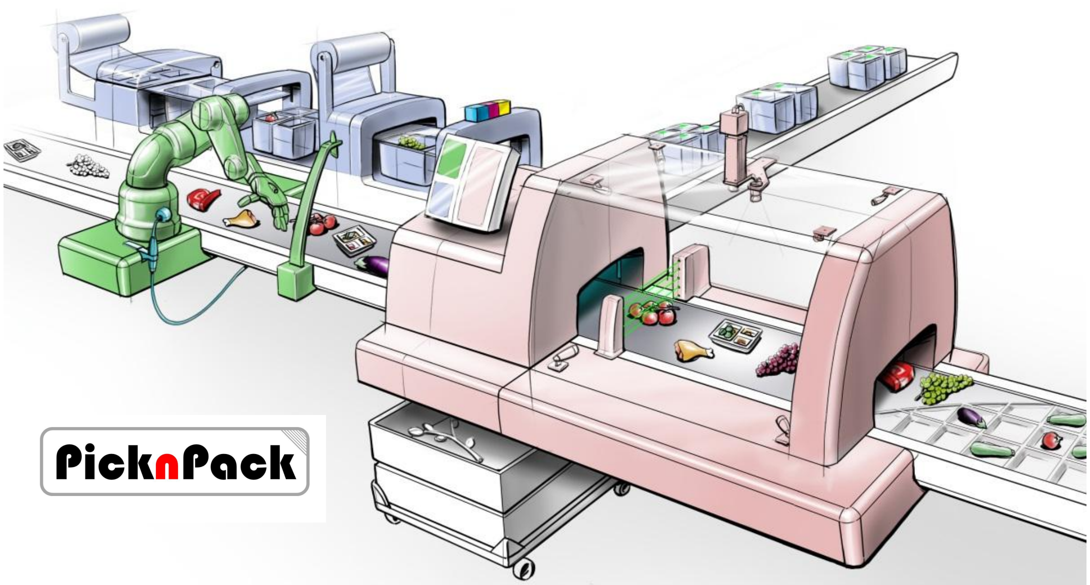
D2.4 — Integration of domain specific knowledge with the component model

Herman Bruyninckx

25/10/2014



Flexible robotic systems for automated adaptive packaging of fresh and processed food products

SEVENTH FRAMEWORK PROGRAMME

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement n^{o} 311987.

Dissemination level		
PU	Public	**X**
PR	Restricted to other programme participants (including the EC Services)	
RE	Restricted to a group specified by the consortium (including the EC Services)	
CO	Confidential, only for members of the consortium (including the EC Services)	

Table of Contents

1 Introduction & Overview

This Deliverable describes the work of Year 1 and Year 2 in using formally represented knowledge to increase the *flexibility* of food processing modules, more in particular, to let the control software of individual machines or modules *configure* its own food-specific functionalities on the basis of formally represented "ontology knowledge".

More in particular, the aim of *Milestone MS1* is to allow the grasping of a cherry tomato by configuring the visual sensing module and the grasp planning module with "magic numbers" coming from a tomato ontology: color and shape parameters of the vine and the individual tomatoes attached to it; best grasping positions on the vine; best approach and lift motions.

LACQ and, especially, DLO and KUL have been the main contributors to the research leading to this Deliverable:

- LACQ has interacted with DLO for identifying the knowledge about how to grasp vine tomatoes and for integrating that into LACQ's pick-and-place robot system.
- DLO and KUL have cooperated in making DLO's visual tomato detection and classification software ready for runtime configuration via "queries" to a knowledge data base, based on the "System Composition Pattern" (explained in more detail in Deliverable D2.1).
- DLO and KUL have cooperated in the creation of a "tomato ontology" to formally capture the knowledge required in the above-mentioned functionalities.

No working software implementation of the concepts and modules developed for Milestone MS1 have yet been realised, but proof of concepts have been developed for all necessary "bits-and-pieces".

