behaviour* of the function, because they determine the relation between the function’s input parameters and its output parameters. The input/output parameters are *not* part of the function’s behaviour.

2.3.1 Mereo-topological model — Schedule and Dispatch

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

The proper choice of an algorithm’s *control flow* and *configuration* provides the composition flexibility to realise high **runtime adaptability**.⁶ However, in mainstream practice, they are seldom available as **first-class citizens** in the system design toolbox, not in the least because both are almost completely determined by the *context* of the algorithm, and not by the structure or behaviour of the algorithm itself. So, algorithm developers aiming for a *reusability* of their efforts must avoid to assume only one particular execution context in which their algorithms will be executed; and, *hence*, add extra functions *to monitor* and *to configure* their algorithms. In other words, they should make their algorithms **schedulable** within a **4Cs-based event loop**.

The mereological and topological entities and relations with which to formalize this ambition, are:

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

⁶If an algorithm and its functions need never to be adapted during the life time of their system, there is no need to introduce the dedicated mechanisms of control flow and parameter configuration.

In many use cases, that computation yields always the same result, so the scheduling is reduced to a fixed data structure, namely an ordered list of functions. But this document aims to support use cases with a high runtime reconfigurability, which leads to a meta model that separates:

- **scheduling**: to compute the mentioned list of functions to be executed, storing the result in a **schedule** data structure.
- **dispatching**: to execute one or more **schedules** in the `compute()` part of an **event loop**.

2.3.2 Mechanism: data, function, control_flow, and algorithm

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

- **D-block**: the entity to represent **data (structures)**, via a *data model* that describes what are valid structural compositions of data.
- **F-block**: the relation to represent a **function**, that is, the computational element that has one or more **D-blocks** as its arguments, with some of them having the **role** of “inputs”, others of “outputs”, and some of both.
An **F-block** should be a **pure function**, that is, without only *explicitly visible side-effects*: the function changes the some of its output data arguments, and nothing else. Other commonly used names for the **F-block** entity are: (*composite*) **operator**, **action**, or **actor**. **F-blocks** can easily be mapped to a **function prototype** (“signature”) in any specific procedural or functional programming language.
- **S-block**: the relation to represent the **schedule** (or, **control flow**) of a collection of **F-blocks**, that is, **constraints on their execution order**. The model of a schedule is a **D-block** in itself, that represents the order in which the **F-blocks** must be executed. This order model can be **declarative** or **imperative**:
 - **declarative**: representatives are (i) the **control flow graph**, and (ii) its special case, the **directed acyclic graph** (DAG). The latter is introduced in Sec. 2.3.5, as this document’s composable mechanism to represent a **flowchart**.
 - **imperative**: this type has only one representative, namely the **while loop**, that is the mechanism behind the **event loop**.
- **A-block**: an **algorithm** is a composition relation, that is, the “architecture”, of all of the above, including a **collection** of constraint relations, such as invariants and closures. The *model* of an architecture is a **D-block** in itself, that contains the **UIDs** of the (i) composing blocks, (ii) the “**constraint**” or “dependency” relations, and (iii) its own **semantic metadata**.

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

2.3.3 Block-Port-Connector model for blocks

The structural part of the meta model **conforms-to** the Block-Port-Connector meta model (BPC) Fig. 2.3): the domain is the description of an algorithm, and it is obtained by specialising the entity **block** to **F-block**, **D-block** and **S-block**, and introducing domain specific constraints (and meaning) to the **connects** relation. As an example, **ports** represent the

arguments of a function, and they are typed; that is they can be connected to a D-block under the constraint that the digital data representation model of the D-block and the port are compatible.

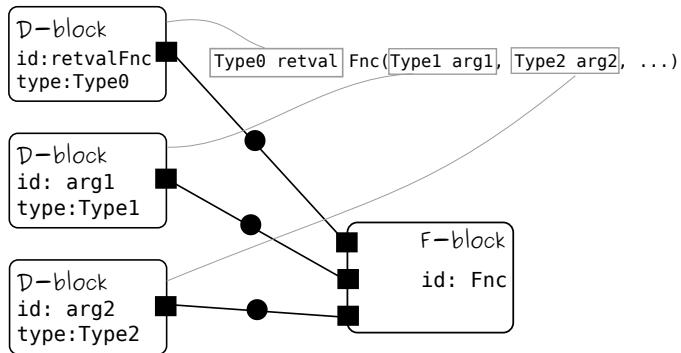


Figure 2.3: A function prototype and its graphical representation as a F-block connected to a set of D-blocks.

2.3.4 Data access constraints

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

- **causality**: a function that changes data values brings in a **cause-and-effect** relation, between the data “before” and “after” the application of the function. Since many functions and data structures represent the physical world, this algorithmic causality might or might not correspond to **physical causality**; this knowledge in itself brings in another relation.
- **function invariant**: the effect of the function on the data values can be modelled *imperatively* or *declaratively* (Sec. 1.5.6): the imperative form is that in which the function is, in itself, a composition of functions and a schedule; the declarative form is a **relation** that holds between the data values “before” and “after” the application of the function.
- **data consistency**: different functions can change different parts of the same data structure, and their function invariants must, *together*, satisfy some **relations** that *must* hold between all parts.

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

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

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

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

2.3.5 Directed Acyclic Graph: declarative model of control flow dependencies

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

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

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

Data flow model

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

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

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

2.3.6 Policy: closure in a context

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

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

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

2.3.7 Policy: application programming interface

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

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

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

Functional programming

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

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

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

2.4 Event loop: synchronous composition of computations with asynchronous data

The **design context** in which computations (**functions**, and their composition in **algorithms**) are developed is that of **synchronous** execution:

- the **order** in which functions are *programmed* in the code of an algorithm, is also the order in which they will *execute* on the CPU.
- the data that serve as arguments in the functions is **always available**, at any moment the function needs it, and without the function developers having to worry about whether that **data's consistency** could be corrupted by the simultaneous access to that data by other functions, somewhere else in the system.

The **execution context** however, is (typically) **asynchronous**: just by looking at the code, one can not know when new data will arrive, and where else it will be used. So, in order to bring their design and execution contexts in agreement, developers need a mechanism to let their synchronous algorithms **react to** asynchronous interactions with other algorithms; the reactive approach is necessary because the timing of the algorithms' execution is unknown in advance.

2.4.1 Event loop: event part and loop part

One essential mechanism in such reactive context is the **architectural pattern**⁷ of the **event loop**. Its mereological parts are:

- **event**: the information that “something has happened, somewhere”. And *new data becoming available* is the major “happening” in the context of this Section.
 - **loop**: the *serialization of reactions* to the asynchronous events of data becoming available, with the synchronous execution of those functions in an algorithm, where the exact contents of that synchronous execution depends on the available data. The loop concept has two parts:
 - **serialization**: the functions of the activity are executed one after the other, according to a particular **schedule**.
 - **looping**: the activity is not running all the time, but only gets “triggered” from time to time. And at that moment, the activity has to decide which parts of its functionalities it is going to run this time around.
- The **activity** is the owner of this decision making, because its design depends on knowledge about the internals of the application.
- The **thread** is the owner of the decision making about the **timing** of the next time instant when the operating system will trigger the event loop execution.

It is the responsibility of the designer to provide an **event loop** version of the algorithms needed in the system: the execution of the algorithm is suspended (“**preempted**”, or “**stopped anytime**”, until a specific type of data becomes available, or until the next time it is “triggered”) and when triggered only a *specific* set of functions in the algorithm are executed. This behaviour is repeated in a loop until the services of the algorithm are not needed any more. This algorithm design approach is complementary to the **batch** version of the same algorithm, where *all* data is available at the start of the execution of the algorithm.

Synchronicity (that is, executing functions in the order they are programmed) is a *first* necessary (but in itself not a sufficient) condition for **explainability**, that is, to be able to predict what the result of an algorithm will be. The *second* necessary condition is that the algorithm stores its “state” in a predictably consistent (“**thread-safe**”) way: all the information that it needs to execute is stored in data structures of which it is the sole owner; hence, no other activity can disturb its execution consistency. The *third* necessary condition is that

⁷It is a specialisation of the **reactor pattern** and the **proactor pattern**, in that it adds particular ordering policies to the execution of all functions that it manages.

each of the functions in an activity has **no side-effects**: a function changes only the values of the data structures that are in the directly visible **scope** of its programme code, and hence no data is changed “behind the back” of the function.⁸

All of the conditions above can not be guaranteed by the event loop mechanism alone; the missing piece are the deterministic **interaction streams**: they let two synchronous event loops exchange data asynchronously via efficient and deterministic buffer mechanisms.

2.4.2 Synchronous, asynchronous, sequential, concurrent, parallel

A composition of functions (i.e., an algorithm) has a synchronous mode of computation (or “execution”) if:

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

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

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

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

Activities compose algorithms, in two complementary ways:

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

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

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

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

⁸A common source of side effects are **system calls**, that is, **interactions** with the operating system. For example: opening, closing, reading and writing of files; inter-process communication; or reading the **clock**.

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

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

2.4.3 Mechanism: 4Cs-based event loop template

The following pseudo code gives a **mereo-topological**⁹ version of the event loop, with a best practice **sequential structure** of the “4Cs”, that is, the *Communications*, *Coordinations*, *Configurations* and *Computations* that a particular activity wants to see executed in the thread in which it is deployed:

```

do {                                // when triggered by its "master" thread,
    // which itself is triggered by operating system.

    communicate() // get all "messages" (i.e., read/write shared data structures)
                  // of events and data, provided by other activities.
                  // (That is, all "computations with side effects".)

    coordinate() // react to events and flags in protocols, (maybe) resulting in
                  // switching in the LCSM or task FSM, or executing a Petri Net.

    configure() // some reactions imply reconfiguration of the event loop,
                  // i.e. changing the activity's schedule of algorithms to call.

    compute()   // execute the currently configured schedule of algorithms,
                  // which in themselves are "side effect-free" computations.

    coordinate() // the computations above can generate events/switch flags
                  // that imply reconfiguration of this event loop.

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

```

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

⁹This model represents a serial *structure*, of 5C-conforming *types* of behaviour.

- the suggested **schedule** (that is, the *relative order* of the “4C” fuctions) makes sense in general, but need not be a hard constraint for every individual application.
- its intention is to be used as a *memory aid* for the human application developers: starting from the template, they can make a *motivated* decision *not* to use the complete version, and throw out some steps.
- the exact execution sequence of the functions, as well as their exact contents, must always be configured by the application developers.
- for *fast* running event loops (that is, with a short or non-existing `yield()`) the “second round” of `coordinate()` and `communicate()` do not add much value, because the same functions will be called in the “first round” of the next iteration of the event loop.

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

The power of the event loop pattern, as described in the simple template above, is that it *stimulates* developers to separate the 4C concerns. That way has also proven¹⁰ to facilitate (but not to guarantee) composability and compositionality in complex distributed systems. This composition of multiple event loops (of *activities*) into one single larger event loop (in a *thread*) expects that all algorithms inside the event loops can be created by means of **coroutines**, and that **preemptive multi-tasking** can be avoided. One necessary, but not sufficient, condition to satisfy this constraint is that all functions in the algorithms have no side effects, have short implementations, and that they access only data that is owned by the event loop activity. Often, that data is stored in the **stream buffers** needed for the **asynchronous I/O** in the `communicate()` parts of the event loop.

Some instantiations of this Section’s event loop template are:

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

2.4.4 Composition of event loops

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

2.5 Activity: asynchronous composition of algorithms and their interactions

This Section’s topic, the *activity* as *the* composition mechanism for **algorithms**, is the essential one in this Chapter when viewed in the document’s broader context of the design of **system architectures**: its design focus is the *application* (system, component), and not the *technology* (algorithm, function, data). It is indeed the application that mandates which activities must be realised, how big they should be, and how much interaction between them is required. Hence, it is also the application that brings in requirements about **effective use of resources**:

¹⁰For example, many servers and browsers work with an event loop architecture.

mechanical, electronics, computational, communication, and, not in the least, **time**. In this document's vision, the activity is *the* level of system composition that has:

- the **ownership** of resources in general, and
- **responsibility** for satisfying resource constraints and usage dependencies.

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

2.5.1 Activities are responsible for time-based behaviour

At the most abstract view on a system, the system needs **activities** to take responsibility for the realisation of the required **behaviour** of the system. Every cyber-physical system (be it of *physical* or *information processing* nature) has **behaviour**. And every behaviour has **time** as a primary parameter.

An example of *physical* behaviour is the motion of the air that flows around a quadrotor drone and through its propellers. An example of *information-processing* ("cyber") behaviour is the *control* of the lift force of that quadrotor by steering its rotor velocities. Cyber-physical systems are *engineered* to realise behaviour that has a particular **purpose** during a particular time, namely, to execute the tasks of the applications the systems were developed for. Of course, many systems show the same behaviour only for some time before they change to another behaviour, which is at that time more useful to fulfill the application's requirements.

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

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

Time is, always and both, a major resource and quality metric.

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

2.5.2 Mereo-topological meta model

Mereo-topologically, an activity is the **composition** of its:

- **interior**: the **functions** that run in **algorithms** inside the activity, **synchronously** coordinated by an **event loop**.
- **exterior**: the **asynchronous interactions** via which the activity exchanges **data** with other activities.
- **ports** (or "interfaces"): the connectors between the interior and exterior parts.

"Activity" is the *pars pro toto* term for the following *composition hierarchy* in **types** of information-processing behavioural entities:

- **function**: the data, function and control flow entities and relations of Sec. 2.3, and which form the foundation of all **computations** in software components.
- **algorithm**: the **synchronous composition** of several **functions** which share data. That is, functions are scheduled in a **serialized** way, and all data can be used exclusively by any function in such a series.
- **program**: the **asynchronous composition** of several **algorithms**, due to **concurrent execution**, that is, via **cooperative multitasking** on one single CPU core. Care is to be taken that one function does not write data *at the same time* that another one is reading or writing that same data, *but* that **coordination** is fully under the control of the **program**, because the whole **program** has the same **computing context**. Badly coordinated asynchronous execution of **functions** can indeed introduce inconsistencies in the data that is shared. But the challenge is not to control the *time* at which operations take place, but their relative *order*. The mechanisms available for the coordination are (i) **interaction primitives** between several of the **program's algorithms** (e.g., **flags** and **streams**), and (ii) clear **ownership** of each **resource** (that is, the model that represents what an **algorithm** can do on a resource, and at what time).
- **activity**: the **asynchronous composition** of **programs** due to **parallel execution** on different CPU cores, even on different networked computers. The extra challenge compared to **programs** is that **state must be duplicated** between **programs**, so every state-based decision making must be made robust against the parallel execution.
- **component**: this is the default interpretation of the term “activity” when used in the context of **information** and **software architectures**, later on in this document. It conforms to the mainstream interpretation of **component-based software engineering**, where *all* of the following mechanisms come together:
 - **to compose** functionalities into behaviour.
 - **to deploy** that composition into an **event loop**, so that the behaviour can be executed in a *thread* on a *CPU*.
 - **to interact** with the resources, and with other activities via asynchronous *communication* interfaces.

Any type of activity must provide its “**data sheet**” to other activities, with the model of the exteriorly visible behaviours, states, and interactions.

2.5.3 Asynchronicity implies stateful activity interfaces

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

- algorithms must **interface** each other’s, **producing** data for, and **consuming** data from, each other.
- algorithms must be **configured** to the particularities of the **computational platforms** of hardware and operating system on which they are executed. This configuration often needs to be done by a “third-party” activity, either because that one has the *knowledge* about what the best configuration is (e.g., a **skill** activity), or because it has the exclusive **ownership** of the platform (e.g., an **operating system**).
- in addition, algorithms need to be **synchronized** with **physical resources** (e.g., sensors and motors), whose software interfaces typically do not respect the optimal synchronization expectations that algorithms would like to see satisfied.

The term “**interface**” has the following meaning in this document: it is the formal model of

the semantics and the purpose of a information exchange between activities. Indeed, while *physical activities* interact by exchanging energy, *information activities* interact by exchanging **information** (“data”, “messages”, “events”,...), about the status of the physical activities (motion, force,...) and the symbolic activities (task, performance,...) whose interactions they control. So, all **interface** choices (or interface **policies**) require two complementary **mechanisms** to be available:

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

So, the typical interface situation involves three parties:

- the exchange mechanism, which is often a full-fledged activity in itself.
- the producing activity in the exchange of information must be able to hand over its information to the exchange mechanism.
- the consuming activity must be able to get information from the exchange mechanism.

The generic interface model allows both interacting activities to be producer and consumer at the same time, for different types of information that are being exchanged in their interaction. Anyway, except for simple situations, **stateful interfaces** between activities are inevitable: one activity must provide some information to other activities about the state of its own behaviour, because that influences the type and quality of its interface performance.

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

The exchange mechanism behind an interface can be a **passive data structure**, an “activity behind the screens” (for example, a **stream**), or a full-blown activity in itself (for example, a **broker** activity). The producing and consuming activities can be made unaware of the concrete passive/active situation, via a **Block-Port-Connector** model of an interface: producer and consumer are *blocks*, the exchange mechanism is a *connector*, and the decoupling is realised via *ports* (that can abstract away the active/passive implementation). *Ports* in activities are typically called *services*, and *Ports* in libraries are typically referred to as the *API* of the library.

2.5.5 Continuous data, discrete event, logic flag, symbolic query

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

- **data structures** for the **continuous** world: the real world is continuous in time, space, energy, etc., and this document gives the name “continuous data” to every formal representation of that continuous world.
- **flags and events** for the **discrete** and **logic** world: activities must be able to influence each other’s **control flow**. This typically happens by *sending events* (or “signals”) between the activities, or by letting them *observe* the value of **flags** that they both have direct access to. In their physical and social representations, **flags** are pieces of fabric with meaning encoded in colours and structures in the fabric, and **events** is the name given to all sorts of **happenings** that people organize or that take place autonomously, in nature and in society. In their cyber representation, flags and events have very similar

meanings and purposes, and they are represented, of course, also as data structures. Typically, only very small ones, because the only semantics they have to represent is the discrete (changes in the) state of the world. This document uses the following terminology:

- *flag*: to represent that “*some condition has a particular logical value*”, or, “*a `has` a certain `status`*”.

For example, a traffic light is green; a door is open; or, an activity has a particular behavioural mode.

The **affordances** of a flag are: to be *created*, *set*, *observed*, *cleared*, and/or *deleted*

- *event*: to represent that “*something has happened*”.

For example, a traffic light has changed colour; or, a `door_open` button has been pressed.

Its affordances are: to be *wired*, *fired*, *handled*, *consumed*, and/or *processed*.

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

- **propositions** to model the **logical** world: formal statements about the world that are **true** or **false**.

For example, “*this robot has two serial-chain arms*”.

- **queries** and **dialogues** with models for the **symbolic** world: activities must be able **to represent**, **to discover**, **to configure** and **to update** the **functions** that they use themselves or that others use, and do that at **runtime**. In other words, to represent current or possible behaviour (of oneself, or of the other one), and to provide new “computations” and new “data types” to each other. Such symbolic interactions take place via a **query protocol**, that composes a particularly sequence of **models** that represent interpretations of the world.

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

- *data*, to refer to all three semantic variants.
- *information*, to refer to data that comes with *metadata* that explains how *to interpret* the data.
- *knowledge*, to refer to information that comes with a *context* that explains what information is *needed* at what time.

2.5.6 Best practice: resource ownership

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

- every resource has one and only one owner at every moment in time. Ownership can be transferred, temporarily or permanently, from the owning activity to another activity.
- the values of the properties of the resource’s model can only be changed by functions in the owning activity.
- other activities that need access to the resource must do so via **queries** to the own-

ing activity. Access can be granted in a way that requires no explicit query for each individual access, by means of query protocols for *transfer of ownership* (for example, “producer–consumer”, “broker”), or *sharing of access* (“borrowing”).

- when activities receive *copies* of the state of a resource, “*the*” state of that resource exists only in its owning activity. This explicit “ownership hierarchy” allows a system to work with state data that can be inconsistent between owning and borrowing activities.
- an activity has extra “[non-functional](#)” algorithms, for the sole purpose of configuring and monitoring the resources that it owns.
- when the owner of a resource is stopped, the resource is not accessible anymore. *Or* the system architecture has a protocol with which the owner transfers its ownership explicitly before stopping.

2.6 Finite State Machine: behaviour coordination in one algorithm or activity

This Section presents the meta model of the [Finite State Machine](#) (FSM), as the [abstract data type](#)¹¹ to represent the [behavioural state](#)¹² of [one single activity](#). The description maximizes the *separation* of (i) *mechanism and policy*, and (ii) *structure and behaviour*. This decoupling is not an obvious ambition, because state machines have a very long history, in many application contexts, and most often those application contexts have implicitly entered the many models, of lesser or higher [degree of formality](#), with which various FSM meta models have been “standardized”. Examples of such domain-centric couplings are [Harel statecharts](#), [Mealy machines](#) or [Moore machines](#), [29, 160].

2.6.1 Mechanism Part 1: state to model coordinated behaviour

The FSM mechanism is introduced because an activity must be able [to realise different behaviours](#). In each of its behavioural “[states](#)”, the activity executes a different composition of algorithms. The meta model of this Section introduces two closely related [strict constraints](#):

- an activity executes [only one behaviour at a time](#);
- at [any given time](#), an activity has a [uniquely identified behaviour](#).

In other words, an activity is in one and only one state at each moment in time, and, to the “outside world”, it never is in an “unknown” state. Each of these to-be-coordinated behaviours is represented in the FSM meta model by a discrete parameter (an “[enumerated value](#)”), called a [\(discrete\) state](#), or a [mode](#).

Take the example of an [activity](#) that realises LIDAR- and IMU-based velocity control for a robot. Its behaviour will be different when other control and estimation algorithms are used. And it makes no sense to switch between these algorithms randomly, or instantaneously, so the switching must be *coordinated* explicitly.

¹¹Indeed, an FSM is a passive model that represents information about activity and behaviour, but it is *not a behaviour* or an *activity* in itself. In other words, the activities and behaviours that use an FSM data structure for their coordination must look up the state of the FSM model themselves, whenever it is an “appropriate time” to do so. There is nothing in the FSM model itself to help make that decision.

¹²“State” is probably the most over-used term in systems science and engineering. Hence, the meaning of the word can only be complete if the context in which the term is being used is given explicitly, too. In this Section, *state of an activity* would be a semantically more complete term.

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

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

This monitoring, and the state switching **decision making** that it triggers, requires a dedicated algorithm, which is the subject of a [follow-up Section](#).

To the **outside** of an activity (that is, an activity’s behaviour that can be observed by other activities), the state just *represents* a behaviour with an “identity”. One does not have to know the details about *how* that behaviour is realised, that is, to know about the exact composition of algorithms inside the activity. The important information is about [*how to interface*](#) with the behaviour. This is the subject of the [next Section](#).

2.6.2 Mechanism Part 2: events to model coordination changes

This Section introduces the model of the **coordination interface** that an activity provides to other activities via its [data sheet](#). The purpose of this mechanism is:

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

Events are the proven [mechanism](#) to realise this ambition: an event has a very simple semantics, in that it just represents “that something has changed”. That change information can then be **communicated** via any type of [interaction channel](#) between activities. The “other end” of such a communication is responsible to react “appropriately” to incoming events. That reaction is often called **event handling**. In its most generic form, it consists of the following operations:

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

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

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

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

For the purpose of *separation of concerns*, it is important to avoid a common *worst practice*: the activity that fires an event gives it a name that reflects the behavioural switch that is the *intended outcome* of the event at another activity. The better approach is to give a name that reflects the *constraint violation* that took place inside the firing activity, and that was the cause for firing the event. For example: `distance_to_obstacle_below_MINVALUE` is a better name for a LIDAR processing activity to use than `switch_to_collision_avoidance_control`;

this latter reaction *could* be the result of the control activity's reaction to the event, but that reaction decision is the sole responsibility of that control activity, and not of the LIDAR processing activity.

2.6.3 Mechanism Part 3: transition and event reaction table to model coordination behaviour

The **structural** part of the meta model has the following primitives (Fig. 2.4):

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

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

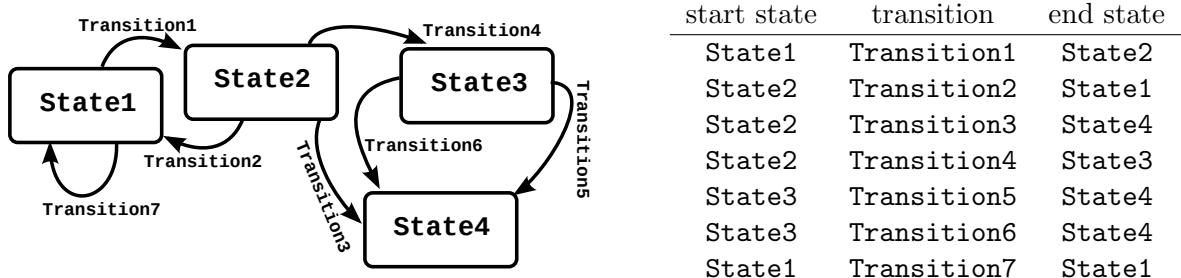


Figure 2.4: An example of the **structural** (mereo-topological) part of a *Finite State Machine* model. Left: graphical representation; right: the semantically equivalent tabular textual form. In a **property graph** representation, **states** are nodes, and **transitions** are relations.

The **behavioural** part has the following primitives (Fig. 2.5):

- the **event**: a **Boolean variable** that represents (the change in) the **truth value** of one of the **above-mentioned** pre, per and/or post conditions, or of a **logical composition** of them.
- the **event reaction table**:¹³ the data structure that models the reaction of a FSM to incoming events. That is, it supports the decision
 - to transition to another **state**, and
 - to fire a set of **events**, as a result of the transition.

2.6.4 Canonical meta model of a Finite State Machine

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

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

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

¹³This term *event reaction table* is not standardized outside of the scope of this document. **Decision tables** and **decision trees** are popular instantiations of the concept. The **Karnaugh map** is one of the traditional representations of the logical computations.

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

Figure 2.5: One possible **behaviour** for the FSM in Fig. 2.4, modelled as an *event reaction table*, in **canonical** form.

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

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

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

Policy models are introduced in the following Sections.

2.6.5 Policy: event composition

Table 2.6 shows an extension of the canonical finite state machine behaviour that is used in many applications: the application has *multiple events* that it wants to react to in order to trigger *one particular* transition.

handled events	resulting transition	events fired
$e_1 \vee e_3$	Transition1	
e_2	Transition2	E_2
$e_3 \wedge e_1$	Transition3	E_1, E_3
$\neg e_4$	Transition4	E_4
$e_1 \wedge e_3$	Transition5	

Figure 2.6: An example of an *event reaction table* with *composed events*.

The *best practice* policy is, without any loss of semantic expressivity, to introduce a **event monitoring** function that generates the **single canonical** event whenever the required **logical composition of application-centered events** occurs. Similarly, a monitoring function can capture a transition generated via the canonical event reaction table, and react to it by firing the requested set of application-centered events. The advantage of this practice is that the same FSM can be used with different event monitoring logic, which allows to decouple three complementary but different design efforts:

- **inside behaviour:** the FSM is the mechanism to structure the internal behaviour of an activity, and this is best done by designers who are responsible for the ins and outs of the “thing” whose behaviour must be controlled.
- **outside behaviour:** an activity fires events to let the outside world know that it has “done something”. The naming of these events should conform to the terminology used in the **data sheet** of that activity. So, the naming of events is best done by designers who are responsible for the outward facing behaviour of the activity.

- **system behaviour:** the system designers, and *only* them, have the information about how to couple the outward facing behaviours of several activities together. This often requires renaming events, or grouping them together in logical monitors, etc.

2.6.6 Policy: event distribution and conversion — Event queue

The events handled by an activity's FSM can be generated by other activities and/or by the activity itself. Similarly, the produced events can be reacted to by other activities and/or by the activity itself. The consequence for the system architecture is the need for:

- **distribution** of events between activities, using [event queues](#).
- **conversion** of event streams from several sources into the ones in an FSM's event reaction table, and the other way around for the generated events.

It is the *system architect*'s responsibility to avoid the *event name space pollution*, that can occur under the just-mentioned policy composition; the *best practice* here is [to inject](#), into the event loop schedule, the *translations* of event names between the application's context and the canonical FSM form.

2.6.7 Policy: hierarchical states

The *structural* relation of **hierarchy**, Fig. 2.7, is a commonly used addition to the canonical Finite State Machine. The *behavioural* part of the FSM does not change: the activity **must** always be in one and only one of the **leaf states** of the hierarchical structure. What *does* change is the **view** on the outward facing behaviour: the hierarchy allows system designers to reveal different levels of detail of an activity's behaviour to different other activities in the system. The hierarchical structure model has the following semantics:

- **containment relation:** a **state** can be [contained](#) in another “super” **state**.
- **containment tree constraints:** a **state** can only be [contained](#) in one single “super” **state**, and these containment constraints can only form a **tree**.
The **leaf states** of this tree are the real behavioural states.
- **shared transitions:** a transition from a super **state** to another **state** represents the set of **transitions** from **all** of the internal **states** to the same other **state**.
- **non-shared event conditions:** each internal **state** **can** have a different event condition for the above-mentioned shared **transition**.
- **initial_state, final_state:** when the overall state machine **transitions into** the super **state**, **one** internal **state** **must** be selected as the **final_state** of *that transition*; that state then becomes the **initial_state** of any *subsequent transition*, so one often sees this state being tagged as “initial state” in graphical representations of an FSM. One internal **state** **can** be selected as **final_state**, which means that an internal **transition** into this **final_state** **automatically** gives rise to a **transition** away from the super **state**, *if* that super **state** has only one possible **transition**.
- there **can** be a policy **to remember** the internal **state** that the FSM is in when a **transition** takes place out of the super **state**, so that this so-called **history state** will be selected as the **initialstate** for the next **transition** into the super **state**.

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

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

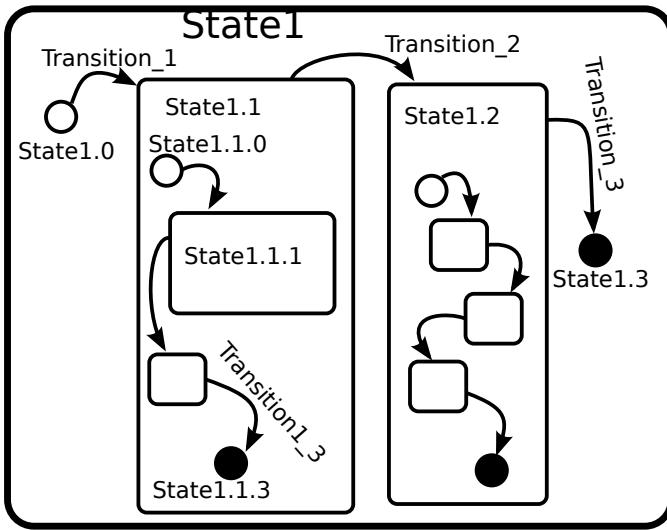


Figure 2.7: An example of a hierarchical *Finite State Machine*. The open and filled circles represent `initial_state` and `final_state`, respectively.

- **improved semantic value of communication:** it *may* be useful to provide for each the many complementary interaction purposes (logging, reporting, introspection, interfacing, information hiding,...), another view on the FSM than the full structural model.
- **just-in-time creation**, or **lazy evaluation**, of state machines from a model. The reason can be to reduce the effects of state explosion on the **runtime memory consumption**, but the policy also allows for **runtime adaptation** of the discrete behaviour of a system by the *just in time* creation of new state machines via **runtime reasoning** on the basis of (declarative) behavioural models in the application.

2.6.8 Policy: Life Cycle State Machine of activity or resource

Almost no **activity** (i.e., a component, or a system) can provide its **services** to other activities, without itself making use of the services of multiple **resources**. These resources can be:

- **active:** application-level activities, or infrastructural activities like device drivers or CPU cores.
- **passive:** abstract data types, **algorithms**, communication bandwidth, energy, memory, disk space, etc.

It is not realistic to expect these resources to be provided and appropriately (re)configured, *instantaneously*.

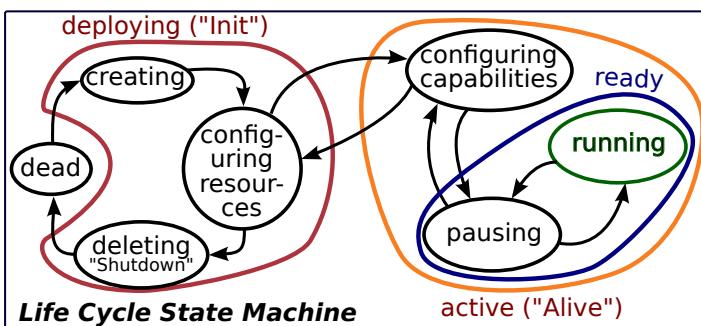


Figure 2.8: The Life Cycle State Machine meta model coordinates the configuration of the internal resources required for a certain (active or passive) service before that service's capabilities can be offered to third parties.

The *Life Cycle State Machine* (LCSM), Fig. 2.8, has been (since decades, in one form or another), the mechanism to coordinate the availability of *internal* resources for the provision of *external* services. That is, to make sure that a component's own capabilities are configured appropriately *before* others can make use of its services. The LCSM concept appears in many incarnations and using many alternative names for the states; this document uses the following terminology and semantics:

- **dead**: the resource is not able to do anything for itself, until after it is *brought to life* by another activity.
- **deploying**: in this *superstate* the activity itself (or, for passive resources, another activity) is busy with finding and configuring all the resources it needs, to be able to offer its services to others. While doing so, the resources **does not react to external coordination events**. This super state contains three semantically complementary states:
 - **creating**: the (software, memory and communication) resources are created, with which the resource does the bookkeeping of its own behaviour and of its usage of its internal resources.
 - **deleting**: the above-mentioned resources are cleanly removed, that is, no other activity will be hindered by possible remnants.
 - **configuring resources**: the resource is (helped in) configuring the service resources it requires for its own behaviour.
- Resource *configuration* also includes *re-configuration at runtime*, so the latter state can be transitioned to and from multiple times during the life time of the activity or resource.
- **active**: the activity or resource is visible to other activities. That is, in this super state, the activity **reacts to external coordination events** or external activities can use its passive resources. This takes place in two semantically complementary behavioural states:
 - **configuring capabilities**: the activity or resource **is not** providing its services, because its own capabilities are (re)configured to provide a particular service.
 - **ready**: the activity or resource **is** providing its services to other systems. This super state again contains two semantically complementary behavioural states:
 - * **running**: the activity or resource provides its services.
 - * **pausing** (or “*idling*”): the provision of the services is put on hold, but can resume immediately. This happens, for example, when it must wait for an LCSM state change in one or more of the “peers” it interacts with; such as during a [All to go, One to stop](#) coordination.

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

(TODO: existing LCSM examples: DS402 of CAN in Automation for motion control; Functional Safety over EtherCat, with a FSM for sessions and connections.)

2.6.9 Policy: don't care, history, timeout

In some contexts, it is useful or even necessary to indicate explicitly that some events do *not* influence the FSM behaviour. One way of realising this is to introduce (runtime changeable) **don't care** tags to some events, so that these are temporarily not considered in the event transition table.

Another relevant policy is that of **history** in the event processing: the fact a **particular sequence** of events and/or transitions has taken place can have the meaning of an event in itself. That is, the activity's application might want to adapt the behaviour of the activity whenever such a sequence has occurred. One common example is that of a **livelock**: the system adapts its behaviour because of a particular event, but the adaptation in itself triggers a reaction in another event, whose change in state causes the first activity "to undo" its former reaction, *ad infinitum*.

Finally, it often makes sense to put a **timeout** on the duration between events and/or state transitions. For example, a robot navigating in a building can decide to take another route if it has been waiting "too long" in front of a path obstruction on its current route to disappear. An FSM with this behaviour is often referred to as a **timed automaton**.

2.6.10 Policy: selection, priority, deletion

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

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

2.6.11 Implicit constraints on event handling meta model

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

- a modelled behaviour is in one, and only one, state at each moment in time.
- an FSM represents the behavioural state of only one activity.

- every state and every transition has a unique ID.
- state transitions take no time.
- event processing takes no time.
- event firing takes no time.
- representation of all parts of the model takes no space in computer memory.
- event queue bookkeeping takes no time.
- the management of the event processing takes no time.

2.6.12 Runtime creation of FSMs via transition systems

Finite state machines are a good model to represent the discrete switching between behaviour, but any somewhat realistic application requires thousands of behavioural states, which compromises scalability. The obvious solution is to use [higher-order relations](#) that represent the links between “pre conditions” of “actions” and the models of finite state machines to realise those actions. The technical challenge is to develop solvers to do the *model-to-text* transformations that generate executable state machines.

(TODO: explain how the concept of [transition systems](#) applies to (i) the online generation of FSM, based on (ii) graph traversals over knowledge about the discrete behaviour capabilities of robot systems. Links with [action languages](#), [fluents](#) (or *terminals* [99]), and *triple graph grammars* [43, 48].)

2.7 Flags and bitfield protocols for peer-to-peer coordination

This Section introduces the **flag**, and flag-based **protocols**, as model primitives via which **to coordinate the control flows of two or more** algorithms or activities. The major reason to introduce flags as [first-class](#) primitives in the design of complex systems is that they are the **simplest possible mechanism** with which to increase the efficiency of making decisions, in a system that has several activities (or algorithms) working **together** and **asynchronously**. Indeed, they solve the following common use case:

- one activity has multiple internal “states”, in which it is executing different “behaviours”.
- that activity has to interact with other activities.
- that interaction depends on the concrete behaviour of the activity, hence on its internal state.
- often, several internal states require exactly the same interaction with the external activities.
- so, the other activities do not have to know *all* internal states, and just one (“conglomerate”, or “super”, or “hierarchical”) state is good enough.
- that single conglomerate state is made visible to the external activities via a flag.
- that flag has Boolean semantics: at any given time, it is either “true” or it is “false”.
- the ownership of each flag is clear: only one activity/algorith can write the flag’s state; all others can read it.

A primary example of this use case of flags is to coordinate two *loosely coupled* activities, that is, one of them just has to adapt its own behaviour to the fact that the other activity is in one particular state of its [Life Cycle State Machine](#) mechanism; most often, that state

is the **running state** (or its hierarchical superior **active**). The use case occurs for all device driver activities, because they must start up and configure the device before other activities can make use of it.

2.7.1 Abstract data type: flags

The purpose of a coordination protocol is to help algorithms in their **logical decision making**. A *flag* has the semantics of a Boolean variable (e.g., `condA`), that is part of decision making via **logic programming** functions. A flag represents (“caches”, “remembers”) the value of a **logical condition on the computational state** of a particular function, or sub-algorithm. A flag can have the status **true** or **false**, but in the advanced semantics of systems engineering, a *flag* should also have the status **uninitialized**, to indicate that decisions to be taken somewhere in the system should *not yet* depend on the value of that flag. The protocol is the **structural composition** of flags, to realise coordination by requiring a specific **order** in which flags become **true** or **false**.

Because control flow relations can change at runtime, and because their order influences the outcome of an algorithm, the execution of control flows must be **coordinated**, inside one single algorithm or between several algorithms.¹⁴ The control flow decisions that each algorithm must make can depend on the values of a set of flags¹⁵ whose values are changed in other algorithms. Hence, the design challenge in coordination protocols in such a multi-algorithms context is that:

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

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

$$\text{if } (\text{condA} \text{ and } \text{condB}) \text{ then }\{\dots\}; \quad (2.1)$$

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

Sequence diagrams are a mainstream graphical representation to model a coordination protocol. However, sequence diagram meta models most often come with rigidly attached semantics (structures and behaviours) from particular application domains. Avoiding this coupling between application-agnostic *mechanisms* and application-centred *policies*, is one of the main ambitions of this document.

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

- **efficiency**: a flag “caches” the outcome of the computation of the condition of a computational state, so condition computations need not be repeated.

¹⁴The case of coordination between several algorithms is conceptually not different from coordination inside one algorithm: this document uses the term “algorithm” to mean “any *composition* of functions, data and control flow that matters to be considered together.” And *coordination* is one of the aspects that determine whether a certain composition scope matters or not.

¹⁵On most modern computer hardware, checking those values can be done efficiently, if they are **digitally encoded** as **bit fields** or **bit arrays**. The encoding allows **atomic evaluation**, of multiple flags at the same time.

- *concurrency*: because of the Boolean type of flags, concurrent algorithms in an activity can read/write flags to keep each other informed about their computational status, without bothering about when the other peers actually read and write the flags they all share.
- *separation of concerns*: the algorithm *designer* must only be concerned with logical correctness, without having to care about:
 - the computational properties of the *execution platform*'s hardware, middleware or operating system.
 - *the reason why* a flag has been given a particular truth value.
 - how *the system will react* to the logical decision that is being made.

2.7.2 Mechanism: protocol array with atomic iterator

Figure 2.9 sketches the mechanism, by which two functions coordinate their own control flows with each other using the following **protocol**:

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

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

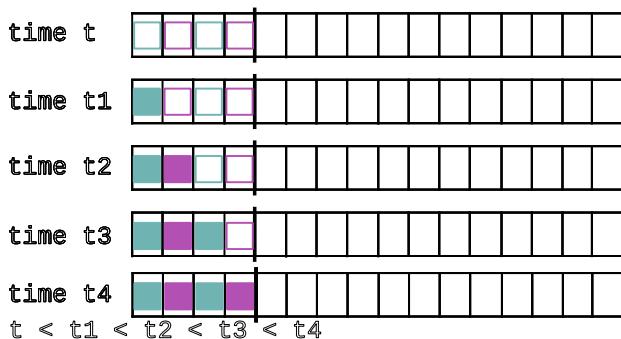


Figure 2.9: A flag array (or “bitfield”): the array provides the structural constraint of the order in which two asynchronous functions (the *green* function and the *purple* function) can set flags. The exact time of raising new flags in the array does not matter, as long as the mentioned strict order is satisfied.

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

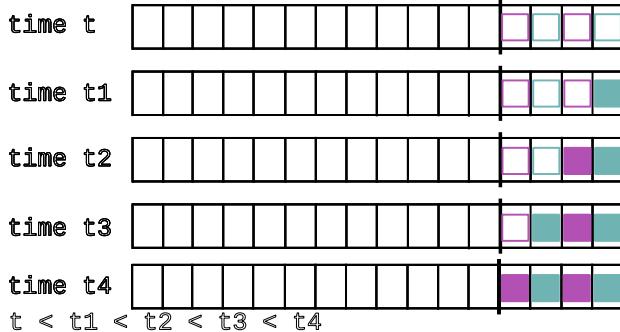


Figure 2.10: The *big endian*, or *left-to-right*, version of the flag array in Fig. 2.9

2.7.3 Best practice: coordination mediator

A best practice in multi-activity coordination is to introduce a separate **mediator** activity, as the *owner* of the coordination. The mediator can execute, both, the *monitoring* computation to set flags, and the *coordination* computation to make a control flow decision. In this way, none of the coordinated algorithms itself must know about how its execution is coordinated with other algorithms, and hence it must not be adapted when it is integrated into a larger or another context. The result is a better **composable** activity.

2.7.4 Policy: big endian versus little endian flag arrays

Figure 2.10 shows an alternative way of representing the flag array of Fig. 2.9. *Implicitly*, this latter Figure assumed the first flag to be set in the protocol to be depicted in the first place of the array when starting from the left, and subsequent flags to take the next places in the array. So, the difference between both figures make clear that the *meaning* of the *order* in a *protocol*, can be *represented* in various equivalent ways. This document uses the following terminology for this alternative representation:

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

The document will mostly use the *little endian* representation of Fig. 2.9 for all conceptual, symbolic illustrations. *Software implementations* often make different choices. For example, most computer hardware is big endian, where a smaller number of filled bits on the “right hand” side of a byte array represents lower numbers, so a flag array fills up from the right, or from the lower-valued bytes.

2.7.5 Stop-and-go coordination — Barrier

Figure 2.11 depicts the simplest possible Petri Net, used to coordinate the behaviour of two activities:

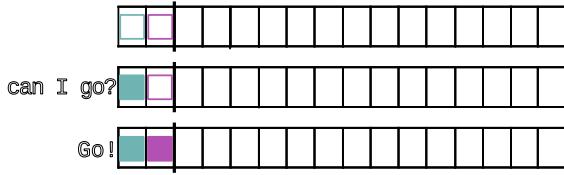


Figure 2.11: *Stop-and-go*, representing the simplest possible coordination.

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

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

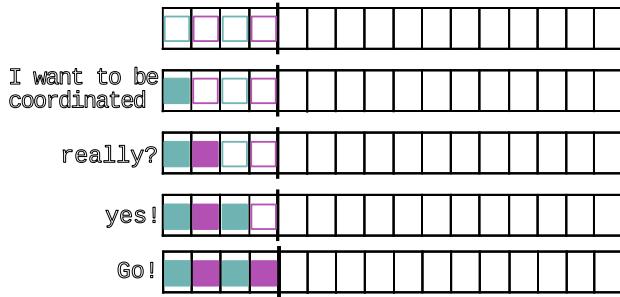


Figure 2.12: *Two-phase-commit* coordination of two algorithms.

2.7.6 Two-phase commit coordination

Figure 2.12 shows another common coordination primitive, **two-phase commit** with only one coordinated algorithm and one coordinator. There can be many reasons for this seemingly superfluous coordination primitive. For example: the coordinator holds the coordinated algorithm until it has observed “something” to have happened elsewhere in the system, but without the algorithm that caused that “something” is itself involved in the coordination. This is an example of the **dependency injection** best practice: the coordinated algorithm does not have to know what it is waiting for, nor how to find out whether the conditions for progress are satisfied; that knowledge can remain in the coordinator, hence improving the composability of the coordinated algorithm.

2.7.7 Data borrowing coordination

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

The flag array in Fig. 2.13 has the same *structure* and *behaviour* as the one in Fig. 2.12. Only the *interpretation* in the context of an application is different.

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

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

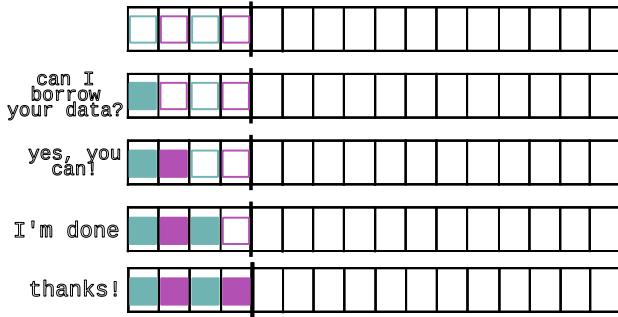


Figure 2.13: *Data borrowing coordination of two algorithms.*

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

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

2.7.9 Race condition: Time of check to time of use

A commonly occurring [race-condition](#) when using flags is known under the name [*Time of check to time of use*](#) (abbreviated to TTCTTOU, TOCTTOU, or TOCTOU). The cause of the problem is that:

- an activity checks the value of a particular flag;
- some time later, it makes a decision on the basis of the value of that flag,
- while in the meantime the flag-setting activity has moved to another state

So, the decision is based on erroneous information, and that can lead to errors in the overall system behaviour. To solve such race conditions, somewhat more complex inter-activity coordination is required, in the form of a [Petri Net](#).

2.8 Petri Net: algorithm/activity coordination via mediator

This Section introduces the Petri Net meta model, as the mechanism to support the **coordination** of the **interaction** between the **control flows** in multiple **algorithms** (and, *hence* between the **activities** in which these algorithms are deployed), while **decoupling** the coordination of both algorithms via a “third-party” **mediator** activity. In other words:

- the algorithms need not know about each others’ existence.
- each algorithm engages in a [flag-based coordination](#) within a mediator architecture.

- the mediator makes the coordination decisions for the algorithms involved, without those coordinated activities being aware of its existence.
- the mediator performs the coordination **decision making** in an internal, synchronous schedule.

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

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

2.8.1 Examples of multi-algorithm coordination

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

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

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

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

2.8.2 Role of algorithm coordination in systems-of-systems

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

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

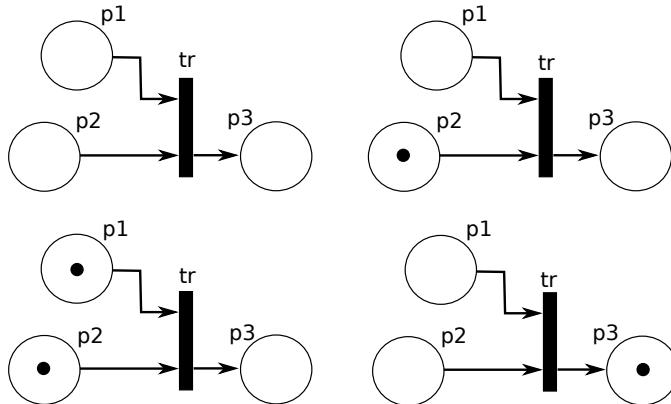


Figure 2.14: An example of the **structural** model of a Petri Net, with places p_1 , p_2 , and p_3 , and a transition tr . The **sequence** of the four structural models gives one possible **behavioural** model of the Petri Net.

2.8.3 Mechanism: place, token, transition

A **Petri Net** is an **abstract data type** with the following **entities** and **structural relations** Fig. 2.14:

- **place**: the **entity** of the Petri Net model. Graphically, it is depicted by an unfilled circle, like p_1 and p_2 .
- **token**: the **relation** that identifies the **state** of a place, that is, a place is filled by a **token**, or it is empty. A **token** is depicted by a black dot, placed in a place.
- **transition**: the **relation** that connects its input **places** to its output **places**. It is depicted by a black rectangle, such as tr .
- **arrow**: the **relation** that connects a **place** to either an **input** or an **output port** of a **transition**. The obvious graphical depiction of an **arrow** is a line with a pointed

arrow. The direction of that arrow represents the input or output role of the place in a **transition** relation.

- **marking:** the **relation** that represents one set of **tokens** in **places**. It is the **state** of the data type, after a certain number of operators have been executed.

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

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

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

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

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

The following **behavioural constraints** must hold:

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

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

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

Figure 2.14 sketches a the behaviour of a Petri Net, as the following series of states (from top-left to bottom-right):

- empty **marking**: none of the **places** at the **input** side of the **transition** tr, p1 and p2, contains a **token**.
- **marking** with one **token** in the **place** p2.

- **marking** in which both **places**, **p1** and **p2**, are filled. The **transition tr** is now **enabled**.
- after the **transition tr** has **fired**, there is now a **token** in the **place p3** that is the **output** of the **transition tr**.

2.8.4 Representation: marking reaction table

The **marking reaction table** is the simplest form of the behavioural model of a Petri Net: it represents the *effect* of the **fire** operator on a **transition** in the form of the removal and addition of **tokens** in the input and output **places** of each **transition**. Here is the *marking reaction table*, for the one-transition Petri Net in Fig 2.14:

input places	transition	output places
p1, p2	tr	p3

And this is the *marking reaction table*, for the one-transition Petri Net in Fig. 2.15:

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

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

This Section explains a **pattern** for the **behavioural semantics** of the **interaction coordination** between multiple activities. Here is the list of **extra** entities, relations and constraints that the pattern uses, in addition to the meta models for the **Petri Net mechanism** and the **flag array mechanism**:

- in the coordination Petri Net data structure, a **place** is tagged as a **source**, if it is not the **output** of a **transition**. Similarly, a **place** is tagged as a **sink**, if it is not the **input** of a **transition**. All other **places** in the Petri Net data structure are tagged as **internal**.
- a **source** is meant to be filled in by the *coordinating activity* (that is, the one that “computes” the Petri Net), *after* the *coordinated activity* has filled the *corresponding flag* in its own coordination flag array.
- the coordinating activity empties every **sink** that has been filled, and then fills in the *corresponding flag* in the coordination flag array of the coordinated activity associated to that **sink**.
- the actions above belong to the **communicate()** part of its **event loop**.
- the coordinating activity does not change the values of the flags that are owned by the coordinated activities.
- there is one exception to the previous rule: the coordinating activity can *reset* the Petri Net and all the connected protocol flags, *when appropriate*. Such a reset typically involves a very simple **higher-order Petri Net**, such as a **two-phase commit**, or even a simple **active flag**. The purpose of this higher-order coordination is that no coordinated activity does anything anymore with its protocol flags until the coordinating activity has had the opportunity to reset everything.

The pattern uses the following modelling parts, Fig. 2.15:

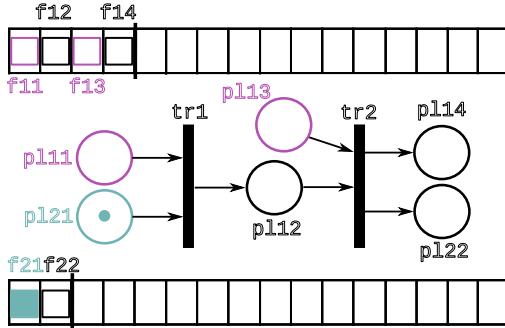


Figure 2.15: An example of the **composition** of one Petri Net (for the coordination **mediator**) with one flag array for each of the two coordinated activity interactions. The similarly coloured and numbered flags in the array, and places in the Petri Net, represent how the places are *produced* and *consumed*.

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

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

Conceptually, the waiting is over as soon as the coordinator puts a **token** in the **output place** that is associated with the waiting algorithm. But each waiting activity can only check whether its waiting can end *after* the coordinating activity has filled in the corresponding flag in the waiting activity's flag array.

The coordinating activity has *exclusive access* (i) to all **places** in the Petri Net, and (ii) to the flags in the flag array buffers that correspond to each of the Petri Net's **sinks**. This guarantees synchronous execution of the algorithm that makes the coordination decisions, and hence **race conditions** are avoided.

Each coordinated activity does *not* place itself a **token** in a Petri Net, it does not know that there is a Petri net involved in the coordination, nor that, and how many, other activities are being coordinated: it only sees the **flag array** via which it engages in its own part of the coordination, with *only* the coordination mediator.

That coordination mediator has the same interaction with each of the coordinated activities. It realises the bulk of the interaction coordination, in a synchronous way. From its point of view, the asynchronous parts of the coordination are the atomic setting of flags in the flag arrays by each of the coordinated algorithms.

The use of a flag-based protocol implies **commitment** of each coordinated activity in the multi-activity coordination to finish the waiting that it has started by filling the first flag in a protocol array. This commitment lasts until it has consumed its last flag in its protocol array.

2.8.6 Policy: flag-to-place-to-flag naming transformations

The pattern of the Petri Net in a dedicated mediator decouples the *mechanism* of coordination in such a way that the coordinated activities do not have to be aware of each other's existence. This decoupling also allows an extra *policy* to be introduced, about naming the flags and places involved in the mechanism:

- for each coordinated activity, the mediator and that activity can use names for the *flags* in their mutual protocol that reflect the **application domain**.

For example, to coordinate perception and control activities in a robot, the developers can use names like `LIDAR-perception-ready` and `MPC-control-can-start` that have a meaning in the *very specific* application context.

- because the mediator is the only activity that “sees” the Petri Net, it can use names that have a natural meaning within one particular Petri Net **pattern**.
For example, `Stop-and-Go` or `Two-Phase-Commit`.
- as a result, in the process of producing the flags in the coordination protocol from places in the Petri Net, the generic place names can be transformed into application-centric flag names. Similarly, the process of producing the tokens in the Petri net from flags in the protocol can transform the application-centric flag names into generic place names.

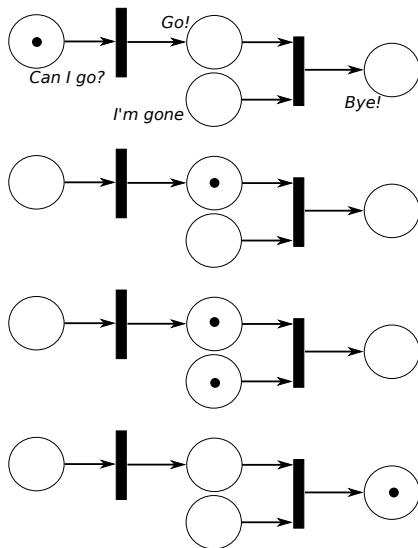


Figure 2.16: *Stop-and-go* with acknowledgement.

2.8.7 Flag arrays as linear Petri Nets

Flag arrays are *linear* Petri Nets, so they *can* be given a Petri Net model. For example, the *stop-and-go* and *two-phase-commit* protocols, Figs 2.11–2.12.

Figure 2.16 depicts the Petri Net version of the *stop-and-go* coordination, adding an extra confirmation step. In this coordination model, the coordinating activity gives the “*Go!*” decision, not by clearing the input place that was filled by the coordinated activity, as it did before, but by filling a new output place. This allows the coordinating activity to acknowledge the coordination explicitly, by clearing that output place.

2.8.8 Stop-and-go coordination — Barrier

Figure 2.17 depicts two possible Petri Nets, to coordinate the behaviour of multiple coordinated activities. The difference between both graphical representations is just graphical *syntactic sugar*.

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

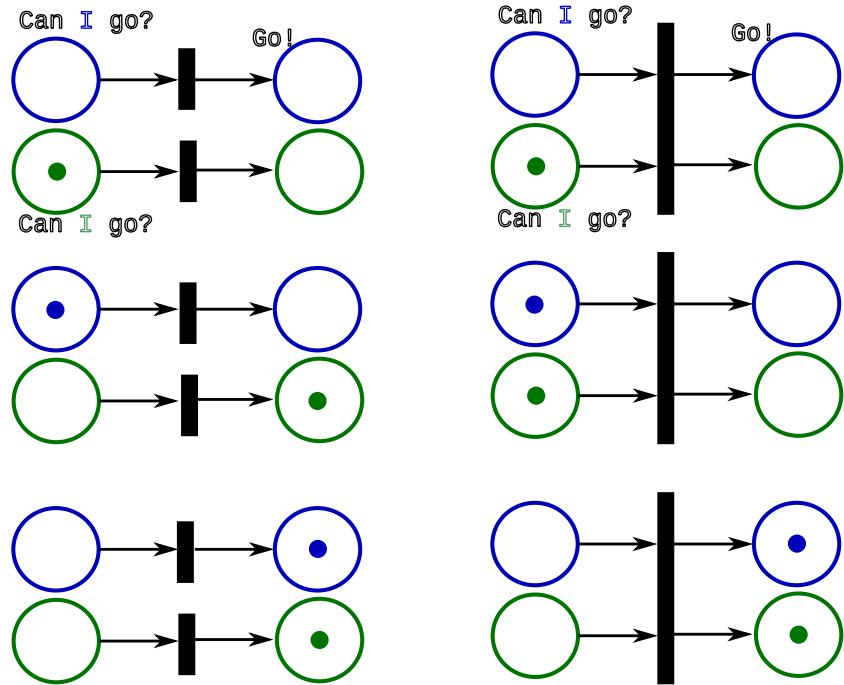


Figure 2.17: Two Petri Net models for a *Stop-and-go* coordination between two coordinated activities. The first one is the union of two times the Petri Net of one single coordinated activity. The second one waits for *both* of the coordinated activities before giving the “*Go!*” signal to *each* of them.

2.8.9 Any-of and All-of coordination

Figure 2.18 depicts two common and complementary coordination semantics: *all-of* and *any-of*. The condition for a **transition to fire** in a Petri Net is that *all* of its input **places** are filled. However, some use cases want a coordination to proceed as soon as *one* of the inputs is filled. (A further elaboration could be to make the distinction between *exactly one* of them being filled, or *any number* of them.)

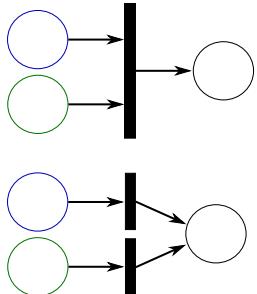


Figure 2.18: *All-of* (top) and *Any-of* (bottom) semantics of logical composition of tokens in places.

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

Figure 2.19 depicts the coordination use case to get a set of activities ready for a shared task:

- each activity that is ready to start the shared task, indicates its readiness by switching its *orange light* on.

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

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

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

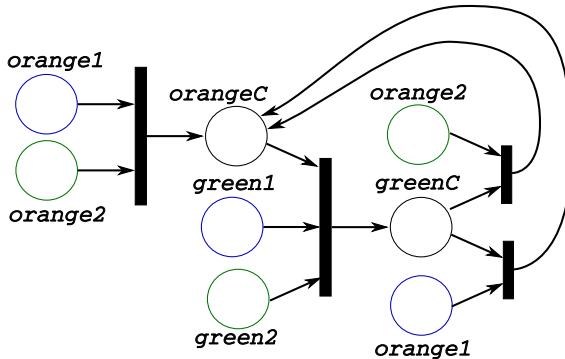


Figure 2.19: *All-to-go, one-to-stop* coordination.

2.8.11 Fork-and-join coordination

Figure 2.20 depicts the Petri Nets that coordinates a *fork and join* of multiple coordinated activities:

- *fork*: the mediator fills the place p_{1-1} , to indicate that it is ready to start the coordination protocol. Some time later, the first transition tr_1 fires, which gives rise to “forking” off two (or more) algorithms, 1 and 2. They inform the mediator that they have started, by filling the places p_{111} and p_{121} . This fires transition tr_2 , which indicates the mediator to go to the *join* phase.
- *join*: after both algorithms have become active, the mediator waits for them to finish in place p_{1-2} . This waiting finishes when both algorithms have filled their places p_{112} and p_{122} . So, transition tr_3 can fire, indicating that both algorithms have “joined” their activities. This end of the coordination can be signalled explicitly via the p_{1-3} place.

2.8.12 Par-Seq-Join coordination

Figure 2.21 shows a coordination situation that is structurally very similar to the **Fork-and-join** coordination: two activities are started in a coordinated way, running together with a third activity, that must decide when *to join* the other activities.

(TODO: DAG with one root, and each branch creates new “tasks” and one “joiner”; the latter mediates the nominal or preemption end of the tasks.)

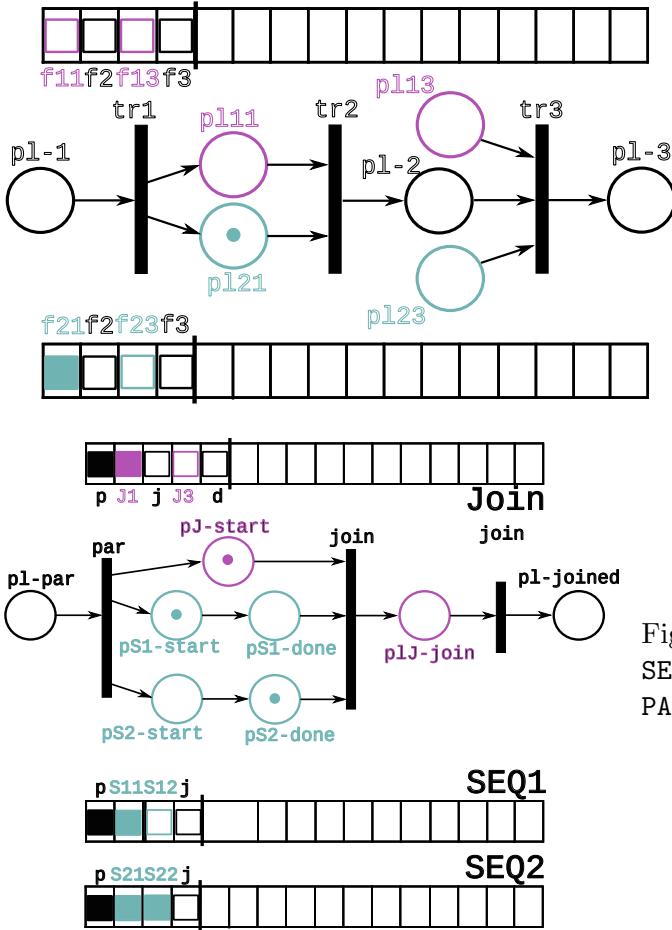


Figure 2.20: The Petri Net structure for a *Fork-and-join* coordination for multiple coordinated activities.

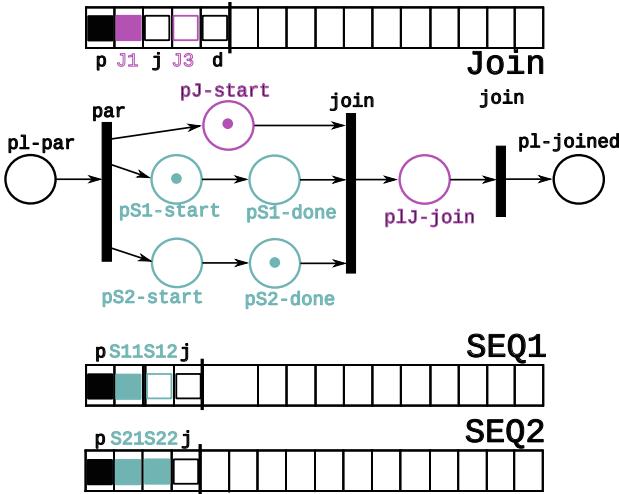


Figure 2.21: Coordination between two SEQuential activities, running both in PARallel with a Join activity.

2.8.13 Transaction coordination: multiple peers with synchronous access to data

In many applications, multiple peers inside the system want to access data (for reading and writing) *at the same time*. There are **two approaches** to this problem, both requiring the presence of (i) a **data protecting peer** dedicated to coordinate the access, together with (ii) a **protocol** that each **data accessing peer** has to follow:

- **transactions**: there is **one copy** of the data, and the data protecting peer and the access protocol make sure that one and only one peer can access the data at each instant in time.
- **streams**: each peer **registers** with the data protecting peer following a protocol, after which the data access peers provides a **copy** of the data exclusively to each peer.

The transaction is conceptually the **simplest**, and there are only two **mechanisms** to realise transactions:

- **database**: a data accessing peer provides the data protecting peer with the information *about what operation* it wants to do on the data, and that data protecting peer then executes that operation. This mechanism works because the data protection peer, and only that one, is really accessing the data.
- **access control**: the data accessing peer provides the data protecting peer with the information *that* it wants access to the data, and that data protecting peer then informs

the data accessing peer *when* it can do so.

In both cases, the access to the data takes place **synchronously** in the execution context of **one and only one peer**. Multiple policies exist about *how* to realise these two transaction mechanisms. The choices of (i) which transaction mechanism to use, and (ii) which policy to use for the chosen mechanism, are essential design decisions in the **information** and **software** architectures of the application.

2.8.14 Policy: hierarchical Petri Net

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

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

Common reasons for introducing a hierarchy are:

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

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

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

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

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

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

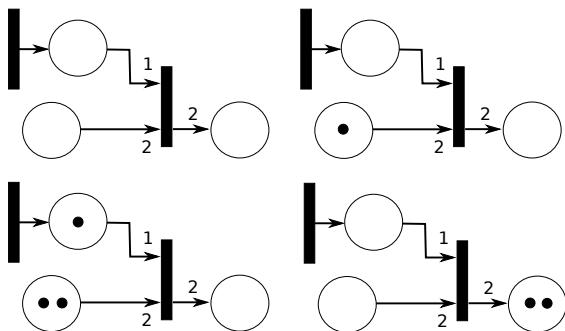


Figure 2.22: A Petri Net with *multi-token* places. The numbers on the arrows indicate the cardinality of **tokens** needed to enable the **transition**.

2.8.15 Policy: multi-token places with cardinality

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

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

That number can be different for each place, it can be zero, but it should never change. In the example depicted in Fig. 2.22, the transition can fire (marking at the top) if there are two tokens in the place connected to the transition with a *cardinality condition* “2”, and one token in the other place with a *cardinality condition* “1”. The outcome of the fired transition (marking at the bottom) is to put “2” tokens in the output place.

It is obvious how to transform a *multi-token* Petri Net into **canonical form**: just duplicate each place with N tokens N times. So, the reason to introduce multi-token places is **syntactic sugar**, which makes designers gain some model creation efforts. What they loose is **mathematical expressivity**: a Petri Net couples the coordination of two or more algorithms, so semantic clarity improves if one can identify, already in the *structure* of the Petri Net, which algorithms are connected to which places.

2.8.16 Policy: coloured Petri Net for typed coordination decision making

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

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

The variation in adding “colours” to the **Petri Net mechanism** is infinite; hence, a large number of mutually non-interoperable Petri Net dialects have emerged in the literature. This document considers the motivations above as **bad practice**, because it violates the **decoupling** design drivers behind this Section’s pattern. Section 2.8.19 advocates better approaches.
(TODO: substantiate the claim above, with concrete examples!)

2.8.17 Policy: Net-wide constraints

This Section lists constraints that can be added to a particular Petri Net:

- when several transitions are enabled, a *particular* firing order choice must be made.
- when one place is the input to more than one transition (Fig. 2.23), a *particular* firing/token consumption choice must be made.
- constraint on the maximum, minimum, or exact, number of tokens that can be present in a Petri Net Invariant at all times.

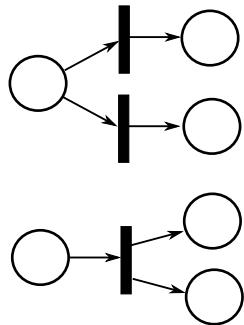


Figure 2.23: Two similar Petri Nets. The bottom one has an unambiguous firing behaviour, but the top one requires a specific transition firing decision policy, to decide between one of these three possibilities: (i) only one of both transitions is fired, (ii) both transitions are fired, or (iii) one is fired now but the token is left in the input place.

- the interaction between the Petri Net-based coordinating activity and the coordinated activities must be mediated: when are flags from coordinated activity transformed into tokens and vice versa.
- the “initial” and “final” markings are *choices* made in the application. Hence, they are **attributes** and not properties of a Petri Net model.

2.8.18 Promises, Await-Async patterns modelled as Petri Nets

The *promise*, or *future*, is a common coordination mechanism, between a *requester* of a result and the *fulfiller* of that result.

(TODO: `then()`, `catch()`, `finally()`, `resolve()`, `reject()`, `any()`, `all()`, `race()`.)

2.8.19 Best practices in Petri Nets

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

- the **naming policy** allows developers to prevent “semantic leakage” of the application into the Petri Net. Hence, they can cover many application-specific coordination use cases with a small, generic set of Petri net patterns. This also offers the possibility to spend more development time in documenting the pattern, and in optimizing its implementation.
- the *commitment status* of a coordinated activity in a Petri Net is always fully observable, for that activity and for the coordinator. When a coordinating activity decides not to commit to a coordination any longer, it is the coordinator’s decision whether and how to inform the other involved activities.
- a hierarchical Petri Net in the coordinating activity (more in particular the *inhibition/activation* semantics) allows “to glue together” Petri Nets with different sets of coordinated activities. This leads to more but smaller Petri Nets.
- each coordinated activity uses **two-phase commit** for each of the higher-order Petri Nets it is involved in, such as **time outs** and **inhibition/activity**. This (yet another higher level of coordination modelling) gives both coordinated and coordinating activity the time to react and reconsider, in use cases where the effect of the coordination decision comes with a “significant” risk. For example, the coordination was about agreeing on a “task contract” or not, and the conditions under which the “offer” was made can have changed in the meantime. Another example is the risk of **chatter**: if the Petri Net structure has loops, it is possible that the same **place** is filled or cleared via two different paths; and depending on the *timing* of the decision making logic in the coordinator, the

filling state of a **place** can go through quick changes. A time-out of the “appropriate” length can provide the **hysteresis** to prevent such chatter.

- a **heartbeat** combines time-out and activity commitment: if the coordinated activity does not refresh its commitment in a coordination fast enough, the coordinating activity *can* take the decisions (i) to de-activate that activity in the coordination it is involved in, and (ii) to follow the above-mentioned policy of informing the other involved activities.
- the Petri Net model identifies the coordinated activities explicitly. Keeping track of which activities are really needed in coordinations, can help to keep Petri Nets small. One observation that indicates that a Petri Net is too large, is that some activities are involved in the “beginning” of a coordination, but not any more later on.
- if a coordinated activity keeps “state” of the different phases of the protocol that it has already gone through, it can support also the coordination of possible **undo** operations (or “**undelete**”, or “**rollback**”): the coordination is aborted before it is finalized, and the state of the coordinated activity is then reset to what it was before it engaged in the coordination process.

2.9 Stream: peer-to-peer interaction via asynchronous exchange of data

This Section describes the **stream** (**channel**, **buffer**), as the **composable** meta model for **data exchange** between **activities**. The essential concepts of a stream are:

- **peer-to-peer**: the operating system is not involved in the data exchange. Which is for example the case with data exchange mechanisms built with **locks**. There *can* be multiple (data) “transformation” peers in between, each operating on the data chunks, that they receive “upstream” from the producer and push “downstream” towards the sink. All of the peers are constrained by the same direction of data exchange.
- **uni-directional**: the major use case is the *unidirectional* data exchange *from* one **producer** activity (that is the “source” of the data chunks in the stream), *to* one **consumer** activity (that is the “sink” of all data chunks), making use of the services provided by a *third* “activity”, the **buffer** (Fig. 2.24).
- **frame** based: the exchange between the peers happens with the granularity of a frame, which is any structure on the data that has meaning in the context of the *application*,¹⁶ typically a combination of (i) *data structures* (e.g., all measurements in a **laser scan**) and (ii) *flow control* (e.g., events like state changes of flags).
- **typed**: the type of the frames exchanged in the stream is fixed for every frame, but can be diverse: binary **data chunks**; **messages** directly from one peer to an identified other peer; or the **publish-subscribe** of *topics* with a particular meaning to and from a **broker** peer.
- **connection** (or **session**) oriented: before data is being exchanged, both peers have *to find* each other, then to create a *connection* between them, after which they have *to maintain* that connection to last for multiple data exchanges, and then finally decide *to remove* the connection.

¹⁶This Section describes the **information** model of streams, and the smallest semantic granularity is then that of a *data chunk* and its *meta data*, taken together in a *frame*. The **packet** is the *software*-level container of a frame, with the extra data needed to get that container over a “network”, from the first peer’s “host” to its destination peer’s “host”.

The life time of a stream must at least be as long as the connection via which it is offered. One single connection can host several streams, concurrently or in sequence. And the same two peers can set-up multiple connections, concurrently or in sequence.

- **asynchronous**: each of the peers can engage in the data exchange at its own pace and timing, and must never wait for the other peer to start or finish its operations on the stream.
- **with unique identities**: each *connection* gets a unique identifier, as does each *stream* in a connection, and each *frame* in a stream.
- **ordered**: the unique identifier of each frame in a stream is *monotonically ordered*, so “missing frames” are easy to detect.
- **contiguous**: ownership of frames in a stream gets *transferred* in **contiguous** chunks, that is, a set of one or more consecutively ordered data chunks.
- **mutable** data: producer, consumer and transformer can all **mutate** the data chunks in the stream part they own, and do so in any order.
- **anonymous**: while connections, streams and frames are identified to the peers involved in the stream, these peers themselves need not know each others’ identities: they connect to a particular *service*, and while that service is provided by a identified peer, it also decouples the peers involved in the data exchange. Of course, the identify of the peers *can* be part of the *application protocol*.

Channel is terminology that fits in the context of “telecommunication”, so often “producer” becomes “transmitter”, “consumer” becomes “receiver”, and “transformer” becomes “filter”. In an *computer science* context, the terms “writer” and “reader” are common.

The important role of “streams” was already clear early in the history of design of operating systems, e.g., [25, p.239], but was “forgotten” in most application-centred system design projects. Apparently, the latter contexts always prove to be too tempting for developers to choose for *too early optimizations*.

2.9.1 Streams in the real world

The stream concept is not restricted to only a digital context, but it is a *digitizations of proven approaches from reality and real life*, where people and organisations use streams to coordinate their interactions. So, streams have become dominant in several digital domains—such as the *World Wide Web*, and in *streaming media*—because they compose:

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

- hence, allowing **in-place** operations on the owned stream parts.¹⁷

2.9.2 Producer-Consumer protocols

This Section provides an overview of the major **stream protocol** families, explaining which coordination problems they solve in the real world. The families are not mutually exclusive, so a particular interaction architecture typically selects a mixture of protocols, for streams between different components, as fits the “local” purposes best. The terminology is *not* adopted from the *real* world, but is how these information exchange patterns are known in the *cyber* world. A system can contain instantiations of *multiple* of the following interaction patterns, even at the same time and between the same two algorithms:

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

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

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

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

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

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

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

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

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

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

- *Submission-Completion*, with *mediator*: the producer *submits* several different pieces of “stuff” to the consumer, at irregular intervals; the consumer has a *mediator* role, so it *completes* some work on each of the submitted “stuff”, and provides an answer back

¹⁷Hence, the name “**stream**”, and not **queue** which comes with the semantics of (i) adding or removing data chunks only one at a time, and (ii) without any processing in between.

to the producer, also at irregular intervals and not necessarily in the same order as the producer submitted everything.

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

- **Blackboard**: in a classroom, both the teacher and the pupils can add and remove writings on the class' blackboard, and all changes are visible to all immediately, and in parallel.
- **Bus**, including **fieldbus**: all participants ("information") can jump on and off the bus at any time the bus passes by. Once one gets off the bus, participants ("information") take no part anymore in the transportation ("communication"), and hence are "lost".
- **Interactive Connectivity Establishment**: interaction protocol to let two peer activities find each other and establish a connection. It is possible that a connection can only be created by involving a third party **relay**.
- **Offer/Answer**: one peer in a connection provides the other peer with one or more offers, each suggesting a concrete choice of how to proceed forward in their interaction. The other peer is expected to provide its answer to that offer.
- **broadcasting**: radio and TV distribute their information to all the public in parallel. This is very effective one-way interaction, but the two-way aspects of reaction and interaction are difficult.
- **actor model**:
(TODO: relies on message passing between actor activities, but have a mediator to decide how to react to incoming messages, and how to react to actor activities crashing.)
- **dialogue queries**: a special case of *Submission-Completion*, but with direct responses from the consumer.
- **push**: the producer takes the initiative to deliver the exchanged data.
- **pull**: the consumer takes the initiative, and triggers the producer to produce.
- **timeout** at either producer or consumer side, when a peer decides not to wait longer for a data exchange to take place.

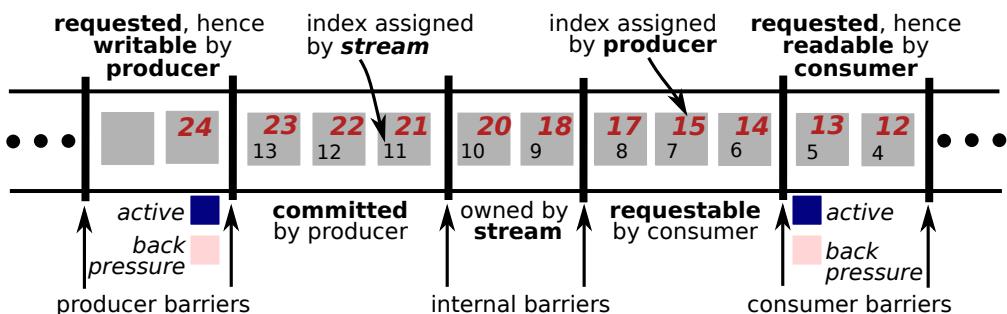


Figure 2.24: Meta model of the canonical *single producer-single consumer stream*, with a third peer activity that coordinates the *buffer* of data chunks between producer and consumer.

2.9.3 Canonical meta model of a Stream: producer-buffer-consumer

The most basic form of a stream involves three activities: **one single producer** activity and **one single consumer** activity, **loosely coupled** to each other by **one single buffer**

activity¹⁸ (Fig. 2.24). The coupling between the three peer activities involves three **structure** and **behaviour** mechanisms, each conforming to the **Block-Port-Connector** (BPC) meta model:

1. **BPC inside the stream's connector:**

- **connector:** this is a **ordered buffer** of frames, or **data chunks**, each with a unique and monotonically increasing numerical **identifier**, owned by the **connector**. These **abstract data types** are **owned** by the **connector**, and subdivided in **committed** and **requestable** regions, with possibly a non-empty region in between.
- **connector ports:** the information about where the region barriers lie in the unique identifier index. One of these ports is connected to the **producer**, the other to the **consumer**.
- the **producer** and **consumer** get temporarily access to a subset of the **connector**'s buffer, according to the stream's **access protocol**.
- **connector block:** the functionalities ("behaviour") to realise the **access protocol**, that is, to decide about (i) how to change the barriers between the above-mentioned regions (that is, to process the **commits** and **requests** of **producer** and **consumer**), and (ii) how to give the unique identifiers to each data chunk.

2. **BPC between producer and stream connector:**

- **producer block:** the functionalities (i) to **request** a contiguous set of data chunks from the stream, (ii) to **write** application-centric contents in the available data chunks, and (iii) to hand over a contiguous set of written data chunks to the stream via a **commit**.
- **producer port:** contains the *read-only* information for the producer about where the **barrier** lies of the data chunks it is allowed to write.
- **stream block:** the functionalities (i) to provide data chunks to the producer when it **requests** them, (ii) to **take** over the data chunks that the producer **commits**, and (iii) to give a buffer-centric **stream-index** identifier to every data chunk. This identifier is is monotonically increasing, and unique for the **stream**. It is not visible to producer or consumer.
- **stream port:** the natural complement of the producer port, with *read-only* and *written* interchanged.
- **connector:** the buffer of the contiguous data chunks involved in the producer-connector data exchange.

3. **BPC between consumer and stream connector:** this is the dual to the interaction between **producer** and **connector**.

Although the duality is not complete:

- at the **consumer** side, it is possible that the **consumer** accesses less **data chunks** than it already owns, because the **producer** has **produced** some extra data chunks *after* the **consumer** has **requested** new **data chunks**.
- at the **producer** side, there is always a complete equality between what the **producer** can access and what it owns.

The following **constraints** must hold on all behavioural operations:

- no transfer of data chunk ownership can ever be revoked.

In the worst case, the **consumer** accesses less **data chunks** than it already owns, because the **producer** has **produced** some extra data chunks *after* the **consumer** has **consumed**

¹⁸The latter need not even be a full-fledged activity, because in many implementations all buffer management behaviour is provides as a *library*.

new data chunks.

- the **barriers** owned by **consumer** and **producer** can never overtake each other.
- **producer** and **consumer** can process (“**mutate**”) all the **data chunks** in that part of the stream that they own, in any order, but without changing that order nor their identifiers.
- all transfers of ownership are done without leaving ownership holes in the stream.
- only the **buffer** is allowed to set the value of the **stream-index** of a data chunk.
- the **buffer** must set that **stream-index** *before* it transfers ownership of the corresponding data chunk to the **producer**.
- the **stream-index** increases monotonically, without holes in the numbering.

Some of the above-mentioned constraints pertain to *coordination* between **producer**, **consumer** and **buffer**, so the mechanisms of **flags**, **protocol arrays**, and **Petri Net** must be introduced.

2.9.4 Policy: flags for producer–consumer self-coordination

The **stream meta model** can *not guarantee* that no data chunks are lost between **producer** and **consumer**, because, for various reasons, the **consumer** can be too slow to keep up with the data producing rate of the **producer**. The simplest version of a policy to add to the Stream meta model is to give **producer** and **consumer** two **flags**, *each*:

- **backpressure** flag: if set by one of the peers, that peer indicates that it wants the other peer to act on the stream:
 - **low water** flag: set by the **consumer**, to stimulate the **producer** to produce more data chunks if the stream is close to “empty”.
 - **high water** flag: set by the **producer**, to stimulate the **consumer** to consume more data chunks if the stream is close to “full”.

These flags support a *reactive* decision making, when the stream indeed becomes “full” or “empty”. It is the responsibility of the application developers to provide a policy about **how to react**: dropping (“discarding”) data chunks, overwriting existing data chunks, or changing the application’s “performance”.

- **active** flag: if cleared by one of the peers, the other one knows that it should not expect any activity on the stream from that peer.

These flags support a *proactive* decision making: the application can *avoid* the problem. These flag allow **producer** and **consumer** to decide about *application-level coordination*, at the cost of introducing *information coupling* between both: both peers can let their decisions about if and how to work with the stream depend on the knowledge about what the other peer’s current status is on the stream. This coordination takes place without any involvement of the **buffer**, although the latter has an indirect role to play: **producer** and **consumer** don’t have to know about each other’s existence, because the **buffer** does the *mapping* of the flag values between both activities.

2.9.5 Policy: index for producer–consumer self-coordination

A second, complementary, policy can be introduced, the **application-index**, to give **producer** and **consumer** more options to deal with non-ideal behaviour of a stream, or of each other: before the commit, the **producer** adds an *extra identifier*, the **application-index**, to every data chunk. That application-centric identifier need only make sense for **producer** and **consumer**, and the stream connector does not even have to be aware of it. The purpose of

this policy is to give the **consumer** information about **how many data chunks were lost**, irrespective of the cause of that loss.

The **stream-index** and **application-index** have complementary roles:

- the **stream-index** adds *structure* to the **resource** of the stream.
- the **application-index** adds *structure* to the **usage** of the stream.

Both indices have similar *constraints*:

- only the **producer** is allowed to set the value of the **application-index** of a data chunk.
- the **producer** must set the **application-index** *before* it transfers ownership of the corresponding data chunk to the **buffer**.
- the index is *monotonically increasing*, and *unique* between the *activities* that use the stream.

2.9.6 Policy: Petri Net for request and commit coordination

(TODO:)

2.9.7 Policy: request-less data chunks

(TODO: all available data chunks are always allocated to producer and consumer, without their explicit **requests**. Hence, there is no need for a separate **buffer activity**, or for requesting free space. Indeed, the **producer** and **consumer** free up all the data chunks they do not need anymore, and these are added directly to the available space in the stream. In other words, their **ports** always have the latest available information about the data chunks they can start using.)

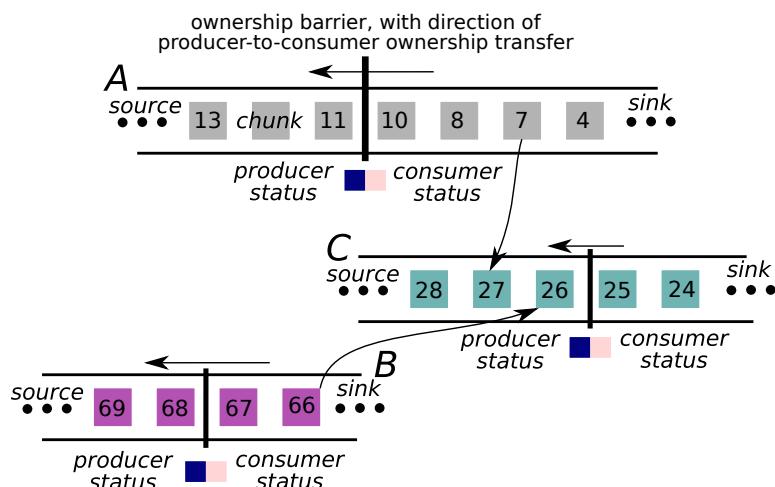


Figure 2.25: Example of *producer-consumer streams*, and their compositions. The **producer** activity of the C stream acts as a **consumer** on the two other streams.

2.9.8 Mechanism: composition of streams — Stream-to-Stream

The composition of streams is conceptually very simple, Fig. 2.25: one form of processing that a **consumer** can do on a stream is to copy the data from one stream and use them as

producer on another stream. Asynchronicity and ownership remain fully transparent and unambiguous under this composition. (Which would be lost by the alternative composition mechanism of *sharing* data chunks between two streams.)

2.9.9 Mechanism: composition of streams — Adding transformer peers

The *producer-buffer-consumer* pattern can be applied repeatedly on one single stream, to realise **pipeline** of producer-consumer peers, Fig. 2.26. The terminology “producer” and “consumer” only reflects the *role* that a peer activity plays on a particular part of the stream: each peer acts as a producer for its “downstream” part of the stream, and as a consumer for its “upstream” part. The strict order in the identifiers, and the transfer of ownership behaviour, must both be maintained under this serial **peer-to-peer** composition. From the *application’s* point of view, the following terminology is sometimes introduced:

- the first **producer** on the pipeline is called the “**source**” of the stream.
- the last **consumer** on the pipeline is called the “**sink**” of the stream.
- the activities that act on a stream downstream of the source and upstream of the source are sometimes called *transformer* activities.

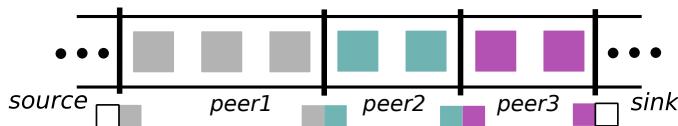


Figure 2.26: One stream shared by more than two data chunk processing peers in a pipeline.

2.9.10 Best practice: Life Cycle State Machine of a Stream — Session

The Sections above show that a stream has two major stakeholders:

- the *buffer owner* that manages the buffer as a *resource*, and coordinates the access of producers and consumers to that resource.
This coordination requires the **Life Cycle State Machine** pattern:
 - **creation**:
 - **configure_resources**:
 - ...
- the *producers* and *consumers* that use the stream for their asynchronous data exchange.
The producers and consumers don’t see the buffer’s full LCSM, but only their own customized **protocol**:
 - **discovery** (“signalling”):
 - **register** (“session initiation” and “session description”):
 - **acquire** (“request”):
 - **produce & consume** (“commit”):
 - **release**:

(TODO: session manager: finds components, connects them to the session, configure their properties, attach streams,...)

2.9.11 Policy: lossless, missing data, messages, bags

The **policies** with which data chunks are produced and **consumed** can be diverse:

- **lossless stop-and-wait** so both **producer** and **consumer** can **react** to each other's data exchange speed.
- **lossy producing** when the **producer** produces data faster than the **consumer** can **consume**, and, hence, the **producer** has to throw away some data.
- **missing data**: when the **producer** has not been able to generate a data chunk under nominal conditions, the **consumer** can get aware of this situation by checking whether there are holes in the **application-index**.
- **send-and-forget** exchange in which the **producer** does not keep track of which data it has already exchanged.
- **messages** with explicit **identifiers**. This involves some form of *dialogue* between producer and consumer that allows both to *refer explicitly* to messages exchanged in the past.
- **bag**: one data chunk in a stream between two *activities* can be in itself the composition of data chunks of the multiple *algorithms* that run inside the activities. This policy is used when data exchange happens at regular times anyway, and data that happens to be available in any of the algorithms at the producer side is “bagged together” (“marshalling”, or “serialised”) into one big message, and communicated. The consumer side *de-serializes* the bag and distributes its parts to the appropriate internal algorithms.

2.9.12 Policy: multiple producer–multiple consumer stream

In many applications, each activity must interact with multiple other activities, and often share the same information between these multiple activities. The easiest way to make an architecture for this situation is to introduce information streams that have more than one producer and/or more than one consumer (Fig. 2.27).

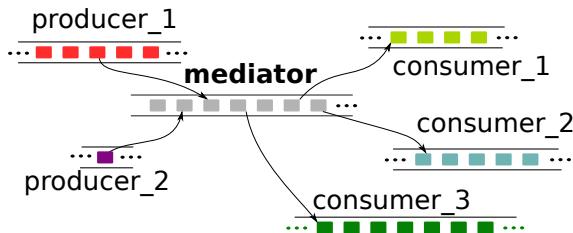


Figure 2.27: Single stream with multiple producers and consumers. Such an architecture requires explicit coordination between all producers, and another explicit coordination between all consumers, because *ownership* of each data chunk in each shared stream is not clear.

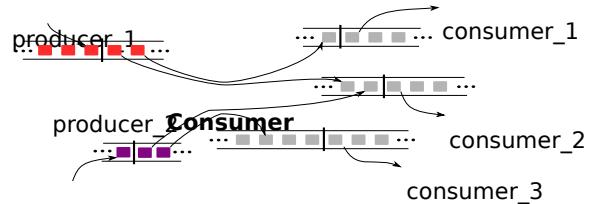


Figure 2.28: An equivalent architecture to Fig. 2.27 (*only* for the left-hand side of that Figure, covering the sharing of the “grey” stream): **one** intermediate **Consumer** activity is introduced in the architecture, and that activity is the **single** reader on the multiple producer streams, an also the **single** producer on each of the consumer streams.

This approach introduces several **data chunk ownership complications**:

- how to decide which producer is allowed to update which data chunk in the stream.
- when is the ownership of each chunk transferred.
- how to decide which consumers get access to the same data chunk.

Figure 2.28 sketches a more **explainable and composable** design:

- it introduces an **extra consumer** activity, whose sole purpose is to be the **only consumer** on each of the producer streams, and the **only producer** on each of the consumer streams.

This solves, both, the ownership and access coordination problems. But also the [head-of-line blocking](#) problem is avoided, because the suggested policy uses the technique of [virtual output queueing](#).

- it introduces an extra copy operation for each data chunk.

Indeed, the “green” coloured consumer streams on the right-hand side of Fig. 2.27 are not directly produced from the grey “shared” stream in the middle, but first the “local” grey streams in Fig. 2.28 are filled for each consumer before that latter one can process those data chunks to produce its own “green” stream.

2.9.13 Pattern: composition of data and metadata streams

In the above-mentioned context of using streams to increase the [loose coupling](#) in system architectures, it makes sense to let streams carry the information that allows the consumer to interpret the producer’s data chunks without any interaction with third parties. In other words, a stream must have a mechanism to [piggyback metadata](#) on top of the data, to increase the [data lineage](#) (also called “[data provenance](#)”).

Figure 2.29 shows the simplest approach of adding a dedicated metadata stream. The metadata is typically a lot smaller than the data, and one entry in the metadata stream can represent the interpretation of many (often, even all) chunks in the data stream. Of course, the chunks in both streams are highly dependent and are hence best generated together by the same producer.

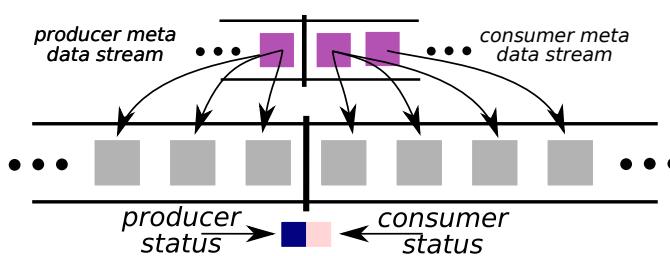


Figure 2.29: Producer-consumer stream composed with an extra metadata stream. Each chunk in the latter encodes metadata for a contiguous series of chunks in the former. Both streams share the same status flags, and ownership barriers.

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

2.9.14 Pattern: Submission-Completion streams — Task queues

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

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

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

The **protocol** of Submission-Completion streams is an extension of **Request-Reply** towards *series of related* (CRUD) requests:¹⁹ producers of requests add them to the submission stream, and come back later to collect the result from the completion stream, Fig. 2.30.

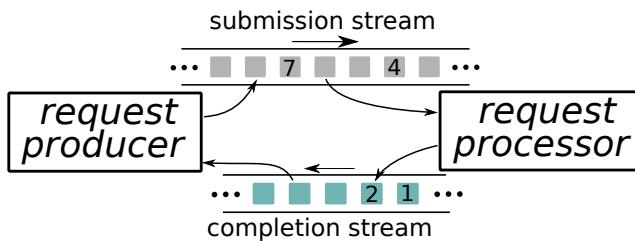


Figure 2.30: Submission-Completion streams. The processed stream goes to the original producer; the order of the items in the *completed* stream can be different from the order in the *submitted* stream.

The pattern involves a **duplex** stream, that is, an input (“submission”) stream, and an output (“completion”) stream:

- the producer submits its request to the *request submission* stream, and the request processor takes it off and processes it when it sees fit.
- the request processor submits its result to *request completion* stream, and the producer takes it off when it sees fit.
- the request processor need not return the processed requests in the same order as the producer has submitted them. That means that the index in the stream can not double as unique identifier of each request, so the latter must contain extra fields for that sole purpose of identification of a request.

The Submission-Completion streams pattern has been used since decades already, such as in **Channel I/O** for mainframe computers; **REST** interfaces on the Web; **disk access** libraries; or **multi-host communications**. More recent instances appear in: **asynchronous I/O** via “**I/O rings**” that an **operating system kernel** provides to a **application**. This document’s **main use** of the pattern is to realise **task queues**. Not in the least because the pattern supports the **pre-emption**, **pausing** or **cancelling** of previously submitted but not yet executed tasks.

2.9.15 Pattern: journaled CRUD interfaces

The **mechanism** of submission-completion supports the explicit bookkeeping that data chunks have been submitted and completed. Of course, an application can add **metadata** to the data chunks that refer to the *semantic contents* inside the data chunks. Hence, when a peer puts a data chunk into the stream, it can add a symbolic reference to a specific part of the other peers earlier data chunk to which it reacts. The result is a dialogue, about the whole or a part of an original data chunk, Fig. 2.31.

A common type of interaction between activities are the so-called **CRUD** operations (Create, Read, Update, Delete) on the **abstract data types** and **data structures** that represent models, events and data. The *mechanism* to support such CRUD interactions between asynchronous activities is the **Submission-Completion pattern**, (Fig. 2.32) and more in particular an approach such as **write-ahead logging**, in which the stream buffer acts as the “**transaction log**”, or the “journal”.

¹⁹The simplest version of this pattern is the **Command stream**.

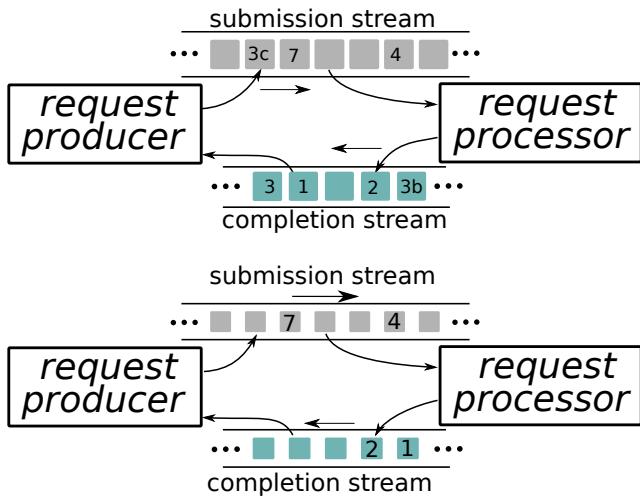


Figure 2.31: Submission-Completion streams with partial dialogues: both parties can react to parts of each other's data chunks, with explicit reference to the reacted to parts.

Figure 2.32: Submission-Completion streams as a mechanism to realise CRUD operations between producer and consumer.

This mechanism originates in data bases technology, and (hence) fits well to all “world model” aspects in robotic systems: control, perception and monitoring activities need to exchange information with the world model, and the latter must serve the needs of multiple interacting activities at the same time, while still guaranteeing a good trade-off between data consistency and latency in reacting to the requests. The trade-off can be steered by:

- **granularity** in the streams. Both in number of streams that one single activity can establish with the data base, and in magnitude of the transactions. For example, a robot can make one transaction to the world model per control cycle, or it only updates the world model after it has travelled through a complete “area” on the map.
- **task-level coordination protocols.** CRUD operations should be robust against an *identified* set of communication and computation disturbances to the system behaviour. Examples of disturbances are: delays in communication, non-availability of resources, or saturation of streams.

The system architects decide about the envisaged robustness: in many use cases, it makes little sense to design the trade-off to higher performance than needed in the application. For example, many robotics systems can tolerate the loss of large parts of a world model, because they have the possibility to (re-)execute tasks that can make the system recover from such a failure.

Despite the obvious observation that tasks, activities and CRUD operations are essential design primitives, they often remain overlooked as first-class design drivers, and very few *software frameworks* support task-level models. The reasons are manifold:

- it is difficult to express, explicitly, the knowledge relations that link platform resources on the one hand, and task capabilities, and performance and robustness requirements, on the other hand.
- system design spans several levels of abstraction and requires many scientific and engineering disciplines to be integrated. Hence, task-centric trade-offs put high demand on the knowledgeability of the system designers.
- many a software project starts *bottom-up*, around some algorithms and middleware that the developers can get “to work”; this often results in the proverbial “**law of the instrument**”: the instrument designers bring in (implicitly and unwillingly) artificial constraints on the applicability of their software, and hence it is they who put limits on system performance, not the system designers.

- and, often as a result of all of the above, the design of coordination protocols, that guarantee consistency under disturbances, is a tough design challenge.

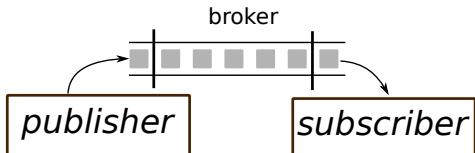


Figure 2.33: *Publish-Subscribe* communication realised by a streams composition. Both publisher and subscriber have ownership of only one single entry in the stream.

2.9.16 Policy: Send-and-Forget, Publish-Subscribe, Request-Reply

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

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

The [Publish-Subscribe](#) and Request-Reply communication patterns can be realised as particular cases of the basic *Send-and-Forget* semantics of [streams](#):

- *Publish-Subscribe* (Fig. 2.33): the publisher has a stream to the broker, without [flow control](#) or [backpressure](#) support, and with the extra policy that the publisher can only fill the stream one entry at a time. A similar configuration holds for the stream between the broker and subscriber.
- *Request-Reply*: this is the Submission-Completion architecture, with the same constraints as in the Publish-Subscribe case.

The major differences between *publisher-subscribe* and *producer-consumer* policies are summarized below:

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

[Request-Reply](#), is a “hybrid” between both (which merits to be identified as a third major pattern): it is a producer-consumer dialogue with a memory of one *request* message and one *reply* message.

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

2.10 Roles in Coordination and Configuration

The entities in the title of this Section represent two families of roles in Coordination:

- *representation* of the *coordination state*: flag, protocol, Petri Net, and FSM.
These abstract data types keep track of how far into a coordination “behaviours” have already progressed.
- *communication* of the *coordination state*: event and stream.
These *active* abstract data types make the above-mentioned *passive* data types for coordination state representation available for *execution* to all activities involved in the coordination.

2.10.1 Petri Nets, Finite State Machines, flags and protocol arrays

The **Petri net** mechanism for the **coordination of the interactions between multiple activities** is complementary to the **Finite State Machine** mechanism, for the coordination of **behaviour of one single activity**. Both share some parts of their *structural abstract data type*, namely:

- “circles”, representing the *state* of the coordination mechanism.
- “arrows” connecting “circles”, representing the *transitions* between states.

This *structural* resemblance sometimes leads to confusion about which model to use for which purpose, but that choice depends on the fundamental *behavioural* differences between both coordination mechanisms. Or, rather, between the *four* coordination mechanisms introduced in this document, because there are also the **flag** and **protocol array** mechanisms. The **simplest** one is the **flag**:

- it has only one state, with Boolean semantics: “true” / “set” or “false” / “clear”.
 - it has only two transitions: from true to false, and from false to true.
- Strictly speaking, there should be a third state, namely “not initialized yet”, with the obvious corresponding transitions.

The **second simplest** mechanism is the **protocol (flag) array**:

- it has an *array* of flags, with a *serial order*, or **list**, semantics of the array as a container of flags.
- it introduces a **sequential** dependency between transitions: transitions can only occur from earlier flags in the array to later flags.
- transitions *need not* be between *contiguous* flags in the array.
- strictly speaking, the whole protocol array has also a third state, namely “not initialized yet”, with the obvious corresponding states and transitions of each of the flags inside the array.

Semantically, a flag and a protocol array are both a simple boundary case of a Petri Net, as well as of a Finite State Machine. Indeed, the *structure* of Petri Nets and Finite State Machines is a **graph** model of **dependencies** between behaviours that are more **richer** than the above-mentioned **single state** one of the single flag, or the **list of states** of the protocol array. Flags and protocol arrays can be used *stand-alone* (that is, they are the *full* representation of the coordination state), but they are often also used *together with* Petri Nets and Finite State Machines (that is, whenever **one** coordinated activity need only *view* a simplified part of the coordination state). In other words, they link the *insides* of state machines and Petri Nets to their *outsides* (i.e., how activities can *observe* and *control* the coordination process).

2.10.2 Events and streams

Events and **streams** add the last piece of responsibility in coordination: they *connect* (the actual status of) the representations of the coordination data types between multiple activities, providing (some level of) guarantees in timing (both partial ordering and latency). Events have complementary roles and properties with respect to states:

- events are *produced* (that is, created out of nothing), *consumed* (and then, consequently, deleted for ever).
- events can be *duplicated* for being sent to other activities over multiple communication streams.
- states are non-duplicatable, are created once, and are then *written* (*set* and *unset*) and *read*, but not consumed. Eventually they are *deleted*.
- a side-effect of the *change of the value* of a state *can* be the *firing* of an event that encodes that change.
- vice versa, a side-effect of the *handling* of an event *can* be the change of a state.

2.10.3 Declarative and imperative behaviour models

FSM and Petri Net models are declarative representations of the bookkeeping needed to coordinate the execution dependencies of algorithms and activities. Neither of them is an algorithm or an activity in itself, but just an abstract data type, and the algorithms and activities that use FSM and Petri Net models for their coordination look up the *state* in the FSM model and the *markings* of the Petri Net model, whenever it is an “appropriate time” to do so, and they should do so in conceptually “zero time”. In other words, algorithms and activities “execute” the declarative FSM and Petri Net models, by

- *scheduling*: the declarative model is the input of a scheduling algorithm that computes which transitions *can* be executed.
- *dispatching*: sooner or later, an activity’s *event loop* *selects* the schedule for execution, and (only) then state and marking data structures are updated.

A proper *architecture* is needed to prevent that activities are blocked, because one of their *algorithms* must *wait* while the above-mentioned execution of coordination takes place.

2.10.4 FSM as boundary case of single-token Petri Net

The following observations are obvious but nevertheless important:²⁰

- modelling a coordination with a Petri Net that is constrained to have only one input place per transition, gives it the equivalent semantics as a Finite State Machine.
- a coordination model of a Petri Net with multiple places per transition, and (hence) multiple tokens, can be transformed into an equivalent Finite State Machine as follows:
 - (i) every composition of markings becomes one state in the FSM, and (ii) the FSM’s event transition table contains the composition of the transition logic of all Petri Net transitions connected to the filled places.

The conclusion is that it is, technically speaking, not *necessary* to introduce the two concepts, because a model in one meta model can be transformed into a model in the other meta model. Nevertheless, both meta models have their place to support *human developers* in representing complementary semantics of *coordination*, i.e., the coordination of one single activity’s behaviour, or of the interaction between multiple activities.

2.10.5 Policy: higher-order model for pre/per/post conditions

The transition rules of a the *FSM* or *Petri Net* coordination mechanisms are often *hard coded*, in order to speed up reaction *latency*. So, an event reaction table represents the *first-order* (“factual”) behaviour the coordination mechanism’s transitions. But *to explain why* a transition takes place requires *higher-order* (“semantic”) relations. Indeed, designers typically have knowledge about the system that they can formalize in the form of the following **three types of constraint conditions**:²¹

- **pre conditions**: to be satisfied before a transition should be *entered*, that is, to allow the activity to select the corresponding behaviour.
- **per conditions**: to be satisfied *during* the whole duration of the a *state*’s behaviour.
- **post conditions**: to be satisfied before a state should be *left*.

²⁰References to the literature are still missing!

²¹In many systems, these conditions are not made explicit, which makes it difficult to assess semantic correctness of an activity’s behaviour. Only the first condition is relevant to a transition in the Petri Net context; all three are relevant for Finite State Machines.

These conditions are not events in themselves, but the violation of one of them is an indication, to the activity coordination, that the activity may have to switch to another behavioural state, or to set or clear algorithmic flags. In other words, these constraints encode the *when* and the *why* behind a switch.

When the conditions are available to the system software at runtime, they serve as [assertions](#) to be monitored. That is, one can (i) *check* whether a computed transition is justified, and (ii) *explain why* it is, or it is not.

2.10.6 Good and bad practices

In addition to selecting the right coordination mechanism, system developers can profit from being aware of the following good and bad practices.

- *Bad:* use names for the states that reflect the *reason why* a state has been reached. For example, names appear like “configured”, “error”, “started”, etc. These name choices restrict composability by introducing implicit constraints and assumptions on the *transitions* that should be present in the system.
Good: use names that reflect *what behaviour* the activity is providing while being in a state.
- *Bad:* to use Petri Nets to solve *scheduling* problems; for example, making the decisions about the *order* in which to start actions.

2.10.7 Design similarities and differences

An activity can *be* in only one behavioural state, but can *have* multiple status flags at any given moment in time. For example, the readiness of the activity to be involved in various inter-activity coordinations, such as communication or [data ownership transfer](#) protocols. The consequences for an activity with respect to its FSM and Petri Net models are:

- it is involved in only one FSM all the time, but can participate in zero or more Petri Nets part of the time.
- an FSM can be defined within the scope of one activity; a Petri Net’s design scope is typically at system level, requiring “someone” to introduce the coordination model and the coordination activity.
- every activity owns its FSM completely, and the transition decisions are done completely synchronously with itself.
- an activity commits itself to one or more Petri Nets, and the transition decisions are done by another activity.
- in an FSM, the scope of the decision making is defined very locally, by the current state; while the decision making in a Petri Net must consider the full graph every time.

So, the asynchronicity complexity of a Petri Net coordination can be much higher than that of an FSM coordination. In each of the Petri Nets, the scope of the decision making is, canonically, all of the transitions. However, the second-order coordination via the [inhibition/activation](#) place, and the third-order coordination via [time outs](#) or [heartbeats](#) reduce the scope of transitions to consider.²² That scope influences the complexity (and hence the timing) of computing the transition tables. Often, a larger influence comes from the [monitoring algorithms](#) in all activities that have to compute the local conditions under which their activity must produce or consume a token or an event.

²²TODO: more examples and motivations.

Chapter 3

Meta models for point and polygon geometry

This Chapter introduces the meta models for the **simple** geometrical entities: **point**, **line**, **polygon** and **polyhedron**. In other words, the geometrical “**body**” entities in one-, two- and three-dimensional spaces. Primarily the **Euclidean** space, of **mechanics**, but also the less constrained **affine** and **projective** spaces, whose concepts of **incidence**, **cross-ratio**, and **parallelism** are very relevant in robotics.