2 Knowledge representation — Ontologies

DLO entered the Pick-n-Pack project with proven expertise in the domain of formal modelling of knowledge (so called *"ontologies"*), more particular in the domain of physical units, [2, 3, 4]. This prior expertise is indeed very relevant, since one of the major stumble blocks in flexible system integration is the fact that the software of different modules implicitly uses incompatible physical units; for example, the mismatch between expressing *length* in millimeters in one module, and in meters in another module, is a frequently occurring situation that typically leads to integration errors that are hard to identify.

During the first two years of the project, DLO and KUL iterated multiple times on a document that (eventually) contains the formal representation of "all" relevant knowledge about tomatoes, necessary to detect, classify and manipulate them, [4] (attached as appendix to this document). During this research, the still rather poor state of the art in formal knowledge representation showed up as an important "showstopper", in the following areas mainly:

- *multi-scale knowledge*: most of the ontology research is still limited to *symbolic* relationships, with OWL and/or RDF being the representation languages of choice. These languages, however, have no support to represent knowledge in the *continuous* scales of physical properties (especially time and space), or of the *discrete* aspects of sensori-motor control, that is, the discrete switches in the sensing and motion control of mechatronic devices, such as robots or food processing machines.
- *n-ary and hierarchical relationships*: OWL and RDF are also limited to "triples", that is, a *relation* that connects *two* concepts. A lot of the real-world relationships, however, connect more than two concepts; for example, a food processing machine consists of multiple machines, interconnected in

multiple physical ways; one cherry tomato vine has connections to multiple tomatoes; the shape of one tomato requires geometrical, texture and color models at various levels of detail, in various parts of the food processing chain; etc.

The latter problem shows up in all projects in which KUL is acting as the integration partner, which has lead to concrete suggestions to improve upon the state of the art in how to structure knowledge via *hierarchical hypergraphs*, [1] (attached in appendix).

All above-mentioned problems are being tackled, step by step, in the ongoing DLO-KUL collaboration, but progress remains slow, due to the lack of any similar knowledge representation activities worldwide that the authors are aware of.

3 Knowledge representation — Software

To increase the "flexibility" of food processing lines is a major objective of the Pick-n-Pack project,[1] and *software improvements* are expected to contribute most to that objective. As in the case above of "ontologies", also in the domain of formal knowledge representation about the software aspects of complex systems, the state of the art is extremely poor. Especially when the ambition, as in Pick-n-Pack, is that the devices themselves have sufficient knowledge about their *structural* and *behavioural* properties to be able to configure the software components that must support the *interactions* between two or more modules. KUL has realised rather unique contributions in the domains of formal representations of, both, software architectures and task specifications for complex devices, [1, 5].

A "proof of concept" implemenation was realised, in which a software system was extended with a "querying" component to retrieve configuration information from a "server" on the "internet". Figs 1–2. The *Redland RDF Libraries*[2] were used as framework basis for this experiment.

4 Milestone MS1

The *Description of Work* of Work Package 2 of the Pick-n-Pack project states the following Milestone summary: *The component and Task-Skill-Motion models for a simple robot-gripper-sensor sub-system are realised.*

This has been achieved, but be it at a low *Technology Readiness Level*[3], more in particular *TRL3*:[4]

- for all of the necessary "bits and pieces", progress has been made in understanding the problem, and in providing *proof of concepts* in the implementation;
- the *integration* into something that could readily be called a "knowledge-driven component" of a system with real-world functionality and performance has not been fully reached;
- the major (and still rather fundamental) "showstoppers" are:
  - *lack of maturity* in tooling and frameworks for the knowledge representation at the symbolic, discrete and continuous levels of abstraction, in an integrated way;

[1] Deliverable D2.1 has more information about how the project tackles these challenges.
[2] http://librdf.org/
[3] http://en.wikipedia.org/wiki/Technology_readiness_level
[4] TRL 3. *Analytical and experimental critical function and/or characteristic proof of concept.*