A **geometric chain** model extends the body model, by adding **relative motion constraints** between the geometric entities: some of them are not free to move with respect to each other in all degrees of freedom of the embedding space. The connectivity structure (“**topology**”) of a geometric chain relation can be a list, a tree, or a graph. And the motion constraints can come in static or time-dependent versions. The simplest example of a time-*independent* geometric constraint between points, polygons and polyhedra, is the **rigid body**: all the points in the geometric body remain at the same relative distance, at all times.

Coordinates are a primary example of a semantic relation between attachment points, being the link to **measurements**, manual or via sensors. The same set of geometric entities can have multiple coordinate relations: each of multiple measurements provides one such coordinate set; the **physical units** of two coordinate sets can be different; one set of coordinates can represent the **expected** or **desired** values; etc.

Geometrical models have multiple parameters, often determined by sensors. This inevitably introduces **uncertain** and **partial** information about those parameters. Hence, geometrical models must allow to represent worlds with **inconsistent**, **ambiguous** and/or **incomplete** geometrical relations.

3.1 Geometric spaces and their spatial relations

This Chapter introduces meta models for the following common geometrical entities and relations:

- a **point** in a **space**, as the **axiomatic** primitive of geometrical modelling.
- a **line segment** as the composition of two points.

- a **polyline** as the composition of two or more line segments.
- a **polygon** as the composition of (i) a polyline and (ii) the *constraint* that its start and end points are identical.
- a **polyhedron** as the composition of several polygons that enclose a *volume*.
- an **attachment** (point) as a **collection** relation on one or more of the above entities. One single point can, in its own, already serve as an attachment. The role of the attachment concept is to serve as an argument in knowledge relations, where it represents the symbolic access to (a set of) geometric entities.
- (**semantic**) **tag** as a (**symbolic**) **relation** in an application domain that has at least one attachment as an argument. Each semantic tag serves as a **type** for a relation in a knowledge graph.
- the semantic tag of **geometric chain** constraint, of a (partial) order of any of the geometric entities above.
- the semantic tag of (**polygonal**) **shape** as a geometric chain of polygons (or rather, a graph), with a **closure** constraint.
- the semantic tag of the **type** of the space, as one of **Euclidean**, **affine** or **projective**; the semantic tag of a second **type** of the space, its **dimensionality**, as **0D** (a “*point*”), **1D** (a “*curve*”, with a “*line*” as special case), **2D** (a “*surface*”, with a “*plane*” as special case), or **3D** (a “*volume*”).
- the semantic tags of (**relative**) **position**¹ and **direction**, and their time derivates of **velocity** and **acceleration**.
- the semantic tag of **motion**, as the composition of the semantic tags of position, velocity and acceleration. These three always come as a package, in an object’s *state* of motion.
- the semantic tag of a **motion constraint** on the relative position, direction, velocity or acceleration.
- **line** and **plane**, as **relative position constraints** between a possibly infinite number of points in a space. The generic term *space* will be used to denote any line, plane or space.
- the semantic tag of **rigid body**, as a **motion constraint** on points in a Euclidean space that imposes **constant relative distance**.
- the semantic tag of **frame**, as a particular instance of a rigid body with an **origin** and a number of **direction vectors**, equal to the dimensionality of the space. The principal use of a frame is as an argument in a **coordinate system**² relation.
- the semantic tag of a **Cartesian** coordinate system, adding the constraints of **unit length** and **mutual orthogonality** to the direction vectors of a frame.
- the semantic tag of **coordinates**, as a relation that connects the relative motion of two geometric entities to numerical values, with each of these values having:
 - a **dimension** (e.g., *length*) and a **physical unit** (e.g., *meter*).
 - two geometric entities to make the interpretation of the numerical values unambiguous: (i) the “*observer*” geometric entity from whose point of view the relative motion is quantified, and (ii) the “*frame*” geometric entity (or **datum**) that serves as coordinate system.

So, in total four geometric primitives must be identified explicitly before the coordinate values of a geometric motion can be correctly interpreted. Often, one and the same

¹Relative position is the simplest form of a **spatial relation**.

²Also called a *coordinate reference system*.

geometric primitive plays the *role* of two or more of the required four primitives.³

The same relative motion of the same two entities can have multiple coordinate relations.

- the semantic tag of a **map**, as a collection of relations between geometrical entities, and entities and relations in a particular domain. A map comes with a **legend** of domain-specific semantic tags that can be found on the map, and with one or more **indexes** into the map’s “database”.
- the semantic tag of an **atlas**, as a collection of maps.
- the semantic tag of a **gazetteer**, as the composition of a map with domain-specific data about the entities in the map. The robotics context of this document uses the term **world model** for such a composition of geometric information with the perception and control **affordances** of robots.

A model *can* conform to multiple of these types at the same time; for example a *planar* (1) *polygon* (2) in the *3D* (3) *Euclidean space* (4). A model *must* conform to the composition of **one** of the types from **both** the dimensionality and the space categories. Not every combination of such types is semantically correct; for example, a rigid body constraint is a Euclidean relation, without meaning in affine or projective spaces.

3.1.1 Semantic IDs for geometry in 1D, 2D and 3D

Section 1.8.1 describes the generic foundations of semantic metadata, namely the usage of IDs as “symbolic pointers” between models and meta models. This Section adds geometry-specific specialisations, with the following suggestion for the meta meta model IDs (**MMID**):

```
{"MMID" : [ "Geometry", "3D", "Euclidean", "PointPolylinePolygonal" ] },      (3.1)
```

with similar notations for other geometric entities and relations, that are relevant in 1D and 2D spaces. The *order* of the semantic tags in the MMID container "[..., ..., ...]" above has meaning: **Geometry** is the **top-level** meta meta model, and the 1D, 2D and 3D tags are more *specific* than **Geometry** but at the same time also more *generic* than the identification of the geometrical space of the described entities and relations as **Euclidean**, **Projective** or **Affine** (Sec. 3.1.2); the last tag identifies a the *most specific* geometric meta model, namely that of *Point-Polyline-Polygon* introduced in this Chapter.

All of these entities and relations belong to the realm of the *mathematics* meta meta model. This document sometimes uses *composite* MMID tags P(2), A(2), E(2) and P(3), A(3), and E(3) for, respectively, the projective, affine and Euclidean spaces in two and three dimensions; the composite MMID tag R(n) represents the real line in n dimensions. Further MMID tags are introduced in the Sections below.

The 1D meta model is trivial: it represents a **line**, a **circle**, or a **curve**, or any other geometric primitive that can be mapped, continuously, to (a segment of) the real line. The 2D meta model (Sec. 3.2.1) is the core meta model, representing a **manifold**⁴ of zero-dimensional (“0D”) Points, and with simple composition rules to build all other entities. The 3D meta model (Sec. 3.2.2) is presented separately, because:

- 2D geometry is sufficiently relevant in itself to merit its own meta model.

³For example, as represented by the concepts of **allocentric** and **egocentric** motion, which complement that of motion with respect to an *absolute*, or **geocentric** reference.

⁴A *manifold* is **continuously mappable** on \mathbb{R}^n , with \mathbb{R} the *real line*. Hence, \mathbb{R}^n is always one of the meta meta models of any geometric meta model in this document.

- all 2D semantics hold in 3D too, but not the other way around, so 3D geometry is a *composition* of the 2D meta model with some *extra* semantics. For example, the 2D concept of *area* keeps its meaning in 3D, but *volume* is a meaningless concept in 2D.

3.1.2 Geometric relations in projective, affine and Euclidean spaces

All **entities** and **relations** introduced in this Section have meaning in Euclidean, affine and projective spaces. The **point** is the fundamental entity. The composition of points into **lines** brings extra **constraint** relations, namely **collinearity**, **are-equal**, **intersect** and **have-ratio**, which hold for *all* mathematical spaces referred to in this Section. The line relation is where the three mentioned spaces differ from each other, and there is a clear **hierarchy**: projective geometry has the smallest amount of relations; affine geometry conforms to all of them *and* extends them with its own (rather large) number of relations; and finally Euclidean geometry is on top of the hierarchy, with a couple of relations more. None of these spaces has a **natural origin**, because any point in the space serves equally well as reference. This implies that “position” can never be a *property* of the representation of a geometric *entity*, but only a property of a *relative position relation* between geometric primitives.

The **projective space** has as key relations (i) the **incidence** of points and lines, and (ii) the **cross-ratio** between sets of two points on *four* lines.

The **affine space** extends the projective space with the **relations** of (i) **parallelism** of lines, (ii) (ii) **barycentric** coordinates, and (iii) **convexity**, and (iv) **Pappus’ law** relation between two pairs of three points on *two* intersecting lines. The latter means that an affine description of points and lines keeps the relative order between them; for example, an affine geometric model that has a street between two other streets, keeps that street in the middle, no matter what **affine transformation** is applied to the model. Examples of affine transformations are **translation**, **scaling**, **similarity**, **reflection**, **rotation**, and **shear**.

The **Euclidean space** represents relative position and **translational motion** of points. Euclidean space has a **distance** relation, which is a **quadratic form** that maps two points to a real scalar. That means the terms **length** and **angle** have meaning. The relations **are-orthogonal** (of lines) and **triangle inequality** are consequences of the **distance** relation. Within the Euclidean space, the following *invariants* exist:

- **similarity**: a first object is similar to a second object, if both have the same shape. That is, the first object can be put exactly on top of the second object by some combination of (i) *translation*, (ii) *mirroring* and/or (iii) *scaling*.
- **translation**: a translation of an object preserves its **orientation**.

3.1.3 Non-Euclidean space for rigid bodies

When points and lines are being connected together to form “rigid bodies” in the Euclidean spaces E(2) and E(3), a new relation of **orientation emerges**. It is just the shorthand for the set of **constraint relations** between all points that their mutual distance is constant over time. All of the above-mentioned mathematical entities and relations remain meaningful, but extra semantics is introduced because of the dependency between *translation* and *orientation*: because of the constant-distance constraint, translating one point has an impact on

the translation of the other points it is rigidly connected to. It turns out that the semantics introduced by the constant-distance constraint has nice mathematical (*Lie*) group properties, represented by the following two spaces:

- the **Special Orthogonal group and algebra** in two and three dimensions (**$\text{SO}(2)$** , **$\text{so}(2)$** and **$\text{SO}(3)$** , **$\text{so}(3)$** ; MMID: `Geometry::SpecialOrthogonalGroup::2D/3D`) represent the semantics of “pure” orientations. This space has a well-defined “distance” metric between any two orientations (the smallest angle over which to reorient the first orientation to make it **equal-to** the second orientation. But there is no “unambiguous zero” rotation, because rotating a body over 360 degrees brings it back to its original orientation.
- the **Special Euclidean group and algebra** in two and three dimensions (**$\text{SE}(2)$** , **$\text{se}(2)$** and **$\text{SE}(3)$** , **$\text{se}(3)$** ; MMID: `Geometry::SpecialEuclideanGroup::2D/3D`) represent the semantics of the **coupling** between **translations and orientations**.

One of the major constraints on $\text{SE}(2)/\text{SE}(3)$ is the **lack of a bi-invariant metric** relation [75, 74, 88, 89]. 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 [88].

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

3.1.4 Projections in geographical coordinate system

Geographic information systems work with geometries that are different than the ones in the previous sections: each of the many choices for a **geographical coordinate system** on the Earth comes with its own **map projection** mathematics. The more or less spherical shape of the Earth implies that even projective geometry is not sufficient, because the map projections transform **straight lines** in coordinate space to **curved lines** in map space, and the other way around.

3.1.5 Qualitative spatial relations — Symbolic geometric chains

Geometrical information is not always numerical and/or directly observable. Many geometric relations are qualitative (and hence purely symbolical), such as:

- *the robot is in front of the table.*
- *move towards the door until it is within arm reach.*
- *hand over the bottle from right to left hand.*
- *there are two cars in front of this car.*
- *move to your left.*

The list of qualitative relations is long: **connects**, **contains**, **borders**, **intersects**, **on_top_of**, **left_of**, **behind**, **in_front_of**, **in_between**, **in_middle_of**, etc.

An important argument in these geometric relations is the **point of view**: does a relation hold from one own's point of view, or from someone or something else's point of view? For example, for one observer, an object can be **behind** another object, while it is **in_front_of** that object for another observer.

(TODO: qualitative Spatio-temporal reasoning, spatial relations between regions in space, e.g. Region Connection Calculus (RCC8), DE-9IM, or Cross Calculi [21, 42, 83].)

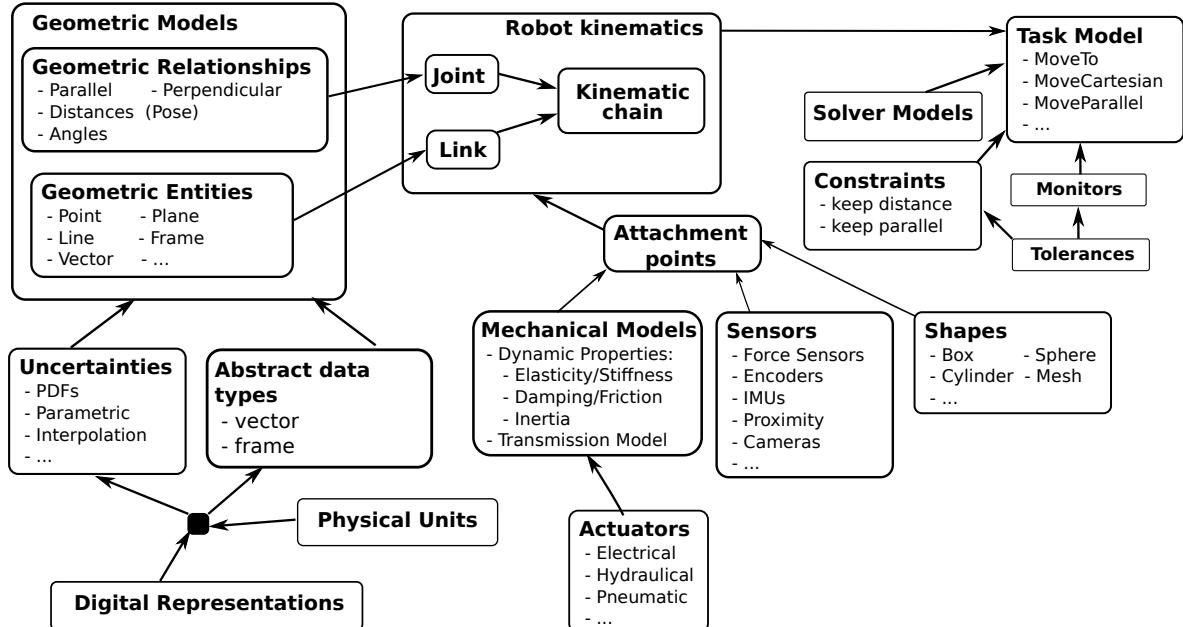


Figure 3.1: The **mereo-topological** overview of the “motion stack” in which a majority of the depicted blocks is a *structural* composition of geometrical entities and relations, or a “*higher-order*” composition on top of such a geometrical structure. Each arrow represents such a composition of complementary and separated *models*. The “black square” arrow represents an *n-ary* composition relation.

3.1.6 Taxonomy of geometric meta models

This document introduces the **partially ordered hierarchical structure** of Fig. 3.1 as a modelling axiom (a so-called **taxonomy**) for the (mostly geometric parts of) **motion**, **perception** and **world modelling** “stacks”. This partially ordered modelling structure serves as **structural backbone** of reasoning, querying and model transformations; each of the entries in itself is candidate to be (meta) modelled by at least one formal model, with **semantic metadata** linking between them. The lower levels of abstraction (with the lowest numbers in the descriptions below) represent the (almost) domain-independent aspects of mereology (the *set* of “things” in the model) and topology (the *connections* between “things”). The gradually more “domain”-specific levels are motivated by specific sets of use cases in robotic world modelling, motion and perception. As far as the *geometry*⁵ parts of the taxonomy are concerned, this document

⁵The taxonomy includes **dynamics** too, which is covered by the mathematical theory of **differential geometry**.

introduces the following partially ordered set of levels of abstraction, using the *kinematic chain* as the running example to illustrate the kind of knowledge representation required at each level:

1. **Mereology**: this abstraction meta meta level is introduced in Sec. 1.7.1, and models the “whole” and its “parts”, with just **has-a** as relevant relation, and the resulting **collection** as relevant higher-order entity.

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

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

A kinematic chain **has-a** its whole-part relation with its **links** and its **joints** as **objects**, and its **workspace** as the **manifold** of all reachable positions of its parts.

2. **Topology** with its **Contains** and **Connects** relations. This abstraction level introduced in Sec. 1.7.2 models “neighbourhood”, more in particular the **contains** and **connects** relations, in, both, discrete and continuous spaces. There are also topological relations in non-geometric *taxonomies*, for example in all knowledge models where hierarchical **hypernym/hyponym** relations have been identified (not only for objects, but also for verbs).

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

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

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

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

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

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

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

The context of a **Task** implies that a kinematic chain **has-a** lot of **points** attached to its **links**, and whose geometrical **position** and **velocity** in space are of interest in the task for which the kinematic chain is used.

- 3.1 **Affine geometry**: this introduces non-metric geometrical entities, such as **point**, **line**,

and **hyperplane**, and relations such as **intersect**, **parallel**, and **ratio**.

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

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

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

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

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

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

A kinematic chain transforms mechanical energy between (i) the motors attached to its **joints**, and (ii) its **links**. The latter can themselves transform mechanical energy with objects in the chain’s environment; for example, when pushing a box over a table.

- 4.0. **Differential geometry**: this is the domain-independent representation of physical systems, that is, all features that are shared between the mechanical domain, hydraulic domain, electrical domain, thermal domain, etc. The most fundamental concepts are: **Tangent_space**, **Linear_form**, **Vector_field**, and **metric**.

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

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

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

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

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

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

3.1.7 Levels of representation in geometry

Section 2.2 introduced a natural hierarchy in the composition of knowledge representations, and this Section applies this hierarchy to the context of formal meta models in geometry, *and* adds two other representation forms that are cutting across the generic hierarchy: *physical units* and *uncertainty*. These formal models apply to (a selection of) the geometric meta models described in Secs 3.1–3.1.6.

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 [20, 34], and these semantic relations have the **dependency structure** of Fig. 3.2. A summary of the relevant levels of representation is:

- **mathematical**: each geometric entity can be represented by (“composed with”) multiple mathematical models. For example, a **LineSegment** is the set of all **Points** that lie on the **Line** between a **start Point** and an **end Point**.
(This document does not pursue the mathematical modelling, because that is beyond its scope.)
- **abstract data type**: each of the above can be represented with multiple *abstract data type* models. For example, a **LineSegment** is an **array** with two members, with one member having a metadata tag of **start** and the other member a meta tag of **end**.
- **data structure**: each of the above must be represented in a concrete programming language with the available (instances of) *data structures*. For example, the **array** in **C** comes in the following forms of an ordered list (array), or an ordered list of ordered lists (matrix):

```
int cat[10]; // array of 10 elements, each of type int
int a[10][8]; // an array of 10 elements,
               // each of type 'array of 8 int elements'
```

- **digital storage**: each of the above must be represented with a particular digital representation model, to describe how the information is encoded in bits and bytes in the **RAM** and the **hard disks** of computers, and in the **messages** that computer processes exchange the information with. For example, each **int** in the array above is represented by four bytes, with the **little endianness** arrangement.
- **physical units**: *all* of the above must be composed with a model of the relevant *physical units*. For example, the above-mentioned **floats** are of the type **length** and have a unit of **meter**.
- **uncertainty**: *all* of the above must be composed with a model of the possible **uncertainty**. For example, a **Point**’s uncertainty can be represented by the **standard deviation** of the numerical values in its digital representation.

3.2 Mechanism: point, polyline and polygon models

This Section presents the meta models for **spatial information** that is represented in **vector graphics** format. The major semantic concepts are:

- the **entities** of **points**, **lines**, **line segments** and **polygons**.
- the **relation** of a **map** as an ordered or unordered **set of sets** of geometric entities.
- the **relations** of **instantaneous motion** of the entities.
- **rigid body** as a **constraint** on that instantaneous motion relation.

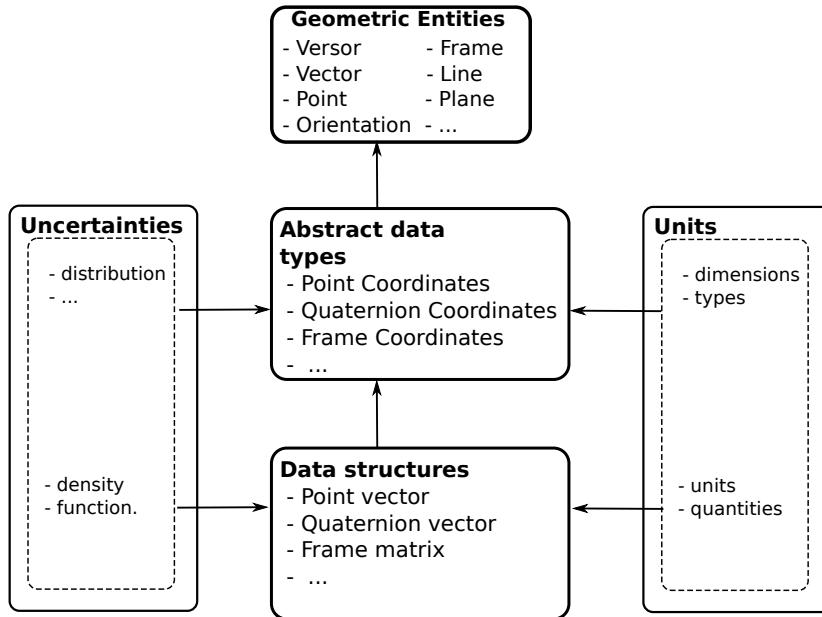


Figure 3.2: The mereo-topological model of the representations of geometric entities and relations, that is, the container entity for “mathematical” (i.e., *geometrical*), “abstract data type” and “data structure” representations. An arrow represents *composition* of complementary aspects in that numerical representation. Models of physical units *must* be composed always; models of uncertainties *can* be composed, when needed.

These concepts are fully in line with this document’s focus on *simple* meta models, because they play a **first-class citizen** role in all representations of the real world, but remain easy to comprehend fully, and (hence) to compose in more complicated entities. The adjective “simple” is also a tribute to the **Open Geospatial Consortium**’s efforts in standardizing **simple features** for all kinds of **geospatial** purposes. The meta model is hierarchically structured into various levels of abstraction:

- **mereo-topological** naming of entities and relations (Secs 3.2.1–3.2.4).
- abstract data types for symbolic model validation and model-to-model transformations (Sec. 3.2.6).
- data structures to carry **coordinates** (Sec. 3.3).

Each meta model on a “higher” level of abstraction can be represented by multiple meta models on a “lower” level. Hence, separating the definition of all these meta models helps **composability** of modelling efforts.

Modelling in this Section starts with the **Point** entity, and all other entities and relations are **compositions** of that **Point** model. (The **monospaced font style** is used in this Section to denote keywords in the meta models. For example, **Point**.) This model composability approach is motivated by:

- the desire to keep the number of “basic” *abstract data types* small.
- the desire to put entities, relations, and the constraints on those relations, into separate meta models, because there are many use cases for the separate meta models.
- the fact that most sensors in robotics *measure points*, and not lines, planes or bodies.

- the fact that for points the concept of *uncertainty* is uniquely defined, but not for lines, planes or bodies.
- the straightforward way to provide all possible kinds of “*attachments*” to compose into richer semantic structures.

Examples of the latter are *geometric chains* and *kinematic chains* (Chap. 4), or *maps* or *world models* (Chap. 5). The knowledge in the combination of this Section’s meta model for geometry and those for kinematic chains and “OpenStreetMap”-like world models, is sufficiently mature and complete to allow their formal representation to start a community-driven process towards a **vendor-neutral open standard**.⁶

3.2.1 Point entity and its composition relations

Point — A **Point** is the *zero-dimensional* element of the space **manifold**, so it has no properties of length, area, volume, or shape. None of this knowledge must be represented explicitly; just referring to the **mathematical meta meta models** suffices: does the **Point** live in a 2D or 3D space, and is this space Euclidean, affine or projective? In other words, the mathematical representation of a **Point** is just a **symbol**, to represent its **identity**. This document chooses the notation

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

to identify a model of a **Point**. With all **Semantic_ID** information, the full version of such a **Point relation** is depicted in Fig. 3.3. The following Sections will only use the short version.

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

Figure 3.3: Mereological model of a **Point**, with **Semantic_ID** metadata.

Vector — This **ordered** list of two **Point** entities adds three constraints: (i) the *ordering* that gives an *orientation* to the **Vector**, (ii) that list contains *exactly two* members (and not all the points on the line in between), and (iii) the two **Points** in the list are different. One of the **Points** gets the *attribute* of **start**, and the other that of **end**. These parameters are not a *property* but an *attribute*, because different contexts can give different *orientations* to the same **Vector**. The model is the composition of two **Point** models:

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

The model above uses ID strings that refer to **Point** models as in Fig. 3.3. If desired, the shared **MID** and **MMID** information can be taken out of the **Semantic_ID** of each of the composite **Points** and put in the **Semantic_ID** of the **Vector**.

⁶With clearly identified advantages with respect to the robotics *de facto* standard **URDF**: (i) all possible kinematic chains can be represented and not just tree structures with one-dimensional revolute or prismatic joints; (ii) any other meta model can be *composed* with the kinematic chain model without requiring that other meta model being visible or even known; and (iii) in particular, any type of actuator and control algorithm, including **multi-articular** configurations.

The following are **Vectors** with *constraints*:

- **Line_vector**: the **start** point is constrained to be *somewhere* on the **Line** through the **start** and **end** **Points** of the **Vector**.
- **Free_vector**: the set of **Vectors** starting in every point of the space, all forming parallelograms with every other **Vector** in the set. (Because of its dependency on the **Parallel** relation, this concept does not exist in projective space.)
- **Versor**, or **Direction_vector**: a **Free_vector** that represents just an **orientation**, that is, whose **length** has no meaning. Of course, the constraint remains that the **start** point is different from the **end** point.
- **Unit_vector**: a **Direction_vector** in the Euclidean space, whose **length** is constrained to be of *unit length*.

Their formal models are straightforward compositions of the **Vector** model, with the extra constraint relations.

Polyline — A polyline is an **ordered** set of **Points**. It is equivalent to an ordered list of **Vectors**, with the **constraints** that (i) the **start** point of one **Vector** is the **end** point of the previous **Vector** in the ordered set, (ii) each **Point** belongs to exactly two **Vectors**, except for the **start** point of the first **Vector** and the **end** point of the last **Vector**. The notation is:

$$\{ \text{Polyline} : [\text{aPoint-ID}, \text{bPoint-ID}, \dots, \text{zPoint-ID}] \}, \quad (3.4)$$

where **aPoint-ID**, **bPoint-ID** and **zPoint-ID** are the unique IDs of **Points**. **Length** is a scalar real-valued **property** of the **Polyline**, whose value is the sum of the **Lengths** of each of the **Vectors**. The **orientation** of the **Polyline** is an **attribute**, that is also inherited by every **Vector** in the **Polyline**. The notation is

$$\{ \text{Polyline} : [\{\text{start} : \text{aPoint-ID}\}, \text{bPoint-ID}, \dots, \{\text{end} : \text{zPoint-ID}\}] \}. \quad (3.5)$$

Simplex (in a 2D space) is an **ordered** list of **two non-collinear Vectors**, with the **constraint** that both have the same **start** point. The notation is:

$$\{ \text{Simplex} : [\{\text{start} : \text{aPoint-ID}\}, \text{bPoint-ID}, \text{cPoint-ID}] \}. \quad (3.6)$$

The extension to a 3D space is obvious: just add one more **Point**, and adapt the **Semantic_ID** accordingly.

Orientation⁷ is the **attribute** of the ordering: the order *from bPoint-ID to cPoint-ID* being **positive** or **negative** is a *convention* given by the application context.

Frame is an entity that only has meaning in the *Euclidean* context, because it adds two **metric constraints**⁸ to the **Simplex** entity: (i) the **length** of each **Line_segment** is the unity length, and (ii) the **Line_segments** are **perpendicular** (or **orthogonal**). The **orientation** of a **Frame** is typically an *essential* attribute, because of its use to represent **Coordinates** (Sec. 3.3), so the notation has somewhat different semantics than the **Simplex**:

$$\{ \text{Frame} : [\{\text{origin} : \text{aPoint-ID}\}, \{\text{end} : \text{bPoint-ID}\}, \{\text{end} : \text{cPoint-ID}\}] \}. \quad (3.7)$$

⁷**Right_handed** and **Left_handed** are often used synonyms.

⁸These constraints are the same in 2D and 3D.

The extension to a 3D space is obvious: just add one more **Point**, and adapt the **Semantic_ID** accordingly.

The **Frame** is a spatial shape entity, but it is often just used for its attribute of **orientation**, in 2D and 3D spaces. (The **Simplex** too, for that matter.) Some common *policies* to represent **orientation** are:

- explicit ordering of the **Frame**'s **Vectors**, via **semantic tags** that reflect *numerical* order (1, 2, and 3), *alphabetic* order (*X*, *Y* and *Z*), or *color coding* order (*R*, *G* and *B*, after the deeply established **RGB** color model).
- the binary **orientability** choice between **Right_handed** or **Left_handed**.

The notation for a **Frame**-based **Orientation** is:

$$\{ \text{Orientation} : \text{Frame_a} \}. \quad (3.8)$$

Polygon — This is a **Polyline** with the extra **constraint relation** that the two not yet connected **start** and **end** **Points** must now coincide. The notation (for an oriented **Polygon**) is:

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

The **orientation attribute** is inherited by all **Polylines** inside the **Polygon**. The **interior** of the **Polygon** is represented by an extra **attribute**, namely one single **Point** outside the **Polygon** lines:

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

In the Euclidean context, **area** is a **property** of the polygon.

Polygon_pair — The **unordered** set of (i) a set of two **Polygons**, and (ii) a **Point** that has the **attribute interior**. The notation is:

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

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

$$$$

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

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

In the Euclidean context, **area** is a **property** of a **Polygon_pair**, namely of its **interior** part.

Multi_Polygon — An **unordered** set of **Polygon_pairs**. In the Euclidean context, **area** is a **property** of the **Multi_Polygon**, as the sum of the areas of each **Polygon_pair**.

3.2.2 Extra composition relations in 3D

All of the 2D entities keep their meaning in 3D too. This Section introduces the extra 3D-only entities.

Polyhedron — This is the generalisation of the **Polygon** to a 3D shape with a surface that has no holes. That is, it consists of a set of **Polygons**, which all mutually share some **Polylines**. The notation is illustrated with the simple example of the **Polyhedron** in Fig. 3.4:

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

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

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

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

]]. (3.20)

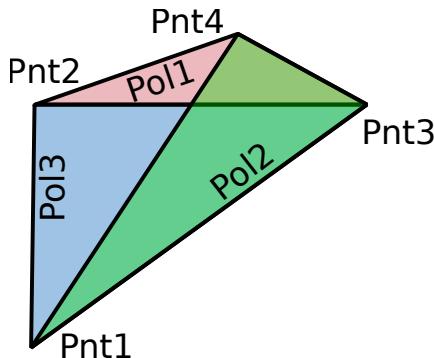


Figure 3.4: A **Polyhedron** as the composition of four **Polygons**, each the composition of three **Points**. (The fourth polygon is not shown in colour, in order not to overload the drawing.)

The model above could have been composed differently: some of the four **Polygons** could have their own model already, so that the **Polyhedron** can refer to them. For example:

 Pol1 : [{start : Pnt2-ID}, Pnt3-ID, Pnt4-ID, {end : Pnt2-ID}], (3.21)

 Pol2 : [{start : Pnt1-ID}, Pnt4-ID, Pnt3-ID, {end : Pnt1-ID}], (3.22)

 Pol3 : [{start : Pnt1-ID}, Pnt3-ID, Pnt2-ID, {end : Pnt1-ID}], (3.23)

{ Polyhedron : [Pol1-ID, Pol2-ID, Pol3-ID, (3.24)

 Pol4 : [{start : Pnt1-ID}, Pnt4-ID, Pnt2-ID, {end : Pnt1-ID}] (3.25)

]]. (3.26)

Volume remains a valid *property*, as is **orientation**.

Simplex, **Frame** — The 3D versions differ from the 2D versions in that (i) there are **three Vectors** in the **Simplex** or **Frame**, and (ii) their mutual constraint is on being **non-coplanar**.

Plane — A **plane** is the third axiomatic primitive of geometry (next to **Point** and **Line**), that represents a *surface* in the 3D space, or, in other terms, a two-dimensional *subspace*. Different ways to represent a **Plane** are:

- the *join* of three **Points**.
- the *join* of two intersecting **Lines**.

- these two **Lines** can come from two **Vectors** in a **Simplex**.

All three ways are equivalent in their composition of **Points** in somewhat different ways. A fourth **Point**, or a third **Vector**, can be used to determine the **orientation** of the **Plane**.

Half_space — This contains all the points in space which lie on one side of the *supporting Plane*. The defining **Simplex** of the **Plane** defines the **orientation** attribute of the **Half_space**. A common alternative **orientation** representation is to give one single **Point** outside the **Plane**.

3.2.3 Position and Motion relations of a Point

This Section extends the semantics of the **Point** entity and its mereological composition entities, with the relation of the **Motion** between two such entities. A **Motion** of entities is always **relative** with respect to each other, or relative with respect to themselves. Hence, **Motion** is not a *property* or *attribute* of one single entity by itself, but it is the property of the (geometric) *relation* between two entities. Hence, the same geometric entity can have several **Positions** and **Motions** at the same time: one for each other entity involved in a **Position** or **Motion** relation.

Motion has three parts: the **Position** relation, and the relations of **Velocity** and **Acceleration** that represent the first- and second-order variations over time of the **Position** relation. All geometric entities are compositions of **Points**, so it suffices, strictly speaking, to model the **Motion** of a **Point**. **Motion** has different interpretations, as a mapping over **time** and/or over **space**:

- **time mapping**: the **time derivative** of the **Position** of one particular entity gives its **Velocity**, and the time derivative of its **Velocity** gives its **Acceleration**.
- **spatial mapping**: two entities of the same type are a **Displacement** away from each other, irrespective of where they are, or whether or not they are moving with respect to each other. So, the **Motion** maps the first entity onto the second one.

Motion has **passive** and **active** semantics:

- **passive**: the relation between *two* moving entities **has-a** set of properties that represent their *mutual Position, Velocity and Acceleration*.
- **active**: a **Displacement** can be interpreted as an **action** to move *one* particular entity from one **Position** to another **Position**.

The following paragraphs introduce the *notations* and *terminology* used in this document to represent **Motion**.

Position and **Displacement** — These are the relations between two **Points** that represents where they are with respect to each other in space. Both relations are mathematically equivalent, represented by a **Vector**, because this is already a geometric relation between a **start** point and an **end** point. The nomenclature **Position** and **Displacement** is used to indicate the following semantic interpretations of the **Vector**:

- **Position**: the two **Points** in the **Vector** are two **Points** with a different identity; the **Vector**'s geometric interpretation is that of the *instantaneous* relative position of these two **Points**.
- **Displacement**: the two **Points** in the **Vector** are twice the *same Point* (that is, with the same *identity*) but at different moments in time. The time scale is not represented explicitly, because it is not considered relevant. Only the *order* over time is relevant.

The notation of the **Position** of Point “ePoint” with respect to Point “fPoint” is an obvious extension of that of a **Vector**, with the tags **start** and **end** replaced by contextually more meaningful synonyms:

$$\{ \text{Position} : [\{\text{of} : \text{ePoint-ID}\}, \{\text{with_respect_to} : \text{fPoint-ID}\}] \}, \quad (3.27)$$

where **ePoint** and **fPoint** are two **Point** entities. Hence, a **Position** has a **direction** property, equivalent to the **orientation** property of the equivalent **Vector**. There exists an (obvious) **inverse** relation:⁹

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

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

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

Velocity, Acceleration — These are the first and second time derivatives of the **Position** relation. The **Velocity** of Point “ePoint” with respect to Point “fPoint” is denoted as:

$$\{ \text{Velocity} : [\{\text{of} : \text{ePoint}\}, \{\text{with_respect_to} : \text{fPoint}\}] \}, \quad (3.30)$$

The velocity of a point with respect to itself is a physically valid relation.

3.2.4 Position and Motion relations of a Rigid_body

The **Motion** relation of a **Rigid_body** in Euclidean space is the set of the **Motions** of all of the **Points** in the **Rigid_body**. However, there is the **constraint** that the **Motion** preserves the **distance** between all these **Points**, and, *hence*, also **length**, **angle**, **area** and **volume**. Mathematicians have realised that this constraint allows to represent the above-mentioned possibly infinite set of **Motions** of all **Points** on the **Rigid_body**, to a finite-dimensional mathematical group structure, irrespective of how many points are considered in the **Rigid_body**. That group structure has **three dimensions** in $E(2)$ and **six dimensions** in $E(3)$. For the **Position** part of a **Motion**, these groups got the names of, respectively, **SE(2)** and **SE(3)**, the **special Euclidean groups** of the **Displacements** of a **Rigid_body**. For the **Velocity** part, the dimensionalities of the groups are the same as for **Displacements**, and they are denoted by, respectively, **se(2)** and **se(3)**, the special Euclidean *algebras*. “Algebra” indeed, because there is not just the *addition* operation in the group of **Velocities**, but also a *multiplication* operation, the so-called **Lie operator**. In all mentioned three-, respectively six-dimensional, spaces, any **Motion** relation can be separated into:

- two, respectively three, **translational** degrees of freedom of **one Point** that is chosen arbitrarily on the rigid body.

⁹In other words, both **Position** relations are constrained by a higher-order relation, that of being each others’ inverse relations.

- one, respectively three, **rotational** degrees of freedom, independent of whatever choice of **Points** or **Frames** on the rigid body.

This observation was already made in the 19th century, by the French mathematician Michel Chasles (1793–1881), [28], who formulated¹⁰ the following theorem: the most general **Displacement** for a rigid body is a **screw motion**, [7, 9, 23, 60], i.e., there exists a line in space (called the **screw axis**, [9, 70, 124], or “twist axis”) such that the body’s motion is a rotation about the screw axis plus a translation along it. This theorem (and hence the screw axis relation) holds for *Velocities* too.

Although a rigid body is a composition of points, the *Euclidean metric* for points has no equivalent metric on the space of rigid bodies, [75, 74, 88, 89]. In other words, the **distance** between two rigid bodies, the **length** of a rigid body **Displacement**, and the **magnitude** of a rigid body **Velocity**, are not well-defined properties of the **Motion**: these values change when one changes the **Point** on the **Rigid_body** that is used as reference in the **Displacement** or **Velocity** parts of the representations.

The following paragraphs summarize the notations and relations of the **Motion** of a **Rigid_body**.

Position — A **Rigid_body** (or **Simplex**, **Orientation** or **Frame**) is a geometric entity for which the above-mentioned **Rigid_body** constraint holds. As for **Points**, the representation of the **Position** of a **Rigid_body** or a **Frame**, is always relative to another **Rigid_body** or a **Frame**:

$$\{ \text{Position} : [\{\text{of} : \text{aBody-ID}\}, \{\text{with_respect_to} : \text{bBody-ID}\}] \}, \quad (3.31)$$

that is, the position of rigid body **aBody** with respect to rigid body **bBody**. Similarly for other combinations of **Rigid_body** or **Frame**, such as:

$$\{ \text{Position} : [\{\text{of} : \text{aBody-ID}\}, \{\text{with_respect_to} : \text{fFrame-ID}\}] \}, \quad (3.32)$$

and

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

Orientation — The above-mentioned **Position** between two rigid bodies **A** and **B** specifies *only* the three **translational** relative degrees of freedom between both bodies. The missing three degrees of freedom represent the relative **orientation** of the two rigid bodies. The notation is:

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

Pose — This is the name of the *composite* relation of the **Position** and **Orientation** of a **Rigid_body**. That composition adds no extra semantics, it’s just the *set* of both composing

¹⁰The notion of twist axis was probably already discovered many years before Chasles (the earliest reference seems to be the Italian Giulio Mozzi (1763), [27, 56, 101]) but he normally gets the credit.

relations. The notation is symbolically identical to the `Position` and `Orientation` ones:

```
{ Pose : [{of : {Rigid_body : aBody-ID}}},  
  {with_respect_to : {Rigid_body : bBody-ID}}]  
}.
```

(3.35)

`Linear_velocity` — The time derivatives of a `Position` of a `Rigid_body`. Again, the notation is symbolically identical to the ones above. The `Linear_velocity` is a `Line_vector` because it is constrained to point through the `Point` on the `Rigid_body` whose change in `Position` is represented. That choice of `Point` is arbitrary. Hence, and in general, the same `Motion` of a `Rigid_body` can have an infinite amount of `Linear_velocity` attributes. The difference must be made semantically clear by adding an `extra semantic tag` to the `Linear_velocity` model, namely the ID of the `velocity_reference_point` that is chosen in the model.

`Angular_velocity` — The time derivative of the `Orientation` of a `Rigid_body`. Again, the notation is symbolically identical to the ones above. The `Angular_velocity` is a `Free_vector`, because it does not depend on any choice of `Point` on the moving `Rigid_body`.

`Velocity` of a `Rigid_body` — The time derivative of the `Pose` of a `Rigid_body`. There are a large number of possible and equivalent representations. For example, the `Velocity` of at least three `Points` on the `Rigid_body`.

`Twist` — This is a very popular choice to represent the `Velocity` of a `Rigid_body`, due to the fact that it requires only the minimum possible number of independent variables to represent a motion, namely three in 2D and six in 3D. A `Twist` is indeed the combination of a `Linear_velocity` vector and an `Angular_velocity` vector. (In a 2D space, the latter reduces to the choice of a scalar.)

Similar formal models hold for the second-order time derivatives, that is, the time derivative of a `Twist`: `Linear_acceleration`, `Angular_acceleration`, `Acceleration` (or `Acceleration_twist`, or `Rigid_body_acceleration`). There are two different types of time derivative of a `Twist`, hence introducing the need for two extra *attributes*:

- **Eulerian** (or **ordinary**) time derivative: one looks at the matter that flows under one particular fixed point in space, and measures the change of velocity observed at that place over time. So, the `Acceleration` is the difference between the `Velocities` of *two different* particles, at the *same place* in space but at *different instants in time*.
- **Lagrangian** (or **material**) time derivative: one follows one particular point fixed on a rigid body, and measures the change of that body-fixed point's velocity over time. So, the `Acceleration` is the difference between the `Velocities` of the *same particle* at *two different instances in time*, and hence also at two *different places* in space.

The difference between Eulerian and Lagrangian time derivatives is only relevant for `Accelerations`. To derive `Velocity` from changes in `Pose`, the Eulerian time derivative and the Lagrangian time derivative give the same results. Both derivatives also give the same result for `Acceleration` from *rest*, i.e., from zero initial `Velocity`. Different domains use these two different definitions, most often without explicit information, which results in **non-composability problems in robotic systems** that must integrate different domain choices.

3.2.5 Constraint relations on mereo-topological entities

The previous Sections provide *imperative* models: the geometric entity or relation is constructed by a “recipe”. This Section describes *declarative* ways to model geometric entities and relations, that is, the latter are described by a set of *constraints*, and a *constraint solver* is needed to find or check the entity or relation. The constraint relations make use of only the [mereo-topological](#) semantics; constraint relations on numerical [coordinates](#) values are introduced in the [Section on geometric chains](#).

Examples of such [mereo-topological](#) relations are depicted in Fig. 3.6: a distance between a [Point](#) `pPoint-ID2` and a [Line](#) `Line1` can be interpreted as the *shortest* distance between `pPoint-ID2` and another point lying on the [Line](#), or as the length of the projection of `pPoint-ID2` on the [Line](#). Table 3.4 and Table 3.5 resume the different interpretations of linear distances and angular distances.

`Intersection (E, A, P):`

`Cross_ratio (E, A, P):`

`Parallel (E, A):`

`Orthogonal (E):`

`Projection (E, A):`

```
{ Projection : [{from : AsSeenBy_entity}, {to : FromEntity}, {result : ?projection}]}.
```

(3.36)

`Distance, Length (E):` In the Euclidean context, `length` of a [Vector](#) or [Line segment](#) (or the `distance` between the two [Points](#) in these entities) is a *property* of the [Line_segment](#), with a *real scalar value* of physical dimension `length`.

`Angle (E):`

A [Vector_field](#) is a *set* of one [Vector](#) in each [Point](#) in (a subset of) space.

[Line_segment](#) — An **unordered** set of **two** [Points](#) is enough to represent *all* the points on the [Line](#) that runs through the two [Points](#) in the set; it is also enough to represent all the points *between* the two [Points](#) in the set, and that geometric entity is called a [Line_segment](#). The notation is

$$\{ \text{Line_segment} : [\text{aPoint}, \text{bPoint}] \}, \quad (3.37)$$

where `a` and `bPoint` are [Points](#).

[Line](#): is an entity that has a *axiomatic* meaning, in the Euclidean, *projective* and *affine* contexts. That means that one cannot define it formally and completely from more primitive concepts. So, this document *represents* (not “defines”...) a [Line](#) as the *join* of two [Points](#),

that is, the linear combination of the `start` and `end` points in a `Line_segment`.

`Half_line` — The positive or negative part of a `Line`, derived from the defining `Line_-segment`.

`Half_space` — This contains all the points in space which lie on one side of the *supporting Line*. The defining `Vector` of the `Line` defines the `orientation` attribute of the `Half_space`. A common alternative `orientation` representation is to give one single `Point` outside the `Line`, to represent the `interior` of the `Half_space`.

(TODO: `distance` between `Lines`, [71]?; harmonize symbols of points and lines with previous subsections in this Chapter.)

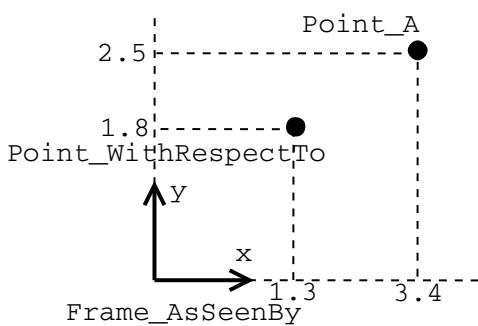


Figure 3.5: The three geometrical entities involved in a generic coordinates abstract data type (in a 2D case): the `Point` (“`Point_A`”) whose coordinates are being represented; the entity `with_respect_to` which the `Position` coordinates are represented (in this case, this is also a `Point`, namely “`Point_WithRespectTo`”); and the numerical values of the coordinates are `as_seen_by` a reference `Frame`, with name “`Frame_AsSeenBy`”. The coordinate values are `x:1.1` and `y:1.3`.

3.2.6 Abstract data types for Coordinates of position and motion

An **abstract data type** (or **coordinate model**, or **coordinate representation**) adds (i) **numerical values** to the mereo-logical models of the previous Sections, and (ii) the **semantic tags** to identify the correct interpretation of those numbers. Recall that for each geometric entity and relation, there exist multiple equivalent possibilities to represent its `Motion`, and for each of these possibilities one can make multiple choices for coordinate model. In general, such choices already exist for each of the **three** essential semantic parts of a `Coordinates` model for a `Point` (Fig. 3.5 and Table 3.1):

- the identity (“`Point_A`”) of the `Point` whose `Coordinates of Motion` (i.e., `Position` or `Velocity`) are modelled.
- the identity (“`Point_WithRespectTo`”) of the `Point with respect to` which the first `Point`’s `Motion` is modelled.
- the identity (“`Frame_AsSeenBy`”) of the `Frame` in which the `Motion Coordinates` numerical values are computed. In other words, these numbers represent the `Motion as seen from the Frame`. They are computed as the `Projection` of the `Motion’s Vector` on the axes of the `Frame`.

For the specific case in which one represents the `Motion` of a `Rigid_body` by a `Twist`, two **extra semantic tags** are needed (Table 3.2):

- the `screw` tag, identifying the `Linear_velocity` and `Angular_velocity` vectors in the `Twist` model.
- the identity of the `Point` that serves as `velocity reference point` of the `Linear_velocity` part of the `Twist`.

```

{ Coordinate_Point_Entity_Frame_Array_Meter :
  { { relation: Position },
    { of: { Point: a} },
    { with_respect_to: { Entity: e} },
    { as_seen_by: { Frame: F} },
    { coordinates: [ {F.x: 1.2}, {F.y: -0,1}, {F.z: 3.2} ] },
    { units: meter },
    { ID: Coordinate143sGa },
    { MID: Coordinate_Data_Structure },
    { MMID: { Coordinates, Euclidean_space,
              Point, Entity, Frame, QUDT } }
  }
}

```

Table 3.1: Example of Coordinates model, in this case the Position of a Point.

```

{ Coordinate_Twist_Frame_Frame_Array_Meter :
  { { relation: Twist },
    { screw: [ linear_velocity: v}, { angular_velocity: w} ] },
    { of: { Frame: b} },
    { with_respect_to: { Frame: r} },
    { as_seen_by: { Frame: F} },
    { velocity_reference_point: as_seen_by_frame_origin },
    { coordinates: [ {v: [ {F.x: 1.2}, {F.y: -0,1}, {F.z: 3.2} ] },
                    {w: [ {F.x: 0.2}, {F.y: 0.8}, {F.z: -2.2} ] } ],
      { units: meter, seconds },
      { ID: Coordinatehd4if7 },
      { MID: Coordinate_Data_Structure },
      { MMID: { Coordinates, Euclidean_space,
                Twist, Frame, QUDT } }
    }
}

```

Table 3.2: Example of a Coordinates model of the Twist of a Frame "b" with respect to a Frame "r". The screw representation has a Linear_velocity and an Angular_velocity part. For the Linear_velocity and Twist of a Rigidbody, the extra semantic tag velocity_reference_point is needed; its value is one out of an enumerated set of possibilities: of_point, on_screw_axis, as_seen_by_frame_origin, with_respect_to_point, etc.

Some common choices of velocity reference point are:

- a point on the entity whose Motion is represented.
- a point on the **with respect to** entity.
- the origin of the **as seen from** Frame.
- a point on the **screw axis**.
- a point that is **relevant** for the robot's current Task.

Geometrical relation	Abstract data type
Position	Position vector
Orientation	Euler-axis angle array
	Rotation matrix
	Euler angles array
	Roll-Pitch-Yaw angles array
	Quaternion array
Pose	Homogeneous transformation matrix
	Finite displacement twist array
	Screw axis array
Linear_velocity	Linear velocity vector
Angular_velocity	Angular velocity vector
	Rotation matrix time derivative
	Euler angle rate array
	RPY angle rate array
	Quaternion rate array
Twist Rigid_body velocity	Homogeneous transformation matrix time derivative
	six-dimensional twist array
	Pose twist array
	Screw twist array
	Body-fixed twist array
	Instantaneous screw axis array

Table 3.3: Commonly used abstract data type representations for geometrical relations.
(TODO: update and complete.)

Because (almost) all geometric entities and relations are compositions of the `Point` entity, defining the `abstract data types` for all these entities and relations is rather straightforward, using the approaches of `higher-order` relations and `semantic_ID` metadata (Sec. 1.8.1), respectively; Table 3.1 shows examples. So, the complexity of modelling abstract data types does not come in the first place from the inherent complexity of the models, but from the large number of *different but equivalent choices* that are commonly used in practice. This *freedom of choice* is often the source of incompatibility, and of lack of composability, of models and implementations created by independent development teams, for the simple reason that not all choices are made explicit, and are not available in formalized form via which system software can check semantic compatibility at runtime, and introduce the appropriate model transformation when an incompatibility is identified. Table 3.3 summarizes the most common choices; the arguments in each composition come from the following list:

- the `name` of the composition relation (from Table 3.3). For example, `Position`, or `Twist`.
- `semantic_ID` metadata. This contains entries like `Geometry`, `E(3)`, and `QUDT`.
- the list of the *arguments* in the relation, each representing a composed entity in the relation. This includes the three or four `semantically tagged` entities, for example a `Frame`, that serve as *references* for, both, the `with_respect_to` and `as_seen_from` fields. Each argument can be a composition relation in itself, with its own `semantic_ID`. A common practice is “to raise” metadata that is the same in different arguments, to the highest level of composition in the relation; for example, it makes sense to use the same physical dimensions for all quantities in a composite model.
- the key-value pairs to represent the numerical values for each of the entities. For example, for the `Position` of a `Point` that means a set of *named* scalars values on the *real*

line:

```
{ [ x: { type = "real" }, y: { type = real" } ] },
```

in 2D spaces, and a triplet

```
{ [ x: { type = "real" }, y: { type = real" }, z: { type = real" } ] },
```

in 3D spaces. The symbol "x" in the **Position** relation must refer to the argument of the reference with the same name. That is why its semantics is that of a “symbolic pointer”, because in the context of abstract data types, “to represent” just means “to be able to refer to symbolically”. For example, an application can already check formally whether the "x", "y" and "z" fields in a model indeed all have the correct type of "real". It can also add extra constraints, for example, a *range limit* on the "z" field. No *concrete numerical* values are given in the representation (yet), just their symbolic type names. The numbers get filled in later, in the next level of composition, namely that of the [Coordinates data structures](#).

- a set of key-value pairs that describe the *properties* of the composition relation. (Or, equivalently, the *attributes* that the composition relation gives to the entities it composes.) For example, the **start** and **end** tags for the two points in a [Line_segment](#). And the **physical dimension** of the numbers: `length`, `length/time`, `angle`,...
No *physical units* are encoded, yet, just their symbolic names. Physical units, such as `meter` or `radian`, are filled in later, as *attributes* in the above-mentioned concrete [Coordinates data structures](#).

There is seldom a unique abstract data type representation for a given geometric concept. For example, it *is* possible that more than three reference values are used to represent a geometric entity in 3D space; That means that there must be a *constraint* that links the four or more reference values, and that constraint must be modelled too. (The advocated way to do this is by referring to a meta meta model in the [Semantic_ID](#) where the constraint is formally encoded.) Another example comes from the fact that the abstract data type contains symbolic information about the [reference](#) with respect to which its model has meaning; that involves the choice of a reference [Frame](#), and while [Cartesian reference frames](#) are a common choice, some use cases are better off with variants like, for example, [polar](#), [cylindrical](#) or [spherical](#) references. In a [projective space](#), one uses [homogeneous coordinate](#) representations, allowing also the points at infinity to be represented by finite coordinate values. In an [affine space](#), one uses [Barycentric coordinates](#), as soon as a [Simplex](#) is given as the basis for the coordinates.

3.2.7 Metadata for coordinates

The numerical values in coordinates often come with extra semantic tags that describe how to interpret the numbers. For example:

- [provenance](#): how were the values generated? That is, with which sensor, using which device and software, etc.?
- [coverage](#): a specific part of provenance, [modelling](#) the *spatial* origin of the numbers.
- [resolution](#): for example, of the [sensor](#), the [time](#), the [computer](#), or the [map](#).
- [accuracy and precision](#): the former refers to closeness of the measurements to a specific value, the latter refers to the closeness of the measurements to each other.
- [tolerance](#): a quantitative indication of acceptable, expected or requested precision.

- **version**: an indication of when changes to the coordinates were realised.
- **time series** and **trajectory**: models of the temporal order of a series of motion coordinate values.
- **encoding**: a model of the representation of the numerical values as bit patterns. For example, the **HDF5**, **RDF Data Cube**, or **Apache Arrow** formats.
- **data catalogue**: the data can be part of a larger catalogue or **data warehouse**.

3.2.8 Operators on coordinates

Because **Coordinates** are not unique attributes of **Motion** of geometric primitives, there exist many transformations between equivalent **Coordinates** representations. This Section describes the different categories of such transformations, and the operators with which to realise them.

Changes of coordinates under changes of:

- **with_respect_to**: (TODO)
- **as_seen_by**: (TODO)
- **velocity_reference_point**: (TODO)

Addition of velocity:

- the **Coordinates** of two **Twist** representations can be added, numerically, but **only if** all the semantic metadata of both representations are identical: same geometric entity, same **with_respect_to**, same **as_seen_by** and same **velocity_reference_point**.
- non-**Twist** representations of **Velocity** do not allow the numerical addition of their **Coordinates** representations. For example, for the representation of the **Velocity** of a **Rigid_body** by means of the **Velocity** representations of three or more of its **Points**, it does not make sense to add individual **Point**'s **Velocity** coordinates, because there are constraints to be satisfied between these **Velocity** representations.

From Velocity to Position, and back:

- the **exponential map** from **Velocity** to **Position** maps a **Velocity** to a **Displacement** (that is, a change in **Position**) of a geometric entity that corresponds to applying the **Velocity** on that entity for **one unit of time**.
- the **logarithmic map** is the inverse of the exponential map: it maps a **Displacement** to the **Velocity** that is required to cover the **Displacement** in **one unit of time**.

3.3 Coordinates — Uncertainty

A *data structure* model adds to things to the *abstract data structure* model:

- the **Coordinates** data structure of a concrete programming language with a set of *numerical values* representing the **quantity** of each coordinate. In the example of Fig. 3.5, the coordinates are an array with two values, the projections of the **Point**'s **Position** onto the **x** and **y** axes of a **Frame**.
- the semantic tags of the **Physical_units** of the numerical values; for example, **meter** or **radian**.

The mapping from the abstract data type to the coordinate data structure is *one-to-many*, because of the choices that can be made in, for example, (i) the *naming* and the *ordering* of the values in the data structure, (ii) the reference **Frame**, (iii) the physical units of each numerical value, (iv) how the data *access* is performed (e.g., via *named keys* or via *anonymous*

indexing), (v) whether the numerical values of the coordinates are projections of the point on real-valued coordinate axes, or integer-valued indices on a grid,¹¹ or (vi) whether several coordinate representations share the same units and reference frame.

One pragmatic approach in the choice of a coordinate representation is to use an already existing standard for the data structure part of the **Coordinates** model. Prominent examples are [Hierarchical Data Format](#) (“HDF5”), [GeoJSON](#), or [Geography Markup Language](#); Sect. 3.3.1 gives a more detailed overview. Standards like HDF5 cover multiple levels of abstraction, i.e., the abstract data type, data structure *and* storage representations.

Important facts about coordinate representations are:

- a **Point** has no (need for a) coordinate representation; a unique symbolic **Semantic_ID** suffices.
- coordinates are needed in the *quantitative* representation of the **Position** of a **Point**. The coordinates represent the relative position of the **Point** to other geometric primitives.
- the same **Point** can be an argument in multiple **Position** data structures at the same time.
- the relation between a **Point** and the coordinate representation of a **Position**, is *not* that the **Point** **has-a** data structure of coordinates, but the other way around: the **Position** **has-a** relations to, both, the **Point** abstract data type, and the data structure of the numerical coordinate values.

(TODO: **Coordinates** representations of all other geometric entities and relations.)

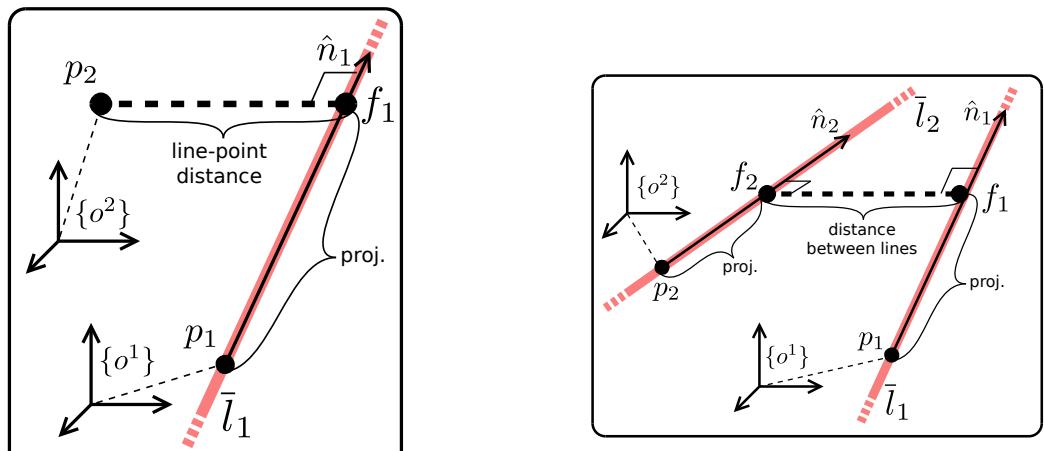


Figure 3.6: Graphical representations of the five possible relations between a point and a line 3.6a, and between two lines 3.6b.

3.3.1 Standards for coordinates

The relations between the data, its digital data representation model, and the meta-model used to define a specific data representation holds among the different alternatives, as de-

¹¹The [TopoJSON](#) standard uses this approach.

Table 3.4: 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	

Table 3.5: Summary of angular distances relations between versor and plane entities.

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

picted in Figure 3.7. The following Sections explain the popular **Coordinates** representation approaches in robotics.

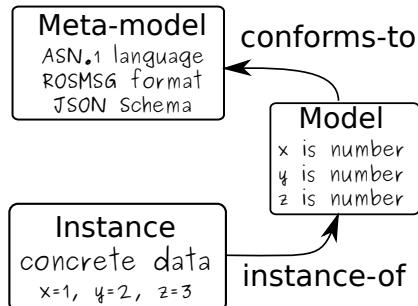


Figure 3.7: Digital representation models: a data structure in software is an **instance-of** a digital data representation model, which is a formal description that **conforms-to** a meta-model. A concrete example is shown in Fig. 12.3.

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

- **expressivity** of a meta-model to describe different properties over the data, including:
 - basic, built-in data types available;
 - possibility to indicate *constraints* on the data structure;
 - customisation over the memory model to store the data (instance of the model);
- **validation**: availability of a formal schema of the digital data model, meta-model and tools to validate both data instance and model schema;
- **extensibility**: the possibility to extend (by composition) the expressivity level of a digital data representation model;
- **language interoperability** (also called neutrality): the capability of a model of being language-independent; this requires a specific compiler to generate code-specific form of the digital data representation model;
- **self-describing**: optional capability of injecting the model in the data instance itself (metadata), or at least a reference to it; this enables reflection and run-time features.

Here are some widely supported international standards that satisfy these requirements:

- the **GeoPackage Encoding Standard**, of the Open Geospatial Consortium (OGC), to represent the **simple geometric entities**.

- OGC's [OGC Abstract Specification Topic 2: Referencing by coordinates](#), to represent coordinates and coordinate reference systems. This includes both earth-bound, [geodetic](#), reference systems, as local “flat earth” [engineering](#) reference systems.
- the [QUDT](#) stack of quantities, units, dimensions and types, [64].

3.3.2 Uncertainty in geometric entities and relations

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.

An uncertainty model represents the variability over the [Coordinates](#) representation (numerical, category, symbolic class,...) of a geometric entity or relation. It adds the extra semantics of a [Probability_distribution](#), with abstract data type, numerical [Coordinates](#) representation, and a representation of physical units and dimensions.

An example is to represent the uncertainty on the [Coordinates](#) of the [Position](#) of a [Point](#) in Euclidean space with a [Gaussian probability distribution](#) (also called “normal distribution”), which can be numerically represented by its mean vector and its covariance matrix, Fig. 3.8. 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.

There is little choice, in how to represent uncertainties on [Position](#). The situation is more complex for most compositions of two or more [Points](#), such as [Line_segments](#), [Lines](#), [areas](#), [Frames](#),... For example, representing the uncertainties on a [Line_segment](#) by means of Gaussian uncertainties on its two end points is *different* from representing it by means of a Gaussian uncertainty on its [start Point](#) together with an uncertain direction [Vector](#).

3.3.3 Covariance of a Frame has no meaning

An important **mathematical fact** is that SE(3) does not have a bi-invariant metric, [75, 74, 76]. In engineering terms this means that it makes no sense to talk about:

- the “distance” between two [Frames](#), or between two rigid bodies.
- “to add or subtract” two [Frames](#).
- to compute their “mean”.

Hence, also the “covariance matrix” on rigid body [Motion](#) or [Force](#) is void of any physical meaning. In practice this means that one *always* must introduce a a weighing factor to balance the physical dimensions of the translational and angular parts, and this weighing factor is *always arbitrary*.

A representation of the uncertainty on a [Frame](#), *with* consistent physical meaning, consists of *choosing three Points* on the [Frame](#) (e.g., the [Point](#) at the [Frame](#)'s origin and the [end](#)

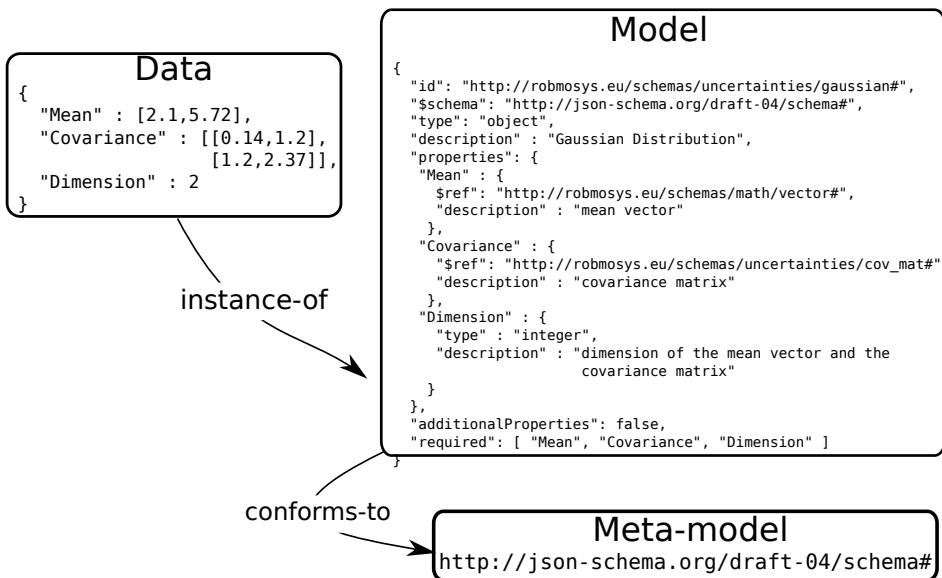


Figure 3.8: A valid data instance of a [JSON-Schema](#) model representing a Gaussian distribution. The schema is a composition of other schemas (for vectors and covariance matrices) and includes a few constraints on the data structure, such as the values required for the validation of the JSON document. Moreover, the schema [conforms-to](#) a specific meta-model of [JSON-Schema](#).

Points of two of the Frame’s three Cartesian unit Vectors), and adding a Position covariance matrix to the coordinate representations of all of them. This results in a *non-minimal* representation of the uncertainty; the constraints that have to be added reflect the facts that the two Vectors must have *unit length* and are *orthogonal*. Of course, this representation *also* introduces an arbitrary weighing between translation and orientation, albeit in an indirect way, via the choice of which points to select in the representation.

3.4 Compositions relations — Chains and maps

The previous Sections in this Chapter introduced the *simple* (“elementary”, or, “primitive”) geometric entities. Of course, these can be *composed* in multiple ways, and the following Sections introduce specific compositions that are highly relevant for the context of this document.

3.4.1 Geometric chain: constraint relations on geometric entities

Robotic applications must keep track of the relative positions and motions of several geometric entities, while:

- some of these parameters are *observed by sensors*. This introduces several types of *inaccuracy* on the types and values of the geometric entities.
- some of them are *coupled by knowledge* about their types and values. This introduces possible *ambiguities* between what is known and what is measured. For example, an autonomous car has very good information about what types of other traffic partici-

pants its sensors can detect in the environment, but it is not trivial to associate all measurement data to uniquely identified instances of these traffic participant types.

So, both cases bring in **constraint relations** between representations of the relative position and motion of geometrical entities. They allow the robot to work with “sufficiently full” information about the world, even when the robot can itself only observe a subset of all parameters in its *model* of the world. In the example from autonomous (or also human) driving: drivers have a mental model of a particular layout of traffic lights, traffic signs, and ground markings, and when they see one or more of those they can infer where to expect the others, even without spending perception efforts in that direction. Another example is a robot that estimates its position with respect to a cup on a table (Fig. 3.9) by a combination of (i) *observing* how it moves with respect to the table, (ii) *remembering* where the cup was on the table previously, and (iii) *knowing* how a cup’s motion is constrained by it being on top of a table.

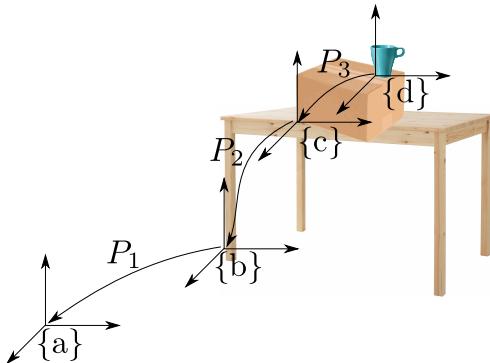


Figure 3.9: A **Geometric_chain** representing the relativeposes of a cup, a block, a table and the floor, where the geometrical constraints between them are of the type “being *on-top-of*”.

Figure 3.9 shows frames as representations of all geometric entities involved, but the constraint relations can also involve less than six-dimensional relations. For example, a two-dimensional constraint that represents that the block is resting on the table because one of its bottom points must lie in the top polygon of the table.

The simplest constraint graph is a **chain** of relations, which is also called a **geometric chain**: the chain representation comes with a specific *sequential* order, in which poses and their time derivatives are to be computed. For example, a cup can be placed on a block, that is itself lying on a table, that itself is standing on the floor, which itself is located in a particular room (Fig. 3.9). This situation represents the common case where the motions of several (rigid) bodies have relative motion constraints due to **natural** causes, such as gravity and the stiffness of the bodies’ materials.

3.4.2 Kinematic chain: engineered mechanical constraints

A [later chapter](#) is dedicated to the case of **engineered** motion constraints, in the form of “robots”, or *kinematic* chains, where the mechanical motion constraints are designed on purpose. Any kinematic chain is also a geometric chain, but not vice versa.

3.4.3 Map: collection of geometric entities

The sections above used *sets*, or [collections](#), of geometric entities, more in particular points and their compositions. The [polyhedron](#) is already a collection of collections, and this is the primary **mereological composition** of all geometric entities of this Chapter. Of course,

there exist many geometric entities and relations that do *not conform-to* the *Point-Polyline-Polygon* meta model. For example, spheres, ellipsoids and cylinders; or *clothoidal* and *spline* curves. Each of those geometric *types* can also be given meta models, and each *instance* of these types can be *connected* to an instance of the *Point-Polyline-Polygon* type.

That concept of **collection of collections** is a fundamental and generic *composition relation* for all sorts of meta models, especially but not exclusively, *geometric* meta models. This document uses **map** as the generic name for any such collection of collections of geometric instances, of whatever type. A semantically rich case is briefly introduced *below*, and a *later chapter* is dedicated to subject in more detail. It extends the mereological sets in a map, by adding **constraints** of different types: topological, geometric, dynamic, semantic, etc.

3.4.4 Semantic map: map with meaning

A semantic map is a map **composed** with **semantic tags** (or “annotations”, or “metadata”) for some of the map’s entities or relations. The meaning of the tags is relevant in the context of the semantic map *model*. The semantic map meta model is introduced in more detail in a *later Chapter*.

The simplest version of a semantic map uses tag names that only *hint* at application contexts, assuming that the meaning is “stored” implicitly in the background knowledge of humans familiar with the application. Examples of such semantic maps are *bus* or *metro* lines, the *public road map*, or a ski area, all have their own specific tags: a metro station does not appear in a ski resort, while, conversely, a *green ski trail* does not make sense in a metro system, although both types of world models make use of the same geometric primitives.

3.5 Differential geometry as meta meta model — Manifold, (co)tangent space, linear forms

The geometrical and mechanical entities and relations introduced in this document, have many similarities in other domains that are relevant for cyber-physical systems, such as electrical engineering and information theory. It is the role of **meta meta models** to make such similarities explicit, and **differential geometry** (e.g., [26]) is *the* mathematical theory offering a rich nomenclature and concepts to support that formalisation. Hence, it brings the *most abstract structure* in the models of this document, building upon a solid scientific foundation. Here is an overview of these meta meta level entities and relations to be used in models of the dynamics of mechanical systems:

- **manifold** $M(P, p)$ of **Positions** p of a **Point** P .
In robotics, a “point” in the manifold can also be a line, a plane, and often also a **Rigid_body**. These manifolds are *not metric space*, since “the” distance between two elements in those manifolds has no unambiguous meaning.
- **tangent space** $TM(P, p, v)$ of all **Velocities** v of the **Point** P at a given **Position** p in the manifold M .
- **second-order tangent space** $TTM(P, p, v, a)$ of all **Accelerations** a of a **Point** P at a **Position** p in the manifold M that has the **Velocity** v in the tangent space at that **Position**. There is a **constraint** between p , v and a , in that a is the acceleration of P at p when v is already the velocity of the *same Point* P at the *same Position* p . This constraint relation is sometimes referred to under the name of “jet”, [32].

- the **composition** of Motion relations has the following properties:
 - Displacement compositions form a *multiplicative group* in the manifold M . (So, not the more familiar *additive* one!)
 - Velocity compositions form an *additive group* on TM . It has no natural metric, but does have a natural origin, namely the “zero velocity”.
 - Acceleration composes *additively* on $TTM(P, p, v)$, and depends nonlinearly on the Position p and the Velocity v . So, it is not really a group operation, just a continuous **manifold**.

In this context, the concept of a **jet** fits well to the geometrical composition of position, velocity and acceleration of the *same point* or rigid body.

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

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

In mechanics, f is called a **Force**, [26, 53]. The pairing w of force and velocity is **Power**, that has the physical units of **Energy** per unit of time.

More in particular, **work**, **power** and “acceleration energy”¹², respectively, [118, 126, 168].

- **mass (inertia)** \mathcal{M} is a linear mapping from a **Velocity** v from TM onto an element \bar{p} from \overline{TM} , called the **Momentum** of the mass \mathcal{M} with the **Velocity** v :

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

The relation between momentum m and force f is **Newton's law**: force is the change of momentum over time. It is a **conserved quantity**.

The mass can also serve as a **metric** on the manifold of velocities, by combining the momentum map with the force-velocity pairing, applied to the *same* velocity. The result is indeed a **bilinear form** that maps that **Velocity** v from TM onto a real scalar w :

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

The resulting energy is the **Kinetic_energy** stored in the moving mass. It can serve as a **metric** on TM [26].

A mass \mathcal{M} can also serve as a metric on TTM , the space of **Accelerations**. In this case the scalar is sometimes given the name of *Zwang* [55], or *acceleration energy* [157].

- **damping** \mathcal{D} is a (not necessarily linear) mapping from a **Velocity** v from TM onto a linear form f_d (“friction force”) in \overline{TM} .

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

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

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

The resulting energy is **Heat** lost in friction; it has physical units of **Energy**. Damping can only serve as a metric on TM when it is linear.

¹²This is not a commonly used term in the English-language scientific literature. Gauss [55] introduced the concept with the German term “*Zwang*”.

- **elasticity** \mathcal{K} is a (not necessarily linear) mapping from a **Displament** d between two **Positions** on M onto linear form f_e (“elastic force”, “spring force”) in \overline{TM} .

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

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

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

The resulting energy is the **Potential_energy** stored in a spring. Because the limit of a **Displacement** (p, p') is a **Velocity**, \mathcal{K} can serve as a **metric** on TM .

- **impedance**: the combination of one or more of the relations **mass**, **damping**, and/or **stiffness**. In general, **impedance** relations are **non-linear**, but in many engineering contexts, their **linear** approximations suffice:

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

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

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

(TODO: much more explanation about what understanding of differential geometry is exactly needed to exploit it constructively in the design of robotics and cyber-physical systems.)

Chapter 4

Meta models for a kinematic chain and its instantaneous dynamics

The geometric chain model of the previous Chapter becomes a **kinematic chain** model, as soon as the chain's motion constraints are due to **mechanical joints** between (rigid) bodies in the chain. Such a joint is, from a modelling point of view, a special case of a time-dependent geometric constraint, in that (i) the constraint is *bidirectional* (one can “pull” and “push”) and (ii) its time-dependency can be represented with a finite number of “joint coordinates”. Revolute and prismatic joints are most common representatives, with just one joint coordinate.

A kinematic chain becomes a **dynamic chain**, as soon as the **mechanical behaviour** of **inertia**, **damping** and **elasticity** is composed onto the geometrical entities and constraint relations. Hence, one way to describe the motion of a robot is as the instantaneous **transformation of energy** between the robot's **motors** (in the so-called **joint space**) and its various **end effectors** (in the so-called **Cartesian space**). The *mechanical structure* of the robot's **kinematic chain** thereby acts as the physical **energy transformer**.

The contribution of this Chapter is to model (i) the **types** of motion constraints, (ii) the behavioural relations of *mechanical dynamics*, and (iii) the **abstract data types** that model their coordinates.

The next extensions to the geometric, kinematic and dynamic chain meta models make the link with the **task** meta model: (i) the forces in the actuators of the chain are task *resources*, (ii) the motions and forces of the chains' rigid bodies are task *capabilities*, and (iii) attachments points on the chains allow to add *task constraints* on forces and accelerations. Together, these semantic entities suffice *to specify* any type of **instantaneous motion** that is physically allowed by the kinematic chain.

4.1 Kinematic chain

This Section introduces the entities and relations that the meta models of the **Kinematic_chain** add to those of **geometric meta models**.

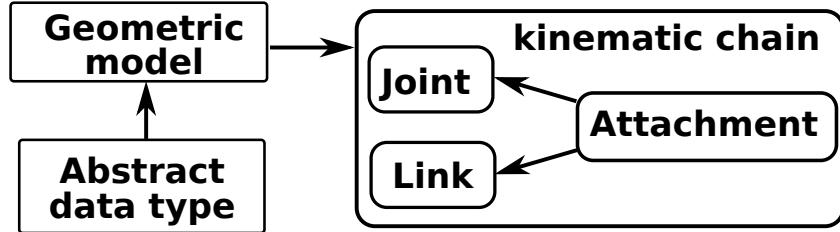


Figure 4.1: A kinematic chain model is a graph of Link and Joint entities, which are themselves defined as compositions of *geometric* entities (Sec. 3.2, Fig. 3.1) and their *abstract data type* representations. Each link can have one or more Attachments, to allow extensions by means of *model composition*, such as, geometric shape, dynamical properties, sensors, actuators, or handles for task specifications. This sketch only represents the *mereo-topological* parts of the Kinematic_chain meta model; an arrow represents a *is-part-of* relation, which is the *inverse* of the *has-a* relation.

4.1.1 Mereo-topology

The **mereo-topological** meta model of a Kinematic_chain has the following parts (Fig. 4.1):

- a collection of Links. Link is just another name for Rigid_body, in the semantic context of kinematic chains, that is, it refers to a Rigid_body that gets semantic tags to build a kinematic chain with.
- a collection of Attachments rigidly connected to each Link. An Attachment is a geometric entity (e.g., Point, Vector, or Frame) that serves as an argument in **composition relations** between Links and other relevant domain models. For example, at an Attachment, the geometrical properties of a Link can be connected to (models of) mechanical inertia, geometric shape, perceivable markers, or motors, sensors and tools.
- a collection of Joints, each being a special case of the mentioned composition relations, namely a **constraint relation** on the relative Motion between two Links. Major arguments in each such constraint relation are:
 - the Attachments on each Link. There should be one and only one such Attachment on one particular Link for one particular Joint.

Each Link can be constrained by more than one Joint; for example, most Links in robot are part of a serial connection of Joints.

One Joint can constrain more than two Links; for example, human joints like the shoulder have tendons that are attached to (and hence, constrain) multiple bones via multiple muscles.

 - the **type** of the motion constraint.

4.1.2 Types

The two common families of mechanical motion constraints are:

- **lower pair** motion constraints, with the one-dimensional Revolute (Fig. 4.2) and Prismatic joints as major representatives. These are **bi-directional** constraints (generating positive and negative constraint forces), and they have a **state** variable (“ q ”) whose value and its time derivatives are a one-on-one representation of the relative

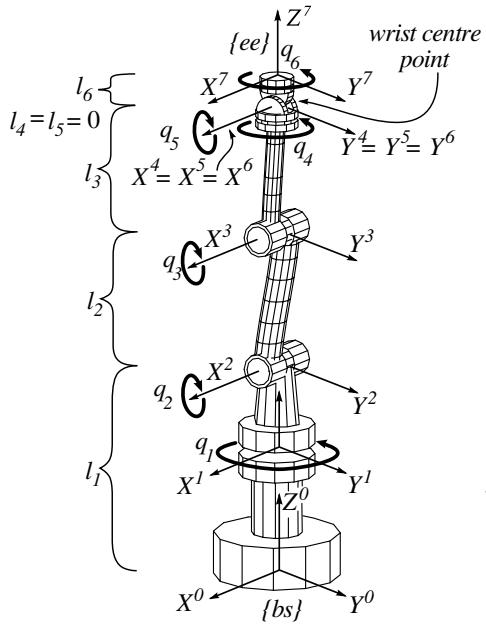


Figure 4.2: The most common kinematic design of *industrial manipulator arms*, using only revolute joints as bi-directional motion constraint between seven rigid bodies. This particular geometric configuration is a *kinematic family* parameterized by the symbols in the drawing. What is not symbolically represented on the drawing are the facts that (i) joints 2, 3 and 4 have parallel axes, (2) joints 4, 5 and 6 have intersecting axes, and (iii) joints 1 and 2 have orthogonal axes. All members of that family have the same mereo-topological model and the same geometrical parameters of the attachments of joints to links, but each member has different numerical values of these parameters.

Motion of the two constrained Links:

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

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

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

- **higher pair** motion constraints, with *wheels* and *cables* as major representatives. These constraints do not have position-level state variables, but only velocity-level variables. Hence, their name of **non-holonomic** motion constraint: the constraint can not be “integrated” to an equivalent constraint formulation with position-level variables. Higher pair constraints have some **uni-directional** constraint subspaces, in which constraint forces in only one direction are possible. For example, cables (Fig. 4.3), edge-surface contacts (Fig. 4.4) as in wheels, or vertex-surface contacts (Fig. 4.5).

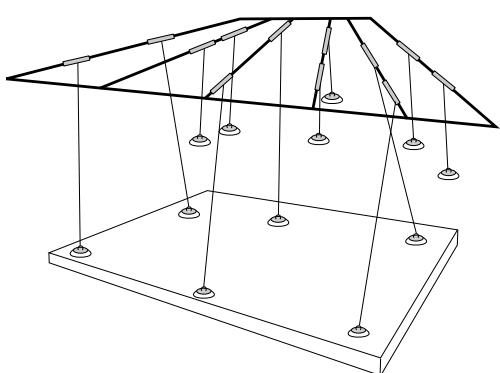


Figure 4.3: A *cable support* as uni-directional motion constraint between two rigid bodies. The mechanical framework on the top contains motors to control the length of the cables, as well as their position on the horizontal guides. The load is depicted at the bottom of the drawing, with unspecified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

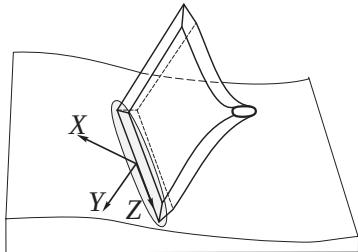


Figure 4.4: An *edge-surface* contact as unidirectional motion constraint between two rigid bodies. This model represents (part of) the geometrical abstraction of real-world motion constraints such as wheels or skates.

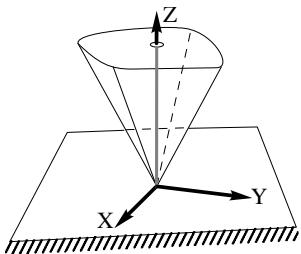


Figure 4.5: A *vertex-surface* contact as unidirectional motion constraint between two rigid bodies.

4.1.3 Coordinates and behavioural state of a chain's motion

Coordinate representations of motion constraints are needed to formalise the geometric **motion behaviour** of a kinematic chain, that is, the **constraints** that the chain adds to the relative *motion* (*position/pose*, *velocity/twist* and *acceleration*) of its individual *links*. Figure 4.2 shows an example of a geometrically parameterized model of a kinematic chain. The abstract data type for coordinates is a *collection* with the following entities:

- the geometrical dimensions of the **Link**.¹ This is done via the meta models of Sec. 3.2.
- the **Coordinates** of each **Attachment**. These **Coordinates** use the same *as_seen_by* reference frame as the **Coordinates** in which the **Link**'s geometrical dimensions are expressed.
- the *type* of **Joint** that is meant to be connected to each **Attachment**.
- the *model* of the **Joint**'s mathematical expression as a motion constraint. Some parameters in that model come from the **Attachments** on the **Links** involved in the motion constraint, some come from the type of the **Joint**, and still others represent the motion limits of the specific **Joint** instance.
- the **state** of a **Joint** is the subset of the parameters in a **Joint**'s mathematical model that represent the **motion behaviour** of the **Joint**. That is, the numerical values of the actual relative *Position*, *Velocity* and *Acceleration* of the **Links** connected by the **Joint**.

The *type* of the **Joint** must be represented formally, to allow software tools to check whether (i) both **Attachment** and **Joint** have the correct geometric primitives in order to be composed together, and (ii) the motion constraint model below links its state variables to appropriate geometric entities.

For some **Joint** types (such as `Revolute_joint` and `Prismatic_joint`), a **finite** set of **state** parameters can be given: one `joint_position` for the relative *Position*, one `joint_velocity` for the relative *Velocity*, and one `joint_acceleration` for the relative *Acceleration*. The span of these numerical values is called the **joint space** of that **Joint**, and that joint space model (always) has constraints on the allowable *range* of the **state** pa-

¹A **Link** is considered to be a **Rigid_body**. Flexible links are not treated in this document. However, the mereo-topological modeling still applies, since rigidity is a geometric concept.

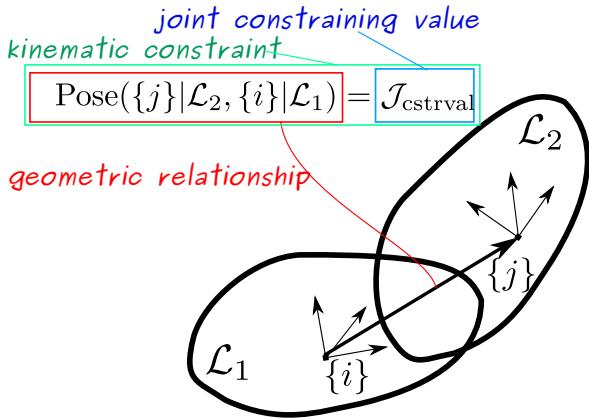


Figure 4.6: A generic model of the mathematical expression for the simplest `Kinematic_chain`, namely one with two `Links` and one `Joint`. The `Joint` has an unspecified type, but it has a `state` representation. That means that the relative `Pose` relation can be expressed as a *bijection* between relative `Poses` and one single joint position value J_{cstrval} . The frames $\{i\}$ and $\{j\}$ serve as *Attachments*, fixed to the `Links` L_1 and L_2 , respectively.

rameters. The set of spatial configurations of the `Links` that correspond to the joint space range is called the **Cartesian space** of that `Joint`.

Figure 4.6 depicts the generic mathematical model for a `Joint` motion constraint that has a `state` representation. The drawing sketches only the position level of the motion constraint, with a `joint_position` state value J_{cstrval} ; the straightforwardly extensions to the velocity and acceleration levels include the time derivatives of the `joint_position` state. The *dimensionality* of that state parameter J_{cstrval} is a number between “6” (rigidly connected) and “0” (not constrained at all). The coordinate representation of the constraining value must be compatible with the coordinate representation of the geometric relationship. As mentioned above, the `Revolute_joint` and the `Prismatic_joint` have constraint expression as a *function* of just one variable. More complex `Joints` (such as the higher-pairs) do not have clearly defined `Joint_positions`, e.g., the `knee` and `shoulder` joints in human bodies, or the `unilateral motion constraints` of contacts or cables.

Because a `Kinematic_chain` is just a collection of geometric primitives with semantic tags that have meaning in the `Kinematic_chain` context, the formal representation of a `Semantic_map` is the obvious model primitive to apply. Table 4.1 shows an abstract example of a mereo-topological `Kinematic_chain` model.

(TODO: examples of concrete models; e.g., for a robot arm of the 321 kinematic family. Make the model in Table 4.1 complete and consistent with the textual description.)

4.1.4 Geometrical operations — Forward, inverse and hybrid kinematics

Modelling the `Motion` behaviour of all `Links` in a `Kinematic_chain` requires no extra operations in addition to the geometrical ones already introduced in Sec. 3.2.8. At the level of abstraction of the whole `Kinematic_chain` however, extra operations are needed:

- *structural* operations of composition and decomposition of `Kinematic_chain` models. These operations are introduced in Sec. 4.4.
- operations on the `Coordinates` of the *dynamical behaviour* of `Kinematic_chains`. These operations are introduced in Sec. 4.2.3.
- operators that *transform* between joint space entities and Cartesian space entities. These are further described in the paragraphs below.

One identifies the following three categories of the latter transformation relations, where a

```

{ Semantic_map :
  { { ID : KC_ID_123XYZ },
    { MID : Kinematic_chain },
    { MMID : [ Geometry, Euclidean_space, Frame, QUDT ] }
    { links : [ Link_1_ID, ..., Link_2_ID ] },
    { attachments : [ { Link_1_ID : [ Link_1_Att_1_ID, Link_1_Att_2_ID ] },
                      ...,
                      { Link_6_ID : [ Link_6_Att_1_ID, Link_6_Att_2_ID ] },
                      { Link_7_ID : [ Link_7_Att_1_ID ] }
                    ] },
    { joints : [ { { ID : Joint_1_ID },
                  { MID : Revolute },
                  { first : Link_1_Att_2_ID },
                  { second : Link_2_Att_1_ID }
                },
                 ...,
                 { { ID : Joint_6_ID },
                   { MID : Revolute },
                   { first : Link_6_Att_2_ID },
                   { second : Link_7_Att_1_ID }
                 }
               ],
      ...
    }
  }
}

```

Table 4.1: Example of Coordinates model, in this case the Position of a Point.

later one encompasses all of the earlier ones:

- **Position** transformations: some joint space position values are given, as well as some Cartesian position values, and the operator takes these as inputs and generates as outputs all the non-specified entities. An additional outcome is a measure of the *consistency* of all specified inputs.
- **Position–Velocity** transformations: similarly of for the **Position**-only transformation, but now with inputs and outputs also at the velocity level of abstraction.
- **Position–Velocity–Acceleration** transformations: idem, now including also the acceleration level of abstraction.

Special cases of these transformations are when the inputs consist of only a complete set of joint space values, or a complete set of Cartesian values; the terminology is, respectively, **Forward_kinematics** and **Inverse_kinematics**, with a semantic tag indicating the relevant level of motion abstraction. The generic case is seldom used; this document names it the **Hybrid_kinematics**.

Formalization of these transformations is straightforward:

- add an **Attachment** for each of the input and output entities. Or reuse an already existing one; this holds in particular for the joint space specifications, because the model of a **Joint** already required the introduction of **Attachments**.
- add a **Specification** relation for each of the **input** entities, with as arguments:
 - the part of the **Attachment** that is used for the specification.

- the symbolic tag `input`.
- the value of the specified `input` entity, together with its physical `units`.
- add a `Specification` relation for each of the `output` entities, with as arguments:
 - the part of the `Attachment` that is used for the specification.
 - the symbolic tag `output`.
 - the symbolic `variable` of the desired `output` entity, together with its physical `units`.
- add two symbolic status variables, one `input_status` and one `output_status`, that encode, respectively, the desired and actual consistency of inputs and outputs.

It is indeed possible that the specified inputs can not give rise to a uniquely computable set of outputs (Sec. 4.1.5). The `input_status` symbol encodes the choices that can be made in computing the transformation; the `output_status` encodes the consistency that is realised by the transformation. These choices depend on the type of transformation, and on the numerical solver that is used to implement the transformation computations.

4.1.5 Inconsistency, redundancy, singularity

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

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

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

4.2 Rigid body dynamics — Composition of force and motion

This Section adds the **dynamics** behaviour relations for a rigid body, to its *kinematics* relations of position, displacement, velocity and acceleration. That is, the relations between **force** and **motion**, and their **compositions**.

4.2.1 Abstract data types and data structures for dynamics

The *abstract data type* for a **Force** is (formally but not semantically) equivalent to that of a **Twist**: it has `Coordinate` representations that look very similar, but only the semantic tags for the same parts are different. It is sometimes given the name of **Wrench**, and is composed of the two parts of a **screw**: a force `Vector` in 2D or 3D, and moment `Vector` in 1D or 3D, respectively.

A similar observation as with **Twists** was already made in the early 19th century, by the French mathematician Lous Poinsot (1777–1859), [112], who formulated the following theo-

rem: *any system of forces applied to a rigid body can be reduced to a single force, and a couple in a plane perpendicular to the force.* This is a formulation of the **screw axis** in disguise. Table 4.2 shows an example of a **Force** representation using the **screw_axis** model.

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

```
{
  Coordinate_Force_Frame_Array_Newton_Meter :
  {
    {
      relation: Force },
      {
        screw: [ force: f}, { moment: m} ] ],
      {
        on: { Frame: b} },
      {
        as_seen_by: { Frame: r} },
      {
        moment_action_point: as_seen_by_frame_origin },
      {
        coordinates: [ {f: [ {r.x: 1.2}, {r.y: -0,1}, {r.z: 3.2} ] },
                      {m: [ {r.x: 0.2}, {r.y: 0.8}, {r.z: -2.2} ] } ],
        units: newton, meter },
      {
        ID: Coordinatehd4if7 },
      {
        MID: Coordinate_Data_Structure },
      {
        MMID: { Coordinates, Euclidean_space,
                 Force, Frame, QUDT } }
    }
  }
}
```

Table 4.2: Example of a **Coordinates** model of the **Force** on a **Frame "b"** with respect to a **Frame "r"**, and using the **screw** representation.

4.2.2 Motion as result of constrained optimization — Gauss' Principle

The motion of a rigid body under the influence of forces (including gravity) is represented by the **Newton-Euler equations**. This is a **procedural** approach to describe motion; **declarative** alternatives exist too, via the general, differential-geometric concept of a **geodesic motion** on a manifold equipped with a *metric*, the latter being the mass distribution in the manifold; or via the dedicated methods of **d'Alembert** [33], **Lagrange** [85], **Hamilton** [66], **Gauss** [55], **Jourdain** [73] or **de Maupertuis** [36]. The declarative (or “**variational**” or “**constrained optimization**”) approaches are overkill for the *free* motion of a rigid body, but become practical whenever *constraints* occur on the free motion of the body, such as via mechanical contacts.

The most generic “*Principle of Least Constraint*” is that of Gauss: it provides an optimization relation between (i) the instantaneous acceleration of a non-constrained point or rigid body, (ii) the geometrical properties of a **motion constraint**, (iii) the instantaneous acceleration that satisfies that motion constraint, and (iv) the constraint force.

4.2.3 Coordinates for dynamics state

This Section links the **mechanical dynamics** meta model with **geometrical Coordinates** meta model. The composition represents the **physical phenomena** that:

- a **physical body** has **mass** and hence its (change in) motion is (*always*) related to an (inertial) force, via **Newton's laws**.

- a force can (*in some situations*) act on a body via a **spring** or a **damper** connected between a force and a body.
- the interaction between force and motion is regulated by **mechanical energy constraints**. More in particular the **conserved** energy components of **potential** and **kinetic** energy, and the **tribologic** energy (friction, lubrication and wear) that is **dissipated** into **thermal energy**.

For a robot, the forces have “external” sources (interaction forces or gravity) and “internal” sources (torques at the joints generated by actuators connected to the robot’s `Kinematic_chain`).

Coordinates representations for these forces are formally very similar to, but obviously semantically different from, the **Coordinates** representations for **Velocity** and **Twist**:

- **Cartesian Force**: these **Coordinates** have already been introduced in Sec. 4.2.1, as the **dual** concept of the **Cartesian Twist**.
- joint state space **torque** or **force**: many robot designs have motors that act on the same mechanical axis as the one that realises the 5D motion constraint, or equivalently, that drives the 1D motion freedom. So, the **Cartesian Force** that is generated by the motor to drive the relative **Motion** between the two **Links** constrained by the mechanical axis is faithfully represented by one *scalar* number, called (joint space) **torque** or **force**, for rotational or linear motors, respectively.

The formalizations of these **Coordinates** are analogous to those of **Motion**, just differing in the relevant semantic tags, see Table 3.2.

4.2.4 Coordinates and behavioural state for impedance

“**Impedance**” is the collective term for the mechanical mappings of mass, elasticity and damping. Their **models** are easily composed with the `Kinematic_chain` model:

- one **Attachment** is added to a **Link**, to represent the abstract data structure of the **Link**’s rigid body **mass**. That abstract data structure is *always* a **matrix** (“inertia tensor”, “mass matrix”, . . .), because the mapping between **Velocity** and **Force** is always *linear*, as is Newton’s Law that connects **Acceleration** to **Force**.
- one **Attachment** is added to each of two different **Links**, to serve as connection arguments in elasticity and damping relations between the two **Links**.
- the mapping relations of *elasticity* (between **Displacement** and **Force**) and of *damping* (between **Velocity** and **Force**) are **higher-order relations** that are typically *non-linear* functions of the **state** of the **Motion**. In practice, their **linear approximations** in the form of a **stiffness matrix** (and **Hooke’s Law**) or a **damping matrix** are popular.

The formal **Coordinates** representations of all of the above semantic primitives are straightforward, either as models of matrices, or as symbolic expressions of mapping functions.

(TODO: make formal models explicit, with examples.)

4.2.5 Geometrical operations revisited — The role of “virtual dynamics”

It seems logical to decouple the geometrical and dynamical meta models of kinematic chains, because the models of masses, springs and dampers (introduced in this Chapter) can be composed onto the geometric models (of Chap. 3) via **Attachment** primitives. But dynamical relations do occur already at the geometric level, albeit in disguise:

- whenever the specification of a (forward, inverse, or hybrid) kinematics transformation does not lead a unique solution, **redundancy** and **singularity** relations are to be introduced to determine the relative magnitude of the contribution of various components to the solution of the problem. Such trade-offs are typically realised by introducing “weighing” or “cost” functions with the components as arguments.
- the mathematical formulation of these weighing and cost functions reveals that they *must* have the physical dimensions of mass or elasticity.

Hence, the behavioural meta model for kinematic chains couples the geometrical and (mechanical) dynamical entities and relations together, all the time. The difference at the geometrical level of abstraction is that the impedances introduced by the solvers are artificial and arbitrarily chosen by the solver programmers. That is, they typically do not correspond at all to the real mechanical properties of the kinematic chain under consideration.

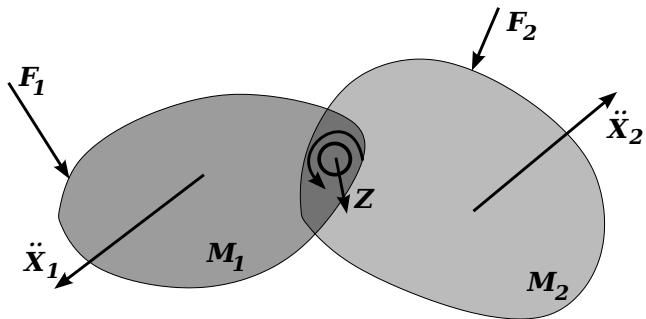


Figure 4.7: The entities involved in the dynamic behaviour of a (revolute) joint constraint. The spatial direction (\mathbf{Z}), force (\mathbf{F}) and acceleration ($\ddot{\mathbf{X}}$) have *six* degrees of freedom; the mass (\mathbf{M}) is a 6×6 relation between them.

4.2.6 Dynamic relations under a 5D motion constraint

The simplest possible **serial_composition** in a kinematic chain has *one* single bi-directional **Joint** between two **Links**, and that **Joint** has only *one* degree of motion *freedom*. In other words, the **Joint constrains** the two connected **Links** in five degrees of freedom, that is, the generated constraint forces span a five-dimensional space. Figure 4.7 sketches the components involved in the dynamical behavioural relations for such a **Joint**. The Figure shows an *unactuated* revolute joint, that is, there is no motor present on the mechanical joint axis. But adding a joint **torque** is obvious.

(TODO)) The modelling for other types of **Joints** uses similar entities, relations, and constraints.

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

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

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

\mathbf{M}_1^a the so-called *articulated body inertia* [49], i.e., the increased inertia of **Link 1** due to the fact that it is connected to **Link 2** through an “articulation”, that is, the revolute joint in this

case. \mathbf{M}_1^a of Link 1 is the sum of its own inertia \mathbf{M}_1 and the **projected** part \mathbf{P} of the inertia of the second Link. The term “projection” is adequate because the matrix \mathbf{P} satisfies the conditions for being a **projection matrix**, namely the **idempotence** relation $\mathbf{PP} = \mathbf{P}$. These algebraic properties are those of a weighted **Moore-Penrose pseudo-inverse**, [100].

4.2.7 Energy relations — Bond Graphs

The previous Sections describe knowledge relations about how force and motion are coupled via the “dynamics equations” of a **Point** or a **Rigid_body**. These relations represent the **instantaneous** coupling between force and motion, but any mechanically moving system has **state** entities that determine the force-motion couplings over longer periods of time. **Energy** is a major state entity, that appears at all time scales of **Motion**:

- **Acceleration_energy**: the map of mass and acceleration into a form **acceleration times mass times acceleration**. Hence, it is linear in the mass and quadratic in the acceleration. This form of “energy” can not be stored or dissipated, but is the **measure** that is optimized in Gauss’ principle, to find the instantaneous motion of a *constrained* point or rigid body.
- **Kinetic_energy**: the map of mass and velocity into a form **velocity times mass times velocity**. Hence, it is linear in the mass and quadratic in the velocity. This is one way of storing energy in a moving point or rigid body.
- **Potential_energy**: the map of mass and displacement/position into a form **g(mass, displacement)**, which is linear in the mass but *can* be nonlinear in the displacement. Two major examples in mechanics are: the potential energy of a mass in the gravity field, and the potential energy of a deformed spring. This is another way of storing energy in a moving point or rigid body.
- **Work**: the map of force and displacement into a form **force times displacement**. Hence, it is linear in both arguments. This mapping has the physical dimensions of energy, and can not be stored or dissipated, but is a **measure** to represent how much energy is needed to realise a particular non-instantaneous, finite displacement motion of a point or rigid body.
- **Damping/Friction**: the map of force and velocity into a form **h(force, velocity)**. It is linear in force but *can* be nonlinear in velocity. This mapping has the physical dimensions of energy, and represents energy dissipated in an instantaneous motion via mechanical friction or damping.

Physics has resulted in generic composition meta models that represent the (instantaneous) **exchange of energy** between rigid bodies (or non-mechanical equivalents in other **system dynamics** domains such as **electromagnetism**, **thermodynamics**, or **chemical engineering**). **Bond graph** theory is major example² of such a meta model, with its foundations built on the flows of energy via **Block-Port-Connector** mechanism.

²Unfortunately, its designers did not have the same *separation of concerns* principles in mind when formalizing the theory, which has led to too much coupling between the mathematical and the abstract data type levels of abstraction.

4.3 Kinematic chain dynamics — Instantaneous motion in a dynamic chain

Figure 4.8 (left) gives a schematic overview of all the variables needed to represent the **state** of a kinematic chain: position, velocity, acceleration and force, in each of the joints as well as of each of the links in the chain. Of course, all these variables are interconnected by the geometric and dynamics constraint relations introduced by the joints, gravity, and/or contacts with the environment. These constraints are sometimes called the **natural** constraints, because they are introduced by the physical world and are to be satisfied all the time. In addition to the natural constraints, robot controllers introduce **artificial** constraints, by means of artificially chosen forces and/or acceleration energy constraints at the joints and/or the links.

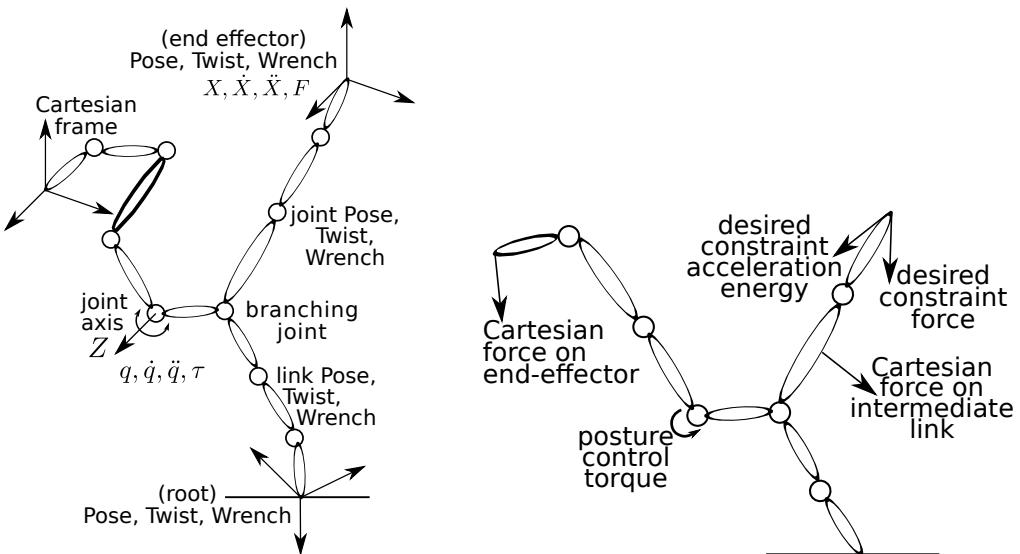


Figure 4.8: The entities in a geometric and mechano-dynamical model of a kinematic chain. On the left, all the type of variables that make up the *state* of the chain. On the right, the three possible types of *drivers* for the instantaneous motion (*change of state*) of a chain: (i) a torque generated by a motor at a **Joint**; (ii) a Cartesian force at a particular **Attachment** on a **Link** (with or without a mechanical contact at that **Attachment**); and (iii) a constraint on the allowable Cartesian acceleration energy at a particular **Attachment** on a **Link**.

4.3.1 Gauss' Principle of Least Constraint

Several centuries of research on the equations of motion of mechanical systems have led to the insights that *all* natural and artificial constraints on the **instantaneous motion** of (a connected or not-connected set of) rigid bodies can be formulated by one single theory, namely **Gauss' Principle of Least Constraint** [55, 113, 157, 158]. The consequence of this Principle is that there exist three, and only three, possible ways to change the **instantaneous motion state** of a kinematic chain (Fig. 4.8, right):

- joint space **forces** and/or **torques** on the **Links**, generated by the actuators at the **Joints**. These are sometimes called the “**posture control**” torques/forces, because a major use cases for them is to control the internal posture of the kinematic chain.

- **Cartesian Forces** on the **Links**, caused by “active” force sources, or resulting from “passive” contacts with the environment.
- **constraints on the Cartesian acceleration** of **Links**. The constraint function to optimize in this case is the **acceleration energy** \mathcal{Z} (from the word “Zwang” in the original German literature). \mathcal{Z} is the sum of all terms of the form **acceleration times mass times acceleration**, which is the “second-order” version of the similarly expressed *kinetic energy*.

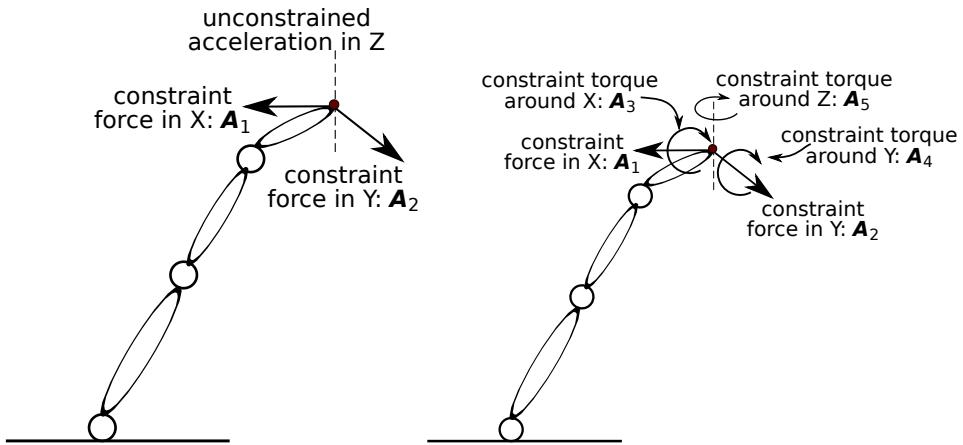


Figure 4.9: Two examples of artificial motion constraint specification via acceleration constraints. On the left: the last **Link** of the chain is constrained to have one of its **Points** move on a vertical line (irrespective of the orientation of the **Link** with respect to that line), by introducing two constraint forces that push the **Point** towards the line whenever it deviates from it. The right-hand specification adds constraint torques around all three orientation directions, in order that also the full orientation of the **Link** remains unchanged during the motion.

4.3.2 Specification of instantaneous motion

Because the physics of mechanical dynamics provides three complementary ways to make a kinematic chain move, it is appropriate to introduce a meta model to represent these physical facts as **instantaneous motion drivers** for robots. The formalization of Cartesian and joint space forces has already been introduced in earlier Sections, but a meta model for an acceleration constraint needs a bit more work. The solution uses the approach of [Lagrange multipliers](#):

- the constraint is modelled indirectly, by the introduction of **unit constraint forces** that counteract any acceleration in the constrained directions.

Figure 4.9 gives some examples of how Cartesian motion constraints on a **Link** can be represented as a **collection** of constraint **Forces** on the **Link**.

- the **acceleration_energy** that the constraint forces are allowed to generate, needs to be specified.

In the most common case, the allowed acceleration energy will be zero, representing the desire not to have any acceleration in a particular direction at all.

- the **magnitudes** of the constraint forces are then to be computed by solving the constrained dynamical equations.

Section 4.7 explains the algorithmic foundations for such solvers.