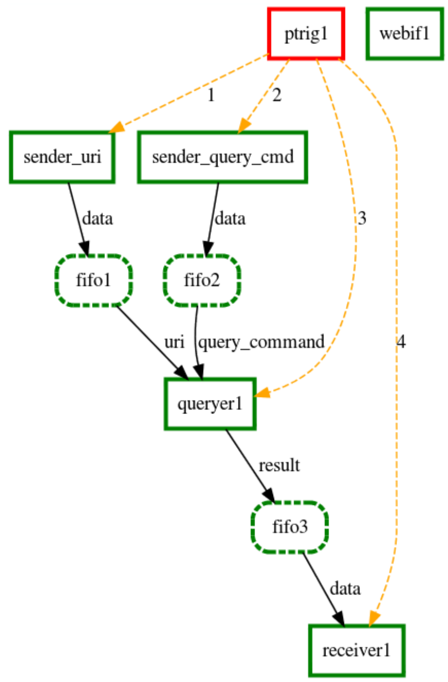


Figure 1: A "proof of concept" implementation of automatic configuration of a software module via a "query" to a knowledge server.

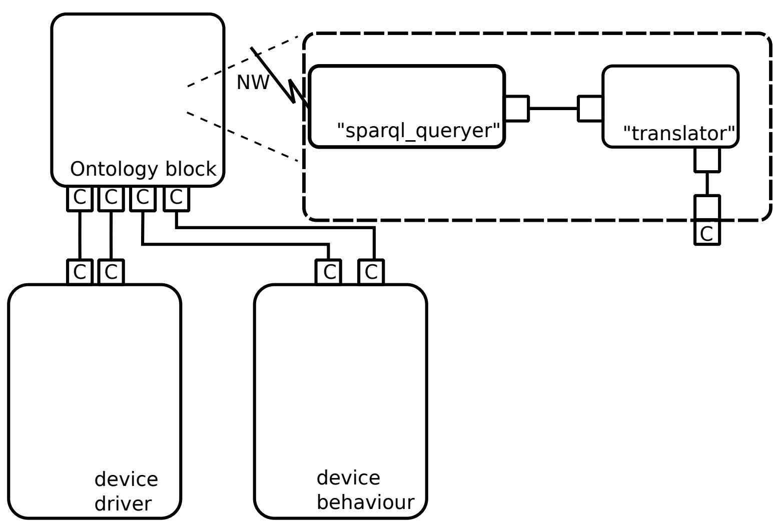


Figure 2: Internal component structure of the "proof of concept" implementation of automatic configuration of a software module via a "query" to a knowledge server.

– *lack of formally represented knowledge*: it is very labour-intensive to encode knowledge about food products (or about any other domain, for that matter) in a formal way that can be used by online reasoning components.

5 Conclusions

From the experiences of the first two project years, the most generically achievable breakthrough that the Pick-n-Pack project will probably be able to realise is to produce food processing machine and module software systems that are able to self-configure their interactions, to the level of the *mechatronic hardware*.

Self-configuration of the modules' *food processing functionalities* with the knowledge about individual food products will see a *proof of concept* integration, with insufficient amounts of useful knowledge available to improve upon the existing practice of fully manual "tuning" in a commercially viable way.

The efforts to reach a full knowledge representation are just too heavy to realise completely in the context of a research project, and only because realistic under conditions of commercial exploitation of those efforts. Nevertheless, the "proof of concept" realisation contains all the necessary components to base such a commercial version on.

References

[1] Herman Bruyninckx, Azamat Shakhimardanov, Markus Klotzbücher, Hugo Garcia. *Hierarchical Hypergraphs for Knowledge-centric Robot Systems: a Composable Structural Meta Model and its Domain-Specific Language NPC4*. Final draft, to be submitted to *Journal of Software Engineering in Robotics*.

[2] Hajo Rijgersberg. *Semantic support for quantitative research*. PhD Thesis Vrije Universiteit Amsterdam, 2013.

[3] Hajo Rijgersberg, M. F. J. van Assem, and Jan L. Top. *Ontology of Units of Measure and Related Concepts*, *Semantic Web*, 4(1):3–13, 2013.