In practice, acceleration constraints are not popular to specify motions, and velocity constraints are used more often (and sometimes even position constraints). However, because only acceleration-level constraints have physical meaning, the velocity and position specifications constraints only make sense when composed with the acceleration-level physics via a *constraint controller* [13].

The formalisations of forces and accelerations have already been introduced in Chap. 3, which leaves only `acceleration_constraint` and its `acceleration_energy` as the new semantic relations to be modelled. The [mereo-topological](#) formalisation of a **instantaneous motion specification** of a `Kinematic_chain` then becomes a simple composition of the models of the `Kinematic_chain` with:

- the `Attachments` to connect motion drivers to.
- the motion driver specified in each `Attachment`.

Table 4.3 gives an example of such a mereo-topological specification model. The extension with [abstract data type](#) models is done in Sec. 4.7.

```
{ A_motion_specification :
{ { ID : Motion_Spec_ID_XX64Hy },
{ MID : [ HybridDynamicsMotionDrivers, KC_ID_123XYZ ] },
{ MMID : Kinematic_chain }
{ motion_driver : [ { { attachment : Link_1_Att_2_ID },
{ force : joint_torque_ID_34df } },
{ { attachment : Link_4_Att_1_ID },
{ acceleration_constraint :
{ dimension: 2 },
[ { unit_constraint_force :
{ ID : CF_1 },
{ attachment : Link_4_Att_1_ID.x },
},
{ unit_constraint_force :
{ ID : CF_2 },
{ attachment : Link_4_Att_1_ID.y },
},
],
],
},
],
}
}
```

Table 4.3: Example of `Motion_driver` model, with one joint torque and one acceleration constraint (in the *X* and *Y* directions of a `Frame` attached to a `Link`). The symbols used in the model refer to the `Kinematic_chain` with ID "KC_ID_123XYZ", Table 4.1.

4.3.3 Operations: forward, inverse and hybrid dynamics transformations

A specification of the drivers for an instantaneous motion of a `Kinematic_chain` is equivalent to specifying an **operation** on the chain, namely the transformation between the forces represented in the motion drivers and the forces one, and instantaneous (change in) motion of, each of the `Links` in the chain. For now, this transformation is still *implicit*, since one needs a [solver](#) to make the transformation explicit. A solver that accepts all forms of motion

drivers is sometimes called a **hybrid dynamics solver** [49]. When only joint torques are provided as inputs (together with the **Motion** state of the **Kinematic_chain**), the transformation is known under the name of **forward dynamics**. Similarly, in the **inverse dynamics** formulation, only the Cartesian forces are provided as inputs, together with the **Motion** state of the **Kinematic_chain**.

4.4 Composition and decomposition of kinematic chains

The **primitive** kinematic chain has one single joint connecting one attachment on each of two links. Compositions of this primitive result, in general, in a **graph** of interconnections, and this Section describes the entities and relations pertaining to such graphs of interconnected joints.

4.4.1 Composition — Serial, tree, graph

The following categories of **composition relations** are relevant:

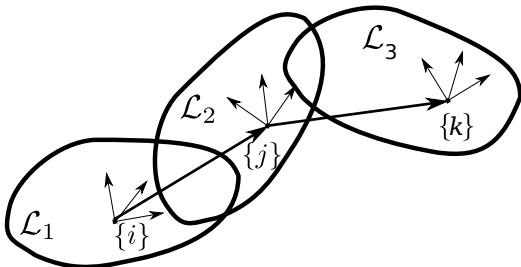


Figure 4.10: Serial composition of kinematic chains.

1. **serial_composition** (Fig. 4.10): either a *primitive Kinematic_chain*, or an already existing **serial_composition** to which one connects an extra **Link**, via a **Joint** to a **Link** in the **Kinematic_chain** that has already only *one* other **Joint**.

The result is a **strict order** of all links, attachments and joints in the composite kinematic chain. The first and the last Link in this list get the semantic tag **leaf_link**. The order does not have an absolute *sense*, in that there is no reason to call one of the **leaf_links** the “first” and the other the “last”. This sense *is* often added by models that compose the kinematic chain into a particular context, where that sense has a natural meaning. For example, it is natural to give one **leaf_link** in the industrial robot arm of Fig. 4.2 the tag “0” (the one that is the “base” of the robot, bolted to the ground), and count up from there till the other **leaf_link** (the “end-effector” of the robot, to which **tools** are connected).

Serial composition does not require new semantic operations: the composition relation of the *mass matrices* of two serially connected links has already been introduced in Sec. 4.2.6.

2. **branch_composition** (Fig. 4.11): the connection of one **Kinematic_chain** to another not yet connected **Kinematic_chain** via a **Joint** between (i) a **leaf_link** in the former **Kinematic_chain**, and (ii) a **non-leaf_link** in the latter **Kinematic_chain**.

Both Links loose their **leaf_link** tag; the latter Link gets the a **branch_link** tag instead (in case it does not already have that tag). Each of the connected **Kinematic_chains** is a **branch** of the composite **Kinematic_chain**.

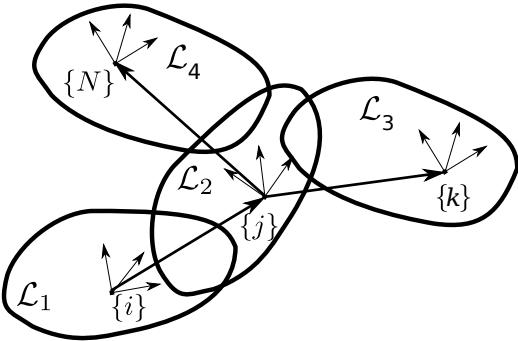


Figure 4.11: Branch composition of kinematic chains. The composite chain has three branches, and Link \mathcal{L}_2 is the `branch_link`.

Branch composition requires one new semantic operation, namely the composition the *mass matrix* of a branch `Link` and the mass matrices of two of its branches. This composition is linear, so the `Coordinates` of the mass matrices can be added, as soon as all their semantic tags are identical, that is, all coordinates use the same velocity reference point, the same as-seen-by reference, the same physical units, the same ordering of linear and angular parts.

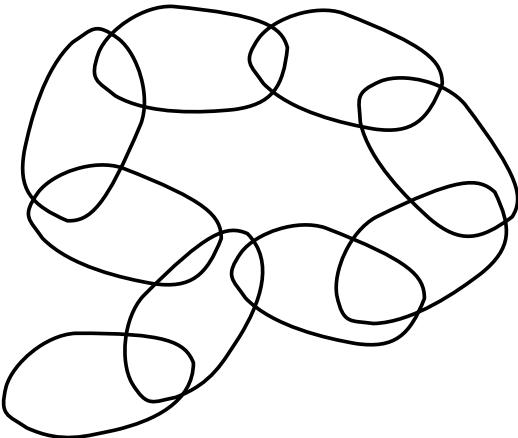


Figure 4.12: Loop composition of kinematic chains.

3. `loop_composition` (Fig. 4.12): this is a composition in which a `leaf_link` is connected via a `Joint` to a different `Link` in the `Kinematic_chain` to which the `leaf_link` belongs. The `leaf_link` loses its semantic tag. The other `Link` loses that tag too, in case it was one before the connection; if it had already two or more `Joints`, it gets the semantic tag of `branch_link`.

Closing a loop has a non-trivial impact on the operation of composing the mass matrices. There is not one single way of realising this operation, because an arbitrarily chosen “cut” of the loop into two branches is required (Sec. 4.4.2), after which the mass matrix of the cut link must be divided over the two branches, the serial composition of mass matrices is to be applied to each branch, and the branch composition of mass matrices has to be applied at the branch link of the cut loop.

Of course, higher-order composition relations are possible too; for example, loops within loops, like one finds in the `musculo-skeletal` structure of animals and humans.

4.4.2 Decomposition — Spanning tree

Many applications need to work with only those parts from a `Kinematic_chain` that are relevant to the applications' tasks; for example, only one arm of a `two-arm mobile manipulator` is needed to grasp an object. The semantic relation of the `Spanning_tree` supports such decomposition of a `Kinematic_chain` into sub-chains. It composes a model of a particular kinematic graph with another kinematic graph model (the so-called “`Spanning_tree`”) to **select a particular view** on the original graph, and has the following properties:

- *tree*: the `Spanning_tree` graph is composed of only serial chains connected at branching links, and each of them are sub-graphs of the original graph. For example, one way to map a kinematic graph onto a `Spanning_tree` is by cutting all of its loops.
- *semantic tags*: each serial sub-chain and each branch point get a `Semantic_ID` that identifies them as part of the original graph and as part of the `Spanning_tree` derived from it.

The model of the `Spanning_tree` stores the information about which cuts where made, and adds an extra `Attachment` at both ends of the cut. This allows to add **loop closure** relations, that mathematically represent the information about how to close the original loops again, or, equivalently, how the state of a `Spanning_tree` violates the motion constraint relations of the original kinematic graph.

One particular `Kinematic_chain` model can be mapped onto multiple `Spanning_tree` models at the same time. The concept of the `Spanning_tree` is relevant for all topological models, with serial, tree as well as graph topologies.

4.4.3 Policy: iteration via sweeps

A tree topology structure simplifies not only the mechanical construction of a robot, but also the computational solvers needed to make computations with its kinematic state. For example, a tree topology has only one single path between any two of its nodes, which simplifies computational `iterations` over lists of `Links`, `Joints` and `Attachments`. The `hybrid dynamics solver` makes extensive use of different sweeps.

The symbolic, model-centric equivalent of the computational iterator over a data structure is the `database cursor`, to make **graph traversals** over a property graph more effective when one has to solve a series of queries on the same graph, i.e., the same `Kinematic_chain`.

For the above-mentioned reasons, the majority of commercially used robot mechanisms *have* already a tree topology, and even the simplest form of a tree, a serial topology. For similar reasons, the majority of numerical solvers are designed to work on tree topologies only. That means that applications that need real graph topologies must compose their task specification with a `Spanning_tree` policy.

4.4.4 Policy: input-output causality assignment

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

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

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

4.5 Taxonomy of kinematic chain families

This Section introduces the top-levels of the `Kinematic_family`³ taxonomy. From a **modelling abstraction** point of view, it indeed makes 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.
- each category has a specific numerical solver, optimized for the particular geometric properties of the family.

Concretely speaking, all members in the same class of the `Kinematic_family` taxonomy share the same constraints: the number of joints, the type of the joints, the singular configurations, over- or under-actuation properties, and the abstract data types representing joint space and Cartesian space state.

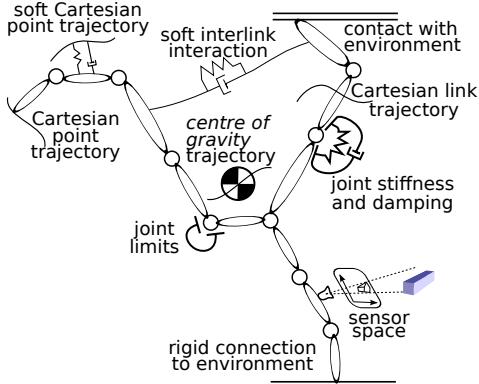


Figure 4.13: Sketch of the kinematic chain model of a dual-arm manipulator, with all(?) possible **motion** constraints on **links** and **joints**.

4.5.1 Serial chains

Serial kinematic chains, or “arms” (Fig. 4.13):

- `Kinematic_family::Serial-321`: traditional industrial robots
- `Kinematic_family::Serial-3-X`: many “cobots” like Universal Robot, or Mabi.
- `Kinematic_family::Serial-313`: KUKA iiwa, ABB YuMI,...
- `Kinematic_family::SCARA`
- `Kinematic_family::snake` or “elephant trunk” family.

4.5.2 Mobile platform chains

This family has three major sub-categories, depending on the **controllability** and **backdrivability** of the mobile robot platform:

- **under-actuated** (“**differential drive**”): Figure 4.14 shows the kinematic model of the mainstream in industrial **automated guided vehicles**:
 - two actuated wheels on the same axle.
 - one or more non-actuated and **rigid** caster wheels.

³This terminology is not standardized whatsoever.

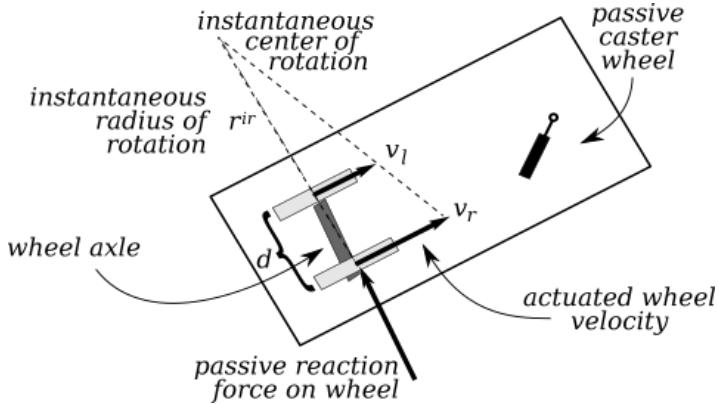


Figure 4.14: Kinematic chain model of a *differential drive* mobile platform: two actuated wheels connected to the same axle to control the platform's motion, and one or more non-actuated caster wheels to prevent it from tipping over. A force *along the axle* can not move the platform, and, dually, the wheel actuators cannot generate a velocity component in that same direction. This also implies that the velocity of the platform always has an *instantaneous* centre of rotation: the point from which the velocities of both wheels can (virtually) be generated by a pure rotation.

The resulting driving capabilities are very “car like”: one can control the *steering direction* and *longitudinal speed* independently of each other. (Although the saturation of the actuators will introduce a coupling sooner or later.)

- **holonomic (“fully backdrivable”):** Figure 4.15 shows the kinematic model of a mobile robot that can *be moved* in all three degrees of freedom of the plane:
 - two actuated wheels each on its own axle.
 - one or more non-actuated and caster wheels.

The passive mechanics of all wheels is of the [swivel](#) caster type. With this structure, external forces on the robot can *move* it in all three degrees of freedom of the plane, but the robot can itself only *control* two of these three degrees of freedom, *instantaneously*. (Because of the swivel caster suspension of the actuated wheels, the control problem becomes similar to that of a truck with a [trailer](#).)

Both wheels can also counteract each other, applying opposite forces on the platform; this configuration represents an *actuation singularity* in the control.

Figure 4.16 shows an alternative suspension for the two actuated wheels, connecting them on the same axle. This prevents the above-mentioned actuation singularity. The left-hand side of that Fig. 4.16 has [swivel](#) caster wheels; this construction keeps the full backdrivability of the generic kinematic configuration of Fig. 4.15. The right-hand side has two [rigid](#) caster wheels, sharing the same axle; this construction introduces the *instantaneous center of rotation* motion constraint again.

- **over-actuated (“redundant”):** Figure 4.17 shows a kinematic chain design with four of the above-introduced two-wheel drive units, each on a swivel caster. This results in five degrees of over-actuation for the three mechanically allowed motions in a plane. Hence, the control design needs to introduce a *force distribution* policy.

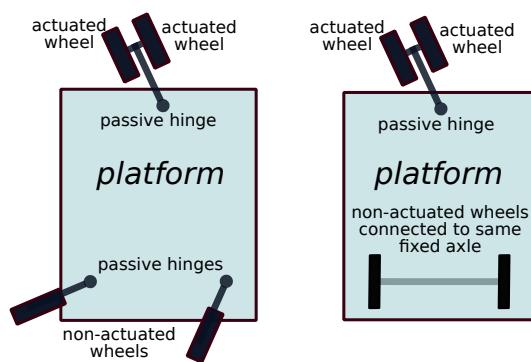
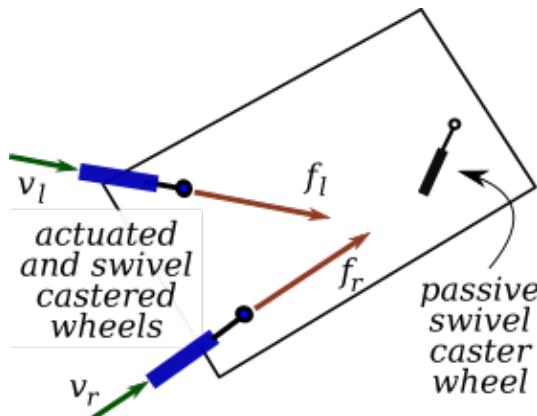


Figure 4.15: Sketch of the kinematic chain model of a mobile platform, with two actuated wheels that are each also a caster wheel. This gives three *passive* degrees of motion freedom, two of which can be *controlled*.

Figure 4.16: A kinematic chain like in Fig. 4.15 but with the two actuated wheels mounted on the same axle.

Left: the two passive wheels have independent pivot points, allowing motion in all three degrees of freedom of the plane.

Right: they share the same axle, hence one motion degree of motion is taken away. The remaining two degrees of freedom are fully controllable.

4.5.3 Parallel chains

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

(Fig. 4.18):

4.5.4 Multirotor chains

4.5.5 Hybrid chains

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

4.5.6 Cable-driven chains

Fig. 4.19,

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 **locomotion** towards targets over the earth, driven by wheels, legs, propellers (in air as well as in water),...

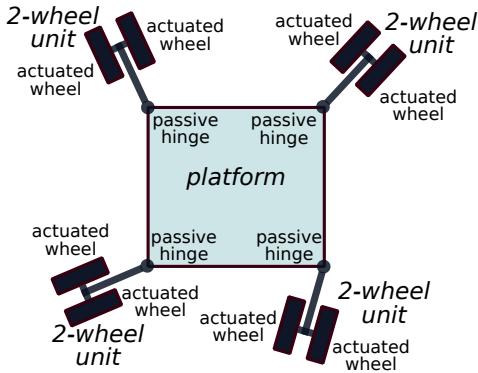


Figure 4.17: Sketch of the kinematic chain model of an over-actuated mobile robot: eight actuated wheels, connected in four “two-wheel drive” pairs, that are connected to a rigid platform via swivel caster joints. The design drivers behind this platform are (i) passive backdrivability in all configurations, and (ii) holonomic motion behaviour in all configurations and with very homogeneous actuation power in all directions.

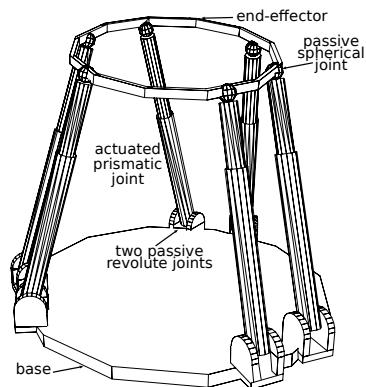


Figure 4.18: Sketch of the kinematic chain model of a parallel kinematic chain, with six legs each with six 1D joints.

- **balance:** the extra motion capabilities to do any combination of the above capabilities, at the same time, while keeping the robot in a desired range of configurations that the task context considers to be “in equilibrium”.
- **reach** (approach, accede, enter,...): the “arms” or “manipulator” navigate towards targets in local 6D space.
- **orient:** the “wrist” can direct grasped objects locally in various directions.
- **grasp** (encompass, lift, support, hold,...): the “gripper” or “hands” to manipulate objects by *force closure*, *form closure*, or *non-prehensile* holding.
- **touch:** to sense and manipulate by *form features* such as nails, finger tips, or whiskers.

The robotics domain has (informally) introduced *semantic tags* like “mobile manipulators”, “humanoids”, “gantries”, or “welders”, as specific compositions in particular application domains. How to specify the desired motion behaviour of a kinematic chain brings in the application’s task context, and is the subject of Chapter 6

4.7 Solver meta model for hybrid kinematics and dynamics

Section 4.2.3 introduced the entities and relations needed to specify the instantaneous motion of an ideal kinematic chain. This Section adds the information about the functions and abstract data types needed to create a **solver algorithm** that computes such an instantaneous motion from the model and its specification.

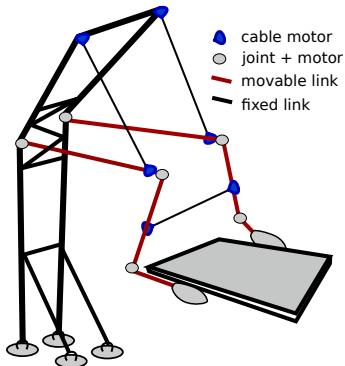
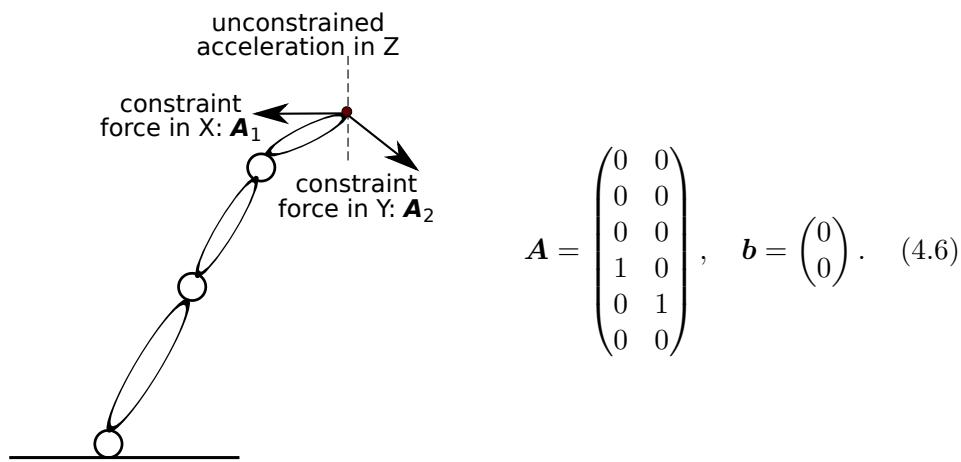


Figure 4.19: Sketch of the kinematic chain model of a cable-driven robot, with a “hybrid” topological structure.

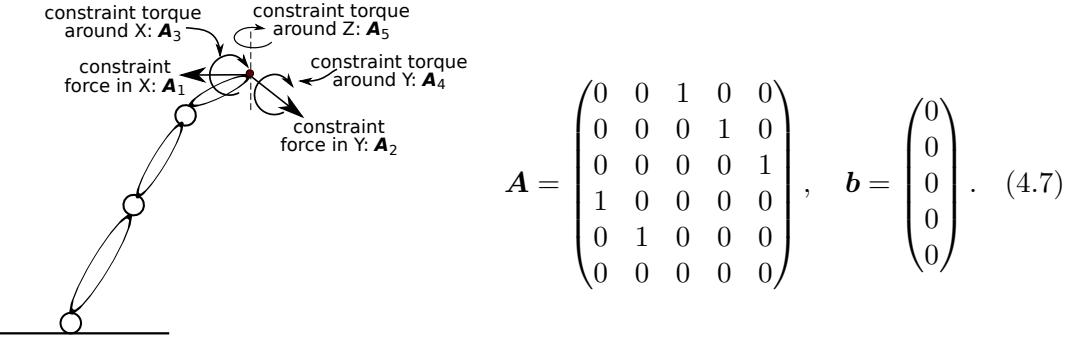
4.7.1 Mechanism-I: motion drivers

Section 4.3 introduced the mereo-topological models for the three different “motion drivers” with which to specify the instantaneous motion of a kinematic chain: joint torques, Cartesian forces, and Cartesian acceleration energy constraints. Acceleration energy is represented as a product of acceleration and inertia, whose formalisation with [abstract data types](#) gets the form $\ddot{\mathbf{X}}^T \mathbf{M} \ddot{\mathbf{X}}$; or, equivalently, of force and acceleration, of the form $\mathbf{F}^T \ddot{\mathbf{X}}$. In the Figures below, the matrix \mathbf{A} represents the matrix of the constraint force basis, that is, the set of “unit” versors along the spatial directions in which the acceleration constraints can generate constraint forces. For example, to constrain the motion of a reference *point* on the segment *partially* in the vertical direction, the constraint matrices can be chosen as follows:



The columns of \mathbf{A} are constraint forces in the horizontal X and Y directions, that must keep the acceleration in those directions to zero; the three rows of zeros on the top indicate the absence of acceleration constraints on the rotational degrees of freedom. The \mathbf{b} vector is used in a *motion task specification*, indicating that one wants zero acceleration energy to be generated/consumed in the constrained directions.

A second example is about moving the segment vertically *without allowing rotations*. Hence, the constraint matrix \mathbf{A} now represents five constraint forces:



That means that the constraining forces and moments are allowed to work in all directions, except the vertical Z direction.

Here is the “traditional” case of giving the segment a *desired acceleration energy* \mathbf{b}_d in full 6D:

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

4.7.2 Mechanism-II: procedural sweeps

All of the solver algorithms introduce an ordering in their computations, constrained by the structure of the kinematic chain. These control flow *schedules* are commonly referred to as “sweeps”; Fig. 4.20 depicts the three sweeps in the generic example of a tree-structured kinematic chain. Algorithm 1 summarizes the computations⁴ that provide the “inverse dynamics” of a serial robot, bolted to the ground, with N joints, one external force, and one acceleration constraint.

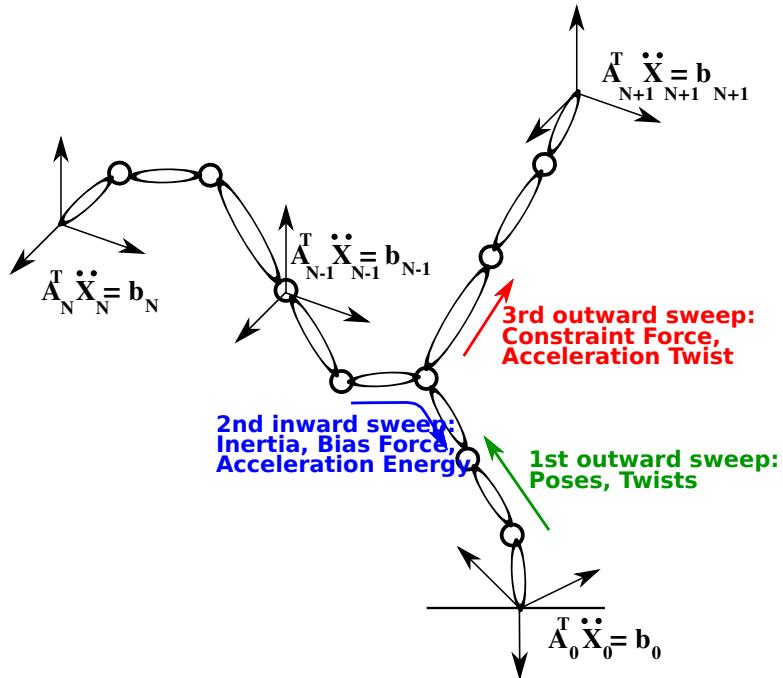


Figure 4.20: A hybrid dynamics solver uses the topological structure of the kinematic chain to schedule all the behavioural functions required to answer a particular query.

⁴The bookkeeping of the indices is not yet fully consistent in the presented pseudo-code...

Algorithm 1: Hybrid dynamics solver, according to Popov-Vereshchagin [158]

```

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

```

The *core properties* of such hybrid dynamics solvers are that:

- there are three **drivers** in the hybrid dynamics solver that can make the kinematic chain move; these are visible in the second last line in the solver Algorithm, via (i) *joint torques* $\boldsymbol{\tau}$, (ii) the *external forces* \mathbf{F} on links, and (iii) the Lagrange multipliers $\boldsymbol{\nu}$ generated by the *constraint acceleration energies* \mathbf{b} .
- when a *Cartesian force* is given as *input*, the chain's *motion* (first of all, acceleration, but via simple integration also velocity and position) is computed as an *output* of the solver. It is often appropriate to introduce a *monitor* in the third sweep, to check whether that computed motion is not violating any specified task constraints.
- when an *acceleration constraint* is given as *input*, the resulting *force* is computed as an *output*. As in the previous case, a *monitor* can be introduced in the third sweep to check whether this computed force is not violating any specified task constraints.
- the chain has a **dynamic singularity** if the articulated mass matrix projection goes to infinity, that is, one or more of the D scalars is close to zero.
- the τ_i are the (only) coupling with the physical actuator dynamics, where electrical power consumption or torque limits have to be specified, monitored, and/or optimized.

4.7.3 Policy: scheduling options in the third sweep

- when solving for the Lagrange multipliers ν , one can **weigh** the various acceleration constraint drivers.
- **prioritization** between the three types of drivers can be done by sequentially scheduling third sweeps for each of them separately: later third sweeps “win” over earlier ones.
- the effect of gravity can be computed separately from the effects of the motion drivers.
- joint torques caused by **joint friction and/or elasticity** can be added to the τ_i drivers, as *feedforward* functions.
- after the second sweep, one has **faktored** the whole kinematics and dynamics in pieces that can be (linearly) **composed** together in **various** ways in third sweeps.

For example, one could do a linear search to find the acceleration energy driver \mathbf{b} that gives a desired constraint force (via a re-solving of the linear system of equations connecting the \mathbf{b} to the Lagrange multipliers ν), or, conversely to find a Cartesian force \mathbf{F} that generates a joint torque with a desired magnitude and sign, or one that just does not saturate the joint actuators.

4.7.4 Policy: free-floating base

(TODO)

4.7.5 Policy: tasks in the mechanical domain

This Section explains how to configure the properties of the different sweeps in a hybrid dynamics solver to satisfy various types of mechanical Tasks, i.e., motions.

(BEGIN TODO:

- How can one let the robot do two (or more) tasks at the same time, and give relative priorities to the different tasks?

“Task” means: instantaneous motion, via force and or acceleration.

- How can the hybrid dynamics solver be used on a robot which does not have a torque control interface, but only a velocity control interface?

- What is the dynamic equivalent of a *kinematic singularity*?

Remember: at the velocity level, and for a serial robot, a singularity was defined as a joint space configuration in which the Jacobian matrix loses rank; or where some Cartesian forces require no joint torques to be supported; or where some joint space velocities result in no Cartesian velocity.

- Is it possible that acceleration constraints are not consistent with each other? How would one find out? What can one do about this situation?

- How can the algorithm be used to model the propulsion of a ship?

- How can one find out whether joint limits are violated? What can one do about this situation?
- Give an example where applying a force somewhere on the robot will not make the robot accelerate in that direction. What can you do to guarantee such minimum acceleration? How is this guarantee dependent on joint limits?
- How can one separate the parts of the joint torques that contribute to compensate gravity from those that work against the inertia of the robot to make it move?
- How can you find out to what extent two instantaneous motion specifications on the same kinematic chain are (in)consistent?
- How can you find out whether one could regenerate energy in a particular joint motor?

END TODO)

4.7.6 Policy: serial kinematic chain

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

4.7.7 Policy: branched kinematic chain

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

4.7.8 Policy: kinematic chain with a loop

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

Chapter 5

Meta models for dynamic world models, semantic maps and situational awareness

World models¹ are runtime representations of the current “status” of the world, including sensor measurements, uncertainties, multiple hypotheses about what objects are actually observed, etc. In other words, a world model is a **system component** that provides a **spatio-temporal context** to other components in the system. Of course, in this document’s **holonic** vision, that system can consist of multiple subsystems, each with its own world model(s).

The foundations of world models are the meta models of (i) **point and polygonal geometry**, (ii) **geometric chains**, and (iii) **kinematic chains**. This document focuses on *knowledge-based* systems, so a world model must also be able to store information about “the world around” a robot that that robot *can not deduce* from its own models, task specification, actions and sensors, but nevertheless *requires* in its decisions making. Hence, it must get that information from “somewhere else”. That information is not restricted to measurable geometric or physical properties of the real world, but also includes **symbols** that convey **meaning**, in the form of: intention, desire, context, dependency, importance, risk, utility, etc.

So, this document’s **paradigmatic** world modelling mechanisms are **semantic maps**, which are *collections of collections* of the above-mentioned world model foundations. A semantic map **structures** these collections, into **(spatio-temporal) situations**, each featuring several **action association hierarchies**. These association relations support three complementary aspects of *situations* [35]:

- **situation knowledge:** the entities, relations and their properties that define each situation.
- **situation awareness:** the perception needed to hypothesize about the possible situations that a system finds itself in.

¹The word “model” is somewhat misleading, because a world model is a fully instantiated *software* component, and not just a symbolic property graph of entities and relations. Hence, better names could be: *geospatial database*, *active map*, etc.

- **situation assessment:** the decision making needed to recognize which is the “most likely” of these hypotheses.

5.1 From simple geometrical entities to world models

Previous Chapters introduced the **simple** geometric entities, and two special *compositions*: **geometric chains** and **kinematic chains**. A world model composes the above, and all other application-specific geometric entities and relations, such as:

- *areas* with *semantic tag* points attached to them.
- *kinematic chains* connected to some of these attachments.
- attachments with other *smoother representations* of geometry, such as *spline control points* for **NURBS**, or **solid mechanics**.
- attachments with *discrete representations* of geometry, such as **point clouds** and **occupancy grids**.
- **geospatial coverage** of space-time varying phenomena.
- attachments with (references to) compositions of the *geometric operations* needed in the three top-level robotic “services”: *manipulation* by fingers in grippers, *reaching* by arms on torsos, and *navigation* by legs, wheels, wings or propellers on platforms.

Each of these entities is modelled as a **composition** onto a polygonal base model, via a “semantic tag” **attachment point**: each tag is *geometrically rigidly* constrained somewhere on the polygon, and it is a mandatory argument in the relation that represents the semantic properties of a world model entity.

In addition, the contextual information in semantic tags is often time-varying; for example, whenever a robot is within a particular area in the world, it can extend, update or replace the models it has “on board” with some of the world model parts, hence reducing the runtime role of the latter. Or rather, not the *role* of the world model, but the *place* in the system where *multiple versions* of world models are located in various system components, each with (possibly) different ownership, focus, resolution, accuracy, etc. Indeed, this document’s approach is to foresee a role for *a* world model in *any* component, and to avoid relying on *the* world model being available as one single “central” component for all components in the system, at all times.

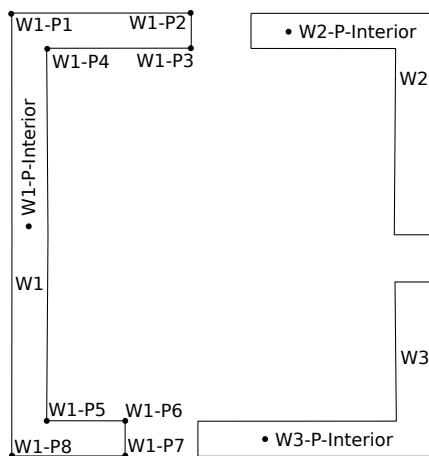


Figure 5.1: Map of the **walls** in a (part of) a building, as polygons. In order not to overload the drawing, the naming of the points is not complete, but just serves to give the gist of how a naming scheme could look like. The line segments are also not named explicitly.

This Chapter introduces mechanisms and policies *to create* world models, *to query* them,

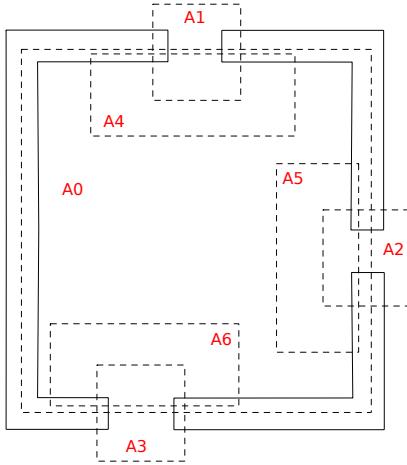


Figure 5.2: The Map of Fig. 5.1 is extended with some **areas** in the building. The areas are not physical, but serve as abstract conventions to allow humans and robots to indicate particular parts of the spatial domain.

to update them, and to connect them. All with an explicit goal- and task-directed focus, and structuring these mechanisms and policies independently of their abstraction, complexity and scale. The architectures of how and where a system introduces world models is the subject of later Chapters.

A **map**, is the simplest form of a world model, being just a *collection of collections* of simple geometric features. Figures 5.1–5.2 show some simple maps, with only instances of the point, polyline and polygon type. While the maps represent only *geometry*, the captions of the Figures already hint at an *interpretation* of this geometry in the context of an application. An important **topological** constraint that a semantic map adds to a map model is that of *layer*, *view*, and *difference*. These are all complementary *collections* of geometric primitives that “make sense” to be referred to together.

A **semantic map** is a map to which some **meaning** is added, in the form of **semantic tags**. For robotics systems, a major form of such meaning comes from the **Task meta model**. Figures 5.3–5.4 give examples, where the semantic tags on the **floor plan** get meaning in an application that help robots navigate through an indoor environment.

Standardized meta models for maps have matured in the **cartography** and **Geographical Information Systems** (GIS) ecosystems of [OpenStreetMap](#) and [OGC](#). For example:

- simple features standards, such as [GeoJSON](#) or [GeoPackage](#). They represent the Point-Polyline-Polygon entities and relations.
- web-based [maps](#), [features](#), and [coverages](#).
- [CityGML](#), to represent cities, and more in particular buildings: external and internal walls, floors, staircases, rooms, doors and windows.
- [HDF5](#) and [sqlite](#) as mature file and in-memory formats to store geometrical and coverage data.

See also the section on coordinate reference systems, earlier in this document.

5.2 Mechanism: Semantic map, Situation, Level of Detail

Since the end of the 1990s, *computer-aided maps*, or **cybercartography** [41, 121, 143, 144], has emerged as the “holonic” composition of modelling, systems theory and software engineering (“cybernetics”). In that broader context, the introduction of **semantic maps** [11, 14, 83, 103, 156, 170] is the evolution connected most to this document’s focus on *knowledge-driven*

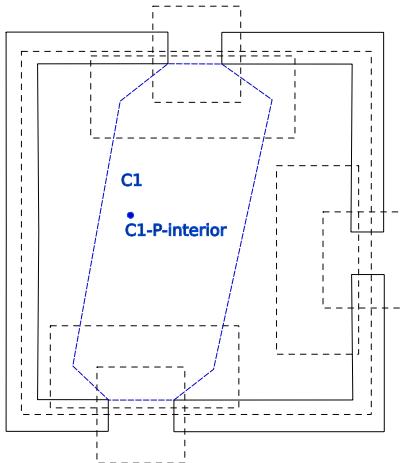


Figure 5.3: The Map of Fig. 5.2 is extended with one particular extra *area* that receives the semantic meaning of a **corridor**. That is, the semantic tag carries an **intention** of the purpose of that area.

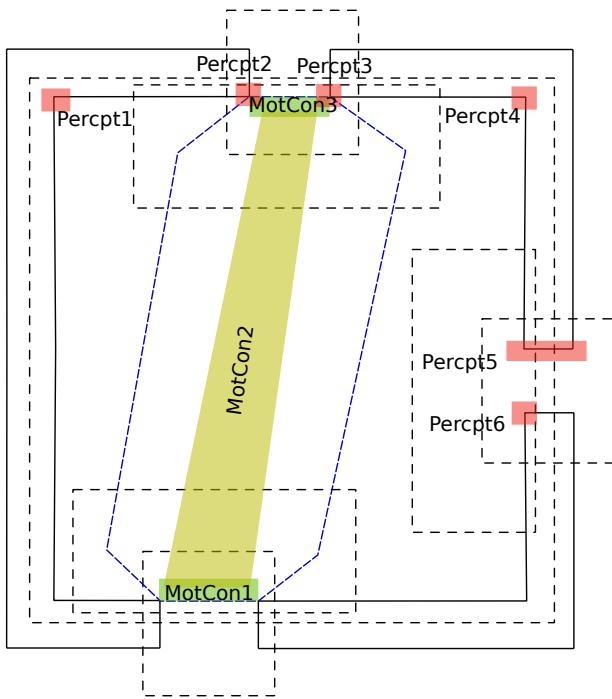


Figure 5.4: Map of the **motion** and **perception** areas that are relevant for a particular Task in the Map of Fig. 5.3. In order to follow that (virtual) “corridor”, the robot software should take into account the light-green “control” constraints, and the light-red areas that are best fits for the robot’s “perception” capabilities.

autonomous decision making systems: a semantic map does more than *passively representing* “the world”, because it becomes the *active mediator* [116] between spatio-temporal **situation** models and task-centric *decision making* models (i.e., for control, monitoring and map updating).

5.2.1 Semantic map

A **semantic map** is a map on which a set of the map’s **geometric areas** are linked together by a **property graph** of **symbolic relations**. Often, these relations also have a strong dependency on **time** and not just on location. Section 5.3.3 introduces this document’s major use case of semantic maps, namely that of a **situation**.

For all robotic applications, and for many cyber-physical applications (e.g., smart energy

grids), the *geometry* of the environment (location in the world, geometric shapes of objects and areas, etc.) plays a fundamental role. Hence, world modelling *starts* with

- modelling a **base map**, representing **locations** and **areas** in the world, and
- attaching **semantic labels** to them, so that
- **higher-order relations** can refer to them.

It is this set of areas, labels, and higher-order relations that turns a *map* into a *semantic map*. The higher-order relations are, for the largest part, coming from the **domain of discourse** of an application context, representing the meaning and purpose of these semantic areas, the system's actions in relation to the spatio-temporal context, and whatever else is relevant for the application. The purpose of making a particular application's *domain of discourse* explicit and formal is to allow *scientific methodology* into the *systems engineering* of that application. The methodology separates complementary design efforts in such a way that their outcomes can be (i) **validated** and (ii) **composed**:

- the **completeness** and **consistency** of the design and implementation of one particular semantic map can be discussed in detail by the system developers. Indeed, the domain **closes the world** of all “things” that the system has to be prepared for, and hence for the specifications of its software stack.

A world model is called **complete** for a particular application, if it provides **explanations** for **all** the **magic numbers** in that application's higher-order relations.

- the **appropriateness** of different semantic maps to the application context can be discussed in detail with the application's customers.
- the **decision making models** about when the system should select which semantic map can be made so explicit that also the system can reason about them, at runtime, while executing its tasks.

Any composition of models in one or more **levels**, gives rise to the need to give a world model context to that composition. **Geometry** is (again, paradigmatically) chosen as the **spatial “database”**, on top of which all other relations are attached. That database provides the information about where objects are with respect to each other in time and space, about which data was used to create that information, and about what data processing was used to generate the information. In other words, such a database serves as the foundation to anchor all other relations in a world model:

- *geometry–geometry*: the representations of the *shapes* of objects and robots, and how they shape the *motion constraint areas* between and around objects.
- *geometry–perception*: the representations of how geometric properties of objects are *detectable* in the raw data of various sensor types.
- *geometry–motion*: the representations of how task representations use geometric properties of objects as *targets* of the robots' motions.
- *geometry–task*: the representations of the geometric aspects of the *actual, desired, hypothesized,...* states of the world, depending on the task requirements.
- *geometry–information*: in some areas in the world, the robots are advised to contact on-line sources of information about how they can adapt their behaviour to the local context.
- *geometry–situation*: the representations of which geometric properties of objects are to be present together, to form the *hypothesis* that the robot finds itself in a particular situation.

The main features that a “cybernetic” **implementation** of a semantic map must support, are:

- **responsiveness to queries**, both for fast manipulation of the map data (insertion/update) and data requests (inference). The real-time boundaries are given by (i) the application’s tasks, and (ii) the latency requirements from the motion and perception stacks.
- **composability: views and levels of abstraction**. Each view must be defined by a proper meta-model supported by the world model implementation. That is, each view can provide a application-specific Domain Specific Language (DSL) to support queries.
- **distributed**: in all but the simplest applications, a world model must be distributed and deployed in different components, or even different hardware platforms. Each “local” world model must not strive to be the complete answer to the system’s tasks, but focus on the concrete data exploited by the *local* motion and perception stacks, with special attention to the present limitations in each single sub-system, and on knowing which other sub-systems to contact to get missing information.
- **history and logs**: a world model should not contain only *instantaneous* information, but also previous states of the situation and of the robot’s actions. This enables *a posteriori* task execution analysis, as well as the application of (online and offline) learning algorithms to improve future executions.
- **framework-agnostic**: the world model should be independent of a software framework implementing the semantic maps, or of any communication middleware used to interact with them. Instead, it should be trivial to integrate world model and semantic map functionalities within any type of component.

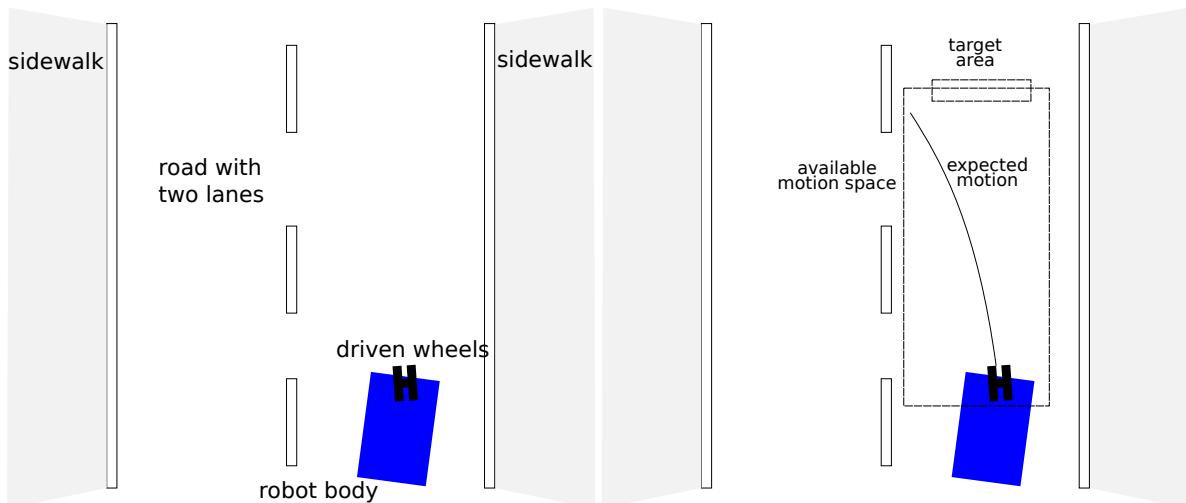


Figure 5.5: An example of a **situation** in traffic.

Left: the **semantic areas** of the situation: the “robot car” is in the right lane on a bi-directional two-lane road, bordered by sidewalks, and with painted traffic markers indicating the areas where driving is allowed.

Right: the **actions** of the situation: the robot drives towards a target area, positioned inside the right driving lane semantic area, and its driving action must remain inside an “available motion space” semantic area, which is itself fully inside the area where driving is allowed, and which in turn is itself in the right lane semantic area.

5.2.2 Situation

This document defines a **situation**² as the combination of (Fig. 5.5):

- a **semantic map** of symbolically labeled **areas**.
For example, different lanes of a traffic layout map. The markings painted on the road, and the traffic signs put next to or above the road, are symbols that all users of the traffic system must recognize, and interpret as constraints and/or suggestions for their driving behaviour.
- a list of **actions** that a system *can* realise in, and/or between, these areas.
An action transforms a particular (symbolically represented) “pre” situation to one or more particular (symbolically represented) “post” situations. More specifically, the transformation goes from one particular “occupancy” of a set of the semantic areas to another “occupancy”. Often, it makes sense to represent such a transformation as a continuous (“per”) relation (“constraint”, “invariant”) on the world model.
For example, **driving** maneuvers on a two-lane road: cruising, crossing intersections, overtaking, parking, making U-turns, etc.
- a set of **task-centered relations** that connect:
 - labels in these **areas**,
 - to a particular **behaviour** (perception, motion, decision making, information communication, . . .),
 - in the application’s **context** (available resources, desired execution quality, meaning, purpose, intention, risk, utility, . . .).

For example, the limitations that the traffic markers and signs impose on how a cruising action on a two-lane road can be executed.

5.2.3 Level of Detail, Level of Development (LOD)

(TODO: Cartographic generalisation; figure in [18]; [63]. **Level of Development** in **Building Information Modelling**: shows evolution over time, of progress in **built environment** projects. The evolution in the Level of Development of a building is very correlated with an increase of the Level of Detail: operations on a building start with the ground floor and foundations, later operations add the shell (“casco”), and more details are added in the finishing operations.)

5.3 Spatial association hierarchies

A world model does not only contain information about *where* things are in space, but obviously **spatial locations** (“*areas*”) of all resources and tasks play a **first-class citizenship** role as the basis for all knowledge representations in world models, [14, 16, 82, 170]. One important type of such knowledge are the **hierarchies of association relations** between an *area* in the world and the *behaviour* for which a system has the *capability* to realise that behaviour in that area. One important feature of these knowledge relations is that they often have a

²This approach extends the concept of **affordances**: for humans, an environment (model) comes “naturally” with the “plans” of what those humans can do with the objects in that environment, that is, we don’t have to think about those plans consciously, and we all share (almost) the same perception and control capabilities. Of course, for use in robotics, plans, perception and control are not built-in, and must be formalized in computer-processable form. This remains a huge technical challenge, even after many decades of research.

strong *hierarchical structure*, with as foundation the location geometry, or “*situation*”. Here is a list of the *generic* types of such **association hierarchy** structures:

- the association hierarchy in [location](#).
- the association hierarchy in [spatio-temporal](#) phenomena.
- the association hierarchy in [perception](#).
- the association hierarchy in [control](#).
- the association hierarchy in [navigation](#).
- the association hierarchy in decision making for [execution systems](#).
- the association hierarchy in decision making for [skills](#).
- the association hierarchy in communication.

One and the same *area* can be connected to (or, “semantically tagged with”) more than one *situation* model, and with more than one *association hierarchy* model. Some of these association hierarchies are introduced below; others are provided with much more detail in the later architectural chapters.

5.3.1 Location association hierarchies

Location in space is a primary feature of all engineered systems, for the simple reason that **hardware resources take up space**. This Section introduces a “natural” **spatial containment hierarchy**, that is, hardware resources at different [levels of abstraction](#), and that are *nested inside each other*. Here are some containment hierarchies, for the contexts of “navigation”, “traffic”, and “manufacturing”:

Navigation	Traffic	Manufacturing , according to the “physical model” in the ISA95 standard
<ul style="list-style-type: none"> • world. • continent. • country. • city. • neighbourhood. • building. • floor. • ward. • corridor. • room. • furniture. • rack. • drawer. 	<ul style="list-style-type: none"> • zone. • road. • lane. • marking. 	<ul style="list-style-type: none"> • enterprise. • site. • area. • process cell. • unit. • equipment module. • control module.

The nesting structures above are not absolute or standardized, and every application domains comes with its own version. But all domains *do have* such a location hierarchy somewhere in their designs, often implicitly, to introduce structure in, both, decision making (in configuration and control), and interaction channels (of material, energy and information). The *structure* of the hierarchy is to be complemented with the *relations* that **associate behaviour** on two or more levels of the structural hierarchy.

A primary example of such behaviour association is the **decision making** that takes place at all levels in the hierarchy, in an application-dependent way, to let such decisions influence each other. The **interaction behaviour** typically takes place much more [hierarchically](#), for example:

- primary, secondary and tertiary roads.
- railway, airplane and waterway networks.

Although the interactions themselves can be heterarchical, the *configuration* of the interaction behaviours *is* a decision making activity, which does exploit the hierarchy.

5.3.2 Spatio-temporal association hierarchies — Coverage

All robotics and cyber-physical applications feature phenomena that vary in time and space. The digital representation of the corresponding data is sometimes referred to as [coverage](#), [12]. Coverages are inherently multi-dimensional, and their dimensionality is structured as follows:

- 1D sensor timeseries ($x(t)$).
- 2D images ($I(x, y)$).
- 3D image timeseries ($I(x, y; t)$).
- 3D surface information ($S(x, y, z)$).
- 4D surface timeseries, such as density, temperature, intensity,... in every point on a surface ($T(x, y, z; t)$).
- 5D surface tangent vector timeseries, such as velocities in every point on a surface ($V(x, y, z; v_x, v_y)$).
- 6D spatial vector timeseries, such as forces in every point on a surface ($F(x, y, z; f_x, f_y, f_z)$).
- ND coverages, adding images, scans and their [tomograph](#) sections, communication traces,... to a (selection of) points in space-time.

In addition to the *continuous* spatial and temporal dimensions in a coverage, *semantic labels* can be attached too; for example, surface areas with a particular meaning.

5.3.3 Association hierarchy of actions in a situation

The model of a situation, together with the action models defined inside, **decouple** the models of all the TSR programming components (task specification, plan, perception, control, monitoring): none of them has to know about the others, because they influence each other only indirectly via the action descriptions in a situation. For example, perceiving the presence of traffic markings or traffic signs is necessary to identify the current driving situation, but the execution of the driving behaviour is specified and controlled via the “state” of the current situation, as it is available in the world model, irrespective of via which perception activities that information got there.

The relations between situation and **plan** are the first ones to make explicit, because they give *purpose* to the other relations.

A situation also allows **to decouple** different *hierarchical levels* in action models, by means of a containment tree of semantic areas on the map, so-called *tiles*: a “tile” on a higher level of the situation association hierarchy gives a [TSR context](#) to tiles for task specification, control, perception and monitoring on a lower level of the hierarchy. Such a context can take different forms: different levels of detail, different focus, different degrees of freedom in the configuration space, etc.

5.3.4 Association hierarchy in navigation

This Section introduces a natural hierarchy in *navigation*, as a special case of [actions in situations](#). In navigation applications, a robot moves itself around in the environment, so one of the main decisions it has to make is about its desired awareness about how much

the current situation will evolve in time and space. This document suggests this awareness is modelled via a nested set of **horizons** on the robots' maps, that provide scope to their various decision making responsibilities:

- **map** horizon: the areas whose [semantic contents](#) are *taken into account* to influence a robot's decisions. In other words, making it explicit which parts of the map are *not* relevant.

- **task** horizon: the areas that provide parameters (objective functions or constraints, and their [magic numbers](#)) to the robot's [task specification](#).

The task horizon has a *hard dependency* on the map horizon, because one *can not* specify a task without the semantic tags that connect it to the map.

- **perception** horizon: the areas in which the situations must be perceived that the [skills](#) need to realise a robot's task.

The perception horizon has a *soft dependency* on the task horizon, because it is *not strictly needed* to search, find and interpret *all* environment features that a task specification relies on.

- **control** horizon: the area over which an "[MPC](#)" problem is solved to make a robot move.

The control horizon has a *hard dependency* on the perception horizon, because it is *necessary* for a controller to estimate its "control error" on the basis of the sensor information.

- **avoidance** horizon: the area in which the control part of a robot's task is replaced by collision avoidance. That is, the progress of the nominal task of the robot temporarily gets a lower priority than the avoidance of a collision.

The avoidance horizon has a *soft dependency* on the control horizon, because it is *only necessary* when collision becomes likely.

- **emergency** horizon: the area in which the task is replaced by a graceful degradation, that is, to reduce the impact of an inevitable collision to a minimum.

The emergency horizon has a *hard dependency* on the avoidance horizon, because it becomes *necessary* when collision is inevitable.

5.3.5 Association hierarchy in manipulation

(TODO:)

5.4 Horizontal and vertical composition of situations

(TODO: horizontal = enlarge horizon in time and space; vertical = increase resolution and detail in time and space. Stability horizon of a skill in a situation: horizon in time and space over which a skill can *predict* (and maybe also *guarantee*): progress; quality of service; risk and cost;...)

5.4.1 Mini-meso-macro motion situations

(TODO: robot hardware: hand on arm, arm on platform. Macro motion: platform reaches target area; meso motion: arm reaches finer-grained target area; mini motion: hand touches environment with desired task tolerance. Coupling between levels must be such that *freeze-*

and-resume execution is possible; requires appropriately modelled task specification and task execution monitoring semantics (is each “part” still in its “good enough” performance area?).)

5.5 World model activities

5.5.1 Blackboard

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

5.5.2 Semantic database

For any somewhat realistic application, the **tworld model** contains several hundreds to several thousands of entities, relations and constraints, with an order of magnitude more *parameters*. It is appropriate to call this composition a “**semantic database**” (Sec. 1.5); the added value with respect to a normal database is:

- the parameters in the data are linked together with relations that have semantic meaning, with several **higher-order** relations and constraints, via the many interconnected levels of abstraction and types of information.
- **queries** on the database can use semantic terms reflecting the **intention**, **causality** or **dependency** that holds between query arguments; of course, *if* these higher-order relations have been added to the models. Hence, one can get explanations about *why* the query yields the results it does, or about *the context* in which the answer holds.
- the provided knowledge links can be exploited in the computation of the **query answer**, because **graph traversal** becomes possible, instead of the more general *graph matching*. The latter is the default “solver” for relational database, while the former becomes more and more mainstream in graph databases.

In the context of robotics systems, this means that the **coupling** between **control** and **perception** can be adapted to the **context** provided by the *Task-Situation-Resource* paradigm:

- the expected capabilities.
- the available resources.
- the past, actual, and expected state of the environment.

For example, a robot controller can switch its perception algorithms depending on the knowledge it has about which features in the environment fit best to, both, the available sensors and sensor processing software, and the required feedback and monitoring in the motion control loops. More concretely, when driving through a corridor in a hospital or office building, the robot can actively search for the *semantic tags* that have been put in the building with the explicit purpose of guiding its users towards the various destinations. A similar situation holds

for outdoor navigation (car and truck driving, or plane and helicopter take-off and landing) where traffic signs and signalling are put up to increase the **freedom from choice** of the traffic participants, and hence their “cognitive load” connected to taking part in the traffic.

5.6 Stable states in robotic applications

(TODO: robustness against disturbances; preemption points in execution, to persist, revive and hot-swap an application; abstraction to start/stop planning, monitoring and scheduling, to reduce computational horizon (hence also computational complexity); semantic label in situation and task specification, linked to cause of preemption, set of disturbances, observability, etc.)

5.7 Best practices

- don’t put entities on the map that have no meaning for the robot: if it can’t do anything useful with it, it has to avoid colliding with it based on the “raw” sensor data “map”. Some unknown blobs can have multiple hypotheses, each with modelled actions, e.g., some “strange” sensor measurement could be a human, or ghost reflections, or..., but in each of these three hypotheses the robot has a plan (i) how to find out more information about the hypothesis, and (ii) how to make action decisions.
- **position** is not a *property* of an object on a map, but a property of a relation between object and map. The same object can have multiple relative position relations at the same time, with different other entities on the map.
- decide about *how to forget* map data, at any level of resolution and in every context: memory policy is a higher-order relation.
- A world model should not only focus on the *instantaneous* state of the world, but should also support *memory* and *predictions*, with various degrees of resolution and horizon in both time and space. For example, a navigation application needs to know more details about the immediate past, future, space and objects than about the more distant ones, and the platform should support continuously sliding “windows” over time, space and resolution.
- The data stored in the world model should not cover all the aspects of the environment as a whole, but only those application-dependent information needed for the correct execution of the tasks. Or, expressed differently, many applications are expected to require several world models to run asynchronously, because the different sub-systems in the application have different needs. Hence, all modelling efforts should take into account the technological challenges of **distributed synchronisation** and **(eventual) consistency**.
- world models must support multiple **view layers**, each offering a different aspect, interpretation and/or level of abstraction, of the same entity. Examples of the latter are structured along the same lines already used to represent levels of abstraction in the motion and perception modelling:
 - **mereology:** describes the **has-a** (and **collections**) relationships between entities (and the world model instance itself). All entities represented in a world model must be uniquely identified by an **uid** (by means of a IRI, URI or other forms), which is sufficient to describe the *existence* of a certain entity instance in the world model, in a symbolic form;

- **topology:** evinces the *connectivity* of different elements in the scene, e.g., connected rooms in an map environment. Topology can also be expressed symbolically by means of *spatial* descriptors, such as `left-of`, `near-to`, `on-top-of`, `inside` to, etc;
 - **geometry:** including affine geometry (e.g, point, line) metric geometry (e.g., displacement, distances, dimensions). Geometric entities can be expressed by means of the geometric semantics described in Section 3.2, whether they are physical objects or virtual artifacts that describes a *feature* used in the task specification;
 - **dynamics:** this view is offered to those motion and perception stack algorithms that are capable of dealing with kinematic motion models (differential geometry) and their interaction with the environment (in terms of “physical forces”, deformations, etc).
- the concrete properties of an entity modeled in the world model can be available in different phases of an application, sometimes only after that a certain perception task has been performed. For instance, a task specification can indicate the presence of a ball, but its location and pose ball can still be unknown (or are not relevant to the task, yet): in this case, the entity “ball” is modeled only at the “symbolic level” (mereology). Additional topological information can be used as “first guess” of a perception algorithm: if “the ball is on top of a table” and the geometric properties of the table are known (i.e., geometry view), the location of the ball can be roughly determined by inference reasoning. In a later phase, the visual tracking/servoing task of the ball is possible, updating the geometry view of the ball in the world model;
- in many applications, different views and levels of abstraction are needed to reduce the search space. It is the case for touch-based active sensing tasks, which combines the “act” (motion stack) with the purpose of localising (perception stack) a physical object (and its properties). For instance, in [147] an object is localised by decoupling the active sensing activity in different sub-tasks, each one working on a different configuration space of the world model;
- two or more robotic systems can share the same environment, sometimes accomplishing cooperative tasks, sometimes acting as competitors on sharing the environment resources. In this scenario, the world model is *distributed* and plays a crucial role in the success of the application, since i) the models of the robotic systems (the “agents”) must be included in the world model; ii) the negotiation of the resources happens on the basis of the information available in the world model, which must be (“sufficiently”) coherent for all the “agents”; iii) each agent may have different limitations in terms of available computational, memory and communication resources; iv) each agent may have a different *local* world model, managing only relevant information for its own tasks, without being aware of the needs of the overall application; v) the previous challenge is even larger in case of communication issues, e.g., in rescue scenarios where the communication interferences can be very high.

Chapter 6

Meta models for tasks

Users of engineered systems—and of robotic systems in particular—*expect* these systems to have the **capability** to execute the quasi infinite amount of **tasks**¹ in the users' **application domain**. And these users also *know* that these tasks *can* be executed with:

- the **resources** available in the system.
- the **quality** these resources can provide.
- the **environments** in which the system and its task-related **objects** live.
- the **knowledge** about the **affordances** [57] of these environments and objects. That is, the specific ways “things” can—or should—be manipulated and perceived.

This Chapter introduces the **mechanisms and policies** to turn those expectations and knowledge into **formal specifications**, in such a way that later Chapters can realise the next step:

- to create **system architectures** that actually **execute** those specifications,
- on the **resources** that a system has at its disposal in the **real-world**,
- while monitoring the **progress** and **quality** of that execution in the context of the specified expectations.

The **mereo-topological** part of such task specifications come in the form of three paradigms:

- the **Task-Situation-Resource** meta model represents the **knowledge** to couple all perception, control and decision making activities within one particular **situation** that occurs in the world.
- the **perception for task control** meta model represents the **hierarchy** in the **composition** of “control”: the proprio-ceptive, extero-ceptive, and cartho-ceptive control levels within *one single* robot system, and the **shared** perception and control models for *multi-robot* and human-robot interactions.
- the **hybrid constrained optimization** meta model represents an approach to specify a task in a way that can be coupled to software implementations for its execution, with as major design driver the (configurably) high levels of reactivity and adaptability in the robots' behaviours to what is happening in the world around the robot.

¹Alternative names for *task* are: *order*, *programme*, *work*, *job*, *schedule*,...

6.1 Task models — What, why, and how well?

System customers only want to pay for the visible *economic added value* of tasks, such as the assembly and delivery of a number of products. But the *execution* of tasks requires many “infrastructural” tasks and resources that are invisible to the customers, such as logistics, hardware and software for perception and control, condition monitoring, quality inspection, redundancy, software for world model updating, etc. These typically only add to the *cost* of the system, and not to its added value. This document focuses not on the mentioned *customers* of systems, but on their *developers*. And for them, this document’s approach about how *to specify and execute* tasks is the same for added value tasks as well as for infrastructural tasks, and *cost, utility* and *risk* are all part of task models.

6.1.1 Task model association hierarchy

This document introduces task models for **application developers**,² as the **association** of several “levels of abstraction”:

- task **specifications**, representing **what** has to be done, using the **capabilities** that are present in the system **to act** on the **affordances** of the objects in the system’s environment.
- task **quality monitors**, representing **how well** the **quality and progress** during an execution of a task must be to continue its execution.
- task **situation** models, representing the world around the system for which the above-mentioned task models have been designed. That is, the situation offers all the “attachment points” for the above-mentioned task specifications, and quality and progress monitors.
- task **coordination** models, representing the **built-in reactions** to switch between different parts of a task model, whenever the expected progress and quality are (not) realised during the execution.
- task **drivers**, representing **why** a task has to be done. That is, one makes explicit what the **intention** is to change (a specific part of) the **actual state** of the **world** (physical and/or informational) into a **desired state** of the world.

The simplest possible task model in an application is just a *series of specifications*. When multiple “levels” are present, the order in which they are described above is only a *partial order*. And at each level, other “association hierarchies” can (but need not) be introduced, such as: the **continuous, discrete and symbolic** hierarchy; the **proprio-ceptive, extero-ceptive** and **carto-ceptive** hierarchy; the **motion “stack”** hierarchy; etc. The resulting **task model** is a knowledge graph with many **higher-order relations**.

System developers decide for their application how much of the above-mentioned associations they need to represent the *what?*, the *why?*, and the *how well?*, of the tasks they need their application to perform. The selection they make is then the information underlying the task **execution** (or “**skill**”) architectures they create. Each skill offers *one particular way*

²Task models for *customers* can be built on top of the task models in this Chapter, as a user-centric specialisation, or **domain-specific model** (DSL), of the developers task model: it presents only those parts of the task models that customers need to configure, and using terms and conventions that are “common knowledge” in the particular application domain. It is well possible that two DSLs for two different application domains conform to exactly the same developer task models, but use very different terminology. For example, “robotic **shuttles**” driving around a parking lot, or “**Autonomous Mobile Robots**” driving in a warehouse, can both have the same technology under the hood.

of *how* a task must be realised, and pertaining to more or less knowledge to be used in (and by...) the robot controller.

The resulting task model *knowledge graph* is built from models that, qualitatively speaking, can not be made any smaller without loosing the meaning of “*being a description of a task*”. What that granularity is exactly, depends on the application requirements. What *is* independent of the domain is that all task models have some form of “[world model situation](#)” inside. Each world model must always be larger than the world state that is strictly relevant for the task, as that broader horizon determines (implicitly) the set of **disturbances** that the **execution** of the task could be made robust against. In other words, it determines how much of the “[open world](#)” the robot can deal with.

6.1.2 Specification model

The specification model represents **what** a robot has to do, with respect to perception, motion control, world model updating, and monitoring.

For example, a robot car must do a [parallel parking](#) task, in an appropriately sized *parking slot* at the right-hand side of the road. Therefore, it follows a sequence of back-and-forth manoeuvres, with the steering wheel turned to its two extremes, and with focusing its laser scanner processing to those areas that are relevant for each manoeuvre.

6.1.3 Quality of progress model

This model represents **how well** the **progress** during an execution of a task must be to continue its execution.

For example, the speed of the “transversal” motion of the robot into the parking spot should be between 5% and 10% of its nominal “longitudinal” speed of the robot in road in front of the parking spot.

6.1.4 Situation model

A situation model offers:

- the “attachment points” for the other models.
- the relations that link perception to control.
- the relations that link various task specifications to the “higher-order” intention (or “eventual target”) of the task.

For example, the specification of one task in the task model is determined by what is needed in a task that comes later in the envisaged task execution. Such as in the case of parking a car, where the initial position task of the car with respect to the parking space is not making any physical “progress” towards that parking spot, on the contrary. Another major example is **active sensing**: the robot performs a motion task only because that brings its sensors in a better position to collect information, without already making the robot progress physically towards its task target situation.

6.1.5 Coordination model

The simplest coordination reaction is *to report* the quality that is realised during task execution. An intermediate level of reaction is to report also *why* the quality or intention of the task were not reached. Somewhat more advanced reactions to the latter situations include

the switch to a *Plan B* task execution. While the most advanced reactions not only have *Plan C* and *Plan D*, but also the **resilience** capability to prevent the failure of *Plan A* the next time around.

6.1.6 Driver model

The driver model represents **why** a task has to be done, that is, what the intention are of reaching the above-mentioned “what”.

For example, the parking is required because the robot car must *pick up* or deliver goods from a *docking station*; or because it has *no other tasks* left anymore; or because it must *move out of the way* for other robots passing by.

6.1.7 Capability model

This document follows the hypothesis that the design of the *hardware resources*—in the form of *material* as well as *energy*—in engineered systems (manufacturing and logistics equipment and warehouses, or energy creation, distribution and storage) is done *together* with the *tasks* that the system is expected to execute, in such a way that **vertical composition** of tasks is optimised for the storage and transport of **stable** hardware sub-products.

The task models *depend* on the models of **motion**, **perception**, and **control** that are introduced in other Chapters. But also the *inverse dependency* holds: the requirements of the robot’s task influence the configurations of how the robot decides about how to move, to perceive and to control.

A useful “side effect” of the various task model representations in this Chapter is that a particular composition of models from several of the above-mentioned paradigms can be used as the **capabilities model** of a robotic system. Such a capabilities model serves as a **data sheet** of a robotic system’s behaviour, in the form of the set of tasks that the system can be expected to execute. Ideally speaking, such a data sheet can be used as a **contract** between providers and customers of the robotic system.

The “*bad news*” in this Chapter is that application developers *must* be aware of the large complexity of task specification for robotic systems, before they can provide capabilities models that are rich enough to serve as **vendor-neutral** data sheets that can service in commercial markets. Indeed, they must be sufficiently knowledgeable about all details from the actuator dynamics to the environment safety concerns. The “*good news*” is that the structured approach advocated in this Chapter supports **composability** of task specifications **out of the box**, so that developers need only *be aware* of the complexity (in order not to compromise *compositionality*), without (necessarily) having *to master* all details of that complexity.

6.2 Hierarchies in resources imply hierarchies in task models

The complexity of robot task models grows together with the complexity of (i) the task **plans**, and (ii) the capabilities of the **resources** that the system has available to realise those plans. Many of the **complications** are caused by the fact that any somewhat realistic robotic system, and hence the corresponding resources and plans, require multiple **levels** of abstraction to be composed together.³ The multi-level modelling approach presented in

³Much of the failure of the **Winter of AI** was due to the fact that then-available software support, like **Prolog** could not deal very well with such hierarchical structures.

this Chapter [conforms](#) to this document's underlying [knowledge pyramid](#) paradigm, to bring **structure** in the inevitable [complications](#) of multi-level system models, and, hence, system designs.

The paragraphs below explain which hierarchical structures in the models of the resources that are involved in a task immediately lead to [conforming structures](#) in task models. These hierarchical resource structures in themselves are *modelled* with *symbolic*, *discrete* and *continuous* parts in all three of the [representation association hierarchy](#) structure. This document advocates to have the latter hierarchy present in *all* models.

6.2.1 Spatial hierarchies — Location

The origin of this type of *levels of abstraction* is that of the [location association hierarchies](#) in world models. More concretely, spatial *areas* are almost always an inherent part of robotic systems, because:

- the system's *resources* are located somewhere, and take up space, which brings in **constraints** for the task model.
- *moving* towards or in areas is an important part of the **added value** of the system. Of course, the [kinematic chain](#) "areas" of the robot systems themselves introduce important spatial **constraints**.

The granularity and abstraction of spatial areas that are used in task models must match those of the other parts in those models. The hierarchical structures of many of these other parts are introduced in the Sections below.

6.2.2 Execution resource hierarchies — Detail

Industrial and agricultural manufacturing is realised by means of lots of different [machines](#), many of them being [computer-controlled devices](#). Human [operators](#) often also contribute to the *execution* of the manufacturing tasks. Not only do all these execution resources take up space and are located somewhere, there is always a (application-specific) [association hierarchy](#) involved too. That hierarchy is most often a strict *containment*: each resource at one level of execution representation, contains a number of other resources, and the structure helps to choose a *level of detail* to talk about the execution that has to be executed, or that is currently under execution. Commonly speaking, a *factory* contains a number of *manufacturing cells*, each of which consists of a number of *machines*, and the latter then contains a number of *tools, mechanisms, sensors* and computers. The paragraphs below introduce the terminology of one particular standardization context, the [International Society of Automation](#), that has given names to the hierarchy, and the entities involved.

[Manufacturing Execution Systems](#) (MES) are a commonly appearing category of applications in a manufacturing context. They often appear together with [Enterprise Resource Planning](#) (ERP) systems, in the higher levels of industry standards such as [ISA-95](#). In this document, the following three levels in that standard are relevant, not in the least *because* (i) the levels are introduced based on *spatial proximity* and *spatial occupancy* (with a strong *situation* structure), and (ii) the performance of their interaction behaviours resulting from the spatial proximity:

- **Level 1: Intelligent devices.** These are responsible for the **continuous space and time execution** of sensing and manipulation in **material processes**, very close to the

hardware interfaces to those processes. That material continuity *implies* a **high** spatial proximity.

A typical example in manufacturing is the so-called **PLC-based** control.

The corresponding *task models* are **motion oriented**: they specify *how* the controlled machines must move.

- **Level 2: Control systems.** These applications add an **discrete execution sequencing** level of perception, motion and task control, monitoring, and supervision to the *intelligent devices* above. Such decisions can be made on the basis of **information exchange** about the material processes, hence, **medium spatial promimity** can be tolerated, as long as communication bandwidth and latency can be *guaranteed*.

A typical example in manufacturing is a so-called **SCADA** system (*Supervisory Control And Data Acquisition*).

The corresponding *task models* are **(discrete) operation oriented**: they specify *what* operations the controlled machines must execute; and each of these operations is most often the composition of several motions and other discrete decision making actions.

- **Level 3: Manufacturing operations systems.** These applications add the **execution of the management** of all *material and information flows* between several *Level 2* systems. This level of decision making has **low spatial proximity** requirements.

Typical examples in manufacturing are the so-called **ERP** (*Enterprise resource planning*) and **MES** (*Manufacturing execution system*).

Core parts of these systems are *orders* of products to be produced on manufacturing lines (which is the responsibility of a *MES* application), implying streams of *parts* between warehouses and the production lines, together with their bookkeeping, logistic scheduling, and ordering with external suppliers (which is the responsibility of a *ERP* application).

The corresponding **physical object oriented** task models are not a focus of this document. But the *outcomes* of these control levels *do* introduce very important *spatial constraints* on the other two levels of control.

Although these three ISA95 levels are *relevant*, this document does not adopt them as a **best practice**, because:

- they are not consistent with the same standard's **resource hierarchy**.
- the **separation of concerns** is too rough, preventing good **composability** of any design that would take them as a **paradigmatic** foundation.

6.2.3 Strategic, tactical and operational task levels

Cyber-physical and robotic applications add only **economic value** when they cover several “levels” of specification and control, [47]. The following terminology is sometimes used to refer to particular vertical compositions of such levels:⁴

- **strategic:** decisions about what investments in resources are needed to create profit.
- **tactical:** decisions about how to realise these strategic investments best.
- **operational:** decisions about which existing resources to deploy to create required capabilities.
- **supervision:** decision about accepting the performance of provided capabilities, or to adapt them.

⁴The terminology is inspired by the same “managerial” subdivision of industrial job descriptions as the **ISA-inspired** levels.

- **coordination**: execution of capabilities, monitoring, and reconfiguration.
- **control**: continuous time execution control.

The focus of this document is on the latter three levels.

6.2.4 Perception capability hierarchy — Proprio-, extero- and carto-ception

In the context of this document, a task model has “on-board” perception as a major component. Which naturally leads to the following three-level task model hierarchy based on perception capabilities:

- **proprio-ceptive tasks** refer only to the robot’s own **instantaneous hybrid dynamics** resources and capabilities. In other words, the **kinematic chain** of the robot *is* the **world model**, and its behaviour is that of an (electro-)mechanical energy transforming system, using only its own **proprio-ceptive** perception capabilities and its own actuators’ control capabilities.

Examples of proprio-ceptive tasks are: the instantaneous motion under the influence of a pushing force at any of the links in the kinematic chain, or the instantaneous open loop motion under the influence of torques applied at the joints, possibly together with instantaneous (artificial) acceleration constraints. Some monitoring use cases are: the motion makes the robot reach the planned position in space, as far as this can be interpreted by the robot’s own sensors, or the motion stops when a contact transition is detected via current, force, tactile or IMU sensors mounted at various attachment points on the kinematic chain.

The semantics of the **word stem** “proprio” is that of “ownership”, in other words, everything that is “one’s own”. This does not only refer to the *sensing* and *actuation* that the robot can do without the help of any other machine, (via its own **fieldbuses** and **general purpose IO**), but also to the *decision making* and *communication* that it has full ownership of.

- **extero-ceptive tasks** add **non-robot-body-centric sensors** like **cameras** or **laser range finders** localise and track **object features**, or **landmarks**, in the environment, and the control activity adapts its proprio-ceptive task execution behaviour accordingly. In other words, the range of the **robot’s extero-ceptive sensors**, and the information processing capabilities of the sensor signal processing, determine what the world *is*; for example, via *visual servoing*.
- **carto-ceptive** tasks add “virtual sensors”, to localise and track **object** features in the environment that are indicated as semantic tags on a **map**. In other words, the **robot’s extero-ceptive sensors** determine what the world model *is*, **together** with an **environment-centric map**. That map extends the perceivable *world model* with **semantic labels** (or **semantic tags**) attached to areas or objects, so that the task specification can also take (instantaneously or permanently) non-observable properties and relations into account, [14, 82]. For example, **speed limitation zones** (30km/hour, or 20 miles/hour) are not directly observable as *area* themselves, but only the **semantic tags** that indicate the area boundaries are observable. Hence, using maps also brings in knowledge representation and reasoning, to link the symbolic information in the map legend to concrete perception and control actions.

semantic waypoint areas
metric waypoint areas
metric trajectories
position, velocity, acceleration
torque
energy, power, dissipation

Figure 6.1: The **motion stack** hierarchy. No two entities in the hierarchy can be connected together in a task specification *without* specification of properties of the entities in between. Properties of entities higher up in the hierarchy can also depend on the values of lower-level entities. For example, a trajectory generator could be constrained by the amount of power available to the actuators, and by constraints on the desired positions of the robot.

6.2.5 Motion capability hierarchy — From power to semantic waypoints

Moving itself around is *the core* of every *robot* task. Of course, that motion is in itself not enough, because the *modalities* of the motion are *influenced* by a lot of things in the context in which the robot must operate:

- what is the *purpose* of the motion?
- how much *energy* (force, speed,...) is appropriate?
- how should the motion *interact* with other “things” in the environment?
- how *sure* must the robot be before it decides to move?
- ...

All of these contextual influences again appear at various levels of abstraction when representing “motion”. Concretely, this Section introduces the **motion type hierarchy** meta model depicted in Fig. 6.1. This **motion stack** starts with the level closest to the hardware, where the controller takes the decisions to put real *energy* into the robot’s actuators:

- *power*: depending on the type of actuator (electrical, pneumatic, hydraulic,...), an *energy source* provides power that is converted into mechanical power. The conversion often comes with inevitable *energy dissipation into heat*.
- *torque/force*: the physical cause of mechanical power, and the outcome of the transformation of energy in an *actuator*.
- *acceleration*: the first level of “geometrical motion” that is caused by an actuator applying a torque or force on a mechanical body.
- *velocity*: the second level of “geometrical motion”, mathematically related to acceleration by integration over time.
- *position*: idem, integrating velocity over time.
- *trajectory*: a *geometrically continuous curve* in space, resulting from changes in position of a mechanical body over a particular period of time.
- *metric waypoint areas*: a *discrete set of positions* in space, as *geometrical target area* for the robot’s motion.
- *semantic waypoint areas*: idem, but without the geometrical details, that is, just *symbolic labels* on a *map*. For example, the task of a mobile robot is to drive from its **docking-station** to **rack-45B**, travelling through **corridor-4** and then **corridor-5**.

The role of the motion stack meta model is manifold:

- each of the entities in the motion stack requires a combination of *proprio-ceptive*, *extero-ceptive* and *carto-ceptive* specifications from the **perception capability hierarchy**.

Indeed, even at the lowest level, *power*; for example, for a battery-based robot:

- *proprio-ception*: the battery has a power management control unit, sensing its

voltage, current, and temperature status.

- *exteroception*: in temperature-critical situations, it makes sense to monitor the battery temperature, and its thermal influence on the mechanical structure in which it is mounted, with external sensors.
- *cartoception*: knowing where the battery is mounted, and where on the robot body the power cables run from the battery to the actuators, can be essential knowledge in tasks with constraints on energy safety and **EMC** compliance.
- each of the levels in the motion stack requires a different **spatial extension**.
- each of the entities in the motion stack requires a combination of the *symbolic*, *discrete* and *continuous* specifications from the **representation association hierarchy**.

Indeed, even at the lowest level, *power*; for the battery-based robot example above:

- *continuous*: the battery’s power management control unit deals with the continuous values of voltage, current, and temperature.
- *discrete*: the power management can have different control states, depending on actual capacity and energy demand.
- *symbolic*: knowing the *type* of the battery, can be essential knowledge in tasks with safety constraints.

In other words, it is in the *control* models, that one starts to recognize **structures** of the **coupling** between perception, intention and world modelling of *mechanical* motion.

6.3 Single-level task specification: instantaneous motion of one robot

This Section *introduces and illustrates* the important “first principles” of task modelling, via its simplest form: the specification of the **instantaneous motion** of **one** single robot. This is **simple**, because the complexity of the model pertains to only the one level of abstraction of the **kinematic chain** of that robot, relating “joint space” and “Cartesian space” motion parameters. More concretely, this Section introduces:

- **declarative** descriptions of tasks,
- structured by the meta model of the kinematic chain,
- but still capturing all *dependencies* between the various task model primitives.

With this fine-grained approach, the focus shifts from trying to standardize what a **robot** can do, to standardizing *task models* that require some combination of **robotic capabilities** for their execution. So, each particular robot system must describe in its **data sheet** which (combination of) task models it can *interpret*, and turn into actions that it can *execute*.

6.3.1 Partial force and acceleration specifications in multiple reference points

The first of the above-mentioned fine-grained meta models introduces entities and relations that are semantically the same as in the **physical dynamics** of a robot (Fig. 4.8):

- *actuators*: they generate the forces on the *joint space* side of the robot’s kinematic chain, and measure their own changes in position.
- *tool points*: these are chosen by human programmer, somewhere in the Cartesian space of the robot’s body, as geometric primitives to attach a motion specification to.

- *kinematic chain*: (the model of) the mechanical transformation of motion and force between joint space and Cartesian space.
- *force, position, velocity, acceleration*: the four fundamental parts of mechanical motion, in joint space as well as in Cartesian space.

Because of the physical dependencies between all these entities and relations, the corresponding meta model can indeed be considered as the **smallest possible one**,⁵ in the **robotics domain**, with consistent **physical meaning**. The physical meaning introduced *before* was only about the **natural constraints** that physics imposes on any (ideal) kinematic chain be it part of a robot or not. That is, the relations between forces and accelerations at various parts of the chain, satisfying the constraints of **Newtonian dynamics**. This Section is about *motion specification*, and semantically this is a very simple extension with respect to the physics: some of the entities and relations that are defined in the task specification represent **artificial constraints** (that is, **desired** forces or accelerations) on top of the natural constraints (that is, forces and accelerations, due to physical causes such as gravity and mechanical contacts), [93].

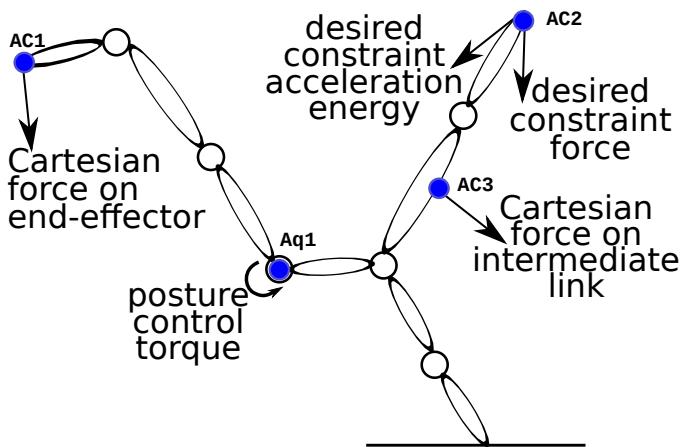


Figure 6.2: Example of multiple instantaneous *force-acceleration* specifications. Each specification is expressed in an *attachment* that is added to the model of the kinematic chain: AC1, AC2, and AC3 are Cartesian attachments to a *link* of the chain; Aq1 is an attachment to a *joint* of the chain. The specifications can be *partial*, in that they do not constrain all six degrees of freedom of the Cartesian configuration space.

Figure 6.2 sketches what a motion specification consists of, when it fully conforms to the **mechanical dynamics** of a robot system:

- one or more **attachments** are selected on the kinematic chain, to which a specification is connected.
- the physics implies that two complementary **types** of specification are possible: **force** and **acceleration energy**, [55].
- the **force** specification can be **partial**: for each attachment, not all **six degrees of freedom** of a Cartesian “wrench” (linear force plus angular moment) have to be given, and any subset is physically meaningful.
- the **acceleration energy** specification **can not be partial**, because it is just *one single scalar* number. It is **composable**: acceleration energies from different specifications just add up. Hence, it is even **superposable**.
- each specification can be given a **tolerance**: instead of just one single number as a sharp **setpoint**, the specification gives a (*continuous*) **range** of values that are all equally acceptable. That is, the task programmer indicates within which boundaries

⁵Of course, this meta model already depends on the (set of) meta models in **geometry** and **kinematics** introduced in earlier Chapters. In turn, those depend on meta models of physical units and dimensions.

the *controller* that will *execute* the specification is allowed to let the generated actual velocities lie.

- each of the **joints** is just a special case of an attachment point: the joint-space force or torque specification is naturally constrained to be only partial, namely corresponding to the degrees of freedom of the actuator. By far the most common case is a one-dimensional **rotational joint torque**.
- more than one specification can be attached to the same link, even to the same attachment.

Here is an example that illustrates how a task specification could be formalized:

```

MOVE: // name and ID of instantaneous specification:
  name: My Hello-World specification example
  ID:   force-torque-spec-rfdjk3rf
TYPE: { force-specification,
        acceleration-constraint-specification,
        velocity-progress }
CONTEXT: // "knowledge" from "elsewhere" that this specification relies on:
  1: Geometry { 3D, point, frame, line, force, acceleration }
  2: Units { QUDT }
    // "magic numbers" composed into the specification from "elsewhere":
  3: World { AC1, AC2, AC3, Aq1 },
  4: Spec { force-min, force-max, acc-constr-tol, posture-torque },
  4: Progress { vel-tol, vel-des },
DO: // specification of "drivers" of the motion:
  SpecF1: apply AC1.X.force between Spec1[force-min] and Spec2[force-max]
  SpecF2: apply AC2.Z.force between Spec1[10 N] and Spec2[20 N]
  SpecF3: apply AC3.X.force Spec1[force-min]
  SpecT1: apply Aq1.Z.torque Spec1[posture-torque]
WHILE: // constraints to be satisfied:
  SpecA1: keep AC2.origin.acceleration-constraint
           less than Spec1[acc-constr-tol]
  SpecP1: progress AC2.Y.velocity more than Spec1[vel-des - vel-tol]

```

Composability is satisfied, because:

- each statement in the specification has a unique identifier, and can hence be accessed, even at runtime, for reconfiguration, introspection, coupling to data streams, etc. For example, the *statement* of the posture control torque is

force-torque-spec-rfdjk3rf.DO.SpecT1.

- the same holds for each actual specification. For example, the *value* of the posture control torque is

force-torque-spec-rfdjk3rf.DO.SpecT1.Spec1.

- the TYPE statement in the specification provides the symbolic information about which *meta models* are needed to interpret the specification. And it is accessible as:

force-torque-spec-rfdjk3rf.TYPE.

- the CONTEXT statement in the specification provides the symbolic choice of where to get the TYPE statement meta models from. And it is accessible as:

force-torque-spec-rfdjk3rf.CONTEXT.

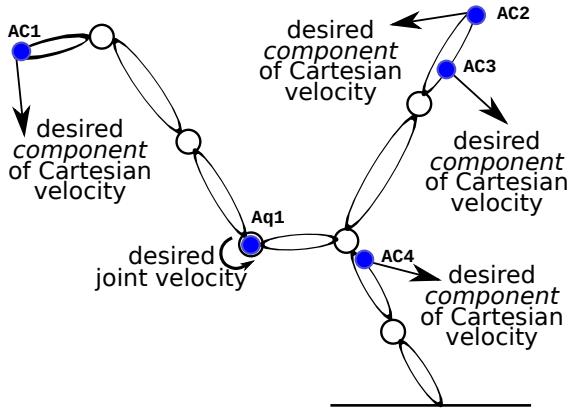


Figure 6.3: Example of instantaneous *velocity* specifications: specifications can be *partial*, and *multiple* specifications can be composed at *different* attachment points on the kinematic chain.

6.3.2 Partial velocity specifications in multiple reference points

Figure 6.3 sketches what a *velocity*-level motion specification consists of (without repeating the aspects that are shared with the force-level specification):

- in each of these attachments, a **partial** velocity specification is given.
- in each of the joints, a joint velocity specification can be given.
- “partial” means that the magnitude of the instantaneous velocity of the attachment point is specified for only one or more of the maximum of six motion degrees of freedom of that attachment point.

Of course, such a *velocity* motion specification can be given the same sort of formalisation as the *force* and *acceleration energy constraint* specification of Sec. 6.3.1, just replacing the force and acceleration keywords with velocity-level keywords. And both types of specification can be composed into the same specification, by simple composition of the specification statements and the **TYPE** meta models statement.

6.3.3 Reactive task specification — I. Guarded motion

The previous Section introduced **instantaneous motion** specifications, with force, acceleration and/or velocity domain entities and relations. This Section **composes** such a motion specification with the missing parts needed to realise the simplest possible **task model**, that is, a **sequence** of so-called **guarded motions**. The word “guard” in the motion specification refers to the **conditions** that let the robot controller decide to stop the execution of the current specification, and to start executing another one. Guarded motion task specifications go a long way back in the history of robotics [68, 93, 163], but are often “forgotten” in the current mainstream of robotics, where “(position) trajectory generation” is the dominant specification paradigm. The approach is still relevant, because it has lost none of its original appeal:

- it is **simple**, that is, realisable already with a robot’s “single-level” **proprio-ceptive** capabilities.
- it is applicable to a huge number of applications.
This includes the ones dominated by “trajectory generation”, because the *executuion* of a guarded motion specification *generates* such a trajectory *at runtime*.
- it can be used on all commercially available robots.⁶

⁶However, the support in the commercial robot vendors’ software libraries for the task specification as

Irrespective of the simplicity of its motion specification, the *guarded motion* task specification of this Section has all components of the full-fledged task specification model introduced in a later Section:

- **world model:** the robot’s kinematic chain model *is* the world model. Or one adds a model of some objects in the environment, to which one can also link attachment points and hence task specifications. For example, to direct the specified velocity in the direction of an object feature in the environment.
- **control and perception:** the task programmer must select an activity that can turn a specification into torque or velocity setpoints for the joints of the robot.
- **monitoring:** the executed motion is compared to the specified one, and if the specified tolerances are violated (that is, the “*guard*” is satisfied), an event is fired.
- **plan:** a [Finite State Machine](#), with one guarded motion specification in each state, and the state transitions are triggered by the guard events.

The selection of the control, perception, monitoring, and coordination *activities* needed *to realise* the specified task (that is, the *how* of the task execution), is *not* visible in the *specification* (that is, the *what* of the task). These activities will be modelled explicitly in the [Skill](#) that *consumes* task specifications, and *produces* correctly configured and coordinated activities.

6.3.4 Reactive task specification — II. Modulated motion

(TODO: feedforward function reacts instantaneously to state of world model. For example, visual servoing, reactive coverage.)

6.3.5 Reactive task specification — III. Adaptive motion

(TODO: adaptation function reacts non-instantaneously to state of world model, but changes some task specification parameters based on trends in some world model parameters. For example, load and friction compensation.)

6.3.6 Reactive task specification — IV. Progress monitoring

All of the task specification parts of previous Sections result in a task model that will be executed, sooner or later. Task modellers almost invariably have a mental model of what would be a “good enough” execution of their specification. Any formalisation of this “good enough” criterion—called a *progress metric*—is a condition that can be “guarded” like any other condition. Here are some examples of progress metrics:

- *speed range:* an interval on the measured speeds at one or more of the specified attachment points, representing whether or not the robot is reaching its task targets “fast enough’.
- *cycle-free* state transitions: when guard conditions are triggered, the execution switches to another state in its “discrete” or “supervisory” control Finite State Machine. It is probably not good iff those transitions lead to cycles in that FSM.
- *energy efficiency* from actuators to tools:

described in this document is partial, at best. To use these task specifications in practice, one has to add an external control computer, who takes the specifications, and interfaces to the robot only at the *joint space* parameters.

6.3.7 Textual model of guarded motion specifications

The textual specification below, together with Fig. 6.4 as “world model”, show an example of a guarded motion specification for a mainstream industrial robot type. The guards and their events can not only come from *motion* conditions, but also from any other symbolic “pre” or “post” condition that the robot programmer needs to integrate into the controller. For example, the pre-condition that a feeder has a part available for the robot to go and pick.

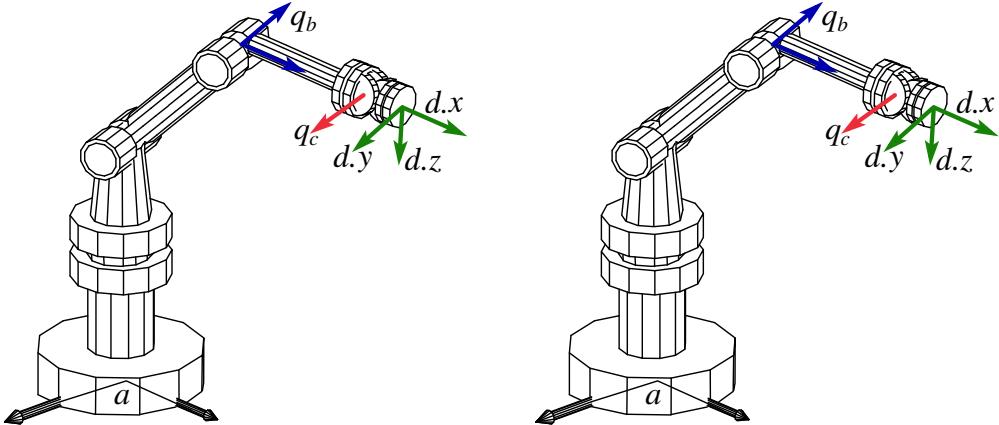


Figure 6.4: Example of an “industry-level” guarded motion. The model of the robot’s `Kinematic_chain` is extended with the `Attachments` a , b , c and d , via which the task is formally specified as a set of desired/monitored parameter ranges.

```

MOVE:      // name of specified motion
  1: d.origin in direction d.z
CONTEXT:
  1: Pre  { dist-orig, height-max },
  2: World { a, b, c, d },
  3: Spec { vel-min, angle-min },
  4: Post { dist-max }
ID:      guarded-move-velocity-23f5e3df
TYPE: {velocity-specification, frames, 3D}
WHEN:    // pre-conditions
  1: d.origin further than Pre.[dist-orig: 50 cm] away from a.origin
  2: d.z is larger than Pre.[height-max: 75 cm]
WHILE:   // per-conditions, to be satisfied during the whole MOVE
  a: keeping d.origin.speed between Spec[vel-min: 0.1 m/s] and Spec[vel-min: 2*vel-min]
  b: keeping d.origin further than Spec[height-max: 50 cm] away from a.origin
  c: keeping b.q angle larger than Spec[angle-min: 0 degrees]
  4: keeping c.q angle larger than Spec[angle-min: 10 degrees]
UNTIL:   // post-conditions, i.e. reasons to stop the MOVE
  x: d.z is smaller than Post[dist-max: 75 cm]

```

Figure 6.5 sketches the *world model* that serves as part of the `CONTEXT` for another guarded motion specification example, this time for a mobile robot moving inside of a room:

MOVE:

```

1: d1.force in direction d1.z
2: d2.force in direction d2.z
CONTEXT:
1: Pre { distance }
2: Spec { T1, T2 }
3: World { d1, d2, X2, X3, X4 }
4: Post { distance }
ID: guarded-move-force-velocity-8da83add
TYPE: {force-specification, frame, line, Tube, 2D}
WHEN: // pre-conditions
1: d2.origin further than Pre.[distance: 100 cm] away from Line(X3,X4)
WHILE: // per-conditions
1: keeping Line[d1,d2] within Spec[Tube(T1,T2,d1.origin.x,d1.origin.y)]
UNTIL: // post-conditions
1: d1.origin OR d2.origin is closer than Post[distance: 150 cm] to Line(X2,X3)

```

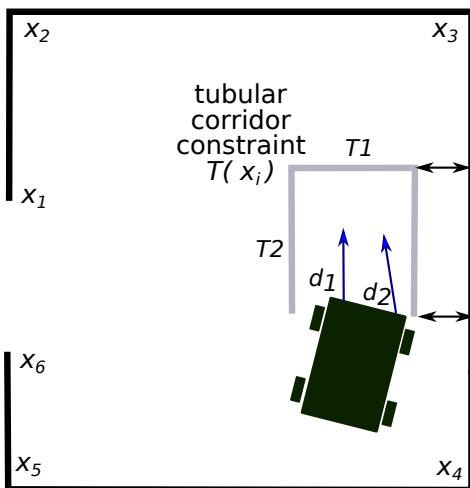


Figure 6.5: Example of a guarded motion specification to make a mobile robot drive within a “corridor”, that is defined with respect to the wall of a room. The specification uses the IDs of relevant points in the world model: d_1 and d_2 are the points on the robot where (virtual) actuation forces can be applied by the control system; x_1, \dots, x_6 are the corner points of the room. The corridor is specified as a *tube* with length and width parameters (T_1, T_2), and origin parameters.

6.4 Two-level specification: guarded execution of an order

This Section introduces another simple but common form of task specification: the specification of the **execution** of an **order** on a robotic “cell”. For example, in the context of manufacturing, an **order** is a model of what assembly operations (“Assembly Graph”), or packaging operations, must be realised on a set of parts (“Bill of Material”, BOM), with a set of resources (“Bill of Resources”, BOS). Optionally, an order also contains (references to) models containing information about *how* to realise the assemblies. An order is **guarded** if it also comes with models of:

- **progress metrics** for all transitions specified in the Assembly Graph model.
- **feedback events** about *how well* the execution of the order is progressing.

(Guarded) order execution is a common use case in the manufacturing industry, where it appears, for example, between **two “neighbouring” levels** of the **multiple ISA95 levels** in a factory. For example, between the *area* level and the *process cell* level:

- the *area* has a *Manufacturing Execution System* (MES) that *produces orders* for the *Assembly Execution System* (AES) in one or more *process cells*. The AES *realises orders*, and makes the assembled products available to the logistics part of the MES.
- the MES also produces orders for the *Logistics Execution System* (LES) in one or more *warehouses*. A LES is very complementary to an AES:
 - it brings “kits” of “parts” from a warehouse to AES cells, and returns (sub)assemblies.
 - one LES can service multiple AES’s and multiple warehouses in one order, and one AES can be serviced by multiple LES’s.
- at any given time, an MES coordinates multiple AES and LES nodes.

(TODO: more details about what exactly are the entities and relations in an order model.)

6.5 Context aware tasks: Task-Situation-Resource and Skill

The previous Sections introduce simple and less simple use cases of task specification models for some robotic applications. The not-so-simple cases require *composition* of multiple relations, linking multiple features of resources, environments, and task specifications, over multiple levels of abstraction. More in particular, the complexities of composition have the following origins:

- *task specification* is a complicated part of robotic systems design, because many influences and dependencies appear whenever a robot must decide itself about its actions depending on (i) what it is expected to realise as a *task*, (ii) what environmental *situations* it is prepared to work in, and (iii) what it has on-board as perception and control *resources*.

For example, an autonomous robot is seldom the only moving object in the world, so it has to capture and interpret the behaviour of many other things around it, because the motion and material properties of these things have an impact on how the robot is allowed to move, *and* the other way around, too.

- *task execution* must not only *react* to what its sensors and data exchanges provide as information about the past, but it must also *anticipate* to what could happen next in the situation the robot finds itself in.

For example, when navigating in *sparse traffic*, a robot can increase its speed, and anticipate how it will make, say, a sharp turn to the right by moving an appropriate distance to the left when approaching the turn. In *dense traffic* however, this speed-dependent anticipation is useless, and replaced by anticipation to, say, start earlier in maneuvering from the third lane on the left to the exit lane on the right.

6.5.1 Mereo-topological model

To realise this ambition, this Section introduces a **task meta model paradigm** that tackles the mentioned complexity in a methodological way. The paradigm’s top-level, **mereo-topological** view (Fig. 6.6) shows the *structures* via which a task specification can integrate the entities and relations needed in all the **activities** and their **interactions** needed **to execute** a specified task:

- **reality reflection** and, especially, **recognizing the situation** of the world around the robot.

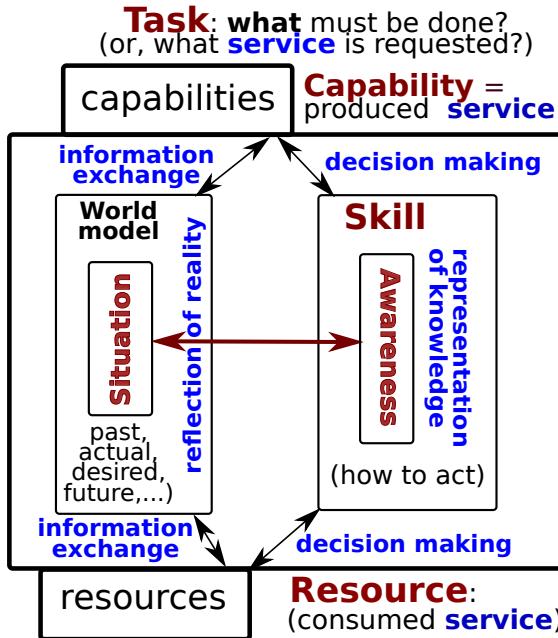


Figure 6.6: The conceptual version of the Task-Situation-Resource meta model. The Skill has the knowledge (i) to recognize *situations* in the world model, (ii) to interpret them, and (iii) to activate the perception and control that the *Task* requires. The drawing represents *knowledge relations*; so, the arrows and blocks in the drawing should *not* be interpreted as a *software architecture*.

- **knowledge representation**, and, especially the knowledge and reasoning required to add the **awareness** of the robot system to the recognized situation, *given*
 - the **resources** that the task can rely on, and
 - the **capabilities** that the execution of the task must provide to “third parties”.
- **decision making** about how, in the given situation, the task specification is “best” realised. That is, to decide which perception and control actions to execute next.
- **information exchange** required in the given situation. This includes decision making about (i) with which other “agents” to interact, (ii) at what moment in time, (iii) about what, and (iv) with what **quality of service**.

This document gives the name **Skill**⁷ to any composition of *activities* that can *realise* (“*how to do it?*”) a specified set of tasks (“*what to do?*”) with a specified set of resources (“*with what to do it?*”), in a specified set of situations (“*in which context?*”). In other words, a Skill provides an *implementation* of an *external behaviour* contract; that contract is a *model* that links the *models* of tasks, situations and resources to *executable programs* connected to a robot’s sensors, actuators and communication channels. The following Sections introduce the *internal behaviour* activities needed inside a Skill, and the knowledge and methodological structures and relations to be used to design and implement Skills.

6.5.2 Task execution activities: Skill and World model

This document’s task meta model has two complementary types of *activities* that interact with the *control* and *perception* activities of the robots, and compose these capabilities for the *execution* of tasks:

- **Skill**: the activity **to coordinate and configure** the “lower level” activities (control, perception, etc.). The design of a Skill is based on the knowledge about (i) how to execute a task given the available resources, (ii) how to interpret the contextual situation,

⁷This naming conforms to what is known under the same name in experimental psychology [79, 94]. And to the *everyday speech* interpretation of a person who is referred to as a “skilled worker”.

(iii) which trade-offs can be made in perception and control, and, especially, about (iv) how to realise robustness of the task execution against **disturbances**.

- **World model:** the activity **to produce** information about the state of the world to the decision making activities, and **to consume** the data about the state of the world from perception activities.

The top-level activity is the Skill, because without it there *is* no task execution. The model to link both activities is the **situation**, that is, those parts in the world model that a skill **must be aware of**, at any given moment in the execution of a task. The **dependencies** which make up this “awareness” are of the following types:

- **constraints** on the behaviour of an activity, or on the interactions between two or more activities. The task specification can prescribe some constraints to be satisfied before execution can start (**pre-conditions**), but also while task execution is in progress (**per-conditions**), or only after the task execution has finished (**post-conditions**). The satisfaction of the latter type of constraints is often the *reason why* a task execution ends.
- **constraints** between entities and relations that must be present in the world model before a skill can do something task-centrally useful with that information.
- **tolerances** on the above-mentioned constraints, to be “guarded” during task execution.

6.5.3 Reality reflection: capability, resource, world model, perception, monitoring

This Section and the following introduce an extra level of abstraction to that of the previous Sections. The **reality reflecting** parts [150] of the *Task-Situation-Resource* meta model are needed to **create, configure, store, update and process** the state of the real world:

- **resources:** the model of the robot’s mechanical structure, equipped with actuators, sensors and tools; and of the *constraints* of what can be provided by the resources that are necessary for the execution of the specified task. For example, mechanical strength, energy availability, computational and communication hardware properties, etc., together with the *quality of service* metrics which represent how well the resources are being used, and the *costs and risks* connected to that resource usage.
- **affordances:** “objects” or “things” in the environment offer a (physical or digital) *interface for interaction*. That interface defines (explains, documents,...) *how* it can or should be used.
- **capabilities:** the *types of tasks* that the robot can execute, including the *constraints* of what the *execution* of the specified task can deliver to users, together with the *quality of service* and *task progress* metrics, that represent how well those capabilities are being provided.
- **world-model:** the *information* the robot has about the world around it, and that *needs to be shared* between several activities. There can be multiple “versions” of the “world” present at the same time, for example, past, present, and future states. The **world-model** is also the place to remember the past, and to predict desired or possible future worlds, both with various degrees of uncertainty.
- **monitor:** the model of the relations on world model parameters with which to check whether the *actual task* behaviour corresponds sufficiently enough to the *expected* behaviour. This is the **discrete** version of **perception**, that is, the one that triggers *events*

in the **task** behaviour as soon as the mentioned checked turns out to be negative.

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

6.5.4 Decision making: plan, control, knowledge-based reasoning

Decision making in the *Task-Situation-Resource* meta model is done in:

- **plan execution:** the **supervisory control** of coordination and configuration of all other activities, according to the task specification. A *plan* is a model of which activities (perception, control, etc.) to execute, in which sequence and/or in which mutual concurrency, and under which conditions they can start or they must stop. The **plan** is the **discrete** part in the overall **control** behaviour, that is, the one for which the actions are triggered by *events* via which the **task** tries to go from the actual model of the world to a desired world model.
- **control:** the model of how to move the robot towards the targets specified in the task, taking into account the constraints introduced by the task specification and the resources. This model contains the **continuous** aspects of **control**, that is, the one for which the actions are triggered by *data* streams containing real numbers. The latter, sooner or later, always end up with the sensors and actuators in the robots' hardware.
- **reasoning:** to support the reality reflection and decision making activities, whenever there are no **hard coded** solutions available. In other words, finding out what **magic numbers** to set, in which activities, to which "right" values, based on knowledge that designers have encoded symbolically, about the application and about the dependencies between the activities.

In other words, this reasoning uses the information that represents (i) the *context* in which the other models are used, and (ii) the couplings ("associations") between these models. In other words, the information about how the magic numbers in all these models are connected to each other and to the real world, for each specific set of capabilities and resources. For example, texture or color information of an object in the world that is optimal for a particular type of sensor (processing) to detect. This knowledge is accessible symbolically from the world model via **semantic tags** that "point" to the knowledge relations.

A *plan* is a *model* used by the decision making activities, that contains the information to configure, coordinate and execute *activities* when specific *guards* are violated. One particular plan can be realised in many different ways, even given exactly the same resources. This is where the **Skill** comes in, with a model of the knowledge about **how** to realise a task. One Skill model can be "better" than another one, in a *specific* application context, because it provides better choices of the many **magic numbers** and algorithms inside the same Task model.

6.5.5 Interactions: world model, skill, perception and control activities

The **interactions** on the task meta model appear as follows:

- **inter-activity interactions via World model and Skill:** the connections between the decision making parts take place only indirectly,⁸ via interactions with the **World**

⁸At least on the mereo-topological level of system abstraction: in the **information architecture** description

`model` and the **Skill knowledge model**: the former to provide them with the “state of the world”, the latter to provide the “state of the knowledge”, and both are indispensable to base decisions on.

- **interaction channels of data, event and query:** an **information architecture** that conforms to the Task-Situation-Resource meta model will introduce several information channels between the activities that implement the activities in that meta model, of the types described in Sec. 2.9.

Queries are the most “symbolic”, knowledge-based type of interactions: each query is a complete *model* that is being exchanged between activities, allowing for higher levels of declarative interaction descriptions. In the context of this Chapter, *Task specification* models are the most relevant type of queries. And the **task queue** pattern is the obvious one to use in the interaction. Typically, the *execution* of a task specification involves several *submission* and *completion* interactions.

To illustrate the role of task specification models in complicated systems example, it is worthwhile to study the success of the “Web”. That success is based on the fact that full HTML models are communicated, that are “task specifications” for the browser: the HTML models *what* to visualise, and the **browser engine** interprets that model and decides on *how* to execute the visualisation. The HTML “task model” is in itself also a very good example of a **composable** model: an HTML file in itself composes other web standard formats, such as SVG or JPEG. Even when a receiver can not “render” the full model, the composition semantics is clear enough (i) to allow local decision making about what to render or not, and (ii) to communicate back a “status report” to the sender explaining which parts of the sent model gave problems.

The interactions can be *physical* (via the mechanical tools on the robot’s structure), or *data driven* (via “device drivers” directly coupled to the hardware), but all but the simplest of applications involve a decent amount of *symbolic information* communication, via data flows, event broadcasts, and query solving.

6.5.6 Affordances: knowledge replaces computations

(TODO: **affordance**; affordance taxonomies. The robot does not have to compute what an object can “do” or how to interact with it, because that object (or rather its software representation) can “tell” the robot all about it.))

6.5.7 Lazy skills

(TODO: solvers are *satisficing* instead of *optimizing* performance. What *is* optimized is the amount of room given to the *5Rs*: Redundancy, Robustness, Resource efficiency, Reliability, and Resilience. Has explicit *stability horizon*, in map, perception, action, and decision making of one “DOF configuration space”, after which a lower “DOF configuration space” can be used with the same expected quality of skill execution performance.)

of the system, it is possible that a *direct* data channel is established between, say, a perception and a control activity. But that operational decision is then made by the activity in charge of, say, the world model.

6.6 Horizontal task composition

If a robot system gets more motors and sensors, it can, in principle, perform more tasks at the same time. Of course, such task composition has many facets, is not unique, and inevitably comes with **dependencies** between the composed tasks that must be dealt with.

6.6.1 More tasks — Multi-tasking

The following three types of task execution coordination models are needed:

- **coordinated** tasks: **one** sub-system (a robot or not) provides all other robots or non-robot sub-systems, online, with (i) individual task specifications, and (ii) the events to coordinate the local executions.
- **orchestrated** tasks: all motion subsystems have already the task specifications **on-board**, and only the coordination events must be communicated.

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

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

For example: human-aware robotic manufacturing cells, where the reactions of the robots to the presence of humans in their neighbourhood are pre-programmed (or, better, modelled), but the coordination events inside and between robot control systems are to be generated by the latter control systems themselves.

6.6.2 Different tasks, shared world — Co-existence

6.6.3 Same tasks, multiple robots — Distribution

Some tasks involve more than one robot, or one robot can execute more than one task at a time; for example: dual-arm tasks; platooning of cars or boats; etc. The various activities involved in the task execution **share at least** a part of the world model, but in most cases also parts of perception, control and monitoring, Fig. 6.7. 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 **constraint-based control methodology** applies as for the composed tasks individually, and in many cases “all” that has to be changed are the *Coordination* and *Configuration* parts (Sec. 9.1.1) in the system’s architecture.

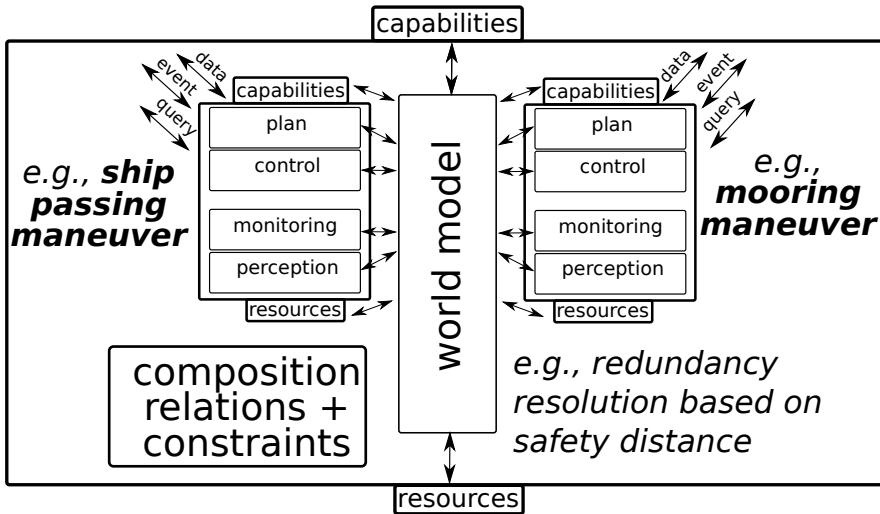


Figure 6.7: Horizontal composition of tasks, illustrated in a context of [shipping](#) via [inland waterways](#).

6.6.4 Shared control between robot and human

Robotic systems are increasingly expected to work together with humans, even in physical contact with each other. The following levels of so-called **shared control** have crystallized over the years:

0. **No assistance.** The human provides all physical power to move an object.
1. **Unconstrained force amplification.** The human's power is captured in a device, and one or more degrees of freedom of the captured human input is amplified by the robot.
2. **Partially constrained force amplification.** The robot's power amplification is constrained by virtual “fences” or “surfaces”.
3. **Autonomous obstacle avoidance.** The robot adds extra motions to the ones put in by the human, to avoid obstacles that it can perceive via its autonomous sensing capabilities.
4. **Partially autonomous trajectory following.** The robot adds extra constraints to the motion by a virtual “trajectory” (only the geometrical part of it, not the timing part!), from which it deviates as little as possible.
5. **Full autonomy.** The robot does the full control.

All of the above cases deal only with the **continuous control** aspects of a task. Even in the *Full autonomy* case, there is still a *sharing* of the control for the **discrete control** aspects: humans do the perception and monitoring in the continuous task spaces, and their inputs are not interpreted as continuous domain power setpoints, but as discrete switches between different continuous control modes for the robot.

6.7 Vertical composition: resources become capabilities

This document uses the term “vertical” to the integration of [several hierarchical levels](#) of task representations into one composite task specification. In the simplest form of vertical compo-

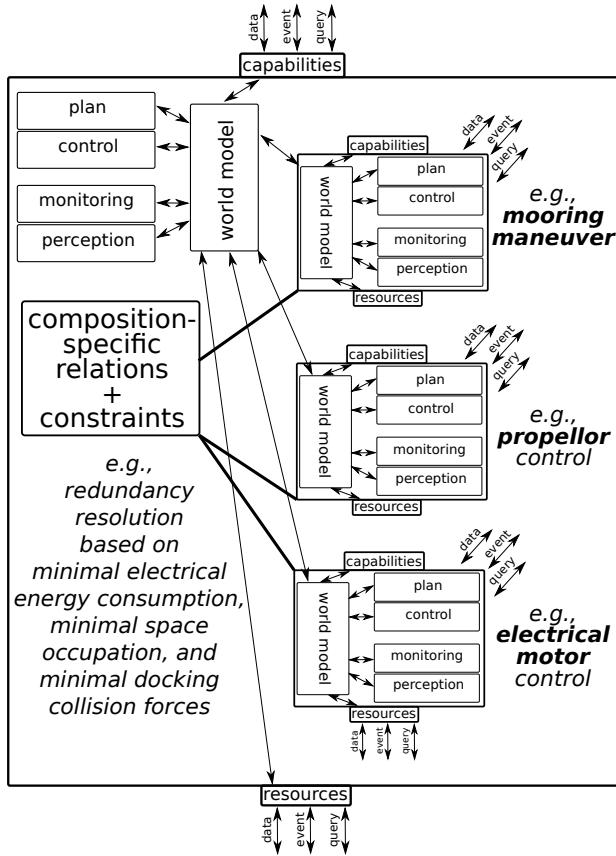


Figure 6.8: Vertical composition of tasks.

sition, the *capabilities* of a lower level become the *resources* of a higher level, Fig. 6.8. And the composite task specification **adds constraints** to the various capabilities and resource models; for example, when using the task specification meta model of constraint-based optimization:

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

For example, an electrical motor influences the mechanical joint motion control via motor heating and energy efficiency constraints; and that mechanical joint motor control influences the kinematic chain motor control via position or torque limits. The objective functions to be optimized for the motors and for the kinematic chain can be designed independently of these constraints, and it is only by the *solution* of the whole constrained optimization problem that the integration takes place. That is, the *monitoring* of the constraints gives rise to switches between several optimization problems to be solved. This approach yields a very **composable** way of sub-system integration, and the overall system behaviour emerges from the individual task executions' behaviours with high predictability, except for (i) the exact time on which the system will react, and (ii) the exact sequence of states that the system will evolve through.

6.7.1 Navigation

The high level of structure in office buildings provides a good **spatial context** to the control system of a mobile robot with indoor navigation capabilities, such as the [two-wheel driven](#) kind depicted in Fig. 6.9. The task model connects all of the following bits and pieces:

- what [resources](#) are available in a robot system to specify tasks for? What is the “best” use of the resources, in the particular spatial context that the robot is moving through?
- what are the perception, control, monitoring, and world modelling activities in the system? And how are they used “best”?
- what are the entities and relations at the three natural levels of tasks: [proprio-ceptive](#), [extero-ceptive](#), and [carto-ceptive](#).

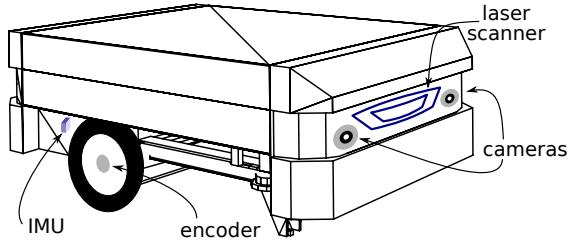


Figure 6.9: A popular hardware architecture of a two-wheel driven mobile robot: a [2D laser scanner](#) and two cameras are mounted at its front, encoders on its wheel motors, and an *Inertial Measurement Unit* on its body.

Resources

Here is a list of hardware and software commonly found in mobile platforms, and which is, hence, the list of resources to be covered in task specifications:

- the motors are [brushless DC](#) (BLDC) motors, using [field-oriented control](#) (FOC), embedded in a stand-alone [servo drive](#).
Hence, *constraints* come from [torque saturation](#), [overheating](#) with too high currents applied during too long time periods, or [charge depletion](#) of the batteries.
The *objective* to be optimized is the trade-off between realising the desired torque within the desired time and with the desired accuracy.
- the [motor drives](#) interface the motors on the one side and an [industrial PC](#) on the other side. More often than not, that PC runs a (possibly [real-time](#)) version of the [Linux kernel](#). One of the PC’s activities executes the tasks that the robot gets from its users, linking control and perception functionalities; to do so, it might need separate activities dedicated to interfacing the drives, possibly via a network connection such as [UART](#), [CAN](#), or [EtherCat](#). Another process is a server (e.g., a [Node.js](#) instance), responsible for all networking with computers elsewhere in the local network; for example, to schedule and dispatch tasks for one or more robots, and to monitor the execution. There can be another process for (possibly [web browser](#)-based) [graphical user interface](#) (GUI). Hence, *constraints* come from avoiding (i) to overload the available RAM, CPU cores, and disk space, and (ii) to violate control loop timing [latency](#).
The *objectives* to be optimized are the trade-off between realising the desired software functionalities within the desired timing, and with guaranteed data consistency.
- the robot has [rotary encoders](#) on its motors; an [inertial measurement unit](#) (IMU) and a [2D laser scanner](#) on its body, and two [colour cameras](#) embedded in its front [bumper](#). The latter is equipped with [proximity/contact sensors](#), interfaced via [general purpose IO](#) pins that are [memory mapped](#); the encoders and IMU are interfaced via the EtherCat

bus, that also provides an estimate of the motor current; the Lidar and cameras use **USB**.

Constraints come from **discretization errors**, **random** and **systematic** “noise”, and **non-linearities** such as saturation or **deviations from linear scale** measurements.

The *objectives* are to acquire the desired information about the “state” of the robot and its surroundings, with guaranteed reliability.

- a resource that is often not present in *indoor* mobile robots is a **mechanical suspension**, for example of the **rocking-bogie** type. This resource is often not instrumented with sensors, hence its impact on the robot’s motion must be **estimated** from other sensor data, such as IMU and encoder. To do so “well enough” is the *objective* of the task, and the *constraints* are the relations that link the raw measurements to the estimates, via a model of the transmission dynamics.

Proprio-ceptive task specification

The most *context-free* task models must **only take the robot’s own body into account**, that is, the “world” consists of the relative positions of relevant points on the robot: motors, sensors, links, joints, and the **tool points** on the robot’s body that are relevant to the task. Example tasks are:

- *to control* the motion of one of the tool points on the robot towards a goal location. The specification of a goal can be as simple as involving only one single point of the robot (e.g., *move one meter further*), or as complex as involving multiple references on the robot and multiple configuration spaces (e.g., *keep the two corner points of the robot on its left side on a straight line when moving forward, while pushing with a maximal torque of XX Newtons at the front side of the robot*).
- *to estimate* which motions the tool points make when actuating the motors. Actuation can be via torques or velocities. The motion can be represented in an infinite number of ways. For example, via low-complexity mathematical curves, such as **spline** or **clothoid** curves of one or more tool points.
- *to monitor* whether the executed motions satisfy the constraints provided in the task specification model.

The sensors used in proprio-ceptive control are **encoders** and current sensors on the actuators, torque sensors on the transmissions, force sensors in tool points, and/or one or more **inertial measurement units** (IMUs) on body parts of the robot.

Extero-ceptive task specification

This level extends the world model, **to take also landmarks in the environment into account**. The robot must be capable of *perceiving* those landmarks that live outside of its own body, so task specifications can include perception and landmark entities and relations. Extero-ceptive generalisations of the example tasks above are:

- *perceive the motions that the robot makes with respect to the corner of the corridor that is on the left in its direction of motion.*
- *control the mentioned pushing motion of the robot while keeping its left side at least YYcm away from the wall on its left.*

The sensors used in extero-ceptive control are **LIDARs** (laser-based distance measurements), cameras, radars and ultrasound distance sensors.

Carto-ceptive task specification

This level adds **landmarks on a map** to the robot's world model, that is, tasks can be specified using world model features that are not directly perceivable by the robot's sensors. Carto-ceptive generalisations of the example tasks above are:

- estimate how long it will take for the currently ongoing motion of the robot to reach its destination two corridors away on the map.
- control the mentioned pushing motion of the robot while keeping its left side at least YYcm away from the wall on its left, and anticipating that someone could show up from the corridor intersection that is coming up next, because the map has tagged this corridor as “busy”.

6.7.2 Mobile manipulation

The situation example above involved the mechanical composition of one single robot arm and one single mobile platform. This is often called a **mobile manipulator**, Fig. 6.10, and it is quickly becoming a very desired robotic system because it extends the useful workspace of the manipulator arm at a relatively modest extra cost.

The Figure shows that, despite that the *mechanical* composition has strong dependencies on “platform” and “arm”, the earlier *task specifications* for a single robot can be applied to the *composite robot* case too, **without any semantic extensions**. The reason for this composability is that this document introduces the *world model* as the mechanism that decouples the *physics* of the “world” from the *information* about that world.

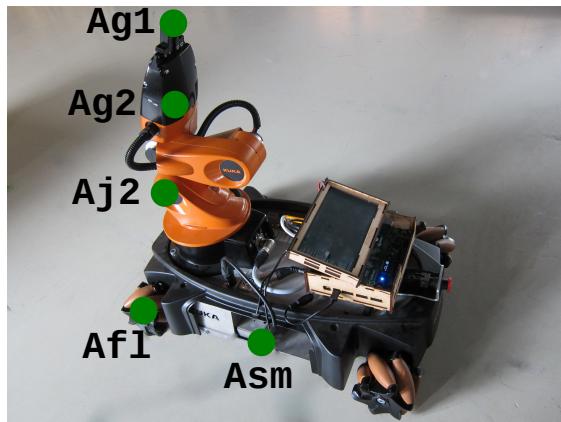


Figure 6.10: Guarded motion attachment references on the kinematic model of a mobile manipulator robot. A stands for “Attachment”, g for “gripper”, 1 for “left”, s for “side”, f for “front”, and m for “medium”.

However, the message above is too simplistic in many cases, because the dynamics of the mobile platform are most often different from those of the arm. For example:

- wheels can slip.
- tyres can deform.
- driving induces friction with the ground.
- driving over non perfectly smooth floors or grounds induces vibrations, and the effect increases strongly with speed.
- (hence) mobile platforms need suspensions, which adds more weight, more mechanical joints, and more flexibilities and damping.

The result is that the parameters in the dynamical models for a mobile platform are *inherently* less accurately known than in the model for the arm on top of the platform. Hence,

the need to introduce *coordination* between the activities that *execute* the arm tasks and the platform tasks. Both indeed have typically an order of magnitude difference in their mechanical dynamics: measurements are less accurate on wheel (not *actuator!*) motions than on arm links; the mobile platform carries more load than the arm and hence has lower bandwidth; the tolerances on arm tasks are sharper than on mobility tasks; etc.

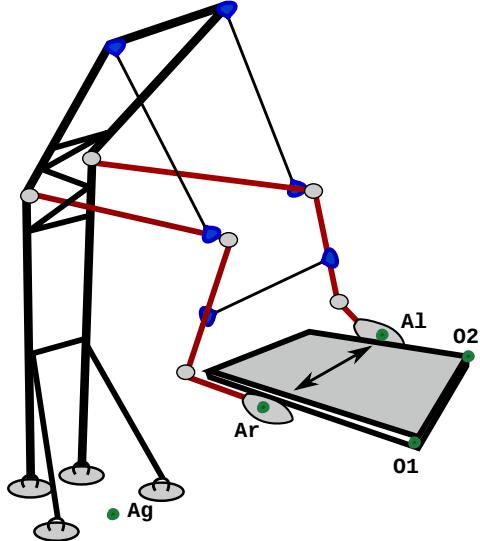


Figure 6.11: Guarded motion attachment references on the kinematic model of a dual-arm robot, carrying a plate as load.

6.7.3 Dual-arm manipulation

Figure 6.11 shows again the *composability* of the task specification, thanks to the decoupling responsibility of the *world model*. A specification example is:

- to press both hands hard enough (at the attachments A1 on the “left” arm, and Ar on the “right” arm) so that the object does not slip out of the grip.
- to keep the relevant object features O1 and O2 far enough from the body attachment Ag, in horizontal direction as well as in vertical direction.

This case does bring **one semantic extension** to the task specification model: the internal force between two attachments.

```

MOVE:
 1: dual-arm load carrying with internal force
CONTEXT:
 1: Pre  { ... }
 2: World { Ar, A1, ... }
 3: Spec  { force-internal, ... },
 4: Post  { ... }
ID:   guarded-move-internal-force-xHH43Udf
TYPE: {force-specification, ... }
WHEN:
 1: ...
WHILE:
  IntForce: keeping internalForce: {Fi, {A1,Ar}} } larger than SpecIF[force-internal]
UNTIL:
  x: ...

```

This is not equivalent to specifying two identical and opposite forces in each of the attachments independently,

```

CONTEXT:
  3: Spec { force-internal, ... },
WHILE:
  Fr: keeping Ar.X.force larger than SpecIF[force-internal]
  Fl: keeping Al.X.force larger than SpecIF[force-internal]
UNTIL:
  x: ...

```

because the *physical coupling constraint* information is lost: both forces are *always* equal and opposite in direction, not by specification but by physics. And physics also does not satisfy artificial specification choices such as that both forces are *pointing towards* each of the attachments' X axes.

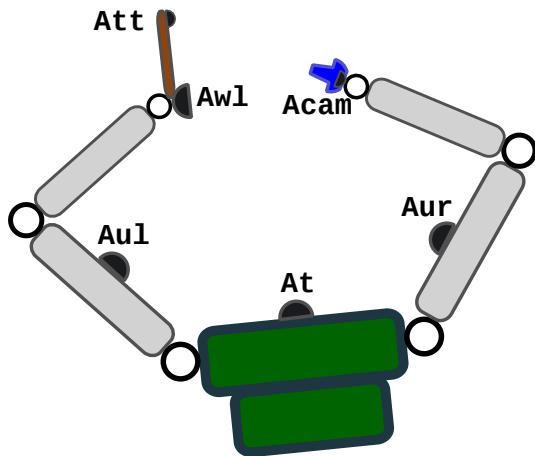


Figure 6.12: Guarded motion for an *eye-hand* coordination, with attachment references on the robot, on the tool, and on the camera.

6.7.4 Eye-hand coordinated manipulation

Figure 6.12 sketches this specification use case. It again adds **one semantic extension** to the task specification model, namely that the camera attached to the right arm at Acam should always **track** the tool that is attached to the left arm's wrist at AwL:

```

MOVE:
  1: dual-arm load carrying with internal force
CONTEXT:
  1: Pre { ... }
  2: World { AwL, Acam ... }
  3: Spec { track-tolerance, ... },
  4: Post { ... }
ID:   guarded-move-eye-hand-tracking-6sad66df
TYPE: {tracking-specification, ... }
WHEN:
  1: ...
WHILE:
  TrackCam: keeping Acam.Z tracking AwL.origin with SpecT[tolerance: track-tolerance].
UNTIL:
  x: ...

```

6.7.5 Best practice: composable specifications

Here is a list of suggestions to make one's specification models prevent common ambiguities or difficulties:

- two linear velocity specifications on the same link result in the indirect specification of an angular velocity. Two linear force specifications on the same link result in the indirect specification of a torque.
Practice shows that humans have much more difficulties to think (and control) in orientation spaces, not in the least because any orientational parts have a non-linear coupling effect on the translational parts. Hence, the just-mentioned *indirect* approach to specify angular motion components is often much more comprehensible and predictable for humans than the specification of torques and/or angular velocities.
- specifications are composable, but that **comes at a cost**: the mathematics of Sec. 4.7 show that the cost of the best known solver is:
 - *linear* in the number of joints.
 - *cubic* in the number of constraints.
- the following *trade-off* can have an enormous impact on the whole application: (i) adding all relevant specifications at the same time (letting the *solver* do the hard work), or (ii) introducing a **FSM** in a **Skill**, adding specifications in much smaller groups, in the “right” order. This trade-off is complex “to get right” because it can only be made at system level.

The guarded motion specification includes entities, “**ports**”, that are *the* mechanism to realise **composability** with the rest of the system. It is through these ports that some of the *magic numbers* in the specification get their concrete values, from “elsewhere” in the system. The reasons why the port mechanism helps composability are:

- the motion specification makes it explicit where decisions about the values of parameters are taken: as an inherent **property** of the specification (hence, not changeable elsewhere in the system), or as an **attribute** (for which the specification needs to get a value from elsewhere).
- the motion specification itself includes no decision making whatsoever: it is not an **activity** that can “do” something, but just an **abstract data type**, that can be passed as *information* to **task** execution activities elsewhere in the system.
- also all other decisions are made elsewhere: which task specifications to be executed together on a particular robot system, with which priority, or which user of the system is allowed access to the specification, etc.
- the actual content of the motion specification does not introduce “hidden parameters” for any of the above-mentioned control or decision making activities, because it is designed for **introspection**.

Because the ports connect this motion specification to its context, their formal representation is specified in the **CONTEXT** part of the specification, which structures the origins of magic numbers in the following four complementary sources:

- **Pre** and **Post** provide magic numbers for the *pre* and *post* conditions to be checked when the specification is executed.
- **World**: provides information about geometric and other entities in a world model.
- **Spec** provides the magic numbers of the motion specification.

The example illustrates that **composability** is designed into the specification:

- the specification is a model with a **Semantic_ID**. This allows the “external” world to

refer to this specification.

- this specification can get parameters from the “external” world and vice versa, via the `World`, `Spec`, `Pre` and `Post` identifiers. A `mediator` architecture is needed in the background, coordinating the different activities that give values to, or get values from, the specification’s ports..
- every line in the specification has its own identifier, which is the composition of (i) the uniquely occurring `reserved tags` (`CONTEXT`, `ID`, `TYPE`, `World`, `Spec`, `Pre`, `Post`, `MOVE`, `WHEN`, `WHILE`, and `UNTIL`), and (ii) a line tag (number or string) that gives a unique ID to that line within the local context of a reserved tag. This identifier is only locally unique within the context of one single specification, but the composition with the `Semantic_ID` of that specification results in global uniqueness. Hence, the specification can be created and updated by the system itself, at runtime.

For example, in the line `WHILE.a`, the `vel-min` parameter has been giving a default value, `0.1 m/s`, but this can be changed at execution time by having the mediator change the parameter `Spec.vel-min`.

Another example is the event “`guarded-move-velocity-23f5e3df.WHEN.1`” that will be fired when that particular condition is violated during execution; that is, when the origins of frames `a` and `d` are closer together than the specified distance.

6.7.6 Policy: redundancy dependencies in guarded motion compositions

A `guarded motion specification` as introduced above, can be impossible to execute *as is*, because some of the specifications it contains are **over-constrained**, (**contradictory** or **inconsistent**), that is, they can not all be satisfied together, all the time. Similarly, **under-constrained** situations occur, in which all specifications together do not provide enough constraints to determine all available actuated motion degrees of freedom of the robot. Both situations can also occur together, on subsets of those degrees of freedom; *and* they can some and go during the task execution, because they often depend on the **actual configuration** of the robot’s kinematic chain. The task designers can introduce a policy into the specification, as an extra higher-order constraint relation, so that the task execution has all information in advance about how to do the **redundancy resolution** between over- and/or under-constrained specifications. Here are some commonly used examples:

- *weighting* between specification statements, and/or between parameters in these statements.
- *prioritization* of them.
- the weights can be constants, or be specified as functional relations themselves.

(TODO: more details and discussion.)

6.8 Guarded motions involving physical contacts

This Section introduces a list of commonly occurring contact situations between the object that a robot arm is holding and its environment, together with a `guarded motion specification`.

6.8.1 Guarded motion: move-to-contact

Figure 6.13

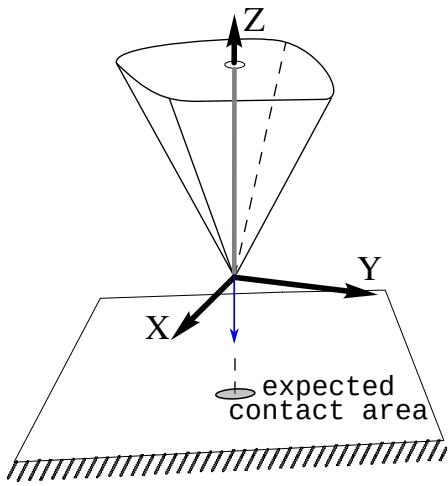


Figure 6.13: A task specification *intended* to bring a *vertex* of the manipulated object into contact with a *face* of a surface in its environment.

```

MOVE:
  1: ...
CONTEXT:
  1: Pre { ... }
  2: World { ... }
  3: Spec { ... },
  4: Post { ... }
ID: ...
TYPE: { ... }
WHEN:
  1: ...
WHILE:
  ...
UNTIL:
  x: ...

```

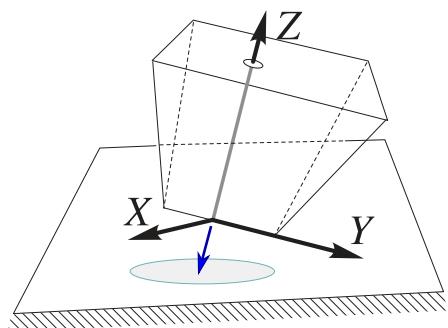


Figure 6.14: A task specification *intended* to align an *edge* of the manipulated object until an *edge-face* contact is reached

face

6.8.2 Guarded motion: Align edge to surface

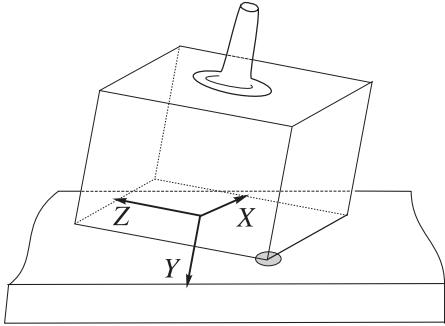
Figure 6.14

6.8.3 Guarded motion: Align surface to surface

Figure 6.15

6.8.4 Guarded motion: Align block in corner

Figure 6.16

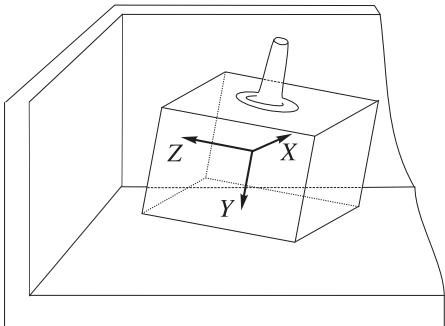


```

MOVE:
  1: ...
CONTEXT:
  1: Pre   { ... }
  2: World { ... }
  3: Spec  { ... },
  4: Post   { ... }
ID:   ...
TYPE: { ... }
WHEN:
  1: ...
WHILE:
  ...
UNTIL:
  x: ...

```

Figure 6.15: A task specification *intended* to align a *surface* of the manipulated object until a full face-face contact is reached.



```

MOVE:
  1: ...
CONTEXT:
  1: Pre   { ... }
  2: World { ... }
  3: Spec  { ... },
  4: Post   { ... }
ID:   ...
TYPE: { ... }
WHEN:
  1: ...
WHILE:
  ...
UNTIL:
  x: ...

```

Figure 6.16: A task specification *intended* to align three *surfaces* of the manipulated object until a full contact with a (geometrically matching) corner is reached.

6.8.5 Guarded motion: Track vertex over surface

Figure 6.17

6.9 Trajectory generation versus trajectory specification

The reactive approach of motion generation, as realised by the guarded motion mechanism, has the advantage that it does not rely on the specification of a **desired trajectory**; on the contrary, the trajectory is the *result* of the execution of a **motion-generating behaviour** that takes the information of the guarded motion specification as an input set of constraints on the robot's behaviour.

The above-mentioned examples show specifications within the rather restricted context of

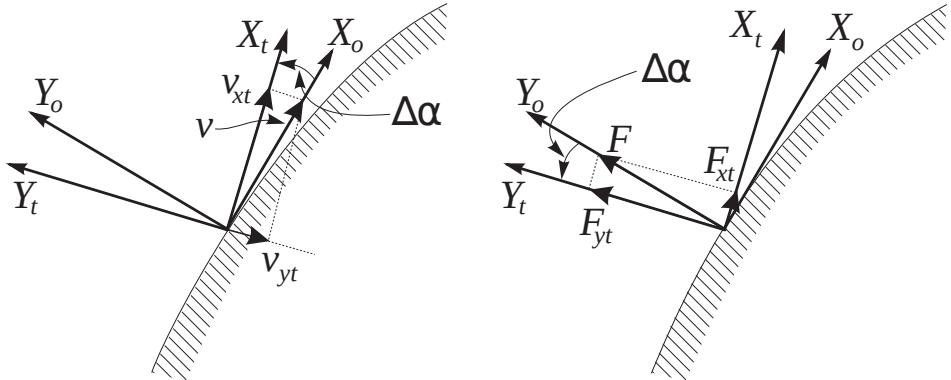


Figure 6.17: The physical principles behind *tracking* the manipulated object to follow the current contact situation. (Only shown in one dimension.)

world and robot behaviour model, namely [proprio-ceptive control](#). This context restriction was necessary, because of the computational and tooling limitations of robotics systems of the 20th century. But the exact same mechanism still applies to systems with a lot higher configuration spaces, as demanded by 21st century robotics applications. Or rather, the methodology is *even better* suited to scale than its *planned trajectory* alternative, because:

- the presented motion specification mechanism allows to take only those constraints and/or motion degrees of freedom into account at the time of *specifying* the motion that the task programmers really want to be constrained during the *execution* of the motion.
- planning geometric trajectories is rather easy, because a lot of algorithms and tooling exist to do so. But developers often forget that specifying a *trajectory* implies that *all* motion degrees of freedom are already chosen, and this compromises composability of multiple partial motion specifications.

6.10 Mechanism: specification as constrained optimization, satisfaction and reasoning

Previous Sections introduced [mereo-topological structures](#) that represent the complexity of robots acting to realise tasks in their environment, with representations in [continuous](#), [discrete](#) and [symbolic](#) domains. This Section adds one particular methodological approach to add **behaviour** to that structure, because that approach fits very well to this document's emphasis on **composability**:

- a task is **specified** as a [search problem](#) under [constraints](#).⁹
- the robot's setpoints needed to execute the task are then generated by **solving** that specification.

Two flavours of solvers (that is, [search algorithms](#)) are relevant:

⁹The system developers should aim for a representation with has enough information to create a [feasible state space](#) that is (i) a faithful representation of all task requirements and resource and environment constraints, and (ii) [computationally tractable](#).

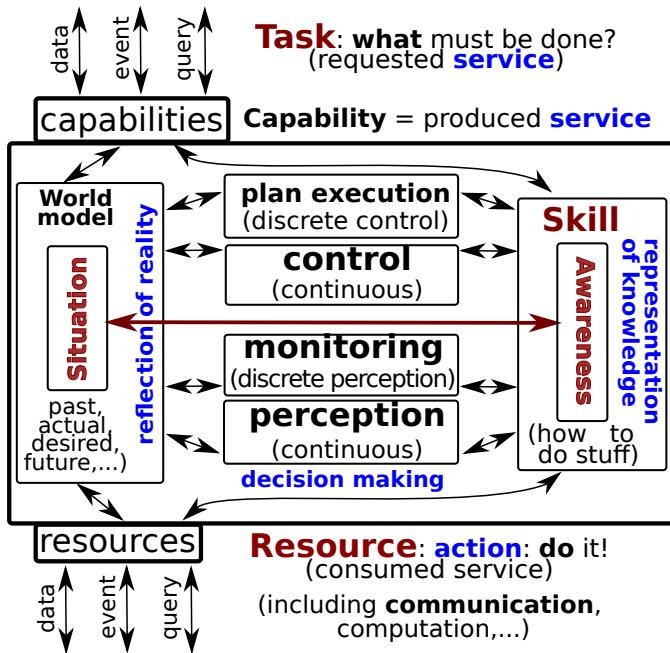


Figure 6.18: A somewhat more detailed, mereo-topological, sketch of the Task-Situation-Resource meta model of Fig. 6.6, introducing the essential **activities** (plan execution, control, perception and monitoring) required to realise tasks. The major relations in this drawing are the ones that are *not* there: the activities **do not interact directly**, but indirectly via (i) information exchange with the *world model* activity about the *current state* of the world, and, (ii) decision making delegated at runtime to the *Skill* activity.

- **optimizing** solver: one wants to find the *best* solution, where the quality of the solution is represented by the problem's **objective function**.
- **satisficing** solver: one is satisfied with a solution that is "*good enough*", i.e., the search stops as soon as a value of the objective function has been found that is above the threshold value of the desired minimum quality.

In many cases, the *optimizing* approach implies that computational resources are spent on reaching a result that is (i) not needed, and (ii) sensitive to small disturbances in the problem formulation. The role of *tolerances* in the problem specification is to indicate how well constraints and objective functions are to be realised by the solvers. With that information available, the solver's computational complexity can be reduced, because the optimization algorithm is allowed to return a result as soon as it finds one that satisfies the problem formulation in an *adequate*, satisficing, way.

6.10.1 Specification in the continuous domain: hybrid constrained optimisation

For the continuous aspects of a system, the methodology is to formulate a task specification as a (*hybrid*) *constrained optimisation problem*:

Hybrid constrained optimization problem

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

Its *outputs* are the *instantaneously desired changes* in the robot's continuous motion state $q \in \mathcal{Q}$, that is, torques or velocities. Its *inputs* are the following entities, relations and constraints, [10, 20, 37, 86, 108, 109, 130]:

- **configuration space:** all the parameters in the models of the structures (hierarchies and information associations) in the previous Sections. For example, the joint space parameters of a robot and its actuators, q , and Cartesian space parameters X .
- **desired configuration** (X_d, q_d): the sub-sets of the whole joint and Cartesian configuration spaces that the execution of the task should have as its outcome.
- **objective function(s):** relations $f(X, X_d, q)$ on these parameters that the task execution is expected to optimize. For example, the closeness to a target area, the distance to obstacles, the progress of the task execution, or the energy consumption of the robot,
- **constraints:** equality relations $g(X, q) = 0$ and/or inequality relations $h(X, q) \leq 0$, on the configuration space parameters, that must be satisfied during the execution of the task. For example, joint limits, singular configurations in a kinematic chain, non-collision, or remaining within dedicated areas.
- **tolerances:** inequality relations $d(X, X_d, q, q_d) \leq A$ that describe how well the constraints have to be satisfied, and/or how far the objective functions have to be optimized.
- **monitors:** the algorithms that compute the values of specified tolerances, and that have **magic numbers** to turn these values into *events*
- **solvers:** the algorithms that take all of the above inputs, and compute the instantaneous "drivers" for the robot's actions (motion as well as perception).
- **hybrid:** the solver algorithms also react to monitor events, and then (possibly) switch their solver algorithms.

For each particular **domain**, one must fill in the *types* for f , X , q , etc., as well as a particular *type* of solver. For each particular **application** in that domain, one has to fill in specific constraints, objective functions, monitors and tolerances. One concrete example of a constrained optimization problem is that of [hybrid kinematics and dynamics](#).

6.10.2 Specification in the discrete domain: constraint satisfaction

A [constraint satisfaction](#) problem is the discrete complement of the [continuous constrained optimization](#) problem. Its *inputs* are *dependency relations* between the behaviour in [activities](#). For example:

- partial ordering constraints between activities, about when each of them can or must start and end. Such as opening the gripper of the robot only *after* the object in that gripper has been placed on the table. Or the ordering of the [three sweeps](#) in a dynamics solver.
- protocols to access shared resources and to communicate information. Such as the transfer of ownership in [interaction streams](#).

Its *outputs* are [computation schedules](#) and coordination models that make the task execution activities satisfy the dependency relations. These coordination models are a composition of the previously introduced coordination primitives: [flags](#), [flag \(arrays\)](#), [Petri Nets](#), and [Finite State Machines](#).

Robotic systems have such a large variation in discrete satisfaction problems that no generic problem formulation can be provided, as is the case in the continuous domain.

6.10.3 Specification in the symbolic domain: knowledge-based reasoning

[Semantic reasoning](#) is the symbolic complement of [continuous constrained optimization](#) and [discrete constraint satisfaction](#). Its *inputs* are knowledge relations that represent and link the behaviour in the system's activities; for example:

- traffic rules: how do traffic signs and signals influence the motion of a robot.
- the algorithms that a robot has at its disposal to solve particular perception and control problems.
- the physical pre-, per- and post-conditions connected to actions of the robot.
- qualitative specifications of spatial actions, [40, 81, 83, 84].

Its *output* is a complete and consistent set of dependency relations that provide all information needed to create the discrete and continuous optimization/satisfaction task specifications.

Again, no generic problem formulation exists, because of the huge variation in use cases and contexts.

6.10.4 Policy: task requirement as objective function or as constraint

One should be careful about what to use as objective functions in an optimization problem, for several reasons:

- functions like *time*, *energy consumption*, or *safety*, are *derived* quantities: their values can not directly and predictably be influenced by the actuator signals q . Hence, it's often better to select them as *inequality constraints*, that are guarded by monitor activities during the task execution, to allow the plan to switch to another control approach when the *realised* time, energy, etc., fall outside of the *specified* boundaries.
- motion and effort variables are more directly influenced by the actuators, hence it is typically easier to use objective functions that combine one or more of such variables. For example: deviations from geometric paths; or predicted violations of tubular regions after a certain time horizon in the future.

- it is mathematically easy to specify a *multi-objective optimization*, via a weighted combination of various objective functions. However, this *requires* the choice of weighing factors, which are often impossible to derive from the task context in a unique or deterministic way. Hence, the weighing factors often remain very arbitrary, and not motivated by knowledge insights into the task challenges.

A primary example of such difficult weighing choice is the trade-off in the task specification between:

- **utility**: the value that a successful execution of a task brings to the system.
- **cost**: the investment in resources required to realise the task.
- **risk**: the expected extra cost of the task execution when that execution can not proceed in the expected optimal way.

The resulting **best practice** is to choose **one single objective function** per state in the Task plan, and to foresee other control states to switch to whenever one or more of the other *monitored* (but not *optimized*) functions exceed specified tolerances.

6.10.5 Policy: dimensionality reduction through task constraints

Robots can live in high-dimensional configuration spaces, with many sensors and actuators, many dynamic objects in the world each with its own state, etc. The task specification can reduce that complexity by introducing constraints between several dimensions. For example, every **physical contact** between the robot and part of its environment is an opportunity to reduce the complexity of specification, interpretation, monitoring and control:

- the number of parameters needed to model the motion of the robot in *physical contact* are reduced compared to the unconstrained situation, because many motion parameters are then also *mathematically constrained* to belong to a sub-manifold of the unconstrained manifold.
- the task can introduce *control* with the explicit intention to keep the robot on that sub-manifold, [13].
- the task can introduce a *plan* to solve a six-dimensional uncertainty as a series of lower-dimensional control problems, [147].

For example, specifying the motion of a point on the robot's end-effector on a surface is a two-dimensional problem, and not a three-dimensional one anymore. Letting the robot find the corner of a box can be done using a plan that, first, lets the robot find *any* point on the box by specifying its motion in a two-dimensional plane until a contact is detected, and then moves over that contact plane until it finds the plane's edge, and so on.

6.11 Composition of guarded motions — Situations

6.11.1 Situation: composition of specification–world model–coordination

Figure 6.19 shows (parts of) a **situation**, that is, a commonly occurring set of tasks for a robot (or for a set of robots) whose *model* and *specification* can be given in a very parameterized form, designed for reuse and further specialisation in concrete applications. This document defines a situation as the *composition* of:

- a **symbolic identification**.

The example in the Figure might be called "mobile manipulation on a table with a corner".

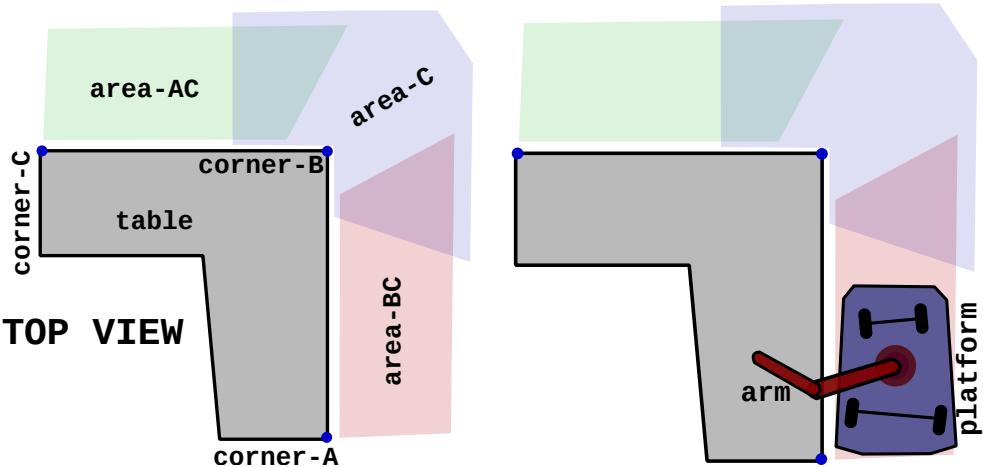


Figure 6.19: Example of a *situation* that composes several guarded motion task specifications with a world model, and with a *coordination* model.

- a **world model** with areas that have a **purpose** in the task.

The example in the Figure has polygonal models of the table, and of three overlapping areas, each representing¹⁰ the working areas for the robot in front of both sides of the table, and around the corner. The symbolically relevant features are the line segments in the polygon, and the three relevant corners at the robot's side of the table.

- a set of **activities** that must be **coordinated**.

The Figure does not have an explicit mentioning of these activities:

- the arm manipulates objects on the table.
- the mobile platform navigates around the table, as a service to the arm, to keep the latter in sufficiently comfortable kinematic conditions.
- various sensors localise and track the relative positions and motions between table, arm and platform.

- the **task specifications** for each of the activities, specified as functions of (i) properties and relations in the world model, and (ii) “progress flags” of the (coordination of the) activities. Examples of the latter are:

- **ramp-up**, **ramp-down**: with these arm-owned flags, the arm signals that it wants to be starting, respectively stopping, the motion that “something” is providing to its base, to move along the table.

The platform owns similarly named flags, to signal that it is going to speed up, respectively slow down. The reasons need not be communicated explicitly. For example, an obstacle must be avoided, or the platform wants to reconfigure its kinematics, to adapt the efficiency of the energy transmission between actuators and platform.

- **move-further**, **move-closer**: the arm prefers to be moved closer to, or further from, the table.

- **cornering-upcoming**, **cornering-finished**: the sensor-based perception activities signal that the corner of the table is coming closer, respectively that the corner

¹⁰For reasons of simplicity, no efforts was done in the Figure to add all necessary geometric properties and relations.

has been rounded sufficiently far so that “normal” activity along the side of a table can be resumed.

(TODO: add explicit textual models of the [guarded motions](#) of all mentioned tasks.)

- the **causal relations** that link specifications to the *reason why* the specifications are what they are.

Examples in the context of the table cornering task are:

- for each of the above-mentioned flags, there is a reason to raise (and lower) it. For example, the kinematic and dynamic “efficiency” of the current configuration of the robots involved, arm as well as platform; or the uncertainty and [integrity](#) of the sensor processing data.
- for each of the constraints in the [task specification](#) has a reason to have been added to the specification. So, when that constraint is violated, there is (already at least) one hypothesis about *why* that happened.

6.11.2 Situations versus templates

At first sight, one could interpret a *situation* as a special case of a [template](#), for [progamming](#), or [web](#) and [text](#) publishing. But a *situation* adds important functionalities (and hence inevitable complexities):

- it covers continuous, discrete, and symbolic levels of abstraction:
 - *continuous*: covered by the parameters in the textual [task specification](#).
 - *discrete*: covered by the coordination model; most often, some form of [Petri Net](#).
 - *symbolic*: each combination of one or more association relations for [control](#) or [perception](#) that is relevant in the situation model deserves its own dedicated representation. Hence, these various representations are mutually linked by these *types* of association relations.

For example, the symbolic identifier "mobile manipulation on a table with a corner" is already the highest level of abstraction. And there are three tasks at this level (arm, platform, perception), coordinated by an instance of the [Par-Sq-Join](#) Petri Net.

Going one level more in detail, there are more tasks, and hence also their coordination is more involved. The arm’s task execution activity must transition between [approach-to-object](#), [manipulate-object](#) and (maybe) [dispense-object](#) tasks, where the middle one might want the base *not* to move when high accuracy is required, while the other two might have very low expectations about the “accuracy” of the mobile platform’s motion. The mobile platform’s task execution activity must transition between tasks that navigate along the table and that “dock” against the table; also dealing with kinematic non-optimal configurations, and obstacles on the road, are necessary tasks.

- heterarchy of coordination interactions
- causality relations
- the formal representation in a model can not be transformed into software by mainstream [context-free languages](#): a situation is modelled by *graphs*, and hence “filling out” all the parameters and data exchanges requires [graph querying](#).

6.12 Task specification, data sheet, contract, responsibility and commitment

Models of tasks are necessary to represent *what* a system has to achieve as results. But systems are more than just about functionality and performance: **non-functional requirements** are very important to assess the added value of a system deployed in a particular application context. This Section introduces some relevant higher-order relations that connect a functional task description to some non-functional **contracts** that the system subscribes to when it accepts *to execute* a task description. Contracts come in various shapes and forms, and the purpose of this Section is not to be exhaustive, but the following two categories are **common**:

- **commitment**: the task executing party accepts a **best effort** contract, in which it guarantees that it uses the agreed-upon amount of **resources** to reach the task requirements.
- **responsibility**: the task executing party accepts the **liability** to realise the agreed-upon **outcome** of the task.

In practice, contracts are sterile relations between two parties, unless there are quantifiable ways to assess the extent to which each party respects the contract. No new *mechanism* is needed to represent a contract model: the generic **higher-order relation model** suffices. Indeed, also the non-functional *requirements* for a task must be transformed into a set of *constraint* relations, with *tolerance* relations on top. “All” the contracting parties have to do is to agree on:

- *how to measure* these tolerances.
- what *costs* are implied by which level of tolerance violation.

6.13 Taxonomy of action, actor, actant, activity, agent

This Section introduces terms that (sometimes) appear in the literature as complementary *stakeholders* in task representation and execution.

The “**action**” noun is a **semantic hypernym** for the two nouns **motion** and **perception**, and it represents a **model of what happens in the world**. (The same action model can have various interpretations: “actual”, “desired”, “possible”,...) Action models appear in *all* robotic systems, at various **levels of abstraction, resolution or granularity**, encoded with various (often hierarchically) **interconnected higher-order relations**, and with a large variety of “**performance**”.

More semantic concreteness comes from the identification of (i) the **actor** that is responsible to execute the **action**, and (ii) the **actants** (that is, **objects**) that are required to make the action succeed, or that the actor has to take into account as possibly impacting the successful execution of the action. Any robot can move itself (hence the actor and the actant are the same), but a hand grasps “something”, that is, the grasped object is the actant; a pinch grasp is performed with only the thumb and the index finger. This example makes it clear that the **spatio-temporal scope** of each term becomes smaller if the term is attached “deeper” in the hierarchy.¹¹

The terms action, actor and object, as introduced above, represent **knowledge models** in this document. This knowledge is used in a **activity**, which is a **process** that implements

¹¹The oldest(?) reference that explicitly introduces such natural hierarchy of **increasing order of intelligence with decreasing order of precision** is [127].

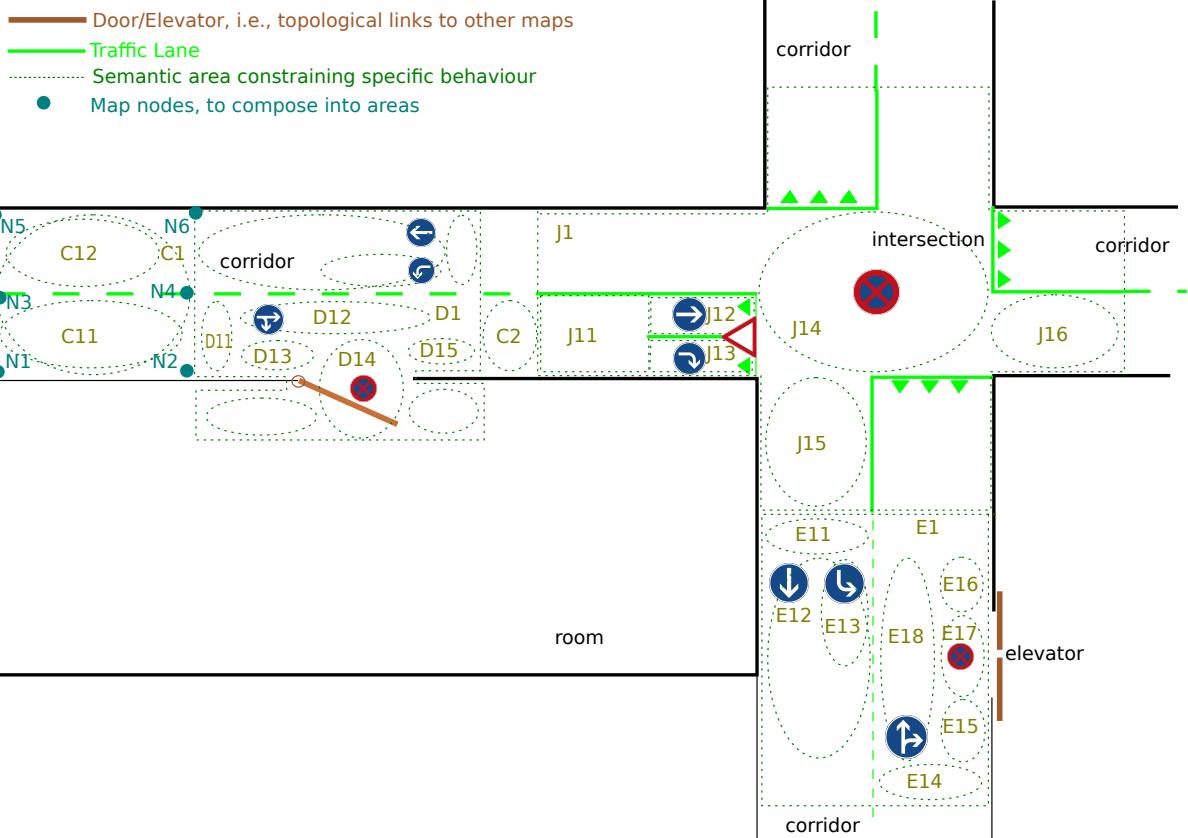


Figure 6.20: *World model* of an indoor “two-corridor-with-intersection” area: the solid black lines represent the **geometrical** properties of walls, and the red lines represent doors. The other map entities are the **semantic tags** of the traffic signs and markings. (Figure courtesy of Nico Hübel.)

the execution of an action, with “physical processes” or “digital twins” for actor, actant(s) and action.

Finally, the name **(software) agent** is given to the **system** of activities that belong together, as being executed by one single physical system in the real world.

6.14 Bad practices in task specification

(TODO: instantaneous reactive control, e.g., via potential fields; weighing of translation and rotation; weighing of objective functions and constraints; not making parameters dependent on the context, but use as fixed **magic numbers**; solving problem to the optimum, every sample again instead of tracking solutions and using a satisficing approach; defining the error functions as instantaneous errors on setpoints in position, velocity and acceleration; neglecting the fact that constraints on joint velocity have seldom physical or economical sense.)

6.15 Use case: semantic indoor navigation (revisited)

This Section extends the short [introductory description](#) of indoor robot navigation, towards **semantic** navigation. Fig. 6.20. The inspiration comes from the **traffic system**, or “driving” in general, as a major societally relevant systems-of-systems example. One can assume that everyone is acquainted with the a formal representation of (i) tasks and of (ii) the perception, control and monitoring required for the execution of those tasks. That is, recognizing building features or traffic signs, localizing them in their spatial context, and inferring their influence on the robot’s current actions.

Traffic lanes, signals and signs are amongst the most familiar **semantic tags** worldwide: all drivers are trained to interpret those **symbols**, and to let them influence (i.e., constrain, as well as optimize) their driving behaviour with cars and bikes, and their traffic participation as pedestrians, individual as well as groups.

The symbols model the *constraints* that every robot must respect when it drives through the area. Each such **symbolic** tag is, directly or indirectly, attached to one or more **geometrical** features of the map. These symbols model “motion” in a fully **declarative** way, because they do not prescribe **how** a traffic user should move, but rather which relations the actual motion is expected to satisfy. The real-world **instantiations** of the traffic symbols are **designed to be perceivable** by human drivers in (almost) all environmental and weather conditions. One of the meanings attached to traffic symbols is the area they cover in the world, and the shape and extension of these areas are **designed** to fit to the **control bandwidths** that can be safely expected of all actors that take part in the traffic. Together, a traffic layout is an **architecture** of semantic traffic primitives in the world that (almost) guarantees that **humanly controlled systems** can drive safely and efficiently. (That is, as long as the latter reduce the risk during participation in traffic.)

The main purpose of this Section is to explain how *engineered systems* can make use of the [task meta model](#). The first step in that direction is to add the **geometric** level of abstraction, Fig. 6.21: the **world model** is filled with (models of) the entities in the environment: the location and shape of the traffic areas, and the shape and motions of the robot’s **kinematic chain**. In addition, the world model gets relations that link some of its geometric primitives to perception capabilities that are present in the robot control system: there are some “walls” in the environment of the robot that its laser scanner sensor processing algorithm can detect and track, so that the position of the robot in the world model can be updated when new sensor information comes in.

6.15.1 Geometric world models with semantic tags for control & perception

Geometry. Figure 6.20 depicts a **world model**, with information at the *geometric level of abstraction*: it has **geometric primitives** such as **points** and planar **polygons**, and it has **semantic tags** (each attached to a geometric primitive) to model **landmarks** (i.e., *task-relevant* places in the world) that have **features** (i.e., *task-relevant* properties of a landmark used in **motion** and **perception** models). The Figure sketches an indoor area of two intersecting corridors, with doors to rooms and elevators. These doors and walls form **geometric constraints** for any **motion** that robots execute in the modelled world, because they have to steer clear from collisions with these “hard” world landmarks.

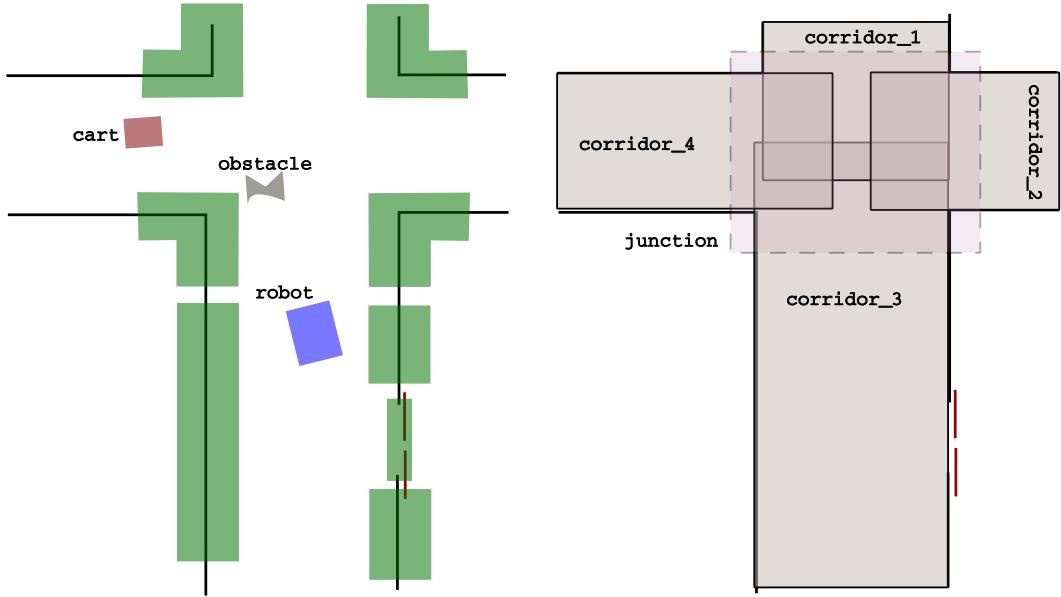


Figure 6.21: The semantic “base layer” of a world model, with several types of “tags”. Left: perception tags (in green) for a laser range finder with sensor processing capabilities that can detect straight wall segments and rectangular corridor corners. The drawing also contains the tags (and the spatial extension) of objects that are relevant for the Task: the **robot** of course, and the **cart** that it has to navigate to; but also an unidentified **obstacle**. Right: five (overlapping) **free space** areas, one for every **corridor** that ends up in the **junction**.

Plan. The next step after the modelling of the world, is the modelling of the task plan. The simplest form of such a plan is a *finite state machine*, with in each state a choice of a model for the *control*, the *perception* and the *monitoring* that the robot is expected to realise in that state; the monitoring provides the *events* to trigger a state change in the *plan*.

Figure 6.22 sketches how to use world model landmarks to attach some essential tags used in a **plan** model: a “tube” is connected to some tags in the traffic model, to indicate the area within which the *control* must keep the robot, while its *perception* measures whether it is making “good enough” progress towards some other traffic tags; various *monitors* will follow the approach to one or more of these target tags, and signal the *plan* when the robot has “reached” one of them. More concretely, the robot should (i) not drive into the natural constraint of the wall but follow it, (ii) satisfy the artificial constraint of the traffic lane, *and* (iii) be ready to stop in time in front of the intersection.

Control. In symbolic form, the control model can be as simple as the two blue arrows in Fig. 6.22: the arrows represent two driving forces:

- the *origin* of the symbolic force driver arrow is a reference to a physical attachment point on the robot;
- the *end* of the arrow is a reference to some of the target tags in the world model.

More concretely, in the Figure, the driving forces are directed towards the semantically tagged area of the intersection of this corridor with the junction area. In a full formal representation,

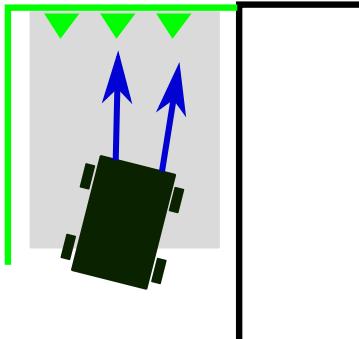


Figure 6.22: Semantic world model, extended with a motion **control** specification, in the form of a **tube** (the shaded gray area that represents the constraints of the control) and **motion drivers** (the blue arrows that represent (artificial) forces that generate the **instantaneous motion** of the robot).

this “pointing to” corresponds to a topological connection between the instances of object features on the real robot and the real environment.

The Figure also sketches another type of semantic tags, representing a **right of way** constraint. This tag has an *area* of influence, that is indicated symbolically by the green line segments.

These symbolic forces, and the symbolic constraint, are inputs to a motion controller, that transforms them to actual actuating torques on the motors, maybe by solving a **hybrid constrained optimization problem** (HCOP) in which the symbolic relations are used to (i) select the parts that must be included in the HCOP as relations, and (ii) some **magic numbers** in those relations. In this way, the control algorithm remains simple and constant, because it is rather straightforward **to inject** the context-dependent information (that is embedded in the world model) into the algorithm.

The geometry-based semantic tags are **not enough** to specify the controller’s HCOP completely. Here are some examples of missing information, that must still be *injected* too:

- *progress measure*: the geometry can provide information about the **direction** of the motion, but information about what is an appropriate **speed** must come from other types of knowledge. For example: maximum, minimum and/or optimal energy-motion curves for the robot; time-sensitivity of the task (e.g., a rush order); safety regulation.
- *tube following policy*:
- *obstacle avoidance policy*:

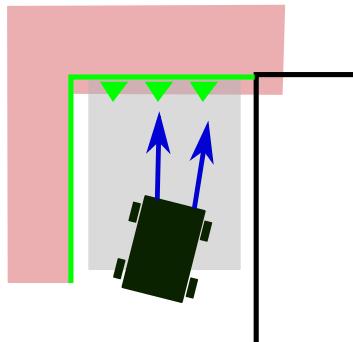


Figure 6.23: Semantic world model of Fig. 6.22, extended with the **local perception tags**: the green area focuses the perception of tracking a “wall” feature, at a resolution high enough to support the motion control; the red area focuses the monitoring on the detection of “any” feature in the direction of motion to react to, at a resolution low enough to react in time.

Perception. Some (possibly different) semantic tags in the world model also serve the robots’ **perception**. For example, Fig. 6.21 (Left) represents the **perception** tags for a simple laser range finder type of sensor: each tag is an *area* connected to a landmark on the world model (e.g., a *wall*) and the area indicates the “attraction region” in which *all* LIDAR measurement points can, or should, be associated with the wall landmark. Figure 6.23 adds

a second perception tag: the nearby “rest” of the environment in the direction of motion, to be monitored for the presence of “obstacles”; that is, *any* LIDAR measurement point in that area makes the monitor “fire”. That signal can be taken up by the plan coordinator, to switch to another part of the *plan*.

An important added value of the represented task *knowledge* is that all the sensor data that comes from beyond this local horizon *need not be processed*, because it does not have an impact on the currently executed parts of the *plan*. In resource-constrained applications such as robotics, it is indeed as important to know what *not* to spend effort on as it is to know what *must* be done.

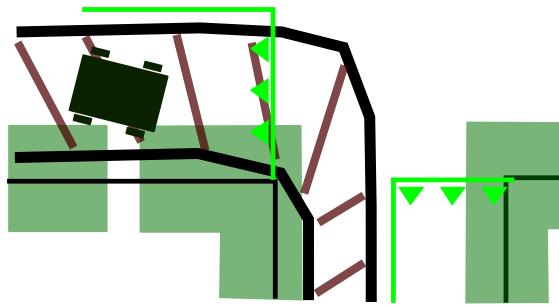


Figure 6.24: Semantic world model, with a task horizon that covers the sequential composition of two or more sub-tasks. More in particular, given this world model areas to the robot controller suggest that, in its current motion execution, it should already take into account that “something” might come from “behind the corner”. A possible control behaviour that would profit from this knowledge can make the robot move around the corner, further away from that corner than it would have done without the “preview” information.

Preview control. A more advanced task plan can include a *preview control* model, as depicted in Fig. 6.24: one can compose two or more sequential sub-tasks “to look ahead” to the next area on the map that the robot has to drive through, and to provide more extensive natural and artificial constraints that come with this extended context. The composite task plan is again a “tube” as in Figs 6.22–6.23, but now one with a bend around the next expected corner. Again, nothing changes in the perception, control and monitoring functionalities, because all extensions are added to the world model, and to the plan.

6.15.2 Specification of an activity to realise a task specification

Activities are an important target for task specification, because the declarative geometric specifications of the previous Section must, in one way or another, be transformed into *actions* of the robots, and *activities* are the system architecture primitives where that transformation responsibility resides. More in particular, an activity adds the following to a specification model:¹²

- *algorithms*: to interpret the raw sensor data in the context of the task specification, and to generate setpoints for the actuators. These core algorithms are complemented by monitoring algorithms, that evaluate how much “progress” is made towards the task goals.
- *coordination*: a robotic system of a realistic complexity has many sensing, control and monitoring algorithms active concurrently. Hence, there must be a coordination of which

¹²The details of the components in an activity are explained in more details elsewhere in the document.

algorithms to run, on which data. Such coordination has, of course, an algorithmic basis itself.

- *communication*: different activities can run concurrently on the same CPU, or in parallel on different CPUs or machines. Hence, some data must be exchanged between them, either via coordinated access to shared memory, or via message passing mechanisms. All of these also involve their own dedicated algorithms.
- *event loop*: all of the activity parts described above result in a set of functions to be executed on a set of data structures, and the event loop is the place where the functions are actually executed.

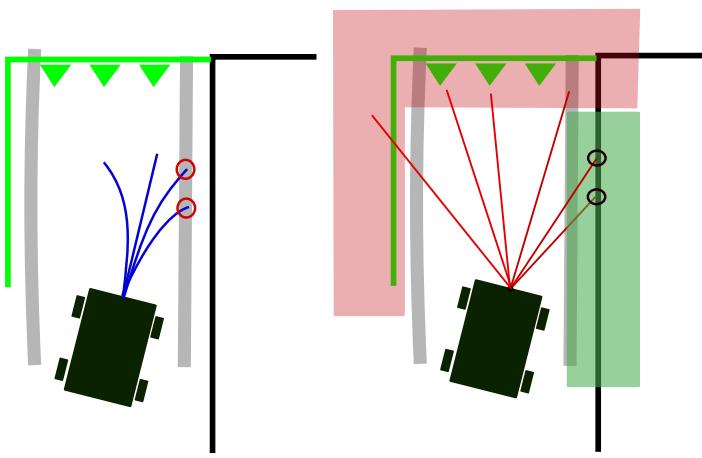


Figure 6.25: Left: control part of the task. The best trajectory for the controller to choose is the one that (i) comes closest to the end target area of the currently traversed section of the world, and (ii) is furthest away from intersection with the (artificial) “task tube”.

Right: perception part. The localisation of the robot is the estimate that (i) makes the simulated world fits best to the sensor measurements, and (ii) deviates minimally from the previous localisation estimate.

Figure 6.25 sketches possibly the simplest set of activities that warrants to be called a “robotic application”. When the sensors are limited to [encoders](#) on the wheels, and a [LIDAR](#) on the robot, activities of the following kind are needed:

- *motion activity*: the Figure sketches the primitive motion that the robot can realise, that is, when applying constant setpoints to its motors, in [open loop](#), the robot will realise a curved trajectory, as represented by one of the blue curves in the Figure. A possible [feedback control](#) activity works as follows. The controller tries different motor setpoints, on a *model* of its motion behaviour, which results in the series of blue curves depicted in the Figure. It then selects that trajectory which has the “best” motion within the “tube” specified in the task model. In this example, the “best” trajectory is the one that comes closest to the end of the current world model segment available for motion.

If that “best” one results in the robot moving too slow towards the end point of the segment, the *speed* setpoint is increased, for example with an [ABAG](#)-like controller. If too many simulated trajectories intersect the “tube” on the left or on the right, the *steering angle* is increased, also with an ABAG-like controller. The waveforms of the ABAG controllers are to be chosen by the control developers.

- *sensing activity*: the Figure sketches the usage of a laser scanner, with two “hits” encircled in black. These are used to update the position of the expected wall “line”, for example via a (Bayesian) localisation algorithm. The latter is also a constrained optimisation problem, because it finds the *minimum* error between the measurements on

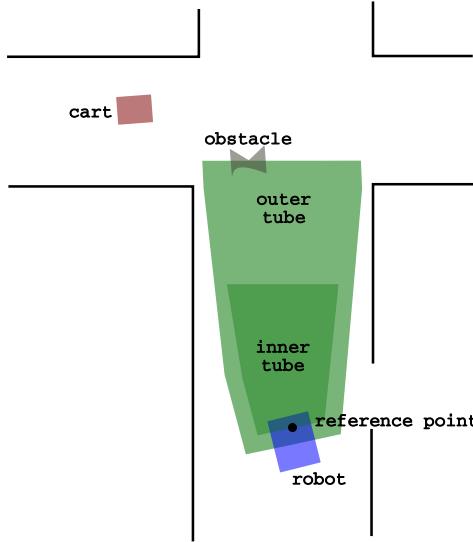


Figure 6.26: In the environment sketched in Fig. 6.21, this drawing sketches the first of three “tube” specifications, for the task of moving the robot to the cart. This first part must bring the robot to the junction.

the one hand, and, on the other hand, the modelled world with the estimated parameters of the wall filled in.

- *world modelling activity*: this activity updates the information it already has about what objects are present in the robot’s environment, and where they are. The information comes from, both, a *map* that is available a priori, and the sensor processing of the LIDAR, often via one or more levels of *data association*.
- *monitoring activities*: the task specification has modelled the desired progress of the robot, in a model of the task that was specified on the a priori map. One or more monitoring activities compute to what extent all of the constraints in the task specification are satisfied. (But also to what extent the required resources are available for the task execution.) This can be done in many different levels of the *data association* (e.g., wheel unit, platform, environment), and considering many different features (e.g., motion, effort, time, oscillations, etc.).
- *communication activities*: all of the above-mentioned activities share one or more data structures, via interaction with the world model. That interaction is a form of “communication” between these activities.
- *coordination activity*: when the *control* and *perception* errors are “too big”, as quantified by the *monitoring* activities, the coordination activity reacts to this situation by selecting another combination of *control*, *perception* and *monitoring*, as available in the *plan* model.

6.15.3 Example task plan: dock to cart in next junction left

This Section gives an example of what the just-mentioned *coordination activity* could look like, when the context of the task specification is broadened a little bit, from the *single action* context of the previous Section to the context depicted in Fig. 6.26.

The broader context requires the **composition** of several of the “primitive” task specifications introduced before. The task **plan** can have the three sub-tasks of Figs 6.26–6.28.

The first specification wants to bring the robot to the end of `corridor_3` in which it currently resides; it uses an “*inner*” and an “*outer*” tube to reach this goal. The robot

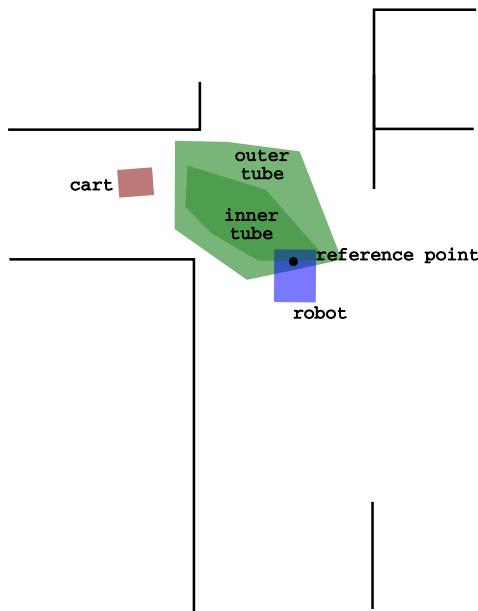


Figure 6.27: The second “tube” specification makes the robot traverse the junction, towards the corridor on the left.

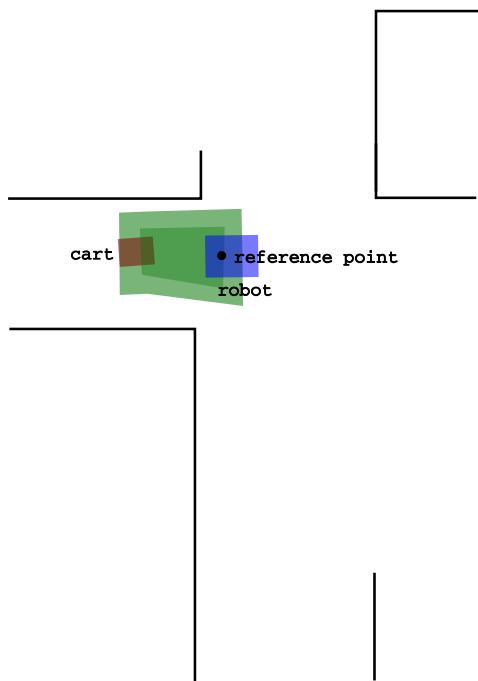


Figure 6.28: The third “tube” specification makes the robot move towards the cart.

must cross the junction towards the `corridor_4` where the `cart` is, but because the sensing and world modelling activities have placed an `obstacle` on the map, the specification in Fig. 6.29 has been added to the plan. That one first passes the obstacle “on the right”, before starting the third “nominal” task specification, Fig. 6.28. All specifications are materialized as particular instantiations of the generic inner-outer tube template.

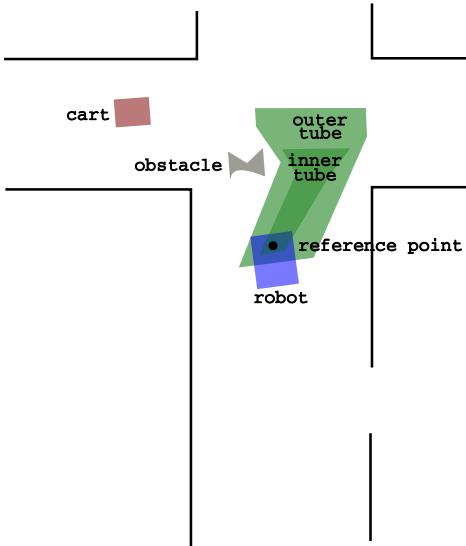


Figure 6.29: When in the task specification of Fig. 6.26, an obstacle shows up somewhere in the areas covered by the “tubes”, and obstacle avoidance task has to be introduced. This could be done as sketched in the drawing, which is just another specific instantiation of the generic inner-outer tube template.

6.15.4 Example task plan: escape from a room

This Section provides another example of composite task specification, linking the various specification parts in the [generic Task model](#): *to escape from a room*, Fig. 6.30.

One of its simplest possible **plans** has the following nominal *sequence of states*:

1. initialize sensors and motors, without motion control, perception or monitoring;
2. move forward till wall is detected, with the simplest possible control and monitoring;
3. move while *following wall on the right*, with somewhat more extensive control and monitoring, and with wall detection perception.
4. turn right at first large enough hole, with extra hole detection perception and monitoring.
5. stop.

The sequence above only represents the *nominal* plan, that is, it is not ready to cope with an execution of the robot’s actions that would bring it in other states than the mentioned sequence. A **robust** plan, i.e., one that can cope also with non-nominal executions, must have monitors in all of its states, to check whether the sensor measurements still satisfy the assumptions that hold in each plan state, within a task-specific tolerance. Typically, a robust plan requires an order of magnitude more design efforts than a nominal one.

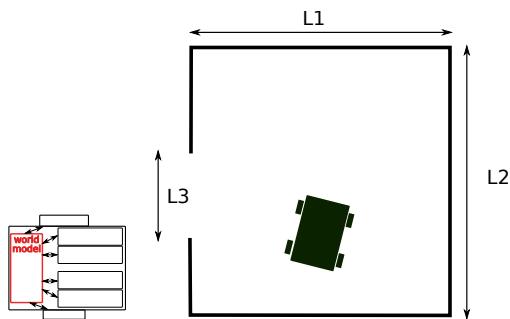


Figure 6.30: The **task** of the robot is to drive out of a rectangular room with a hole in one of its walls. The size and position of room and hole are unknown. The small figure on the left refers to the generic task model of Fig. 6.18, and indicates its parts that are involved; in this case, only the *world* is being modelled.

The **resources** available to realise the task are assumed to be:

- *laser range finder*: it provides at regular intervals in time an array of rays, regularly spaced in a range of orientations, and indicating the free space within a minimum and maximum range of distances.
- *encoders*: they provide the change of the robot's wheel rotations over time, and hence an estimate of the instantaneous velocity of the robot.
- *velocity control*: it tries to realise the instantaneously specified desired velocity of the platform, via control of the corresponding wheel velocities.
- *effort value*: one scalar that represents the percentage of “full” available power used for the current motion.
- *keyboard button*: events from the keyboard of the human operator.

At **initialization**, the following knowledge is **assumed** to correspond to the real environment of the robot:

- the robot is *inside* a room.
- the room has a *rectangular* shape as in the figure, with unknown lengths of the walls.
- the room has *one door*, *wide enough* to let the robot pass through.

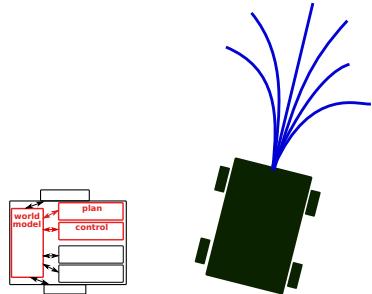


Figure 6.31: A possible *control* strategy for the escape-from-room capability in Fig. 6.32: to select from between a discrete set of pre-computed open loop trajectories, each corresponding to one particular time-invariant input at the actuators.

The **control** part of the task can as simple as making a selection between various “open loop” motion trajectories, Fig. 6.31:

- when a set of constant speeds is applied to each wheel, the result is a set of known trajectories of the robot in the near future. These trajectories can be obtained from a model only, or can be identified on the real robot.
- the sparsity and density of these trajectories can be chosen, in a plan-directed way, to reduce the computational requirements to a level that does not allow to separate between trajectories that are closer together than the sensing resolution, or than the tolerance allowed in the task specification.
- time and space horizons can be chosen for the trajectories, again in a plan-directed way.
- the *control* action can then be as simple as *selecting* the best open loop trajectory and apply the corresponding wheel constant velocities.

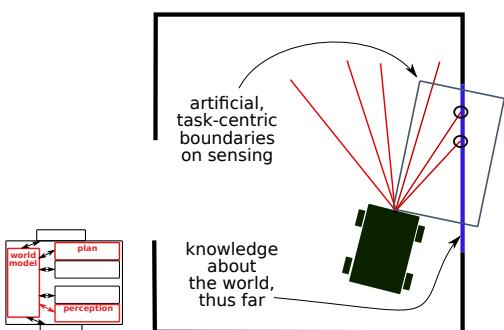


Figure 6.32: A possible *perception* strategy for the escape-from-room capability in Fig. 6.32.

The **perception** part does not need all the data provided by the distance sensor, since it could use the following algorithm (Fig. 6.32):

- select a *region of interest* (the grey box in Fig. 6.32) 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.

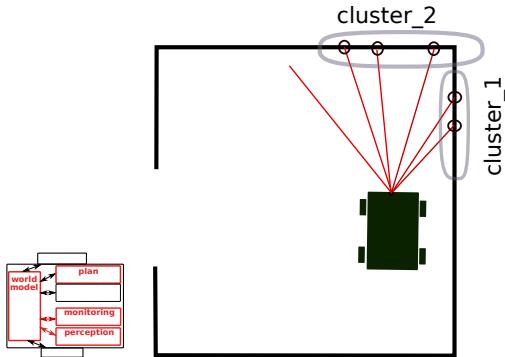


Figure 6.33: The *monitoring* strategy for the escape-from-room capability in Fig. 6.32 must follow the *wall on the right* (which is needed for the motion control) and look out for a *wall in front*.

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.

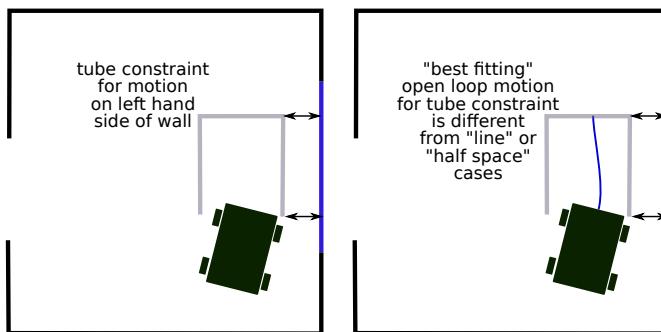


Figure 6.34: Left: a possible *motion specification*, in which a “tube” constrains the allowed motions, but does not command an explicit motion trajectory. Right: a corresponding *control choice*.

The **motion specification** needed in the **control** can be as simple as the “tubes” in Fig. 6.34:

- 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. 6.31 that fits best, according to a task-specific metric. **Alternatives** for the **control** design are depicted in Fig. 6.35.

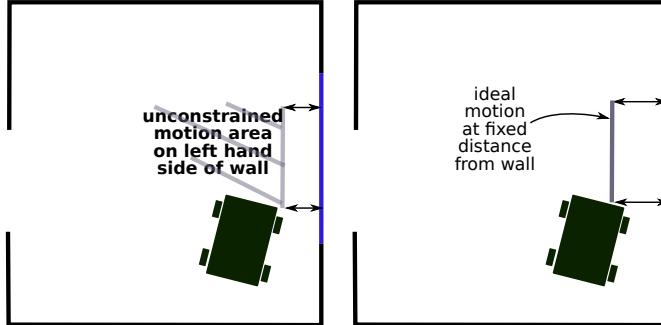


Figure 6.35: Two alternative controller approaches for the motion specification in Fig. 6.34. Left: a open half space, to the left of the wall. Right: a line trajectory at a specified distance from the wall.

The **world model** entities and relations needed in the **plan** are as follows:

- room has four **corners**, one **door**, and five **walls**.
- each **wall** is represented by two **nodes**, that is, a point in the 2D plane.
- the **robot** has a **pose** with respect to the **room** features.

This gives rise to the topology of Fig. 6.36. 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.

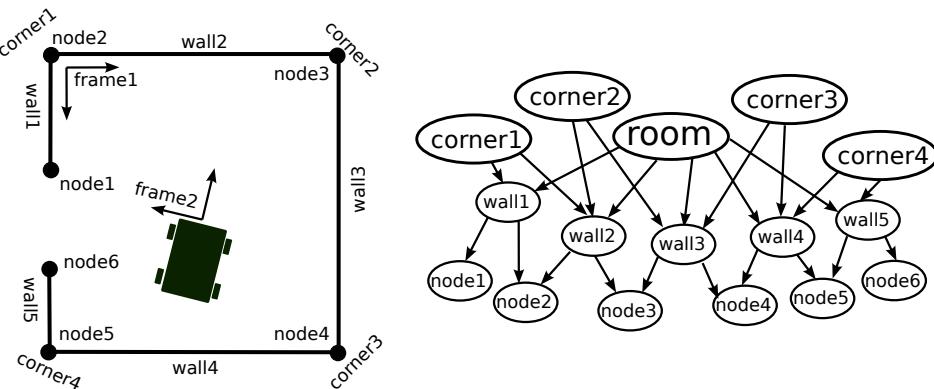


Figure 6.36: Right: topology of *room* model entities (“data structures”) and relations (**has-a**, and **connects**). Left: geometrical properties of all entities. The robot’s pose in the room can be represented numerically by position coordinates of the frame attached to the robot with respect to the frame attached to the room.

6.15.5 Platform Independent/Specific Model, Domain-Specific Language

This Section gives examples of *domain-specific languages* that bring the specification of tasks (as introduced in the previous Sections) in line with the terminology used in particular application or technology domains.

(TODO: explain the relations between [PIM](#), [PSM](#), [DSL](#))

6.15.6 Task ontology and its standardization

Modelling all relevant task-level compositions is a huge undertaking, but will hopefully and eventually result in a (*standardized!*) collection of domain/application specific **ontologies**. This undertaking is not a main focus of this document, but all of the document's contents serves as a foundation for that undertaking. The good news in the short-term is that even the “poor” mereological top of this structure as introduced here is very useful to support discussions between human developers, not in the least to make the scope of their developments explicit.

Sooner or later, the **stakeholders** in the **robotics domain** must **agree on a *standardized ontology*** for all these terms, because that is the **only** way to realise a vendor-neutral digital platform that can serve as the basis for composable innovation.(After more than half a century of robotics industry, there are still no significant results in the direction of semantic standardization of the field.) The second added value of creating a widely supported domain ontology with a [hypernym–hyponym hierarchical structure](#) is for the robot controller software to exploit: **only** when the mentioned ontological information is available, at **runtime** and in **formal representations**, one can expect robots **to reason** about their actions on the “most appropriate” level of abstraction, **to assess** whether what they are doing corresponds to what they are supposed to do in their current **task**, and **to adapt** their action plan accordingly.

6.15.7 Semantic mobile robot motion primitives

Abstracting a bit from the concrete examples in the previous Section, the following semantic motions could form the basis of a mobile robot's “**platform** motion stack capabilities”:

- *start-to-cruise* (and its *inverse*, *cruise-to-stop*): how to get the robot start its motion and reach a “cruising” motion behaviour, when all it has to do is to drive “straight ahead” within its current lane, and that lane continues till “far” beyond the dynamic bandwidth of the robot.
- *cruise through tubular area*: the *world model* for this semantic motion has landmarks on the robot are geometrically constrained by a “tubular” area in the Cartesian space.
- *overtake obstacle during cruise*: this is a composite cruising task, with lane-changing motion capabilities added. Reference [59] 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, . . .).

6.15.8 Semantic robot arm motion primitives

The mobile robot example in the previous Sections is conceptually the easiest to grasp, because the world is mostly flat, *and* the shape of the robot is mostly constant. For arm-based robotic systems, the full 3D Cartesian space and the full n D joint space are to be taken into account, but at the mereo-topological level of abstraction, very similar semantic motion primitives can be defined. Figure 6.37 sketches some simple examples, with two levels of resolution in representing the mentioned configuration spaces.

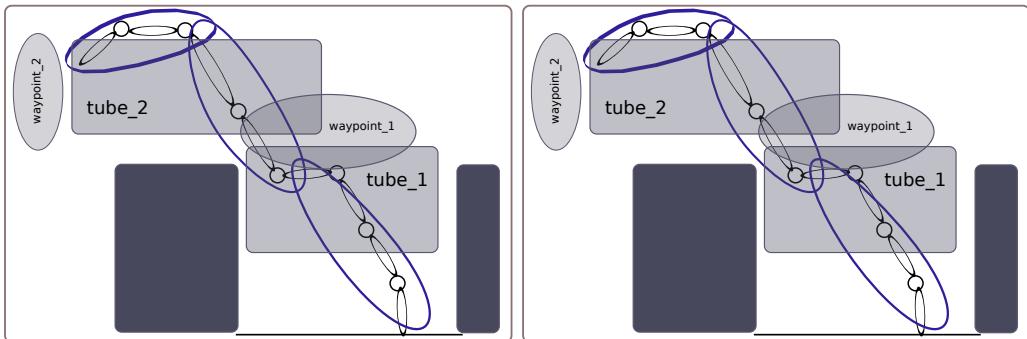


Figure 6.37: Model of a “high” (left) and a “low” (right, in blue) kinematic resolution motion plan for a serial robot arm during a sequence of tubular motion between obstacles.

Chapter 7

Meta models for continuous and discrete control

Control is that sub-system in a system that is responsible for **realising the behaviour** specified in the **tasks** that are given to the system. Control is always a **trade-off**, between:

- what the **tasks** *require*,
- what the **hardware** and **software** can *do*,
and
- what the **objects** in the **environment** can *afford*.

This Chapter introduces the **mereo-topological** meta model of control (that is, its **feedback**, **feedforward**, **adaptive**, **predictive**, **preview** and **cognitive** components), together with the *natural structures* and *best practices*. It focuses on two of the three control *domains*:

- **continuous** domain: the parameters that are being controlled come from continuous configuration spaces, for example, *space* or *effort*, most often as a function of *time*. Only continuous control activities **put energy into the system**, by deciding what the “best” **actuator** signals are to bring the world from its **actual** state to its **desired** state.
- **discrete** domain: every continuous control activity also has **to monitor** a number of **constraints**, whose violation indicates that it might be time to switch to another continuous controller. The algorithms for the continuous control, the monitoring, *and* the switch decision making, are all specified in a **plan**. *Coordination* is an essential part of every plan (Secs 2.6, 2.7, and 2.8).

Other terms for discrete control are: **supervisory** control, **discrete event** control, or **hybrid** control.

The third control domain is the **symbolic** domain. It is the “highest” in the control hierarchy, and its decision making consists of the **reasoning** that determines which choices to make in the configuration of the discrete and continuous control domains. Many of the this document’s other Chapters contribute bits and pieces of the information required for that symbolic control.

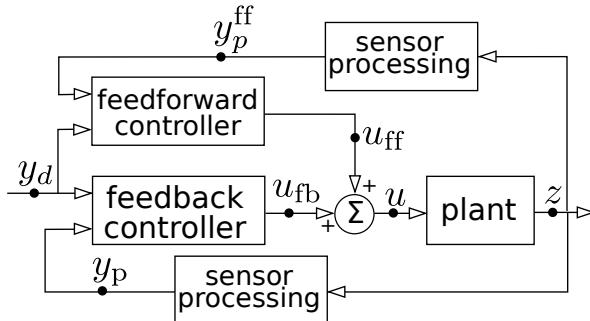


Figure 7.1: Both feedback and feedforward control contribute to the actuation signal u that is given to the plant. Both (can) work with the raw and/or the *processed* versions of the measurements of the current state z of the plant.

7.1 Mereology: feedback, feedforward, predictive, adaptive, preview, cognitive

Figure 7.1 shows the simplest case of a control system, that is, with only **feedback** and **feedforward** components; Figs 7.2—7.4 show the more complicated cases, that can all change the *configuration* of the feedback and feedforward control components in complementary ways. A **controller** (“control diagram”, “control scheme”) is a **composition** of the following contributions to the actuator input signals:¹

- **setpoint**: the specification of how the world *should* look like. The specification can be as simple as the **setpoint** of one single parameter in a world model, (**SISO**, *Single-input single-output*), or of a large array of such world model parameters (**MIMO**, *Multiple-inputs multiple-outputs*).
- **error**: the difference between the *actually measured* value(s) of the world model parameter(s) and the desired setpoint. (As far as that actual state can be **estimated** (“**observed**”) from the raw sensor information.)
- **feedback**: this function generates (that part of) the actuator signals based on the *error only*.
- **feedforward**: this function computes actuator signals based on (i) the actual state of the world, (ii) a **model of how the system is expected to react** to potential **instantaneous** actuator signals (in other words, a **simulation** of the system), and (iii) the **smallest error** over all the simulated actuator signals.
- **predictive**: this extends feedforward to a particular **time horizon** into the future, Fig. 7.2. Often, one also chooses **to constraint** the search space of actuator signals to consider in the optimization,

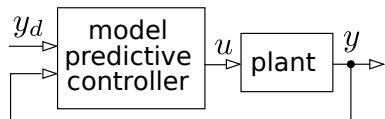


Figure 7.2: Predictive controller.

- **adaptive**: this function does not generate actuator signals itself, but it changes the control **behaviour** indirectly, by adapting one or more **parameters in the models** of

¹The terms in this list have been used in somewhat different contexts, and somewhat different interpretations, in various domains of research. more in particular, in the strongly complementary but not fully overlapping versions of *control* in engineering and in human experimental psychology. Seminal publications in the latter are [79, 94]. Researchers in the (now almost deserted) domain of **cybernetics** have tried to bridge the gaps, [132, 133].

the feedback, feedforward and/or predictive functions, [24, 148]. The basis of adaptation is the **observed history** of control signals and system states.

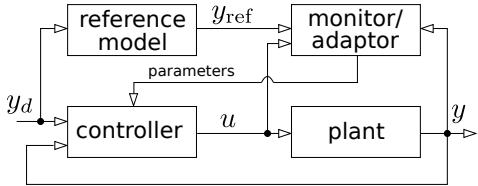


Figure 7.3: Model-reference adaptive controller.

- **preview:** this function changes the **structure** of the control model, because it adds a model of (part of) the control to be realised in the **next task** in the sequence of task specifications. Sheridan [132] introduced this approach in a way that has led naturally to (hybrid) **model-predictive control**.
- **cognitive:** the perception information is **interpreted**² in the context of the **situation** in which the robot finds itself, and **reasoning** is needed to make decisions about which control action to take.
- **schedule:** this is the model that represents *when* all of the above functions must be executed in the control activity. (See also Sec. 7.4.2.) Schedules can be computed in advance, online, or in hard realtime.

7.1.1 Policy: adaptive and preview control

Parameterizing the feedback, feedforward, predictive, adaptive and preview functions provides for higher configurability in the control system, in particular to reuse the same implementations in larger ranges of operational conditions.

Typical examples of **adaptive control** change one parameter in the feedback or feedforward models, while the controller is running:

- many feedback loops first perform a first-order **low-pass filtering** on the measurements or on the error signals, and the **gain** of that filter can be adapted to the observed **noise level** of the signals.
- relevant feedforward parameters are the values of the (one-dimensional) **inertia**, damping/**friction**, or **elasticity** in the mechanical part of the system. The value of that parameter in the feedforward model can be adapted on-line by monitoring the *trend* of the feedback error: if the error is “systematically positive”, the parameter can be decreased, and when the error is “systematically negative” it can be increased. Only one of the three mentioned mechanical parameters can/should be adapted in any one-dimensional adaptive controller. For example, if one tries to compensate for friction and inertia by changing one only feedforward parameter, one can end up in a situation where neither of both values is compensated well.

An example of **preview control** is that, when moving a robot hand towards a door handle, one can already start controlling the opening of the hand, as well as its orientation with respect to the door, starting from a certain parameterized distance to the door. The result

²In the cognitive sciences, the term “*precognitive*” control is used [79, 94, 102], which is semantically compatible with the “cognitive” terminology in this Chapter: the “pre” prefix in the former context refers to “before getting to the level of” cognitive understanding.

could be that the hand is already better aligned when it is going to have to open the door in the next sub-task.

A preview controller *can* improve the execution time of a task by “looking ahead”, but (i) the performance improvement may be hard to guarantee, and (ii) it definitely comes with extra computational costs.

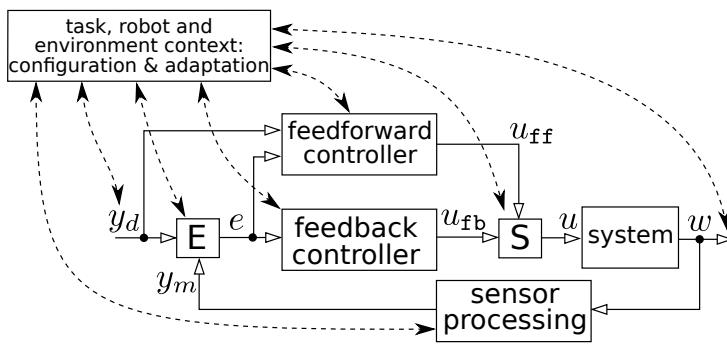


Figure 7.4: The “magic numbers” in feedback and feedforward loops can only be given their appropriate values in the full context of the application. Control loop sketches as in Fig. 7.1 (suggest to) consider only the *instantaneous error* between desired y_d and measured state value y_m . This figure depicts the **context-dependence** more explicitly.

7.1.2 Policy: cognitive control

(TODO: explain how cognitive control makes uses, in explicit models, of (i) the *situation*, (ii) the *association hierarchies* in control, and (ii) *empathic* prediction of the behaviours of other agents.)

7.2 Mechanism: linear control — PID and pole placement

Introductory textbooks on *control theory* (in both the “analog” and the (discrete) “state space” versions) most often work *assume* that (i) the *system to be controlled* has an *open loop dynamical model* that is *linear*, and (ii) the *controller itself* behaves as a linear system. That means that the *superposition principle* is assumed to hold: any linear combination of control signals gives rise to the linear combination of the reactions of the controlled system to the individual control signals. No real-world system satisfies this assumption, for a large number of *different reasons*.

7.2.1 Linear control properties

Linear control is a **popular approach** for many reasons, the major two being:

- *simplicity*: all operations on the *mereological model* of a controller are *linear functions*:
 - the *error* is the *subtraction* of the *setpoint* and the *measured value*.
 - the *feedback* and *feedforward* operations are matrix operations on the *error*.
- *relevance*: many real-world systems can be *given* a *dynamical model* that is linear for “small” and/or “slow” deviations from the “zero” configuration of the system model.

PID control (Proportional, Integral, Derivative; for **SISO** systems) and **pole placement** (for **MIMO** systems) are two mainstream approaches in analog and discrete control design. The **methodologies to design** such controllers are “**optimization**”-based (both **offline**

and online, with the latter being a major ambition of this document): the methodology selects the control parameters as the “best” trade-offs that can be made between:

- **gain**: how fast is the error controlled away?
- **phase margin**: how sure are we that the controller does not cause more problems than it solves? For example, by putting “unlimited” amounts of **energy** into the controlled system.

And this trade-off should work for **any possible combination** of setpoints and input signals. In other words, to keep the error value as small as desired, under all circumstances, and to keep the deviation of the signal *shape* of the controlled system as close as desired (in the *time domain*) to the signal shape of the setpoint.

7.2.2 The role of PID and linear control in robotics

A PID controller deals with **one signal only**. In other words, it represents a **SISO** (Single-input single-output) control system. In any robot of somewhat realistic complexity, the forces generated by each motor influence all “error” signals. In other words, these are **MIMO** (Multiple-inputs multiple-outputs) control system. The good news is that the knowledge exists **to decouple** all controlled degrees of freedom, via a **dynamics solver** that uses the robot’s **kinematic chain** model. Such a solver computes the **feedforward** part, and this yields *not* a linear system but a *linearized* system:

- when computing the control efforts *in the right order*, each of them requires solving only linear equations.
- *superposition* holds only locally, around the current linearization point.

The result is something that looks a lot like linear control but isn’t. Not in the least because, despite the decoupling, the inertia that is felt by each motor changes with the configuration of the robot, and this introduces an (often significant) non-linearity. *Parking a car* is an illustration of such (extreme) non-linear configuration dependency: at any moment in time, the car can just *not* move (itself!) in a direction orthogonal to the orientations of its wheels, Fig. 7.5.

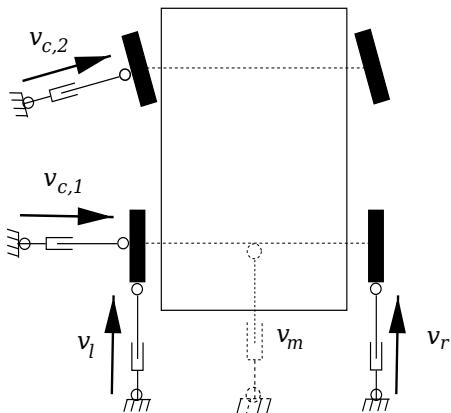


Figure 7.5: A mobile robot, or a car, are major examples of systems that can not be controlled by one single linear controller: there is no way to move the car in the directions orthogonal to its wheel orientations, and that direction changes non-linearly with the steering angle.

Daily experience tells us that the real controller must be a **hybrid controller**: it solves the control problem of parallel parking as a sequence of *back-and-forth* tasks, with feedforward actions on the steering and driving wheels, with monitors to stop the selected feedforward

action, and continuing the process until the car is “deeply enough” into the parking spot and “aligned enough” with the curb of the road.

7.2.3 Situations where linear control is insufficient

The car parking situation is one of the many cases where simple linear control fails. Here is a list of other such situations:

- *errors in non-homogeneous space*: the *subtraction* of the coordinates of two states in the control domain of the system does not represent the *difference* in the physical behaviour of the system in these two states. One major case where this occurs are spaces without a natural “zero”, for example, when representing relative position and motion of physical bodies.

This non-homogeneity is a common error in controller design for robotics systems, because the composition of translational and angular displacement (of rigid bodies or lines) is not a linear operation, irrespective of which coordinate representation one uses to model the system dynamics, [74, 75, 76, 88, 89].

- *hysteresis*: the dynamical response in one direction of its state space is different than in the opposite direction.

Friction is a major mechanical phenomenon where hysteresis occurs.

- *dead band*: the system has a “zero” dynamical response in a certain interval of its state space, so the system will not always react immediately to a change of the control signal. Dead band and hysteresis are *different phenomena*.

Backlash (or, *play*) in a *gear train* or *transmission*, is a major mechanical phenomenon with a dead band.

- *saturation*: after a sufficient increase in a causal force, no further increase in the resultant effect is possible. For example, for every *mechanical spring*, the linear *Hooke’s law* only holds in a finite domain of deformation of the spring. In one direction, adding more force does not deform the spring anymore, because it is fully compressed, like in the downward case of a *pogo stick*; in the extension direction, *plastic deformation* occurs when the extension force becomes larger than the *yield strength* of the spring.

- *nonholonomic constraint*: this is the type of motion constraint that occurs in mobile robots (or *cars*, or bicycles), with an *Ackermann steering* mechanism, or something equivalent. The car can not move *instantaneously* in a direction along the wheel axles, hence any linear feedback on a “control error” in that direction will not have an effort to reduce that error. This is sometimes called the *parallel parking problem*.

- *bounce* or *chatter*: the dynamics of the controlled system is such that it is very difficult to stay in its “zero” position, and feedback control makes the error jump between a small positive and a small negative value.

The chatter phenomenon can have many causes, such as above-mentioned backlash, finite *sensor resolution*, or the below-mentioned discretization.

- *discretization error*: due to the finite resolution of a sensor, and/or to numerical *round-off errors*, the value of the controlled setpoint can not be reached exactly, so there will always remain an error, and hence a linear control action.

- *asymmetry* in the system dynamics. For example, gravity always works in the same direction, so the control effort needed to close an error in the upwards direction is significantly larger than that in the downward direction, because in the latter case gravity acts as a free extra actuator.

7.3 Information associations in control: mechanics and tasks

This document introduces two **association hierarchies** (one in the **mechanical domain**, Fig. 7.6, and one in the **task domain**, Fig. 7.7) to structure the knowledge representations—at various **levels of abstraction**—that underly the *decision making* in all *control* activities: what are the “best” *relations*, or (*associations*), to use in the control algorithms, and what are their “best” magic numbers. These two *control* hierarchies complement the single *perception* association hierarchy, Fig. 8.1.

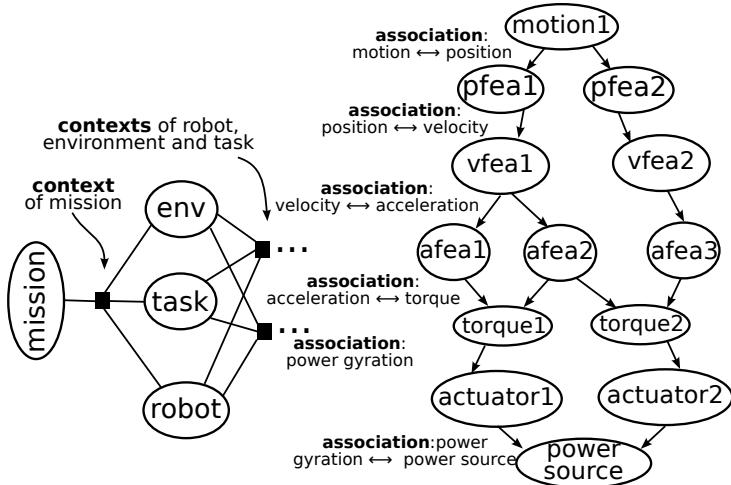


Figure 7.6: The **mechanical domain** hierarchy to bring structure in the *relations* between various levels of control in robotic systems, from energy to position.

7.3.1 Mechanical domain control hierarchy

The left-hand side of Fig. 7.6 depicts the **natural** hierarchy of mechanics:

- **power gyration:** *energy* from various sources (chemical, electrical (mains, batteries), hydraulic, pneumatic,...) is transformed in mechanical *torque* or *force* in an *actuator*. This conversion always takes place with an *efficiency* lower than 100%.
- **acceleration \leftrightarrow torque:** *Newtonian dynamics* provides all entities and relations behind the coupling between force and/or torque and acceleration of a (rigid) body.
- **velocity \leftrightarrow acceleration:** the former is the integral over time of the latter. This is a mathematical relation, without “disturbances”, from the real world.
- **position \leftrightarrow velocity:** the former is the integral over time of the latter. This is a mathematical relation, without “disturbances”, from the real world.
- **motion \leftrightarrow position:** the former is the sequence over time of (changes in) the latter. This is a mathematical relation, without “disturbances”, from the real world.

The ideal version of the couplings between force and torque on the one hand, and position, velocity and acceleration on the other hand, is disturbed by *frictional losses*, physical *motion constraints*, *backlash*, or *elasticity* of the mechanical bodies involved. The container term for all of the above is called *impedance*.

7.3.2 Task domain control hierarchy

The right-hand side of Fig. 7.7 depicts the **artificial** hierarchy induced by the scope of the **world model** for which a control activity is designed:

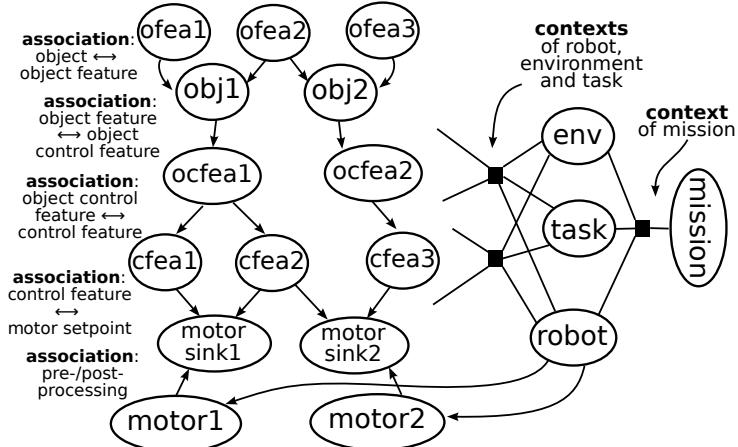


Figure 7.7: The **task domain** hierarchy to bring structure in the *relations* between various levels of control in robotic systems. This is the second of the two control complements of the perception association hierarchy of Fig. 8.1.

- pre-/post-processing:
- control feature \leftrightarrow motor setpoint:
- object control feature \leftrightarrow control feature:
- object feature \leftrightarrow object control feature:
- object \leftrightarrow object feature:

The association hierarchy presents a **control-centric** view on an application; Sec. 6.2.3 presents a complementary **added value-centric** view. An **optimal** control for two or more levels in the hierarchy has been realised first in the chemical process industry [47, 119], because (i) the time scales are amongst the slowest in the industry (compared to, say, motion control or energy generation and distribution), and (ii) huge economic benefits can result from often small changes in the working conditions of a large-scale **chemical process**.

The **manufacturing industry** soon followed. One which has many (conceptual) links with this Section is that of [87], that uses the term “**5C levels**”³ *Connection, Conversion, Cyber, Cognition and Configuration*, to represent the relations between the control of, respectively:

- sensors and actuators, with a focus on the *data quality*.
- devices, machines, robots,... made out of sensors and actuators, with a focus on the *motion control* performance of their tools.
- peer-to-peer networks of such devices, with a focus on the *logistics control* of keeping the individual machines busy in the best possible way.
- production with such networks, focusing on the *operational* added value optimization, within one company.
- configuration of such networks, focusing on the *strategic* added value optimization for the company, in a reaction or anticipation to changes in the market.

7.4 Mechanism: state and system dynamics relations

Any controller (or **control diagram**) is the composition of only four entities (Fig. 7.8) that conform-to the **algorithm** meta model:

- **signal**: the **data** that “flows” between two **function blocks**, that is, the input and output parameters of the functions. The numerical representation of a signal is a traditional

³This document uses the “5C” label for a very different purpose, namely that of the different **roles of components** in a system.

data structure.

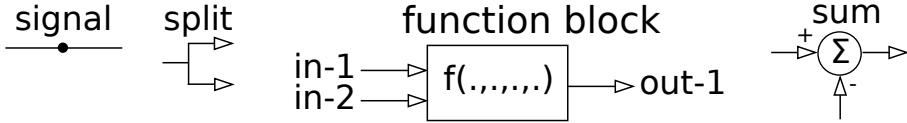


Figure 7.8: The four entities in the controller meta model that conform to the `algorithm` meta model: the `data` blocks `signal` and `split`, and the `function` blocks `function-block` and `sum` block.

- `signal` `split`: special case of a “`signal`” that appears in quasi every control diagram, and that represents the fact that the *same* signal is used as inputs to more than one `function` block. The `split` has a *direction*: there is only one connection to an output of the `function` block that creates the `signal`, but the `signal` can be the input of more than one `function` block.
- `function block`: this is a *pure function*, with one or more `signals` as inputs and one or more `signals` as outputs.
- `sum` block: this is a special case of a `function` block, that appears in quasi every control diagram, and that gives as its output the sum of all its inputs, each possibly with a “`-1`” sign inversion.

The following concepts are common to all controllers, so the meta model adds them as first-class modelling primitives:

- `setpoint`: an entry point of a controller. Its meaning in the context of an application is that this signal gives the *desired* value of the `plant` state. The `setpoint` has no further connections at its own “input” side, at least not inside the scope of the current control diagram. It is clearly the `port` where this control diagram `block` must be `connected` to other system blocks, Sec. 1.8.6.
- `measurement`: another 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.⁴
- `actuation`: an exit point of a `control-diagram`, which provides information for the actuators that can change the `state` of the `plant`.
- `state`: a composition of signals from a control diagram, that make the following sense to the application: together with the system model, the `state` of the system at one moment in time, is the necessary and sufficient information to predict the future behaviour of the system.
- `sample-instant`: the `timestamp` that represents the time of one particular execution of the `control-diagram`.
- `sample-period` (and its inverse, the `sample-frequency`): the 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`. An event is a `discrete` variable, nn

⁴“Difference” and “magnitude” are written with quotation marks, because it is not always the case that the state space of the controlled system is a vector space (where “subtraction” and “error” are well defined concepts) with an invariant metric to define “magnitude” unambiguously.

contrast to all of the above, which are continuous by nature, but which are, of course, *discretized* for computational purposes.