[4] Hajo Rijgersberg, Gert Kootstra, Evert Jans, Herman Bruyninckx, Jan Top. *PicknPack Ontology*. Version *20140313*, 2014.

[5] Dominick Vanthienen, Tinne De Laet, and Herman Bruyninckx. *Systematic Robot Application Development: Applying the Composition Pattern to Constraint-Based Programming*. Submitted to *IEEE Robotics and Automation Magazine*, 2014.

PicknPack Ontology

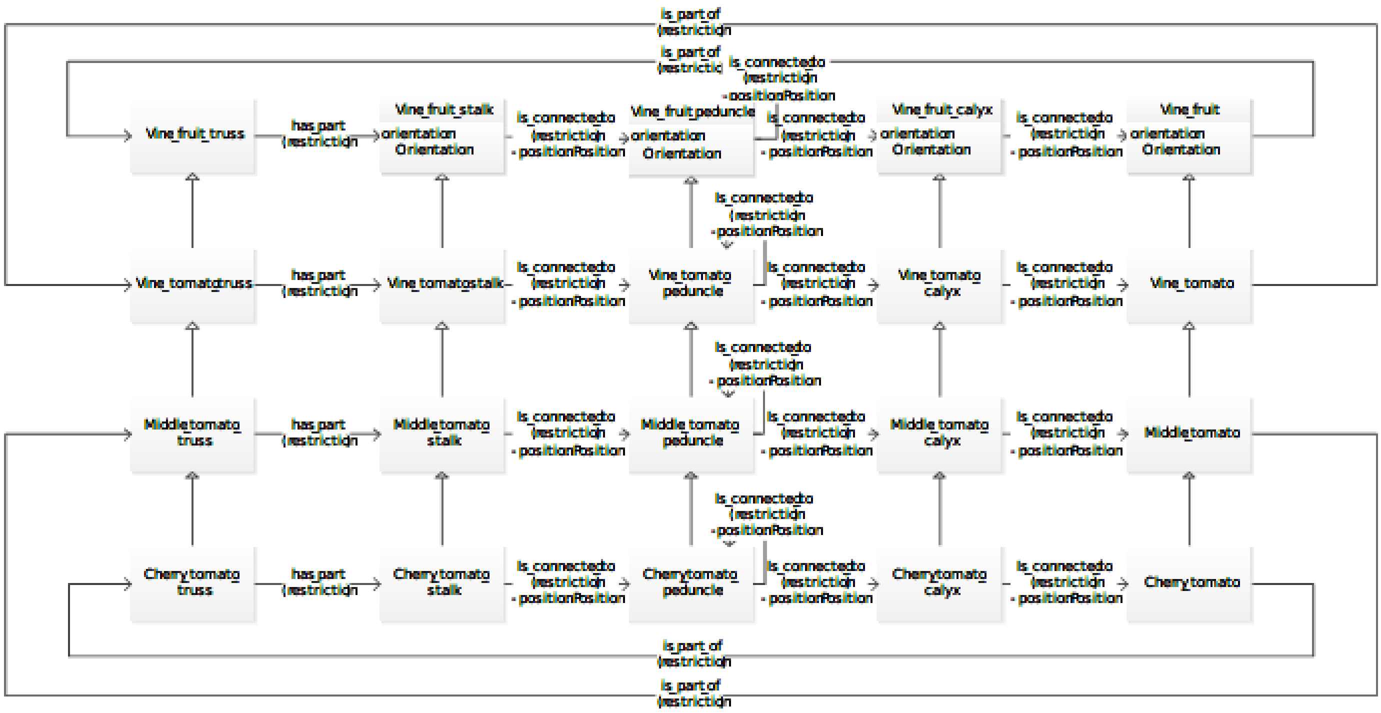
Hajo Rijgersberg, Gert Kootstra, Evert Jans, Herman Bruyninckx, Jan Top
Version 20140313 (the document)

Here a new update of the ontology proposal for the PicknPack project. I have translated the document (from Dutch) and added new chapters on states, and contexts and tasks, based on a meeting Herman, Evert and I had in Leuven in February this year.