7.4.1 Composition

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

- **plant**, or **system**: this is the part of the real world whose **state** the **control-diagram** tries to influence, and with which it interacts via **sensors** and **actuators**. These convert physical values into digital **signals**, and back.
 - **feedback loop**: a **function block** that takes **setpoint** and **measurement signals** as inputs, and computes a **signal** that can be used by an **actuator**. The topology of **feedback** is a “loop” because the **plant** couples the **measurement** and the **actuation**.
 - **feedforward chain**: a **function block** that takes **setpoint** signals as inputs, and uses a *mathematical model* of the **plant** dynamics to compute a **signal** that can be used by an **actuator**. It is *not* a loop, but a serial composition of function blocks.
- Of course, *adaptation* does bring a loop structure, but adaptation is a **higher-order** composition relation.
- **sensor**: a **function block** that gives **signal** values to some physical values of the **plant**.
 - **actuator**: a **function block** that converts **signal** values in the controller to physical values of the **plant**.
 - **adapter**: a (higher-order) **function block** that takes a **state** of the **control-diagram** as input, and computes a **signal** that another **function block** in the **control-diagram** can use to change the value of one or more of the parameters it uses in its computations.
 - **monitor**: a (higher-order) **function block** that takes a **state** of the **controldiagram** as input, and computes an **event** for the application software that uses the controller. That event is not used in the **continuous controldiagram** itself, but it *can* give rise to a change in one or more parts in the **control-diagram**, via the **discrete control** part of the system.
 - **control-diagram**: this is the **composition** of all of the above, which conforms to the constraints of the controller meta model, discussed below. It has **one trigger** entry point, to execute all computations inside the diagram. To this end, every **control-diagram** contains a data structure to represent its **schedule**. In other words, the **controldiagram** is the model of the function that (i) takes the **output state** of the **plant** as one of its arguments, (ii) takes the *desired state* of the **plant** as its **setpoint** argument, and (iii) from them, computes the **input** that should be applied to the **plant** to make the latter evolve towards the desired **state**.
 - **delay**, or **buffer**: signals with different **timestamps** are sometimes used as arguments in the same function. Hence, the controller function must provide a mechanism to store signals for a finite amount of time, and these are the **buffers**. They contain functions and data structures.

In general, a **control-diagram** model is a **cyclic graph**, because of the presence of feedback loops. However, there is one very important property of that graph: each cycle corresponds to a **different sample-time**, hence, time is the appropriate entity to **unroll** the cycle into an **event loop**,

Performance properties of control behaviour (that (only) show up during the **execution**

of the corresponding event loop implementation) are:

- **latency**: the difference between the desired timestamp of the execution, and the actual one.
- **loop-time**: the duration of the execution of once cycle of a **controldiagram**. Ideally, that duration is zero, because that would mean that the actuation signals are applied immediately after the sensor have been read. Of course, this is not feasible in practice. The best one can achieve is that the **looptime** is “sufficiently smaller” than the smallest time constant in the system that one wants the controller to react to. Of course, these **magic numbers** are fully dependent on the application context.
- **jitter**: the statistics of the **latency** and **loop-time** values over a certain period. Especially the worst values are important, because they give rise to the largest **disturbances** that the execution of the control introduces to the desired controlled system behaviour.

7.4.2 Time-triggered and event-triggered control schedules

Two major policies exist as **schedule** for the execution of a control activity:

- **time-triggered**: the control activity is triggered after a (software configurable) period of *time* has elapsed. This approach fits well to cases where explicit dynamical models of the system under control are available, in the form of **ordinary differential equations**, because they have time as the explicit independent parameter.
- **event-triggered**: the control activity is triggered after a particular *event* has occurred. For example, the value of an important parameter in the system has reached a (software configurable) threshold.

Both are often applied at the same time in one single control activity, that decides which of the above-mentioned functions to execute each time it is being triggered. Strictly speaking, time-triggered control is a special case of event-triggered control, with the event being that the time has reached a specific value.

A **guarded motion** is a primary example of an hybrid time- and event-triggered task execution control:

- at fixed sample *times*, the controller reads out the sensors, and uses this data, in the *same sample period*, for (i) a simple feedforward and/or feedback control loop, and (ii) one or more **monitor functions**, to check whether a set of *constraint violation* (or, equivalently, *condition satisfaction*) relations have become true.
- when the latter constraint violation indeed occurs, this gives rise to the *event* that triggers the “discrete” feedback function to be executed, namely one that changes the functions running in the time-triggered loop above.

This execution architecture is a simple instance of the more generic **cascaded control loop** meta model.

7.4.3 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. 7.10. 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. More specifically, this **artificial hierarchy** in **cascaded control loops** has the following parts:

- **power inverter**, with a typical⁵ time constant of 1/100.000th of a second.
- **electrical motor**: transforms electrical energy and power into **mechanical** work and **torque**, via mechanisms such as **field oriented control**, with a typical time constant of 1/10.000th of a second.
- **mechanical acceleration**: the generated torque causes acceleration of the attached mass; the time scales required in robotic applications lie around 1/1000th of a second and up.
- **mechanical velocity**: time-domain integration of acceleration results in velocity; again, an order of magnitude less in required time scale is order.
- **mechanical position**: further integration into position is the final step in the mechanical level of abstraction, with again an order of magnitude lower time scale.

The *battery* is not part of the cascade hierarchy because:

- its time scale is determined by the *chemical dynamics* of conversion of chemical energy into electrical energy, and this is orders of magnitude slower than the dynamics inside the electrical DC-to-AC energy conversion.
- its energy production need not follow the time scales of the control, since that is the responsibility of the **DC-to-AC power inverter**.

For the sake of concreteness, the example of a *battery-driven mobile robots* (e.g, Fig. 7.9) is used to illustrate the structure in the various energy transformations that occur in a robotic system:

1. **battery** as an electrical **energy source**: it can provide electrical energy (current and voltage) via well-known impedance relations, and with constraints on maximum and minimum power, voltage levels, temperature dynamics, etc.
2. **energy transformation** between AC or DC **electrical energy** into **mechanical torque** in (a)synchronous motors: the battery energy is transformed into mechanical energy via impedance relations of multi-phase, multi-pole motor models, and with constraint curves for torque, speed and efficiency.
3. **energy transformation** via a **mechanical transmission** from the electrical motor to the mechanical joint: the joint “consumes” not only the electrically generated torque for its own motion, but the dynamics of the whole chain are coupled in. A transmission can, itself, introduce extra (mechanical, thermal,...) dynamics, for example, via friction and elasticity, heat generation,...
4. **energy transformation** between the **joints and kinematic chain** to produce **Cartesian space motion** of end effectors (and other link attachment points): these are the “hybrid dynamics” introduced in Sec. 4.7.
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.

⁵At least for electric motor actuators.

The model semantics speak about “energy transformation”, because that is the more declarative and non-instantaneous way of expressing the relations, instead of the imperative and instantaneous terms “force” and “velocity”. (This is also the difference in approach represented by the (equivalent!) [Newton-Euler](#) and [Euler-Lagrange](#) theories of dynamics.) Instantaneously, the energy is indeed transmitted via *forces* (Fig. 7.9), and this is a very important fact that *must* be represented faithfully in all models (and hence also software). Indeed, different sources of force can be *added* together instantaneously, which is a major instantiation of the **composability** ambition of the modelling efforts. Position-based relations (positions and poses and their time derivatives), however, are *not* composable consistently in an additive way.

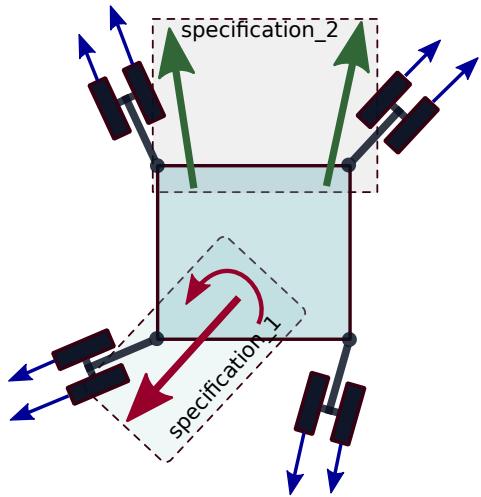


Figure 7.9: Transmissions of forces between actuated wheels (the blue “traction vectors” along the wheels’ rolling direction) and the (red) force and moment on the platform the wheels are attached to. (The green duo of forces represent an alternative way to specify desired/actual inputs to the robot platform.) This relationship model is composable because the combined effect of all actuator forces are physically realised by simple vector addition.

- **gyration**: transformation of chemical, electrical, hydraulic,... energy into *mechanical* energy.
- **transformation**: between the various types of *mechanical* energy, linking torque, acceleration, velocity, position and trajectory.

7.4.4 Composition constraints

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.

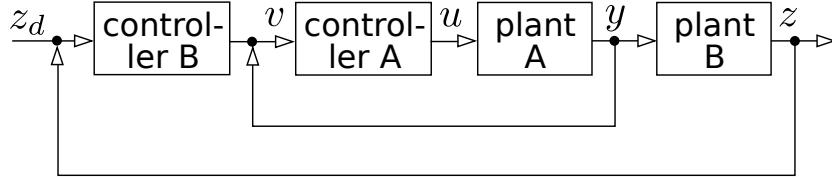


Figure 7.10: Cascaded control loops.

7.5 Mechanism: optimal control specification

The control meta model as introduced above, still allows multiply ways of *how* the control computations are realised. **Constrained optimization** has become a popular approach, because:

- it forces designers **to formalize explicitly** what their design objectives really are.
- it is a perfect fit in the **Hybrid Constrained Optimization Problem** (HCOP) formulation of robotic tasks, introduced in Sec. 6.10. That formulation is *even more generic*, because it integrates the **symbolic**, **discrete** and **continuous** aspects of system behaviour design. The current Chapter deals with only the continuous part of this integrated approach.
- **monitoring** of the values of the objective functions and the constraints in this continuous part is the link from the continuous control part to the **discrete control** part. The link in the other direction comes from the **plan**, where the events are one aspect of the information used to make the decisions to switch continuous control behaviour.
- **reasoning** on symbolically represented knowledge is the third **symbolic control** part. The “feedback”, “feedforward” and “adaptation” at that level correspond to deriving which symbolic relations are relevant to which decisions, and in which way exactly they influence the decision making.

In other words, it brings in the **contextual information** that helps to formulate the *application-, environment- and platform-specific* objective functions and constraints for the continuous and discrete controllers.

Figure 7.11 depicts the generic formalisation (“meta meta model”) of a continuous-domain constrained optimization problem with which to compute a control signal:

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

Figure 7.11: Generic formulation of a *constrained optimization problem* (COP).

The **domain** fills in the *types* for f , X , q for a particular “robot”, and a particular *type* of solver. The **application** then adds choices for the *parameter values* for f , X, \dots , and the

concrete solver and monitor *implementations*.

(TODO: concrete examples.)

7.5.1 Policy: “good enough”, or satisficing control

The *optimization* approach to design controllers (*linear* or not) has its advantages; that is what optimization *is* meant for, after all. But there is *always* an alternative for optimization, namely the *satisficing* approach. Instead of finding the “optimal” control behaviour, its goal is to realise control behaviour that is “**good enough**”: as soon as the controlled system is *within tolerance*, the control actions can stop. Here are examples where humans apply satisficing control:

- *driving in traffic*: humans drive their cars and bikes with a combination of two rather decoupled controllers, one to regulate the driving *speed*, and the other regulating the *expected* motion trajectory in the “near future”. The error for the latter is not just a function of the motion itself, but also of the targets in the environment that are to be taken into account. And for both control sub-problems, it is clear that large margins can be tolerated, in which no control actions are required until the car motion is (expected) to go outside those margins again.
- *overcoming friction*: an actuator always has to build up some force to overcome the *static friction*, so the relation between force and motion is not really linear but *affine*: the mechanical system under control has a *bias*, or an *offset from “zero”*.

Nevertheless, it is rather easy to control this type of friction: one *knows* that sooner or later the system will start moving if one ramps up the actuator force.

So, even when the *specification* is *formulated* as an optimization, the algorithm that *solves* the specification can decide to stop *before* that optimum has been reached. In other words, the optimization outputs a control behaviour that sets the *direction* in which to improve the solution, together with *monitors* that check the conditions that indicate that the actual control is “good enough” for the purpose of the task.

7.5.2 From PID and pole placement to optimal control

This Section introduces a non-exhaustive list of assumptions that the PID or pole placement approaches make, and for which it is often not very difficult to find alternatives. A major reason for the sustained popularity of the traditional PID-centric control approach is **human inertia** against change, culminating in the slow update of control theory curriculum at universities. This document advocates⁶ the transition from the “offline optimized” control paradigm op PID and pole placement to the “online optimized and monitored” control paradigm op optimal control.

- control action scales linearly with magnitude and sign of error.
This leads to the controlled system reacting differently under large errors and smaller errors.
- every sample time, the full controller is executed, and the horizon over which the control reacts is very short. Typically, that horizon is only one time sample deep, but it is possible to extend the state of the system with its states at different time instants.

⁶Without forgetting that there are indeed still many simple control challenges for which the traditional approaches are impossible to beat.

Indeed, some signals do not change significantly during that short period. More in particular, the computation of the “error”, and transforming that error into a feedback actuator control signal, can often be done at much lower time rates than updating the feedforward actuator signal; especially for mobile robots, the same feedforward actuator signal can be held for a long time and feedback must only be applied when the robot runs a risk of going out of its “lane” somewhere in the future.

- there is no semantic difference between a *positive* and a *negative* error.

In mobile or flying robot applications, a symmetric reaction to an error implies a loss of actuation energy: when the robot is moving a bit too fast, there is seldom a need for the controller to slow down actively (and hence spending energy on that braking) because the friction that is always present in the system will do that braking job for free. Similarly, gravity helps a lot in letting a drone move downwards.

- *every* error must be reacted to, or (equivalently but often implicitly), *only a zero error* makes sense to strive for.

Many applications can tolerate a **dead zone**, because reacting to errors smaller than a relevant threshold adds no value.

- *optimality* is the enemy of **good enough**, and makes the controlled system more sensitive to disturbances.

All optimizations (whether they are done offline as in pole placement, or online as in optimal control) are based on models, and models only. That means that the quality of the “optimum” depends heavily on (i) the quality of the model (does it contain the “right” parameters?) and (ii) the quality of how well one knows the values of these parameters. Hence, spending a lot of efforts on finding the optimum most often makes very little sense. It can even be dangerous because many optimization problems have a high sensitivity in the optimum: a small change in parameter values can give rise to a large change in the optimum. For example, the “best” road for a mobile robot to follow when traversing rough terrain is very sensitive to whatever road quality parameters one chooses, and even more to the measurement principles and devices one has to rely on to find those parameters.

The following Sections describe alternative control design approaches, that want to cope with one or more of the above-mentioned simplifications explicitly.

7.5.3 PID alternatives: coping with its limitations

An attractive feature of a PID controller is that it requires only *one single and simple algorithm*, irrespective of the error or the setpoint. Section 7.2 explained where a PID control can fail. This Section describes mechanisms to help controllers cope with some of these limitations:

- an error **area** (Fig. 7.12) instead of an error **value** is used **to select** the:
 - **feedforward** part: the feedforward that is added to the feedback signal depends on where in its configuration space the controlled system is working at this moment. This mechanism is used in, for example, **sliding mode** control.
 - **feedback** part: the feedback signal depends on where in its configuration space the controlled system is working at this moment. This mechanism is used in, for example, in **gain scheduling**.

It is simple to formalise these areas as inequality constraints in the optimal control model.

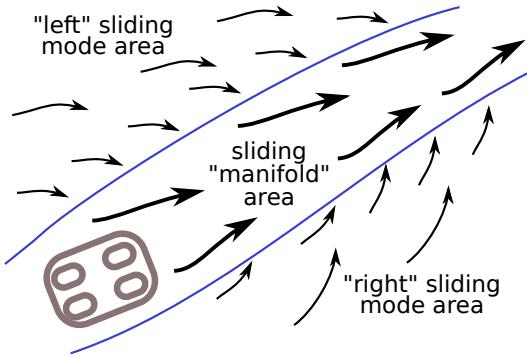


Figure 7.12: The concept of *sliding mode control*: the state space of the robot is divided into areas, each with a particular *a priori* determined control approach. In the strictest version of the concept, the middle area is the specified *trajectory*.

- the **trend** in the error, and not the **value** of the error, is used **to adapt** the **feed-forward** part. So, the behaviour of the controlled system does not depend on the magnitude of the error. This mechanism is used in, for example, **ABAG** control [52].
- the reaction to an error **need not be a linearly scaled version** of that error, but any “appropriate” **waveform** can be chosen instead. This mechanism is used in both sliding mode and ABAG control.
- the reaction to an error **prevents saturation** of the actuators. That is, the controller monitors no only the configuration space of the controlled system, but also that of the actuators. In most cases, this results in scaling down the control signal *before* saturation is reached.
- the **saturation** criterium can also be applied to the **magnitude** of the control signals. This criterium can be composed with the above-mentioned criterium of “distance” to state space areas.
- a controller can apply saturation limits on the feedforward and feedback signals **separately**. In PID control, this is known as **anti-windup**.
- the waveform of the control signal can be chosen to have only a **limited frequency** content. This is one of the ways to improve **stability**, by avoiding to excite the controlled system’s eigenfrequency.

The flexibility in the more-than-PID control approaches comes at a price (because the algorithms are more complex), but also with a benefit, because only marginally extra effort is required to make a controller **hybrid and adaptive**.

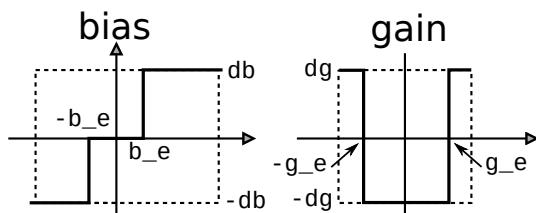


Figure 7.13: The core mechanisms of ABAG control: the *dead-zone* pre-processing on the error for the *bias* and the *gain*.

7.5.4 ABAG control (Adaptive Bias, Adaptive Gain)

ABAG control is almost as simple as PID control. Both have the same application contexts of one-dimensional control problems, they share the same definition of the “control error”, and have a small number of parameters to tune: P, I and D for PID, and B, and G for ABAG. (The latter two, however, are two times three parameters, in practice.) The B part represents the control signal that is *intended* to keep the plant in its current state, and that follows the

trend in the error at “slow” speed; this is similar to the purpose of the I term in a PID, namely to control away **steady state errors**. The G part has the same role as P: it is the part in the control signal that reacts “fast” to an error. For an ideal system in **steady state**, the feedforward contribution via B or I realises the full control, without any contribution of the feedback part G or P. For example, it generates the force to compensate for gravity, or for friction. In the control contexts of this document, models of the dynamic behaviour of the system-under-control are not the exception, but the rule; and, hence, the D part of a PID controller is replaced by a *model-based feedforward* part. The “A”s in “ABAG” indicate that both parts are **adapted** at runtime by the controller itself. Here is the meta model for an ABAG controller:

- the control signal is the sum of the **bias** (the “B” in “ABAG”) term and the **gain** (the “G” in “ABAG”) term.
- adaptation of B: the control error is integrated over time, for example with a first-order filter: $\bar{e}_{k+1} = \alpha \bar{e}_k + (1 - \alpha)e_k$, with e the error and \bar{e} the integrated error. As soon as that integral becomes larger than a specified threshold b_e (Fig. 7.13, left), the bias term of the controller is incremented (or decremented, depending on the sign of the integral) with a constant δb . The adaptation is saturated by a third “tuning parameter”, b_m , the maximum value that the B term can get.
- adaptation of G: as soon as the control error becomes larger than a specified threshold g_e (Fig. 7.13, right), the gain term of the controller is incremented (or decremented, depending on the sign of the integral) with a constant δg . The adaptation is saturated by a third “tuning parameter”, g_m , the maximum value that the G term can get.
- ABAG has **anti-chattering** built-in: the explicitly specified **dead zone** around the zero value of the error makes the controller react only after the magnitude of the error is larger than the specified threshold. The dead zone need not be symmetric for positive and negative error values.

The contribution of the integral term in a PID grows with, both, the *magnitude* of the error, and the *duration* of the error. The result is often more difficult to predict than the impact of the B term in an ABAG controller:

- the adaptation of B on the basis of the error can be made more deterministic than the adaptation of the I. For example, by allowing only a known waveform type, and with saturation. Hence, stability (in the form of controlled injection of energy into the controlled system) is easier to guarantee. While in the PID case, the building-up (and reduction) of the I term depends on the concrete error *magnitude*, which can not be predicted offline.

7.5.5 Policy: model-reference adaptive control (MRAC)

One particular **policy** of optimal control has been given the name **Model Reference Adaptive Control** (MRAC).

Its computation uses a model of the desired plant behaviour, to adapt, “optimally”, some parameters in the feedforward and feedback models.

7.5.6 Policy: model-predictive control (MPC)

Another particular **policy** of optimal control has been given the name **Model Predictive Control** (MPC) [115, 122, 132].

It is a special case of considering the design of a controller as an [optimal control](#) problem. Its computation uses a cost function that includes samples along the *full* predicted trajectory.

7.6 Mechanism: setpoint, trajectory, path and tube inputs

This Section introduces a *natural order* in ways to specify the goal of a controller, that is, in choosing the type of **input** that the controller accepts, The order is from *most constrained* to *least constrained*, where the latter gives the control designer the most degrees of freedom in the design process:

- **setpoint**: only one single **instantaneous value** of the **desired state** of the plant is being used in the control computations. In other words, the **control horizon** is only one time instant “deep”.
- **trajectory**: instead of just one instantaneous desired value for the **plant state** as input to the control design process, a trajectory of desired **plant state** values at multiple sample times over a certain horizon is used. This gives the designer more freedom to spread the inevitable control error budget over a larger space.
- **path**: another mechanism is to use a **path** instead of a trajectory. This is a less constraining input, because the **time is not imposed**. In other words, the **state** is *constrained* to follow the geometry of the path in state space, but not any timing along that path.
- **tube**: this is the least constraining input, because the controller can now also deviate from a given path, as long as the resulting path keeps the plant state inside a “tube”, or “region”, in the **state space**.

The design choice about what is the *input* to a controller defines, implicitly, also the interpretation of what is expected as an **acceptable (control) error**.

7.6.1 Policy: composition of optimal control and tube inputs

The optimal control approach of Sec. 7.5 allows the designer to make the choice of “error model” explicit, by adding corresponding constraints to the problem formulation. In other words, the PID and pole placement methodologies require generically valid **stability** functions, while the optimal control methodology allows the designers to make their specific choice of what it means to have a “stable controller” in the specific context of their application, and even to adapt the control behaviour to the concrete state of the system.

This flexibility comes with (guaranteed) extra costs: the obvious computational costs to be paid by a more complex online solver, but also often less obvious costs of forcing the designers to come up with motivations about what are the “appropriate” specific choices to make.

The flexibility also comes with (potential) added value: the controller need not be designed to react to *all possible* cases with one and the same control function, but it can adapt its instantaneous behaviour to the actual state of the optimization problem: More in particular, the controller can react differently depending on, for example:

- which constraints are violated.
- the sign and/or trend of the error.
- the selection of error function.
- the closeness to actuator saturation.
- the measurement range accuracy and/or noise level of the sensors.

Another form of flexibility that (potentially) adds value lies in the choice of solver for the constrained optimization problem. Some examples of this flexibility are:

- **stop anytime** solver: when a given computational budget is used up, the best available solution to the optimal control problem *at that moment* is used as the input to the actuators.
- **good enough** solver: many tasks do not require the control to be executed optimally, so the solver can stop as soon as the solution it has found is “good enough”.

From an [explainability](#) point of view, it makes sense to add the information of what is “good enough” as an extra set of constraints to the optimization problem. Indeed, that allows to introduce monitors around these constraints, that are hence as close as possible to the source of the decision making.

7.6.2 Policy: control progress objective or constraint

For setpoint and trajectory control, the *objective* of the controller is the same: to reduce the **error** to zero. But *path* and *tube* specifications are not constrained enough to determine the behaviour of the controller completely by a “zero error” condition. Indeed, a **progress specification** must be introduced explicitly in the optimal control formalisation of the controller:

- **progress objective function**: the closer the executed motion of the robot brings this objective function to its extremum, the better the progress.
- **progress constraint**: as long as the executed motion of the robot results in this constraint to be satisfied, the current control action can go on.

7.7 Mechanism: asynchronous distributed control

The Sections above made the **assumption of synchronous control**:

- all computations can be done in *zero time*.
- the computations are executed at the *right time*, say in 1kHz loop.
- the *scheduling* of the execution of the computations is the same every time one computes the whole control loop.

This assumption does not hold anymore for many modern machines, like cars or robots, that have multiple [fieldbuses](#) inside, and many of the computations (e.g., sensor processing) must run on separate processes or even computers. In addition, demands are shifting towards **system-of-systems** applications, in which separate machines come together in temporary systems and must realise tasks together; for example:

- multiple **tugboats** maneuvering a tanker.
- multiple *cranes* moving same load.
- multiple *drones* transporting same load.
- getting cars into and out of a *platoon*.

So, such control loops involve *communication* between control computing processes, Fig. 7.14.

Such a **distributed cascaded control** architecture introduces [asynchronicity](#) into the control problem:

- the closing of *feedback loops* is disturbed, because the latest state information is not available at the theoretically ideal time.
- there is now a need for *monitoring*: each subsystem must monitor how well its own control responsibilities are progressing with respect to what the overall system expects from it. It must provide this information to the other system components, and in

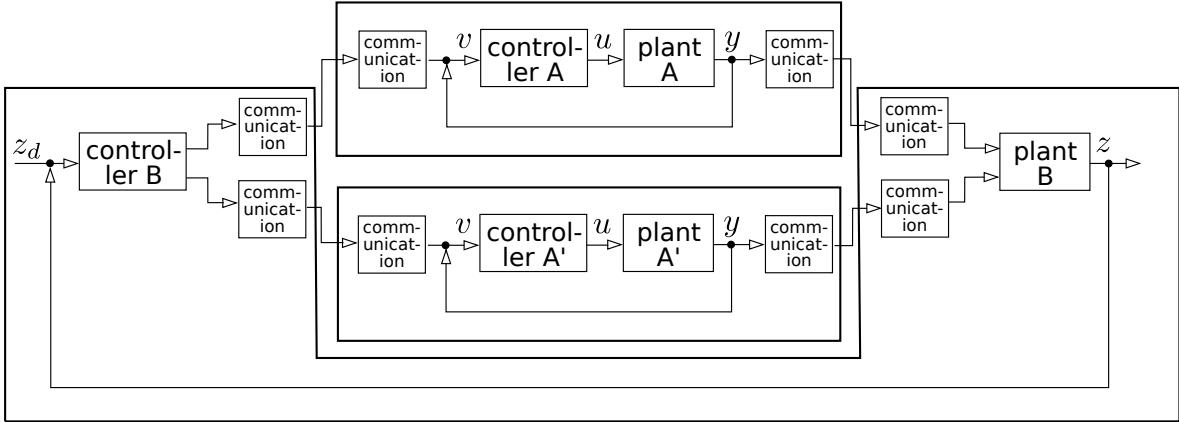


Figure 7.14: Many modern systems must rely on *communication* between sub-systems, in order to realise cascaded control loops.

turn must use the similar progress quality information that it receives from the other components.

- the result is the need for *mediation*: each subsystem must have a (safe, effective) reaction to the situation where the control progress, of itself or of its peers, is “not good enough”.

The result is that every distributed controller becomes an **hybrid event controller**:

- each sub-controller needs a *Finite State Machine*, with different control configuration in each state.
- all sub-controllers must also send *events* to each other, *to coordinate* their FSMs.

7.8 Mechanism: behaviour tree for semi-optimal control

A **behaviour tree** is a mathematical model, with a limited but very composable number of entities and relations, to decide what next *action* to take in a control loop. (It is a more specific version of a **decision tree**, focused on “control”.) It trades off optimality of the quality of the solution for speed of finding a feasible solution. The knowledge encoded in a behaviour tree model is typically known (i) to be “good enough” in particular use cases, and (ii) reflects experience in how to detect the (or rather, “a”) relevant use case via a series of decision making conditions that are fast to compute.

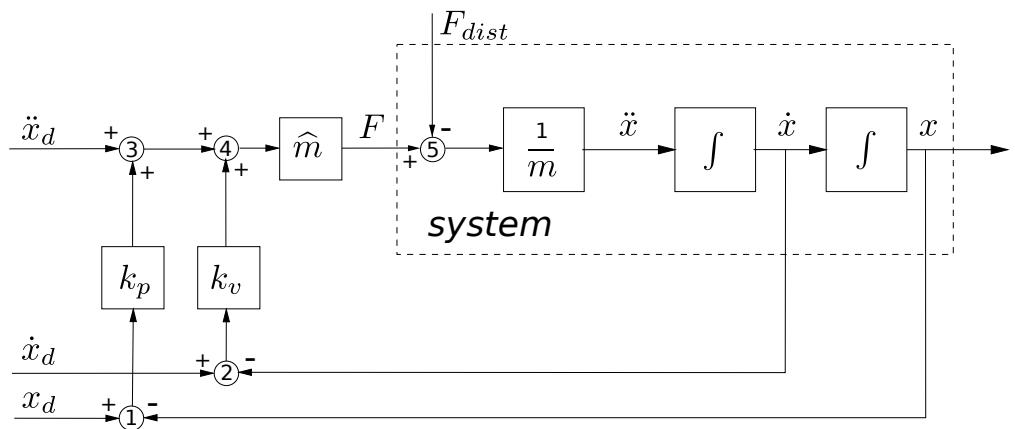


Figure 7.15: A one-dimensional position controller, generating the actuating force F from the desired position x_d , velocity \dot{x}_d and acceleration \ddot{x}_d , via nested velocity and position feedback loops with gains k_v and k_p . The “world model” of the controller consists of an estimate \hat{m} of the moved mass.

Chapter 8

Meta models for perception and its integration in tasks

Perception is *dual* to control, in many aspects and for all engineering systems, so that both control and perception “stacks” can be seamlessly integrated, at all levels of abstraction. However, in a robotics context the amount of perception opportunities (that is, sensors with sensor data processing activities) is huge. However, the information and software architectures for the integrated control-and-perception stacks are copies of those for the control stacks in themselves.

Previous Chapters focused on the *motion specification and control* aspects of a robotic stack. Motion in a robotics context always requires various forms of *perception*: specifying and controlling motion requires access to information about how “the world looks like” at any given moment, and that requirement can only be achieved if the (task-relevant part of the) world is perceived. Examples of such close integration between motion and perception are visual or force-based tracking of the interaction between a moving robot and its environment. So, it does not make much sense to develop all stacks independently, or to deploy their software implementations in only loosely coupled components: the way how things are perceived by robots, how robots are perceived by other agents, or how robots can/should move, depends to a large extent on how the world around the robots looks like, and on what information of that world can be provided by the sensor-based perception; similarly, motion is in many cases important to help perception, especially to improve *observability* of the world model updating process; finally, the task capabilities that a system offers, help to focus the perception to those sensor-processing efforts that are relevant to make progress in the task execution.

The term “stack” refers to the *hierarchical information/model structure* of all entities and relationships involved in perception. The first, **mereo-topological**, step in that direction is sketched in Fig. 8.1. This Chapter first explains that mereo-topological model in more detail, and then adds the meta models with **structure and behaviour** at the geometrical, dynamical, information theoretical levels of abstraction, using, amongst others, **Bayesian information theory** as a scientific foundation of this meta modelling. The connections are made for the **task-centric** integration between motion, perception and world modelling, allowing to model explicitly how task specifications can add artificial constraints on the perception behaviour.

8.1 Information associations in perception

This document introduces the **association hierarchy** sketched in Fig. 8.1, to structure the influence of knowledge representation to a task’s *perception* activities, at various **levels of abstraction**. Indeed, the hierarchy is **needed** to structure the **knowledge** about what are the “best” *relations*, or (*associations*), and the “best” magic numbers to use in the activities’ algorithms.

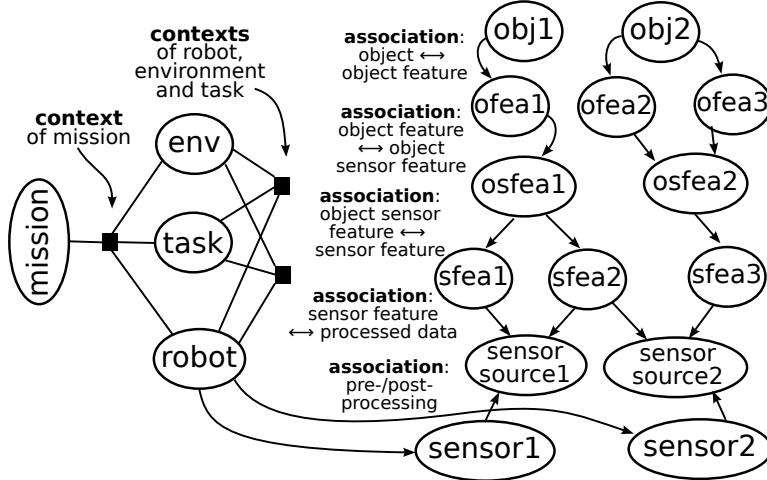


Figure 8.1: A **structure** to underly the complex dependencies in the *associations* between various levels of perception in robotic systems. This is the perception complement of the control association hierarchies Figs 7.6 and 7.7

8.1.1 Pre-processing

Examples of pre-processing operations on raw sensor data:¹

- low-pass filter,...
- conversions RGB to HSL/HSV or grayscale,...
- image pyramid.
- Discrete Cosine Transform, chirplet transform,...
- ...

For example, when knowing the time of day, one can choose better values for brightness thresholds. Or if one has an idea about the passive dynamics of a physical system, one can select better filter bandwidths in the pre-processing.

8.1.2 Pre-processed sensor data association to sensor feature

Examples of sensor features:

- PWM settings for an electrical actuator drive.
- dark-light transition expected in an image.
- corner detection, SIFT or SURF visual features,...
- entropy/texture quantification.

Again, the “higher-order” knowledge behind the association relations are of the following type:

- knowledge about the visual texture differences that can be expected between a sensor feature and its surroundings.

¹Or, in the other “direction” on the actuator setpoints coming out of a control algorithm.

- knowledge about the maximum current allowed in an actuator, or the best resolution that a sensor can provide.

8.1.3 Sensor feature association to object sensor features

Different parts of an object have different “features” for each particular sensor (and/or sensor processing algorithm). Examples are:

- surface **friction** model.
- object edge as separator between two surfaces with different **texture** and/or different **illumination conditions**.
- object region segmentation.
- object corner/surface detection.

Examples of the “higher-order” knowledge behind the association relations are:

- knowledge about expected visual differences in line thicknesses of **visual markers** on a road.
- knowledge about the materials of the floor, the walls, and doors in an indoor corridor.

8.1.4 Object sensor feature association to object feature

Examples of the association relations linking different *functional* parts of an object with different *object sensor feature* parts

- the handle is painted green while the door is painted blue.
- the border of the table is where the contact force of a gripper sliding over the table **drops suddenly**.

Obviously, the above-mentioned examples of knowledge come with specific *quantitative values* of the mentioned properties or effects.

8.1.5 Object feature association to object association

A particular object is recognized as a “sufficient” number of its object features have been detected, in the expected relative order. For example:

- the pattern of doors and windows in an indoor corridor.
- the location of holes and edges in an object that is to be **assembled** in a **manufacturing cell**.

Examples of the **magic numbers** in these association relations are: knowledge about relative sizes, shapes, or scales of the objects’ surfaces.

8.1.6 Association to task, environment and robot context

The objects are part of a particular task, that a particular robot has to execute in a particular environment. For example, delivering logistic goods in an indoor environment, or assembling an aircraft’s wing. Task, environment and robot, together, define the **context** of the perception (and control). Each of the three contributing factors has contributions of these three types:

- **purposive** (or, *intentional*): the perception activities in the robot must be focused on providing that information that is necessary for what the robot is expected to deliver as a result of its actions.

- **causal**: nature inhibits some “states of the world”, and this information is modelled in so-called *scientific laws*. So, the actions of the robot should not try to violate these laws.
- **normative**: human society may want to restrict even more the “states of the world” that the robot is allowed to realise, by prescribing *norms* to be followed, of legal, ethical as well as traditional nature.

8.1.7 Association of a mission with its tasks, environments and robots

The overall mission of a system determines which of the above-mentioned association relations are relevant or important. Examples of where this “intentional context” matters is the decision making of whether or not to continue an ongoing task execution, or to deploy more or less robots in it.

8.2 Mereo-topological meta model: the natural hierarchy in robotic perception

The perception stack model has *structural parts* and *behavioural parts*. The structural part uses *property graphs* to model the fact that n-ary relations exist between entities in the stack. These structural relations conform to the *Block-Port-Connector* meta model. The behavioural models describe the dependencies between the values of the properties in the connected entities, and these dependencies can be continuous, discrete, or hybrid. For robotic systems, every perception model has n-ary *relations* between *entities* of the following types:

- **sensor**: to describe the properties of the **data** generated by sensor devices.
- **actuator**: to describe the properties of the **data** to be provided to actuator devices to make them put energy into the system.
- **features**: relations between *sensor* data and *object* properties that play a role in the context of a *task*, or between *object* properties and their role in a *task* to determine how the *robot* should move, or both of the above within one single relation. “*Task*” can be replaced by a composition of entities in the models of the *task*, the *robot* and the *environment* in which the previous two operate.
- **objects**: have properties that can be linked to data features, for sensing as well as actuation, and to the tasks that describe what robots have to do with them.
- a **robot** model represents the sensing and actuation capabilities and resources of robotic devices and systems.
- **environment** entities, are often relevant for parametrizing perception algorithms (e.g. camera parameters due to lighting conditions) or to select appropriate sets of sensors (e.g. during fog or rain outdoors or when encountering a dark indoor area during night or in the basement). Obviously, this part of the perception stack contains the links to world modelling; an important development within the project will be the “right” separation and composition of perception modelling and world modelling.
- the **task** plays a crucial role in constraining the selection of all other entities ranging from limiting object types that are relevant during that task to the selection of the perception features that need to be detected.
- the **mission** model makes choices of which task, robot and environment models must be used together to realise “long-living” applications.

8.3 Mereo-topological meta model: natural hierarchy in world representation

Classification according to [105, 123]:

- **subordinate**: e.g., boxes and bolts in an assembly workcell, or cars and pedestrians in a street.
- **basic**: the assembly workcell, the street.
- **superordinate**: a factory, a city.

8.4 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.

8.5 Mechanism: goodness of fit, similarity, template matching

Several complementary mechanisms have been developed “to measure” how good the result $R(x)$ of a perception activity is, by measuring how “similar” it is to a given template $T(x)$, and the similarity r is represented as the minimization over the parameters x and y , where the latter runs over the domain of a “deformation function” $g(\cdot, y) : x \rightarrow g(x, y)$:

$$r = \arg \max_y \left(\text{similarity}(T(x), R(g(x, y))) \right). \quad (8.1)$$

Examples of deformation functions are scale, translation, rotation, or perspective transformation.

These are the major *families* of goodness-of-fit measures:

- least squares: including peak signal to noise ratio. Not robust against affine disturbances: offset and scaling.
- correlation and convolution: multiplication introduces offset and scaling robustness. self-correlation: using the result itself as reference.
- entropy, including mutual entropy and [Kullback-Leibler divergence](#). Introduces weighing according to “importance” of data point reflected by *magnitude* of function at that point.

Extensions: other types of [divergences](#).

- energy minimization: the sensor data is compared to a given template pattern, by elastic deformation of the latter to correspond best to the former. In other words, to

reach *maximum similarity* between both patterns while minimizing the deformation, a *cost (or energy) function* is minimized. Such cost functions also help *to improve* the matching by following a path to lower-energy configurations.

Extensions: [Mumford-Shah](#), where the “energy” is computed on (not necessarily) connected *homogeneous regions*.

- trend fitting [145]: minimizes (i) the sum of the absolute k th order discrete derivatives over the input points, (ii) the sum of squared derivatives across input points, or (iii) the total variation of the k th derivative.

Extensions: curves with “*don’t care*” parts; asymmetry of cost between deviations “above” and “below” the curve.

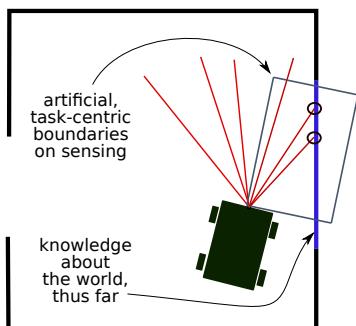


Figure 8.2: Application example of the *escape from the room* task (Sec. 6.15.4) to illustrate various levels in the perception stack hierarchy of Fig. 8.1.

8.6 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.

8.6.1 Escape from a room

Assume the robot has a *laser scanner*, *encoders* on the wheels, and a *cameras* (one looking down to the floor, to use its texture for self-localisation; one looking to the ceiling, for similar purposes; and one looking forward). Part of the sensor models describes the physical units, the mathematical, numerical and digital representations of the data that the sensors produce. For the camera this is a matrix with dimensions defined by the sensors resolution property; each of the values in this matrix is a vector containing the RGB values, which are in turn chosen to be represented as integers between 0 and 255. The laser scanner has a similar representation, but with the 2D RGB image replaced by a 1D vector of depth values.

The digital representation of the camera image is used by one or more *segmentation* algorithms. For example, one based on color is configured with the size and color properties of the expected features in the room; assume that there is a wall with a round green drawing. While the system architect would only choose the type of algorithm, the system builder needs to choose a specific implementation here (e.g. in which color space to look for “green segments”). Also the grounding what “green” means in terms of regions in a color space needs to be grounded in a digital representation (potentially by linking to an ontology describing colors in various spaces). In addition, also the *environment conditions* play a role, because the perceived color depends not only on the object properties but also the lighting conditions.

The output is a set of green regions, which some algorithms might use as prior knowledge for the next iteration.

To simplify the data association problem, it is assumed that only the circle with the highest probability will be used. This circle has a state which is represented as its centroid and diameter. By only looking at the numbers shown in Fig. 8.2, it is difficult to say that these numbers are in image or pixel coordinates. Therefore, it is again important to point at the meta model describing the digital representation and the semantics of the data.

This centroid and diameter found in the camera image are then used as an input to a [Kalman Filter](#) (some additional pre-processing is not displayed). A Kalman Filter is a generic, “platform”, algorithm that needs to be configured with a process and a measurement model, initial conditions, as well as noise parameters. These are configured from the sensor model, task model, and object model. Please note that, in contrast to the perception stack, the task is not explicitly shown in this figure since it is influencing the overall architecture and choices. A Kalman Filter requires a state to work on, which is a (dynamically changing) property of the ball. Again, its digital representation is important as is the semantical context like the frame its position is expressed in (see motion stack).

The green round feature typically has many properties that can also change with every new application. Therefore, the suggested structure allows them to be composed with the “ball” while keeping their semantic context by pointing to the models they conform to. The number of possible object properties is huge and will have to grow over time.

8.6.2 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.)

8.6.3 Ego-motion estimation with visual point and region features

(TODO: point features are abundant in vision, at the detriment of regional features, that are often more difficult to compute, more dependent on the application, and on other features. Typical regional features are: entropy, geometric or texture patterns, and spatial and temporal frequencies, often on the raw pixels but also on pre-processed pixel values.)

8.7 Probability: composition of data and uncertainty

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

8.7.1 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 [62, 72]. 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 [62], resulting in the primary role of **logarithms** as the natural **measures of information**. These axiomatic foundations are as follows:

- I** $I(M:E \text{ AND } F|C) = f\{I(M:E|C), I(M:F|E \text{ AND } C)\}$
- II** $I(M:E \text{ AND } M|C) = I(M|C)$
- III** $I(M:E|C)$ is a strictly increasing function of its arguments
- IV** $I(M_1 \text{ AND } M_2:M_1|C) = I(M_1:M_1|C)$ if M_1 and M_2 are mutually irrelevant pieces of information.
- V** $I(M_1 \text{ AND } M_2|M_1 \text{ AND } C) = I(M_2|C)$

A **model** is a set of relations between entities in a domain, and the model for **information** (or **uncertainty**) is a *Probability Density Function* (PDF) $p(X, Y, \dots)$ over the parameter space of a selection of the properties in the model:

- *discrete* PDF: parameter space has only *finite* number of possibilities.
- *continuous* PDF: parameter space is continuous.
- *hybrid* PDF: parameter space combines discrete and continuous sub-spaces.

Information representation is *subjective*, because a PDF is a multi-dimensional, single-valued function $p(X, Y, Z, \dots)$ that describes the probabilistic relationship between the variables X, Y, Z, \dots in a **model** M :

$$p(X, Y, Z, \dots | M)$$

and the model M is a *chosen* representation of the *chosen* relationships (assumptions, constraints,...) between the variables X, Y, Z, \dots So, the PDF represents what the *system* “knows” about the world, *not* what the world really *is*. Engineers have to *choose* what mathematical representations to use, for domain as well as for information!

Information structure = graph:

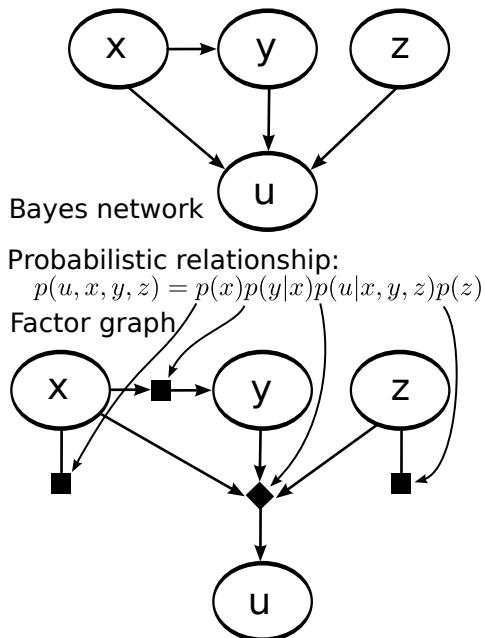


Figure 8.3: Example of a probabilistic model, in Bayesian network form (top), as a factored conditional probability density function (middle), and as a factor graph (bottom).

- **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*.

The most important arcs are the ones that *are not there*!

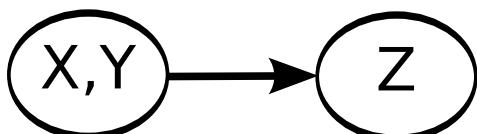


Figure 8.4: Simplest Bayesian network.

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 $x = \mathbf{1}$:

- value “1” is **arbitrary choice/convention!**
- only **relative** value of probability mass is important.

Measure of (change in) information:

- **mutual information**, **relative entropy** of two PDFs P and Q :

$$H(P\|Q) = \int_X \log \frac{dP}{dQ} dP.$$

- “*how much does information change when new data becomes available?*”
- “*no information*” does not exist → always *relative*!

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

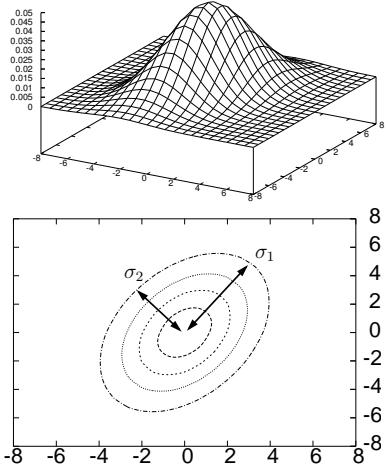


Figure 8.6: 2D Gaussian.

Mean $\boldsymbol{\mu}$, Covariance \mathbf{P} :

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

($\boldsymbol{\mu}$ is vector) (\mathbf{P} is matrix)

Advantages of Gaussian PDF representations:

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

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”:

$$\int \phi(x)p(x)dx \approx \frac{1}{N} \sum_{i=1}^N \phi(x^i) = \sum_{i=1}^N w^i \phi(x^i)$$

8.7.2 Mechanism: Bayes’ rule for optimal transformation of data into information

(TODO: [169].)

8.7.3 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. [77, 107])

8.7.4 Policy: hypothesis tree for semi-optimal information processing

(TODO: [31, 120].)

8.7.5 Policy: mutual entropy to measure change in information

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

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

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

8.8 Geometrical semantics in perception

8.9 Dynamical semantics in perception

8.10 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.

8.11 Mechanism of information update: Bayes' rule

The essential role of Bayes' rule: “*Inverse probability*”:

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

(product rule) \Downarrow (product rule)

$$p(x|y, H)p(y|H) = p(y|x, H)p(x|H)$$

$$\Rightarrow \boxed{p(x|y, H) = \frac{p(y|x, H)}{p(y|H)} p(x|H)}$$

```

graph LR
    X((X)) --> Y((Y))
    subgraph Labels [ ]
        direction LR
        L1[hidden] --- X
        L2[observed] --- Y
    end

```

Bayes' rule, for the inclusion of new data:

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

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

Data: *observed*; Model parameters: *hidden*

All factors are functions of *model parameters*, except $p(\text{Data}|H)$ = often just “*normalization factor*.”

Bayes' rule: important properties:

- $p(M|D, H)$: **function** of M , D , and H .
- PDF on Model parameters “in” \Rightarrow PDF on Model parameters “out.”
- Integration of information is *multiplicative*.
- Computationally intensive for general PDFs.
- Easy for discrete PDFs and Gaussians. (And some other families of continuous PDFs.)

- $p(\text{Data}|\text{Model params})$: requires known table or mathematical function $\text{Data} = f(\text{Model params})$ to predict Data from Model.
- Likelihood is *not* a PDF.
- **Optimal Information Processing and Bayes's Theorem**, Arnold Zellner, *The American Statistician*, 42(4):278–280, 1988.

8.12 Mechanism of perception solver: message passing over junction trees

The *message passing algorithm in factor graphs* [19] plays a similar role in perception as the *hybrid dynamics solver* of Sec. 4.7 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. 4.7), 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.)

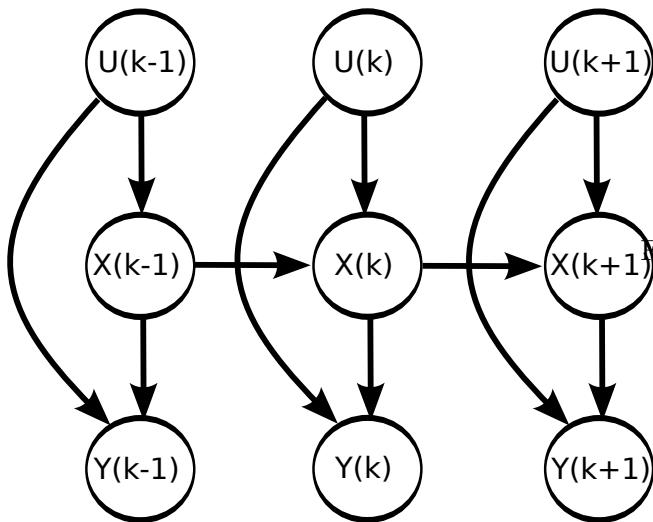


Figure 8.7: Dynamic Bayesian network.

8.13 Policy: dynamic Bayesian network

Figure 8.7 sketches a typical dynamic Bayesian network, where one of the arrows represents *evolution over time*.

Variables:

- U : control inputs
- X : state information
- Y : measurements

Arrows:

- motion model: $X(k+1) = f(X(k), U(k+1))$
- measurement model: $Y(k) = g(X(k), U(k))$

Multiple arrows can be represented by *one function*.

“*1st-order Markov*” = “time”-influence only *one step* deep.

A dynamic Bayesian network is the probabilistic extension of the representation of a physical control system:

$$\begin{cases} \frac{dx}{dt} = f(x, \theta, u) \\ y = g(x, \theta, u) \end{cases}$$

- x : domain values.
- t : time.
- θ : model parameters (PDF, relationships).
- u : input values.
- y : output values.
- f : state function, or “*process model*”
- g : output function, or “*measurement model*”

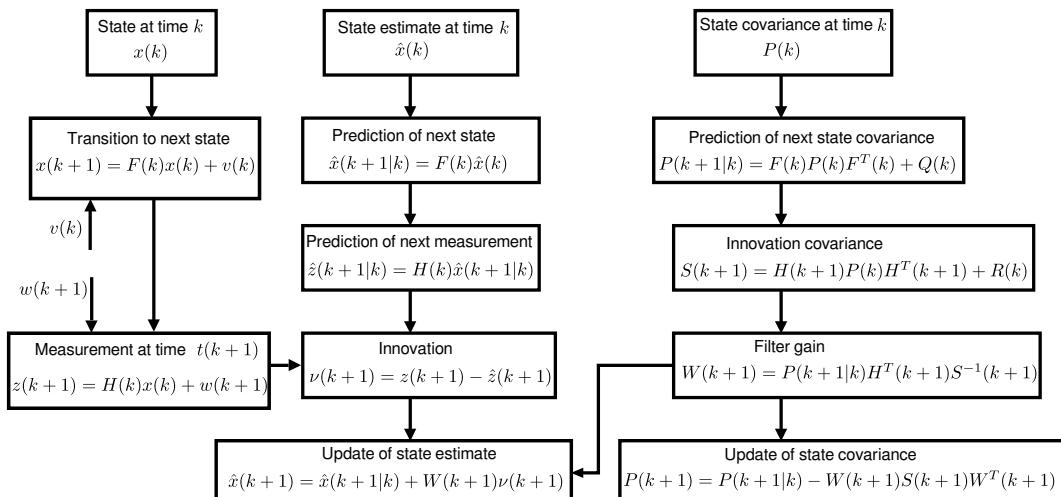


Figure 8.8: Computational schema of the Kalman Filter.

The simplest dynamic network is the *Kalman Filter*.

Required inference: given Y and U , update X .

Assumptions: fast analytical solution possible!

- Process model: $x_{k+1} = F x_k + Q_k$.

- Gaussian “uncertainty” on x_k : covariance P_k .
- Gaussian “process noise”: covariance Q_k .
- Measurement model: $z_k = H x_k + R_k$.
- Gaussian “measurement uncertainty” on z_k : covariance R_k .

Typical application: *tracking* = adapting to *small* deviations from previous values.

Second simplest dynamic network: Particle Filter for localisation.

- functional relationships $f(\cdot), g(\cdot)$: can be *non-linear*.
- PDF *representation*: samples.
- *each sample* is sent through f and g separately, and then a new PDF is reconstructed.

So, a *numerical* solution is needed. **Typical application:** *localisation* with *large* uncertainties.

8.14 Policy: Factor Graphs

8.15 Mechanism: composition of point and region features

8.16 Policy: feature pre-processing

8.17 Policy: deployment in event loop

Chapter 9

Architectural patterns

Previous Chapters introduced two essential ingredients of **cyber-physical systems**:

- (i) **building blocks** for **component** functionalities: **functions** with their **data** arguments, **algorithms**, **activities**, **transactions**, **streams**, and coordination mechanisms (Secs 2.6, 2.7 and 2.8).
- (ii) **functionalities** required in the application domain of **robotics**: kinematics, dynamics, perception, control, world modelling and situations, and the corresponding task specifications and executable skills.

This Chapter adds a third ingredient, namely (iii) **patterns** (and corresponding **best practices**) about **architectures**, to connect components into a **system**, independently of the fourth and fifth ingredients in the design, that is, respectively, (iv) the **application context** and (v) the **software implementation**. The patterns introduced in this Chapter are:

- the **5Cs pattern** for an individual component: its **computation**, **communication**, **coordination**, and **configuration** functionalities are all designed to be **composed** together.
- the **DOM** (Document Object Model) to represent the **hierarchical context** for the configuration and coordination of decision making in the interaction between components.
- the **mediator** for **ownership** and **coordination** in inter-component **decision making**.

9.1 5C component: to compose interaction hierarchy with coordination hierarchy

It is common practice to **decompose** [137] a cyber-physical application into a (potentially large) number of **activities**, in *physical* processes (that is, in nature, or in man-made hardware devices such as mechanisms, sensors and computers), and/or in *cyber* processes (that is, in software). The activities **interact** with each other, in *cyber space* via the software and communication **services** they provide to each other, but also in *physical space*, via the actions they perform on the *resources* they share, and via the *energy* they exchange.

The responsibility of **system architects** is to bring structure in a system, using *components* as the building blocks for which to design the **interactions** between the activities inside the components. This results in an architectural composition:

- that can be **deployed** in the system (again, for both the physical and cyber versions),
- without a need to know about the exact workings of these activities and interactions in the **inside** of the component.
- but providing **data sheets** with the formal model of how a component interacts from the **outside**.

Several other terms are used to represent the concept of a “component”: **service**, **sub-system**, **holon**, **agent**, **process**, **node**, **server**,.... In addition, this document uses the term “component” as a *pars pro toto* or *synecdoche*, to refer to the functionality *compositions* introduced elsewhere in this document: **functions**, **algorithms**, **activities**, and **interactions**. Of course, the quality of a component in a system depends to a large extent on the **composability** scope with which the components’ interior has been designed.

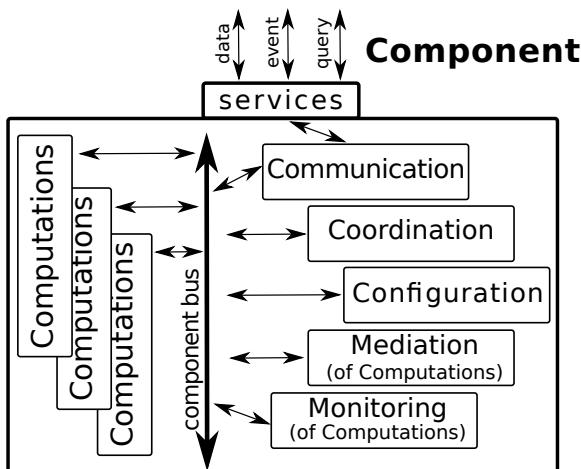


Figure 9.1: The mereo-topological model of a *Component*, structured with the **5C** meta model. The **Computations** on the left are the ones that realise the added value provided by the *Services* that the *Component* makes available. The ones on the right are indispensable “overhead”.

9.1.1 5C’s: composition of 4C behavioural roles into one component

This Section presents the **5C** component meta model,¹ [95, 117, 155], consisting of the following “*four + one*” **mereological** entities and relations (Fig. 9.1):

1. **Computation**: the **algorithmic** entities that realise the components’ “**functional**” behaviour, that is, changes of the component’s “state” in (software representations of) time, space, energy, quality,...
2. **Communication**: the **interaction** entities that **exchange** data, events and models between **synchronously**, **concurrently** or **asynchronously executing** activities, respecting specific constraints on consistency, ordering, memory usage, timing, etc. A *Component*’s internal exchange of information can be organized as one or more (**software**) **buses**, including **streams** in shared memory.
3. **Coordination**: the **logic** algorithmic entities that realise the component’s “**discrete**” behaviour, that is, they interpret the combination of “*states*” in **FSMs**, **events**, **Petri**

¹The 5C model of this Section has its **acronym** in common with that of [87]; while there is a thematic overlap, the meaning of the latter 5C model corresponds more to the “levels of abstraction” discussed in Sec. 1.7.1 and following.

Nets, protocols, and flags, to decide whether or not the component has to change its (Computation and Communication) behaviour. The latter decision is also encoded into a change of some “state”.

4. **Configuration:** the **logistic** entities that turn behaviour-changing **Coordination** decisions into reality. For example, the situation around a robot has changed, and the Coordination functions have signalled *what type* of situation change has occurred. Depending on that type, the robot’s actual Computations and/or Communications must be changed, and realising that change is done in Configuration functions. Such a desired change can mean any combination of the following:

- another set of **Computations** is to be activated. That can also involve the choice of another **Coordination** algorithm.
- another set of **Communications** is to be activated.
- the **Configuration** settings of parameters in components are changed.

5. **Composition:** every **architectural** relation that **couples** a collection of the four entities above into one **component**.

The following distinction is often not made sufficiently clearly in applications: the Coordination functions make decisions *that* a change must take place, but *what* that change really means is realised in the Configuration. In other words, the same *decision* can lead to *different behaviour*, because the behaviour-changing actions of the Configuration can depend on a broader context than the one in which a Coordinator has made a decision.

An example of such a broader context, is that the Configuration behaviour must also take into account the **constraints** imposed by the “state” of the **resources** that components are using. Indeed, a component can not always change its behaviour instantaneously, from one execution time to the next, because (i) configuration requests often *trickle down* to other “resource” components, and (ii) there is *asynchronous state* involved in the process of realising a (re)configuration. In addition, it can happen that multiple Coordination decisions have to be processed “at the same time”, and maybe some subset of them can lead to inconsistent or inefficient resulting behaviour.

So, Configuration is often to be realised by a full-fledged **activity** in itself, and not just by changing the values of some parameters “atomically”. For example, two robots pushing around the same cart must *communicate* about the behavioural change in their shared physical interaction with the cart, and that communication itself takes non-negligible time, but there is often also some mechanical impedance to be taken into account in changing the cart’s motion.

The first 4Cs of an activity only have meaning in the context of composition with other activities; that is, a specific algorithm does not have the *property* of being, say, a Coordination algorithm, but it gets that *role* as an **attribute** in a **Composition**. For example, it only makes sense to separate Configuration of one particular component or system into multiple Configuration components, *if and only if* the following conditions are *both* satisfied:

- there is a design driver to compose that component with other components, now or later.
- that composition *requires* the computations of both components to be configured *together*.

In other words, it only makes sense to spend efforts in dissecting components into 4C parts if later compositions of these components introduce **dependencies** between one or more of these parts, between two or more of the components. If a component already comes with 4C decoupling, it is *potentially more composable* than a monolithic component.

The total 5C meta model has only **qualitative** value, that is, it offers no **concrete composition guidelines**:

- it just makes developers *aware that* each (software) component **has-a** specific set of parts with the above-mentioned roles.
- it does not explain *how to realise* these roles with concrete software components, neither **declaratively** nor **imperatively**.

The constructive design guidelines are introduced later in this Chapter: **DOM models** are examples of *declarative* representations of dependencies of the 4Cs in different activities; the **event loop** is an example of an *imperative* representation of the *execution* of an activity; and templates exist for the integration of large number of activities into one system. All of these are *models of interaction*, but *not* (yet) **software architectures**. Nevertheless, they *do* add value to the system design process:

- these models of architectures can be **pre-processed before configuration and deployment**. This allows, for example, to optimize the amount of **Communication**, **Configuration** and/or **Coordination** components that are needed in a (sub)system.
- the architecture of a deployed 5C-based (sub)system can be **adapted at runtime**, if (i) the 4C *models* of all **Components** are available, and (ii) the **Configuration** component can deal with online *queries* to all **Components** because it needs their cooperation to realise the system's re-composition.
- the model is an excellent (because structured and explicit) **documentation** for *human developers* to discuss the system design, and its trade-offs.

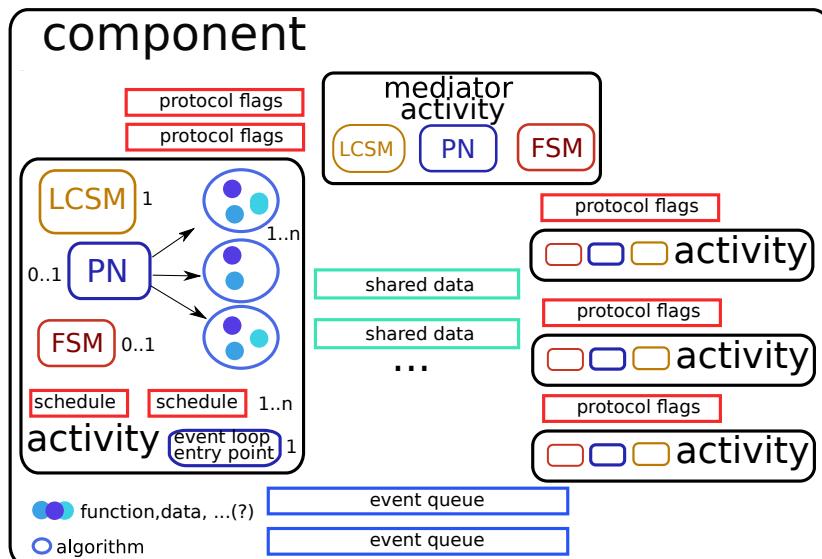


Figure 9.2: Mereological architectural pattern for *one single component* deployed in *one single* process, using shared data for all the interactions between the activities in its *sub-components*. The pattern introduces one dedicated mediator activity for the coordination between the other activities.

9.1.2 Component = activity with 5C-based architecture

This document advocates² to use the (all too common) term of a “**component**” for an **activity** that has all of the **5C behavioural building blocks**, Fig. 9.2:

- one or more **algorithms**, to realise the component’s internal behaviour. Each algorithm requires access to a set of **abstract data types**.
- one **Life Cycle State Machine** (LCSM), to *coordinate* the internal behaviour of the component with its outwards-facing behaviour. The latter is observable by other components in an application.
- one or more **task queues**, that provide the information to the component of what it should be doing, now and in the near future.
- zero or more **Finite State Machines**, to *coordinate* the execution of each individual task.
- zero or more **Petri Nets**, to *coordinate* the algorithms in the component, internally within the component, or externally with a set of algorithms in other components. This coordination requires access to a set of **protocol flags**.

The actions *inside* of the component need to be composed with actions *outside* the component, and these are the necessary entities to do so:

- one **event loop** to trigger the computations in its algorithms, FSM and Petri Net(s). An event loop has two parts:
 - **runtime**: the part owned by, and deployed in, the **thread**.
Threads and runtimes are mechanisms of the **software architecture**.
 - **entry point**: the part owned by, and deployed in the **component**, where the thread’s runtime **executes** the component’s **scheduler** of functions.
The schedule of an activity/component is a mechanism of the **system** or **information** architectures, introduced in this Chapter and the **next**.
- one or more **interaction channels**, to exchange and manage the data that its algorithms produce and/or consume.

The mechanisms of *entry points* and *runtimes* are also needed to support the execution if data exchange over interaction channels.

When the activity inside the component switches its LCSM state, it typically

- executes a (partially) different set of algorithms.
- interfaces with a (partially) different set of channels than the ones it had been interfacing with in the current LCSM state.
- uses a (partially) different set of Petri Nets for its algorithm coordination.
- uses a (partially) different set of configuration parameters for all of the above.

More concretely, for each state of its LCSM, the component’s activity provides a different **event loop**, consisting of one or more of the following **4Cs** blocks:

- **communicate()**: realises the loose coupling with channels, in that it (i) implements the *asynchronous produce()* and *consume()* functions for all of the channels that must be interfaced in the current LCSM state of the activity, and (ii) adapts the *port* configuration through which the algorithms inside the activity access these data structures *synchronously*.
- **coordinate()**: this comes in two flavours:
 - **coordinate(&LCSM)**: decides upon *externally visible* behavioural state changes, via the “state data” of the LCSM.

²The reasons for these suggestions are explained in detail in Chapter 2.

- `coordinate(&PetriNet)`: synchronises *internally* the execution of algorithms, via the “state data” of a Petri Net.

The LCSM and Petri Nets are not fully decoupled: some changes in Petri Net flags should result in LCSM events, and the other way around. Such *conversion functions* are part of the “shared level” of all coordination functions.

- `configure()`: realises the changes in some `magic numbers` in all other function types, whenever a change in LCSM state requires such a reconfiguration.
- `compute()`: executes the functions required for the current activity behaviour, and driven by its current data state.

The particular *composition* of all of the above 4C’s functions makes up for the *fifth* part of the **5C** computational paradigm for behaviour. The 5C composition level also adds the `communicate()` functions that connect some of the *interaction channels* to *outward-facing Inter-Process Communication* channels.

Some common constraints on the 4C patterns above are:

- `configure()` is the obvious choice of schedule to execute the first time an activity enters a life cycle state, because that leaves *all* decision power about what to do in that state to the activity itself, with the `latest possible binding`.
- after being configured, just having a `compute()` block in the schedule makes sense. Just having a `communicate()` or a `coordinate()` does not.

9.1.3 Types of state: state, status (flag), and their changes (event)

“State” is one of those over-used terms in `science` and `technology`, and many other terms are used as (almost) synonyms (“mode”, “phase”, “status”, “flag”,...). This document chooses to use three terms (*state*, *status*, *event*), with the following semantics:

- **computational state of an algorithm**: an algorithm (or computer programme) stores data in variables, and their value can be stored to `persistent storage` at any time, and re-loaded in the computer later to continue the algorithm’s execution. (This assumes all functions in the algorithm are `without side-effects`.)
- **behavioural state of an activity**: this state represents the single *behaviour* that an `activity` is executing at a given moment in time, and consists mainly of (pointers to):
 - the computational state of the algorithms inside the activity.
 - the streams the activity is interacting with.
 - the states in an activity’s finite state machines. These data structures are meant **to be used internally** in the activity’s own coordination algorithms, to decide about which **behaviour to realise** now, or next.
- **state of a interaction**: represents the progress in the agreements (`protocols`) that activities have about how to interact with each other, physically or `digitally`.
- **state of a component** (or system): the composition of the states of all algorithms, activities and interactions in the component.
- **status**: represents the **value of a condition** (symbolic, numerical or logical relation) on some parameters in states of the above-mentioned types.
A status is represented by a **(status) flag**, which is meant **to be observed synchronously** by other activities, to let algorithms in different activities decide which **functions to execute** now, or next. The `Petri Net` is the meta model of a very common approach of this type of coordination.
- **event**: represents the fact that **something has changed** in a state or a status. That

“something” can be the value of a variable (e.g., the distance of a robot to an object), but also the behavioural state of an activity, or the status flag of an algorithm or data structure.

An event is meant **to be communicated asynchronously** towards other activities, to allow their **coordination to react** to what happens elsewhere.

9.1.4 Computation: monitoring, coordinating, mediation, scheduling, dispatching

The following *types of Computation* are so generic (at least in robotics contexts) that this document includes them in its *cyber-physical domain* extension of the generic component meta model :

- **monitoring**: a computational component with a “higher-order” role, in that:
 - it does not contribute directly to the *services* that the *Component* must provide in the application.
 - it needs access to some of the data used by one or more of the computational components that *are* responsible for the just-mentioned *services*.
 - with that data, it computes the **Quality of Service** (QoS) with which these computations are realizing their *expected* service behaviour.
 - it **fires an event** or **raises a flag** when a particular QoS threshold is reached. The threshold can be reached in, both, the “*good enough*” direction, or in the “*not good enough anymore*” direction.
- **coordination**: a computational component that executes the algorithms to execute (i) Boolean logic operations on Flags, (ii) Petri Nets, or (iii) Finite State Machines.
- **mediation**: a computational component with another “higher-order” role:
 - it also does not contribute directly to the *services* that the *Component* must provide in the application.
 - it exchanges data with Configuration algorithm of a Component.
 - it has the extra **application-specific** knowledge about how various Computations are to be **traded-off** when the Quality of Service of the Component goes beyond its configured thresholds.
 - it takes the **decision** to trigger Configuration of some of the Components to react according to the application’s policy. The latter are system-level trade-offs, that the system developers have configured into the Mediation component itself.
- **scheduling**: many algorithms in robotic and cyber-physical systems have high flexibility, because they cover a diversity of computational tasks, large and small, that must be realised “*together*”, constrained by some *inter-dependencies*. For example, the torque control loops of all motors in a machine; various visual feature detectors in different places of a camera image; distributing “work” to various “workers”; ordering the access to shared resources; etc. Such computations are often called **scheduling**.
- **dispatching**: a computation that must:
 - decide which resources (activities, algorithms, components, hardware,...) are responsible for the execution of a schedule.
 - monitor the actual execution.
 - fire the necessary events to keep the system updated about the progress of the schedule execution.

There can be multiple Monitoring and/or Mediator components in each Component model,

the same **Monitor** and/or **Mediator** can get data from several **Computations**, and the same **Computation** can provide data to several **Monitors** and/or **Mediators**.

9.1.5 Coordination: coordinated, orchestrated, choreographed

One possible trade-off to make in the design of a system is that between (i) the amount and latency of the communication required to coordinate components, and (ii) the *autonomy* of each component. This trade-off is represented by the following modes of coordination that components can engage in:

- **coordinated**: *all* components react only when they receive an explicit event to do so, and they expect that event to come with all information about how to react.
- **orchestrated**: less events have to be broadcasted, and each carrying less information, because all components share the same “**score**”. That is, a *model* that represents (i) the expected sequencing of events, and (ii) the expected reaction of each component.
- **choreographed**: the components have a *Plan* of their mutual coordination, and they have *Computation* components that *observe* the behaviour of other components, and *Monitors* that can recognize when the reconfigurations that are scripted in the *Plan* must be executed. Ideally, this can happen without the need to broadcast one single event between the components.

The descriptions above use *events* as the triggers for coordination of *behaviour* of components, but the same three coordination types hold for *flag-based* coordination between multiple *activities* in one or multiple components.

9.1.6 Communication: property & attribute, parameter & stream

One component can influence the behaviour of another component by giving values to parameters that that latter component is using in its **Computations**. There are two ways to do so:

- through **Configuration**, by a component which is “higher up” in a particular decision making hierarchy. Also here are two complementary mechanisms:
 - **property** parameter: the parameter is owned by the component that provides the behaviour, in the sense that the knowledge of its right value needs no larger scope than this component itself. So, no other component can change the value of that parameter.
 - **attribute** parameter: the required knowledge of the right value lies in another component (who uses the behaviour or configures it for other components to use) and the parameter value is *associated* to the component by another component, in a hierarchical dependency. That is, a hierarchical “parent” has determined what the value of that data is, and the current component can not change it, but uses it without question.
- through **Communication**, by any “client” component that is interacting heterarchically. Again, two complementary mechanisms exist:
 - **control flow** parameter: the knowledge of which of the available behaviours of the component is required by a client component lies with that client.
 - **data flow** stream data chunks: the time series of values on which the servicing component must act is provided by the client component, in whatever *heterarchical, peer-to-peer* way.

The example of a [low-pass filter](#) can illustrate all of the above: (i) its *properties* are the values in the different terms of the filter that are there all the time, and never change; (ii) its *attribute* is the desired order of the filter; (iii) a *control flow* parameter is the number of bits of resolution that the client component expects; and (iv) a *data flow* parameter is every data chunk in the time series that the client component provides to the filter component.

9.1.7 Architecture of Finite State Machines in an activity

Each activity architecture uses Finite State Machines for various reasons. As an [abstract data type](#) for [coordination](#), only the concepts of [state](#), [event](#) and [transition](#) are needed. But to make such coordination *executable*, some extra data is commonly required:

- *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. 2.6.8).
 - `final_state(s)`: one or more FSM [states](#) in which the FSM ends its [active](#) state and transitions back to its [deploying](#) state.
 - `state_history`: the [stream](#) of state IDs that the FSM execution has moved through.
 - extensions to the externally visible [event reaction table](#):
 - `onEntry`, `onExit`: these events are added to the table for each [transition](#), so execution behaviour can be triggered every time the [state](#) at the start of the [transition](#) is *exited* and the [state](#) at the end of the [transition](#) is *entered*. It is a best practice to use two events, because this allows the behaviour to be “owned” by the separate [states](#), and not by the [transition](#).
 - `onTrigger`: the [event_loop](#) triggers the `current_state` at regular intervals in time, and execution behaviour can be connected to such triggers by adding a [transition](#) from that [state](#) to itself.
 - a list of [callbacks](#): a callback is a function that the [event_loop](#) executes in lieu of the *coordinated* activities; the latter have *configured* their callbacks first, by [registering](#) these functions as clients for the [event_processing](#), together with the event reaction table that represents the events to which these callbacks will react.
- A typical usage is to realise reconfiguration of an activity, required by a state change in the FSM.

9.1.8 Best practice: separation of mechanism and policy — System composition

One of the major pragmatic problems to compose components into systems is that components often come with [hard-coded configuration](#) choices. The cause of the problem most often is that the component was developed with just one particular application context in mind, and that its developers hard-coded the configuration that was (hopefully) “optimal” and/or “obvious”, at *that* moment and in *that* context. To avoid the problem is (*mereologically*) simple: component models improve [composability](#) via the [separation](#) of [mechanism](#) and [policy](#):

- **mechanism**: what does a component (algorithm, process, agent, piece of functionality,...) do, irrespective of the application in which it is used? Often, mechanism is

subdivided into its **topological** sub-parts of *structure and behaviour*:

- **structure**: how are the parts of the model/software connected together?
- **behaviour**:³ what discrete and continuous “state changes” does each part realise?
- **policy**: **how to configure** the above-described mechanism (structure and behaviour), 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.

In a software engineering context, *mechanism* is sometimes referred to as *unopinionated* software, and *policy* as *opinionated* software, provided in *software frameworks*.

(TODO: good practice are kernel modules in operating systems (data flow plus LCSM plus configuration interface); bad practice are ROS components (data flow is only concern).)

9.1.9 Policy: vendors add value in Configuration, Coordination, Composition

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

9.1.10 Good and bad practices in deployment of coordination

The amount of computation and communication⁴ involved in the *monitoring functions* that feed coordination functions with “events”, is often orders of magnitude higher than the computations required in the functions that compute the transitions in the FSMs and Petri Nets that model the coordination behaviour. So, the decision about where to deploy which monitors has a high impact on the overall system performance. Here are some good and bad deployment choices:

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

Good: to put such a monitoring computation into another activity than that of the FSM’s or the Petri Net, and interconnect them with an event communication. While at the same time designing in extra states and logic rules in the FSM/Petri Net that make their behaviour robust against the undeterministic computation and communication timing of the monitoring events.

For example, if another activity is needed to make a decision, based on the inputs of several other activities, the Petri Net should model *when* the other activities have

³Or *function*.

⁴For example, before deciding to accept a “contract offer” during a task coordination, the accepting activity may need to check whether, and under which conditions, it should commit itself. That checking can involve a lot of interactions with resources or other activities, and computations to estimate costs and returns.

provided their inputs to the decision making, and *when* the decision making activity has made a decision. The data about the decision itself (inputs as well as outputs) are to be found elsewhere in the architecture, in *data channels*.

- *Bad*: to add the “guard” (monitoring) computation to a transition, since that makes the transition take 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`.

9.1.11 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: the policy *decides* what value to give to the attribute. Here are some all too common examples (hence the name *bad practice*) where an attribute was set by a component designer *as if it were a property* of that component:

- a control gain is *not a property* of a *controller*, but an *attribute* given by the *task* that needs the controller to improve a particular *task-dependent* performance metric.
- *colour* is not a property of an *object*, but of the *relation* that connects the material properties (texture, paint,...) of that object with the *lighting conditions* and the *visual perception properties* of a camera.
- *the shape* is not a property of a link in a kinematic chain, because various applications will require other shape representations for the same link. For example, to make fast computations with only a first-order accuracy, or to add a specific mesh for collision deformation simulation, etc.
- the identity of the peer components with which a component interacts, is an attribute that is given to the component via some sort of *mediation*. The same goes for the streams via which the component interacts.

9.1.12 Vertical and horizontal composition

Vertical composition (or, “*internal composition*”) is about composing **activities** into one single **component**, at various **levels of abstraction**, and at various levels in the **association hierarchy**. “Single” means that (i) all the composed activities are always deployed together, and (ii) they are not visible anymore as individual activities to the “outside world”, because only the top-level component interacts with that outside world. Vertical composition is also possible, with the same design approach, for various *algorithms* in one *activity*, and for various *functions* in one *algorithm*.

Horizontal composition (or, “*external composition*”) is about composing **components** into a (**distributed**) **system**. “Distributed” means that:

- components communicate over networks (in the broad meaning of the term).
- oftentimes so-called **middleware** components bring extra integration challenges into the system (with as expected added value their provision of tailor-made solutions).

9.1.13 Lifecycle of compositions

Composition of activities into components, or components into systems, need not be static. Especially in robotics contexts, the variability in tasks and environments increasingly require

runtime re-composition. This can only be realised if re-configuration is explicitly designed-in as one of the **lifecycle phases** of a composition. Re-configuration encompasses horizontally and vertically composed behaviour inside a component, as well as the behaviour of a component that is made visible to the component's outside.

9.1.14 Block-Port-Connector for components, activities and functions

In multiple places, this document refers to the major building blocks (or “levels of abstraction”) for “behaviour”: **functions**, **algorithms** and **activities**, composed together into **components**. At each level of composition, a component exposes a part of its *inside* behaviour to the *outside*, that is, to other components, via **ports**, so that that behaviour can be *connected* to the behaviour in those other components. Inevitably, and at the same time, ports also realise **information hiding**: the system designers can decide to keep details about the *inside* behaviour hidden behind a component’s ports, whenever there is an advantageous trade-off between (i) the reduction of system-wide *interaction complexity*, and (ii) the *freedom of choice* to the design and implementation of components’ insides. This composition freedom can be realised in many different ways, and the **forces** that determine the behaviour of the composition are:

- *encapsulation, information hiding, security.* For example, in the context of **functions**, the representation of these various forms of data “*protection*” is possible by dedicated constraints on the accessibility of the **D-blocks**. This is best done by composition of the algorithm’s **A-block** with a **BPC model** in which some **Ports** are connected to selected **D-blocks**, **F-blocks** and/or **S-blocks**, and those **connect** relations get **attributes** that model the kind of “*protection*” to be composed in.
In other words, the *closure* of the algorithm is not defined by the functional developer, at design-time, but by the component supplier, who provides the port-based view on the algorithm that fits best to the concrete context in which its functionality is to be used.
- *run time adaptability:* there is no hard technical constraint that prevents the just-mentioned port-based *views* to be adapted at runtime.
- *resource management:* the execution of an algorithm makes use of two resources of the computer hardware, namely its *memory* and its *CPU*. Because the meta model contains explicit and complete information about the memory requirements (size as well as access constraints) of *D-blocks* and the execution sequences of *F-blocks*, an application developer can provide tooling to deal with possible **exhaustion** of those resources; e.g., the various management policies for **data buffers** or **stacks**, and for **iterators** or **execution schedules**.

9.2 Event handling association hierarchy for Configuration and Coordination — Document Object Model

This Section introduces the meta model of the *DOM* (or **Document Object Model**), as a **mature standard** that provides a meta model to connect an **computable object** to a **tree-structured hierarchy** that models the *discrete parts of system behaviour*, in the form of **coordination and configuration dependencies** between a set of components. That hierarchy **conforms to the meta meta model** of the **knowledge association hierarchy**:

- *data*: the data part of the DOM consists of (i) the *values* of the parameters with which “to tune” the components’ behaviours. and (ii) *events*
- *information*: the are the dependency relations that are encoded inside the DOM.
- *knowledge*: the **higher-order** coordination and configuration relations, that allow the decisions (made by humans at development time, or by the system itself at runtime) to be made about which DOM model to use at any given time.
- *wisdom*: the *human* developer has the insights in the application and its context, to encode the “right” knowledge relations into the DOM hierarchy.

A DOM model represents how to turn that discrete behaviour information in the model into software execution, and how software can **create, read, update and delete** information in the model.

A DOM *does not execute* activities itself, nor does it provide the *architecture* of a number of interacting activities. But it represents the entities and relations involved in the “discrete control” execution of such a set of activities. That is, in the systems engineering context of this document, a DOM is a *meta model* that provides the **dependencies** between the structural and behavioural parts of activities, and those dependencies are **conforms to the 5C meta model**, as least as far as its **Coordination** and **Configuration** parts go.

The “document” part of the term “DOM” is a bit unfortunate in the context of systems engineering, but the term has achieved such widespread acceptance, in application domains that work with *documents* of all kinds, such as books and articles—and especially on the Web—that it is kept unchanged, despite the semantic confusion it can introduce. An alternative name that would conform better to this document’s semantics would be “DBOM”: *Discrete Behaviour Object Meta model*.

9.2.1 Mechanism: DOM model structures the decision making

A DOM is the hierarchical structure (as sketched in Fig. 9.3) of (an explicitly identified subset of all) system components, and it represents the **scope** and **structure** of the decision making **command hierarchies** between the components in the same DOM structure. **Standardized** DOM *meta models* exist,⁵ through which one can change, at runtime, the hierarchical structure that a *conforming model* represents. The most widespread use of DOMs is now in **Web browsers**, **graphical user interfaces**, and **declarative markup languages**⁶ like **HTML5**, **SGML** or **GML**. But they apply perfectly well to robotic and cyber-physical systems, too.

Figure 9.3 shows the **mereo-topological** entities and relations of a DOM model:

- **element** (or, **tag**, or **node**): an *element* is syntactically scoped by an opening tag `<comp...>` and a closing tag `</comp>`,⁷ and it represents a *component* in the DOM model. That component’s contents, between the markup tags, consists of a tree structure of other *elements*; the behaviour of these contained *elements* is influenced by the containing *element*, in various ways, as explained in the paragraph below about an *element’s attributes*.

⁵ As well as rich ecosystems around them, providing software tooling and programmatic interfaces for offline and online use.

⁶ Because XML is the most common host language for DOM markup languages, the example in this Chapter also adopt an XML syntax. Semantically, the meaning of the elements and attributes can be mapped one-on-one to the JSON-LD host language that was used in previous Chapters.

⁷This is an arbitrary syntactical choice, but an effective one. It has been made popular by markup languages like **SGML**, and was inherited by **HTML**.

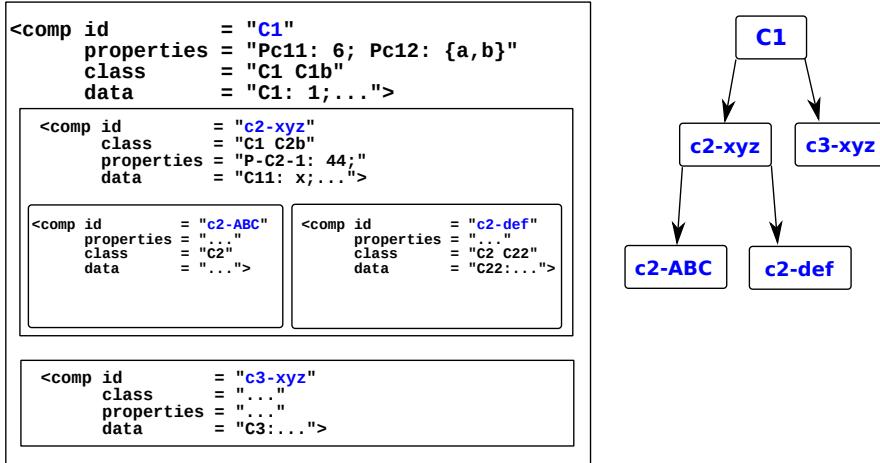


Figure 9.3: Example of tree-structured hierarchy in a Document Object Model. The tree structure provides the *scope* for the *configuration tags* (*class*, *data*) in the component descriptors, and for the *event handling* (see Fig. 9.4).

- **containment** relation: the components are ordered according to a tree hierarchy. This tree structure involves a form of the *Block-Port-Connector* meta model, where “connected-by” means “is-contained-in”, or *is-descendant-of*. The obvious inverse connection relations are “contains” or “*is-ancestor-of*”. The containment relation is depicted graphically in Fig. 9.3, but it is a *best practice* in text-only representations to use the nesting of matching tags to represent containment:

```

<comp id="C1" properties="Props1" class="C1 C1b" data="C1d:...>
  <comp id      = "c2-xyz"
    properties = "P1: A; Props-c2-xyz: 4;"
    class     = "C1 C2b"
    data      = "C11: 4;">
  <comp id      = "c2-ABC"
    properties = "Props-c2: 5;"
    class     = "C2"
    data      = "C11b: ...;">
  </comp>
  <comp id      = "c2-def"
    properties = "P1-s P2xx: ...;">
    class     = "C2 C22"
    data-C11  = "C112: ...;">
  </comp>
</comp>
<comp id      = "c3-xyz"
  properties = "..."
  class     = "C3 C33"
  data      = "C3: ...;">
</comp>
</comp>

```

- **attributes**: as seen in the example above, a starting `<comp...>` tag contains the *attributes* of the component, that is, its meaning, role, purpose,... These **attributes** come in four complementary *types*, each with its own *semantic interpretation*:

- **id**: this attribute is **given** by “something” beyond the scope of the DOM hierarchy; often going as “high” up as the human developers.
- **class**: these attributes are **owned** by an *element*, but their **values** can also be **inherited** from the same **class** attributes that appear in any of its “ancestor” elements. That is, the components in the DOM model that are positioned higher up in the DOM hierarchy. When several of the component’s ancestors provide the same **class** attributes, the one of the closest ancestor takes precedence.
- **property**: properties of a component are fully **owned** by that component, irrespective of whatever “ancestor” or “descendant” components exist; in other words, the property values depend on that component and only on that component, but not on the position of that component in a system, nor even its position in a DOM hierarchy;
- **data**: these parameters are **used** by the element, and are **given** a value at runtime, by other components anywhere in the system, via some form of **data exchange**. The behaviour of such data exchange is not specified in the DOM model, directly. Or rather, it is within the scope of *another* DOM model, namely that of the data exchange itself, as a first-class citizen of the component model of the application.
- **shadow** DOM relation: the dependencies between a DOM tree and one of its sub-trees can be **encapsulated** explicitly, by the **higher-order relation** “**shadow**”. This “**shadow**” relation is a **barrier** on the **propagation** of configuration and coordination through the tree structure. This has only an impact on the **class** attribute.

A DOM representation structures how a system makes decisions about:

- the **value of parameters** in the components of the system, e.g., in the *configuration* and *coordination* of their *composition*, *communication* and *computation*.
- which components have **to react to events** generated elsewhere in the same DOM structure, and in which order these reactions must take place.

The tree structure of a DOM model supports three behavioural models, two **hierarchical** and one **heterarchical**:

- the way “**magic numbers**” **cascade** from the “top” of a DOM model to its “bottom”. That is, (i) a parameter defined at a “higher” level in a DOM model also applies to any “lower” level, but (ii) the value of that parameter can be overwritten by the “lower” level. For example:
 - the top level of the DOM model for the torque controllers in all actuators of a robot can specify *one* error tolerance value for *all* controllers lower in its tree.
 - the decisions about how big the buffer must be in **stream buffers**. Or about the reaction to *HighWater/LowWater* events and to the *active* flag.
 - the decision about the best thresholds for the different guards in a set of **guarded motions** are not (necessarily) to be taken by the components involved in the execution of the guarded motion.
- **coordination** by event handling via **capturing** (top-down) and **bubbling** (bottom up): when a monitor triggers in one of the components in a DOM model, the model can specify how and where other components in the same DOM model must react to that same event, irrespective of whether they are ancestors, siblings or descendants of the triggered component. For example, when one of the above-mentioned torque controllers in a robot notices that its actuator has reached its saturated state, the DOM model can encode how to change the other torque controllers, but also the controllers elsewhere in the robot’s **motion stack**.

- the DOM tree **structure** can be used to attach the *identifiers* of the *ports* that components have to connect to, for their data exchange with (possibly *heterarchically* organised) other components in the DOM. This means that the implementations of those components need not have those identifiers built in; they just have built in the *place* in the DOM where they have to look for these identifiers.

9.2.2 Policy: decision making in a context

The DOM meta model serves multiple purposes:

- the DOM hierarchy provides a **context**: it allows the decision making in one component **to take into account** its dependencies on other components, **without having to know** anything about the components that introduce the dependencies. For example, the context can relate the value of a gain in a controller algorithm to the actual or tolerable uncertainty in a perception algorithm.
- **multi-inheritance of contexts**: one component can be part of **multiple DOM** models, for complementary decision making purposes. For example, the same above-mentioned controller gain can also be in relation to the desired safety level of the robot system. And yet another context can provide the **higher-order** trade-off relation between both controller gain dependencies.
- the structure is a **tree**, so **efficiency of implementations** is guaranteed.
- the DOM model can be part of a component's **data sheet** because it is a formal model that can be encoded in any human-readable host language.

9.2.3 Policy: syntax, custom tag, paths, diffs and multi-tree

A common purely **syntactical shortcut** is the introduction of **custom tags**:

`<comp id="C1" properties="Props1: x;" class="CustomClass C1b" data="C11:...;"/>`
is shortened to

`<CustomClass id="C1" properties="Props1: x;" class="C1b" data="C11:...;"/>.`

Another equivalent syntax could be like this:

```
<CustomClass>
  <id>C1</id>
  <properties>Props1: x;</properties>
  <class>C1b</class>
  <data>C11: ...;</data>
</CustomClass>.
```

Because the descendants of an element can be elements in themselves (e.g., a **<property>** can have its own **id** and **class**), constructions like the following are possible:

```
<CustomClass>
  <id>C1</id>
  <properties> <id>prop347</id> <class="prop-class-Z"> </properties>
  <class>C1b</class>
  <data> <class="integer">C11: ...;</data>
</CustomClass>.
```

Because a DOM model is built with only tree structures, it is straightforward to transform the textual syntax used in the examples above into any other syntax representing the same tree structure, in **XML**, **JSON**, **JSON-LD**, etc. That tree structure implies that implementations

can rely on *fast indexing* of elements and tags, represented by a simple grammar, such as [XPath](#) or [UNIX file paths](#). For example, starting from the top of the hierarchy of the DOM model above:

`C1/c2-xyz/c2-ABC`

Or with relative paths, for example starting from `c2-def` to reach its *sibling* `c2-ABC` or its “nephew” `c3-xyz`:

`../c2-ABC ../../c3-xyz`

Such paths also allow for efficiently finding *structural differences* (“*diffs*”) between two DOM model trees.

Finally, one can allow one particular component to appear in several DOM models at the same time, as a means to provide different contexts to that component. The more precise semantics of “to be in” is that several DOM models can *refer to* the same component via that component’s unique ID. In other words, it is straightforward to extend a DOM model from a single tree to a [multitree](#). This policy is:

- **composable**: (i) one model’s *root* component can be added to any other DOM tree as a new *leaf* node, and (ii) such a composition can add new class or data attributes, or overrule existing ones, respecting the standardized cascading constraints.
- **compositional**: dependencies are explicit and strictly hierarchical.
- **robust**: the same model entities (like class attributes) can be defined at different levels of the DOM hierarchy, and this *redundancy* gives system designers opportunities for extra consistency checks during composition of DOM models.

9.2.4 Cascading Style Sheets (CSS) represent top-down parameter configuration

Figure 9.3 shows the tree-structured containment of component representations in a DOM model. The *cascading* semantics⁸ described in this Section pertains to how the parameters in the `class` attributes get their values, via a configuration structure that follows the tree structure of the DOM model. The relevant [mereo-topological](#) entities and relations are:

- *inheritance* of values of attributes is top-down [according to](#) the tree structure. The `class` attributes are interpreted as [Semantic IDs](#) of meta models, so [multiple conformance](#) applies to their composition.
(TODO: give examples.)
- *selectors* can be added [to fine-tune](#) the inheriting behaviour.
- *variables* and *data attributes* can be added to configure the same values in different elements. They can be computed at runtime, and stored in [local or remote storage](#).

Both meta models are fully declarative, hence supporting **composability**.

9.2.5 Event bubbling represents bottom-up reactivity

Figure 9.4 shows bottom-up propagation of [event handling](#) in the containment tree structure of a component DOM model. This type of propagation is called *bubbling*.

For example, when one of the torque controller activities in a robot notices that its actuator has reached its saturated state, the DOM model structures in which order the other torque

⁸The origin of the name of this Section lie in the [World Wide Web](#), with [Cascading Style Sheets](#) (CSS) being one of its core components, together with [HTML](#), [JavaScript](#), and [SVG](#).

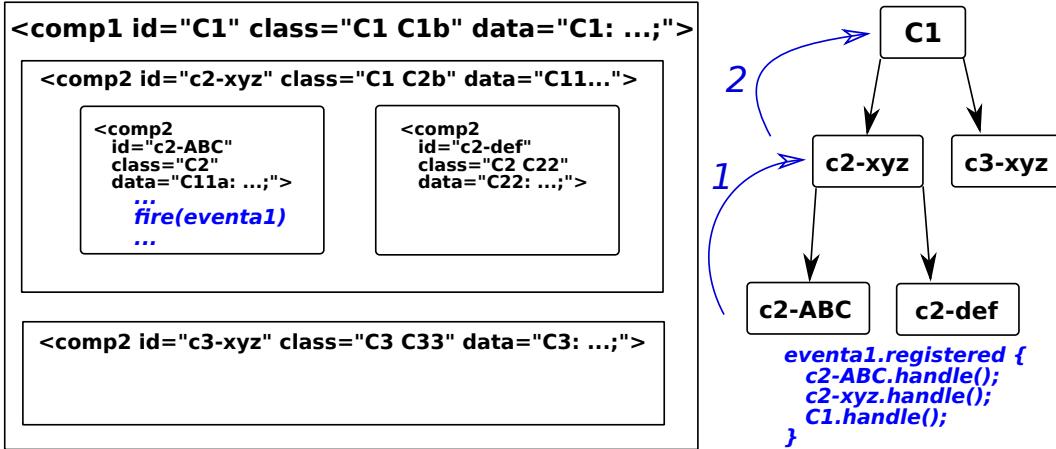


Figure 9.4: **Event bubbling** in the DOM of Fig. 9.3

controllers react to this situation; and even controllers or other activities elsewhere in the robot's motion stack.

The mereo-topological entities and relations of `events` are:

- `event`: has one unique `type`, and a (possibly ordered) list of targets that have registered to be informed about the occurrences of the event.
- `event_target`: a component that wants to be informed about the occurrences of an event.
- `event_dispatch`: the mechanism that delivers the occurrence of a particular event to each target that is registered to that event.
- `event_listener`: the function that a target executes when the event is dispatched to it.

In a composable system architecture, all four aspects are kept *loosely coupled*: the decisions about *how to implement* one of them should not influence the decisions about how to implement the others. Of course, the coupling that can not, and should not, be made loose is that about the *meaning* of each event type. A DOM model couples the responsibilities for:

- the *semantics* of event types.
- the *structure* that the event dispatching must satisfy.
- the *configuration* of target and listener activities.

9.2.6 Petri Net coordination for event bubbling

Figure 9.5 sketches how the concept of `event bubbling` is realised by a Petri Net model that must be defined behind the "screens" of a DOM model:⁹

1. the *conceptual* arrow of event bubbling "comes in" into a component *A*.
2. this conceptual "event" is represented by putting a `token` into the *incoming place* of the component's representation in the DOM.

⁹The DOM model represents *what* coordination is expected. The Petri Net model represents *how* the *execution* of that coordination is *structured*. The architecture still needs a *software implementation* of the Petri Net, and of the event functionalities (such as transition decision making, and event listening), before the DOM model can effectively be *executed*.

3. some application-specific **mediator** activity decides when, and for which reasons, the **transition** is realised. The DOM can configure the **transition** decision function in that mediator.
4. after the **transition** has taken place, the **handled place** is filled, and **incoming** is cleared.
5. the mentioned mediator decides that, for a certain reason, the event is completed for component **A**, and the bubbling upwards towards component **B** is realised by filling the latter's **incoming place**.
6. similarly as in **A**, the mediator decides when, and for which reasons, the transition in **B** must be made.

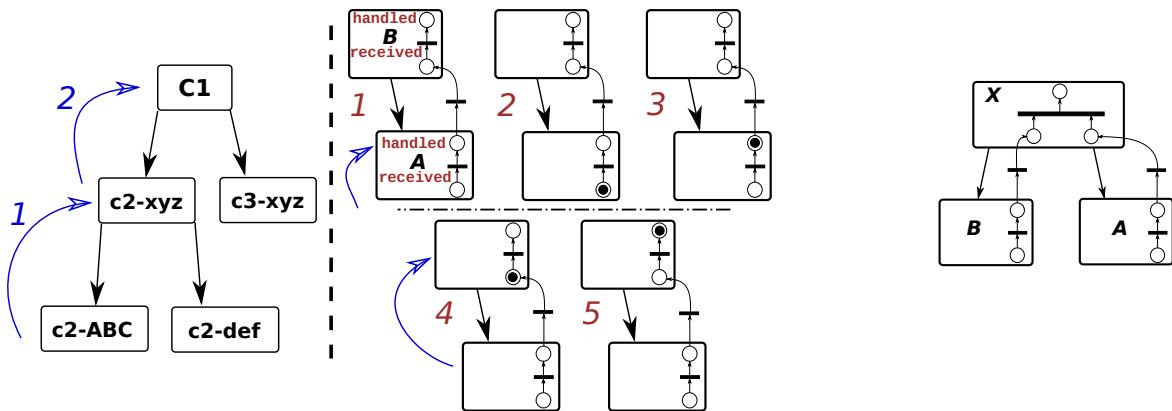


Figure 9.5: **Petri Net** models of the coordination behind event bubbling.

Left: single upwards propagation. Right: joining of two upwards propagations

9.2.7 Event capturing represents top-down reactivity

Figure 9.6 sketches top-down propagation of **event handling** in the containment tree structure of a component DOM model. This type of propagation is called **capturing**.

Event capturing extends event bubbling (and *requires* it as a first step), because it allows “higher-level” components to react first to events in “lower-level” components. For example, when the torque controller activity for one of the robot’s joints notices a saturation, it can first propagate this event up to the “top”, where that top component decides how all the influenced controller should react, by a top-down parameter configuration propagation. Only after that configuration, the individual lower-level control activities actually react to the event.

9.2.8 Petri Net coordination for event capturing

The coordination behind the screens of event capturing is not just on top-down iteration through the DOM tree, but rather a set of three sweeps (Fig. 9.7):

1. first bottom-up: from where an event is created to the highest level in the DOM where a reaction needs to be realised. In this “bubbling”, no decision making is done already, but only the fact that a particular event has taken place is communicated.
2. top-down: the highest level can configure the the lower-level components’ reaction to the event, by the **cascading** mechanism.

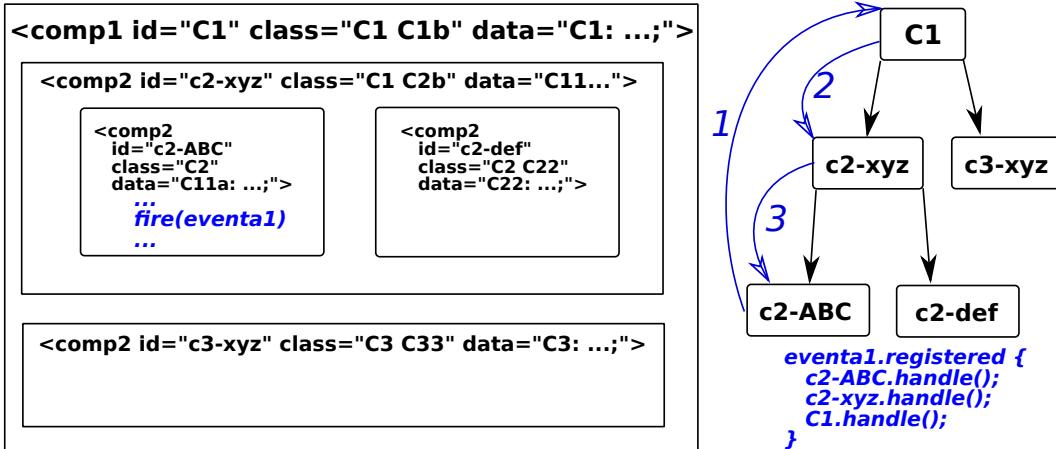


Figure 9.6: **Event capturing** in the DOM of Fig. 9.3

3. second bottom-up: this is the “normal” **event bubbling** mechanism.
Of course, both event bubbling and event handling policies can be combined in the same DOM.

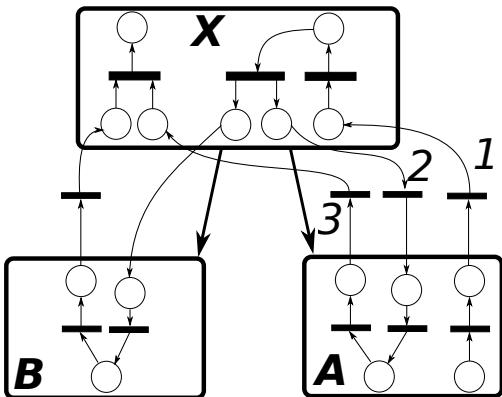


Figure 9.7: **Petri Net** models of the coordination behind event capturing.

9.2.9 Components are not *in* a DOM, but their behaviour is *represented* by it

This Chapter deals with *modelling* of systems, not of **information architectures** or **software architectures**. So, every time that a **component** is mentioned in the models of this Chapter, that model *represents* a component that has complementary information models, and that is somewhere executed inside a software activity. That representation can also be:

- *partial*: not all aspects of a component are represented in the models of this Chapter in which it is referred to.
- *multiple*: the same component can be represented in several models of this Chapter, for other purposes.

The role of the *information architect* is then to decide in which models each of the system’s components need to be represented, and *what* data it must exchange with other components. The role of the *software architect* is to decide in which software activities a component needs

to be executed, and *how* it can exchange the data with other components.

9.2.10 Graph connections as constraints between DOM model trees

By means of the systematic use of [symbolic pointers](#), or [Semantic IDs](#), one can add constraint and tolerance relations between entities and properties of relations represented in more than one DOM tree structure, and in more than one DOM document. The result is a **graph-structured** coupling between **tree-structured** DOM models.

The system designers must encode the knowledge about the system, its applications and its context, with the “right” composition of:

- DOM models: their **tree structure** brings **efficiency** to the **decision making**, about (i) the values of “magic numbers” in the components represented in the DOM model, and (ii) which of these components reacts to which of the events represented in the DOM model.
- graph constraints between these DOM models: they bring the **semantic richness** that can be used in the above-mentioned **decision making** functions.

9.2.11 DOM models for activities, algorithms, functions, data

This Section introduces concrete DOM models to the meta models of the building blocks for activities: [data](#), [functions](#), [algorithms](#), [data streams](#), and function execution [event loops](#).

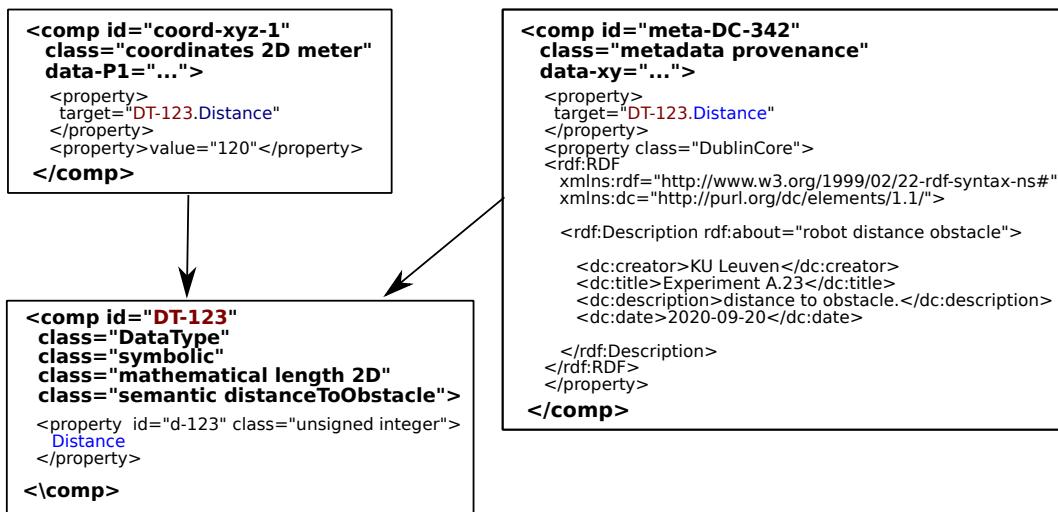


Figure 9.8: Sketch of a DOM model for a [data type](#), with a relation to (i) a [digital structure](#) DOM for its coordinates, (ii) a [metadata](#) DOM.

Figure 9.8 sketches a DOM for a [data type](#). Standardized DOM meta models with (more or less) the semantics of what this document advocates exist, such as [Apache Arrow](#) or [HDF5](#). This document assesses these standards as mature and stable enough for widespread adoption.

Figure 9.9 sketches a DOM model for a [function](#), and Fig. 9.10 that for an [algorithm](#). The latter DOM model adds *dependencies* between functions, in the form of [constraints](#) on the execution order of the functions in the algorithm:

1. constraints on the [serial](#) and [parallel](#) execution of functions inside an algorithm.

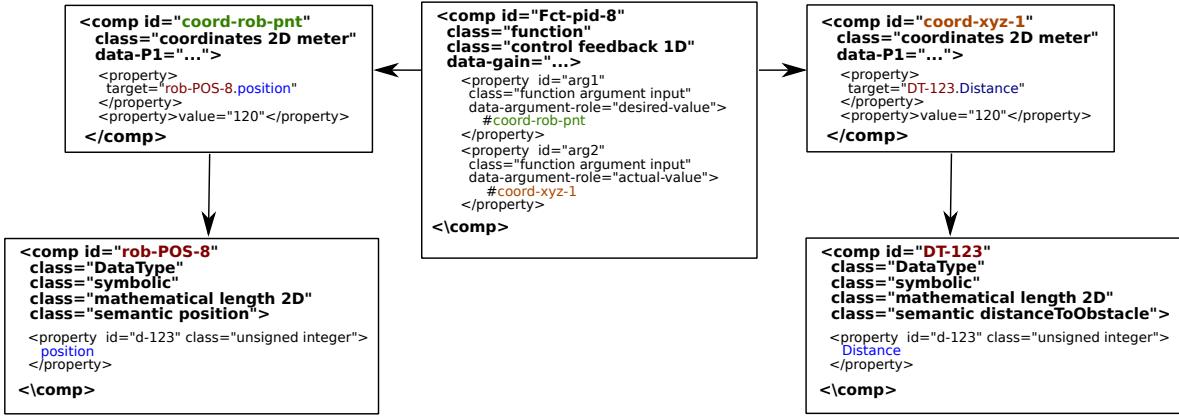


Figure 9.9: Sketch of a DOM model for a [function](#), with three arguments, each being a data structure as in Fig 9.8. The relation to the function’s metadata (not shown in the drawing) is similar as in the latter Figure. The arrows and colours are not part of the textual DOM, but are added to the drawing to help the reader follow the symbolic pointers inside the various DOM elements.

2. constraints on the [conditional branching](#) in an algorithm’s [control flow](#).
3. constraints on the access to the data arguments of the functions.
4. [Life Cycle State Machine](#) constraints on the selection of which (sub)algorithm to activate at any moment.
5. [Petri Net](#) constraints on the configuration of *flags* in the algorithms. These flags influence (part of) the above-mentioned conditional branching, for all those conditions whose logical value depends on data that is updated by algorithms “elsewhere” in the system.