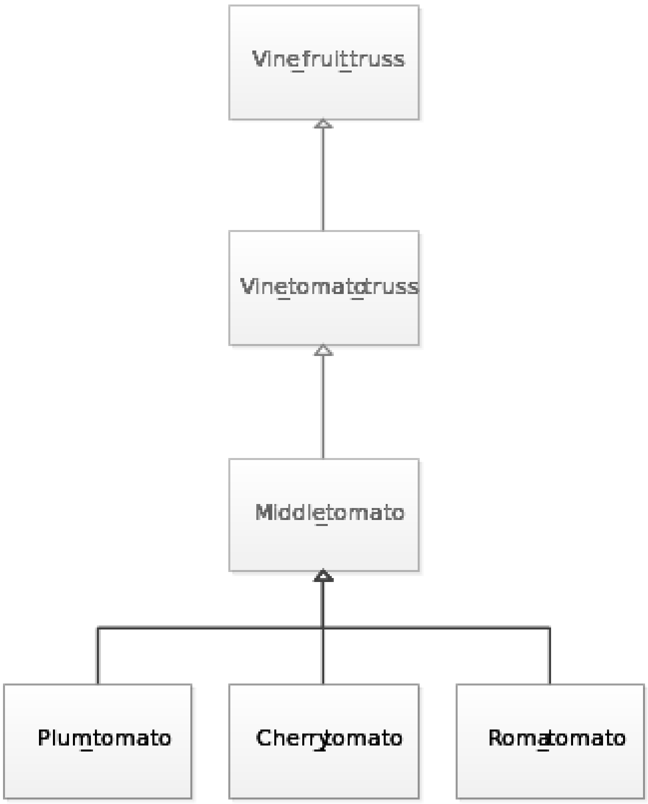
Contents:

1 Vine fruit hierarchy

First of all the component vine fruit hierarchy (don't be overwhelmed by the picture):