The first two items are not in the focus of this document; and to the best of the authors’ knowledge, no widely supported standards exist.¹⁰ However, these constraints, and also the third one, are just special imperative cases of the more declarative models of state machines and Petri nets. These more complex mechanisms are *inevitable* anyway whenever algorithms, functions and data have to be composed in concurrent or asynchronous deployments.

The *generic* DOM model of the *algorithm* entity (Fig. 9.10) is the (little bit of) hierarchical structure that exists between (i) functions and data (at least, the *producer* and *consumer* functions of data streams), and (ii) the coordination of functions via flags, state machines and Petri Nets.

An [activity](#) composes:

- one or more [algorithms](#), to process data in a way that adds value to the application.
- one or more [streams](#), to exchange data between algorithms (irrespective of whether all of the algorithms involved run inside one single activity or multiple activities).
- one [event loop](#), that serializes the [4Cs](#) of all the algorithms that run inside the activity.

Streams and event loops *couple* data and functions together, in **heterarchical** ways; hence, they are the complement of the **hierarchical** structure with which one decouples *data*, *functions* and *algorithms*. The *activity* is the “**holon**” where all this comes together: the activity

¹⁰It is not even clear whether such standards could add value.

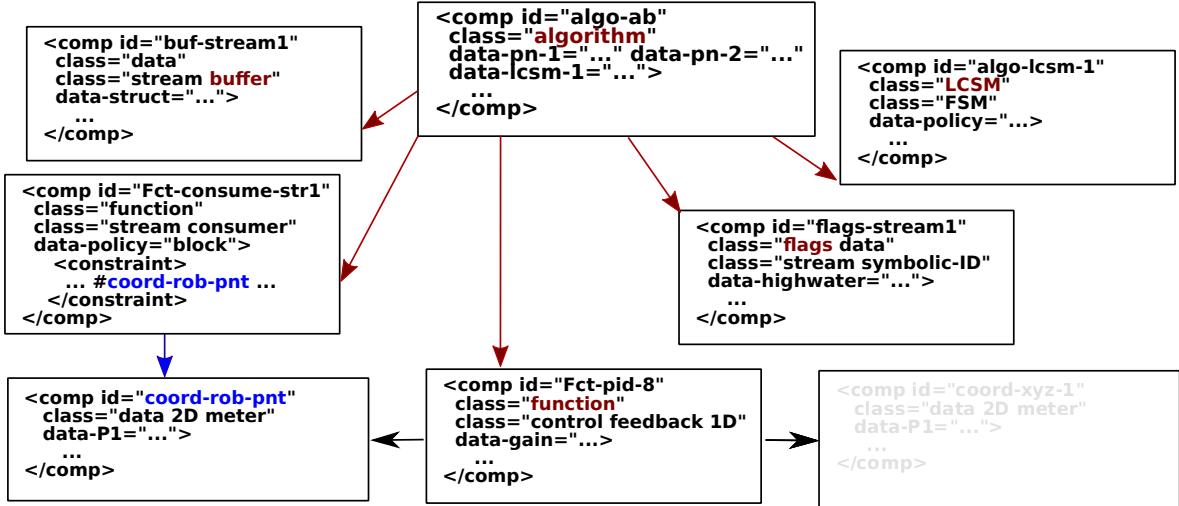


Figure 9.10: Sketch of a DOM model for an [algorithm](#), where the top-level `algorithm` element is the parent composition of several children components. One of those children, the *producer/consumer* part of a stream, provides a constraint on the first data argument of the function. Access to the function's second data argument is not constrained. The stream's *produce* function is taken out of the scope of this Figure.

must *know* about which couplings are necessary for the application, to take the right decisions how how and when to configure and to coordinate streams, event loops, and algorithms. This decision making itself is structured in a hierarchical DOM. Whenever a stream represents *heterarchical* couplings between several activities, there is a need to introduce a *dedicated activity* to take ownership of the *inter-activity* decision making about that stream. Figure 9.11 sketches the structure of the DOM of a stream:

- the stream *coordinates and configures* the stream's (i) buffer, (ii) its coordination flags, and the (iii) *consume* and *produce* functions that get data in and out of the buffer.
- the activity must *connect* the stream functions to the data that the algorithm functions work with, making sure that data consistency *constraints* are met.
- there are many *mechanisms* and *policies* in streams whose configurations must be set appropriately for the application context.

This structure models the configuration and coordination of a stream's "outer" part, that is, those aspects that can not be done by the algorithms that use the stream and that own the configuration and coordination of the "inner" parts. The terms "inner" and "outer" get their meaning from the meta meta model of the [Block-Port-Connector](#).

... (TODO: include figure.)...

Figure 9.11: Sketch of a DOM model for a [stream](#) which coordinates the *data access* between two functions: (i) a "useful" function for the application (i.e., the one in Fig. 9.9) and (ii) a "infrastructure" function that consumes the stream for the "useful" function and makes the stream data accessible.

Figure 9.12 sketches the structure of the DOM of an **event loop**:

(TODO: DOM of event loop; DOM of activity composing the DOMs of stream and event loop.)

... (TODO: include figure.)...

Figure 9.12: Sketch of a DOM model for an **event loop** which coordinates the *execution order* between functions.

9.2.12 DOM model of Block-Port-Connector relation

Figure 9.13 depicts the generic DOM model for two **Blocks** connected via one **Connector**, making the inherent (shallow) hierarchy explicit. The approach of attaching **ports** to the entities that are connected with separate inside and outside **docks** decouples the connected entities in such a way that they only have to provide to other entities the *possibility* to be connected with externally visible behaviour, without having to know the details about the internal behaviour.

(TODO: more concrete examples.)

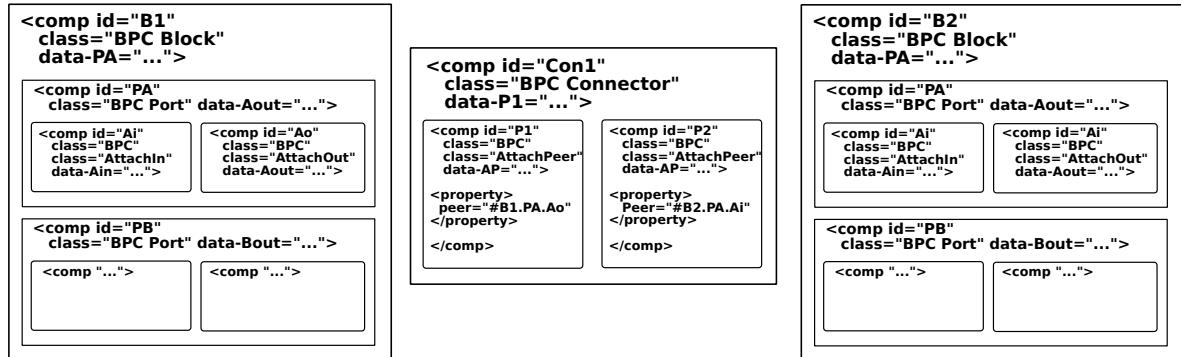


Figure 9.13: Generic DOM model for two **Blocks** being connected by a **Connector**, via **Ports**. The **Connector** “points” to the **Blocks**’ outside-facing **Port** attachments **docks** via the symbolic pointers `#B1.PA.Ao` and `#B2.PA.Ai`.

9.2.13 DOM model for relation-constraint-tolerance models

The composition pattern of **relation**, **constraint**, and **tolerance**, has a DOM formalization as sketched in this generic example:

```

<relation id="R1"          // A relation of type relA
  class="relA"            // with two arguments
  data-pars="max">        // and one constant parameter
  <property id="propA"   class="role-input1"></property>
  <property id="propB"   class="role-output1"></property>
  <property id="MaxVal"  class="role-parameter" value="max"></property>

```

```

</relation>

<relation id="C1"          // A constraint relation of type
  class="constraint1"      // "constraint1" with parameter cM
  data-constr-par="cM">    // on two properties of relation R1:
  <property id="arg1"  class="role-subject"> #R1.propA </property>
  <property id="arg2"  class="role-object"> #R1.propB </property>
  <property id="limit" class="role-parameter"> cM </property>
</relation>

// A constraint relation of type "cnstrX", with parameter cc,
// on one property of relation R1:
<relation id="C2" class="constraint cnstrX" data-constr-par="cc,dd">
  <property id="arg1"  class="role-subject"> #R1.propB </property>
  <property id="limit" class="role-parameter"> cc </property>
</relation>

// A tolerance relation of type tolZ on two properties
// of constraint relations C1 and C2:
<relation id="T1" class="tolerance tolZ" data-tol-par="tol1">
  <property id="arg1"  class="role-p1"> #C1.data-constr-par.cM </property>
  <property id="arg2"  class="role-p2"> #C2.data-constr-par.cc </property>
  <property id="limit" class="role-tol-limit"> tol1 </property>
</relation>

// A tolerance relation of type tolX on one property
// of constraint relation C1:
<relation id="T2" class="tolerance tolX" data-tol-par="tol2">
  <property id="arg1"  class="role-X1"> #C1/.ata-constr-par.cM </property>
  <property id="limit" class="role-tol-limit"> tol2 </property>
</relation>
```

Not surprisingly, all these expressions have the same DOM structure, which is that of a *relation*. Their semantic differences are expressed by the *attributes*. The various models present a **semantic hierarchy**, via the **dependencies between parameters** in the different expressions. Indeed, the **constraint** and **tolerance** expressions *refer to* the **relations** they pertain to, using their symbolic *IDs*. So, the hierarchically “highest” expressions are the **tolerances**, because specifying a tolerance only makes sense if there is a constraint relation with parameters; and such a constraint relation is meaningless without “normal” relations that can be constrained.

However, this semantic hierarchy between relations, constraints and tolerances is **not a DOM model in itself**:

- the parameters in a tolerance relation are not used *to configure* parameters in a constraint relations.
- similarly, the parameters in a constraint relation are not used *to configure* parameters in a “normal” relation.

9.2.14 DOM model for a guarded motion specification

Section 6.3.3 introduced the “**DSL**” version of a guarded motion specification, for example:

```

MOVE:
  1: d.origin in direction d.z
CONTEXT:
  1: Pre  { dist-orig, height-max },
  2: World { a, b, c, d },
  3: Spec  { vel-min, angle-min },
  4: Post  { dist-max }
ID:   guarded-move-velocity-23f5e3df
TYPE: {velocity-specification, frames, 3D}
WHEN:    // pre-conditions
  1: d.origin further than Pre.[dist-orig: 50 cm] away from a.origin
  2: d.z is larger than Pre.[height-max: 75 cm]
WHILE:    // per-conditions, to be satisfied during the whole MOVE
  a: keeping d.origin.speed between Spec[vel-min: 0.1 m/s] and Spec[vel-min: 2*vel-min]
  b: keeping d.origin further than Spec[height-max: 50 cm] away from a.origin
  c: keeping b.q angle larger than Spec[angle-min: 0 degrees]
  4: keeping c.q angle larger than Spec[angle-min: 10 degrees]
UNTIL:    // post-conditions, i.e. reasons to stop the MOVE
  x: d.z is smaller than Post[dist-max: 75 cm]

```

The same information structured along the DOM meta model syntax used in earlier Sections in this Chapter looks like:

```

<comp id      = "..." 
      class     = "guarded-motion MOVE"

  <comp id      = "..." 
      class = "CONTEXT">
    <comp id      = "1" class = "Pre: {...};">
    <comp id      = "2" class = "World: {frame: a,b,c,d};">
    <comp id      = "3" class = "Spec: {...};">
    <comp id      = "4" class = "Post: {...}">
  </comp>

  <comp id      = "..." 
      class = "WHEN">
    <comp id      = "1"
      class      = "inequality"
      properties = "LHS: d.origin.speed, RHS: ..." >
    <comp id      = "2"
      class      = "inequality"
      properties = "LHS: d.z, RHS: ...">
  </comp>

  <comp id      = "..." 
      class = "WHILE">
    <comp id      = "a" class      = "..." properties = "..." >
    <comp id      = "b" class      = "..." properties = "..." >
    ...
  </comp>

```

```

<comp id      = "..." 
      class = "UNTIL">
  <comp id          = "x" class       = "..." properties = "..." >
  ...
</comp>

</comp>

```

The different keywords in the specification represent different hierarchies:

- MOVE: this top-level component is just the *container* of all other components below, with only `id` and `class` tags in itself.
- CONTEXT: this is the *top level* of the hierarchy, because this is the entry point of **data** and **class attributes** that can “trickle down” into the other components of the specification (given below) that are involved in the task execution.
- WHEN:
- WHILE:
- UNTIL:

9.3 Activity to execute Configuration and Coordination — Mediator

For system developers, this Section is maybe the most important one in this document, because it *couples together* a large number of the models, patterns and best practices introduced before. It helps system developers in the many cases where they are confronted with the following design challenge:

- **decisions** have to be made about the **interaction behaviour** of a subset of the activities in the system.
- it has to be clear how the decision **impacts** the inter-activity interactions **to add value** to the application.
- **none** of the activities involved can make the decision **on its own**, because it **has incomplete information** about the situation.
- **ownership of the decision making** has to be unambiguous and **decoupled**: the system has **no** other subset of activities whose similar decision making about *their* inter-activity interactions leads to conflicting results.

The system architecture primitive that has emerged as the right one for this inter-activity decision making is the **mediator pattern**: an *extra* activity is introduced in the system architecture (Fig. 9.14), with the *sole* purpose of making *one particular* kind of decision about inter-activity behaviour.

Here are some examples where a mediator is indispensable. A fleet of **search and rescue** robots must inform each other, *and* the human-operated coordination centre, about their findings. This involves a lot of communication, and because communication bandwidth is always a scarce resource, some activity in each robot must decide which of the locally generated pieces of information will actually be sent, taking into account the available bandwidth as well as the current needs of the operation. Similarly, a set of robot manipulators share the workspace of one single assembly cell, and a decision must be made about which robot can use which part of that workspace, during which time intervals. Or finally, in both examples, all robots have

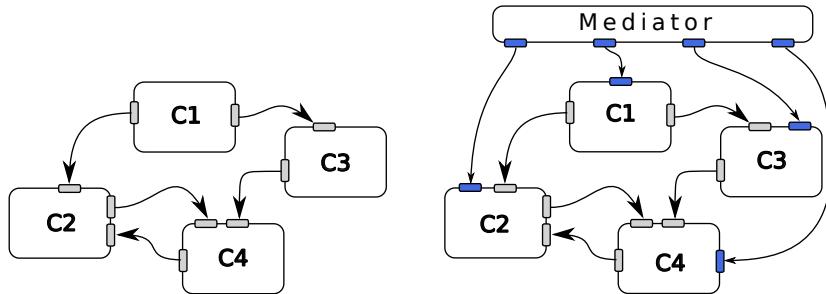


Figure 9.14: Left: unmediated components, with *heterarchical* data exchange. Right: Mediator component adds one level of *hierarchy*, to coordinate and to configure the behaviours of the other components via configuration and coordination ports.

sensors whose processing contributes to updating the information about the world around the robots, but decisions must be made about (i) which sensor processing results of which robot are integrated in the *world model state*, and (ii) which part of that *world model state* is made available to which robot.

The decision-making mediator is an *extra activity*, but the system architects can decide to host it on one of the involved robots. In other words, it is not necessary that an *extra agent* is introduced in the system to execute the mediation activity.

9.3.1 The mediator pattern in human society

Human society has introduced the mediator pattern many centuries ago already, via societal “system architectures” that are now considered as obviously resilient subsystems. Here are some examples where an extra *mediator* organisation or entity has been introduced to cope with the complexity of the increasing number of societal interactions:

- in the context of *tasks* to organise the many interactions between humans into a loosely coupled societal hierarchy, people organise themselves in families, they form neighbourhoods, make up villages, organise them into a metropolitan area around a central town, all assembled in regions and countries.

Each of these political structures has its own government as mediator.

- in the context of *tasks* that building infrastructures must support, rooms are connected by corridors, that form wards with some form of functional cohesion, aligned into floors, that form wings of buildings.

Each of these functional building structures has its own floor and task managers as mediators.

- in the context of *tasks* to organise a production industry, devices are interconnected with high-performance logistic networks into workcells; those form more loosely interacting production lines; these lines, in turn, are laid out in plants, that are logically interconnected and coordinated as a company.

Each of these factory compartments has its own **foreman** or task execution **supervisor** as mediators.

- in the context of *conflicts* between people and/or organisations, the mediator *helps* the disputing parties to come to decisions that can resolve their conflict. A large variety in forms of mediation exist, with respect to the amount of decision making that is done

by the mediator and by the mediated parties.

Other examples of societal instantiations of the mediators are: the *coach* in a team of athletes and staff; the *CEO* of a company; the *mayor* of a town; the *project manager* of a large public infrastructure construction; etc.

9.3.2 Meta model of a mediator

The *architecture* of a mediator *activity* (Fig. 9.14) represents how *to couple*, structurally and behaviourally, two or more other components. A *best practice architecture* allows *to configure* and *to introspect* the status of the architectural couplings between these components. More concretely, the mediator connects (some) *configuration* and *coordination* ports on each mediated component with some of its *own* ports, instead of directly with ports on the other mediated activities. Typically, *data exchange* ports between the mediated activities are left untouched, and not routed through the mediator, *unless* the mediating requires *changing data chunks* in ways that the mediated components can not.

The *mediator pattern*¹¹ contains the following entities and relations:

- introducing **one dedicated activity**,
- to satisfy an **explicitly specified** set of **constraint relations** (or “*dependencies*”),
- between the behaviours of an **explicitly specified** set of (the so called “mediated”) activities,
- by adding (only!) **one single layer** of **mediation constraints** (hence, it is appropriate to introduce a **DOM** as its model),
- and using this information to create (only!) **coordination** events for the mediated activities, and to compute the associated **configuration magic numbers**.

In other words, a **shallow hierarchy** of decision making is added (Fig. 9.14). (The *data exchange* between the *ports* on the mediated activities can remain **heterarchical**.) For the example case of the search-and-rescue robots, the decision about which information to broadcast, is taken together with the decision about which activities must run, and with which configuration settings; the communication itself can keep its original structure of peer-to-peer, or broadcasting, via a broker, etc.

Of course, the same mediator pattern can be applied several times on subsequent composition scopes, each time adding one level of hierarchy in decision making. (Again, this fits very well with the composable structure of **DOM** models.) For example, the traditional layout of a manufacturing plant has the composition levels of a workcell, an assembly line, a production unit, and the plant.

Such compositions of mediators result in a **military-like** hierarchical decision making *structure*, while allowing the following important design trade offs in the decision making *behaviour*:

- each decision can be **prepared** by more or less extensive exchange of information between the involved activities, cross-cutting the decision making hierarchy and allowing to reach agreements.
- each decision can be **monitored** by each involved activity, and disturbances can be reported, again possibly cross-cutting the decision making hierarchy.

As ever, finding the “right” balance between hierarchy and heterarchy [114] is often only achieved by the best human system designers. And introducing mediators in a system archi-

¹¹The article on this pattern on the Wikipedia does not cover the “(meta) meta” and “coordination” aspects, but focuses on the software engineering aspects only. This document’s focus is on the former aspects, as well as on more aspects than data communication only, in particular, on *coordination* and *composition*.

tecture often requires adapting the architecture of the mediated activities too: they must provide the appropriate configuration ports, must react to the appropriate coordination events, and must introduce appropriate decision execution monitoring and decision preparation dialogues.

9.3.3 Mediator for a data exchange stream

Streams are a prime example where a mediator is useful to (help) make the decisions. The **stream** activity is the mediator and the *owner* of the stream's data chunk buffer, and it structures the coordination of the behaviours of the stream's *producer* and *consumer* with a DOM model like this one:

```

<comp id="RB1"
      class      = "stream dataChunks withBackpressure withActivity"
      properties = "size: 100;">
  <comp id    = "producer-9r342"
        class = "producer">
    <data = "isConfigured: true;">
      <data = "isactive: true;">
        <data = "highWater: false;">
          <properties> counter: 1234; size: 12; </properties>
        </data>
      </data>
    </data>
  </comp>
  <comp id    = consumer-hh4se"
        class = "consumer">
    <data = "isConfigured: true;">
      <data = "isactive: true;">
        <data = "lowWater: false;">
          <properties> counter: 1231; size: 4;</properties>
        </data>
      </data>
    </data>
  </comp>
</comp>
```

These are the effects of that DOM model on the behaviours of *stream*, *producer* and *consumer*:

- the *stream* provides a buffer of the following “classes”:
 - its data chunks are just bytes. And it has 100 of them in its buffer.
 - the buffer has “higher-order” coordination flags to work with **backpressure** and **activity flags**.
- the *stream* mediates between the user peers (producer and consumer), by offering a **hierarchy** of how they have to work with the **status flags**:
 - a peer is responsible for its own **properties** as a user of the stream, namely the counter of the how far it is in the stream, and the number of data chunks it is currently using.
 - **before** changing these properties, the peer should check the backpressure flags, **lowWater** or **highWater** for producer or consumer, respectively. For example, the consumer might decide not to act on the stream unless the **highWater** flag is set.

- **before** looking at the backpressure flag, the peer can look at the *active* flag, because if the other peer is not active on the stream, it makes little sense to become active itself.
- **before** looking at the activity flag, the peer should look at the *isConfigured* flag, because if the stream is not configured for use, it makes little sense to try to use it.

If each of the three peers provides a software implementation that [conforms-to](#) the DOM model above, just changing (and distributing) the model can lead to an automatic reconfiguration of the peers' software. *And* the reconfiguration can take place in a coordinated way, because the mediator is the single source of decision making about the reconfiguration protocol for each peer to follow.

In the broader scope “higher up” in the application architecture, the system developers can give a mediator even more mediation responsibilities, such as in the case where several streams need to be mediated together:

- it can [marshall](#) or [serialise](#) the data chunks on all the streams, so that they can share the same “slow” network connection.
- when such a slow network saturates, the mediator takes the decision about which data chunks from which streams it throws away.

The DOM model above then needs to be “extended upwards” by appropriate component specifications.

9.3.4 Mediator as a DOM-coordinating DOM

(TODO: [25, p.240] already provided the architectural pattern of a *component DOM*, to coordinate the behaviour of a series of (loosely or closely) connected kernel components. Represent the decision making in the mediator as CSS and event reaction *between* DOM models of other activities.)

9.3.5 Mediator for events, flags, protocols, Petri Nets, FSMs

Chapter 2 introduced the complete set of mechanisms needed to coordinate activities: [events](#), [flags](#), [flag arrays](#) (or protocols), [Petri Nets](#), and [Finite State Machines](#). This Section introduces DOM models that represent the hierarchies that exist in the *ownership*, *production* and *consumption* of each of these mechanisms, and of their compositions. The generic DOM model is very simple, and hence very effective to satisfy in implementations and to add to [data sheets](#). Here is an abstract example (or “meta level template”) to illustrate the pattern (eliminating for now the `id` and `properties` fields wherever they are not necessary to support the explanation):

```

<comp id      = "..." 
      class    = "mediator owner coordination"
      data     = "resource {flag: a, flag: b, flag: c}" >
<properties>
  1: resource.c = resource.a AND resource.b
</properties>

<comp id      = "..." 
      class    = "producer">

```

```

<data = "resource.a, resource.b;">
</data>
</comp>

<comp id      = "..."
      class = "consumer">
<data = "{resource.a, resource.c;">
</data>
</comp>

</comp>

```

The mediator **component** is *owner* of the **resource** that is essential in the coordination process. That **resource** is, in the example above, a composition in itself, with different parts: **a**, **b** and **c** in this case. The **component** provides a mediating role between two other **components**, one which has the role of (or, is of the “**class**”) **producer** and the other one has the role of **consumer**. The **data** tags of all components indicate which parts of the mediated **resource** each one needs to have access to, in its specified role. The example does not model explicitly how the **producer** and **consumer** components realise their role; but the **properties** of the **owner** give a glimpse about how to formally represent such action: in this case, it is a simple logical expression on Boolean flags, where the mediator is responsible for guaranteeing the logical constraint between the flags, as specified in its **properties** part of the model.

The specialisations of this generic pattern to the particular coordination mechanisms are as follows:

- **events**:
 - **owner**: the event queues to and from which **producer** and **consumer** get events; logical rules to combine a particular event pattern into another event; translation of naming of events; etc.
 - **producer**: fire identified set of events.
 - **consumer**: handle identified set of events.
- **flags**: this is a simplified version of the **event** case, where *firing* and *handling* are as simple as setting a **flag** to **true** or **false**.
- **protocols**: this *extends* the **flags** case in two ways:
 - grouping a set of **flags** together in an *ordered* array.
 - **properties** that represent the *constraints* on the order in which **flags** are allowed to be set or cleared.
- **Petri Nets**:
 - **owner**: it is the only peer that has access to the **PetriNet** model; it owns the **flags** and **protocols** needed to connect the **PetriNet** model to the mediated peers; it registers those *parts* of **flags** and **protocols** that are accessible to the **producers** and **consumers**; it consumes **flags** from the mediated peers and producing **places** in the **PetriNet**, then *executes* the **Petri Net**, which results in producing **places**, of which some are *transformed* into **flags** for the mediated peers.
 - **producer/consumer**: can *set/read* its part of **flags** and **protocols**; guarantees to satisfy the **protocol** order.
- **Finite State Machines**: fully similar to the Petri Net case, with **events** replacing the roles of **flags** and **protocols**, and with execution of the Finite State Machine model

replacing the execution of the Petri Net model.

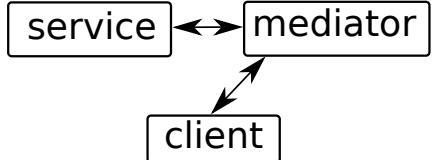


Figure 9.15: Mereo-topological model of the *mediator* pattern, to help system designers introduce *loose coupling* between “*service*” and “*client*” activities. The arrows represent *data exchange constraints* between the interacting activities.

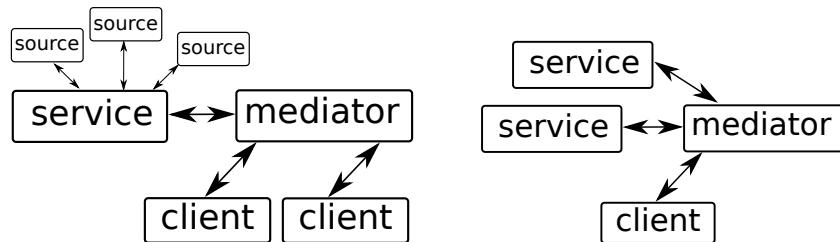


Figure 9.16: Variants of the *mediator* pattern of Fig. 9.15, with multiple sources for the service, and multiple clients or services for the mediator.

9.3.6 Mediator architecture to coordinate interacting activities

Architectural compositions as in Figs 9.15–9.16 show up in many use cases: a *client-server* behavioural composition needs to be realised between a “*service*” activity and a “*client*” activity (or multiple of them).¹² For reasons of runtime (re)configurability, system developers want to avoid *direct* behavioural coupling between both activities, that is, to let them know about each other’s existence, name, software component port address, data access policies, programming language implementation, software version, etc. But, of course, both activities *do have couplings*, in the form of: the *model* of the *information* they exchange, the relations between *data* and *metadata*, and the events through which they *coordinate* and *configure* their behaviours. The important insight is that these latter couplings depend only on the *type* of their activities, and not on the concrete *instances*; while the former couplings do all depend on instance properties.

Many application contexts have components that do not behave clearly as a “*server*” or as a “*client*”, so one often uses the more neutral and bi-directional term **peer**. Anyway, the terminology in itself is not relevant, but the specific dependencies between the specific behaviours in all interacting peers are. The relevance of the separation between, on the one hand, *coordination and configuration* (realised by the *mediator activity*), and, on the other hand, the *computation and communication* (realised by the mediated peer activities themselves), increases with:

- the *number* of interacting peers (Fig. 9.16),
- the *complexity* of the interaction protocols,

¹²The exact meaning of the terminology “*client*” and “*server*” is not widely standardised, to say the least. For example, in the domain of *cloud computing*, the terms “*cloud*”, “*fog*”, “*edge*” and “*client*” are more mainstream; in databases one speaks of “*backends*” and “*frontends*”, and of *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.

- and the *statefulness* of the peers' interfaces.

The **design driver** force behind introducing the **mediator pattern** in information architectures is to concentrate all **knowledge and decision making** about the **behavioural coupling** of several activities **inside the mediator activity**. This activity knows (or rather, it has a *model of*):

- the **information exchange dependencies** between the mediated peers. For example, a mediator:
 - decides which **streams** to connect to which activities.
 - initializes them properly.
 - coordinates access to a shared resource.
 - monitors whether the dependency relations remain satisfied.
 - reacts appropriately when they don't.
 - decides to continue or not with its “**brokerage**” activity.
 - decides to include **glue code** activities where necessary.
- which Finite State Machine **events** the various peers emit and/or react to. So, it can change their behavioural states itself, check whether the mediated peers have events with the same semantic meaning, and include **event forwarding** glue code where necessary.
- which **protocol flags** each mediated peer commits to a **Petri Net** based coordination of each of its algorithms. The mediator is itself the activity that **executes the Petri Net**, and takes care of the asynchronous update of all protocol flags, without the mediated peers being aware that, and how, they are being mediated.
- which **model checking and adaptation** to realise in the **resource configuration** step of the Life Cycle State Machine in a mediated peer. This can, in itself, require a Petri Net coordination between mediator and mediated peer.
- about the **tasks** that the mediated peers have to support, together. The mediator can add extra components to realise **Service Level Agreements**, or to monitor the **quality of service** of the coupled behaviour and to fire the appropriate events to the appropriate peers when the task service falls below the configured threshold.

In summary, the mediator is the clear and unique **owner** of all decisions about how to coordinate, protect *and* optimize access to the peer activities it is mediating. That decision making role makes this pattern more specialised than the simpler **message broker** pattern that is used for **message-oriented middleware**, **object request brokerage**, or **enterprise service buses**.

Exactly *because* a mediator has all this knowledge about the workings of all the activities it mediates, it makes sense, in several use cases, to let the mediator activity be the unique representative towards the rest of the system, of all the activities it mediates. This is the core insight behind **holonic** architectures.

9.3.7 Mediator components in a system's life phases

System developers make different mediator models for the different phases in the life of their system:

1. *design*: the **structure and behaviour** of component interactions can already be represented, implemented and evaluated, except for the dependencies on the available *resources*.
2. *deployment*: the enlarged mediation model adds the extra information needed to deploy the above-mentioned implementations onto the physical system architecture (hardware

and software).

3. *runtime configuration and coordination*: the enlarged mediation model adds knowledge about the properties in the mediation model that can be reconfigured at runtime, allowing the runtime mediation to react to the actual status of coordination protocols that the components are involved in.
4. *runtime introspection*: the status of the mediation models themselves can be *queried* at runtime, by “higher-order” components, that are themselves not part of the above-mentioned mediation model.

In order to adhere to the best practices this document advocates for **composable modelling**, the mediation model in a higher-numbered phase composes the models of the lower-numbered phases with with a higher-order mediation relation. That is, one that uses properties of lower-numbered mediation models. Examples of such model properties are:

- structure of connectivity: a representation of which *ports* are connected to which *blocks* and which *connectors*, with type and cardinality constraints.
- structure of exchanged data: a representation of the data structures that travel through *ports* and *connectors*.
- coordination of connection: a representation of the flags visible via the *ports*, and of how they connect to the Petri Net or Finite State Machine inside a *connector* or *block*.

9.3.8 Policy: extend a mediator into a broker

In its simplest form, the mediator activity has “just” the responsibility to *coordinate* other activities. The interaction primitives needed for such coordination are simple and **lightweight**: **flags**, **flag (arrays)**, **Petri Nets**, and **Finite State Machines**. All coordinated activities must be *registered* with (or, *subscribed* to) the **stream** that exchanges the small data chunks of the just-mentioned coordination primitives. It is only a small increase in activity complexity to let the component behind the stream take more responsibilities; for example, to become a

- **data centric broker**: in addition to doing the bookkeeping of registered peer components, the broker version of a mediator also provides services on top of the raw data: *buffering*, *filtering*, *time stamping*, *delivery guarantees* in **pub-sub** communication patterns, etc.
- **application centric decision maker**: in addition to the brokeraging, the application also “outsources” (some forms of) decision making from the component to the mediator component. Because that latter component knows about (the status of) the interactions between all other components, it is the logical place to make system-wide trade off decisions, such as **prioritization**; **voluntary data loss**, **curation** or **recovery**; **garbage collection**; **dependency injection**; etc.

The more traditional¹³ *broker* pattern works well under the assumptions that (i) the broker can remain the same all the time, and (ii) the communication **middleware** loses no data, ever.¹⁴ The *mediator* pattern offers more opportunities to introduce policies to cope with violations of both assumptions.

¹³Standardized in the 1990s, with **CORBA** (*Common Object Request Broker Architecture*) and **Data Distribution Service (DDS)** as major representatives.

¹⁴Typically, middleware projects provide one or more of the following three policies to guarantee message delivery: **exactly-once**, **at-least-once** or **at-most-once**. Guaranteeing such policies always comes with high costs, under conditions of communication disturbances.

9.3.9 Mechanism: reification of a Connector and its Ports

The [mediator pattern](#) is essential to allow systems to deal with *dependencies* between sub-systems, by hosting all the *decision making* about the dependency coordination inside a dedicated component. One way to look at this pattern is to consider it as the [reification](#) of the Connector of the [Block-Port-Connector](#) meta model, adding the following information to itself and/or to each of its Ports:

- *ID*: because the Connector becomes an activity in itself, it can be logged, monitored, inspected, etc. Of course, that requires that the Connector (and its activity) can be uniquely identified.
- *behaviour*: a Connector can have internal behaviour, while still representing itself via a stateless Port.
- *higher-order dependencies*: when a system becomes larger and/or more complicated, it is inevitable that system developers want to reconfigure a Connector that was static before, in order to take higher-order dependencies into account that did not exist before. For example, to adapt the size of a buffer inside a Connector to a change in the latency and bandwidth constraints available in a newly deployed communication capability.

(TODO: more details and examples.)

Chapter 10

Information architectures for skill-based execution systems

The **information architecture** of a **system** is the **design** layer between an **application**'s *task requirements* on the one hand, and, on the other hand, the *software* and *hardware resources* that the system has available to realise these application requirements. The Chapter on **system architectures** provides the *application-independent* aspects of system architectures; this Chapter *specializes* these patterns to *robotics-specific* applications, providing *information architectures* that **conform to** the **Task-Situation-Resource** system design paradigm. More in particular, such an information architecture add two complementary sources of knowledge to the generic architectural patterns:

- **skill**: the knowledge about **how to execute** the system's tasks, with the provided resources, and in the expected system environment.
- **execution**: the knowledge about (i) **how to distribute** a company's *production* over the hierarchy of its *facilities* (plants, lines, cells, devices,...), (ii) **which skills are at every level** of that hierarchy, and (iii) **how to interconnect** the different skill levels.

An information architecture design has multiple complementary objectives, to go from the above-mentioned *knowledge* to a concrete set of *software components*:

- to embed **synchronous** behaviour inside **activities**.
- to execute activities **asynchronously** inside **components** with **interfaces**.
- to compose these components into a **system**, in such a way that their **interactions** respect, both, the:
 - **ownership** of all parts in the full system design.
 - **design contracts** between the developers and the customers of the **application**.
- to give that composition a **holonic** system architecture, applying the appropriate **architectural patterns**.

While an information architecture represents the **structure and behaviour** of a *concrete* application, it does so in a way that **does not depend** on concrete choices¹ of:

¹To couple the information architecture to all these “platform” choices is the task of the **software architects**, who must make the system *work*, and work *efficiently*.

- **software**, that is, programming languages, development tools, communication middleware or protocols, operating systems, etc.
- **hardware**, that is, sensors, motors, energy sources, CPU architectures, memory layouts, communication networks, etc.

However, an information architecture **does depend** on the application's **tasks**, because it must guarantee the **completeness**, **correctness**, and **semantic explainability**² of how the components in the architecture realise those tasks. The task requirements introduce **artificial constraints** on the realisation; the resources bring **digital and physical constraints**.

A finer granularity in the information architecture's interfaces, and a richer behaviour in its interaction mechanisms, allow—when done right—a higher **composability**, at the cost of more components and more complicated interfaces. In other words, a particular architecture design **trades-off** the system's **reconfigurability** with its **compositionality**. A higher reconfigurability comes with extra costs (at design time as well as at deployment time and at runtime) but (should) result in a system that can **adapt** more to changing application requirements and environmental situations.

10.1 Methodology to design information architectures

This Section explains this document's approach to create an information architecture for an application.

10.1.1 Analysis and synthesis — From requirements to specifications

All design and development processes iterate over *analysis* and *synthesis* steps. In this document's scope of highly self-explainable and -configurable systems, the synthesis step gets an extra “compositionality” focus:

1. **Analysis:** find out *what is required*.
 - identify the application's **situations**: the **semantic areas** and the **actions** that the application requires to be available as **capabilities** of the robots in that situation.
 - decide which **task** the application will need in each relevant *situation*. That is, which *combinations of possible actions* will *effectively* be needed.
 - identify which **dependencies** this selection of *tasks* introduces for the perception, information processing, and/or motion control *functionalities* to be present in the robots.
 - identify how these dependencies influence the decisions about which **task executions** can/must run in sequence, and which ones allow/require concurrent execution.
2. **Synthesis**—Short-term horizon: design *how to realise* the requirements.
 - create the **state models**,³ to represent the application's behaviour in each task, for each situation.
 - create **functions** that turn task and state models into behaviour.
 - decide which **activities** will realise which behaviour.

²The **buzzword** of this decade is **explainable AI**. Like with all buzzwords, the motivation and expectations behind it are absolutely relevant and real. But most of its realisations do not live up to these expectations. Most often because they lack a **refutable** definition of their interpretation of *explainability*.

³Often also called “world models” in this document

- decide which **data exchange streams** are needed, and which **lock-free mechanisms** applies best to each of them.
 - create the flags, Finite State Machines and Petri Nets needed **to coordinate** the **asynchronous** execution of activities.
 - design the **event loop** inside each activity, to execute the behaviour including the streams (`communicate()`) and the coordination state (`coordinate()`).
 - decide which activities to put together in a **component**.
 - decide which **streams** to attach to each **component**.
3. **Synthesis**—Long-term horizon, maximizing **composability and compositionality**: assess to what extent the *design remains the same* under *changing requirements*.
- make components **discoverable**, with **queryable interfaces** exchanging **data sheet** information.
 - add *factories* to realise runtime changes in the information architecture for: (i) *hardware* (sensors and kinematic chains), (ii) *software* (perception, control, monitoring, decision making), (iii) *tasks*, and (iv) *situations*.

10.1.2 Policy: ownership, loose coupling, semantics

Below are generically relevant **cross-cutting responsibilities** to all above-mentioned **design decisions**, that have to land somewhere in activities in the information architecture:

- **ownership**: the decision about which activity owns the information in the system, not in the least the information to access the resources in the hardware (often via the operating system). Complemented by the decision under which conditions ownership can (or should) be transferred to other activities; often only for limited periods in time.
- **loose coupling** of data, solvers and activities: let the activities *observe* the behaviour of each other during their interactions, and let them react to it instead of hard-coding that interaction upfront.
- **semantic meaning** of the information: it is the mediator activity that links the architectural aprts to documentation and offline/online tooling, to guarantee that information is interpreted in an unambiguously correct way, by human developers as well as by the mediated activities.

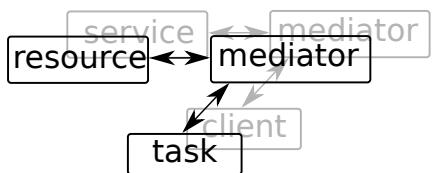


Figure 10.1: The special case of the **mediator pattern** applied to the interaction between *task* and *resource* activities.

10.2 Situation-aware skill architecture

This Section pertains to the architecture of a system whose core is a *skill activity*, designed along the lines of the **Task-Situation-Resource** paradigm (Fig 6.18). That skill is a **mediator** activity between resource activities of all kinds, Fig. 10.1, that must decide about **how**:

- to assess the **impact** of the actual physical environment on the quality and performance of the ongoing task executions.
- to further execute these **tasks** with a **quality control** that **satisfies** the application requirements.

- to use **resources** with an **efficiency** that is **affordable**.
- to be **robust** against those **disturbances** that are **relevant**.
- to exploit **opportunities** offered by the actual **situation**.

More concretely, all property graph “arrows” in Fig. 6.18 must be turned into decisions on how to exchange which data between two or more activities, synchronously and/or asynchronously, and those decisions must be embedded in “higher-order” activities.

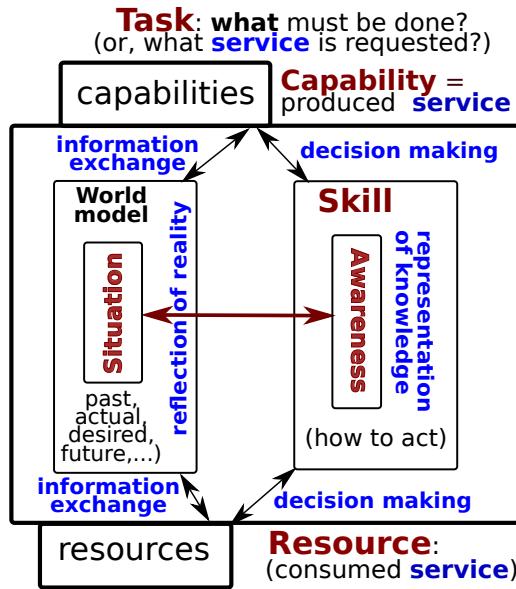


Figure 10.2: Copy of Fig. 6.18, representing the Task-Situation-Resource meta model.

10.2.1 Task execution via constrained optimization solving

This document’s skill primitive represents a **plan** as a **hybrid constrained-optimization problem** with objective functions and constraints collected from the *task* requirements, the robot motion and sensing *resource* capabilities, and the *situation* around the robot in which to realise the task. This approach yields a **declarative** specification, that is, it is a model (generated off-line or online) that expresses the logic of *what* the task wants to see realised, without describing the **control flow** of *how* that must be done.

This implies that the skill architecture *requires* an activity running a **solver algorithm**, to take in the declarative specification and *to generate* an **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 specification. The above holds both for *control-based* approaches [3, 92] (online, reactive, but maybe suffering from local minima problems) and for *plan-based* solvers [86] (typically, but not necessarily, offline, less reactive, but exploring a bigger search space).

The term “skill” was used in the paragraphs above as a container term for each of its parts: task, plan, control, monitoring, perception, and resource capabilities. 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 skill. 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 landmarks, in a particular order and with a frequency of revisiting the relevant landmarks, that depends on, both, the tolerances required for that localisation, and the risk involved in making possibly erroneous decisions.

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

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

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

Policy: dynamic programming

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

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

This is a broad family of “declaratively specified” algorithms (or *solvers*, Sec. 6.10), and hence they come close to exploiting all features of the presented algorithmic meta model.

Policy: static spanning tree pyramid

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

Policy: feasible and optimal solutions

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

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

Satisfying versus **optimizing** solvers. The former form of solver uses the iterations towards the most optimal solution, until the current intermediate solution is already adequate (“good enough”). That is, within a specified, set of the constraints. The latter form of solver only returns a solution when this solution is the optimal one. Of course, both solver types should also stop as soon as they find out that the solution can not be computed, for some reason.

Policy: sweep scheduling

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

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

Policy: tolerances

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

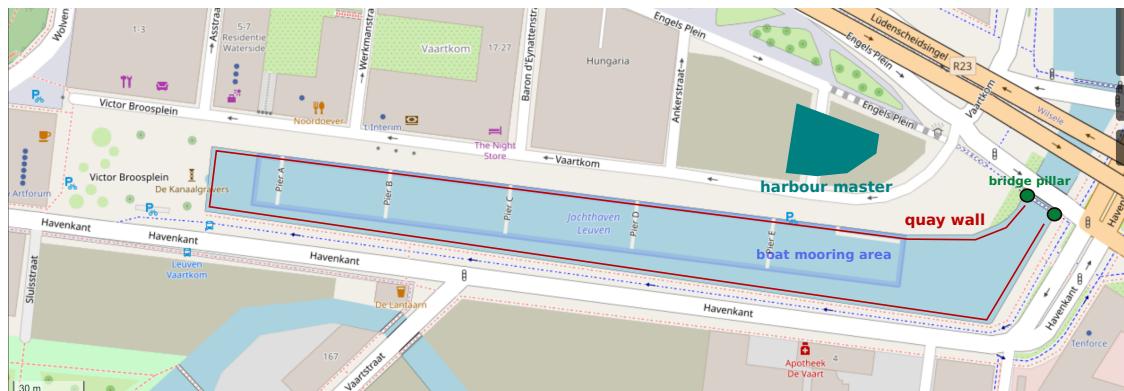


Figure 10.3: Base map of the [city harbour of Leuven](#), with a perception layer of four semantic tags: the bridge pillars, the quay walls, the harbour master office, and the mooring area for boats.

10.2.2 Situational aware architecture in the “edge”

This Section introduces a [reference information architecture](#) for the carto-ceptive control of a varying number of peer robots in an edge composition, where *situational awareness* is a shared common task. The presented reference architecture is rather “empty”, in that the **largest part of the concrete design** consists of **application-specific** data structures and

coordination protocols. To make the architecture somewhat tangible, the “edge” is considered to consist of a city harbour as in Fig. 10.3, with a couple of vessels as robotic peers, and the harbour master as the local navigation authority peer. The latter is also in charge of the opening and closing of the bridge over the harbour, which will be a “task” that the vessels can ask it to execute at their behalf. Although explained in the application context of “autonomous shipping”, the design of the reference architecture is very similar to other robotic contexts, such as logistic AGVs in a warehouse, or multiple robot arms sharing an assembly cell.

Base map: this is as depicted in Fig. . The list of semantic tag layers that are available to the vessels is to be standardized by the inland waterways authorities. Not just the list, but also the information models that can be accessed through the tags. All vessels are obliged to have a particular version of the standardized map on board, in their own extero-ceptive navigation pilot.

Communication: the vessels are obliged to use the communication channels and messaging protocols prescribed by the waterways authorities. This involves several steps, of discovery, session initiation, authentication, heartbeating, decision logging, etc. Each of these steps could require the use of different channels and protocols. Two major responsibilities for the communication are (i) to support each peer in the edge network to update its dynamic map with information from the other peers, and (ii) to enable the coordination of the shared decision making needed to let each vessel execute its intended journey, while sharing the area with other vessels, in an explainable and legally compliant way.

Situational aware perception: each vessel extracts from its own internal extero-ceptive control activity the subset of landmarks, vessels and “obstacles” that it recognizes; Fig. 10.4 sketches what that could mean in the city harbour case. It communicates that information in its edge network, as its **dynamic annotated map**. It also obtains similar dynamic annotated maps from the other peers. So, it can improve its own assessment of its environment, and assess what the other vessels in the neighbourhood are aware of themselves. The quality of each vessel’s dynamic map, and hence the self-localisation of that vessel on the base map, can be scored by the number of perceivable tags on the base map that have actually been recognized.

Situational aware task execution: on top of the situation awareness provided by the above-mentioned dynamic map, each vessel:

- decides about its own navigation task.
- makes a “trajectory” of planned maneuvers in the near future available on the dynamic map.
- adds its own *control limit zones* around these planned maneuvers, that is, two zones around the hull of the vessel in which it (i) can still avoid collision when another vessel or obstacle would enter that zone, and (ii) can only react with an emergency stop, without guarantee for collision avoidance.

This input allows for a **preview control** approach: the task decisions of each vessel can be taken with information about the *intended future* maneuvers of other vessels, Fig. 10.5. Of course, this concept requires the standardized coordination of all vessels’ individual tasks. For example, via **distributed consensus** coordination protocols, such as:

- **delegate MAS** [69]: peers interact individually with a selected number of other peers in the edge network, and take full responsibility for the decision making.
- **promises**: robots decide to cooperate voluntarily in their mutual action coordination by

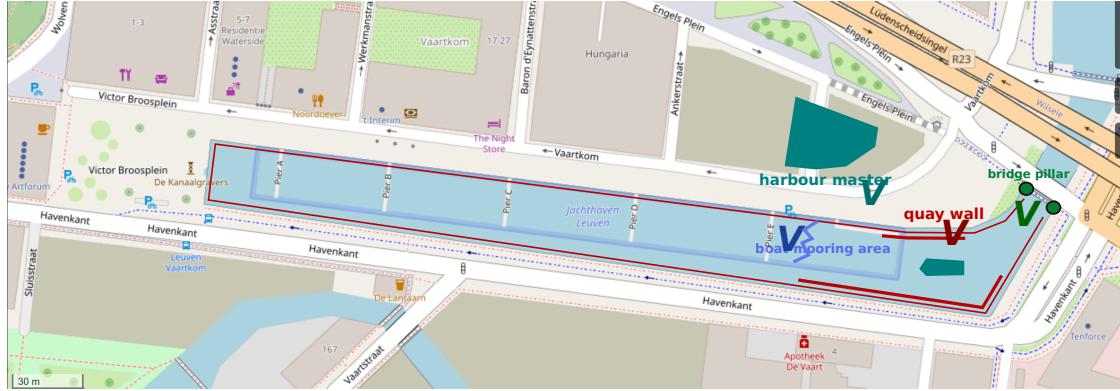


Figure 10.4: A sketch of what the dynamic annotated map for a vessel looks like. The vessel has recognized four of the tagged landmarks on the base map; this recognition is represented by the big V. The harbour master office is sensed via its *4G communication*. The two bridge pillars are both recognized in its *Lidar data*. The recognized part of the quay wall (also from *Lidar data*) is indicated in bold red lines. *Some* occupancy of the boat mooring area has been recognized, but no individual boats or objects could be identified in the *Lidar data*.

publishing their intentions (“promises”) to each other.

- *mediator*: the peers communicate their intentions to a (unique and agreed upon) peer (for example, the local harbour master), that has the authority to make decisions in name of the involved peers.

Both approaches can use similar methods, such as [auctions](#).

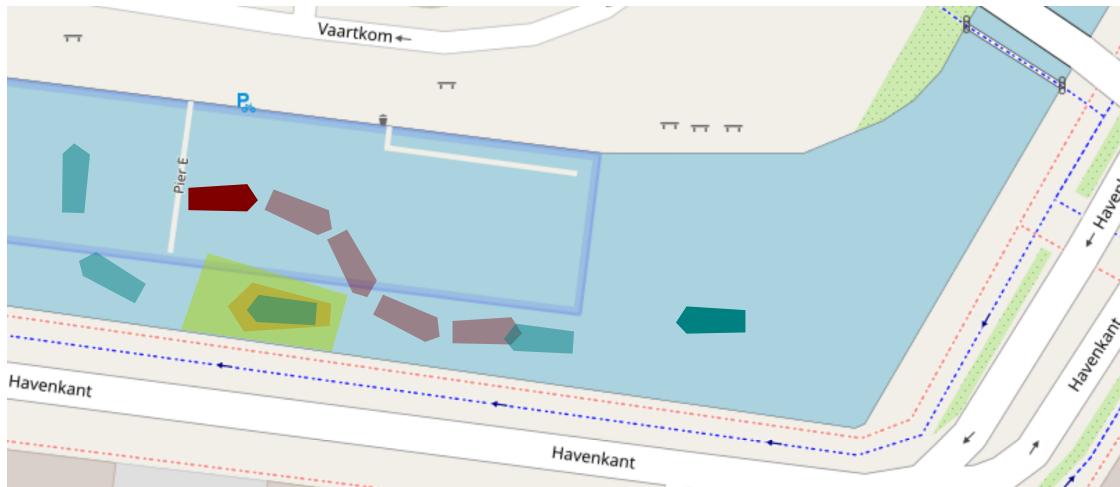


Figure 10.5: The red and the green vessel communicate their intended maneuvers for the four next periods of 30 seconds. This allows all peers in the edge network to *preview* the change in the actual situation, and to engage in arbitration protocols when needed. One of the preview locations of the green vessel is annotated with a green and an orange zone, indicating the zones in which the vessel’s own navigation control can, respectively, can not, avoid collision when another vessel or obstacle enters that zone.

In summary, the major message of this Section is that:

- the reference architecture of a situational aware “edge” network of peers is very **structured and conceptually simple**.
- it can only work reliably, predictably and with clear liability, if **dozens** of application-specific protocols and data structures are **standardized and compulsory** for all peers.

10.2.3 Sense-Plan-Act, Three-Layered, Behaviour, Subsumption

(TODO: Three-Layer architecture [5]; **TLA**, **three-tier**; decisional/deliberation = planning (symbolic control), execution = decision (discrete control), reaction/functional = (feedback) control.) Identification of too strong couplings.)

10.2.4 Task-Situation-Resource versus Sense-Plan-Act

This document advocates the *Task-Situation-Resource* paradigm, as a replacement of the mainstream **Sense-Plan-Act** paradigm. The former let systems *start* any Task execution with a *plan* of *guarded motions*, and with an explicit *world model* (“**map**”) that contains **geometrical** as well as **symbolic** tags. Both are pieces of *knowledge* one can expect to be available, and that one can expect “agents” to interpret correctly in the context of the full map. The system then uses its sensors that provide data (i) to guard how well the plan can be realised, (ii) to suggest updates to the world model, (iii) to associate the symbolic tags with real-world geometric entities, (iv) to adapt the current plan, and (v) to decide to look for a better plan in the Task’s plan repository.

The Sense-Plan-Act paradigm, on the contrary, expects systems *to generate* their plans themselves; most often in the form of nothing more than a *trajectory*, because trajectory control is the only control action that can be realised in a sensor data-only context. Indeed, it is only when the *context* of the motion is available, that the context-dependent knowledge of a Skill can be introduced. In other words, the shortcoming of SPA is *not* in that the robot can not *react* to changes in the environment during motion execution (this is as easy as starting another SPA cycle, whenever deemed appropriate), but that the choices made in determining the *act* model can not influence the *sense* or *plan* choices anymore. This reduces the opportunities for task-dependent configuration of sensing and planning.

10.3 Execution system architectures

(TODO:)

10.3.1 Manipulation execution architecture

(TODO:)

10.3.2 Manufacturing execution architecture

Section 6.4 introduced examples of relevant entities and relations for the *task modelling* of manufacturing systems. The *architectures* of these systems—such as the three **levels** defined in **ISA-95** standard—most often consist of a version of the “**Edge**” architectures, as depicted in Fig. 12.2. The algorithms in the activities are mostly involved in bookkeeping, data base and scheduling, and the communication of orders (of parts and of their manufacturing) takes place

via an “cloud” network internal to a company, which possibly has different manufacturing plants, each with possibly different production lines.

The *execution* of the orders is done in an “edge” network of devices with architectures as in the previous Section. The functionalities of such order execution systems are as follows:

- they **receive messages** with the information about **orders** and the corresponding **production** information).
- they also **receive physical goods** via conveyor belts or mobile robots, which they have to handle via robotic devices.
- they **send out status information** about how well the production of the orders advances, together with statistical data for long-term analysis of the plant’s efficiency.

The **appropriate data exchange mechanism** for all this messaging is the **Submission-Completion** meta model, because, typically, (i) one “order” message gives rise to lots of “status reply” messages, and (ii) one does not want to throw away any of these messages.

10.4 Best practices in activity–stream architectures

Activities interact, because they need to influence each other’s behaviour, like letting one activity do some work because another activity depends on the result of that work. One type of interactions is a *data flow* from one activity to another one, via a one-directional *send-and-forget stream*. The other type of interaction is a *bi-directional dialogue* between a “master” activity and a “server” activity, where the former uses the stream to send over information about “work” that it wants to be done by the latter.

This Section introduces some best practice architectural approaches about how to integrate these two types of interaction behaviour between activities.

10.4.1 Interact via requests instead of via commands

In many existing architectures, activities behave as if they are *master* of any interaction they are involved in, that is, they typically and implicitly interpret an interaction as a *uni-lateral command*: by providing the “server activity” with the information about the command, the “master activity” just expects that that command will be realised, without further ado. However, systems become more composable if every interaction between activities is treated as a *friendly request*, and as a part of a *bi-directional dialogue* between the “master” and “server” activities. Indeed, this approach allows peer activities **to adapt** to their mutual expectation and performance, at runtime. all the time, The dialogue serves many purposes, such as: sharing the decision making about what is a “good enough” trade-off between both activities’ objectives; the “server” informing the “master” about the progress that it makes in realising the “work”; the “master” informing the “server” about a change in execution performance expectations, including the preemption or abortion of the work request; etc.

10.4.2 One Life Cycle State machine per activity

Each activity has its own “life”, independent of any other activity. Because that “life” must be a **singleton**, there is one and only one **Life Cycle State Machine** in each activity to coordinate the different phases in the activity’s life. Only via a full LCSM, an activity can reconfigure, at runtime, the resources it owns. (Of course, that is only true *if* that resource has the functionality of runtime reconfiguration.)

Activities are not just essential as information architectural entities to represent “life” of their own, but also to give “life” to the system: they interact with each other to realise a system’s desired behaviour. Two necessary (but not sufficient) conditions to make sure that such interactions produce **predictable behaviour** are that:

- each interacting activity is itself in the internal state that is designed to support the interaction.
- it can communicate explicitly about the interaction with the other activities it interacts with.

Here is an example: an [EtherCat](#) communication can not be used in real-time between the “master” activity and all of the “clients” in the EtherCat slaves, before the appropriate [handshake](#) is performed. That means that the master as well as all slaves go to the same state in the standardized [EtherCat life cycle state machine](#).

10.4.3 Every entity and relation has one explicitly identified owner activity

It is that owner, and that owner only, that decides about the operations, and possibly the related policies, that are allowed to act on the data of the entity or the relation. Ambiguous ownership leads to the confusion about which activities should or can configure resources, at the wrong moment.

(TODO: examples: actuator or sensor; task; state of a solver; resource allocation; etc.)

10.4.4 Explicit causality in data access policy

Activities must interact with each other to realise tasks. And each task brings (models of) causal dependencies between task functions operating on task data: a particular function should only operate on a particular abstract data type when certain conditions are met. Such causal relationships must be modelled explicitly, by relations representing so-called *data access policy* constraints.

For example, the *real-time* thread in a dual-thread activity is the one who does the *writing* from the source of the data to any shared data structure, because it is the creator, and hence the owner, of that data. When the *non-real-time* thread would do the copying, the data transfer could be interrupted by preemption of that thread.

(TODO: more examples.)

10.4.5 Every task 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.

10.5 Minimal latency data exchange — Double and triple buffers

This Section introduces the use case of a *single producer–single consumer stream* in which the consumer wants only the **latest available data chunk** from the producer, and can discard all older versions without compromising its usefulness. Major examples of this use case of the producer *overrunning* the consumer are:

- images from a camera: an image can be large, and its processing can take longer than the image acquisition time that the camera can achieve. So, processing all images would be more *time* and/or *memory* consuming than allowed in the application. If the images are to be used in an **extero-ceptive control** loop, it makes sense to only spend processing time on the latest available image.
- repetitive database queries: an activity can subscribe to updates from a database, but is only interested in processing the last update it received.
- data in a real-time control loop: almost by definition of “real-time control”, old data is not useful anymore.

The **ring buffer** mechanism is not appropriate to support this “faster producer than consumer” use case. Indeed, making the buffer larger does not avoid the situation that it *will* be full, sooner or later, in case the producer produces data faster than the consumer, on average.

This Section introduces the **double buffer** and **triple-buffer** (or *three-buffer*) mechanisms that *can* give that guarantee. Only the triple buffer is explained in detail, because:

- the double buffer is a simplified version of the triple buffer, using only two versions of the data chunk in memory, instead of three.
- the double buffer can not guarantee the *wait freedom*. This case occurs when the consumer has finished reading the one-by-last available buffer, and wants to start with the latest one, but that one is still being written by the producer.

The **structural** part of the triple buffer is simple (Fig. 10.6): it has **three slots**, each of them can be accessed by producer and consumer. These slots have **no order** that has any semantic meaning in the context of the data exchange stream. (While in the case of a ring buffer, there *is* such meaning, namely the monotonically increasing ID of the data chunks generated by the producer.) The **behavioural** part is as follows:

- each of the three slots must, at all times, be in one, and only one, of the following **states**: **free**, **written**, or **reading**.
- the *producer* searches for a **free** slot. *After* it has finished its operations on the data chunk in the slot, it **atomically** switches the state of the slot from **free** to **written**. *At*

fast producer	fast consumer	slow producer	too fast producer
<p>new data chunk available from producer</p> <p>20</p>	<p>16</p>		<p>new data chunk available from producer</p> <p>22</p>

Figure 10.6: *Triple buffer*. Except for the case in which all slots are **free**, only the following three cases are possible.

Fast producer: the consumer is busy **reading** one slot (with data chunk “16”), the producer has already **written** a second slot (with data chunk “19”), *and* is now about to start filling the **free** third slot (with its next data chunk “20”).

Fast consumer: the consumer is **reading**, but the producer has not yet been able to produce a new data chunk.

Slow producer: the consumer is ready to start **reading** a slot, and there is one **written** slot available immediately.

Too fast producer: if the consumer misses more data chunks than desired, the triple buffer protocol *could* be extended with a **backpressure** flag **freezeProducer**.

the same time, it switches the state of the slot that it had **written** previously to **free** again.

- the *consumer* searches for the **written** slot, by cycling through the three available slots. It is possible that all three slots are **free**, indicating that the producer has no new data chunk available. *If* the consumer does not find a **written** slot, it decides to do “something else”, and come back to the buffer “later”. *If* the consumer does find a **written** slot, it **atomically** switches the state of the slot from **written** to **reading**. Or rather, when cycling over the slots, it tries this atomic switch on each of them, so “finding a slot” is the same as “switching the label of that slot to **reading**”. *After* the consumer has finished its reading operations, it atomically switches the state of the slot to **free**.

When initialised and executed correctly, the following **invariants** hold for the triple buffer at all times:

- all slots are initialised as **free**.
- there is always *at most one* slot labeled **written**.
- there is always *at most one* slot labeled **reading**.
- a producer never fills a slot with a **written** label.
- a consumer never reads a slot with a **free** label.

The invariant of “*at most one written*” can be difficult to implement in practice, because *two* variables have to be changed atomically: the label of the **written** slot and the label of the **free** slot. So, only under ideal atomicity conditions, the protocol introduced above can **guarantee** that:

- the producer must **never wait** to write a slot. That is, the worst case is that it has to visit all three slots to find one that is **free**, because (i) the other two are either **written**

or **reading**, and (ii) the consumer will never touch the **free** slot.

- the consumer must **never wait** to read a **written** slot *when it finds one*.
- the consumer always reads the **latest** data chunk that the producer has written.

So, the protocol is guaranteed to be **wait-free** for the producer, but **not** for the consumer. It is indeed not certain that the consumer can read the latest available slot in a **deterministic finite number of operations**, because the following two situations can occur:

- the consumer systematically *outruns* the producer: it then has to come back several times before the producer has made a new data chunk available.
- the producer systematically *outruns* the consumer: then, the following **race condition** can occur:
 - the consumer has atomically checked a first slot, and that turns out to be **free**.
 - the consumer then tries the second slot, and that turns out to be **free**, too.
 - in the meantime, the producer has filled the first slot that the consumer has visited, switching its status to **written**.
 - the consumer then tries the third slot, and that turns out to be **free**, too.
 - in the meantime, the producer has filled the second slot that the consumer has visited, switching its status to **written** and the status of the first slot to **free** again.
 - so, when the consumer tries the first slot again, that has become **free** again.
 - etc.

The consumer can not distinguish between both situations, because both result in the same observations at the consumer's side of only encountering **free** slots. To guarantee that the consumer can (i) observe whether the producer has added new data, and (ii) make race-free *deterministic progress* a **higher-order** abstract data type is added to the buffer (Fig. 10.6):

- a **monotonically increasing counter**, that is *written* by the producer and *read* by the consumer. The counter is initialized to any positive integer number.
- a **status flag freezeProducer** that is *written* by the consumer and *read* by the producer. The flag is initialized to **false**.

The behavioural changes in the extended protocol are as follows:

- before writing on the buffer, the producer checks the **freezeProducer** flag. When it is set, the producer fills a **free** slot with new data, increments the **counter**, and then refrains from trying to write any more slots until the flag is cleared again by the consumer.
- the producer's activity on the buffer since the last time the consumer looked at the buffer is now observable to the latter, because it can compare the **counter** value it stored that last time with the **counter** value it reads now.
- the consumer **can** decide **to set** the **freezeProducer** flag whenever it observes that it has missed “too many” producer writes. (That threshold is a *configurable policy* of its behaviour.)
- the consumer **must** decide **to clear** that flag again, after it has succeeded in **reading** a **written** slot or earlier, and setting its state to **free**. If it doesn't clear the flag, a deadlock will occur in the data exchange between producer and consumer, because both will wait indefinitely for each other to act.

This extended protocol has the following impact on the performance of the buffer:

- the guarantee disappears of having the producer make its *most recent data* available to the consumer.
- the producer keeps its guaranteed **wait-free** property, at the cost of the extra operations

to check the `freezeProducer` flag, and to increment the `counter`. Both operations can take place atomically, so the delays are deterministic and minimal.

- the consumer wins in *determinism*, because it decides for itself how long it wants to wait to get hold of a newly `written` slot, before setting the `freezeProducer` flag. In case the producer is indeed systematically producing faster than the consumer can consume, the latter will only have to retry once or twice to find a newly `written` slot.

The cost at its side are the extra operations to (un)set the `freezeProducer` flag, to read the `counter` and to compare it to the value it saved from its last successful reading.

(TODO: add the Petri Net that models the two protocols.)

10.6 Task queue and worker pool — Configuration and Coordination for task execution

The systems that are in the scope of this document must be ready to execute multiple tasks (“jobs”, “orders”, “commands”, . . .) at the same time. These tasks may, at first sight, have no inter-dependencies at all, but, on closer reflection, such inter-dependencies are inevitable if only because the system has only a finite number of resources at its disposal to *execute* all tasks. In other words, where there are *tasks* to execute, one needs *workers* to do so. Hence, an important responsibility of a system is **to assign the right workers to the right tasks at the right time**.

Imperatively programmed task execution behaviour is most often realised with an architecture built upon **application programming interfaces** (“*APIs*”): the imperative approach (“**hard coding**”) towards the solution of a problem invites developers **to fix the control flow** of the solution already **at design time**. In other words, the *system designer* already decides which “worker” will execute which “task”, even before the system is brought alive, and before concrete tasks are coming along.

In other words, every different case of a “request” for a solution has been identified, and is met by one particular “command” encoding to the solver of the problem, and that command is then solved by a dedicated algorithm. Hence, such algorithms tend to be monolithically designed, that is, not taking into account the **5C separation of concerns**. In addition, an API-centric system design invites “clients” to tell their “solution provider” *how* to solve their problem, instead of providing information about *what* their problem is.

Declaratively programmed systems follow the latter approach: they ask *runtime solution providers* (“*solvers*”) **to compute** the control flow **reactively**, that is, **at runtime** and based on the state of a data structures that encode the (declarative represented) dependencies between the problem representation (that is, the “task” or “order”). (That data structure is most often a **directed acyclic graph**.)

The architectural pattern that supports such a reactive behaviour can already be introduced in the **information architecture**. The solution is inspired by how real-world organisations are organised: the solution pattern introduces **task queues** and **workers**, and whenever a worker is available it selects a task from one of the queues, and starts executing it. This solution is *declarative*: workers describe the skills they offer, and tasks describe the skills they need, and “something” matches them together. Obviously, that match making is a form of **mediation**, introducing a “third party” that helps in making the decision when and which worker to select from a queue of workers that the application has available; such a queue is

commonly called a *pool*.

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

- which tasks are waiting on the queue, to be processed.
- which tasks have been assigned to which workers, and which intermediate task execution results are already produced.
- which tasks have already been processed, but for which the results have not yet been consumed.
- which tasks have already been processed, and for which all results have already been consumed.
- which workers are available in the pool, offering processing capacity.
- which workers are being assigned to which tasks, and which resources are they consuming.
- what are the lists of all tasks that each worker has produced.

The **submission-completion** interaction pattern supports the structured bookkeeping of these interwoven information exchanges about requested, consumed, and produced tasks and resources.

The obvious ways in which the task queue and worker pool pattern allows performance trade-offs to be made in system architectures are:

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

Examples of declaratively designed behaviour abound in all engineered systems of somewhat realistic complexity:

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

10.7 Coordination and Configuration for runtime creation activities — Factories

This document has a focus on long-living systems that ideally should never be [rebooted](#), and that ambition implies that activities must be manageable at runtime. More in particular, it must be possible to create and configure, at runtime, all *data structures* that are needed to store the [state](#) of activity, as well as [to \(un\)load executable modules](#).

To bring system to a “frozen” state for hot-swappable reconfigurations of everything, Life Cycle State Machine support must be available everywhere in the system. This requires a “7Cs” adaptation of the [factory](#) pattern. The pattern adds an extra [factory](#) activity to the system architecture, that is *dedicated* to this purpose, and to only that purpose. That [factory](#) activity creates “software objects” (data structures and functionalities) of a particular type, and coordinates their [life cycle](#). Of course, a system can have more than one [factory](#) activities, as long as the behaviours of the artefacts they create are [composable](#).

Implementing a software architecture with one or more [factory](#) activities requires a significant investment from the system developers. In the ideal case, software that is *designed* to be useable in a [factory](#) has the following *design invariants*:

- *all* of its *status variables* are stored in data structures, (and not in compile-time [#defines](#)) and are *modelled*.
- *all* of its *behaviour* (activities, configurations, protocols, interactions, etc.) is *modelled*.
- the [symbolic indirections](#) between the *models* and the *binary representations* of the above-mentioned data and behaviour is present, as a separate data structure (with a symbolic meta model of itself).
- hence all parameters and functions that are present in a system can be [discovered](#) and [adapted at runtime](#).
- all components conform to the [5C meta model](#), so they have the required [Life Cycle states](#) of [configuring resources](#) and [configuring capabilities](#).

Hence, one can let a running system change its behaviour at runtime, by (un)loading and (re) configuring- the executable components. [Instrumentation](#) is a major example of the functionalities that become available in this way:

- one can add and remove *scopes* to a running system, to log a configurable selection of its runtime state variables.
- one can add and remove *monitor functions* to a running system, to generate events when particular conditions are reached in the runtime state variables.

10.7.1 Discovery, connection, session, configuration, flow control

This Section explains another behavioural “cascade” structure, via which stand-alone systems can, temporarily, interconnect with each other and form a bigger system-of systems. The [containment hierarchy](#) of such behavioural cascade is the following:

- [discovery](#): the automatic detection of software components (“services”, “systems”,...) on a computer network.
- [connection](#): [signaling](#), [Interactive Connectivity Establishment](#) (ICE), connection ID, [heartbeat](#), [piggy backing](#).
- [session](#): [session description](#), [session initiation](#),
- [configuration](#): [perfect negotiation](#),
- [flow control](#) : [flow control](#), marshalling,

Chapter 11

Software architectures for components

A **software architecture** is the **system design** layer between (i) the **application-centred information architecture** and (ii) the **physical resource**-centred hardware architecture; often, the latter is already encapsulated in the **hardware abstraction layer** of the **operating system**. In other words, software architects create the **software components** to implement an application's *information architecture*. This Chapter provides them with the following mechanisms, that the *software level of abstraction* adds to this document's architectural **trilogy** of **system-information-software architecture**:

- the **multi-threaded process**, for deterministic ownership of data and resources, reconfiguration of software deployments, and efficient, **realtime** performance.
- **circular buffers** for efficient and composable data exchange.
- **symbolic indirection** for runtime reconfiguration, inspection, creation, deletion and discovery.
- **memory barriers** for the “asynchronicity coordination” of the application’s algorithms with the local CPU cores and RAM memory.

The transformation from an application-centric design into a software-centric implementation is very structured, because both the information and software architectures share the **same handful of building blocks**: **activities**, **algorithms**, **stream buffers**, **finite state machines**, **Petri Nets**, **(protocol) flags**, and **event loops**. The difference is that an information architecture *models what* the behaviour of an application must be, while a software architecture *implements how* that model is realised in executable code.

This Chapter focuses on the **component level** aspects of software; in other words, **computer science** and **software engineering in the small** [38, 39]. So, the decisions to be made in the design are:

- which **data structures** to implement, and in which granularity.
- which **functions** to implement to process these data structures.
- how to compose data structures and functions into **algorithms**, in a way that can be realised in **any** programming language.
- which **programming languages** fit best to which deployment context.

- which **algorithms** to deploy in which **activities**.
- which **solvers** to use to **schedule** these algorithms for execution in their activity.
- which **activities** to deploy in which **event loops**, to trigger their execution activation.
- which event loops to deploy in which **threads**.¹
- which threads to schedule on which **CPU cores**.
- which **environment variables** to get from the **shell**, at start-up time.
- which **channel** implementations (e.g., **streams**) to deploy between which activities, choosing the “best” policies for **buffering** in shared, **contiguous memory**.
- how to realise the **ownership** constraints of information and hardware architectures.
- which libraries to choose, in which programming languages.
- which **coordination** to introduce between (algorithms deployed in) activities, for (i) their **access** to shared data, (ii) their **synchronization** of their execution schedules, and (iii) their deployment on **multicore CPUs** and on **caches**.
- which quality to aim for in data **freshness** and **consistency**.

In its examples, this Chapter uses the following established technologies: the **C** programming language and **standard library**; the **Linux/UNIX** operating system; **POSIX threads**; and the **Bash shell**.

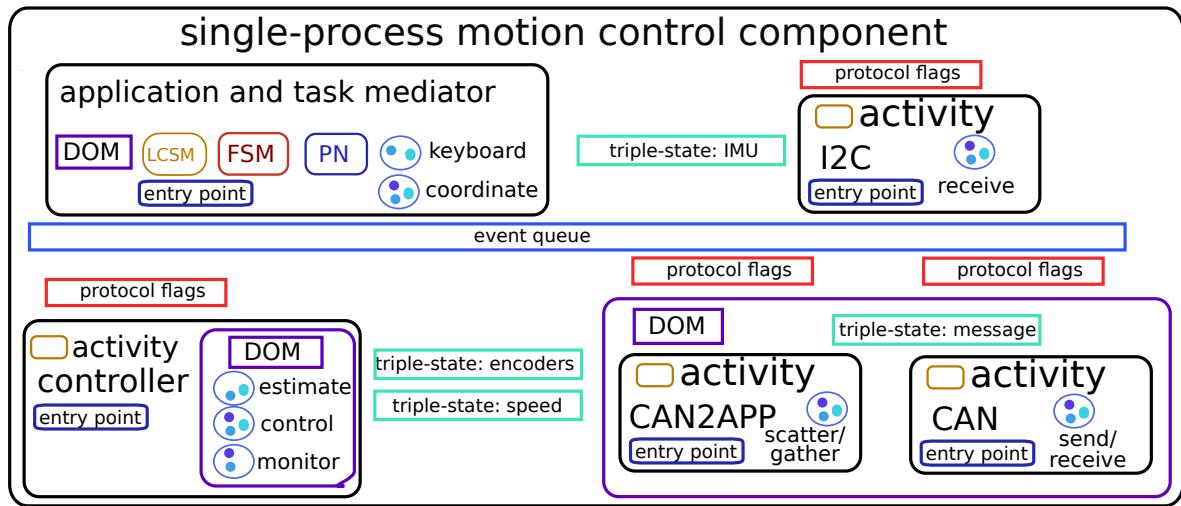


Figure 11.1: Information architecture for a *single-process* motion control application, that uses I2C and CAN busses to access encoders, IMU and motor drives.

11.1 Single-process 5Cs component: hierarchy of threads, activities and algorithms

This Section documents the **pattern** with which system developers can turn an information architecture into a *software architecture*. The mechanisms for such a software architecture are:

¹Or to run them on the **bare metal** of the CPU. The term “thread” is used as a *pars pro toto* for every software mechanism that can **host the execution of code**.

- one **single process**, with a **hierarchical** structure of a component's **behaviour**.
- **shared memory**, to implement the **heterarchical** structure of the **interactions** between algorithms in a component. That is, about which data can be accessed by which algorithm, and when.

These mechanisms are the foundations with which *to transform* the (application-centric) information architecture of, say, Fig. 11.1, into (a variant of) the (software-centric) **multi-threaded, single process** software architecture template of Fig. 11.2. The purpose of this Section is to explain that template in full detail.²

The **single process** **creates** multiple **threads**. Each thread **connects** one part of the application behaviour to the operating system, to let the latter take care of the **execution** of the thread's software. More in particular, each thread **schedules** multiple **activities**. Activities **organize** the application behaviour in **components**. Each activity **schedules** multiple **algorithms**. Algorithms **implement** the application behaviour with **functions** and **data structures**. The *application* software takes care only of the *scheduling*, that is, it decides in which *order* functions are being “called”, including those for inter-activity data exchange. It is the *operating system* software that decides about *when* the actual *execution* of these functions on a core of the CPU takes place.

The explanations in this Section are structured along the lines of the **5C component model**. That model is used here as **pattern** (and not just a **best practice**), because its design trade-offs are available *explicitly*, via the 5C building blocks' *policies*. A major example of such a policy, from the software architecture point of view, is the *choice* to adapt the architecture to the number and types of cores offered by the CPU hardware: assigning one thread to one core, achieves *synchronous* execution semantics for all activities in that thread, *if* the operating system is *configured* not to deploy more than that single thread on that particular core (including its own “behind the screens” behaviour).

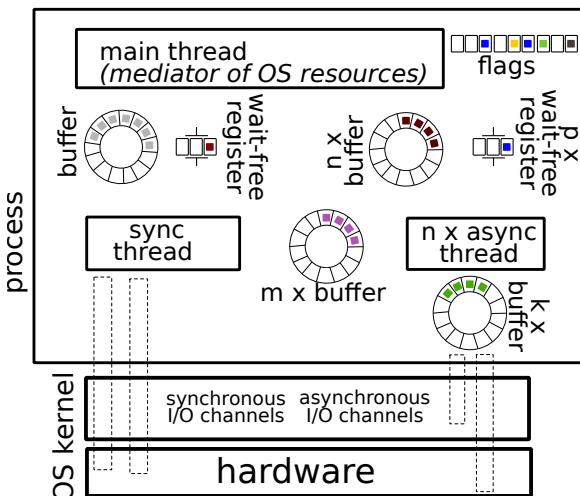


Figure 11.2: The generic *software template* of a process, with one *main thread*, one *synchronous thread*, and one or more *asynchronous threads*. All threads exchange their information via a composition of three primitives: **ring buffers** (when ordered “memory” is required), **registers** (when just the “most fresh” value of a variable matters), and **flags** (variables that can be written and read “atomically”).

11.1.1 Process: main thread owns interactions with operating system

This Section covers the inevitable software building block in all applications: the **process**. Indeed, to execute an application, a **process** with the application's software *has* to be started

²Interesting background literature for deployment on a Linux kernel is in this document: [Is Parallel Programming Hard, And, If So, What Can You Do About It?](#).

in a **shell** of the operating system. Or on the **bare machine**, that is, without using an operating system. In the systems-of-systems context of this document, not using an operating system is seldom an option.

A **process** *always* comes together with a **main()** function in the *default thread*. This is *not a choice* but a *constraint* imposed by the technology: starting a program in an operating system inevitably implies that the operating system or the processor hardware starts executing that **main()** **entry point** function (with, respectively without, creating that **main()** thread). The developers do have *a lot of choice* of their own, still, about:

- which *behaviour* to put in that **main()** function.
- which data *structures* to give the **process** exclusive ownership of, and which ones to give to the individual **threads** created in the just-mentioned **main()** function.

Starting with the *behaviour* part, the **type signature** of the **main()** function is as follows (in the **C programming language**):

```
#include <stdlib.h>
int main(int argc, char **argv, char **envp);
{
    ...here comes the application's code...
}
```

with:

- **argc** the *number* (“argument counter”) of the **command line arguments** that were given to the **process** in the shell from which the **process** was started.
- **argv** the *pointer* to where the “argument values” can be read. These values have been allocated by the operating system, which also filled in their values before handing over to the **main()** function.
- **envp** the *pointer* to the **environment variables** of the **shell** from which the **process** was started. Such environment variables are a primitive form of **DOM** approach to *configure* parameters “inside” a process by giving them values from the “outside”. The **process** can read and update environment variables during its whole life cycle, while arguments are a “one-off” way of configuring the **process** at start-up time.

Common examples of *command line arguments* are:

- the **internet address** of a “service” that the **process** expects to be available.
- configuration values for the amount of bytes to allocate in buffers.

Common examples of *environment variables* are:

- the version of the operating system kernel on which the **process** is executed.
- the **account name** of the user who launched the **process**.
- the language that that user selected as the **locale** to use for interaction via screen, keyboard and/or **log files**.

The software entities **to interact with the operating system** (and hence also with the **hardware** that the operating system encapsulates) are under the ownership of the **process**. They are:

- the data structures connected to the creation and configuration of:
 - *computational* mechanisms, that is, **threads**, needed by the application, other than the **main()** thread.
 - *communication* mechanisms offered by the operating system, such as the **standard streams**, **file descriptors**, or **pipes**.

- *coordination* mechanisms offered by the operating system, such as **mutexes** or **condition variables**.
- the runtime *configuration* of all of the above. That is, the algorithms that make use of the above-mentioned data structures to implement the interaction with the operating system.

With respect to the *computational* (i.e., thread-related) **behaviour** of a process, the following minimal **example** didactically instructive:

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <pthread.h>
5 #include <unistd.h>
6
7 #define NUM_THREADS 5
8
9 void *perform_work(void *arguments){
10    int index = *((int *)arguments);
11    int sleep_time = 1 + rand() % NUM_THREADS;
12    printf("THREAD %d: Started.\n", index);
13    printf("THREAD %d: Will be sleeping for %d seconds.\n",
14           index, sleep_time);
15    sleep(sleep_time);
16    printf("THREAD %d: Ended.\n", index);
17    return NULL;
18 }
19
20 int main(void) {
21    pthread_t threads[NUM_THREADS];
22    int thread_args[NUM_THREADS];
23    int i;
24    int result_code;
25
26    //create all threads one by one
27    for (i = 0; i < NUM_THREADS; i++) {
28        printf("IN MAIN: Creating thread %d.\n", i);
29        thread_args[i] = i;
30        result_code =
31            pthread_create(&threads[i],
32                           NULL, perform_work, &thread_args[i]);
33    }
34
35    printf("IN MAIN: All threads are created.\n");
36
37    //wait for each thread to complete
38    for (i = 0; i < NUM_THREADS; i++) {

```

```

39     result_code = pthread_join(threads[ i ] , NULL);
40     printf("IN MAIN: Thread %d has ended.\n" , i );
41 }
42
43     printf("MAIN program has ended.\n");
44     return 0;
45 }
```

The example creates five threads, and each of them does nothing more than to wait for some time and then print a message. But the example does contain all the data structure bookkeeping needed to work with ([POSIX](#)) threads:

- line 7: a hard coded “data structure” with the number of threads to create.
- line 15: the largest “activity” of each thread is to [sleep](#), that is, it informs the operating system that it should not be scheduled during a particular period of time.
- line 20: the [void](#) parameters in this [main\(\)](#) function indicates that it does not provide variables to get access to the [command line parameters](#).
- line 21: the data structures provided by the POSIX library, to store all the state information for the threads, as the interface between operating system and application.
- line 31: the [pthread_create\(\)](#) function call fills in the above-mentioned data structure. For the application, the important argument is the [perform_work](#) function pointer to the code that the thread will start executing as soon as the operating system has scheduled it on a CPU core.
- line 30: the [pthread_create\(\)](#) function call returns an integer value which is the ID that the operating systems uses to refer to the created thread. For example, to put it on the scheduling queue of a CPU core.
- line 39: the main thread waits for each created thread “[to join](#)” again: the operating system is informed that the thread need not be scheduled anymore, and that it can delete the above-mentioned state data structure for that thread.

Most of the [pthread_t](#) data structure is only used by the operating system, for example, to connect a *function* to the [program counter](#) of a CPU core, and to store the [conditions](#) under which the operating system should “*wake up*” the thread. The [identifier](#) in the thread data structure can also be used by the application, *to configure* some parts of the operating system’s behaviour with the thread; for example, the [scheduler](#) or timing policies. Such configuration takes place by [system calls](#), for example to [sleep\(\)](#) (line 15), or by selecting which [signals](#) should wake up the thread. These signals allow interaction from the shell to influence the process execution; such interactions take place via the keyboard, with key combinations such as CTRL-D or CTRL-C, or via [signal system calls](#). Signals are not present in the [code snippet above](#), but [here](#) is a much more elaborate code example that does use signals.

With respect to the thread-related ([data](#)) [structures](#), it is a [best practice](#) to keep the data about *all interactions* with the operating system inside the [process](#); or more correctly, inside the activity that is executed in the [main\(\)](#) thread of the [process](#). The reason to do so is that all of them are *resources* with a [life cycle](#), and spreading the management of that life cycle over one or more other activities than the [main\(\)](#) thread would introduce inefficient coupling. Not in the least because that [main\(\)](#) thread is the place where these resources are *created* and *configured*. Even when the application wants an activity eventually “*to write to a file*”, that coupling of what the activity thinks it should be doing with the operating system primitives,

can and should be avoided. The proven approach is to add to the `main()` function the data structures and functions to *mediate* between the application's information architecture and the concrete operating system:

- developers of the information and software architectures should not let activities “write their data to file”, but they should let them “log their data over a logging *stream*”, and that logging stream is fully under the control of the *application*.
- it is the mediator's responsibility to decide in which *operating system*-facing *streams* that logged data ends up in, outside of the process where it was created: files, pipes, communication to other processes, etc.
- the *only* responsibility of the `main()` function in the `process` is to create that mediator activity, so that it does not have to do that *multiple producer–multiple consumer mapping* itself. That responsibility should indeed belong to the application and not to the process. But because it *must* end up somewhere in that `process`, the `main()` thread is the most appropriate place.

In summary, the purpose of the `main()` thread is to implement all **process-centered** behaviour, boiling down to “just” the management of the threads and of the different forms of I/O for which the services of the operating system are indispensable.

The typical **Life Cycle State Machine** of a `main()` thread has the following behaviour in its states:

- *creating/deleting*: in many applications, these states are empty, because, by definition, the `main()` thread must be “kickstarted” with some *code* before it can start doing something. And all of the other process-related bookkeeping is done in the **configuring resources** state.

But in a design built around the **factory pattern**, these states must create (and later delete) the data structures needed to start up factories. This often involves reading the command line parameters and environment variables, because they contain the “kickstart” magic numbers. For example, to create a **memory pool** in which a factory can allocate the “objects” it provides to the application. That is sometimes called an **object pool**.

- **configuring resources**: the bookkeeping of threads centers around the `pthreads_t` data structure, to store all the “state” of each thread, and to configure it via **system calls**. Similarly, the process must do the bookkeeping of all operating system facing services, like sockets etc.

All the above-mentioned services run via data structures that are designed to be managed at runtime, which fits into the **factory pattern** that this document advocates for all software entities.

One very important configuration to make is the **choice** of how to bring the **application-centered** behaviour to life. Two major options exist to launch the **main activity** of the application:

- launch it in the form of a *thread function* in a **separate thread**, that is the so-called the “application main thread”.
 - launch it in the form of an *activity* in **this process**'s `main()` thread. This implies that also this `main()` thread must (create and) configure the data structures for *activities*, as explained in Sec. 11.1.2.
- **configuring capabilities**: for example, (re)programming timers, or changing thread priorities.
 - **running**: if the `main()` thread is configured to execute some application activities, it

then calls the `schedule` function provided by the application. In other words, it behaves exactly like any other thread in the `running` state.

- **pausing:** in this state, “nothing” should be done with the application’s threads, which boils down to executing a `no-op` function. Or possibly some resource-centric bookkeeping and logging functions: how many times was the thread executed, with which latencies, etc.

11.1.2 Thread: executing an application via activity schedules

As explained in the previous Section, the application-centric `threads` are created in the `main()` `thread` of a process (or in one of the threads created in that way already). A thread’s role is limited to supporting the `execution` of (part of) an application: it is merely a container exposing `one function` from the application—its `entry point`—to the operating system. Hence, it is the application’s responsibility to make sure that that (externally visible) entry function is connected (internally inside the application) to (the entry point of) the appropriate activity, at all times. In other words, this document advocates the following `loose coupling` approach: the application takes care itself of the `correct configuration` of the `scheduling` of its activities, at any moment in its active life time, within the thread it is assigned to. In this way, all application-specific “mediation” between activities is realised *inside* one or more of the application’s activities, and the result is *accessible* by the thread via its activity schedule data structure. Whenever that thread is executed by the operating system, it just calls the configured schedule of activities, via a `function pointer` in that schedule data structure.

Each activity in itself `implements the same mechanism`, but then to make sure it connects its (thread-visible) entry point to the (not thread-visible) entry points of the *algorithms* inside the activity.

Data structures: schedules of activities

The following data structures support the above-mentioned *loose coupling* between (i) a thread, and (ii) the activities inside:

- *list of activities* that *can* be executed in this thread. For example:

```
typedef enum { Activity_1, MyActivity, Her_Activity } Activities ;
```

Providing an explicit data structure for this list, fits in a `factory` approach, which is the pattern to allow activities to be added or removed from threads at runtime.

- *list of schedules* of activities: each of them is an *ordered sequence* of activity executions, which is relevant to the application. For example:

```
schedule1 = Activities array[1] = { Activity_1 };
schedule2 = Activities array[2] = { MyActivity, Her_Activity };
schedule3 = Activities array[3] = { Activity_1, MyActivity, Her_Activity };
Schedule_list = array[3] { schedule1, schedule2, schedule3 };
```

One activity can occur more than once in a particular schedule, and in more than one schedule.

The major use case of the schedule data structure is to provide one single place to store all information about an activity’s behaviour in each of a thread’s `life cycle` states.

Exactly the same mechanism is used inside each activity, for the bookkeeping of the *algorithms* that are executed in each of that activity’s `life cycle` states.

- *actual schedule* to be executed *next time around* in the execution of this thread. For example:

```
Current_schedule = schedule2;
```

It is this, and only this, `Current_schedule` function (pointer) that the thread will execute. Hence, the proper behaviour of the application then depends on the proper *configuration* of activity schedules in the application. The best practice in an application's information architecture is to foresee an explicit activity in each thread to realise the required schedule configuration. That activity must be scheduled to run in the thread's `configuring capabilities` life cycle state.

Behaviour: looping, scheduling and yielding in the Life Cycle states

A thread has *only one function* that is *registered* at its creation, and to be called later one when the thread is scheduled for execution by the operating system. The typical `control flow` of a thread function has a `while loop` at its core (implementing the `4C event loop` of the information architecture), with a `yield()` system call inside:

```
// "create" and "resource configuration" Life Cycle states:
int aCondition = 1;
...
while ( aCondition ) {
    // each of the Life Cycle states "configuring capabilities",
    // "running" and "pausing" has the following behavioural structure:
    compute_and_execute_schedule ( ... );
    if ( ... ) then aCondition = 0;
    yield ( ... );
    // It's the application that determines what functions are executed,
    // by configuring its schedule with the function the
    //     compute_and_execute_schedule ()
    // must point to.
}
// "resource configuration" and "delete" Life Cycle states:
...
```

The control flow above has the following complementary behavioural parts, linked to its `life cycle` states:

- `create` and `configure resources` states: before the thread enters its `while()` loop, it goes through these states, which are typically so trivial (just creating and/or initializing the schedule data structures) that it does not add much value to make them explicit stand-alone states in the thread's Life Cycle state machine.
- `capability configuration`:
- `running`: the thread runs in that `while()` loop until it detects a condition **to decide to stop** looping. The behaviour in this state is **to decide about its schedule**, that is, it computes which sequence of activities it has to trigger this time around in the loop. This document advocates that this “computation” is nothing else than following the indirection of the thread's `compute()` function pointer to the function pointer that the

application has provided in its scheduling function (executed in the `configure()` part of the thread's event loop).

After the execution of that schedule, it **decides to yield** to the operating system, that is, it allows the operating system to remove the thread from its CPU schedules, now.

- after the thread has left its `while()` loop, it goes back through its `configure capabilities`, `configure resources` and `delete` states, this time with the purpose of leaving its data structures in a state with which the `process` can do a clean `join()` of all its threads.

Often, the `yield()` and the `while()` functions are combined, like in `condition variables` (e.g., `pthread_cond_wait`), timed `sleeps`, `mutexes`. They all have the side effect that the operating system (i) removes the thread from its current CPU schedule, and (ii) will put the thread on a CPU schedule *again* sooner or later. The application can configure the operating system to select the same CPU every time, but it has no direct control over the decisions that the operating system takes for the scheduling of threads.

11.1.3 Activity: coordinating the execution of algorithmic functionalities

From a behaviour and data structures point of view, an activity has the **same architecture as that of a thread**, but with lists and schedules of `algorithms` instead of `activities`. That is, each activity has one or more schedules, each representing the serialized execution of one or more algorithms.

The difference in the roles of thread and activity in an application, is in the *functionality* that is implemented in both: the activity **encodes** the behaviour of the **application**, while the thread does only the **interfacing** of activities to the **operating system**. This difference is reflected in their **control flow**: the control flow of an activity is a **4Cs-based event loop**, which is a lot richer in behavioural semantics than the control flow of a thread which is behaviourally just a `while` loop with a `yield`. This Section explains which software entities to add in the `communicate()`, `coordinate()`, and `configure()` functions of the generic 4C pattern. The latter's `compute()` function is trivial: just call the current `schedule`, which is one of the outcomes of an earlier execution of the three functions above.

Communicate(): the goal is to identify which events, flags and protocols to take into account in the `coordinate()` afterwards. Here are some common policies of `communicate()` functions:

- **push**: the decision making data structures are filled in, either by information (events, flags, protocols) from the operating system and hardware (in case “something happened” with the resources that they provide to the application), by the application itself (that is, in one of the activities, including possibly the reacting activity itself). In the latter case, the change information is generated in one of the synchronously executing algorithms of the activity, and the reaction to that information is realised completely within the activity, without support from other activities or the operating system.
- **poll**: (an algorithm in) the activity actively scans the data coming in through one or more *streams*, and doing so detects the “something happened” conditions that are transformed into local event, flag and protocol data structures, not shared by other activities.

Coordinate(): the goal is to select which “behavioural state” to reach in the `configure()` afterwards. This is realised by calling a separate `coordinate()` function in each of the activity's **Life Cycle State Machine**.

`Configure()`: the goal is to select the right schedule of algorithms to `compute()` afterwards. The common policies of the `configure()` function are `prepare`, `check`, `dispatch`, `finalize`. They are *bookkeeping* functions, that are (possibly) executed early in each iteration through an event loop, and in that order,³ for each individual I/O source:

- `prepare`: do source-specific configurations to make it ready for `polling`, if needed.
- `check`: read some “register” data in a source’s local data structure, to find out whether it is ready to be polled in this ongoing run through the event loop.
- `dispatch`: actually execute the `poll` and corresponding `callback` function.
- `finalize`: do source-specific configurations to “clean up” a polled source, if needed.

11.1.4 Algorithm: executing the functionalities

Each of the above “4C behaviours” of an activity, executes one or more algorithms. Which ones that are depends for the full 100% on the application. But from architectural point of view, it makes a lot of sense to give every algorithm that has to be deployed into a 4C-structures activity a similar structure itself. This is a rather obvious requirements, because, after all, the simplest possible *activity* consists of just one single *algorithm*.

11.2 Single-process component: policies and best practices

This Section introduces design suggestions, that have proven to work well in particular application contexts.

11.2.1 Process with realtime behaviour: synchronous thread

In the context of cyber-physical and robotic systems that have to interface some hardware with `low latency`, the `process` typically creates two other types of `threads`, in addition to the main thread:

1. one `synchronous` thread (or, *hard realtime* thread, or *low latency* thread).

The `event loop` of this activity relies on the assumptions that *none* of its functions will ever (i) `block`, or (ii) be `preempted` by functions in other threads. That assumption need only hold when the activity is in the `running` state of its `Life Cycle State Machine`, and when the hardware that it is controlling is accessible via wait-free `memory-mapped I/O` or via stream buffers from hardware interface activities in threads that *can* wait for the hardware to react.

The *non-blocking* requirement must be realised *by design* (of both hardware and software):

- its communication needs with the hardware happens with technology that allow synchronous execution. For example, real-time networks like Ethercat or CAN, or memory mapped I/O.
- its interactions with other threads in the process runs always via stream buffers.
- it gets the `highest priority` from the operating system scheduler, and/or
- it is deployed on its `own` private `CPU core`. It is possible to start one such synchronous thread per core, *if* the operating system is configured in such a way to allow only that thread on that core.

³Functions from several sources can be taken together, respecting the same overall order.

- its **virtual address space** is locked into RAM (to prevent that memory from being paged to the **swap area**), via system calls like `mlock()`.
 - the **interrupt** lines on the system are **masked**, except for the interrupts that the thread needs itself.
 - by definition of the word, the “highest priority” can (or, rather, *should*) be given⁴ to this one thread only, because if two threads are given the same numerical priority, only one of them can really be executed first.
2. one or more **asynchronous** threads (or *soft realtime* threads). These execute the activities that will inevitably have to wait for (i) the hard realtime thread, (ii) some hardware I/O, or (iii) the interaction with other asynchronously executing activities, of the following types:
- a stream interface towards activities than run in other processes (including device drivers for hardware, and local area networking).
 - a coordinator for a set of other activities (via event processing on the event streams to and from these activities).
 - a **mediator** to coordinate a set of other activities, by being the first consumer of their event queues, and co-participant on their protocol flags.

Each of the **asynchronous** threads interfaces with one or more other activities, and is allowed *to wait* while doing so. In order not to let the **synchronous** thread wait, a stream buffer between an asynchronous thread and the synchronous thread was given ample⁵ resources at deployment time.

11.2.2 Event loop design trade-offs

An event loop allows the **application** developer (and not the operating system) **to configure** the balance between the following **design forces**:

- **localising synchronization of side effect-full data exchange.** Side effects inevitably take place, via (hardware and software) mechanisms like **interrupt handlers**, **mutexes**, **condition variables** or “lock-free” and “wait-free” buffers, etc. The pattern’s solution is *to copy* the data from/into asynchronous sources into/from a “**thread-local**” **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.
- **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.

⁴Here is the **POSIX** way of configuring priorities.

⁵What “ample” means exactly depends completely on the application context.

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

11.2.3 Application mediation activity

Many applications have one particular activity, the [mediator](#). Its role is to make sure that all other activities in the application are appropriately managed together, without them having to know which other activities there are in the application. Such a mediator does following things in its [Life Cycle State Machine](#):

- in the [creating](#) state:
 - check statically allocated resources for all threads.
 - creating the streams that buffer data between activities
 - creating the finite state machines that coordinate the behaviour of each activity.
- in the [configuring_resources](#) state:
 - setting intial state of all activities' LCSMs.
 - initializing all streams: pointers and status flags.
 - configuring the “Ports” of the activities that must access the streams, as producer or as consumer.
 - configuring their initial states.
- in the [configuring_capabilities](#) state: (TODO)
- in the [running](#) state: (TODO)
- in the [pausing](#) state: (TODO)

11.2.4 Support of operating systems

Operating systems offer support for [resource reservations](#), of CPU, [memory](#), and I/O. For example, the Linux operating systems offers software support in the form of, for example, [mmap](#), [mlock\(\)](#), [cgroups](#), or [systemd](#).

Operating systems differ in the set of software entities that are [owned](#) by a process on the one hand, and by each of its threads on the other hand. [This source](#) gives concrete details, more in particular about the differences between [POSIX](#) and [Linux](#) threads. Especially the wide variety in [unique identifiers](#) (“IDs”) in the (non-formalized) meta model of the process is an indication of its central role in any software architecture.

11.2.5 Mediation on shared resources

The [mediator](#) can rightfully be called a “[software pattern](#)”, because there exist already various realisations, with mature and large-scale application track records. Here are some of the common realisations in the domain of ICT infrastructure:

- [bandwidth throttling](#).
- [CPU throttling](#).
- [process throttling](#).
- [garbage collection](#).
- [memory pool](#).
- [thread pool](#).

- *isolation of runtime*: all the runtime's data are copied, such as the heap, calling stack and garbage collection, with a different *context* for each *application activity*.

The application's "shared" and "global" data are copied for each of a number of concurrent activities. For example, `V8:Isolate` and `V8:Context` in the [Chrome V8](#) runtime engine. (The isolation and context pattern is in itself *not sufficient* for *thread-safe* execution.

(TODO: best practice: every shared resource needs a mediator activity for [resource management](#).)

11.2.6 Event loop with ring buffer and scatter-gather I/O

The software architecture in this pattern has the following parts:

- the *ring buffer* data structure (Sec. 11.4)
- an event loop (Sec. 2.4) that executes the producer activity in one process.
- an event loop that executes the consumer activity in another process.
- both event loops also run the activities that realise the [vectored I/O](#) (also called [scatter-gather I/O](#)) that is needed at both producer and consumer sides.

The scatter-gather activities are needed because each message in the stream between both processes can contain chunks from different consumers and/or producers in each process, so the messages have to be composed before sending and decomposed after receiving, and their parts copied to the correct local destinations.

Some [memory management](#) and [compute kernel](#) hardware has this functionality built in.

11.2.7 Best practice: multi-rate time series streams

The [event loop](#) is the computational work horse in software activities, and the following use case presents itself in many control systems:

- the event loop services multiple activities, each with one or more solvers, that each process one or more time series
- [multi-rate sampling](#): the iterations for several solvers run at an order of magnitude difference in time scales. For example, a current control loop at 5kHz, a torque control loop at 1kHz, a platform velocity control loop at 100Hz, and a Finite State Machine for the task [plan](#) at 10Hz.
- the exact frequency is not so important; the order of magnitude is.
- many of the time series require [timestamping](#).
- many use cases want to process the various data streams together, with time as the major index.

If system designers have gone through the effort of making models for each activity and stream, because tooling can use the models to exploit having the overview of everything that has to happen in each iteration of the event loop, **to configure** the latter's implementation with the following optimizations:

- one time stamp can serve all streams.
- all provenance metadata need to be recorded only once.
- the timing in the multi-rate sampling can be chosen to be [powers of two](#). For example, 8Hz, 124Hz, 1024Hz and 4096Hz, instead of the above-mentioned 10Hz, 100Hz, 1000Hz and 5000Hz. This allows efficiency gains in the selection of which solvers to trigger in

each iteration: the 64-bit integer that the event loop uses to count its “ticks” just needs an efficient [modulo operation](#) for this selection.

- the overhead of [vectored I/O](#) (or [scatter-gather I/O](#), of individual data chunks in individual streams) can be avoided by serializing all data chunks to or from one single I/O stream.
Tooling support to do this efficiently, network-order independently, while keeping [direct access](#), exists in mature projects like [FlatBuffers](#).
- often, a solver in an individual activity needs two forms of [iteration](#):
 - incrementing its “tick”, to select the next entry in the stream buffer to read from and/or to write to.
 - iterating over its own algorithmic serialization. For example, [to visit](#) all links and joints in a kinematic chain.

The event loop can take care of the former, efficiently, using the [fetch-and-add](#) operation to set the starting position of the second iterator to the corresponding position in the stream; the solver can then just increment that second iterator, as if it were starting from zero.

11.3 Mechanism: stream as a ring buffer byte array

This Section designs the [ring buffer](#) (or [circular buffer](#)) implementation of a *single producer-single consumer stream*. It uses a [finite array of bytes](#), with one [port](#) for the producer and one for the consumer, to store these activities’ [indices](#) of their data chunks in the stream.⁶ The buffer is a *shared resource*, hence it must come with a [mediator activity](#), providing ownership of the resource, its [Life Cycle State Machine](#), and the protocol flags for the *peer-to-peer* coordination of the *producer* and *consumer* activities.

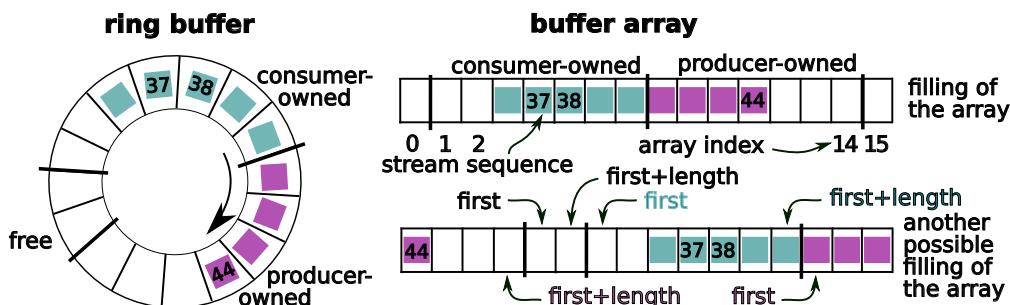


Figure 11.3: The conceptual primitive of the [ring buffer](#) is mapped onto the concrete data structure of a finite, contiguous *array of bytes*. The mapping is not unique: the same ring buffer can be realised by different fillings of the same buffer array. The *indices* on the array are owned by three activities: the *owner* of the buffer; a *producer* and a *consumer*, who register with the *owner* to get access to the buffer. Each of the three activities has its own *port* on the array, storing its own *first* and *length* parameters.

⁶To the best of the authors’ knowledge, this mechanism was first introduced in the [Disruptor](#) variant of the ring buffer.

11.3.1 Data structures: array, port, index

The ring buffer model is mapped onto a finite array of contiguous bytes in the **main memory** of a computer, Fig. 11.3, with the following (symbolic version of the) **data structures** at its core:

```
// A "RingBuffer" is a contiguous array of bytes in main memory:  
RingBuffer : {  
    Address : Start ; // the array's first data chunk in main memory  
    Integer : Size ; // number of entries (>0) in the buffer's array  
    Port    : Owner ; // the owner's own Port  
    Port    : { Producer, Consumer} ; // Ports of registered activities  
}  
  
// A "Port" gives an activity access to the buffer:  
Port : {  
    Buffer : ... ;           // to which this Port is attached  
    Index  : first ;        // first chunk in this Port  
    Size   : next ;         // number of chunks in this Port  
    Size   : end;           // number of contiguous chunks in this Port  
                           // towards physical end of array, when  
Boolean : wrapped ;       // there is a hole in between contiguous  
                           // sub-arrays in this Port  
Boolean : activity ;     // Active (true), Inactive (false)  
Boolean : backpressure ; // HighWater (true), LowWater (false)  
Integer : backpressureThreshold ;  
                           // threshold on buffer usage before it is too full/empty  
}
```

A **mediator** architecture is used to interconnect the **owner** activity of the ring buffer, and two other activities that register with the **owner** in their roles of **producer** and **consumer**:

- the **owner** owns the following data structures:
 - **Buffer**: this stores the information of where in main memory the array is deployed and how many bytes it contains; in addition, it keeps track of the **Port** data structures of the **producer** and **consumer** activities.
 - **Port**: the **index** in the array of the first byte owned by this peer, and the length of the owned area; the flags to indicate wrapping at the end of the physical memory; **status flags** of **activity** and **backpressure**.
- the **producer** and **consumer** each own their own **Port** data structure.

11.3.2 Mechanism: contiguous data for producer and for consumer

Keeping the buffer memory of the producer and of the consumer contiguously together in memory can improve **memory access performance**. However, the implications of this particular implementation of a stream buffer are:

- when adding entries to a sub-stream makes the array overflow, one has **to copy all** entries to the beginning of the array.

- to guarantee that this can always be done, the size of the ring buffer array must be **double** the size that is guaranteed to the consumer or producer to own.

Indeed, even with an almost full ring buffer, the producer activity must still be able to copy its sub-stream from the “end part” to the “start part” of the ring buffer array, so it is possible that the consumer substream must be copied first towards the “middle part” because it still occupies the “start part” of the array.

In summary, this design should only be chosen if the application has a much more frequent need for *processing* the data in the buffer than for *filling* the buffer. For example, for algorithms that can exploit the specific vector processing capabilities of the CPU hardware.

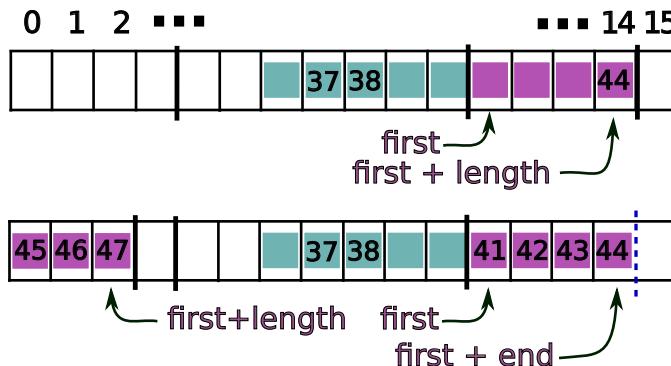


Figure 11.4: A ring buffer with “messages” stored only as *contiguous* byte arrays in main memory. This implies that sometime a hole must be left in the array between the last byte of a contiguous message and the physical end of the array.

11.3.3 Algorithms

register, acquire/release, etc.

monitors: check buffer invariants (indices must not overtake each other, must remain in bounds, infinitely increasing counter,...)

Efficiency:

Consistency: atomic operations (CAS, store)

- the **sequence number** of a data chunk in the stream buffer is an **integer number**, the **BufferIndex** k .

The sequence number is **incremented perpetually**, every time a new entry is produced. “Perpetually” is not completely true, but, even at a filling rate of one million entries per second, a 64-bit integer counter can guarantee that a **buffer overflow** will not occur for more than half a million years.⁷

- the **buffer’s array** has an **integer number** N of entries, represented by **Size**.
- referring to an entry in the array requires an **integer number**, called an **arrayIndex**. Such **arrayIndex** numbers are not constants, but are constrained to the **inclusive interval** $[0, N]$.
- by making the **Size** N a power of 2, the **arrayIndex** i of an entry in the stream can be computed cheaply from the latter’s **BufferIndex** k , via a **modulo** operation: $i = k \bmod N$. Indeed, a $\bmod 2^N$ operator is implemented as a very fast **bit shift**.

⁷ $2^{64}/10^6/60/60/24/365 = 584\,942$ years. A 32bit integer, with a sample rate of 1000Hz, only lasts for a couple of weeks: $2^{32}/1000/60/60/24/365 = 1/7.5$ years.

The example in Fig.11.8 has $N = 16$, so that `BufferIndex` 36 has `arrayIndex` 4, and `BufferIndex` 80 has `arrayIndex` 0.

- this particular choice of pointers and ownership simplifies the implementation of the **operators** to transfer data chunks from one activity to another, because that can be done without locking, in a thread-safe and **wait-free** way. The producer activity can indeed **change** the `LastFreed` pointer, without risking a **race condition** with the consumer activity, because the ownership transfer operation involving `LastFreed` does not overwrite any consumer-owned variables.
- ownership transfer just involves changing pointers, and **Compare-and-Swap (CAS) operations** on such 64-bit integers are supported as **atomic operators** by all operating systems, and even natively by CPU hardware architectures.

This guarantees **correctness**.

- putting all the `arrayIndex` pointers in their own **cache line** (filling the rest of the cache line with unused “padding” bytes) guarantees that they can be updated independently without causing each other’s cache lines to be refreshed.

This improves **performance**, without compromising **correctness**.

- the following **data consistency constraints** must hold at all times:
 - `Size` and `Start` should only change in the `configureResource` state of the buffer’s LCSM.
 - the buffer can not be filled to more than its capacity, that is, the `LastFreed` pointers of subsequent activities should never overrun each other.
- each activity acting on the buffer can apply a **backpressure** policy, with which to set the value of its `HighWater` and `LowWater` status flags. They indicate to the activity’s downstream and upstream activities whether it wants them to fill or empty their parts of the buffer, and can be computed by introducing two local integers, `HighWaterThreshold` and `LowWaterThreshold`, as follows:

$$\text{HighWater} = (* \text{ NewestToConsume.BufferIndex} - \text{LastFreed.BufferIndex}) \quad (11.1) \\ > \text{HighWaterThreshold},$$

$$\text{LowWater} = (* \text{ NewestToConsume.BufferIndex} - \text{LastFreed.BufferIndex}) \quad (11.2) \\ < \text{LowWaterThreshold},$$

Both status flags should be initialized to `false`, indicating that no backpressure policy is used.

- the status flags for backpressure and for **Activity** are of an **enumeration type**. These can be read and written atomically on all hardware platforms, hence, updating the status flags does not introduce race conditions.

11.3.4 Life Cycle State Machine

- **dead**: the activity is not able to do anything itself, and must be *brought to life* by another activity.

- **deploying**: the activity is busy with finding and configuring all the resources it needs, to be able to offer its services to others. While doing so, the activity **does not react to external coordination events**. This super state contains three semantically complementary states:
 - **creating**: the software resources are created, with which the activity does the bookkeeping of its own behaviour and of its usage of its external resources.
 - **deleting**: the above-mentioned software resources are cleanly removed.
 - **configuring resources**: the activity is configuring the service resources it requires for its own operation.

Resource *configuration* also includes *re-configuration at runtime*, so the latter state can be transitioned to and from multiple times during the life time of the activity.

- **active**: the activity is visible to other systems, to coordinate with them its engagement in mutual interactions. So, in this super state, the activity **reacts to external coordination events**, but with two semantically complementary behaviours:
 - **configuring capabilities**: the activity **is not** providing its services, because it (re)configures its own capabilities to provide a particular service configuration.
 - **ready**: the activity **is** providing its services to other systems. This super state contains two semantically complementary states:
 - * **running**: the activity provides its services.
 - * **pausing** (or “*idling*”): the provision of the activity’s services is put on hold, but can resume immediately. This happens when it must wait for an LCSM state change in one or more of the “peer” activities it interacts with; for example, during a [All to go, One to stop](#) coordination.

11.3.5 Protocol flags for producer-consumer-owner coordination

11.3.6 Policy: late binding of data chunk type

(TODO: just allocate an array of bytes, to be interpreted by the application, even at runtime.)

11.3.7 Policy: composition of data and metadata

The software-centric additions to the information architecture description in Sec. 2.9.13 are:

- the metadata `data_chunk` data structure must be defined. In the worst case, every data chunk requires its own metadata chunk, so the size of the metadata stream is that of the data stream.
- (hence) the metadata stream does not need another `backpressure` support or status flags in addition to the ones in the data stream.
- the ownership of the producer and consumer pointers in the metadata stream is with the same producer and consumer activities.

The former is a major responsibility of the information architect.

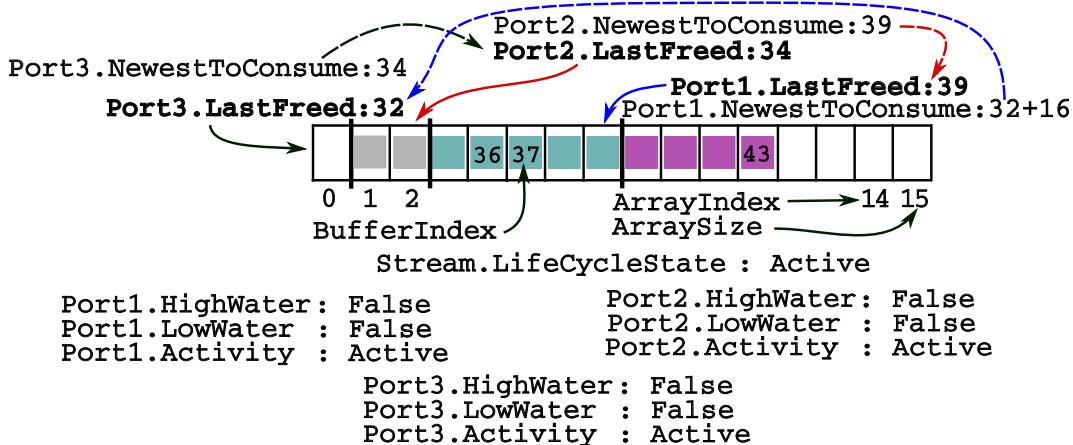


Figure 11.5: Example of an instance of the abstract data type of a `StreamBuffer` with three activities having a `Port` on the buffer. The first activity owns a lot more data chunks than it has already processed; the latter are identified with coloured slots in its `Port`, the former with blank slots.

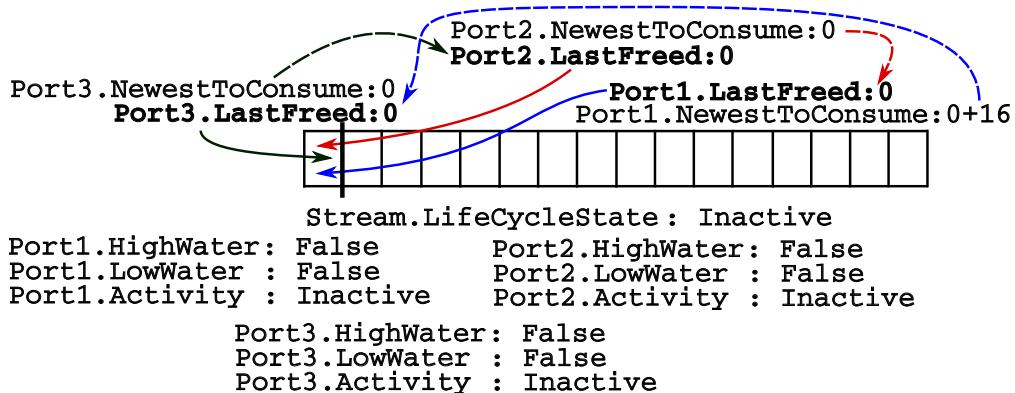


Figure 11.6: Empty/initial state of a `Stream` ring buffer with three activities. That means, that the whole buffer is owned by the first activity's `Port1`. Note that all indices into the `array` are 0, but the logical index into the stream buffer for the last `Port` is the buffer Size 16.

11.3.8 Policy: time series

Time series (or “data streams”, or “event streams”) are a very important use case of streams with metadata. (At least) three variants of the data structure of the metadata exist:

- the *singleton* metadata structure: each entry in the data stream buffer has its own time stamp, and it is part of the data chunk.
- the *metadata stream* version (Fig. 2.29). Each data chunk in such a metadata stream buffer has fields to represent:
 - the *range* of stream sequence numbers for which the following two metadata pieces of information hold.
 - the *provenance*: how was the data created, where does it come from, etc.

- the [time representation](#) for each range.
- the *one-on-one metadata stream*: the metadata stream buffer has the same size as the data stream buffer, and each chunk in the meta buffer contains the metadata of one data chunk in the data buffer.

(TODO: [Apache Arrow](#) software and models; role of developing standards on [getusermedia](#) (sources, streams, tracks and channels; capabilities and property settings; constraints between streams, tracks and channels), [audio](#), [WebRTC](#).)

11.3.9 Policy: event handling synchronisation

(TODO: when different activities have to react to a particular event, they can be put in a chosen order as consumers on a stream with just that event, so that they react to it in turn. Submission *and* Completion can be done in this way, by adding the same activity more than once in the “right” place of the stream.)

11.3.10 Policy: composition of stream and memory pool

This Section brings the information-level stream specialisation of the composition of the stream model with an [object pool](#) to the more concrete software level. That is, “objects” are assumed to be [serialized](#) into arrays, Fig. 11.7.

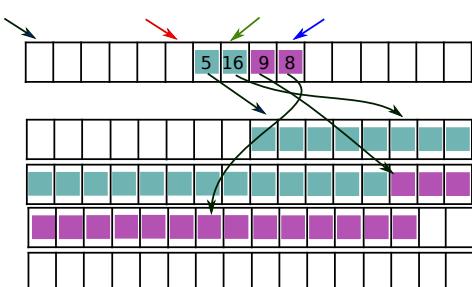


Figure 11.7: Stream buffer with memory pool. Each data chunk in the stream buffer contains two integers: the address pointer into the memory pool of the array that contains the serialized data chunk of the “real” data, and the size of that array. The stream buffer is used for the ownership transformation of these pointers from producer to consumer.

11.3.11 Policy: heartBeat/watchDog mediation for “lazy” stream

In a context in which the producer and consumer activities are [distributed](#) over several processes or computers, the status flags for [backpressure](#) and [peer_activity](#) of one peer can not be read synchronously by the other peer. So, that information has to be sent via communication messages. Then it makes sense to add [heartbeat](#) events: when there has not been a communication for some time⁸ an event is sent, by the producer and/or by the consumer, to indicate that they are still engaged in the stream communication but have had no need to send over data chunks for some time.

The [watchDog](#) approach serves a complementary use case:

- a third [mediator activity](#) is introduced.
- it waits passively for [heartBeats](#) from producer and consumer.

⁸That [timeout](#) is a configuration parameter.

- when they do not come in for a particular period of time, the mediator sends a `watchDog` message to the “delayed” peer, to trigger it in reacting with a `heartBeat`.
- if that also does not activate all involved peers, the mediator fires reconfiguration events for its subsystem of the whole system.

11.3.12 Standards and software projects supporting stream channels

Many established internet protocols are special (simplified) cases of the `Stream` meta model. For example, [SCTP](#) at the [application layer](#), [RTSP](#) at the [transport layer](#). Implementations for these standards come with mainstream operating systems.

At the time of writing, the [WHATWG stream standard](#) is reaching maturity. It includes an explicit meta model, as well as a reference implementation in Javascript. The ZeroMQ ecosystem provides a (partially conforming) model and an implementation in the form of its [exclusive pair pattern](#). Both implementations have chosen for the *policy* of blocking the producer or Consumer when their stream buffer is full or empty. This document follows a less constrained meta model, allowing other policies, such as: to overwrite the oldest or the youngest entries, or to compact a stream.

11.4 Mechanism: stream based on a finite buffer

The meta model of the [single producer/single consumer stream](#) represents how two activities interact asynchronously, in producer and consumer roles, to exchange data *buffered* in a *strictly ordered stream* of data chunks, where the **conceptually infinite** buffer is provided by a third activity. This Section provides the conceptual model with an explicit architectural design, that it is ready to be implemented (that is, using only a **finite buffer**), while still being independent of concrete [software](#) choices, such as data structures, programming language, operating system, processor and memory properties. The core information architectural design choices are:

- mapping the infinite stream onto the mechanism of the ring buffer abstract data type.
- embedding the stream behaviour into a mediator-based activity architecture, with an explicit [Life Cycle State Machine](#) to coordinate the access to the shared resource of the ring buffer.

11.4.1 Mechanism: producer-consumer ring buffer

The [canonical meta model](#) of the stream is specialised, *structurally*, by mapping the **conceptually infinite** stream onto a [ring buffer](#) (or “*circular buffer*”) of **finite length** (Fig. 11.8). The *behaviour* of `stream`, `producer` and `consumer` activities remains unchanged:

- the regions they owned in the conceptual meta model were already *finite* anyway.
- the `request` operation in the behaviour of the `producer` and `consumer` informs them about whether there is indeed space available in the ring buffer.

The `producer` and `consumer` activities transfer ownership as follows:

- they **themselves** move the barrier that they own further towards the oldest part of the sub-ring that they own. In other words, they shrink their own `port` at its oldest side. “Older” means: a lower value of the monotonically increasing logical stream sequence number.

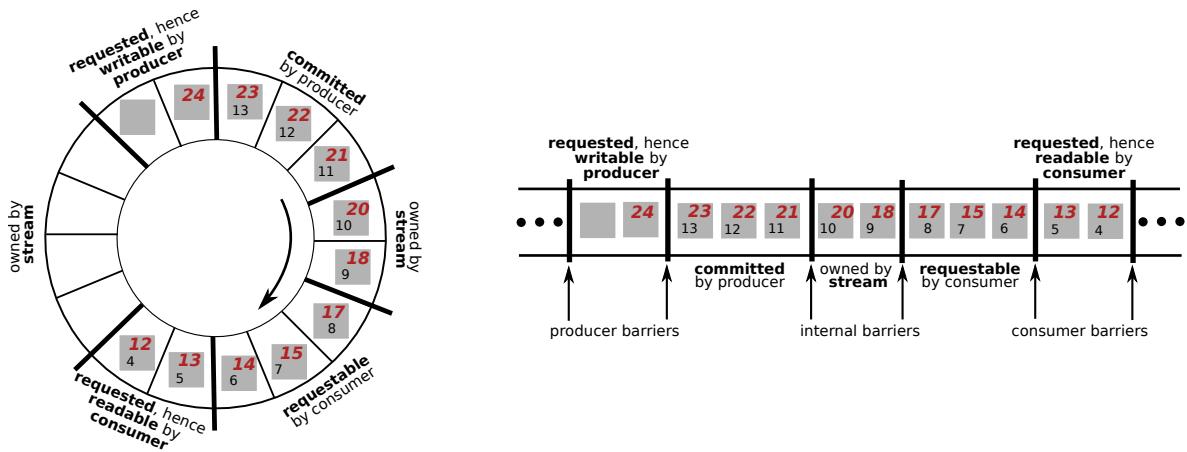


Figure 11.8: The canonical semantic concept of a **producer-consumer stream**, with conceptually infinite length (right), is mapped onto the information architecture model of a **ring buffer** with finite length. The contiguously filled part of the **stream** is mapped to contiguous areas in the **ring buffer**: one owned by the **producer**, one owned by the **consumer**, and one owned by the **buffer**; the latter also owns the *unused* part of the **buffer**. Despite its finite length, the **ring buffer** can order data chunks with a perpetually incremented **stream-index** sequence number.

- they **request** another activity to move the barrier on the newest side, that they do not own themselves.

The overall **behavioural** part of a stream **conforms-to** the Life Cycle State Machine meta model, as follows:

- the **setup** operator **conforms-to** the **configuring resource** step.
- the **register** and **unregister** operators correspond to the **configuring capability** state.
- the other operators correspond to the **running** state.
- the other states in the Life Cycle State Machine are not needed.

Formally, the behavioural model of the ring buffer has the following set of *operators* for each of the three relevant peer activities (Fig. 11.8):

- **owner**:
 - the **setup** operator initializes the stream (**connector** and **ports**), before it allows **producer** and **consumer** peers to register.
- **producer**:
 - (**un**)**register**: triggers the **owner** to (dis)allow the **producer** access (from) to the stream, and (re)initializes the **producer's port**.
 - **acquire**: try to get ownership⁹ of a new contiguous set of data chunks at the “upstream” part of its port. (That set comes from previous **releases** of the **consumer** to the **owner**.)
 - **produce**: release ownership of a contiguous part at the “downstream” end of its port. That part is then available to the **consumer**.

⁹This is Rust’s “move” of a buffer half from producer to consumer. The Go programming language uses this approach natively, via its policy “*Do not communicate by sharing memory; instead, share memory by communicating.*”

- **index**: gives a new data chunk a *unique identifier*, which is one larger than the identifier of the previous data chunk in the stream. When the **producer** can not put all data chunks on the stream that it should, it causes **missing data** gaps in the numerical order of the data chunk **identifiers**.
- **consumer**:
 - **(un)register**: triggers the **owner** to (dis)allow the **consumer** access (from) to the stream, and (re)initializes the **consumer's port**.
 - **consume**: try to get ownership of a new contiguous set of data chunks at the “upstream” part of its **port**, that comes from previously produced data chunks from the **producer**.
 - **release**: release ownership of a contiguous part at the “downstream” end of its **port**. That part is then again available to the **owner**.

11.4.2 Policy: ring buffer without a mediator

Fig. 11.9

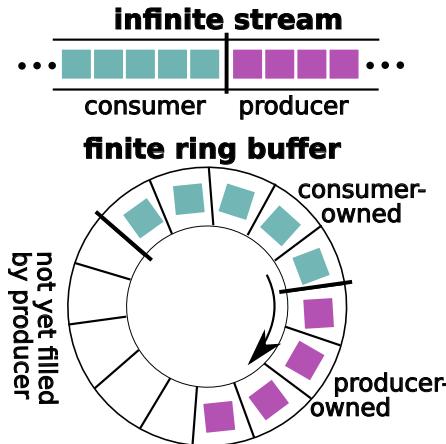


Figure 11.9: The special case of mapping an infinite **producer-consumer stream** on a finite ring buffer, in which all available data chunks are owned by either the **producer** or the **consumer**.

11.4.3 Policy: status flags for activity and backpressure

The following **status flags** can optionally be added to each **port**, to help the activities involved decide whether or not to spend efforts on operating on the ring buffer (Fig. 2.25):

- **activity**: this is an **enumerated type** with which an activity informs the other activities about its actual behaviour on that **port**:

- **Active**: the **port owner** is operating on the stream in its current behavioural state.
- **Inactive**: other activities do not have to expect any activity from the **port owner**.

This flag on a **port** is *owned* by the activity that owns that **port**. The flag is an instance of the generic coordination use case of **All to go**, **One to stop**.

- **backpressure**. A **producer** could fill a ring buffer faster than a **consumer** can process it, or the other way around, so each uses a status flag to inform the other peer algorithm that it wants to “speed up” or “slow down” the throughput in the ring buffer. This reactive mechanism is called the **flow control** on the ring buffer, or **backpressure**. This needs two status flags with the following meaning:

- **HighWater**: the **producer** on a port informs its **consumer** that the producer side of the buffer is getting too full, and, hence, it invites the **consumer** to start emptying its side the buffer.
This flag is *owned* by the **producer** activity.
- **LowWater**: the **consumer** on a port informs its **producer** that the consumer side of the buffer is getting too empty, and, hence, it invites the **producer** to start filling its side the buffer.
This flag is *owned* by the **consumer** activity.

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

Of course, an activity on a ring buffer is free to neglect the information in the status flags, and it is only allowed *to observe* the flags owned by the other activities.

The **activity** status flag is of a **higher order** than the **backpressure** flag: when the other peer is not **active**, it is useless to spend time looking for highwater or lowwater events.

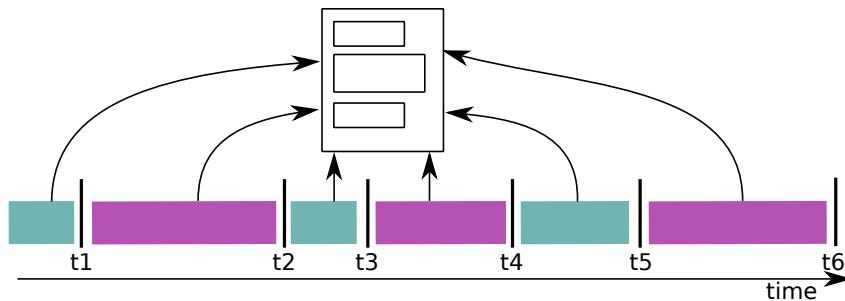


Figure 11.10: *Ping-pong* buffer: the buffer contains only *one data chunk*, and *ownership transfers* back and forth between two activities, to *synchronize* their reads and writes of that data chunk: at each moment in time, only one of them has access to the buffer.

11.4.4 Policy: ping-pong buffer — Synchronization with data payload

Figure 11.10 depicts the special case of the stream mechanism to realise a **synchronised dialogue** between two activities. The figure shows an ordered sequence of six moments in time (t_1 to t_6), where one activity transfers ownership of the buffer’s single data chunk to the other activity. In between these moments, one and only one of the two activities is *owner* of the data chunk. Hence, the owning activity:

- reads what the other activity has provided as an update to that data chunk, the previous time it had ownership.
- interprets that update, and implements its own response update.
- transfers ownership back to the other activity.

In other words, both activities are “ping-ponging” the buffer to each other. Figure 11.11 shows the Petri Net model behind that coordination behaviour. The ping-pong buffer is a **wait-free** [67] alternative for

- **mutexes** and other locking mechanisms. In this case, only the *synchronization* part of the ping-pong buffer is used. The *data chunk* part *can* be used to do some bookkeeping about, say, how many times a synchronization has taken place during a particular time interval.

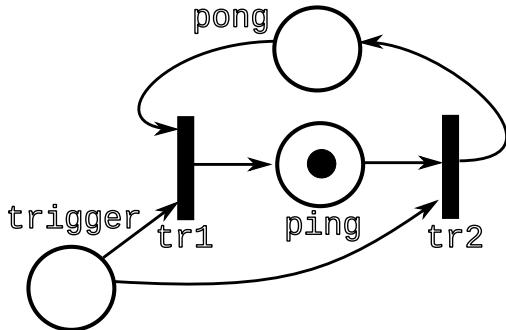


Figure 11.11: Petri Nets for *ping-pong* (or *flip-flop*) coordination. The constraint is that only one token can be in the **ping** and **pong** places, at the same time. If that constraint is satisfied at a certain time t , any **trigger** will continue to satisfy the constraint, resulting in a switch of place occupation between the **ping** and **pong** places.

- **Request-Reply** interactions. the producer writes the request data in the data chunk; after which the consumer reads it, takes the relevant action, and writes the reply in the data chunk; after which the producer can then read and process that reply from the consumer.

A major example of an application architecture that uses ping-ping (and **triple buffers**) is the [implementation](#) of the [EtherCat](#) protocol. (This is realised in the *hardware* of an EtherCat slave chip, and not in its software; but the architecture is the same.) EtherCat also provides a **ping-pong buffer**, called a *mailbox*; its use case is to communicate messages (“mails”) with *configuration* information, and the “ping” and “pong” coordination fits well to the EtherCat master *asking* for a configuration, and the EtherCat slave replying with what it *actually did* with the asked-for configuration.

11.4.5 Policy: triple buffer — Wait-free and last-update only

The information architecture of this type of buffer has been [introduced before](#). This Section extends that architecture with some specific software choices. For example, the [compare-and-swap](#) operation that is available on all modern processors.

All the state switches in the triple buffer protocol (i.e., **free-written**, **written-written**, **written-reading**, and **reading-free**) must be realised [atomically](#). That is, producer and consumer will never see:

- a slot that has no state or more than one state.
- two slots that have both the **written** state, or both the **reading** state.

This Section explains how the producer knows at all times which of the three slots in the buffer is **free**. *If* this hypothesis is indeed true, the producer never has to wait to write its newest data in the right slot. Here is the proof of the hypothesis:

- let's call the *index* of the assumed-to-be-free **free** slot **now**.
- the producer always knows which slot it has **written** itself the last time around. Let's call the index of that slot **min1**.
- it is an invariant fact of the buffer that, at all times, at most one slot can have a **written** marker, and at most one slot can have a **reading** marker.
- the consumer changes the marker of only a **written** slot. The first marker switch is from **written** to **reading**. Later on, after reading, the second marker switch is from **reading** into **free**.
- it is an invariant fact of the buffer that, if it exists, the **reading** slot has been filled the longest time ago. Let's call the index of that slot **min2**.
- it is *possible* that the index **min2** has become equal to **min1**, namely when the consumer

has caught up with the producer, has taken hold of slot `min1`, and has changed the marker of that slot to `reading`. In that case, the third slot is certain to be `free`.

- it is *also possible* that the consumer has caught up even further, and has also switched the marker of the slot with index `min1` into `free`.
- the producer has not been able to observe the two above-mentioned marker switches.
- the producer now makes an assumption: the consumer has not yet taken any slot. In other words, the assumption is that the status of the whole buffer has not changed since the previous time the producer operated on it. Hence, the producer tries an *atomic compare-and-swap* using `written` as the old value in slot `min1` to compare with.
- if that atomic operation *succeeds*, the producer's job finishes here, because:
 - the slot with the newest data is marked `written`.
 - the index `now` of the `free` slot is known.
 - the index `min1` of the `written` slot is known.
- if the atomic operation *fails*, the return value contains the current markers of all slots. If that return value shows that the consumer has set the `freezeProducer` flag, the producer need not update any markers, and can continue with its activity.
Else, the following cases are possible:
 - the marker of slot `min1` is still `written`, and the marker of slot `min2` has changed into `free`.
The producer can now retry the same operation, this time with a guarantee for success.
 - the marker of slot `min1` is now `reading`, and the marker of slot `min2` has changed into `free`.
The producer can now try another atomic operation, this time trying to switch slot `min2` to `written`. If that fails, repeat the procedure.
 - the marker of slot `min1` is now `free`. So, the producer can execute the atomic *compare-and-swap* with this slot, this time using `free` as the “old value” in that operation.

Efficient use of cache:

- the pointers to the buffers should not be in the same cache line.
- the buffers themselves should not have parts in the same cache line.
- all status flags of the buffer are stored in the same atomically readable variable; that means: `free`, `reading`, `written` (once for each of the three slots in the buffer) and the flag `freezeProducer` (only once for the whole buffer). This is “bad” for dirty cache lines, but that overhead is *necessary*, because of the requirement that atomicity is guaranteed.

11.4.6 Policy: heterogeneous data chunks — Message passing

Streams interconnect producer and consumer activities with a permanent interaction channel. It is not uncommon that the contents of the information exchanged between both activities changes over time; often even from data chunk to data chunk. Therefore, the abstract data type of the data chunk in the stream can be different for every message. In this context, one often uses the term *message passing* instead of *data exchange*, because the term “message” captures well that idea of individually changing the size and type of the data that is exchanged. The stream meta model can still provide its ownership transfer and loose-coupling properties as follows: the data in the stream buffer’s data chunks are not the “real” message data, but

just pointers to whatever data structure corresponds to each message (Fig. 11.12).

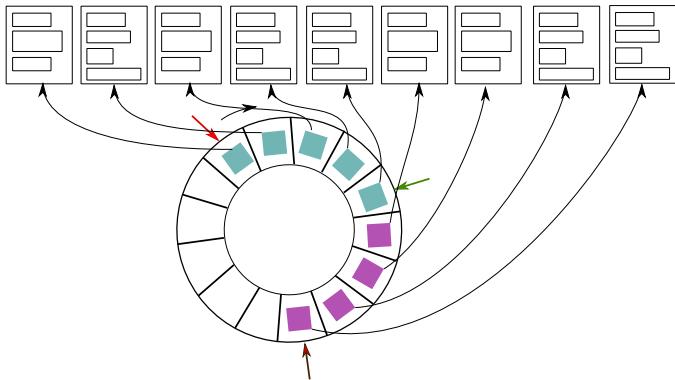


Figure 11.12: “Message passing” stream, with heterogeneous data chunks. The stream buffer is used only for efficient and effective ownership transformation of pointers to any type of data structure.

11.4.7 DOM model for hierarchy in dependencies between activities

The entities and relations in this Section have already received significant modelling attention, for example in the [AADL](#) standard and supporting [software and tools](#). Unfortunately, AADL (and existing alternatives) violates almost all “best practices” advocated in this document, such as: separation of mechanism and policy, [DOM based](#) hierarchical configuration, or compositability. This Section provides a [DOM model](#) that does conform to the compositability and compositionality ambitions of this document. It starts with the DOM models that have been introduced before to represent the [hierarchies of activities inside an application](#), with a focus on configuring all of the application’s [magic numbers](#) in the *right* level of the activity DOM hierarchy.

(TODO: concrete DOM model, tooling from the Web; [CRUD operations](#) on a DOM model; role of a [virtual DOM](#); visualisation; diffing and streaming.)

11.4.8 DOM model for hierarchy in dependencies between heterarchical streams

Section 11.4.7 introduced a DOM model for the *hierarchy in activities*. This Section introduces the complementary software architecture model that represents the *hierarchy of the communications* between activities.

Not surprisingly, this model will not have much hierarchy in tree-structured *containment*, but there is a tree-structure in the *constraints* between the ports involved in the streams.

(TODO: hierarchy of streams and their protocols: type (PubSub, RPC, SubmissionCompletion,...); coordination (with flow control, with active flags,...); configuration (dependencies of type and coordination combinations); operations (CRUD interactions,...).)

11.5 Architecture to compose task queues, workers, event loops, and solvers

This Section introduces the components in the **computational pattern** that is the software counterpart of the similarly named *concepts* of Sec. 10.6, and the *information architecture*

building blocks of Chap. 10:

1. **task queue**: data structure that lists “**what has to be done**”.
2. **solver**: computes “**who does what**”.
3. **worker**: serves as “**the agent that does it**”.
4. **event loop**: triggers the workers to actually “**do it**”.

11.5.1 Pattern: composition of “task queue” and “library API” patterns

System designs need a composition of

- concurrent and asynchronous activities deployed in threads, with
- serialized and synchronous function calls on data structures.

The coordination primitives of FSMs and Petri Nets work only for activities, because they are *declarative* models of *what* has to be done, and not *imperative* control flows that dictate *when* that has to be done. The declarative design approach fits naturally with “task queues”; the imperative design approach fits naturally with “APIs of libraries”. Both are needed in most systems. Only the systems that require that all activities are deterministically defined at compile time, can profit from a “library-only” design. Or rather: from a design where the only declarative part comes from the operating system’s management of resources, and the access to these resources. The latter kind of systems are most often encountered in deeply embedded applications (because of the lack of resources to solve the declarative descriptions in series of function calls), or in aerospace and avionics (because the high verification and certification expectations in these domains have, for the time being, favoured architectures where the software architecture is so-called “statically allocated”). Nevertheless, the latter domains are also the ones that pioneered the introduction of formal FSMs and Petri Nets, because these are the declarative models whose behaviour can be fully proven mathematically. At least, when the decision making is close to first order logic.

The event loop can rightfully be called a “software pattern”, because there exist already various realisations, with mature and large-scale application track records. Here are some of the largest realisations, with [industry-friendly open source licenses](#):

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

The FSM [states](#) of the loop.

The event loop loops over callback [sources](#), each providing [prepare](#), [check](#), [dispatch](#) and [finalize](#) functions; three default types exist: (i) timers and file descriptors (the OS does the waiting and the loop polls for ready events), and (ii) idle ones, that are always ready to be dispatched (that is, to run).

The loop can also deal with also “child thread” signals: CTRL-C, etc.

- [SystemD event loop](#)

States: [states](#)

- [ZeroMQ based event loop](#). It has a fixed attachment between sources, context, and thread.

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

11.5.2 Policy: frameworks, middleware, solvers

Two major instantiations in the domain of software engineering of the coupling between *mechanism* and *policy*, are **frameworks** and **middleware**: they **optimize the “usability”** of software by implementing the policy choices that are relevant in a particular application context. The advantage is *freedom from choice*: developers only have to make choices about the behaviour *of their application*, and not anymore about the *policies* of the *software basis* they rely on. The disadvantage is that these choices are most often hidden inside the framework, and hence **compromise the “reusability” and “composability”** of the application. if one or more of the policy choices are not optimal (or even feasible) within a different application context.

11.5.3 Composition of stream with object pool

This use case is a variant on the previous one, that fits well to an application context where the data structures for all messages must be allocated statically. This use case makes use of the **object pool** pattern, where a fixed number of instances of a fixed number of data structures are allocated in advance, and the producer of the stream just has to select a free one from the pool; the consumer must free that entry afterwards.

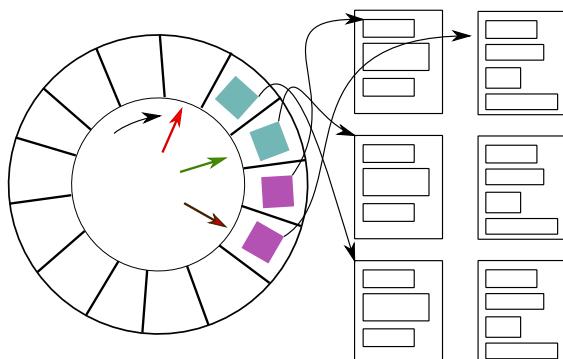


Figure 11.13: Stream with object pool. The data structures on the right are statically allocated, and they are only “borrowed” temporarily and “returned” at runtime, without being deallocated. The stream buffer is used only for efficient and effective ownership transformation of the pointers to the various available object data structures.

11.5.4 Pattern: schedule-based event handling in an event loop

One part of an application’s flexibility in how to realise a task, is to support runtime configuration of all activities involved in the task-solving, more in particular, in how they create their **schedule** reactively as part of their **event loop**. This Section explains the architectural mechanism that extends the event loop pattern with the *structure* and *behaviour* to realise *reactivity* with sufficient performance in:

- *effectiveness*: to realise *consistent access* to all data structures used in its *schedules*.
- *efficiency*: to minimise the *latency* between the moment that data *becomes available*, and the moment of *execution* of a schedule to process that data.

That reactivity is needed in the **5C** approach to system architecture, especially in the connection between the `communicate()`, `coordinate()` and `configure()` parts of the **event loop**. These are the *structural* parts of the mechanism:

- `event_queue`: the various **streams** with **event** data structures that an activity has received and must still process, or has fired itself and must still send out.

- **event_loop**: the data structure that an activity fills in every time it starts a new iteration through its **execution loop**, and that structures the order in which reactions to processed events will be executed.
- **schedule**: the data structure that represents a sequence of functions to call.

The *behavioural* parts of the mechanism meta model represent the **event handling**:

- **event_processing**: the computations that consume events from the **event_queue** streams, to produce (part of) the **schedule** of functions to call, at the end of this loop of event handling.

One often refers to this approach of turning events into functions-to-be-scheduled as “adding **callback** functions to an activity’s **event loop**”.

- **event_monitoring**: because **event_processing** is a first-class activity, it is obvious that it can receive its own set of monitors. Typical monitor functions are:
 - detecting whether a Finite State Machine or Petri Net is “running in cycles” while it should not. For example, the same nominal task execution plan is always interrupted by the same non-nominal situation, and the same error recovery plan that leads to the same task FSM state that the task plan execution started from.
 - event **tracing**: a new event can be fired each time a *particular sequence* (“trace”) of other events has occurred. For example, **activity_1** sends an event, after **activity_2** and **activity_3** have fired an event.

This monitoring is one of the *causes* that fires, from the inside, events to which an activity has to react.

11.5.5 Pattern: splitting a solver into scheduling and dispatching

Many **solvers**, such as **quadratic programming** or **kinematic and dynamic solvers**, consist of multiple **stages**, or sub-algorithms. For example, first converting the problem formulation to a canonical form, then finding a feasible solution, and finally iterating towards the optimum. Nevertheless, their *implementations* most often **run to completion** without allowing to be pre-empted between the different stages, or inside the “big” stage of iteratively approaching “the” solution. Only few solvers are designed with such **stop anytime** execution behaviour in mind. The latter ones are a natural fit for architectures based on **task queues and worker pools**. The pattern that has emerged in this context splits the solver in two complementary parts:

- **scheduler**: the computations that decide which parts of the total algorithm *are ready to be executed*. The outcome of the scheduler is that this list is added to the **task queue** of the activity in which the solver is deployed.
- **dispatcher**: assigning computational *workers* to *pick* a task from the task queue and *execute* it.

This pattern allows to compose the execution on one event loop of several algorithms from several programmes. The added value is that:

- whenever there is something to be executed in a complex activity, it can be executed as soon as the activity gets assigned to a CPU.
- the dependencies between several algorithms (that is, one algorithm needs a partial result of another algorithm) can be optimized by a **mediator** activity that has *knowledge* about these dependencies.

A robotics context in which this pattern is highly effective is that of the integration of **sensor fusion**, **moving-horizon estimation** and **model-predictive control**: all of these parts require

information about the kinematic and dynamic state of the robot to which the sensors are attached and for which the controller is working, but a “run to completion” design of the algorithms results in the same kinematic and dynamic computations to be repeated several times.

Collaborative pre-emption in iterative solvers

(TODO: linked to the [scheduler-dispatcher](#) pattern in information architectures. make 4Cs of the solver such that they allow “to break” the event loop after every iteration (or a coherent part of it), and schedule progress monitoring for the solver.)

Caching and memoization in solvers

(TODO: explain trade-off between redoing computations and storing intermediate computational results: [memoization](#).)

11.6 Best and bad practices

This Section discusses the [composability](#) and [compositionality](#) aspects of commonly used architectural choices.

11.6.1 Best practices

- the [single-writer principle](#) helps to make (i) [atomic mutation semantics](#) easier to realise, and (ii) more efficient to execute, by reducing [cache misses](#). Indeed, a data item is owned by a single [execution context](#) for all [mutations](#).
- [lock-free](#) and [wait-free](#) algorithms for data [sharing](#) [4] help *to avoid* the involvement of the operating system, such that applications have a larger impact on their own behaviour.
- [garbage collection](#): the decision making about what to do with a full or an empty stream buffer is best done in a [mediator activity](#) that is owned by the ring buffer, instead of by producer and consumers themselves; for the simple reason that only then ownership of the data is unambiguous. For example, in tracing, audio processing or control problems, the [common policy](#) is to overwrite *part of* the oldest data at the producer’s side with newly arrived data, even when the consumer has not yet freed up the buffer array part that it occupies. Other policies are:
 - *discard*: drop the newly available data chunks.
 - *compaction*: the data in the buffer that has been produced but not yet consumed (or rather, not yet *claimed* by a consumer) is reduced, according to an application-specific policy rule.
 - *negotiation*: the amount of data per chunk is reduced, so more data can fit in the same stream. For example, [WebRTC](#) follows this approach.
 - *clear* the whole buffer, because the *completeness* of the stream is not guaranteed anymore.
 - *block* the activity until the buffer is not full or empty anymore.

All policies can be composed. This use case can also be dealt with without giving the stream buffer its own first-class activity: the producer is owns its side of the buffer, so it can *execute* the mediation actions itself.

- *Safety PLC* pattern:

(TODO: separate component (function, activity, thread, computer,...) that checks a number of flags, heartbeats, time outs,..., at higher speed and lower latency than the real-time control component(s), and that can shut down the application independently. For example, by cutting power to the energy-introducing parts; and maybe then “rebooting” the failed control components.)

11.6.2 Composable configuration in compilation, linking, launching and execution

This is the natural, strict order in which the software of a system is composed together:

- *compilation*: the programmers software is composed with (pointers to) the functionalities in *standard libraries*, resulting in a binary code. Typically, this binary code is not an *executable* process in itself, yet.
- *linking*: the compiled code of the application programme is composed with the binary code of the “standard libraries”. The resulting binary code is, either an executable process in itself, or a *module that can be loaded* at runtime in an already executing process.
- *launching*: the operating system is given the executable process for execution.
- *execution*: while executing, the process can (un)load executable modules itself.
- *hot-swapping*: the ultimate form of runtime (un)loading of executable modules. The term is the software version of what is known as *hot swapping* for hardware components.

Once factories are in place in a system, it becomes *simple* (though not *easy*) to adapt any configuration realised already at compilation time, because the provide the required infrastructure and mechanisms.

11.6.3 Framework plug-ins (bad) versus library composition (good)

Frameworks are one of the most popular ways towards digital platforms, for some good reasons: *code reuse*, *freedom from choice*, *best practice implementations*, and *tooling*. However, frameworks inevitably make trade-offs towards *usability* within a particular domain, *erroneously assuming* that “*easy*” will imply “*simple*”.

Composability in a framework is typically provided via so-called *plug-in* interfaces: the framework provides several places in its code base where developers can *register* their own functions, that will later be called by the framework’s *runtime engine* at the “right” time. This *mediator pattern* implementation works fine, as long as the framework can provide its users with all services they need. However, frameworks are typically poorly composable themselves with other frameworks or components, because:

- their plug-in interface offers only one single level of composition hierarchy. It is then not possible for developers to introduce their own functions to couple “state” at two different plug-in interfaces.
- a framework’s runtime engine typically expects that plug-ins adapt to the framework’s policies and protocols, and not the other way around. (In other words, frameworks often

consider themselves as an “endpoint of integration”.) So, it is for example not possible to configure the runtime engine to share resources with non-framework activities.

- one major cause is that their runtime’s *event loop* is not a user-accessible implementation primitive. Instead, they opt for major “easy” hard-coded policies, such as:
 - dedicating one whole I/O channel to each *port* instead of multiplexing several channels on one single port.
 - no access to the internal multi-threading architecture. (*If* there is one, to start with.)
 - lock-in into one single programming language, forcing single-language, compile time only, composition.
 - the lack of explicit support for user-accessible mediation and ownership.
- the configuration of computations is forced to be done *in* a plugged-in computation, instead of in the component for which the plug-in provides functionalities.
- similarly for the coordination between several plug-ins: the framework forces each plug-in to make the decision that its behaviour has to change, or let the framework do this. This prevents the framework-independent behavioural coordination *between* plug-ins.
- introducing “easy” coordination instruments to let a plug-in choose its own *priority*, while such a priority is *not a property* of a component, but an *attribute* given by the *process* that composes several components.

The more composable approach is to replace the runtime engine part of the framework, and *generate* an application-specific one, by (i) composition of functions and data from pure libraries, and (ii) generating the code for the application’s runtime, starting from composable implementations of software architecture patterns. Examples of the latter are the event loops and the inter-process communication patterns, for which framework runtimes often have made static, immutable choices.

11.6.4 Bad practices in robustness

The challenge of the software architects is to balance the *software* and *hardware performance* (which is a loosely defined composition of *efficiency*, *efficacy*, and *effectiveness*), with the **productivity** of the *developers* (which is loosely defined as realising a predictable system quality with predictable effort). One major performance challenge is to implement conceptually perfect information processing capabilities with imperfect resources available on computers. Here are some examples of system designs that turn out to be bad practice, because they neglect particular disturbances to performance and predictability:

- the *infinite data extension* of “time series” data streams, while a computer has only finite and/or non-*shared memory* resources.
- the *infinitely fast interaction* between activities, while a computer has only finite resources for computation and communication.
- the *perfect abstraction of an activity*, while computer-implemented architectures must deal with the large *cost vs performance* variations presented by the building blocks offered by *operating system* and *hardware*.
- the *perfect semantic consistency* of information exchange, while *programming languages*, *libraries* and *frameworks* often have different semantic interpretations of data structures and functions.
- *resource contention*, *concurrency*, and *end-to-end latencies* caused by multi-core/multi-computer execution contexts.

- let threads decide themselves about their **priority** and **processor affinity**.

These decisions must rather be made at a system level where the trade-off between *several* threads can be made, in the context of the *full* application.

- let the “event loop” be realized by the operating system.

That operating system does not know anything about the application’s performance indicators, so it should not be asked to schedule the application’s threads. Instead, that responsibility is to be taken in the application’s event loop(s): only there, the right decisions can be made about when activities can be pre-empted, and by which other activities.

11.6.5 Bad practices in resource locking

The sample code in Table 11.1 combines three bad practices of using locks:

- *multiple locks around the same critical section*: (TODO)
- *nesting locks*: (TODO)
- *blocking operations inside lock*: (TODO)

```
struct { int a; int b; } dataA;
struct { int a; int b; } dataB;
...
mutex_lock(mA);
mutex_lock(mB);
f1(&dataA);
f2(&dataB);
printf("A: %d, B: %d \n", dataA.a, dataB.b);
mutex_unlock(mA);
mutex_lunock(mB);
```

Table 11.1: Code example combining three bad practices in using locks.

11.6.6 Bad practices in communication

- to use only the **publish-subscribe** communication pattern. Events often profit from a *broadcasting* policy, and queries about models require one-on-one *dialogues* via **submission-completion** queues. Neither of those use cases is supported well with publish-subscribe.
 - to neglect the **CAP** and **PACELC** theorems, that state that:
 - the assumptions of Consistency, Availability, and Partition tolerance can not be satisfied at the same time.
 - Consistency and Latency are contradicting requirements.
- These theorems, and the **exactly-once semantics** of communication, are major *constraints* for realising consistency of data exchanged between asynchronous activities. Or rather, once system designers are aware of the difficulty to realise all aspects involved, they will look for architectures that are **robust** against (combinations of) communication disturbances.
- to neglect the **fallacies of distributed computing**, *even* between processes on the same computer.

- to communicate *state* information back and forth between distributed components, even when it is possible to deploy all the components' functionalities into the same process and (hence) to store the shared state in a shared data structure.

11.7 Mechanisms: close to the hardware

11.7.1 Mechanism: two-way mapping between symbolic IDs and memory addresses

(TODO: pointer is for binary (hence, fast) indirection in compiled code and data; scripts are for one-way symbolic-to-binary conversions at runtime via [interpretation](#) and [bytecode](#); URL/IRI is for symbolic (hence, slow) indirection between models. This Section explains how both can be integrated, with the following structure:

- the indirection is *two-ways*.
- the mapping is done via *querying* the services of a *third party*, a so-called [broker](#).

Hence, models must be available and editable at runtime. This is an extra cost at development time, but only a [one-off](#) cost. So, once the models are available at runtime, many interesting things become available. Also, many of the [bells and whistles](#) that differentiate “advanced” programming languages from C “just” become model-to-model transformations that can be shifted from compile time to runtime; for example: templates, event handling (including [bubbling](#) or [capturing](#) in a containment hierarchy), closures,...)

11.7.2 Mechanism: memory barriers — Asynchronicity in compilers & CPUs

Activities are a major building block in system architectures, because they connect the *asynchronous* world of an application with the [synchronous](#) context of the algorithms that realise the application behaviour. However, modern compilers and CPUs put significant pressure on the conceptual model of synchronous programming, because both try **to optimize** the amount of computations in two ways:

- changing the order of the execution of programming language statements.
- removing “unnecessary” code.

The consequence of these facts is that software programmers must not only tackle the correct coordination within their own software components, but also that coordination with the behaviour of [compiled binary code](#) and of CPUs. More in particular, writing to (the different levels of [cache](#)) memory is [not an instantaneous operation](#) on modern CPUs, but has more of the properties of “message passing” (Fig. 11.14): the CPU “sends” a message¹⁰ to the cache that a certain value is to be written to memory, eventually. Right after having informed the memory about this, the CPU can continue with something else. The overall result is that the CPU can rearrange the order of execution of the statements in the programmers’ code. Hence, programmers have to add extra coordination complexity in their code, comparable to what is needed for “normal” concurrent programming between activities, to make sure that the compiler or the CPU execute *all of the needed code in the right order*.

One mechanism is the [memory barrier](#) (or *fence*) instruction: it forces all memory operations (read (“[fetch](#)”) and write (“[store](#)”)) that appear *before* it in the software code, to

¹⁰More correctly, the *value-to-be-written* is “produced” into some sort of circular buffer, that the cache then empties as a “consumer” to do the actual store in memory.

complete before any memory operation *after* it can begin. (That completion is a costly operation, because it can involve several levels of [cache memory](#).) In other words, by introducing an explicit barrier operation in their code, software programmers can rely on [sequential execution ordering guarantees](#) by the CPU, at the cost of loosing some execution efficiency.

Memory barriers typically come in *pairs*: an acquire barrier is paired with a release barrier; the same applies to load (read) and store (write) barriers:¹¹

Barrier	C11/C17	Linux kernel
READ	not available	<code>smp_rmb()</code>
WRITE	not available	<code>smp_wmb()</code>
FULL	<code>memory_order_seq_cst</code>	<code>smp_mb()</code>
ACQUIRE	<code>memory_order_acquire</code>	<code>smp_load_acquire()</code>
RELEASE	<code>memory_order_release</code>	<code>smp_store_release()</code>

Another mechanism¹² provided by many CPU architectures is the [compare-and-swap](#): operation: it compares the contents of a memory location with a given value and, only if they are the same, modifies the contents of that memory location to a new given value. This is done as a single atomic operation. The atomicity guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another thread in the meantime, the write would fail.

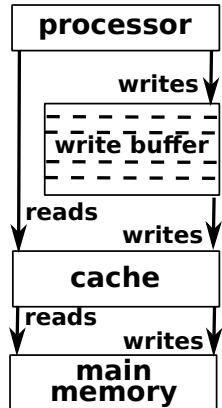


Figure 11.14: The asymmetry between reading and writing to memory. Write operations are not necessarily executed instantaneously, and not necessarily in the same order as programmed in the software.

(TODO: more explanation and examples.)

11.7.3 Atomic and lockfree operators

Replace every synchronous “mutex” area with an asynchronous stream, or with [seqlocks](#).

[Memory barrier](#) operations need to be inserted,

The problem can also be solved by introducing a small ring buffer with lock+value structures, because then the write/read of the value *depends* on the sequence number and compilers will not [reorder instructions](#).

A single-writer stream can be done without locks; multi-writer is seldom needed because it can be replaced by the same consumer for all of the producers in a single writer-reader form, and that consumer makes one or more new composed stream.

¹¹“`smp`” stands for [symmetric multi-processing](#). “`mb`” stands for *memory barrier*. “`r`” stands for *read*, “`w`” stands for *write*.

¹²Often realised conceptually as a composition of the above-mentioned first mechanism.

[Read-Copy-Update](#) is another approach that trades off locking for more copies of written data.

11.7.4 Conflict-Free Replicated Data Type (CRDT)

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

11.7.5 Immutable data type

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

11.7.6 Bitfields for flags, FSMs and Petri Nets

Earlier Sections introduce the information-level coordination primitives of [status flags](#), [Petri nets](#) and [state machines](#). This Section explains their *implementation* by means of [bit fields](#).

Bit fields

A [bit field](#) is a very efficient data structure

- **to store** a rather limited number of *flags* or [enumerations](#), in [unsigned integers](#). Hence, the number of flags that can be represented is limited by the number of bits in the integer; typical sizes are 8, 16, 32 or 64 bits.
- **to process** event logic by means of [masking](#) and [bitwise operations](#).

Hence, the computer representation of the states, the transitions and the events, can happen in the following way:

- the **states** are encoded in a bit field.
- the state **transitions** are encoded in a bit field.
- the state–transition–state connections are encoded in composite data structure with three bit fields:
 - one bit field represents the “start” state.
 - one bit field represents the **transition**.
 - one bit field represents the “end” state.
- the **events** are encoded in a bit field.
- the **event-to-transition** table connected to one particular transition is a composite data structure, with two fields:
 - one bit field represents the **transition**.
 - one bit field represents all **events** that trigger that **transition**.
- the **transition-to-event** table connected to one particular **transition** is a composite data structure, with two fields:
 - one bit field represents the **transition**.
 - one bit field represents all **events** that are triggered by that **transition**.

With these data structures, the event processing runs as follows:

- *firing* of events happens by adding the event bits to a bit field, and *pushing* it onto a **stream**.
- *processing* of events happens by *popping* an event bit field from a stream, using bit operators to get the individual events out, and reacting to the events in a **switch statement**.

Dictionary of linked lists

(TODO: representation in **associative array** to implement **symbolic indirection** between external and internal “names” of flags; advantage of runtime adaptability, at the cost of runtime memory allocation.)

Events or flags?

A status flag for a finite state machine can make *less* state information observable than there is available internally, and with different symbolic tags; for example, it is possible that the status flag for a **Life Cycle State Machine** shows only the values **inactive** and **active**, where the former holds the FSM is in any of the sub-states of the **deploying** super-state, and the latter holds for all the sub-states of the **active** super-state.

Events are connected to changes in state and status flags, as, both, “sources” and “sinks”: a state/status change can give rise to the firing of an event, and the processing of an event can cause a state/status change.

The abstract data type of all of the above-mentioned concepts can be **enums** (“**enumerated values**”) that is, they can take one of a finite number of possible symbolic names.

Events are most appropriate in the **Coordination** and **Orchestration** coordination modes, and status flags for the **Orchestration** and **Choreography** coordination modes. The overlap in the Orchestrated mode is as follows:

- events for the coordination of activities *not* in the same process address space, because their state must be *communicated*.
- status flags for the coordination of activities *in* the same process address space, because their state can be *observed*.

11.7.7 Buffer, queue, socket

The unique identifier of a data chunk in a stream is a *symbolic name* of the data chunk, or, equivalently, a *symbolic pointer* to the data chunk. There are several complementary (and not mutually exclusive) ways the “pointer” mechanism is being used to realise *semantic classes* of activity interactions:

- (shared memory) **buffer**: both activities share the same memory space, and can use symbolic pointers to the data chunks in the buffer.

The *advantage* of the shared memory mechanism is that the data need not be copied, and all of the available information is accessible, directly and repeatedly.

The *disadvantage* is that there is no clear ownership of the data, hence one must add *data access constraints* to all activities that share the information. These constraints must be followed by all functions that use the information, so they interleave their execution in a predictable, consistent, and resilient way; the result is equivalent to the introduction of a stream:

- the stream imposes a strict order, to be followed by the interleaving.
- each data chunk contains the symbolic pointer that is available to the owner of the data chunk.

Most often, these constraints are to be satisfied *by discipline* of the programmers, instead of being guaranteed by construction, in programming languages or software tools.

- (message) **queue**: this mechanism works between two activities in the same address space or not, and both rely on the services of the operating system. That acts as a **mediator**, by taking care of (i) copying the data to and from the streams in the address space of both activities, via **write** and **read** operations, and (ii) deciding which data to forward from a producer to a consumer.

A special case of message queues are *event queues*: one activity can have to react to events from one or more other activities, so a stream buffer is a good solution for each “event queue” between two activities, and the data chunk for an event typically consist of nothing more than a symbolic tag.

The *advantage* of a message queue is that the developers of the reading and writing activities need not bother about the data access constraint: the data must be *copied* anyway, to be sent over the “wire”.

The *disadvantage* is that the execution of each activity has some side effects: the data comes only available one by one, in the strict *first in, first out* order of the **queue**; higher memory usage because of the data copying; and non-deterministic pre-emption of their execution, because the “**locking**” of the buffer pointers can happen implicitly behind the screens.

- **socket**: when activities do not share directly accessible memory,¹³ the message queue mechanism is extended further: the “local” message queue is not really one single queue, but the data that is written to it is sent to another computer or process via a communication channel, where it is copied to the local message queue of the program there. The interfacing activities can still use the same **send** and **receive** operators.

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

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

The operating system on a computer is the natural place to embed all of the above-mentioned mechanisms, exactly *because* it is “just” mechanism that applications can and should compose with their specific policies. Major aims for “mechanism” developments at the platform level is to strive for (i) standardization, (ii) performance, (iii) resilience, and (iv) security. Realising

¹³That already is the case on one single computer, where **processes** have separated memory access, realised by **hardware**.

these competing goals together requires a huge effort, which is another reason to share this effort by all stakeholders of the platform. Examples illustrating this context in the case of operating systems are the inter-process communication projects [D-bus](#) (between processes on the same computer) and [WebRTC](#) (between processes on different computers). In robotics, the [ROS](#) project tries to achieve the same role, but it has yet reached industry-grade maturity in any of the four design goals mentioned above; it also conforms to almost none of this document’s best practices and design patterns.

11.7.8 Async/await

(TODO: [async/await](#) programming language construct to make some forms of asynchronous programming look very much like synchronous programming; discussion on where it fits in the meta models of activities, event loops, and streams. explain problem with cancelling of a function when it is already waiting on a promise, and refer to Submission-Completion as a *task queue* mechanism with a better support for task cancelling; discussion on when to use it and when not; examples needed, for example in graph traversal solvers.)

11.7.9 Iterator

(TODO: [iterator](#) helps to separate the structure of the data from the operations, in a declarative way.)

11.7.10 Maybe

(TODO: [Maybe](#) and [three-valued logic](#) (or “trilean”) type, to represent “not-yet-known” data, or “nonsense”, or “invalid”, or “both true and false”.)

Chapter 12

Software architectures for systems

This Chapter focuses on the **system level** aspects of software. In other words, **computer science** and **software engineering** **in the large**. The extra design decisions to be made in addition to the **component level**, are connected to the differences between **efficient *in-process* communication**, and **inter-process communication**:

- *distributed hardware*: computer, edge, cloud.
- *ownership*:
- *latency*:
- *delivery guarantees*:
- *discovery*:
- *session creation*:

12.1 Architectures for activity deployment

Robotic and cyber-physical systems can contain hundreds or thousands of software components, in the form of algorithms, activities, and data exchange mechanisms. In order to bring **structure** in this abundance of components, system designers must search for **hierarchies**, wherever possible. Some of those hierarchies have already been introduced in the **system** and **information** architectures. This Section adds some more hierarchical structures, introduced naturally by the *resource* components of **computer hardware** and **operating system**.

12.1.1 Hardware hierarchy: core, system-on-a-chip, computer, edge, cloud

Computer hardware provides hard constraints for a software architecture to comply to, caused by the following parts: CPU, memory, bus, I/O, and network. The CPU part comes in some variants on most modern hardware, depending on the degree of sharing memory (“**caches**” and “**RAM**”) and communication hardware. The underlying hierarchy is clear and absolute, hence appropriate to depend on in the design of the system architecture:

- **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**: composes several cores and peripheral hardware on one single integrated circuit, and coordinates their access to hardware shared communication buses.
- **computer**: a composition of several of the above computational mechanisms, integrated with communication, memory, IO and file storage functionalities.
- **edge system**: a composition of computers that are under the management of one single system owner, and are (hence) deployed together, sharing an owner-specific network in an owner-specific location, with the **network-connected storage**, **authentication** and **network security** protocols that can be enforced by the owner.
- **cloud**: composition of several of the above, interconnected by the general purpose Internet instead of by a dedicated network, and with the extra complexity of having a system with multiple owners and stakeholders.
- **fog**: in large-scale industries, a “*nearby cloud*”, or “fog” system, is introduced to bridge the gap between *edge* and *cloud*.

12.1.2 Software hierarchy: runtime, thread, process, shell, container, cloud

A similar hierarchical structure like the one introduced by the **hardware** resources, comes with the *resources* provided by **operating systems** and **compilers**. The structure facilitates (i) **abstraction** and **resource sharing** of the above-mentioned computer hardware, and (ii) **software portability**. The following paragraphs describe the hierarchy in a “top-down” order, that is, starting with the software entity that has the most contextual information about all software dependencies in the executed application:

- **runtime**: the “infrastructure” code that a compiler adds to the source code of an application developer, to facilitate generic functionalities, e.g., event handling. That means that there is an (most often implicit) **DOM model for event bubbling**. There is one runtime per compiled binary, and the compiler has configured the sharing of the runtime objects that can be accessed from functions that have been *compiled* together.
(TODO: explain role of WebAssembly standard, to make “runtimes” also composable and configurable during execution; **WASM** implementations.)
- **thread**: the **thread**¹ is the operating system mechanism that couples the functions in the compiled software code of an application to a CPU on the computer hardware. The thread **owns** very little: just two, but essential, interfaces between operating system and application, that allow activities **to share** the same **synchronous execution context**:
 - **thread ID** (OS centered): the unique identifier via which the *operating system* can **schedule and dispatch** the execution of the thread on a CPU.
 - **function pointer** (application centered): the address of the **event loop** function that composes algorithms in various activities into one single synchronous computational context. When access to various resources provided by the operating system must be coordinated, or when the execution of several tasks has to be coordinated, the event loop function is pointing to a **resource mediator** activity in the *application*.

Both identifiers are **attributes** of the thread and not properties: their values are assigned by, respectively, (i) the operating system when *creating* the thread, and (ii) the

¹More detailed introductions can be found [here](#) and [here](#).

application when *configuring* the deployment of an activity in the thread. The latter also sets other relevant attributes to configure the behaviour of a thread:

- conditions under which the thread wants to be scheduled, with *timing* and *scheduling priority* being major ones.
- constraints on *swapping* of the thread’s RAM that the operating system should respect.

The thread is also the primitive to host the data connected to the **runtime** (re)configuration² and *introspection* of the activities that it executes.

Two “light” versions of the thread concept are:

- A **fiber**: the *operating system* uses *cooperative multitasking* to determine when to execute which fiber, while threads are scheduled via *preemptive multitasking*.
- **coroutine**: provides 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 operating resources required by all threads. That is, the process **owns** the memory spaces needed for:
 - the **stacks** of the threads deployed inside the process.
 - the **static** and **heap RAM** memory needed by the activities executed in the threads.
 - the **handles** (or, **file descriptors**) of all I/O resources, that is, handles to the **file system** and the **inter-process communication** channels needed by the activities executed in the threads.
 - the coordination of how threads are sharing the **CPU cores** and **virtual memory space**.
 - **compile time** configuration in **global variables** of the virtual address space shared by threads.

In other words, the process is the composition primitive that makes threads **share the same process context**. One process can start up other processes, via system calls to the operating system; this functionality and its ownership and access context is the same as for the *shell* below. In other words, processes can be composed in a **DOM model** of indefinite depth.

- **shell**: operating systems provide the shell primitive as a shared **context** for several processes to be active in. The same process *binary code* can be started from within different shells, but then gets another set of **environment variables** that the process uses to configure the behaviour of that particular *process instance*. In other words, the shell is the composition primitive that makes processes **share the same operating system context**.

It is also the primitive to configure the **deployment** of various processes in one application: in which order, and with which context, to launch all the processes needed in that application.

- **container**: has a similar **composition** role as processes for threads, but now between operating systems and the computational hardware (which continues below in this hierarchy list).

In other words, the container is the composition primitive that makes shells **share the same operating system instance**.

- **kernel**: this is *the singleton* primitive responsible for the coordination of all processes (or containers) in their access to the physical resources on one computing platform.

²Some compilers offer support for runtime configuration composition, via the concept of **thread-local storage**.

Process, container and kernel can conceptually be considered as hierarchical versions of the same type of resource coordination primitives. All of them can be **configured** by an application, at runtime via shell scripts, or at compile time via **kernel configuration** settings.

The *information* architects provide an architecture of activities and algorithms, and the *software* architects must then make sure that each activity executes correctly in a thread somewhere on a processor, irrespective of the timing and the number of times that execution is **preempted** by the operating system.

12.2 Systems with 5Cs architecture

The **5Cs pattern** is the core architectural pattern in this document, because it scales well from the most **deeply embedded** applications (e.g., standalone robots with (almost) no remote interactions with other systems), to the scale of the Internet.

12.2.1 Single 5Cs process: activities with shared memory interactions

Figure 9.2 shows the **mereology** of the **smallest executable system** possible, that is, one **component** (that is, an activity designed along the **4C** pattern) that is in itself a set of **sub-components** (each individually also designed along the **4C** pattern) deployed in **one single process** and interacting with each other via **shared memory**:

- each activity is designed with a **5Cs architecture**, that is, it is a **component** in itself, featuring algorithms, state machines, Petri nets, protocol flags, interaction channels, etc.
- there is *one and only one specific* activity, the *mediator component*, responsible for initializing all activities and their interactions, in such a way that, towards the “outside world”, **the whole process behaves as one single component in itself**.
- data structures for three complementary purposes:
 - *shared memory* to encode information about the **world model**. That is the data that some activities read and others write, in order for all of them to have access to the data they need at the moment they need it.
 - **events**: the *specific* data structures meant to encode that *things have happened* somewhere, sometime, because that information is needed to coordinate behaviours in different activities.
 - **protocol flags**: the *specific* data structures meant to encode the coordination between one single activity and one single mediator.

Figure 9.2 is a mereological model, so **not a software architecture**.³

- there will be no data structures with the names **event queue**, or **shared memory**. Instead, there will be application-specific names for data structures that *play the role of* event handling and data exchange.
- the *behaviour* of how all activities access these data structures is still to be decided, in the application’s **information architecture**.

³See Sec. 11.1 for an extention of this mereological model with concrete software choices.

12.2.2 Single 5Cs process with multiple-rate activity interactions

Many industrial devices⁴ are standalone systems in themselves, for which the integration approach is to foresee the following “dual speed architecture”:

- **online (re)configuration:** in this mode, the realtime data exchange communication is configured, for example, selecting the type of data exchange, or the exchange rates and maximally accepted latencies.

This mode fits naturally in the **Life Cycle State Machine’s configuring capabilities** state of the activity that interacts with the device.

- **realtime data exchange:** in this mode, the data exchange takes place, as configured, with realtime performance guarantees on the communication between the activity that interacts with the device and the device “controller”.

This mode fits naturally in the LCSM’s **running** state of the activity that must control the device.

Both interaction modes sometimes make use of different communication hardware and software protocols; for example, **TCP/IP** for configuration, and **UDP**, **CAN** or **EtherCat** for the realtime data exchange. The main difference with the *shared memory* architecture of Sec. 12.2.1 is the *performance* that can be expected from the data exchange: using internet or field busses is always orders of magnitude less powerful than shared memory, with respect to latency as well as to bandwidth. That has an impact on the *quality mobitors* in the application, whose tolerances will have to be configured to “worse” values.

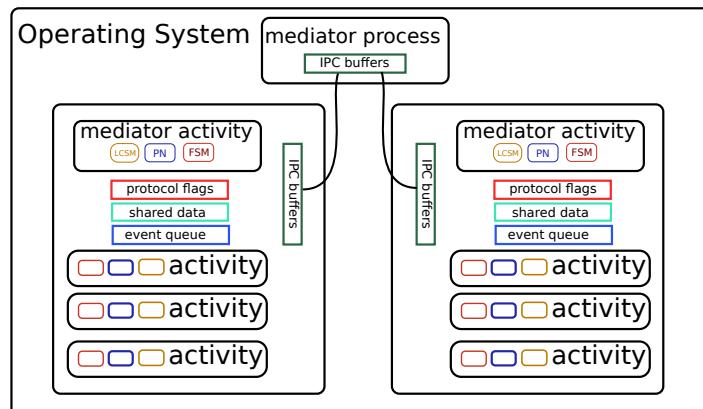


Figure 12.1: Architectural pattern for an application running in multiple processes on one single computer device, or “CPU”.

12.2.3 Multiple 5Cs processes with IPC interactions on one CPU

Figure 12.1 shows the architecture of an application that is built with *multiple processes* on one single computer device, or “CPU”. The differences with the *single-process* architecture are:

- events, flags and world model data can not be shared directly in the same memory space, but must be **communicated** between processes, via one form or another of

⁴For example, most so-called **cobots**.

[inter-process communication](#) (IPC) channels. That communication requires⁵ *copying* of data from and to the communication channel, hence extra care must be given to keep [data consistent](#) inside the whole application.

- the IPC's [system calls](#) make it inevitable that the [operating system](#) (OS) becomes part of the system design.

12.2.4 Device: multiple 5Cs processes on multiple CPUs

The single and multiple process architectures on one CPU of the previous Sections have their limits in performance, because one can not expect the same core and RAM hardware to cope with a growing number of activities that are deployed on it without performance losses. At a certain amount of load on the system, its performance will degrade under the level that can be expected from a distributed architecture. Finding the “soft spot” of when and why to switch from one CPU to multiple CPUs is part of the art of good system design, and no universally valid best practices exist to help the system architects.

From an *architectural* point of view, the difference between systems with or without distributed CPUs is minor. Only the *type* of IPC communication is different, which comes with *performance* differences, and hence probably the need to add (i) extra monitor activities for those new performance measures, and (ii) extra coordination protocols to deal with consequences of new performance loss cases.

12.2.5 Edge: multiple devices with application-controlled cooperation

Figure 12.2 shows the [edge](#) version of the architectural pattern. The difference with the multi-process architecture is that data has to be copied even more: the inter-computer networking requires the introduction of a [network stack](#).

In the [edge](#) architecture, that network is [dedicated to](#), and often also [owned by](#), the application, and so its performance can be configured to suit the application best.

12.2.6 Cloud: multiple applications with server-controlled cooperation

One or more “[edge](#)” systems are connected to a “[cloud](#)”, for shared [backend services](#), such as data storage and replication.

In a cloud architecture, the network is typically the common [Internet](#), so few to no performance optimizations are possible. This increases the needs to foresee fargoing robustness against communication latencies and data losses.

12.3 Modelling languages: mature implementations & tools

The meta and meta meta models of all previous Chapters must, sooner or later, be encoded in concrete “programming” languages. For some of the representations, some such concrete encoding languages have matured into world-wide, vendor-independent standards. Most formalizations have been developed for human developers only, and not for [higher-order reasoning](#) by computers. This Section gives some domain and vendor neutral examples, for which mature implementations and tool support exist.

⁵Some operating systems allow to set up shared memory between different processes, via system calls. The setup will take longer than just being able to share a data buffer, but performance may often be comparable.

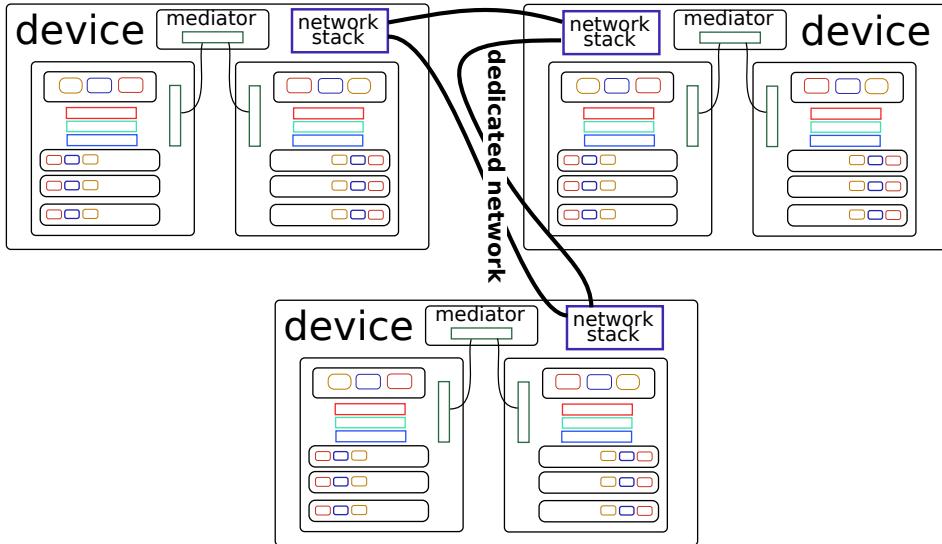


Figure 12.2: Architectural pattern for coordinating an application distributed over different devices on the same dedicated local communication network.

12.3.1 QUDT and UCUM

The *quantity-unit-dimension-type* meta model has already been formalized several times, for example in the **QUDT** initiative [161]. 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**(uality) and the (physical)**U**(nit) parts of QUDT.
- **T** and **D** always come together at the same level of abstraction, as do **Q** and **U**.

The **W3C** and the **Eclipse** eco-system promote a model similar to QUDT: **UCUM** (the *Unified Code for Units of Measure*).

12.3.2 JSON and JSON-Schema

JSON-Schema is a schema to formally describe elements and constraints over a **JavaScript Object Notation** (JSON) document. Instead of relying on an external DSL, a JSON-Schema is also defined as a JSON document. In turn, the JSON-schema must conform to a meta-schema, which is also defined over a JSON document. A concrete example is provided in Figure 12.3.

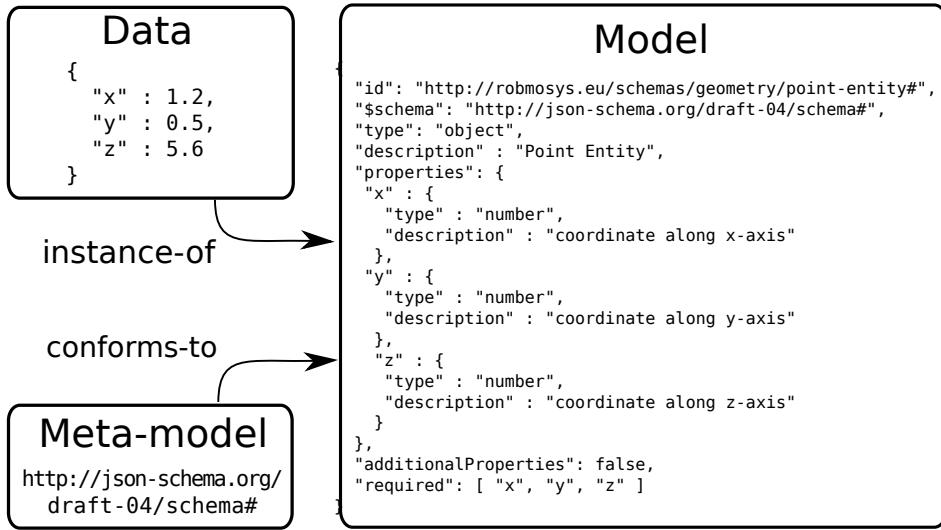


Figure 12.3: A valid data instance of a JSON-Schema Model representing three coordinates. The schema includes few constraints on the data structure, such as the values required for the validation of the JSON document. Moreover, the schema **conforms-to** a specific meta-model of JSON-Schema (draft-04).

JSON-Schema is considered a composable approach, since (i) JSON supports associative array (only strings are accepted as keys) and (ii) JSON-Schema supports JSON Pointers (RFC 6901) to reference (part of) other JSON documents, but also objects within the document itself. This allows to compose a schema specification from existing ones, and to refer only to some specific definitions. JSON-Schema is used in web-technologies and it is very flexible in terms of requirements needed to be integrated in an application. However, it is verbose with respect to other alternatives, as well as not efficient in terms of number of bytes exchanged with respect to the informative content of a message. In fact, JSON-Schema does not provide native primitives to specify hardware-specific encodings of the data values. However, it is possible to compose a schema that cover that roles, in case that the backend component can deal with them. Figure 12.3 shows a example of a typical workflow with JSON-Schema. As a final remark, JSON-Schema is not limited to describe JSON documents, but also language-dependent datatypes.

12.3.3 JSON-LD

[JSON-LD](#), “*JSON for Linked Data*”

(TODO: outline of features; how to use it to represent property graphs, `Semantic_ID` (`@context` for meta model *connections*, `@type` for meta model *conforms-to* relations, and `@id` for model ID; how to formulate queries as graphs on the property graph, with *given* and *queried* property parameter values; how to do graph traversals to answer queries.)

12.3.4 RDF1.1

[RDF](#):

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

12.3.5 Abstract Syntax Notation One (ASN.1)

[ASN.1](#) (“Abstract Syntax Notation One” is an IDL to define data structures in a standard, code-agnostic form, enabling the expressivity required to realise efficient cross-platform serialisation and deserialisation. It has its origins in telecommunication, in the early 1980s. ASN.1 models can be collected into “modules”, which can be composed between themselves as well. This feature of the ASN.1 language allows better composability and re-usability of existing models. However, ASN.1 does not provide any facility of self-description, if not by means of the naming schema used by the compiler to generate a data type in the target programming language. Originally developed in 1984 and standardised in 1990, ASN.1 is widely adopted in telecommunication domain, including in encryption protocols, e.g., in the HTTPS certificates (X.509 protocol), VoIP services and more. Moreover, ASN.1 is also used in the aerospace domain for safe-critical applications, including robotics applications. For example, an ASN.1 compiler is included in *The ASSERT Set of Tools for Engineering* (TASTE), a component-based framework developed by the European Space Agency (ESA). Several compilers exists, targeting to different host programming languages, including C/C++, Python and Ada.

12.3.6 Hierarchical Data Format — HDF5

[HDF5](#) is another internationally standardized format, with a maturity similar to QUDT, offering meta model, models and reference implementations for all sorts of *abstract data type* representations and transformations.

12.3.7 FlatBuffers, Protocol Buffers, Apache Arrow

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

A promising recent development in the direction of improving these deficiencies is the [Apache Arrow](#) project; it has formal schemas for all binary data structures, pays very special attention to support efficient random access to all binary data, and has tooling to convert the former automatically to code that realises the latter.

12.3.8 Common Trace Format — CTF

The [Common Trace Format](#) is a binary trace format, that originates from the Linux kernel, in the context of *event streaming* and *processing*.

12.3.9 BLAS, LAPACK

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

12.3.10 DFDL

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

12.3.11 XML Schemas

Similarly to JSON-Schema, XML Schemas (e.g., XSD) are models that formally describe the structure of a Extensible Markup Language (XML) document. XML schemas are very popular in web-oriented application and ontology description, but also in tooling and hardware configurations (e.g., the *EtherCAT XML Device Description*).

12.3.12 PROV-O provenance ontology

(TODO: [W3C’s PROV-O: The PROV Ontology.](#))

12.3.13 Functional Mockup Interface (FMI)

[FMI](#) is an industry-driven standard for the representation and the [co-simulation](#) of dynamical systems. The host language is XML. The standard tries to be self-contained, which means it repeats a lot of more generic concepts and standards, such as the representation of physical units and types (covered by QUDT), or the digital representation of values (8, 16, 32 or 64 bit unsigned integers, etc., as covered by programming languages). It has fallen into two major traps of “bad practice” modelling, i.e., *inheritance instead of composition* and *properties instead of attributes*. This gives rise to some complementary negative effects:

- *reification* is not built in, hence there is no support to represent higher-order relations (constraints, tolerances,...) on parts of a model.
- a property of a [CoSimulation](#) type is, for example `canReturnEarlyAfterIntermediateUpdate`, but that is a behavioural property of an activity that executes a [CoSimulation](#), not of its model.
- software tools to support the standard must be huge, because the standard is huge.
- slow innovation, because of the many internal model dependencies. For example, the standard is not adapted in mainstream robotics systems, because the latter have a focus on *runtime* execution, and less on *co-simulation*. Although the overlap between both areas is huge, the co-simulation dependencies are very “hard-baked” into the standard, making it less appropriate for reuse in domains with another focus.

Chapter 13

Information and software architectures for robotic systems

This Chapter adds **robotics-specific** domain knowledge into the architectural patterns and best practices of **generic** information and software architectures. The *hardware* architectures of robot systems come in the three *form factors* (or, “mechatronic design patterns”) of **pod**, **flock** and **fleet**. The domain knowledge about the *software* for these robot devices is structured into the following **composition matrix** at different complementary “levels” the robots’ task models:

- world modelling, reflecting all of the real world that is needed for the robots.
- perception: proprio-ceptive, extero-ceptive and carto-ceptive.
- control: power, torque and acceleration, velocity, position, trajectory, metric waypoints and semantic waypoints.
- coordination: flags, finite state machines, and Petri nets.

In the *specific* context of robotic applications, the *generic* architectural design challenges of **composability** and **compositionality** are more commonly known as:

- **interoperability**: any combination of components from the above-mentioned component matrix must be useable as a **holon**, with highly (re)configurable structural and behavioural properties.
- **heterogeneity**: the interoperability must also hold for systems with holons consisting of **very different** compositions of motion stack levels, task specification levels, **vendor** implementations, etc.

These challenges are tackled, in the first place, by (i) the **5Cs** architectural pattern, and (ii) implementing the largest possible components that are *always* reused without ever having the need to break them open into smaller components.

13.1 Form factors for robotic systems — Pod, flock, fleet

The **three form factors**, with which the mechatronics hardware of the large majority of robotic systems is built, are:¹

¹The terminology is inspired by the Horizon 2020 project “ROPOD” (*Ultra-flat, ultra-flexible, and cost-effective robotic pods for handling legacy in logistics*), H2020-ICT-2016-1:731848.

- **pod**: the system consists of one *mechanically indivisible* machine. This can be as simple as a **mobile platform** with two identical actuated wheels, but also as complicated as a humanoid robot or a mobile manipulator, with dozens of motors with largely different power ratings. The mechanical indivisibility allows also the indivisibility of the computing hardware, so most often a pod’s activities are connected internally by means of **field buses**, they run on multi-core CPUS, and share RAM and disk memory. So, one can expect that (i) computations and communications are executed with *low-latency*, (ii) hardware failures in computation or communication are exceptional, and (iii) *if* they occur, they impact the whole system at once and permanently, and not just one single activity, sporadically.

The impact on the architecture is that one can suffice *to monitor* these resources, without having to introduce extensive software robustness against large **jitter** in performance.

- **flock**: the system consists of several *mechanically disconnected* pods, that work together on shared tasks, and in close physical proximity. For example, a number of robotic drones and mobile platforms, to help humans in **search and rescue** missions; or autonomous **tugboats** maneuvering **barges** carrying the large parts needed to assemble off-shore **wind farms**.

A flock implies separated computing hardware, and hence requires a (often wireless) communication **mesh network**. In such a network, pods have to discover each other, and set up cooperation connections (with lifetimes that can vary over many orders of magnitude), which they can decide to break up at any moment in time. Oftentimes, a flock must also cooperate with a **fleet controller**, acting as the **mediator** between (i) the task that the flock is responsible for, and (ii) the derived tasks for each of the pods in the flock.

So, one can expect that, within the flock, (i) computations are still executed with *low-latency* and fail exceptionally, (ii) communications can have largely varying performance, and (iii) *if* low-performance communication occurs, it impacts only some of the cooperating pods.

The impact on the architecture is that (i) one *has* to introduce software robustness against low communication performance, and (ii) one of the pods will act as the **edge** device,² responsible to represent the whole flock in the interaction with the fleet controller. Such an edge device can be put in the *infrastructure* (e.g., a traffic control tower for drones; a lock controller for autonomous boats; or a traffic light controller on a crossroad that is used by mobile robots), it can be one of the pods that takes up this extra role, or it can be provided by dedicated third-party service providers (e.g., companies specialised in dynamic maps and charts, or in ICT services).

- **fleet**: the “centralized” (or **cloud**) infrastructure to manage several flocks, and to store and process their large-scale and long-living data. Communication uses “the Internet”, as is, with close to zero guarantees about latency and connectivity. For example, the on-shore **control and command centre** for the above-mentioned wind farm assembly flocks.

So, one can expect large variety in the number, type and performance of communication connections with individual pods or with their “edge” pod.

The impact on the architecture is that one *has* to introduce software robustness against low communication performance (for example, data **replication** and **sharding**), as is

²Or **fog** device.

typical for a cloud infrastructure.

13.2 Pod architecture for single-process/single-device motion control

Motion [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. Many motion (or process)³ control applications fall in the category of applications in which one single process is appropriate, because the application is built around only one mechanically indivisible device. The key representatives of that category are systems that interface their hardware (sensors, actuators, digital I/O,...) by means of one or more field buses ([EtherCat](#), [CAN](#), [SPI](#), [I²C](#),...) connected to one single computer, embedded in the mechanical hardware. In an application-independent context, Fig. 9.2 showed which *types* of [building blocks](#) occur in the generic system architecture for that category. For one very simple but typical example, Fig. 11.1 shows the [specialisation](#) of the [generic architecture](#) of Fig. 9.2. The following paragraph describe the [components](#) in this architecture in somewhat more detail.

Main/Mediator activity

This activity's role (and hence its *algorithm*'s computation) is the *coordination* of all the other activities, via:

- the application's [Life Cycle State Machine](#) (LCSM) to bring the application's activities up and to shut them down;
- the [Finite State Machine](#) (FSM) to coordinate the system's task execution in the [running](#) state of the LCSM; and
- the [Petri Net](#) to help the the other activities inside the system to select their right *schedules* at the right times.

Part of the *mediator*'s LCSM responsibility is to create and configure all the shared data entities, before they are used by the application-centric activities. Often, a second algorithm in the mediator activity processes the keyboard input, and turns that input into task coordination events. Finally, the mediator activity also executes the [DOM](#) models that realise the configuration dependencies between all algorithms in all activities in the application.

Motion control

This functionality requires only *one* activity, because the sensor processing and control computations are simple algorithms, always executed in the same sequential order. In the [running](#) state of its LCSM, the activity *reads* IMU and encoder data from two [triple-state buffers](#), for use in its estimation and motion control computations, and *writes* the motion control setpoint to another buffer. Because the behaviours of these three algorithms are very closely intertwined, they share one [DOM](#) structure to coordinate their configurations.

³The term “process” is used in two different meanings: (i) [process](#) in computing, and (ii) [process](#) engineering on chemical, biological, material, thermodynamical systems.

I2C interface

This resource requires *one* activity, *to wait* for the data exchange with the I2C bus, and then *to transfer* the received IMU data to a [triple-state buffer](#).

CAN interface

This resource requires *two* activities:

- one to read and write from the fieldbus, being allowed *to wait* for the data exchange communicated via the fieldbus software stack.
- one to transfer the fieldbus data to and from the other activities, via [triple-state buffers](#).

Because the behaviours of these two CAN-related activities are very closely intertwined, they share one [DOM](#) structure to coordinate their configurations.

Coordination support

For all their mutual coordination, all activities share an [event queue](#). For its Petri Net coordination, the mediator activity shares one [protocol stack](#) with each of the other activities.

Data exchange

The **appropriate communication mechanisms** for all the data exchange in the system are [triple-state buffers](#) (or other forms of [ring buffers](#)), because, typically, (i) activities run at close but not exactly identical execution rates, and (ii) in control applications it is not a problem if not all produced data is also consumed, because the “state” of the application is “sent around” continuously.

Summary

A controller event loop (with a simple example depicted in Fig. 7.15) 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. 7.15 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. 7.15), 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”, because 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. 7.15): the natural **causality hierarchy** is to schedule the computations of an inner loop more frequently than those of an outer loop.

- many variables computed in control loops must be **monitored**, and these computations must be integrated in the “right way” into the scheduling of all other control loop computations.
- similarly, some monitors do not only generate *events* to trigger *discrete changes* in the control loop configuration, but also *adaptation* of some *continuous parameters* in the controllers, such as feedback gains or model parameters.
- last but not least, **real-time** requirements of the application have an impact on the model of the event loop.

13.2.1 Simplest pod architecture: proprio-ceptive control

The simplest instantiation of a single-pod robotic system has a motion control component inside, with acceleration, velocity and/or position setpoints and corresponding proprio-ceptive measurements (Fig. 7.15). The component’s *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 rather simple set of additions and multiplications; each circle represents a summation function block. Some of the rectangles also have data blocks 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`.

The very centralized nature of the mechanical and computational devices that were used in the first decades of automation and robotics, implied that implementations of control loops used to require no asynchronous Communication, but just synchronous reading and writing from data in the memory of the computer. The **Programmable Logic Controller** (PLC) works like this, and while it still is *the* workhorse of the automation industry, all modern versions implement the asynchronous and *hybrid* variants. The key hardware-supported technology here is **memory-mapped IO**. But most modern robotic systems now have one or more **field buses**, such as **CAN**, **EtherCat**, or another **Industrial Ethernet** variant, so some form of asynchronous Communication becomes 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 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.)

13.2.2 Mechanism: hybrid event control

No realistic application can be realised by just one single **task**, and no realistic task can be realised by just one single control loop, hence so-called **hybrid event controller** architectures 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, **inter-activity coordination**, etc. Transitions between continuous controller modes are triggered by events, generated by the actually running continuous controller itself, or by external

activities.

Obviously, hybrid controllers fit perfectly in the [event loop](#) approach, because that structures the computations, communications and configurations of the control loops, the FSMs and Petri Nets, the event triggering and processing, with synchronous as well as asynchronous activities.

13.2.3 Mechanism: cascaded control loops and their DOM models

Cascaded control loops are common practice in the [mechanical control association hierarchy](#). For example, a velocity control loop only makes sense *around* a torque control loop. That also means that a “higher” control loop should not only provide continuous time and space *setpoints* to the just “lower” control loop, but that it can also configure (and even *override*) properties of any “lower” control loop, via *discrete events*. Similarly, a “lower” control loop provides *error* and *monitoring* values to “higher” control loops. For example, the torque control *gain* could be *adapted* to the velocity control *error*, and the velocity controller can decide to switch to another control algorithm if the torque reaches “dangerous” levels.

Fig. 13.1 shows an example of a corresponding DOM structure. It is just *a* example, because the DOM model in an application is typically one of its most application-specific parts. The more “levels” an application wants to integrate, the more important the design of an appropriate DOM model becomes. Not because the *control* complexity grows, but the *coordination* and *configuration* complexity, to take care of:

- the *(de)activation* of activities at different levels.
- the *selection* of the right behavioural parameters in every level.

(TODO: more concrete examples.)

13.2.4 Pod architecture for extero-ceptive control

The architecture adds a world model (or, “chart”, or “map”) to extend the [proprio-ceptive](#) data structures of the robot-centric motion controller, with data structures that represent the location of the robot in its environment. By definition, such an extero-ceptive controller is a cascaded controller, because motion control with respect to objects external to the robot involves transformations between “Cartesian space” and “joint space”.

(TODO: more concrete examples.)

13.2.5 Pod architecture for carto-ceptive control

Yet more levels of cascaded control are introduced when a world model (or, “chart”, or “map”) becomes available, to extend the [proprio-ceptive](#) and [extero-ceptive](#) control of *one single robot* to an environment that it has to share with other devices. In other words, the single robot must add [situational awareness](#) to its own control. The data and information model decisions that have to be made in the architecture are very application-specific:

- *base map*. These are typically *metric* maps, that is with numerical coordinates for all landmarks in the map. Landmarks most often are static over “longer” time intervals. For example, [nautical chart](#), such as the [ECDIS](#) standard. Or street maps (e.g., from [OpenStreetMap](#)) or [indoor](#) maps (e.g., from [IndoorGML](#) from the [Open Geospatial Consortium](#)). Or warehouse and assembly line layouts. And even assembly graphs generated by a [Computer-Aided Manufacturing](#) system.

```

<comp id      = "C1-motion"
      properties = "Traject: A; Tol: 3;"
      class     = "Control Motion"
      data       = "C1-waypoint: {1,1};
                  CPos-1: y;
                  CVel-1: aa;
                  CAcc-1: 23;
                  CTorq-1: ss;
                  CPow-1: 3;">

<comp id      = "c2-position"
      class     = "Control Position"
      properties = "P-C2-1: 44;"
      data       = "CP1: x;...">

<comp id      = "c3-velocity"
      class     = "Control Velocity"
      properties = "P-C2-1: 44;"
      data       = "CVel-1: x; CAcc-1:x;">

<comp id      = "c4-acceleration"
      class     = "Control Acceleration"
      properties = "P-C2-1: 44;"
      data       = "CAcc-1: x;...">

<comp id      = "c5-torque"
      class     = "Control Torque"
      properties = "P-C2-1: 44;"
      data       = "CTorq-1: x;...">

<comp id      = "c6-power"
      class     = "Control Power"
      properties = "P-C2-1: 44;"
      data       = "CPow-1: x;...">

```

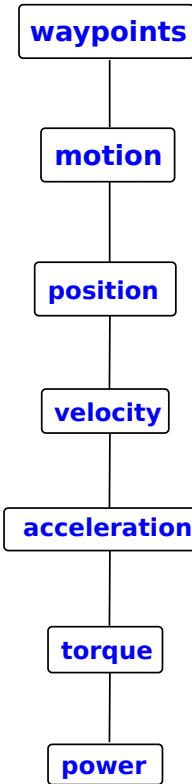


Figure 13.1: Example of DOM model for *cascaded control loops* (left), conforming to the mechanical control *association hierarchy* (right). “Higher” levels can override configuration settings of any of the “lower” levels in the DOM hierarchy, as is shown, for example, with the `CVel-1` and `CAcc-1` configurations in the `C1-motion` component at the top.

- *semantic tags and layers*: any application brings in application-specific extra landmarks, or “tags”, on the base map, and even several layers of such tags, each providing different “semantic contexts” to the robots.

Some examples are semantic layers of traffic signs and markers added to road maps; the dynamic version of this would show the actual traffic, possible with information about the status of each robot on the map and its intended actions for the near future. The [Middle Size League](#) of the RoboCup competition is an example where very dynamic maps are needed to allow a team of robots to play a game of soccer.

- *perception relations* with semantic tags: every robot must be able to connect the raw data from its sensor to the relevant “tags” on the map.

Figure 10.3 shows an example of a base map with semantic tags in a perception layer. The tag is the “symbolic pointer” to more information about how the objects where these tags are attached to, show up in the sensor data of a particular sensing technology.

- *control relations* with semantic tags: each robot must be able to link the actual status of the map to the decisions it makes about its actions.

The semantic tags in Fig. 10.3 can also provide links to the information about the maneuvers a robot boat is allowed to perform in the corresponding areas.

- *differences* between models and *model updating*: in dynamic open worlds, no map is ever complete or consistent, so peers can provide suggestions to improve the current contents of the map.

13.2.6 Policy: trade-offs between 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, because 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
}
```

13.2.7 Policy: real-time activities via the “multi-thread” software pattern

Many robotics and cyber-physical systems contain one or more activities, whose execution must be **predictable** (“**deterministic**”, “**real-time**”) with respect to the computational resources they have available:

- **time**: the execution must take place within a small tolerance of the ideal instance in time. The two key performance measure are *latency* and *jitter*.
- **memory**: the execution must respect consistency constraints on the data structures that are operated upon by the various dataflows used in the activities. A key performance measure is *mutual exclusion*, or *locking*.

- **interrupts:** **interrupts** can preempt most of the software activities on a computer, so real-time applications must configure the interrupt capabilities appropriately. The common configuration options are: (i) to inhibit (“mask”) some interrupts before a real-time activity is launched, (ii) to inhibit interrupts on the set of cores that share cache memories with the real-time core, or (iii) to assign only the real-time relevant interrupts to the CPU core on which the real-time activity is running.

The **good practice** solution, Fig. 13.2, splits the event loops of these activities into multiple parts, each in a separate **thread**, and all contained within the same **process**:

- the **mediator** thread is the one that comes with the process that is deployed in the operation system, to create the other threads in the process, and to manage their *Life Cycle State Machines*.
- the **real-time** thread executes (i) all *Computations* that must be executed *immediately*, (ii) all *Communications* that are done via non-blocking memory-mapped I/O, and (iii) the *Coordination* that is triggered by the real-time event loop itself and must be dealt with immediately (e.g., deciding to switch to a fail safe control mode).
- the **workers** are other threads, each with one single responsibility, such as (i) feeding the real-time thread with the dataflow it needs, (ii) getting the real-time dataflow and distribute it to the registered clients higher up in the control stack, and (iii) getting the diagnostic information back, to allow for online or offline analysis of the control performance. (Without loss of generality, the latter can be seen as just a special case of (ii).)

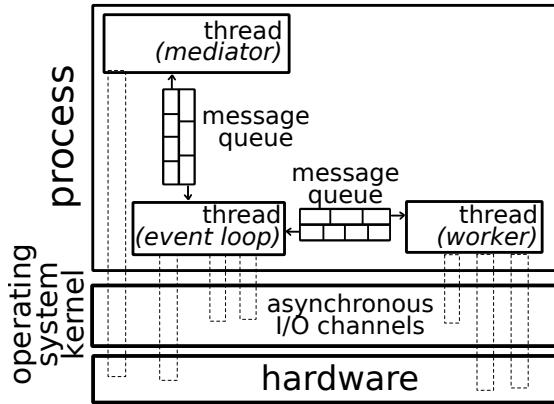


Figure 13.2: Multi-threaded process architecture for the concurrent and parallel execution of activities. The “message queues” depicted in the figure represent any type of fast and local inter-thread communication; e.g., lockfree buffers, circular buffers, etc.

The real-time performance of multi-threaded design depends to a large extent on the choice of buffers between both threads. The two common policies are to use either a **locked** buffer approach (by means of a *mutex* or another **locking mechanism**), or a **lockfree** buffer approach.

The often-used terminology of **hard** real-time and **soft** real-time has no *absolute* meaning, but only *relative* meaning between the realtime thread and the worker threads. A major **best practice** is that the design includes **only one hard real-time thread on the whole computer**. And giving that thread the highest priority allowed by the operating system is just a *necessary* but *not sufficient* condition for reaching this requirement. Putting the hard real-time part on a dedicated computer system that runs no other software, is often the only really deterministic design.

13.3 Pod architecture for Task-Situation-Resource skills

(TODO:)

13.4 Flock architecture for multi-device task execution

(TODO:)

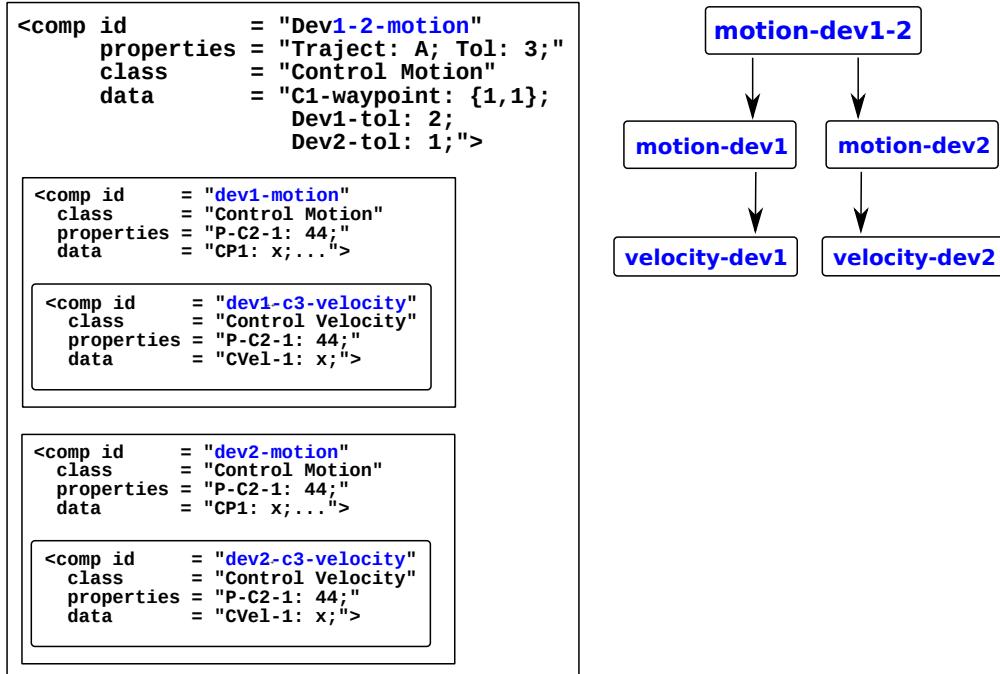


Figure 13.3: The drawing sketches a DOM, with one control component responsible for the coordinated motion of two distributed devices. In the example, it configures the *tolerances* on the parts of the motion executed by each device's velocity controller independently.

13.4.1 Flock architectures

Cascaded control loops are often required too for *distributed* devices, such as physically non-connected machines, or physically connected machines with separate control computation and interfacing hardware. Figure 13.3

13.5 DOM models for geometry in robotics

This Section and the following rely on the previous Sections' *generic* information architectural meta models, to build *particular* **DOM models** for the *information architectures*⁴ of the *particular* robotic domains introduced in earlier Chapters. The purpose of these DOM models

⁴The **software** representations, tools and implementations are introduced in a later Chapter.

is to represent *parts* of an information architecture with a *tree structured* formal representation. The selection of which parts in a particular domain are best suited for such efficient modelling (because of the tree structure) while not compromising the semantic expressivity of the model (which, in general, comes from graph structures), is continuously open for debate and adaptation.

13.5.1 DOM model for polygonal shapes

Representation of 2D and 3D shapes is very important in robotics; all of the other subsections below rely on shape representations in one way or another. There are two major, complementary approaches to represent shapes of **solid** objects:

- **Constructive Solid Geometry**: a solid object is modelled as a set of Boolean operators on primitive solids. For example, one starts with a cube from which one subtracts two orthogonal cylinders, whose axes are intersecting the centre of the cube and run through the outer boundary of the cube.
- **Boundary Representation** (B-Rep): a shape of a solid object is modelled as connected surface elements. The simplest form uses **polyhedrons** (in 3D) as a connection of planar 2D **polygons**. This representation must explicitly represent what is “inside” and “outside” of the solid that is represented by the polygonal faces, and one must take care “to close” the solid appropriately. Many of the domains discussed below can do with non-solid polygonal shapes, such as lines or frames.

The DOM model for polygons and polyhedrons already exists in a very mature and standardized form, as the **SVG 2.0** (Scalable Vector Graphics) DOM meta model. These are the relevant parts from that standard for this Chapter (whose details are not repeated in this document because they are available in detail online):

- **DOM Point**: points are the obvious core primitives in polygonal shape models. The SVG standard supports points not only in **Euclidean space**, but also in **projective space**, that is, a point can lie “at infinity”, and Euclidean shapes can be transformed into **perspective** views.
- **basic shapes**: the **elements** of rectangle, circle, ellipse, line, polyline, polygon.
- **geometry properties**: symbolic names for point and shape coordinates, like centre points of circles, or width and height of rectangles. These land in the DOM model as **attributes** in the elements. For example:

```
<rect x="1" y="1" width="1198" height="398" />

<polygon points="350,75 379,161 469,161 397,215
           423,301 350,250 277,301 303,215
           231,161 321,161" />
```

These examples show *fully specified* DOM models: each of the two elements has all of its necessary geometric parameters filled in with concrete numeric values. Both are compositions of *point* coordinates:

- the **<polygon>** element has a **points** attribute, that is an *ordered list* of, in this case, 2D x and y point coordinates. The element has the implicit⁵ constraint that the first point also acts as the (not represented) last point in that list, in order to close the polygonal shape.

⁵Implicit means that this constraint is in the *meta model*, not in the model itself.

- the `<rect>` element has attributes for the `x` and `y` point coordinates of its (implicitly modelled) top-left corner point, and its `width` and `height` attributes provide the information to reconstruct the other three points of the rectangular polygon. The orthogonality of the lines at the four corners of the rectangle is also not explicitly represented, but symbolically, in the name “rectangle”.
- **composition entities and relations:** meta data definitions, templates, hierarchical grouping, symbols and symbolic pointers. That is, the following tags: `<g>...</g>` (that realises the *tree structure* containment), `<defs>...</defs>`, `<use>...</use>`, and the *functions* `url()` (for symbolic referencing), `calc()` (for computations on symbolically represented parameters), and `transform()` (for moving and deforming the basic shapes into arbitrary shapes, at arbitrary positions and orientations).

One single `shape` element can host multiple *attachments* to connect other elements to, other `shapes`, as well as non-geometric elements. For example, a robotic application can need a model of a `kinematic_chain` that has a low resolution as well as a high resolution model of the `shape` and `inertia` of its links. The low resolution version can just have a `stick figure` shape model, and a `point mass` inertia model.

13.5.2 DOM model for coordinates

The `SVG DOM` of the previous sub-section has a meta model for coordinates. (`QUDT` provides the *knowledge relations* for coordinates. As mentioned elsewhere, both types of modelling are highly complementary.) Figure 13.4 depicts a simple example of the composition of two DOM models to represent a rectangle:

- one for the `SVG DOM` model of the rectangle represented only with *symbolic* coordinates. This model makes use of the CSS `url()` function, that acts as a *symbolic pointer*.
- one for a coordinates `DOM` model containing the *numerical values* for the symbolic coordinates.

Both `DOM` models are *symbolically* connected by the “symbolic pointer” `symRec1`.

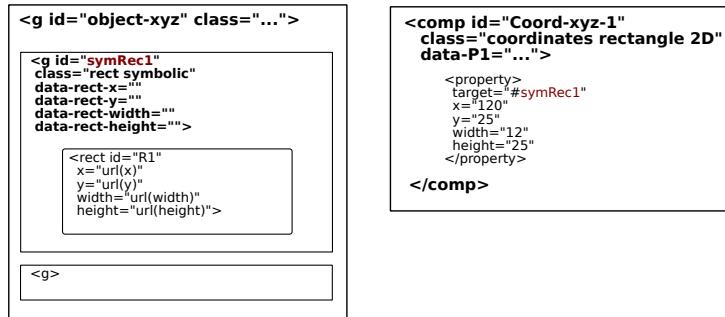


Figure 13.4: Two `DOM` trees, one for a rectangle conforming to the `SVG DOM` meta model, and one for the coordinates conforming to this document’s `DOM` meta model. The symbolic connector between both is the `symRec1` identifier.

One single geometric entity can have multiple `coordinates` `DOM` elements attached to it. From a modelling point of view, this is as simple as introducing multiple `Connectors` from the same geometric `Blocks` to different coordinate `Blocks`.

13.5.3 DOM model for geometric and kinematic chains

The `DOM` model below is an example of a very simple `geometric constraint`, the `kinematic chain`, built from one revolute `joint` between two rigid `links`, each providing some `attachments`, in the form of a `frame`. Again, the textual order of the `DOM` elements below

does not matter, but the hierarchy in the cross-links does. The cross-links, or “symbolic pointers”, are represented in the DOM document by putting a **number sign** # in front of the ID of a DOM element. In the example below, #L1 is the symbolic representation of the **link** L1 in an expression in another element.

```

<link id="L1" class="rigid">
  <property id="attachment.a" class="attachment frame"></property>
  <property id="attachment.b" class="attachment frame"></property>
  <property id="attachment.c" class="attachment frame"></property>
  <property id="attachment.d" class="attachment frame"></property>
</link>

<link id="L2" class="rigid">
  <property id="attachment.a" class="attachment frame"></property>
  <property id="attachment.b" class="attachment frame"></property>
</link>

<relation id="J12" class="constraint geometry motion joint revolute">
  <property id="proximal" class="role-proximal-attachment">
    #L1/attachment.a </property>
  <property id="distal" class="role-distal-attachment">
    #L2/attachment.b </property>
</relation>

<geometric_chain id="Chain1-2" class="kinematic">
  <property id="JointList" class="joint-list"> [#J12] </property>
</geometric_chain>
```

The models of all the elements are still limited to the **mereo-topological** parts only, that is, the minimal amount of symbolic relations required to be able to speak about something in which a domain expert can recognize the semantics of a “kinematic chain”. The element at the top of the DOM hierarchy, **<geometric_chain>**, conforms to all the classes that its children need:

- the **geometric_chain** is a **kinematic chain**, that is, only a specific set of geometric motion constraints are allowed in the model.
- the **<link>** is a geometric entity of class **link** and that link is **rigid solid**.
- the **revolute joint** is a **constraint** on the relative **motion** of both **links**.

Because of the CSS cascading property, it is strictly speaking not necessary to repeat the classes for each of the children; but it helps in storing the components of the full DOM model in separate documents, or in streaming them over a communication channel, without loosing the full semantic context.

The next paragraphs will compose, step by step, extra semantics to the minimal model above. These steps illustrate how composition is done, while (i) keeping *semantic hierarchy*, and, at the same time, (ii) being robust against any accidental *textual nesting*. “Accidental” means that the meta model developers were not aware of (i) the fundamental difference between both hierarchies, (ii) the fundamental differences between **properties** and **attributes**, and (iii) the artificial constraints that textual nesting introduces.⁶

⁶**URDF** (Unified Robot Description Format) is a popular DOM model in robotics, whose design includes many DOM modelling errors. The major cause of the problem is putting *attribute* elements *inside* the node of which they are attributes, instead of introducing a *dedicated* attribution element to encode that particular

The first composition step is that of a `kinematic_chain` model that has more than one `joint`. This does not add a new hierarchical level, but just an element on the already existing `joint` level of the DOM. So, the model below can reuse the model above, and just add the new model elements: a new `link`, with ID L3; a new `joint`, with ID J23; and a new `geometric_chain`, with ID Chain12-23. The previously modelled entities and relations are incorporated by referring to their IDs.

```

<link id="L3" class="rigid">
  <property id="attachment.a" class="attachment frame"></property>
  <property id="attachment.b" class="attachment frame"></property>
</link>

<relation id="J23" class="constraint geometry motion joint revolute">
  <property id="proximal" class="role-proximal-attachment">
    #L2/attachment.a
  </property>
  <property id="distal" class="role-distal-attachment">
    #L3/attachment.b
  </property>
</relation>

<geometric_chain id="Chain12-23" class="kinematic">
  <property id="JointList" class="joint-list ordered"> [#J12, #J23] </property>
</geometric_chain>
```

The composition above is of the `serial` type. The two other types are `branch` and `loop`.

A **branch** connects three `serial` chains via the same `link`. In other words, one `link` is connected not to one but to two (or more) other `links`:

```

<link id="L4" class="rigid branch">
  <property id="attachment.a" class="attachment frame"></property>
  <property id="attachment.b" class="attachment frame"></property>
  <property id="attachment.c" class="attachment frame"></property>
</link>

<relation id="J34" class="constraint geometry motion joint revolute">
  <property id="proximal" class="role-proximal-attachment">
    #L3/attachment.a
  </property>
  <property id="distal" class="role-distal-attachment">
    #L4/attachment.a
  </property>
</relation>

<relation id="J4X" class="constraint geometry motion joint revolute">
```

relation between a node and each of its attributes. (For example, URDF elements can have only one single set of coordinates, because they are represented as a property element nested *inside* a geometric element.) For both attributes and properties, the URDF design neglects the *cascading* and *multiple conformance* features of state of the art DOM models. (For example, the DOM model can not be used in 2D and 3D Cartesian worlds without significant changes, while in principle all that is needed is to change an attribute flag from 3D to 2D, only in those elements that are impacted by the difference.)

```

<property id="proximal" class="role-proximal-attachment">
    #L4/attachment.b
</property>
<property id="distal" class="role-distal-attachment">
    #LX/attachment.a
</property>
</relation>

<relation id="J4Y" class="constraint geometry motion joint revolute">
    <property id="proximal" class="role-proximal-attachment">
        #L4/attachment.c
    </property>
    <property id="distal" class="role-distal-attachment">
        #LY/attachment.a
    </property>
</relation>

<geometric_chain id="Chain1234XY" class="kinematic branch">
    <property id="JointList" class="joint-list ordered">
        [#J12, #J23, #J34, [#J4X, #J4Y] ]
    </property>
</geometric_chain>

```

A **loop** is a serial chain that closes on itself. Assuming we have a `kinematic_chain` with five links and five joints, and the first `link` is the one that is *arbitrarily* chosen to be the one where the loop is closed, this could be (the relevant fragment of) the DOM model:

```

<geometric_chain id="Chain12-23" class="kinematic loop">
    <property id="JointList" class="joint-list ordered">
        [#J12, #J23, #J34, #J45, #J51]
    </property>
    <property id="loopLink" class="chain-loop-link arbitrary"> #L1] </property>
</geometric_chain>

```

13.5.4 DOM model for shape and inertia kinematic chain links

Any `shape` from a `world model DOM` can be used, after it has been given an `attachment` entity:

```

<shape id="Shape1" class="shape geometry 3D">
    <property id="attachment" class="role-proximal-attachment">
        attachment.S
    </property>
    <property id="geo" class="simplex polygon geometry 3D">
        <svg> ... </svg>
    </property>
</shape>

<relation id="L1-Shape1" class="constraint attachment geometry shape link">
    <property id="link" class="role-link-attachment">
        #L1/attachment.c
    </property>

```

```

<property id="shape" class="role-shape-attachment">
    #L1/attachment.S
</property>
</relation>

```

An *inertia* model example is given below. The actual values in the inertia matrix are not yet filled in, but made linkable via the `Mass1.geo` symbolic ID:

```

<shape id="Mass1" class="inertia geometry 3D">
    <property id="attachment" class="role-inertia-attachment">
        attachment.I
    </property>
    <property id="geo" class="inertia geometry 3D matrix">
        <matrix> #url() </matrix>
    </property>
</shape>

<relation id="L1-Mass" class="constraint attachment geometry inertia link">
    <property id="link" class="role-link-attachment">
        #L1/attachment.d
    </property>
    <property id="inertia" class="role-inertia-model">
        #Mass1/attachment.I
    </property>
</relation>

```

13.5.5 DOM model for actuator and transmission in kinematic chain

A mainstream robot has one *actuator*, one *transmission* and one 1-D joint per link. The transmission is (often a constant) *geometric* factor, the *gear or torque ratio*, multiplying the *actuator torque* into the *joint torque*, that is applied to the link. The *dynamic* properties of a transmission are its *elasticity* and *inertia*.

```

<transmission id="trXX" class="geometry 1D">
    <property id="attachment" class="role-transmission-attachment">
        attachment.T
    </property>
    <property id="gear-ratio" class="geometry 1D" data-gear-ratio="3.4">
    </property>
</transmission>

<relation id="J12-Actuator" class="constraint attachment geometry joint transmission">
    <property id="joint" class="role-link-attachment">
        #J12/attachment.a
    </property>
    <property id="actuator" class="role-actuator-attachment">
        #trXX/attachment.T
    </property>
</relation>

```

(TODO: but *multi-articular* transmissions (for example, differential drive torque splitters, or finger tendons), require some extra relations.)

An **actuator** DOM model has the following form:... (TODO: similar to Inertia: there is one actuator connected to a transmission.)

13.6 DOM model for numeric, symbolic and semantic maps of world, building, and floors

Many robotic applications need to represent the location of a robot, on a particular floor of a particular building, somewhere in the world. DOM models for location on the world are already mature and standardized, in the domain of [GIS](#) (Geographic Information Systems); [OpenStreetMap](#) is a very accessible example.

The state of the practice in modelling buildings and floorplans is worse. For buildings, the [BIM](#) (Building Information Modelling) standard exists, but it is optimized for use in building design and construction, and is still difficult to apply to the sensing and control contexts of robotics.

For [floorplans](#) (that is, one single layer in a building), this document introduces the following suggestion for DOM models:

- **base layer**, with *numerical values* for the coordinates of all “corner” landmarks in the map, that is, walls, doors, and windows.
- **symbolic layer**, with (i) the *symbolic DOMPoint* names for these landmarks, and (ii) the *symbolic constraint representations* of the geometrical shapes that connect these landmarks, that is, the *rectangles* and *polygons* of Sec. 13.5.1.
- **semantic layers**, adding meaning to selections of entities in the symbolic layer, that is, to represent **rooms**, **doors**, **corridors** and **elevators**, etc.

Figure 13.5 sketches the concep. Of course, each of the three composing DOM models can consist of a tree hierarchy itself.

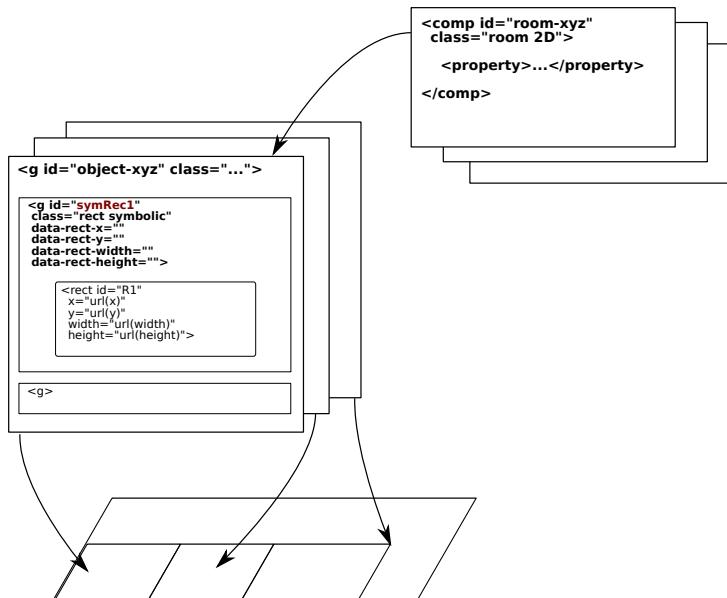


Figure 13.5: Sketch of a DOM model of three layers of other DOM models trees: (i) a basemap with *coordinates* of polygonal shapes, (ii) a map with the *symbolic* names of all relevant corner points on the basemap, and (iii) a map with *semantic* entities and relations between names on the symbolic map.

13.6.1 DOM model for semantic localisation and navigation

(TODO: component classes and tree, that add landmark models to plans; coordination and configuration; model CRUD interaction.)

13.6.2 DOM model for motion tasks

(TODO: component classes and tree; coordination and configuration; data and event interaction streams.)

13.6.3 DOM model for instantaneous motion specification in kinematic chains

Section 4.3.2 introduced the meta model to specify the instantaneous motion of a kinematic chain. The entities and relations introduced there are straightforward to encode in a DOM model:

- Cartesian forces and acceleration constraints on a link need an `attachment` entity on that link.
- actuator torques on a joint are specified in the `attachment` entity that *is* already on the joint.

The formalisation in Table 4.3 is straightforward to encode with the DOM meta model grammar used in this Chapter.

(TODO: chain-wide constraints of redundancy and under-actuation; support chains with loops and with multi-joint transmissions; `agonist-antagonist` actuation.)

13.7 Architecture for a highly overactuated mobile robot

Mobile robots must have a high degree of actuation redundancy, on a passively backdrivable mechanical platform, to deliver efficient trade-offs between task performance and physical safety of humans, environment objects and themselves. This Section introduces the architecture for a very over-actuated platform, namely one with `eight wheels`, deployed in four groups of `two wheels` sharing the same `axle` (Fig. 13.6). The focus is on the geometric, quasi-static and dynamic relations that link the forces on the platform to the torques on the wheels, i.e., traction forces in each two-wheel swivel-caster drive unit. The addition focus is on a platform built as a *composition* of several such units. The chosen platform is overkill for some applications, but it contains all relevant modelling and control aspects, including overactuation, and (hence) redundancy resolution, passive joints, and passive platform backdrivability. *Control* of the platform takes place at, at least, six levels of abstraction:

- the selection of each platform for a particular logistics mission in the organisation.
- the navigation through an indoor environment with areas with different navigation constraints, as in Fig. 6.20.
- the local motion of the platform that realises the navigation, as in Fig. 6.22.
- the motion of the four two-wheel units inside the platform that it uses for its proprioceptive motion control.
- the torque/speed control of the individual wheels inside each unit.
- the electrical-mechanical energy transformation control between mains (or battery) and wheel, via power amplifiers and battery management control.

Controllers at all levels interact in multiple ways, as represented in the task meta model:

- the transportation control must decide where the platform must move through the environment and what type of control mode is needed.
- the “force” and “motion” control of the platform must decide about how to divide the platform driving force over the various two-wheel units, which act as “force” driver resources.
- the control of each two-wheel unit must, in turn, decide about how to actuate both individual wheels taking into account the energy efficiency of their torque generation and the friction constraints of their tyres.

13.7.1 The eight-wheel drive mobile platform

Figure 13.6 sketches the ideal kinematic chain model of an overactuated mobile robot platform. It has one rigid body⁷ as a “platform”, and four identical driving units with each two actuated wheels in a symmetric configuration around a **swivel caster** connection to the platform. The advantage of this design is its *passive backdrivability*: when all power falls away, the platform can still be moved by a human, in all directions and with moderate force. The disadvantages are (i) more complex mechatronics (mechanics, electronics and control), and (ii) the need for active compensation when driving on laterally inclined surfaces.

Assumption: throughout this Chapter, the ground is assumed to be *horizontal*, and the wheel axles too. This means that a non-actuated robot is in a stable equilibrium, and, hence, *no control* is a **passively safe** mode of **failure**.

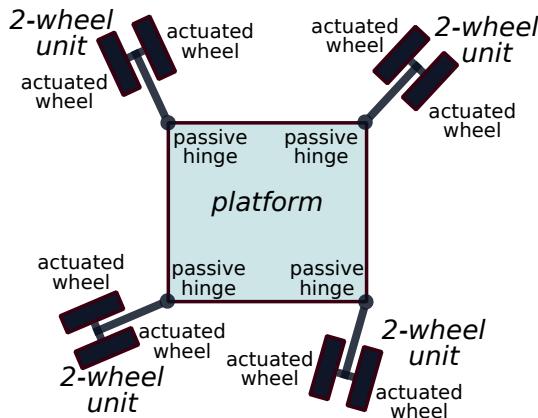


Figure 13.6: Kinematic model of an overactuated mobile robot: eight actuated wheels connected, in so-called 2WD (“two-wheel drive”) pairs, to a rigid platform via passive revolute joints with a caster offset. In real-world implementations, the magnitude of the offset is much smaller than depicted.

The mobile platform is a representative of a *complicated* robot kinematic chain model, because it embodies all of the following non-mainstream properties:

- **underactuated**: the hinges with which each 2WD unit is connected to the platform have no actuator. In principle, they also might have no encoder, so that their orientation must be *estimated*.
- **redundant**: the platform has *three* motion degrees of freedom (two for translation in the ground plane, and one for rotation), but *eight* actuators.

⁷This is an idealisation, since the mechanical platform will have some passive degrees of freedom to allow all wheel units to touch the ground at all times, and under comparable load conditions.

- *floating base*: no part of the robot that is rigidly fixed to the environment, so the robot has no “natural” origin in space.
- *non-holonomically constrained*: the wheels have rolling constraints, which are instantaneous acceleration-level constraints that can not be “integrated” into position constraints.
- *N-DOF joints*: the platform is actuated by four forces, each generated by two actuated wheels *in parallel*. Hence, there exist no *spanning trees* in the kinematic structure that makes solving the kinematics and dynamics linearizable into 1-DOF problems: the force distribution over both wheels must be solved together as a 2-DOF problem.
- *Cartesian friction and slippage*: the forces in the actuators are not guaranteed to be transmitted to the platform in full, since the wheels can slip, and they also lose force transmission efficiency via friction.
- *partially observed*: because of slippage, the sensors on the actuators are not sufficient to measure, or even estimate, the full motion state of the robot. Sensor fusion with extra sensors, like [Inertial Measurement Units](#) or cameras, must be introduced.

13.7.2 Topological navigation and metric motion tasks

The *application context* of this document is the usage of overactuated mobile platforms in indoor environments. Figure 13.7 sketches a typical area in such a context, with “corridors” coming together in “intersections”, and having “doors” to “rooms” and “elevators”. There are several complementary types of “tasks” (and, hence, also corresponding types of “control”, “perception”, “world modelling” and “monitoring”), depending on which *level of abstraction* that one looks at the available fleet of robots: *mission*, *navigation*, *platform motion*, *2WD motion* and *wheel motion*.⁸

The logistics unit in the organisation has the *mission* to provide the right goods at the right time at the right place; to reach that goal, it must decide (“control”) which platforms to deploy, where and when, and in what type of configuration. Each individual mobile platform has the *task* to *navigate* through such environments. Maps of the environments contain *no explicit motion trajectories* for any particular type of robots, but only the *declarative constraints* that *any* moving robot should respect when it wants to navigate through the environment without colliding into it. This document uses the term “*navigation*” to denote a sequence of connected “semantic areas” that the robot traverses,⁹ and the term “*motion*” to denote the (timed) “geometrical path” that the robot platform travels through.¹⁰ The term “*motion*” has itself *three complementary* meanings, depending on whether it pertains to (i) the platform as one rigid body, (ii) one single of the four 2WD units attached to a platform, or (iii) each individual wheel in one such 2WD unit.

The mobile platforms have *to control*, both, navigation and motion, by making the platform/units/wheels move in such a way that the desired/expected navigation or motion is realised. Navigation control pertains to the activity to decide which areas a robot should traverse; this is a [symbolic constraint satisfaction](#) problem that deals with topological navigation. The metric level of abstraction is on the *motion control*, formulated as a [hybrid constrained](#)

⁸This document does not consider the task levels “below”, such as battery control and power convertor control, not the one “above”, such as the decisions about investment and maintenance in robotic technology, or about their integration into the overall logistics operations.

⁹Hence, this is a *topological* abstraction of the robots’ motions.

¹⁰Which is the *metric* representation of the robots’ motions.

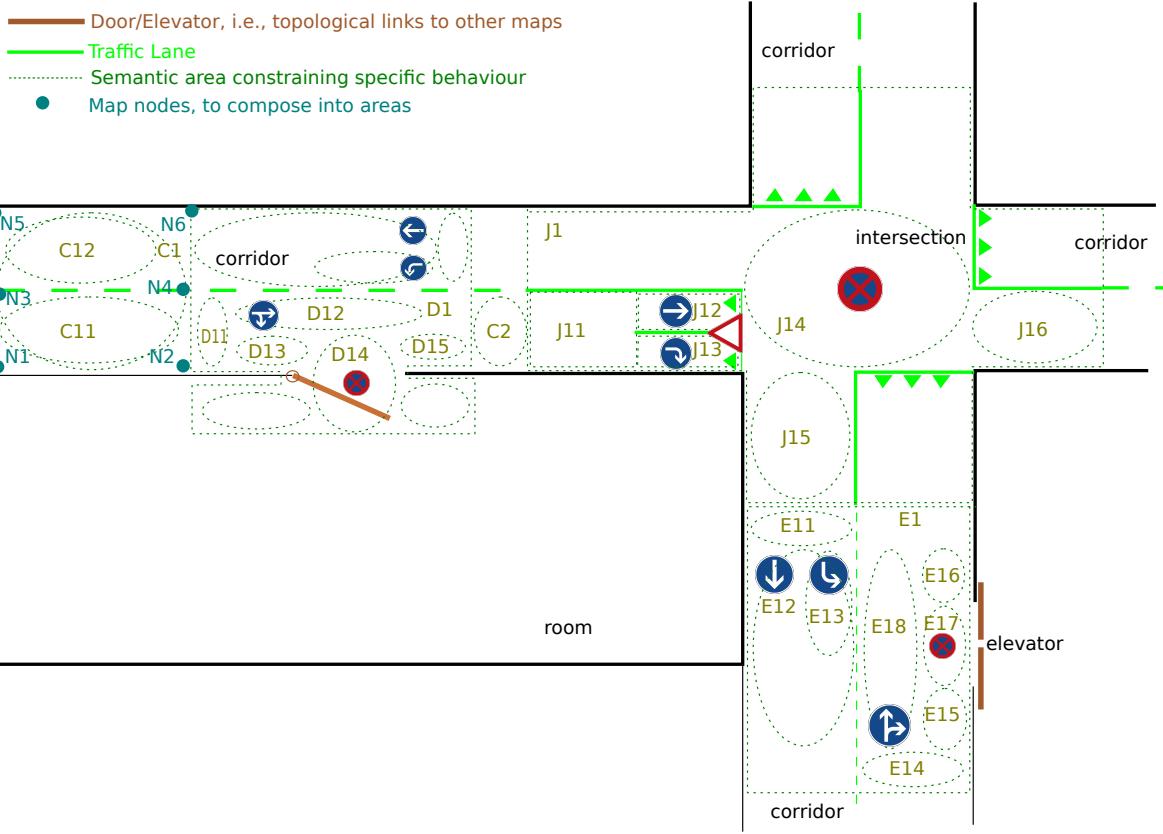


Figure 13.7: A typical indoor environment to deploy mobile platforms in, with a lot of (virtual) *semantic tags* (inspired by real-world [traffic signs and signals](#)) to describe the *intended* navigation usage of parts of the area (such as “traffic” lanes and stopping, halting or waiting areas), and of objects (such as doors or elevators). The drawing does not show *perception* and *action* tags, which are both necessary in a robotics application context.

[optimization](#) problem in the *continuous* (metric) domains of time, space, force, energy,...

A mobile platform can use three complementary types of *motion* control (at all the motion control levels of the **platform**, the **2WD units**, and the individual **wheels**), depending on what are the *inputs* of the navigation task specification and the *measures* with which the *progress* (or “*utility*”) of the task execution will be evaluated:

- “*force*” *control*: the *inputs* are desired force and torque applied somewhere on the platform, together with a target area or tube within which the control must move the platform. The output of the control are desired forces to be created by each of, respectively, the two-wheel units, or the individual wheels. The progress is measured by how well the resulting motion fits in the target tube.
- “*motion*” *control*: the *input* is the desired “*motion*”, that is, acceleration, and/or velocity, and/or position of some reference points on the platform. The output of the control are, either, desired forces, or desired motion, to be realised by each of, respectively, the

two-wheel units, or the individual wheels. The *progress* is measured as (some sort of time-averaged) error between desired and actual motion.

- “*impedance*” control: the *inputs* are desired velocity, and/or position, and the stiffness and/or damping that a “disturbing agent” will feel when physically interacting with the system. The *output* of the control are desired forces to be realised by each of, respectively, the two-wheel units, or the individual wheel. The *progress* is measured as (some sort of time-averaged) error between desired and actual motion.

Of course, every task comes with specific extras on top of the mentioned control inputs and outputs:

- one or more *cost* functions, e.g., energy consumption, time, distance, number of changes in control modes, smoothness of the motion, maximum actuation efforts, etc.;
- one or more *risks*, e.g., how big would be the extra cost incurred by the system if the current task specification can not be realised;
- *constraints* on the admissible values of some parameters;
- *tolerances* on these parameters (and on their constraints) that represent the solutions that are “good enough”.

13.7.3 Platform navigation and motion modes

The choice of a particular control approach is represented by a set of *modes* that a mobile platform can be in:

- *cruising*: the robot traverses an area that has no declarative constraints (“semantic tags”) that can result in “discrete” or “large” transitions in motion speed or heading. The appropriate continuous control is “force” control to keep the platform within the area’s *declarative constraints*, and the discrete control to decide whether or not to bring some 2WD units towards *pushing* configurations;
- *maneuvering*: the robot moves within one particular semantic area, with the goal to reach one specific sub-area (with a “large” tolerance), and this is expected to result in discrete transitions in motion speed or heading.
The appropriate continuous control is impedance control, with discrete wheel configuration switching control.
- *positioning*: the robot must reach a specific geometric *position* (that is, an “area” with “small” geometric tolerance) within one particular semantic area.
The appropriate control is motion control, with discrete wheel configuration switching control.
- *2WD configuring*: no control is active at the *platform* level, since the state of the 2WD units has not yet been established. Hence, the control activity is first to be defined and realised at the “*joint*” level, that is, that of the individual 2WD units.
- *wheel configuring*: before any motion can be controlled, one first has to configure the controllers of the *individual wheels*.

The order in which these modes have been presented is not a coincidence: it is a hypothesis in this document that the order reflects a *strict hierarchy*: the *nominal* task in a “higher” level can be extended by one or more “lower” level tasks that, together, represent a *pre-emption* of the nominal higher-level execution. For example, a *cruising* task can be refined with an *overtaking maneuver*, that consist of: (i) maneuvering from one “lane” to a neighbouring lane, (ii) cruising in that lane till one has moved “sufficiently” further than the overtaken obstacle, and (iii) maneuvering back to the original lane. The “pre-emption task” should *not*

replace the original cruising, but it puts the three mentioned maneuvers on the “stack” of the robot controller, so that the original motion can be *resumed* at any time that the current (and hence also all of the remaining) pre-emption maneuvers are not needed anymore.

13.7.4 2WD force transmission

This Section zooms in on (the quasi-static model of) one two-wheel unit connected to the platform via a swivel caster. Figure 13.8 shows the pushing and pulling forces that can be transmitted (quasi-statically) from the wheel traction forces to the platform. The model assumes that both wheels are identical, that they can provide a maximal force of F^n , that both wheels are a distance $2 \times r$ apart, and that they are a distance l away from the revolute joint connecting the wheel unit to the platform (represented by the black circle in the middle of the figure).

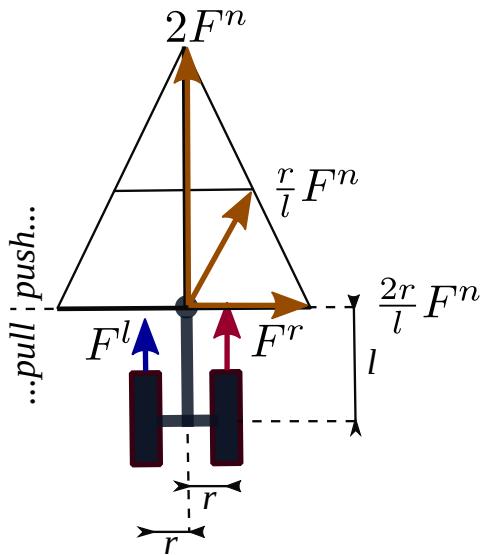


Figure 13.8: The forces transmitted between the traction forces on the wheels to platform, which is connected through a frictionless revolute joint that transmits only forces and no torques. The “triangle” spans the domain of all possible forces that can be exerted on the platform by varying the wheel forces. When one of the wheel forces is larger in magnitude than the other one, and opposite in direction, the wheels pull the platform, otherwise they push it. The three orange forces on the drawing represent the tangential force components resulting from the cases in which $F^l = F^n$ and F^r equals, respectively, F^n (vertical force), zero (middle force), and $-F^n$ (bottom of the triangle). Only the triangle in the “push” direction is shown, that is, when the resultant of F^l and F^r is positive in the direction from the wheels towards the platform.

Because of the revolute joint, the wheels cannot transmit a *moment* to the platform, but only pure forces when represented with respect to the center of the joint. The maximal force in the direction of the wheels is $2 \times F^n$; this forms the top of the triangle. If the magnitude of both wheel forces become smaller, but still equal in magnitude, the resulting force on the platform is situated lower on the vertical centre line of the triangle. If their magnitudes are not equal, a force component off this centre line is generated, because it must offset the moment generated by the unbalanced wheel forces. When the left wheel force remains at F^n and the right wheel force decreases, the resulting platform forces generate the right-hand border of the triangle. When the right wheel force is $-F^n$, both wheels create a pure moment of $2rF^n$ around the centre point in between both wheels, where r is the distance between that centre point and the point where a wheel force is applied to the ground. At the revolute joint on the platform, this pure moment results in a pure force of $\frac{2r}{l}F^n$, in the horizontal direction. When the right

wheel force is zero, the wheel torque is half the value of the previous case, so the horizontal offset force is $\frac{r}{l}F^n$.

The force transmission is inversely proportional to the *castor offset* length l . When that length goes to zero, the transmission goes to infinity. Geometrically speaking, the side of the triangle becomes more and more horizontal, and for $l = 0$, it becomes impossible to generate a horizontal force component by the two wheel forces. This limit case corresponds to the traditional [differential wheeled robot](#) kinematics.

Magic numbers:

- the geometric parameters of the model in Fig. 13.8;
- the geometry of the wheel, that is, the wheel diameter;
- the mechanical dynamics of the wheel, that is, mostly, its mass, its rotational inertia, and the friction of the wheel bearing;
- the electrical dynamics of the wheel, that is, mostly, its torque and velocity constants, torque-speed efficiency curve, and its maximum torque and current;
- the width of the contact patch;
- the friction of the wheel, both rolling and sliding;
- the slip properties, but these are a composed property of the interaction of the wheel with the material of the ground surface.

The mapping from two wheel torques to the force transmitted to the platform via a passive revolute is bijective, and it has no singularities. Hence, no “pseudo-inverse” or “redundancy” choices need to be made, introducing no extra parameters. Of course, there are saturation limits to torque/force transmissions, due to friction at the wheels and actuation limits in the motor.

13.7.5 2WD efficiency of force transmission

The triangle in Fig. 13.8 also helps to discuss the **efficiency** of the force transmission from wheel motors to platform motion. The “efficiency” η^f is being defined here as the *ratio* between the *absolute magnitudes* of (i) the platform force F_i^p that wheel unit i contributes to the overall force on the platform, and (ii) the resultant force vectors F_i^r and F_i^l of the forces in the right and left wheels of the unit:¹¹

$$\eta_i^f = \frac{\|F_i^p\|}{\|F_i^r\| + \|F_i^l\|}. \quad (13.1)$$

The triangle shows that the wheel forces can generate forces on the platform whose *magnitudes* are larger than the magnitudes of the wheel forces; but there are some “hidden” costs behind this observation:

- *instantaneous power* must be conserved.

Since the *work*, or *power*, generated by applying the same forces for a small distance or time should be conserved, larger forces result in smaller motions on the platform than on the wheels. That is because the force on the platform feels a higher instantaneous inertia than just the wheel unit inertia. So, nothing is magically gained here with respect to the propulsion of the platform. On the contrary: as illustrated in Fig. 13.9,

¹¹An *energy-based* efficiency ratio can only be defined when a *full dynamic model* of all hardware components is available.

the efficiency of the transmission goes down with increasing angle between the wheel centre axis and the force vector on the platform created by the wheel unit.

- the *overall energy* required to move the wheel units must be considered over non-instantaneous trajectories.

Since the forces generated at the wheels accelerate, both, the platform and the wheel units, any force on a wheel unit that “points away” from the desired motion of the platform will start accelerating the wheel in the “wrong” direction. Sometime later, the energy put into this acceleration will have to be counteracted with energy to accelerate the wheel in the “right” direction.

- the *friction* and acceleration forces can fight each other.

Since several wheel units are driving the platform at the same time, and since uncontrolled external forces can be acting on the platform at any time, the forces *generated* at one wheel unit are not the only ones that *move* the unit. The result is that *wheel-twisting* sliding friction is inevitable at the point of contact between the wheel and the ground, and hence energy is lost in this way.

- *traction slip* can occur.

The driving forces on a wheel can become larger than the friction between wheel and ground surface.

From the discussion above, it is clear that there are multiple definitions of “efficiency”. For example, some slippage in the traction wheels decreases *energy consumption* efficiency, but can improve *motion progress* efficiency.¹²

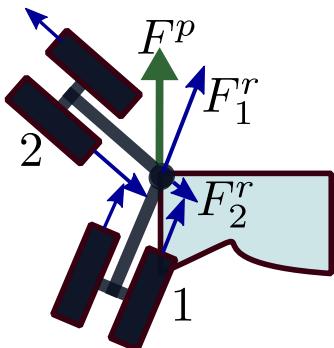


Figure 13.9: The angle between the wheel centre line and the line of the generated force on the platform determines the “efficiency” of the transmission of forces at the wheels to the resulting force on the platform.

In the “push” case, all forces required to realise the platform force are also accelerating the wheels in the direction of further alignment with the desired platform force; in the “pull” case, the wheel accelerations are in the opposite direction. The obvious and most optimal case of 100% efficiency is not depicted, and that is the case in which the centre line of the wheels is parallel to the platform force, so all forces are pointing in the same direction *and* friction is limited to just the *rolling friction* (which is an inevitable physical fact, that can not be improved by control).

Magic numbers:

- the parameters in the various possible efficiency functions;
- the thresholds for the decision making to actuate a wheel or not;

¹²Indeed, the fact that a wheel does not slip implies that it has the potential to provide more traction, so one can only be sure to have obtained maximum wheel traction if a “small” amount of slippage occurs.

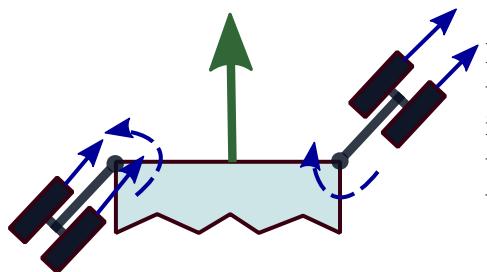


Figure 13.10: The left-hand side wheel unit is pushing the platform, the right-hand side wheel unit is pulling it. The dashed arrows indicate the motion direction of the wheel unit as a result of the forces applied to each wheel.

- the wheel slippage parameters.

13.7.6 Platform pushing is (locally) stable, pulling is unstable

Figure 13.10 sketches the forces on the wheels that are needed to generate a desired platform force, once in a “push” configuration (left) and once in a “pull” configuration (right). In the push case, the forces *improve* the *alignment* between wheel unit and platform force, but they *deteriorate* the alignment in the pull case.

This observation only holds *instantaneously* and *locally*: it might be the case that the current task has the explicit intention of turning the pulling wheel unit around into a pushing configuration, and then multiple types of trade-offs can be made between realising this mode change and contributing to the platform propulsion.

Magic numbers:

- the parameters in the various possible friction functions of the wheels with the ground surface;
- the thresholds for the decision making to push or to pull with each wheel unit.

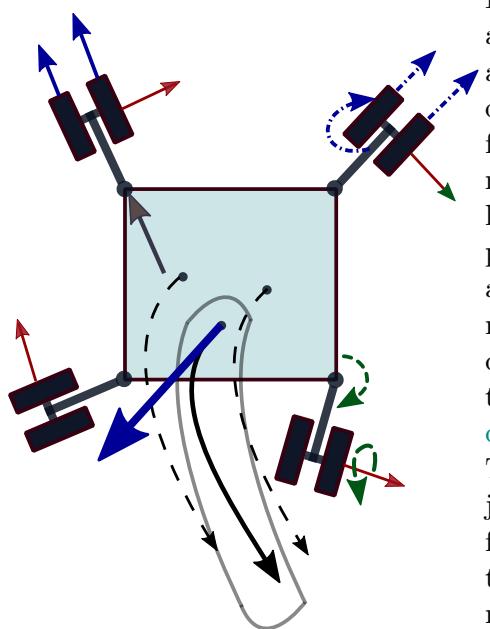


Figure 13.11: The various (physical and virtual) forces acting on the mobile platform: the small blue arrows are active traction forces on the wheels (the F^r and F^l of Fig. 13.9), which result in the grey force on the platform (the F^r of Fig. 13.9); the small blue dashed arrows are passive friction forces (rolling and sliding); the large blue arrow is the (desired or actual) force on the platform body; the red small arrows represent the ideal acceleration constraint forces, orthogonal to the nominal rolling direction of the wheels; and the small green dashed arrows are the friction forces in the bearings of the wheels and of the passive revolute joint in the [swivel caster](#) suspension to the platform.

The full and dashed black arrows are desired motion trajectories for some selected reference points on the platform; the gray “tube” around the centre one represents the *tolerance* that the motion *specification* allows the motion *controller* to have at runtime.

13.7.7 Platform force distribution over all 2WD units

The previous Sections looked at the force interactions between one single wheel unit and the mobile platform. However, that platform has *four* wheel units, with in total *eight* actuated wheels (Fig. 13.11). Physically, all wheels transmit their forces to the platform in parallel, so there are *five degrees of actuation redundancy*, and it becomes the responsibility of the *platform motion controller* to decide *how to distribute*, over the four wheel units, the desired force and torque needed for the motion of the platform. The trade-offs to be made in force distribution are: (i) how much *internal platform force*¹³ the wheel units should produce, and (ii) how much force and energy each wheel unit is expected to contribute to the platform motion.

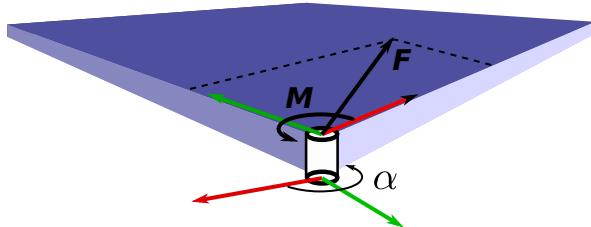


Figure 13.12: The revolute joint that connects the wheel unit to the platform transmits translational forces only. That is, assume that $(\mathbf{F}_x, \mathbf{F}_y, \mathbf{M})$ represents a general force on the platform (expressed in the reference frame on the platform whose origin coincides with the wheel suspension point), then the force \mathbf{F} will be transmitted from platform to wheel unit, but the moment \mathbf{M} will not. In the other direction, a wheel unit can only transmit a force \mathbf{F} to the platform. The angle α represents the rotation between the (arbitrarily chosen) reference frames on, respectively, the platform and the wheel unit axis.

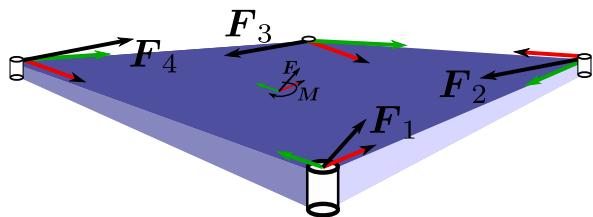


Figure 13.13: The total force \mathbf{F} and moment \mathbf{M} on the platform is the sum of the forces $\mathbf{F}_i (i = 1 \dots 4)$ contributed by all four wheels.

¹³Internal platform forces do not contribute to the acceleration of the platform, since they are the forces that are compensating each other between the driving wheel units. In general, this is a loss of energy, but there might be use cases in which it is useful to use internal forces to make the motion control of the platform *artificially more stiff*: a human trying to push away the platform will feel a higher resisting force than in the case of a non-actuated platform..

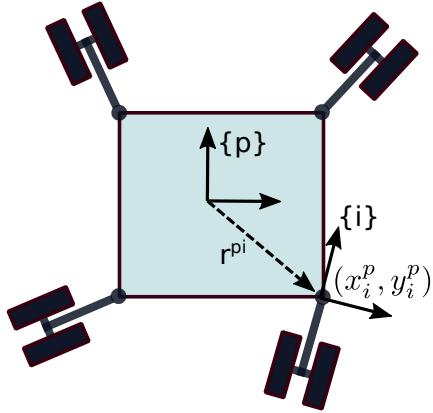


Figure 13.14: For the coordinate representations, and their transformations, two reference frames are relevant: the $\{p\}$ frame fixed to the platform, and the four $\{i\}$ reference frames, with origin in each of the wheels, and with their orientation fixed to the attached wheels. The vector r^{pi} connects the origin of the $\{p\}$ frame to the origin of the $\{i\}$ frame. The coordinates (x_i^p, y_i^p) are those of the attachment point of the joint in the $\{p\}$ reference frame.

Mechanism: forward force composition from wheels to platform

The following (quasi-static, linear) *force mapping* is a (quasi-static) *physical constraint* that must always be satisfied:

$$\underbrace{\begin{pmatrix} F_x^p \\ F_y^p \\ M^p \end{pmatrix}}_{3 \times 1} = \underbrace{\begin{pmatrix} F_x^p \\ F_y^p \\ M^p \end{pmatrix}}_{3 \times 8} = \underbrace{G}_{3 \times 8} \underbrace{\begin{pmatrix} F^1 \\ \vdots \\ F^4 \end{pmatrix}}_{4 \times (2 \times 1)} = G \begin{pmatrix} F_x^1 \\ F_y^1 \\ F_x^2 \\ F_y^2 \\ F_x^3 \\ F_y^3 \\ F_x^4 \\ F_y^4 \end{pmatrix}, \quad (13.2)$$

where F^P represents the force (two components) and torque (one component) on the platform, and F^i represents the force on the platform generated by the combination of right and left wheel in the i th wheel unit. This force has only two components (Fig. 13.12), since the moment component is not transmitted through the revolute joint that connects the wheel unit to the platform.

G is called the *forward force matrix*, because, *physically*, it adds the forces F^i generated by the four wheel units. The *numerical values* in the G matrix depend on the following geometrical parameters and choices, Fig 13.14:

- the reference frame $\{p\}$ that is chosen to express F^p .
- the reference frames $\{i\}$ that are chosen for each of the F^i .
- the geometric parameters of the kinematic chain formed by the wheels attached to the platform. In Fig 13.14, these are the coordinates (x_i^p, y_i^p) for each of the four wheel units.
- the choice of physical units; e.g., meters and radians.
- the choice of digital representation; e.g., first two 32-bit floats for the force component, followed by one 32-bit float for the moment component.

(The latter two choices are not visible in the Figure.) Taken all this together, G has the

following form:

$$G = \begin{pmatrix} c_1 & s_1 & c_2 & s_2 & \cdots & c_4 & s_2 \\ -s_1 & c_1 & -s_2 & c_2 & \cdots & -s_4 & c_4 \\ -x_1 s_1 & y_1 c_1 & -x_2 s_2 & y_2 c_2 & \cdots & -x_4 s_4 & y_4 c_4 \end{pmatrix}, \quad (13.3)$$

$$\text{with } c_i = \cos(\alpha^i), s_i = \sin(\alpha^i). \quad (13.4)$$

Mechanism: inverse force distribution from platform to wheels

In the platform control, the *inverse force mapping* is required, that is, a method **to distribute** each *desired* platform force F^P onto four desired wheel unit forces F^i . This is as simple as “inverting” Eq. (13.2). However that inverse is not uniquely defined, since G is not a square invertible matrix, for two reasons:

- *redundancy*: there are an infinite number of wheel forces F^i that realise any given platform force F^P , so one has to choose a **policy** to determine the distribution. This boils down to “minimize” the internal forces between the wheel units according to some arbitrarily chosen **metric**.
- *singularity*: the wheel units are not one-to-one transformers of force, since there is the **constraint** that the moment component is not transmitted.

With these physical insights, in combination with other arbitrary choices that have to be made, the force distribution problem has the following *conceptual* solution:

- first, decide which wheel units will have *to contribute actively*, and which others will be used in unactuated, *coasting* mode.
- *weigh* the contributions over the wheels with the *apparent inertia*¹⁴ felt at each swivel attachment point.
- *scale* this inertia-weighted distribution via the *efficiency* (Sec. 13.7.5) that each active wheel unit can provide.

The conceptual solution can be realised in many different ways, for which multiple decisions have to be made: when to let a wheel coast or not, which version of “efficiency” is used, what is the allowed tolerance, etc. The *inertia weighted pseudo-inverse* formulation looks as follows:

$$\min_{F_x^i, F_y^i} (GF^i - F^P)^T W^p (GF^i - F^P) + \sum_i (F^i)^T W^i F^i. \quad (13.5)$$

W^p and W^i are the translational parts only of the full Cartesian space inertia matrices of the platform, expressed in the frames $\{\text{p}\}$ and $\{\text{i}\}$. Only the translational force components F_x^i and F_y^i (in each of the four wheel units) are being optimized over, so, the minimization searches for 4×2 numbers, that realises the desired platform force and moment $F^P = (F_x^P, F_y^P, M^P)$, keeping the magnitudes of the force components F_x^i and F_y^i as low as possible weighted by the platform inertia felt at each wheel unit attachment point. One could add the following four scalar **constraints** to the minimization in Eq. (13.5):

$$\forall i, M^i = 0, \quad (13.6)$$

representing the fact that, at the attachment points, moments can not be generated by the wheels anyway.

¹⁴The “apparent inertia”, or “articulated body inertia” [49, 158] can only be computed if a full dynamics model of the platform is available.

Magic numbers:

- the parameters in the various possible weighing functions;
- the parameters in the motion tubes;
- the parameters in the time and space horizon of a non-instantaneous motion specification;
- the *epsilon* and *maximum number of iterations* parameters in the SVD algorithm;
- the choice of the motion reference points on the platform.

Mechanism: force distribution at the wheels

The Section above explains how to find the 2D force (F_x^i, F_y^i) to be generated by each wheel unit i , to realise a desired force and moment on the platform. This Section adds the properties of a 2WD unit, Secs 13.7.4–13.7.5. On one hand, this means to add two more **constraints** to Eq. (13.5):

- the **efficiency** of a unit to transmit forces from the wheel actuators to the platform attachment point must be above a certain threshold.
- the **friction** at each of the wheels constrains the maximum magnitude of the transmitted force.

On the other hand, the 2WD level of abstraction also adds a possible extra **objective function**, that represents the desire to find solutions to the force distribution problem that change minimally over time:

$$\min_{F_x^i, F_y^i} (GF^i - F^p)^T W^p (GF^i - F^p) + \sum_i (F^i)^T W^i F^i + \sum_i (\Delta F^i)^T W^i \Delta F^i, \quad (13.7)$$

where ΔF^i is the difference between the currently computed force magnitude and the one that was applied at the previous sample instant.

Policy: numerical force distribution solvers

When one opts for a *linear* algorithm to solve the force distribution problem, the following linear algebra insights¹⁵ are relevant:

- *pseudo-inverse*: a non-square linear set of equations $Ax = b$, like Eq. (13.2), has a solution $x = A^\dagger b$. Depending on whether A is of full row rank or column rank, the analytical expression for A^\dagger is $(A^T A)^{-1} A^T$ or $A^T (A A^T)^{-1}$.
- *minimization problem*: this pseudo-inverse solution x is also the solution of the following “*least squares*” problem:

$$\min_x (Ax - b)^T (Ax - b) + x^T x. \quad (13.8)$$

In other words, it finds the “smallest” x that is mapped by A to the “closest” possible point near b .

- *weighted pseudo-inverse*: in most cases, the “squares” in the equation above must be replaced by “*weighted least squares*”, because of dimensional homogeneity¹⁶ or task

¹⁵More technical details can be found in linear algebra textbooks such as [61, 141].

¹⁶Indeed, in many cases, different components in x and/or b have different physical units. This holds for the forces (with Newtons as units) and torques (with Newton-meters as units) in the platform dynamics context of this document.

specification trade-offs:

$$\min_x (Ax - b)^T W (Ax - b) + x^T K x, \quad (13.9)$$

with W a *metric* in the *range space* of b , and K a metric in the *domain space* of x .

- *inertia weighted pseudo-inverse*: nature provides a particular way “to solve” Eq. (13.9), where W is a Cartesian-space inertia matrix and K its joint-space equivalent, e.g., [60, 158].
- *Singular Value Decomposition solver*: all of the problems above can be solved via (variants of) the numerically optimal *SVD* algorithm, e.g., [149].
- *runtime iteration solver*: in general, solving an SVD problem involves *numerical iterations* until *convergence* to a user-specified “epsilon”; in a robotics context, one could think of spreading these iterations over subsequent controller time instants, using the last computed iteration as a *feasible* initial solution for the next iteration, with somewhat changed problem parameters.

Mechanism: force trajectory objectives

The previous Sections considered an **instantaneous** problem only: the optimizations only consider the actual values, and have no horizon of several sample instants, to the future as well as to the past. In many use cases, introducing a **finite horizon** makes a lot of sense, to optimize effects that need some time to be visible and/or controllable, such as, for example, oscillations, dampings, or hysteresis effects. The generic answer to this is to introduce constrained optimization over a horizon, which is known as **Model Predictive Control** (MPC). This approach adds a term in the objective functions of Eq. (13.7) and its “predecessors”, for each time instant in the horizon, and that means an exponential increase in the computational complexity of the solvers. The approach to reduce this combinatorial explosion is to introduce **parameterized trajectories** that have only a small number of parameters. In other words, by “sampling” over the parameters, a variety of trajectories is generated, for each one the objective function is evaluated, and then the optimal one is selected. This has several advantages, that reinforce each other:

- **link with control and plan**: these parts of the *platform-level* Task meta model often often already make use of parameterized trajectories, which can be reused inside the force distribution problem.
- some **constraints and/or objective functions** can be **satisfied by design**, because the generated trajectory already satisfies them.
- **contracts about expected and actual progress**: a side effect of using simple local trajectories at the platform-level of the motion, is that also each 2WD unit gets condensed information about what the platform control expects from the 2WD control. This information can be used as a contract between both controllers, and as a reference to measure progress of the control execution at both sides.

Examples of short-horizon trajectories are: constant velocity, constant acceleration, or **clothoid** (linear acceleration changing) curves, as sketched in Fig. 6.31; all have *known* oscillation and smoothness properties.

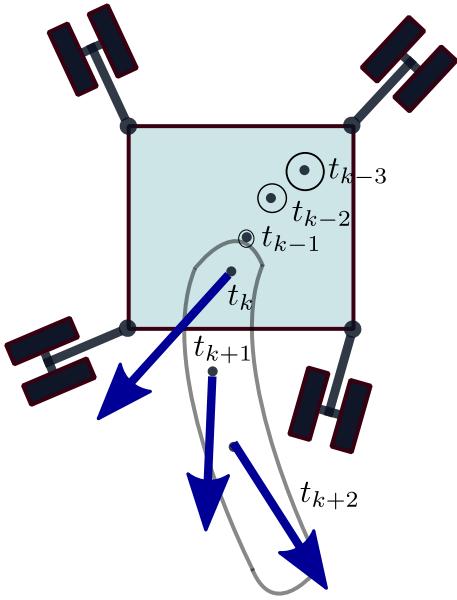


Figure 13.15: At the current time instant t_k , the platform controller primitives are (i) the current desired total force at a chosen reference point on the platform, (ii) the expected n future such force specifications (with $n = 2$ in this example), and (iii) the last m positions that the platform reference point has moved through (with $m = 3$ in the example). The circles around the latter points refer to the uncertainty in knowing their coordinates.

13.7.8 Mechanism: declarative platform motion specification via tubes

A complementary approach to solve the platform control problem not instantaneously, but over a certain “horizon” in time and space, is depicted in Fig. 13.15. Instead of using a *procedural* specification in the form of a parameterized curve, it introduces a **declarative** specification in the form of a “*tube*”. The control must choose platform force/torque drivers $F^p(t_i), i = 1, \dots, N$ over a finite time horizon of N samples, with which to drive the platform in such a way that its motion is (expected) to remain within the specified tube. There can be more than one tube at the same time, as long as they overlap, since each one represents the *tolerance* within which the motion of a selected reference point on the platform must remain.

For online use, it makes sense to extend such “forward-looking” time series $F^p(t_i)$ of driver forces over a particular horizon of N instants, with a “backward-looking” time series of recently passed positions $p^p(k), k \in -1, \dots, -M$ of M instants (Fig. 13.15). The reason is that there is a high uncertainty about where exactly the platform was in the recent past, due to the high number of disturbances that can occur; e.g., slippage, internal forces between wheel units, external forces, etc.

13.7.9 Policy: 2WD control modes, task progress, energy, slippage

By means of the methods explained in the previous Sections, the platform control generates, for each 2WD unit, a local (in time and space) motion specification, together with an indication of the progress that it expects each two-wheel unit to make within that control horizon. The responsibility of the 2WD controller is then to find an “optimal” trade-off between:

- realising that progress. In the context of the **Task meta model**, this is the *capabilities* part of the 2WD Task control level.
- minimizing the energy to do so. This is the *resources* part, where the electrical motor and battery are the resources. (Or similar components for other types of actuation.)
- minimizing the slippage of the wheels. This is the sole responsibility of the 2WD Task controller, not to be shared with any other control level.

- optimizing the efficiency of the unit's force transfer to the platform. Also the sole responsibility of the 2WD Task controller.

A 2WD unit has the following *modes* (with qualitative descriptions of their suggested behaviour) to determine the choice of trade-off:

- *pulling*: only the “outermost” wheel provides traction, while the “innermost” wheel coasts. Otherwise, the wheel unit would make its own position evolve into one with worse efficiency.
- *pushing*: both wheels provide traction forces. The magnitude can be made proportional to the relative efficiency of this 2WD unit compared to the other units.
- *from pulling to pushing*: both wheels coast, while the platform is driven by the other 2WD units, until this unit's efficiency has reached a certain threshold, after which it can start applying traction.
- *from pushing to pulling*: both wheels apply traction, irrespective of the unit's efficiency and the actions of the other units, with an *open loop* trajectory that should turn the unit around its swivel point. Of course, what the other units are doing brings disturbances to the expected evolution of the wheel unit turning.
- *unknown*: because of the partial observability of the platform (Sec. 13.7.1), the configuration of a wheel unit with respect to the platform might not be known with sufficient certainty to decide on a control mode.

13.7.10 Policy: hybrid platform control

The approach presented in this Section assumes that one has good *a priori* knowledge about how the platform behaves under given wheel actuation in the various *control modes* of, both, the platform and each of the 2WD units. The discrete part of the Task control, the plan, makes the decision to switch between such control modes. For example, the following configurations¹⁷ of the wheel units correspond to different force transformations from wheels to the platform:

- all wheel units are “pushing” (Sec. 13.7.6), and with similar “efficiencies”.
- one wheel unit is “pulling”, the others are “pushing”;
- one wheel unit is “pulling” and must transition to “pushing”;
- the platform must transition between two of the possible “motion” control modes: position, velocity, force, impedance, energetic effort, . . . , with or without “tube” tolerances on the setpoints/ranges specified in the control domain space.
- the platform starts a “motion” from a stand-still situation, with none of the wheel units being aligned, neither amongst themselves, nor with the intended platform motion direction;
- ...

The relevant *prior knowledge* for each mode has not yet been identified, let alone formalized in computable algorithms, but the commonalities between all approaches are the following:

- one selects a particular platform control mode;
- one selects particular wheel actuator forces, based on some variant of a *lookup table* or the solution of a constrained optimization problem;
- one sums the effects of all wheel units via Eq. (13.2), which results in a driving force and torque on the platform;
- one integrates this over a time interval, to have a prediction of the resulting motion;

¹⁷This set is not exhaustive, yet.

- if that prediction satisfies the task requirements “well enough”, one applies the computed trajectories to the real wheels;
- when it is not good enough, one increments the feedforward trajectory in one or more of the wheel units, based on the known trends between changes in trajectory specification and changes in the monitored task quality outputs.

Probably the simplest control mode is that of “*all wheel units are pushing and are well-aligned*”. Since this mode is comparable to driving a traditional car, the above-mentioned *open-loop selection* would boil down to something of the following kind:

- to make the platform change the radius of a “circular motion”, all wheel units get the same *desired increment* in *direction* of their platform force F_i^p ;
- to make the platform increase its “average speed”, all wheel units get the same *desired increment* in *magnitude* of their platform force F_i^p .

Such control approach (based on discrete increments on the inputs together with a horizon-based monitoring/prediction of the resulting open loop trajectory) does not fit very well with the traditional work house of feedback control, namely the [PID controller](#), but rather with something like [sliding mode control](#), or [ABAG](#) control, [52].

Magic numbers:

- (TODO)

13.7.11 Navigation: topological motion specification and control

The control level “above” that of the platform *motion* is that of platform *navigation*, that is, to decide how to move the platform through an *environment* as a series of *areas* (represented by semantic tags or “*waypoints*”) to pass. Figure 13.16 sketches the situation, in which a *map* of that environment exists, via which *semantic motion constraints* are indicated; the “legends” that is used in the Figure is that of the mainstream outdoors road signs. At that level of abstraction, a “path” consists of the semantic tags of the “traffic” primitives on the map, together with some sort of *progress measure* that fits to that chosen level of abstraction.

Magic numbers:

- (TODO)

13.7.12 Information architecture

Figure 6.18 shows the graph of relations that connect the core components in the *model* of any task specification, at any of the levels of abstraction discussed in this document (mission, navigation, platform motion, 2WD unit motion, and wheel motion, Sec. 13.7.2).

The following paragraphs provide a reference software architecture for the relevant concurrent mobile robot activities of wheel control, 2WD unit control, platform motion control, and navigation control, together with their hardware drivers and behaviour monitors. (TODO)

Magic numbers:

- (TODO)

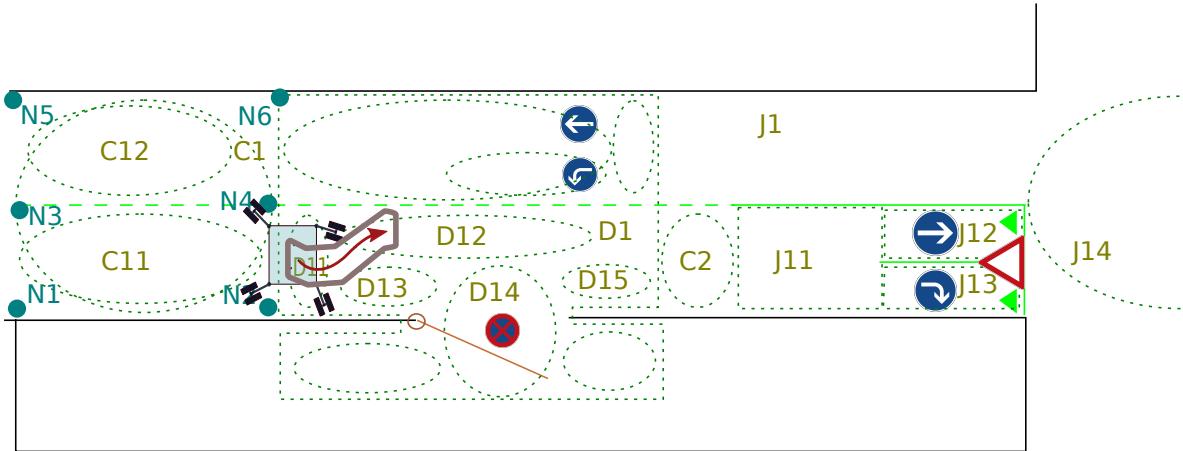


Figure 13.16: In *cruising mode* (Sec. 13.7.3), the navigation controller has to solve the problem of generating a “tube” trajectory for the platform, taking into account the semantic primitives in the map. The example above show a possible “tube” to bring the platform from the semantic area “D11” to the semantic area “D12”.

13.8 Common software representations in robotics

Section 12.3 gives an overview of common standardized “host languages” that can (hence) also be used to model **Coordinates** representations. But it is common to find a digital data representation format (meta-model) dedicated to a **specific framework** or communication middleware (e.g., [CORBA](#), [DDS](#)); in the latter case, the digital representation instance is also called *Communication Object*.

13.8.1 ROS Messages

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

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

The need of having a different data model for each communication mechanism provided reduces the degree of composability of the overall system, enforcing the component supplier to a premature choice. However, it is possible to compose message specifications from existing ones, such as services and action models are built starting from messages models. The expressivity of a ROS message description is limited with respect to other alternatives (e.g., ASN.1, JSON-Schema): it allows to specify different types for numerical representations, (e.g., `Float32`, `Float64` for floating-point values) but there is no support for constraints over a numerical value, nor specific padding and alignment information. Moreover, there is no built-in

enumeration values, which is usually solved with few workarounds¹⁸. 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 mechanism¹⁹).

13.8.2 RTT/Orocos typekits

The [RTT/Orocos typekits](#) for geometry are digital data representation models directly grounded in C++ code, which are necessary to enable sharing memory mechanisms of the RTT framework. However, it is possible to generate a typekit starting from a digital data representation model if a dedicated tool is supplied. For example, tools that generate a typekit starting from a ROS message definition exists.

13.8.3 SmartSoft Communication Object DSL

The [SmartSoft framework](#) provides a specific DSL based on the [Xtext](#) DSL tool of the [Eclipse Modeling Framework](#). It describes a digital data representation for the definition of primitive data types and composed data-structures. The DSL is independent of any middleware or programming language and provides grounding (through code generation) into different communication middlewares, including CORBA IDL, the message-based *Adaptive Communication Environment*²⁰ (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.

¹⁸An `UInt8` type with unique default value assigned for each enumeration item.

¹⁹It is possible to have other representations over ROS messages, e.g., JSON documents, by using a simple `std_msgs/String` message.

²⁰see <http://www.cs.wustl.edu/~schmidt/ACE.html>

Chapter 14

Holonic system architectures

The ambition of this Chapter is reflected in the introduction of the **7C's moniker**, combining the “5Cs” *component design* requirements with the two system design *mechanisms* of:

- *composability*: how to make **components** that are ready to be composed into a bigger system?
- *compositionality*: how to make a **system** out of such components, in such a way that the result offers explainable and predictable behaviour and performance?

The composition of the “5Cs” with the other patterns, is one of this document’s answers to the above-mentioned “*how?*” questions. The other answer is to give all **digital resources** (the *data*, *functions*, and *control flows* for activities, streams, components,...) their own **symbolic metadata**, including their own *model* and (the pointers to) the *meta models* needed to use these models. Indeed, having **models available in executable code**¹ is a necessary condition to support the **runtime** creation, configuration, discovery and introspection of digital resources. This is in turn necessary to allow **explainability** of runtime decisions. And this document’s hypothesis is that runtime explainability is, in turn, a necessary condition for **system resilience**. Ideally, this result must be achieved:

- by the *robots themselves* without human *intervention* but still allowing human *interaction* and *dialogue*.
- within any *architectural context*: different processes on one computer, different computers in a **LAN**, or different computers in a **WAN** or the **Internet**.
- with different component owners and development teams that share no software, but only meta data standards.

14.1 System design process

A system design process has major phases, and iterations over (part of) them is common practice.

¹or, alternatively, **from** the executable code, via “queries” to a “model database”.

14.1.1 Design phases

The design paradigm of this document identifies the following phases as necessary to design an application, and the first one as *the* essential starting point:

- to identify which **tasks** must be realised.
- which *activities* are available for this realisation.
- to identify which *resources* are **owned** by which of these activities (and, hence, can be re-configured by them, at runtime).
- to define which *state* of these resources must be *shared* with other *activities*.
- to define about which data sharing mechanism to use, **transactions** or **streams**.
- to create the software *event loops* to coordinate the execution of the algorithms that provide the *behaviour* of activities and their interactions, including the mechanisms about *how* to share state.
- to map these event loops onto an appropriate number of *computers*, *processes* and *threads*, to exploit the performance of the computational cores and the communication networks provided by the hardware architecture.

A good system architecture provides hooks for **system-level trade-offs** in how to **configure** the above-mentioned activities into **components**, in such a way that the composite system realises the **compositionality** property: if the behaviour of the components is predictable, the behaviour of the system should be predictable too. The underlying **design hypothesis** in this document's paradigm is that the more *composable* the components in a system are, the more *compositional*, that is, *predictable*, the behaviour of the system becomes.

Two crucial decisions system developers must make at design time is to identify: (i) which **decision making** is needed, and (ii) which components are assigned **ownership** of each decision. Also here, the above-mentioned “**4C**” aspects of a component’s behaviour help a lot to come to the “right” system design trade-offs.

14.1.2 Natural hierarchies for decision-making in systems

Because hierarchy is a major approach to deal with **complexity**, the key **meta meta design challenge** for system developers is to introduce as many *hierarchical* structures into a system architecture as possible, without compromising flexibility, [136, 166]. The **holon**² is one architectural design pattern that emphasizes the following aspects of *hierarchy*, in the context of a “resource” activity that is “*coordinated*” (but not *commanded*) by a “higher” activity:

- **ownership of resources**: a higher level in the hierarchy is “in command” of a lower level, in the sense that the latter (*re)configures itself* based on the suggestions or advice of the former.

For example, the activity that decides which torques to apply to a robot’s actuators is the one that *owns* (the software interaction with) the actuators. That is, it is the only one to interact with the motors. But an “higher-order activity” can request to use the robot’s actuators in a task it is executing, and the “mode” of control that it wants to use. The “lower-order activity” should use that information from the “higher-order activity” to configure its actuator control functionality accordingly.

²On mainstream, over-simplified incarnations, a holon structure is often also called *agent*, or *digital twin*. The simplification often comes from neglecting the hierarchical, ownership, autonomy and configuration aspects.

- **co-responsibility of decision making about these resources:** the higher level in the hierarchy *consults* the lower level about every decision making of the latter's (change in) behaviour.

For example, the higher-order task execution activity can first inform the actuator-owning activity about what kind of motion it wants to make, to find out what would be the “optimal” configuration with which the actuator-owning activity can execute that motion. The “higher-order activity” should use that information from the “lower-order activity” to decide about which task execution behaviour it will actually select.

This document’s **holon approach** advocates to assign each such ownership to one and only one holon at the same time, applying both the **mediator** and **DOM** patterns for decision coordination and configuration in hierarchically structured information dependencies. The underlying design hypothesis is that clear ownership, together with clear dependencies, allows to create sub-systems that can behave as full-blown systems themselves, always ready to form larger systems with other holons, all by themselves. The holon structure relies on a *information hierarchy* to structure its *decisions*, but it also allows “loosely coupled” components to interact heterarchically [95, 78],³ via *streams*, to exchange the *data* that supports their decisions.

The **DOM** and holon meta models serve symmetric architectural purposes: a holon can have an internal hierarchical DOM architecture of decision making, and each component in a DOM model can be a holon that interacts heterarchically with other holons that are beyond the decision making scope of this holon’s internal DOM.

This Chapter only introduces the *meta meta level* concepts about **architectures**, but the market needs digital⁴ platforms that can be offered to paying customers. So, application system developers still have to add

- information architectures to structure the **knowledge** and **data** in the customers’ applications.
- hardware architectures with *devices* and *communications* between them.
- software architectures of **components** large and small (that is, algorithms, activities, streams and sub-systems), with their (often *dynamically changing*) **peer-to-peer** interactions.

14.1.3 Hierarchy in knowledge versus hierarchy in architecture

Knowledge representations are powerful because they contain many **graph-structured associations** between entities and relations of the domain that is modelled. Many of these knowledge relations have a **tree-structured skeleton**, representing the “key” associations in that domain, in a hierarchical way (that is, with *parent-children* connections). An example of such a tree skeleton is the representation of the **built environment**, where the the knowledge relations are of the following types:

- **containment hierarchy:** the root-level (or highest ancestor) “container” is the *world*, with *continents* and *countries* as relevant branches (or descendants). Dependent on

³Koestler’s description can be summarized as follows: *holons are self-reliant units that possess a degree of independence and can handle contingencies without asking higher authorities for instructions (i.e., they have a degree of autonomy). These holons are also simultaneously subject to control from one or more of these higher authorities. The first property ensures that holons are stable forms that are able to withstand disturbances, while the latter property signifies that they are intermediate forms, providing a context for the proper functionality for the larger whole.*

⁴Or **business**, or **computational**.

the desired/required levels of abstraction and resolution, the tree branches further via containment relations of *country*, *province*, *city*, *street*, *building*, and *floor*.

- the *floor* is a natural end point (“leaf”) of the tree structure, because the primitives **contained-in** a floor—*corridors*, *rooms*, and *doors*—are interconnected with graph structures and not tree structures.

System architectures have a complementary purpose to knowledge relations: the latter are pieces of information (hence, without any *behaviour* on their own). but it is up to the (eventually, software) *components* in the system architecture to realise the system’s *behaviour*, taking into account the available information. The *decisions* that are made in the components are based on (i) the knowledge relations, and (ii) the *behavioural state* of the system at the moment of decision making.

Of course, these state variables are linked via the knowledge relations, but the latter’s graph structure is not the best way to structure the *dependencies* between *components*: the latter need a context of execution, in which a component hierarchy is the most predictable way to allow efficient *decision making* about each of the components’ behaviour.

This document adopts the following **meta model of hierarchical dependencies** between software components [95, 96, 142]:

- the values of properties at a higher level component in the hierarchy *trickle down* to components at lower levels in the tree.

For example, if a building is sold, the ownership of everything inside that building changes. If one uses the same template model for different houses, parameters like number of floors and heights of ceilings should only be changed at the top level. Or if a corridor is given fire or security doors, the usage of all rooms inside is impacted.

- a behavioural “event” in a lower level component *propagates up* towards higher level components, but also to *siblings* in the tree.

For example, if the door of a room is locked, which other parts of the building should be informed? At what level will the city’s fire brigade be called when a fire has started in a particular room? Or where is the decision taken to let a robot change its planned navigation task through a building when it detects “obstacles” on its way?

14.1.4 Architecture: holonic responsibility structure

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 [54, 69, 78, 110, 136, 153, 152, 150]. The starting point behind “holonic system architectures” is to design a system in such a way that every component in that system:

- is ready to be composed into a larger **system-of-systems**, sooner or later.
- can **decide for itself** *that, when, why, where* and *how* it becomes part of a larger system, or break away from it.
- can **take into account** the inputs that other holons provide, in a **peer-to-peer** fashion: the input is considered to be **advice**, not a *command*, and (the advice in) any interaction between two holons is bi-directional.
- gets **clear and unique responsibilities** to make decisions for **identified** parts of the system’s task execution.

The major responsibilities of application-centred system architects (compared to more technology-centred component developers) are:

- to provide a design that can take **responsibility** for the delivery of **system-level Quality of Service**, and that can **honour a contract** about the realised level of the service. The system architects focus on those specific requirements that only show up at the application level: **autonomy**, **safety**,⁵ **security**, or **resilience**.
- to realise resilience against **software erosion**: this is not a physical phenomenon, because the software does not actually decay, but rather a social phenomenon. Indeed, software projects can suffer from a lack of developers to remain responsive to the changing environment in which the software must work.
- to have an *information architecture* that **does not depend on specific choices** made in the *software and hardware architectures* that happened to be the ones available the first time the information architecture was implemented. In particular, the *configuration values* should be changeable for each actual runtime that is deployed from the software architecture.

Individual components can never *guarantee* these requirements, but can easily *undermine* them by not being able to adapt their own behaviour to the overall behaviour expected at the system level. Hence, this document designs components as “holons”, around one or more instances of the **mediator pattern**, designed to support (but not *to guarantee...*) **single point of decision making** over each of the “resources” the holon is responsible for. Each holon is able to base that decision making on the formal reasoning it can perform with the **knowledge/information relations** it has about (i) its own **internal** behaviour, as well as (ii) how its **externally** visible behaviour influences the behaviour of the other holons it interacts with. These are (necessary but not sufficient) conditions to realise the system’s *predictability*⁶ under interactions with any type of **external** system.

14.1.5 The 7Cs: 5C components, plus their Composability and Compositionality

This document advocates a way to design activities and interactions such that the architected system keeps the **compositionality** property whenever it is built from components that have the **composability** property. The objective is that knowing the component architecture, together with the *data sheets* of the behaviour of the activities in, and the interactions between, the components, suffices **to predict** the behaviour of the system. In other words, a well-done composition architecture behaves itself as a new “primitive” component in a larger system.

This document *designs* the mechanisms of *function*, *algorithm*, *activity*, *interaction* and *component* with the *same* composability-compositionality design drivers. When done right, this allows

- to let a system **explain** all of its decisions, at all times.
- **to instrument** the system with monitoring functionalities.
- **to reconfigure** all of the above, even at runtime.

The *differences* that *need* to be introduced between the five mechanisms, cover the variety in **deployment** constraints. That is, the deployment of information architecture building blocks onto software architecture building blocks, and the deployment of software building blocks

⁵ “Safety is not defined by the absence of accidents, but by the presence of capacity.” Todd Conklin, senior advisor at Los Alamos National Laboratory, 2016.

⁶But not necessarily its *performance* or *robustness*.

onto the building blocks provided by computer hardware (CPU cores, networks, memory) and by the operating systems (processes, threads, system calls, inter-process communication, shared memory regions).

14.1.6 Policy: component model as documentation, specification, or contract

This Chapter and the following introduce several formal models, representing various aspects of system architectures. The **data sheet** of a **component** in the system contains a list of such models, being the externally “visible” representation of the **structure and behaviour** of that component, [15, 91]. At system level, an extra higher-order “tag” is added to these models, representing the purpose of each visible model. The following three purposes are a minimum version of a possibly much longer list:

- **documentation**: the model is meant to be used by human developers only, to provide them with information about the component.
- **specification**: the model comes with an identified meta model, so that a “consuming” component can parse the model and check its meaning with formal methods.
- **contract**: the model and meta model conform to an industry standard, and their contents form part of a liability and **Quality of Service** (QoS) agreement between the organisations that “provide” and “consume” the component.

At all three levels, the functional parts in the models can be extended with *performance measures*, such as latency, energy consumption, safety, etc.

14.1.7 System-level properties of architectures

A **resilient** system has an architecture of so-called **holons**, that is, components that:

- **remain operational** under *any* (lack of) interaction with peer systems (that can come and go dynamically).
- can always **decide for themselves** when and how to switch to what kind of **graceful degradation** behavioural state(s).
- can **explain** their decisions, to whatever other **peer** holon that is interested in the explanation,
- can build up **trust**⁷ in their mutual interactions with *identified* holon peers.

The **sustainability** of a system, however, depends on more than just the above-mentioned technical aspects. Some very important factors are:

- the **eco-system interactions** (social, economic, ethical) of the **stakeholders** (humans, organisations, and markets) in an **application domain**.
- the **safety** that the system can offer, to itself, its users, and its environment.
- the **authentication & authorisation protocols**.
- (hence) the penetration and acceptance of **open standards**, first, and of **free and open-source software** (FOSS), second.

⁷ Trust between cyber-physical systems is not yet an established concept.

14.2 Holon architecture for heterarchical decision making in task execution

Systems are made available because they offer *capabilities* to realise **tasks** that add value for customers while making good use of the *resources* that the customers want to pay for. ([Robotic tasks](#) are specific instances of the more generic task concept in this Chapter.) From a system architecture point of view, the essential aspect of a task is the **ownership** that “someone” has to take about *every* task and *every* resource. It is also not uncommon that one system has sub-systems that can “more or less” be considered independently, with respect to their ownership of rather uncoupled task executions.

It is this context in which the concept of a [**holarchy**](#) (or *holonic architecture*) was born, [78, 136], first of all as a model of *resilient* organisational systems-of-systems, in a socio-economic context.

The previous Sections in this Chapter introduced composable models of the [**5C**](#) aspects of **configuration** and **coordination**, using the *hierarchical* meta meta models of DOM models and [**mediators**](#). Holons care about the two other aspects of composition in the [**5C**](#) meta model, namely the [**heterarchical**](#), **communication** between components, and of the **computation** inside components.

The core idea of the “holon” as being a “standalone” or “sub-system” architecture, is that it *does* introduce a clear *hierarchy* in these by nature heterarchical activities, adopting the following **architectural policy**: **only** the top-level component in the sub-system mediator or DOM model hierarchy

- **owns** the interfaces for communication and computational behaviour towards components in the “outside world”. That is, that mediator is the **only component** in the “inside” architecture that **the outside world should interact with**.
- **configures** the interactions of all “inside” components with the outside world. That is, one of the decisions owned by the mediator is “to open up” its [**information hiding**](#) outside interface, and **to make internal components visible**, to selected outside components and during a selected set of interactions.

In other words, a holon sub-system is a good practice alternative to a “monolithic” component with opaque interfaces, *if* that holon is the composition of a set of components that are really excellently represented by **one single mediator at the top**. Whether or not a set of components can, or must, be composed into a *holon* is *not* a property of these components, but of the system context in which they are deployed to realise particular tasks.

14.2.1 Resilient, stable, robust sub-systems

This document uses the terms [**stable**](#) or [**robust**](#) as synonyms for *resilient*. [**Anti-fragile**](#) systems are outside of the scope of this document, for now; that is, the document focuses on [**known unknowns**](#) only. An alternative conceptual classification that is relevant to this Chapter is that of the [**Cynefin framework**](#): this document deals with the *clear* (“best practices”) and **complicated** cases (“good practices”); its design concepts support the structuring of **complex** cases (“emergent practices”), but not of *chaotic* (“novel practices”) or *disorder* cases (“no practices”).

This document redefines the mereo-topological interpretation of these concepts in the context of the architecture of engineering systems:

*A system has a **resilient** architecture if its behaviour is **explainable**, and remains so under any change in the behaviour of any of the systems it interacts with, as well as any change in the topology of its interactions.*

14.2.2 Dependencies in architectures: hierarchy, heterarchy and holarchy

(TODO: hierarchy for knowledge and context, heterarchy for command and control; holarchy: architecture with, both, explicit allocation and responsibility about ownership of everything, and **theory of mind** awareness of each holon's individual role in an **society of peers**.)

14.2.3 Holonic practices to use DOM models: advice, but don't command

Hierarchy is good for system design, because the predictability, liability and monitoring of decision making increases when the **context** in which decisions are made is hierarchical. However, hierarchy is also a too easy and tempting structure to adopt without thorough motivation, so developers must be careful:

- to support the *good parts* of hierarchy, that is, the **tree structure** that it brings in the **information** required for **decision making in a context**: parameters in multiple components are linked together, and that linking takes place via a tree structure. This implies that a lower level activity can provide information to a higher level activity. For example, if a torque controller component detects **overheating** of one of the actuators in the robot, the hierarchical context model helps to decide which other control levels should be informed about this condition. And that decision can depend on the “status information” that the torque controller makes available about the problem.
- to avoid the *bad parts* of hierarchy, that is, to **hard code decision making** into a hierarchy of components, whenever (i) such a strict structure does not reflect the requirements of the application, or (ii) the components loose their **autonomy** in making decisions for themselves. For example, in the above-mentioned motion control context, one should not assume at a high level of the hierarchy that:
 - all actuators have the same overheating limits.
 - all actuator control activities must react to the overheating of one actuator in the same way.
 - any configuration setting decided in that high level activity is immediately and deterministically applied in the lower level activities.

A major cause of why hierarchy is good or bad, lies in (i) the **amount of interactions** that is needed between components to reach a decision, together with (ii) the **time it takes** to adapt the behaviour of every component impacted by the decision.

The mentioned autonomy aspect is the core differentiator between “normal” **multi-agent** architectures, and **holonic** architectures:

- in a holonic design, each component interprets the DOM parameter configuration and event handling that it is connected to in a DOM, as an **advice** from the other DOM-connected components to influence its own behaviour. But it does not interpret them as *commands* that can not be changed or overruled. So, with that advice, it *investigates* the impact on its own behaviour of each piece of dependency and influence that comes from other components, but it remains *fully* responsible for the decision it makes based on all the received information.

- a holonic component is designed to be influenced and advised by, *and* has influence itself on and advice for, “same-level” *peers*, but also “higher-level” *mediators* as well as “lower-level” *sub-ordinates*.

Such holonic form of autonomy is difficult “to get right”, but society and nature have proven that it is essential to realise **resilient** systems. Human society has several realisations of both good practices for using, or *not* using, hierarchy. For example, the strict hierarchies in the military and multi-national corporations, where generals or CEOs can not interact productively with all soldiers or workers in their organisation. Such constructive interaction *are* possible and constructive in the **hierarchical**, **flat** structures of **startup companies** or **grassroots social movements**.

14.2.4 Explainability: causal driver for robustness, resilience, anti-fragility

Explainability is the first step in a hierarchy of system **behavioural resilience metrics**; **adaptability**, **predictability** and **dependability** are next. And **guaranteeability** is the holy grail. A **working hypothesis** of this document is that the explainability of an architectural design of a system is proportional to the level of **knowledge concentration** that the designer can achieve in providing one **unique mediator** component that makes the decisions about how to coordinate its own system-level behaviour with that of all of its internal subsystems, as well as with all of the external peer systems. In other words, explainability can only be achieved when *all* decisions the system makes to adapt its behaviour to its context, are made in one single place, based on one consistent combination of (offline) *knowledge relations* and (online) *world model data*. Of course, trying to apply this design driver blindly, by **centralising** decision making, has time and again proven not to be a good approach; the more resilient architectures have “holarchies” of resilient sub-systems, with the “right” loose coupling of their behaviours and, especially, their decision making.

The system architects’ responsibility remains huge, because computer reasoning can’t⁸ deliver the **wisdom** of deciding the subsystem boundaries where robustness, explainability or resilience can be optimized.

14.2.5 Best practice: be conservative in what you send, be liberal in what you accept

(TODO: **robustness principle**: .)

14.3 Autonomy as explainable decision making

Like most other terms in this Chapter (“architecture”, “safety”, “system of systems”,...) *autonomy* has been given several different definitions, although none of them is sufficiently constructive to be used as a **refutable** design driver. This Section introduces such a refutable design for autonomy, by associating autonomy with **explainable** decision making. The underlying (refutable) hypothesis is that a system can only be called “autonomous” if it can explain *why* it makes its decisions.

⁸This is a very **refutable** claim. Readers are kindly invited to provide such a refutation.

14.3.1 Endsley's five levels of system autonomy

Already in 1987, [Endsley](#) [44] identified **five levels of autonomy**. Albeit in the context of computer support for fighter pilots, her autonomy levels are relevant to all cyber-physical and robotics systems too; the domain of [autonomous cars](#), has taken Endsley's classification as its inspiration for its own [autonomy levels](#) specification.

Level	Description
5	Computer makes decisions, with the human completely out of the loop.
4	Computer makes recommendations to the human which it will carry out unless the human vetos.
3	Computer makes recommendations to the human which it will carry out if the human concurs.
2	Computer makes recommendations to the human which he may choose to act on or not.
1	Computer offers no assistance, human makes all decisions & actions.

14.3.2 Sheridan's ten levels of system autonomy

[Sheridan](#) refined Endley's scale to **ten levels of autonomy** [106], Table 14.1. Endsley [46] refined this scale further.

Level	Description
10	Computer does everything autonomously, ignores human.
9	Computer informs human only when it sees fit.
8	Computer informs human only if asked.
7	Computer executes automatically, and informs human when necessary.
6	Computer allows human restricted time to veto before starting action.
5	Computer executes suggested action if human approves.
4	Computer suggests one single alternative.
3	Computer narrows alternatives down to a few.
2	Computer offers a complete set of decision and action alternatives.
1	Computer offers no assistance, human makes all decisions & actions.

Table 14.1: Sheridan's ten levels of system autonomy [106].

14.3.3 Explanation levels for autonomous decision making

Endsley's and Sheridan's scope was limited to the interaction between one *single* machine and one *single* human. Modern robotic and cyber-physical systems must extend this scope to *systems-of-systems* contexts, with *multiple* agents, multiple tasks, multiple resources, multiple vendors, multiple regulators, and multiple machines. This document introduces a definition of “autonomy levels”, Table 14.2, based on the **dialogue** with which a system **explains its decisions** to other agents, human as well as artificial. In other words, this scale is *constructive* (it says *how to measure* whether a level of autonomy is reached), while the scales of Endsley and Sheridan are *qualitative* (it is still only a human who can assess whether a system has reached a particular level of autonomy). The granularity of the levels is designed to allow incremental *step change* developments in autonomy, for very focused decision making

challenges. Note the huge technical challenges to go from “level 4” to “level 5”, and, especially, from “level 6” to “level 7”. These steps introduce two subsequent levels of **empathy**: (i) to reflect on one’s own actions and put them in the context of the **user** for whom a Task is being executed, and (ii) to create and maintain also the world models of **other systems**, and to reason in their place.

Level	Description
One system — One task — Self awareness	
1	What am I doing?
2	Why am I doing it?
3	How am I doing it,...
4	... and how well am I doing it,...
4b	... and why am I doing it this way?
4c	... and how do I decide to stop doing it?
One system — Multiple tasks — Situation awareness	
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 — Empathy	
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 14.2: Systems’ decision making explanation levels. They represent the various degrees to which systems can (i) perform self-diagnosis monitoring and Coordination, (ii) *explain* their autonomous decision making, and (iii) adapt in order to increase resiliency [90].

14.3.4 Role of meta models in horizontal and vertical task composition

A system’ decision making levels of Table 14.2 are agnostic to the scale of the system, so they are also relevant for any type of horizontal and vertical composition of **tasks**. If such a composition is constructed with knowledge-driven hybrid constrained optimization (KHCOP), the on-line solvers of such KHCOP problems have already answers to some of the questions in Table 14.2, because decisions are made in the Coordination state machines and Task progress is monitored semantically via constraint violation checks.

This is the *first level* of explainable decision making: every decision is associated with a set of constraint violations. The meta model of the system represents the associations between these constraints explicitly and symbolically. System designers must try to add also the *sec-*

ond level of explainable decision making, by the higher-order associations that represent the **reasons why** these constraints have been configured to be the ones to monitor. Task compositions are a major source of this higher-order association: every composition is introduced for a reason, and representing that reason explicitly and symbolically is the way forward. This document’s [Task-Situation-Resource](#) meta model already links three complementary sources of such higher-order associations: the requirements of the task, the situational context of the environment, the limitations of the resources, and the knowledge relations used in the skill that composes them together.

(TODO: examples.)

14.3.5 System safety as explainable decision making

(TODO: explain why safety is a system-level aspect, and composes parts of the HCOPs in all the current Tasks. The role of the LCSM as the model for an independent “[safety PLC](#)”.)

14.3.6 Best practice: safety PLC

(TODO: third-party monitoring, detection and reaction to solve first-party’s unsafe conditions.)

14.3.7 Best practice: active safety behaviour

(TODO: first-party monitoring, detection and reaction to prevent third-party’s unsafe conditions.)

14.3.8 System security as explainable decision making

Security is a system and application level aspect, because (i) no middleware can be trusted, and (ii) no task specification can be trusted. Hence, and in addition to the best practice in encrypting all communications, the application has to engage in *dialogues* with all parties that provide or consume data and task specifications. The dialogue consists of back-and-forth questions and answers with a large enough variety in encrypted keys to exclude [man-in-the-middle attack](#) problems. In this document’s broader context of knowledge-driven systems engineering, these dialogues do not impose a lot of extra overhead because a lot of messages are already exchanged for other “non-functional” purposes, such as heartbeats, resource mediation, and task coordination.

(TODO: lot more concreteness.)

14.4 Data sheets: meta data for digital resources

This document strives for software components that are aware that they have a model and a set of meta models, and that can use them in “dialogues” with other components, via one of its ports. Conceptually, the contents of a data sheet is straightforward, namely the enumeration of all models that represent the behaviour of the component, as visible by other components:

- **Semantic_ID**: a (set of) data structures (strings, numbers, binary blobs,...) that identify the component uniquely.
- **ports**: the *data structures* available via each port, the *interaction protocols* that the port follows, and the *constraints* that may occur on the behaviour of different ports.
- **resources**: a symbolic model of the resources the component has available to provide its services to others.
- **operational range**: constraints on “how good” the component can realise its behaviour.

(TODO: many more models must still be included: coordination, data exchange, configuration.)

14.4.1 Data architecture

(TODO: modelling data structures so that linear binary buffers can be generated. Composition into tables and record batches (using terminology from the Apache Arrow project), and into IPC messages and files. Connection to data flow dependency graphs, and how that data structure is the basis for a data flow runtime. Special cases of “data” being symbolic or logical are the Petri Net graphs and FSM graphs; these specific data flows are the basis for a Coordination architecture.)

14.4.2 Function architecture

(TODO:)

14.4.3 Coordination architecture

(TODO: explain how **FSM** and **events**, Petri Nets and tokens, and algorithms and flags, are composed to realise the architecture of all coordinations in a system, synchronously and asynchronously, in-memory and distributed. Constraint: one FSM contains one or more PNs.)

14.4.4 Information architecture

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

- **domain models**: the property graphs of the various pieces of domain knowledge the application builds upon. For example, the relevant physical units and mathematical solvers; the rules of traffic; etc.

- **application models:** the property graphs representing the knowledge needed about the application: task requirements, social and technical constraints, Quality of Service, introspection and self-diagnosis models, etc.
- **provenance models:** information about how data, entities and relations have been created, who has *ownership* of them and has the responsibility to make decisions about what *to do* with them, including providing them to other “organisations”, etc. Two established meta models for provenance are [Dublin Core](#) metadata, and the [PROV](#) models from the [World Wide Web Consortium](#).
- **abstract data types:** from all of the above, select those entities and relations that the application has to create abstract data types for. This includes models for the data structures, communication messages, operators, etc.
- **activities**, with [event loops](#) to serve all [5C](#) responsibilities, including managing [concurrency](#) of algorithms such that state changes in abstract data types remain consistent.
- **mediators:** each *shared resource* (communication stream, task, space, algorithm, world model, CPU, robot hardware,...) needs a mediator activity to centralize the configuration and coordination decision making about that resource. In some contexts, also all [CRUD](#) operations (Create, Read, Update, Delete) on the shared resource are run through the mediator.
- **solvers:** the algorithms that will realise all the activities behaviour need to be modeled.
- **contexts:** a major design effort in making an information architecture is to make sure that every part in it (activity, solver, function,...) is always active within the correct *context*, that is, the “state” of the system that it is not responsible for itself but relies on to be correctly filled in by other parts of the system.
- **Life Cycle State Machines:** each activity needs its own LCSM to manage the resources it uses, and each task and each event loop need a separate LCSM too. One of the more difficult design challenges is to make the life cycle of the *contexts* explicit for the whole system, and for the whole duration of the system’s life.

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

14.4.5 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.

14.4.6 Hardware architecture

The hardware architects' role in this document is focused on how to design the information and software models of all hardware that the system must bring under computer control; that is, *deploying* software onto hardware, taking the decisions about how to execute the software architecture on computational hardware, communication buses, 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 **All to go, One to stop** policies with them.)

14.4.7 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.)

Bibliography

- [1] ACKOFF, R. L. Towards a system of systems concepts. *Management Science* 17, 11 (1971), 661–671.
- [2] ACKOFF, R. L. From data to wisdom. *Journal of Applied Systems Analysis* 16 (1989), 3–9.
- [3] AERTBELIËN, E., AND DE SCHUTTER, J. eTaSL/eTC: A constraint-based task specification language and robot controller using expression graphs. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems* (2014), 1540–1546.
- [4] AL BAHRA, S. Nonblocking algorithms and scalable multicore programming. *Communications of the ACM* 58, 7 (2013), 50–61.
- [5] ALAMI, R., CHATILA, R., FLEURY, R., GHALLAB, M., AND INGRAND, F. An architecture for autonomy. *The International Journal of Robotics Research* 17, 4 (1998), 315–337.
- [6] ALEXANDER, C., ISHIKAWA, S., AND SILVERSTEIN, M. *A pattern language: towns, buildings, construction*. Oxford University Press, 1977.
- [7] ANGELES, J. *Rational Kinematics*. Springer, 1988.
- [8] ANGLES, R., ARENAS, M., BARCELÓ, P., HOGAN, A., REUTTER, J., AND VRGOČ, D. Foundations of modern graph query languages. *ACM Computing Surveys* 50, 5 (2017), 1–40.
- [9] BALL, R. S. *Theory of screws: a study in the dynamics of a rigid body*. Hodges, Foster and Co, Dublin, Ireland, 1876.
- [10] BARTELS, G., KRESSE, I., AND BEETZ, M. Constraint-based movement representation grounded in geometric features. In *IEEE-RAS International Conference on Humanoid Robots* (2013).
- [11] BATEMAN, J., AND FARRAR, S. Modelling Models of Robot Navigation Using Formal Spatial Ontology In *Spatial Cognition IV* (2005), 366–389.
- [12] BAUMANN, P. A general conceptual framework for multi-dimensional spatio-temporal data sets *Environmental Modelling and Software* 143, (2021), 105096:1–21.
- [13] BAUMGARTE, J. W. Stabilization of constraints and integrals of motion in dynamical systems. *Computer Methods in Applied Mechanics and Engineering* 1, 1 (1972), 1–16.

- [14] BEESON, P., MACMAHON, M., MODAYIL, J., MURARKA, A., KUIPERS, B., AND STANKIEWICZ, B. Integrating Multiple Representations of Spatial Knowledge for Mapping, Navigation, and Communication. *AAAI Spring Symposium Series, Interaction Challenges for Intelligent Assistants*, (2007), 1–9.
- [15] BENSALEM, S., DE SILVA, L., INGRAND, F. AND YAN, R., A Verifiable and Correct-by-Construction Controller for Robot Functional Levels *Journal of Software Engineering in Robotics*, 2:1(2011), 1–19.
- [16] BEUTER, N., SWADZBA, A., KUMMERT, F., AND WACHSMUTH, S., Using Articulated Scene Models for Dynamic 3D Scene Analysis in Vista Spaces. *3D Research*, 3(4), 2010, 1–13.
- [17] BÉZIVIN, J. On the unification power of models. *Software and Systems Modeling* 4, 2 (2005), 171–188.
- [18] BILJECKI, F., LEDOUX, H., AND STOTER, J. An improved LOD specification for 3D building models. *Computers, Environment, and Urban Systems*, 59 (2016), 25–37.
- [19] BISHOP, C. M. *Pattern Recognition and Machine Learning*. Springer, 2006.
- [20] 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.
- [21] BORGO, S. Euclidean and mereological qualitative spaces: a study of SCC and DCC. In *International Joint Conference on Artificial Intelligence* (2009), pp. 708–713.
- [22] BORST, P., AKKERMANS, H., AND TOP, J. Engineering ontologies. *International Journal on Human-Computer Studies* 46 (1997), 365–406.
- [23] BOTTEMA, O., AND ROTH, B. *Theoretical Kinematics*. Dover Publications, 1990.
- [24] BRAUN, L. On adaptive control systems. *IRE Transactions on Automatic Control* 4, 2 (1959), 30–42.
- [25] BRINCH HANSEN, P. The nucleus of a multiprogramming system. *Communications of the ACM* 13, 4 (1970), 238–250.
- [26] BURKE, W. L. *Applied differential geometry*. Cambridge University Press, 1992.
- [27] CECCARELLI, M. Screw axis defined by Giulio Mozzi in 1763. In *9th World Congress IFToMM* (1995), pp. 3187–3190.
- [28] CHASLES, M. Note sur les propriétés générales du système de deux corps semblables entr'eux et placés d'une manière quelconque dans l'espace; et sur le déplacement fini ou infiniment petit d'un corps solide libre. *Bulletin des Sciences Mathématiques, Astronomiques, Physiques et Chimiques* 14 (1830), 321–326.
- [29] 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), pp. 54–68.

- [30] CHEN, P. P.-S. The entity-relationship model—Toward a unified view of data. *ACM Transactions on Database Systems* 1, 1 (1976), 9–36.
- [31] COX, I. J., AND LEONARD, J. J. Modeling a dynamic environment using a Bayesian multiple hypothesis approach. *Artificial Intelligence* 66 (1994), 311–344.
- [32] CRAMPIN, M., AND PIRANI, F. A. E. *Applicable Differential Geometry*, 3rd ed. Cambridge University Press, 1988.
- [33] D'ALEMBERT, J. L. R. *Traité de Dynamique*. 1742.
- [34] 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.
- [35] DING, C., MAO, Y., WANG, W. AND BAUMANN, M. Driving Situation Awareness in Transport Operations. *Computational Intelligence for Traffic and Mobility*, (2013), 37–56.
- [36] DE MAUPERTUIS, P. L. M. Accord de différentes lois de la nature. In *Oeuvres, Tome IV*. Olms, Hildesheim, Germany, 1768.
- [37] 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.
- [38] DEREMER, F., AND KRON, H. Programming-in-the large versus programming-in-the-small. In *Proceedings of the international conference on Reliable software* (1975), pp. 114–121.
- [39] DIJKSTRA, E. W. Letters to the editor: go to statement considered harmful. 147–148.
- [40] DONDRUP, C., BELLOTTO, N., HANHEIDE, M., EDER, K., AND LEONARDS, U. A computational model of human-robot spatial interactions based on a qualitative trajectory calculus. *Robotics* 4 (2015), 63–102.
- [41] EDDY, B. G., AND TAYLOR, D. R. F. Exploring the concept of cybergartography using the holonic tenets of integral theory. In *Modern Cartography Series* 4, (2005), 35–60.
- [42] EGENHOFER, MAX J., AND FRANZOSA, ROBERT D. Point-set topological spatial relations. In *International Journal of Geographical Information System* 5, (1991), 161–174.
- [43] EHRIG, H., ERMEL, C., GOLAS, U., AND HERMANN, F. *Graph and Model Transformation. General Framework and Applications*. Monographs in Theoretical Computer Science. Springer, 2015.
- [44] ENDSLEY, M. R. The application of human factors to the development of expert systems for advanced cockpits. (1987), 1388–1392.

- [45] ENDSLEY, M. R. Toward a theory of situation awareness in dynamic systems. *Human Factors* 37, 1 (1995), 32–64.
- [46] ENDSLEY, M. R. From here to autonomy: Lessons learned from human–automation research. *Human Factors* 59, 1 (2017), 5–27.
- [47] ENGELL, S. Feedback control for optimal process operation. *Journal of Process Control* 17 (2007), 203–209.
- [48] ENGELS, G., LEWERENTZ, C., SCHÄFER, W., SCHÜRR, A., AND WESTFECHEL, B., Eds. *Graph Transformations and Model-Driven Engineering*. Springer, 2010.
- [49] FEATHERSTONE, R. *Rigid Body Dynamics Algorithms*. Springer, 2008.
- [50] FLORIDI, L. The method of levels of abstraction. *Minds & Machines* 18 (2008), 303–329.
- [51] FLORIDI, L. *The philosophy of information*. Oxford University Press, 2011.
- [52] FRANCHI, A., AND MALLET, A. Adaptive closed-loop speed control of BLDC motors with applications to multi-rotor aerial vehicles. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Singapore, 2017), pp. 5203–5208.
- [53] FRANKEL, T. *The Geometry of Physics*. Cambridge University Press, Cambridge, England, 1996.
- [54] GAYER, J. An outline of hierarchical systems theory and its role in Philosophy. *Dialectica* 23:3/4 (1969), 177–188.
- [55] GAUSS, K. F. Über ein neues allgemeines Grundgesetz der Mechanik. *Journal für die reine und angewandte Mathematik* 4 (1829), 232–235.
- [56] GIBSON, C. G., AND HUNT, K. H. Geometry of screw systems—1. Screws: Genesis and geometry. *Mechanism and Machine Theory* 25, 1 (1990), 1–10.
- [57] GIBSON, J. J. *The ecological approach to visual perception*. Taylor & Francis, 1986.
- [58] GLANVILLE, R. The purpose of second-order cybernetics. *Kybernetes* 33, 9/10 (2004), 1379–1386.
- [59] 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.
- [60] GOLDSTEIN, H. *Classical mechanics*, 2nd ed. Addison-Wesley, 1980.
- [61] GOLUB, G., AND VAN LOAN, C. *Matrix Computations*. The Johns Hopkins University Press, 1989.
- [62] GOOD, I. J. A derivation of the probabilistic explanation of information. *Journal of the Royal Statistical Society (Series B)* 28 (1966), 578–581.
- [63] GRÖGER, G., AND PLÜMER, L. CityGML — Interoperable semantic 3D city models. *ISPRS Journal of Photogrammetry and Remote Sensing* 71 (2012), 12–33.

- [64] GRUBER, THOMAS R. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies* 43:5–6 (1995), 907–928.
- [65] HALFORD, G. S., WILSON, W. H., AND PHILLIPS, S. Relational knowledge: the foundation of higher cognition. *Trends in Cognitive Sciences* 14, 11 (2010), 497–505.
- [66] HAMILTON, W. R. On a general method in dynamics. *Philosophical Transactions of the Royal Society*, II (1834), 247–308.
- [67] HERLIHY, M. P. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* 13, 1 (1991), 124–149.
- [68] HILL, J. W., AND SWORD, A. J. Manipulation based on sensor-directed control: an integrated end effector and touch sensing system. In *17th Annual Human Factors Society Convention* (1973).
- [69] HOLVOET, T., WEYNS, D., AND VALCKENAERS, P. Delegate MAS patterns for large-scale distributed coordination and control applications.
- [70] HUNT, K. H. *Kinematic Geometry of Mechanisms*, 2nd ed. Oxford Science Publications, 1990.
- [71] IMIYA, A. A metric for spatial lines. *Pattern Recognition Letters* 17 (1996), 1265–1269.
- [72] JAYNES, E. T. *Probability Theory: The Logic of Science*. Cambridge University Press, 2003.
- [73] JOURDAIN, P. E. B. Note on an analogue of Gauss' Principle of least constraint. *Quaterly Journal of Pure and Applied Mathematics* 8L (1909).
- [74] KARGER, A. Geometry of the motion of robot manipulators. *Manuscripta Mathematica* 62 (1988), 115–126.
- [75] KARGER, A., AND NOVAK, J. *Space kinematics and Lie groups*. Gordon and Breach, 1985.
- [76] KENDALL, M. G., AND MORAN, P. A. P. *Geometrical Probability*. Charles Griffin & Company, 1963.
- [77] KIM, J. H., AND PEARL, J. A computational model for combined causal and diagnostic reasoning in inference systems. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence* (1983), pp. 190–193.
- [78] KOESTLER, A. *The Ghost in the Machine*. 1967.
- [79] KRENDEL, E. S., AND MCGRUER, D. T. A servomechanisms approach to skill development. *Journal of the Franklin Institute* 269, 1 (1960), 24–42.
- [80] KSCHISCHANG, F. R., FREY, B. J., AND LOELIGER, H.-A. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory* 47, 2 (2001), 498–519.
- [81] KUIPERS, B. Modeling spatial knowledge. *Cognitive Science* 2 (1978), 129–153.

- [82] KUIPERS, B. The spatial semantic hierarchy. *Artificial Intelligence*, 119 (2000), 191–233.
- [83] KUIPERS, B. An intellectual history of the spatial semantic hierarchy. In *Robotics and Cognitive Approaches to Spatial Mapping*, vol. 38 of *Springer Tracts in Advanced Robotics*. 2008, pp. 243–264.
- [84] KUIPERS, B., MODAYIL, J., BEESON, P., MACMAHON, M., AND SAVELLI, F. Local metrical and global topological maps in the hybrid spatial semantic hierarchy. In *IEEE International Conference on Robotics and Automation* (2004), pp. 4845–4851.
- [85] LAGRANGE, J. L. Mécanique analytique. In *Oeuvres*, J.-A. Serret, Ed. Gauthier-Villars, Paris, France, 1867.
- [86] 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.
- [87] 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.
- [88] 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.
- [89] LONČARIĆ, J. Normal forms of stiffness and compliance matrices. *IEEE Journal of Robotics and Automation RA-3*, 6 (1987), 567–572.
- [90] MADNI, A. M., AND SCOTT, J. Towards a conceptual framework for resilience engineering. *IEEE Systems Journal* 3, 2 (2009), 181–191.
- [91] MALLET, A., PASTEUR, C., HERRB, M., LEMAIGNAN, S., AND INGRAND, F. GenoM3: Building middleware-independent robotic components. In *IEEE International Conference on Robotics and Automation* (2010), pp. 4627–4632.
- [92] MANSARD, N., AND CHAUMETTE, F. Task sequencing for sensor-based control. *IEEE Transactions on Robotics* 23, 1 (2007), 60–72.
- [93] MASON, M. T. Compliance and force control for computer controlled manipulators. *IEEE Transactions on Systems, Man, and Cybernetics SMC-11*, 6 (1981), 418–432.
- [94] McRUER, D., ALLEN, W., AND WEIR, D. The man/machine control interface. *IFAC Proceedings Volumes* 11, 1 (1978), 2225–2231.
- [95] MESAROVIC, M. D. Multilevel systems and concepts in process control. *Proceedings of the IEEE* 58:1 (1970), 111–125.
- [96] MESAROVIC, M. D., MACKO, D., AND TAKAHARA, Y. Two Coordination Principles and Their Application in Large Scale Systems Control. *Automatica* 6 (1970), 261–270.
- [97] MESAROVIC, M. D., AND TAKAHARA, Y. Multilevel systems and concepts in process control. *Proceedings of the IEEE* 58:1 (1972), 111–125.

- [98] MINATI, G. General System(s) Theory 2.0: A Brief Outline. In *Towards a Post-Bertalanffy Systemics*, (2016), 211–219.
- [99] MINSKY, M. L. A framework for representing knowledge. In *The psychology of computer vision*, 1975.
- [100] MOORE, E. H. On the reciprocal of the general algebraic matrix. *Bulletin of the American Mathematical Society* 26 (1920), 389 and 394–395.
- [101] MOZZI, G. *Discorso Matematico sopra il Rotamento Momentaneo dei Corpi*. Stamperia del Donato Campo, Napoli, 1763.
- [102] MULDER, M., POOL, D. M., ABBINK, D. A., BOER, E. R., ZAAL, P. M. T., DROP, F. M., DER EL, K., AND PAASSEN, M. M. Manual control cybernetics: State-of-the-art and current trends. *IEEE Transactions on Human-Machine Systems* 48, 5 (2018), 468–485.
- [103] NÜCHTER, A., AND HERTZBERG, J., Towards semantic maps for mobile robots. *Robotics and Autonomous Systems* 56, (2008), 915–925.
- [104] NEWELL, A., AND SIMON, H. A. The simulation of human thought Technical Report of *The RAND corporation*, P-1734 (1959), 1–43.
- [105] OLIVA, A., AND TORRALBA, A. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International Journal of Computer Vision* 42, 3 (2001), 145–175.
- [106] 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.
- [107] PEARL, J. Fusion, propagation, and structuring in belief networks. *Artificial Intelligence* 29 (1986), 241–288.
- [108] 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* (2015).
- [109] 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 *IEEE/RSJ International Conference on Intelligent Robots and Systems* (2015).
- [110] 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.
- [111] PHILLIPS, S., HALFORD, G. S., AND WILSON, W. H. The processing of associations versus the processing of relations and symbols: A systematic comparison. In *Annual Conference of the Cognitive Science Society* (1995), pp. 688–691.

- [112] POINSOT, L. Sur la composition des moments et la composition des aires. *Journal de l'Ecole Polytechnique* 6, 13 (1806), 182–205.
- [113] POPOV, E. P., VERESHCHAGIN, A. F., AND ZENKEVICH, S. L. *Manipulyatsionnye roboty: dinamika i algoritmy*. Nauka, Moscow, 1978.
- [114] POTOCHNIK, A. AND MCGILL, B. *The Limitations of Hierarchical Organization. Philosophy of Science*, 79 (2012), 120–140.
- [115] PROPOI, A. I. Application of linear programming methods for the synthesis of automatic sampled-data systems. *Avtomatika i Telemekhanika*, 24:7 (1963), 912–920.
- [116] PULSIFER, P. L., AND TAYLOR, D. R. F. The Cartographer as Mediator: Cartographic Representation from Shared Geographic Information. In *Modern Cartography Series* 4, (2005), 149–179.
- [117] RADESTOCK, M., AND EISENACH, S. Coordination in evolving systems. In *Trends in Distributed Systems. CORBA and Beyond*. Springer-Verlag, 1996, pp. 162–176.
- [118] RAMM, E. Principles of least action and of least constraint. *Gesellschaft f. Angewandte Mathematik und Mechanik (GAMM)* 34, 2 (2011), 164–182.
- [119] RAWLINGS, J. B., AND AMRIT, R. Optimizing Process Economic Performance Using Model Predictive Control. In *Nonlinear Model Predictive Control*, (2009), 119–138.
- [120] REID, D. B. An algorithm for tracking multiple targets. *IEEE Transactions on Automatic Control* 24, 6 (December 1979), 843–854.
- [121] REYES, M. C. Cybergcartography from a Modeling Perspective. In *Modern Cartography Series* 4 (2005), 63–97.
- [122] RICHALET, J., RAULT, A., TESTUD, J. L., AND PAPON, J. Model predictive heuristic control: Applications to industrial processes. *Automatica*, 14 (1978), 413–428.
- [123] ROSCH, E., AND MERVIS, C. B. Family resemblances: Studies in the internal structure of categories. *Cognitive Psychology* 61 (1975), 573–605.
- [124] ROTH, B. On the screw axis and other special lines associated with spatial displacements of a rigid body. *Transactions of the ASME, Journal of Engineering for Industry* 89 (1967), 102–110.
- [125] RUSSELL, S. J., AND NORVIG, P. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.
- [126] 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.
- [127] 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.

- [128] SCHULZ, M. Decisions and higher-order knowledge. *Nous* 51, 3 (2017), 463–483.
- [129] 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.
- [130] SCIOMI, 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.
- [131] SCIOMI, 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.
- [132] SHERIDAN, T. B. Three models of preview control. *IEEE Transactions on Human Factors in Electronics* 7, 2 (1966), 91–102.
- [133] SHERIDAN, T. B. *Telerobotics, Automation, and Human Supervisory Control*. MIT Press, 1992.
- [134] SIMON, H. A. Behavioral model of rational choice. *The Quarterly Journal of Economics* 69, 1 (1955), 99–118.
- [135] SIMON, H. A. Rational choice and the structure of the environment. *Psychological Review*, 63:2 (1956), 129–138.
- [136] SIMON, H. *The Sciences of the Artificial*. MIT Press, 1969.
- [137] SIMON, H. A., AND ANDO, A. Aggregation of Variables in Dynamic Systems. *Econometrica* 29:2 (1961), 111–138.
- [138] SIMON, H. A., AND NEWELL, A. Computer Simulation of Human Thinking and Problem Solving. *Monographs of the Society for Research in Child Development*, 27:2 (1962), 137–150.
- [139] SINACEUR, H. B. Facets and levels of mathematical abstraction. *Philosophia Scientiae* 18, 1 (2014), 81–112.
- [140] STACHOWIAK, H. *Allgemeine Modelltheorie*. Springer, 1973.
- [141] STRANG, G. *Linear Algebra and its Applications*, 3rd ed. Harcourt Brace Jovanovich, 1988.
- [142] TAKAHARA, Y., AND MESAROVIC, M. D. Coordinability of dynamics systems. *IEEE Transactions on Automatic Control* 14:6 (1969), 688–698.
- [143] TAYLOR, D. R. F. The Concept of Cybergartography. In *Maps and the Internet*, (2003), 405–420.
- [144] TAYLOR, D. R. F. The theory and practice of Cybergartography: An introduction. In *Modern Cartography Series* 7, (2019), 25–33.

- [145] TIBSHIRANI, R. J. Adaptive piecewise polynomial estimation via trend filtering. *The Annals of Statistics* 42, 1 (2014), 285–323.
- [146] TOLK, A., ADAMS, K. M., AND KEATING, C. B. Towards intelligence-based systems engineering and system of systems engineering. In *Intelligence-Based Systems Engineering* (2011), vol. 10 of *Intelligent Systems Reference Library*, pp. 1–22.
- [147] TOSI, N., DAVID, O., AND BRUYNINCKX, H. DOF decoupling task graph model: Reducing the complexity of touch-based active sensing. *Robotics* 4, 2 (2015), 141–168.
- [148] TSYPKIN, I. Z. *Adaptation and learning in automatic systems*. Academic Press, 1971.
- [149] 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.
- [150] VALCKENAERS, P. ARTI reference architecture — PROSA revisited. In *Service Orientation in Holonic and Multi-Agent Manufacturing* (2018), no. 803 in Studies in Computational Intelligence, pp. 1–19.
- [151] VALCKENAERS, P. Perspective on holonic manufacturing systems: PROSA becomes ARTI. *Computers in Industry* 120 (2020), 103226:1–14.
- [152] VALCKENAERS, P., VAN BRUSSEL, H., BRUYNINCKX, H., SAINT GERMAIN, B., VAN BELLE, J., AND PHILIPS, J. Predicting the unexpected. *Computers in Industry* 62 (2011), 623–637.
- [153] VALCKENAERS, P., VAN BRUSSEL, H., HAELI, 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.
- [154] VAN LEEUWEN, J. On Floridi’s method of levels of abstraction. *Minds & Machines* 24 (2014), 5–17.
- [155] 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.
- [156] VARANKA, D. E., AND USERY, E. L. The map as knowledge base. *International Journal of Cartography* 4:2 (2018), 201–223.
- [157] VERESHCHAGIN, A. F. Gauss principle of least constraint for modelling the dynamics of automatic manipulators using a digital computer. *Soviet Physics Doklady* 20, 1 (1975), 33–34.
- [158] VERESHCHAGIN, A. F. Modelling and control of motion of manipulational robots. *Soviet Journal of Computer and Systems Sciences* 27, 5 (1989), 29–38.
- [159] VON BERTALANFFY, L. An outline of general system theory. *British Journal for the Philosophy of Science* 1, (1950), 134–165.

- [160] 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, 1994, pp. 128–148.
- [161] W3C. QUDT (Quantities, Units, Dimensions, and Types). <http://www.qudt.org>.
- [162] WIENER, N. *Cybernetics or Control and communication in the animal and the machine*. MIT Press, 1948.
- [163] WILL, P. M., AND GROSSMAN, D. D. An experimental system for computer controlled mechanical assembly. *IEEE Transactions on Computers* 24, 9 (1975), 879–888.
- [164] WINFIELD, A. F. T. Experiments in artificial theory of mind: From safety to story-telling. *Frontiers in Robotics and AI* 5 (2018).
- [165] WIRTH, N. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.
- [166] WU, J. Hierarchy Theory: An Overview. In *Linking Ecology and Ethics for a Changing World*, (2013), 281–301.
- [167] WULF, W., COHEN, E., CORWIN, W., JONES, A., LEVIN, R., PIERSON, C., AND POLLACK, F. HYDRA: the kernel of a multiprocessor operating system. *Communications of the ACM* 17, 6 (1974), 337–345.
- [168] 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.
- [169] ZELLNER, A. Optimal information processing and Bayes’s theorem. *The American Statistician* 42 (1988), 278–284.
- [170] ZENDER, H., MARTÍNEZ MOZOS, Ó., JENSFELT, P., KRUIJFF, G.-J. M., AND BURGARD, W., Conceptual Spatial Representations for Indoor Mobile Robots. *Robotics and Autonomous Systems* 56(6) (2008), 493–502.