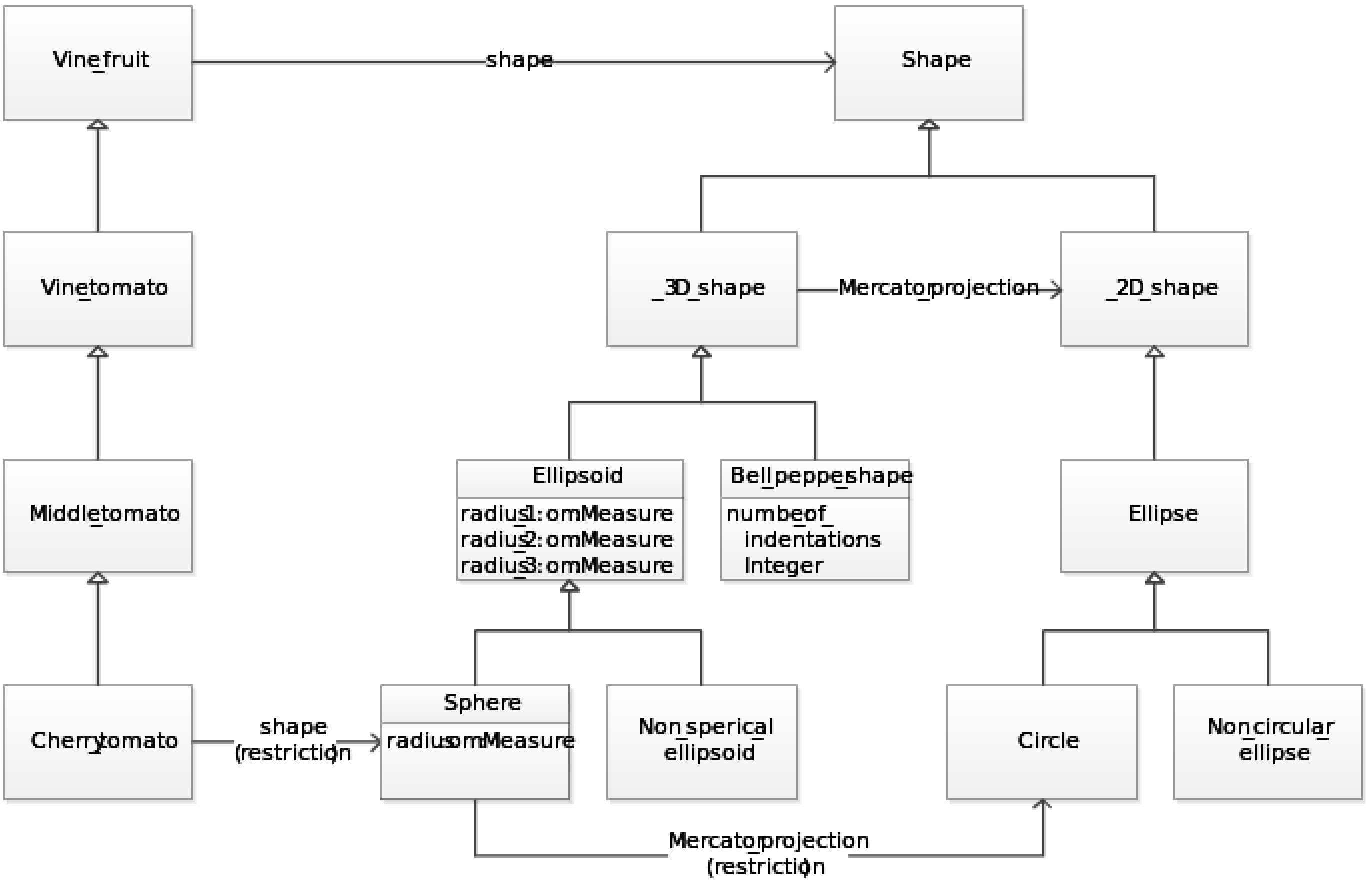


At the highest level we define Vine_fuit, Vine_fruit_truss, Vine_fruit_stalk, Vine_fruit_peduncle and Vine_fruit_calyx. These classes have a property "orientation". "The ontology of Herman" is meant to be used to model orientations, positions and other geometric aspects. For the moment we assume that the range of the property is the class Orientation.
Relations between abovementioned classes are defined using the property has_part and its inverse is_part_of, and is_connected_to (a symmetrical relation). The property is_connected_to has got a property itself, namely position. Also the range of that property should be modeled using the ontology of Herman; for the moment we assume that the range is Position.
At the second level we define the classes Vine_tomato, _truss, _stalk, _peduncle, and _calyx.
At the third level we find Middle_tomato, _truss, _stalk, _peduncle, and _calyx. The middle tomato is the product that we focus on in this project. One subclass for middle_tomato is included in the diagram: Cherry_tomato. But of course there are more, such as:



2 Shape

Now the component shape:

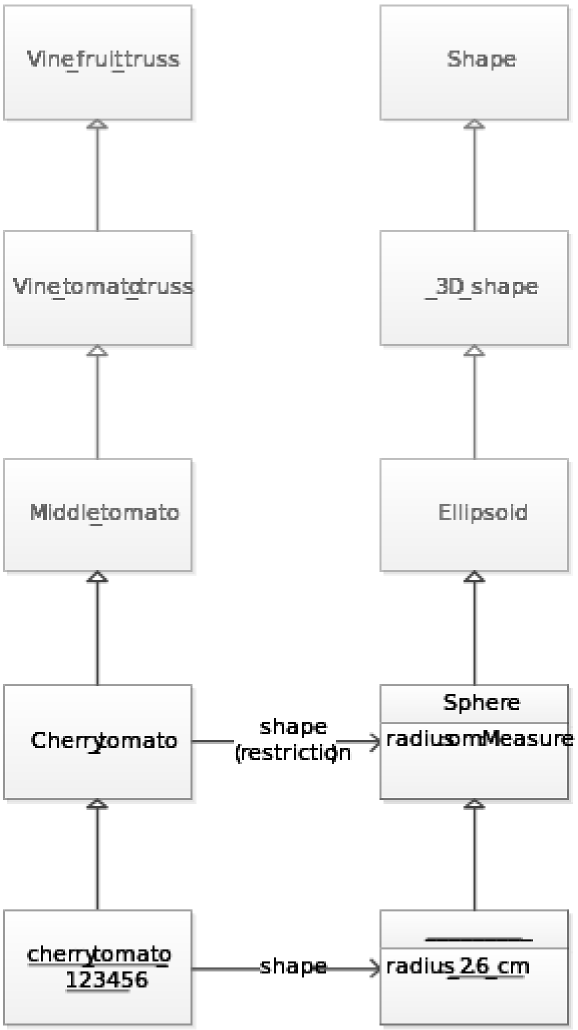


At the right we see the column Vine_fruit, Vine_tomato, Middle_tomato, Cherry_tomato from the previous chapter. At the left we see a hierarchy for shape.

We see that Vine_fruit has a relation “shape” with the class Shape.
At the level of Cherry_tomato this relation is restricted to Sphere. The defined shapes are of course geometrically perfect shapes. In practice one will find no single tomato with such perfect shape. Perhaps we have to define a measure of deviation from the ideal shape and margins how much it may deviate. In fact, margins are important with every concept. One way to deal with margins is to define intervals – we will do that in the next chapter. Other methods are defining distributions (normal distribution, uniform distribution, etc.) or define concepts such as tolerance and deviation as properties of quantitative properties.
Relations may exist between 2D and 3D shapes. One example is included in the figure: the relation Mercator_projection of Sphere to Circle. Many, many other kinds of projections are of course possible. At this moment we will not give priority to this subject.
One open question is how we can indicate at the the level of Sphere that radius_1 = radius_2 = radius.

3 Shape and radius

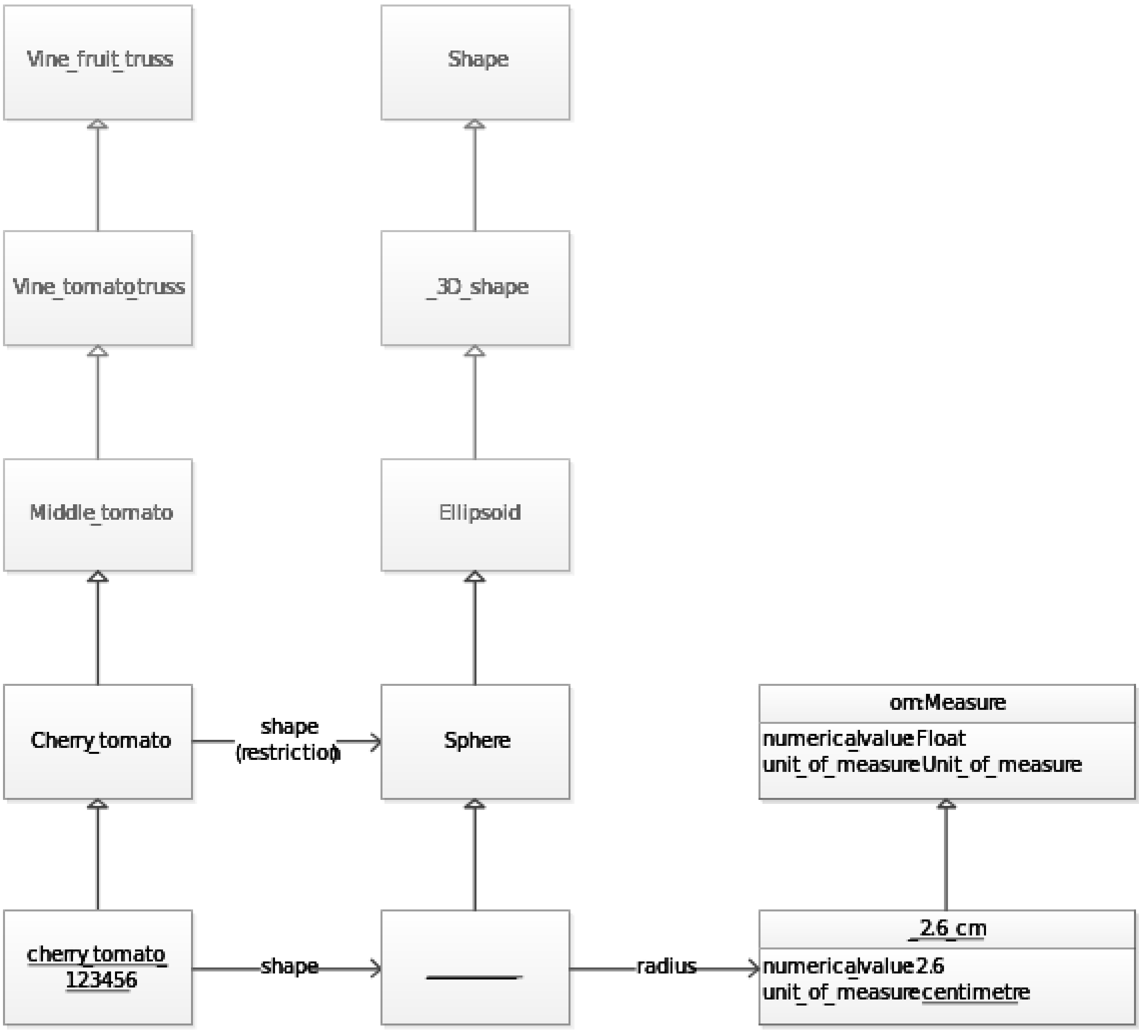
Now I would like to discuss how to specify specific radii for Cherry tomatoes. We will do this using OM, the Ontology of units of Measure and related concepts, because units are involved.
The radius of the shape will be specified for a Shape instance, not for the Cherry tomato itself. As a consequence a specific Cherry tomato will have to have a specific Sphere as shape:



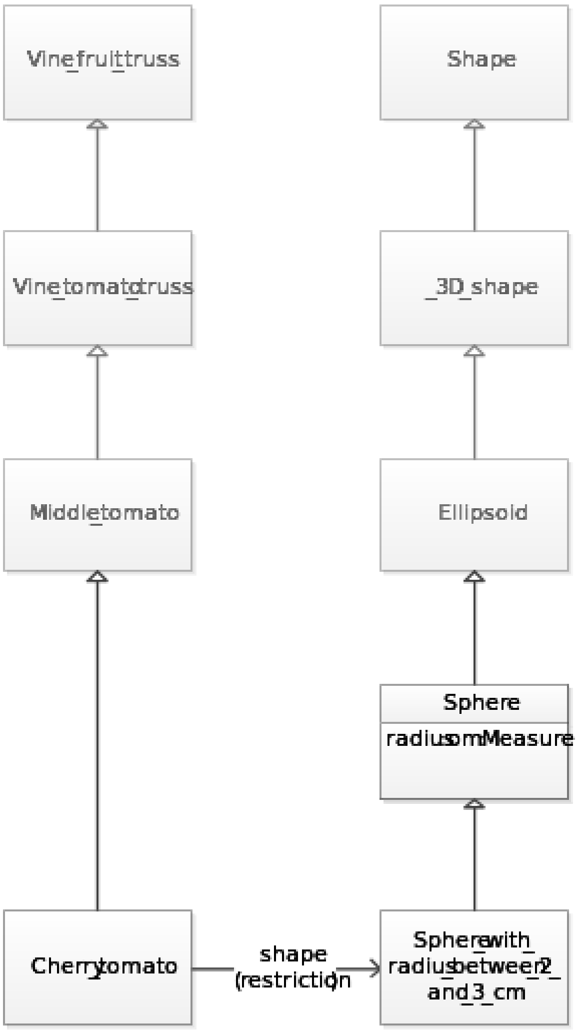
cherry_tomato_123456 is an instance of Cherry_tomato, and _________ is an instance of Sphere (this represents an anonymous instance). This figure represents a structure like:

cherry_tomato_123456.sphere.radius = 2.6 cm.

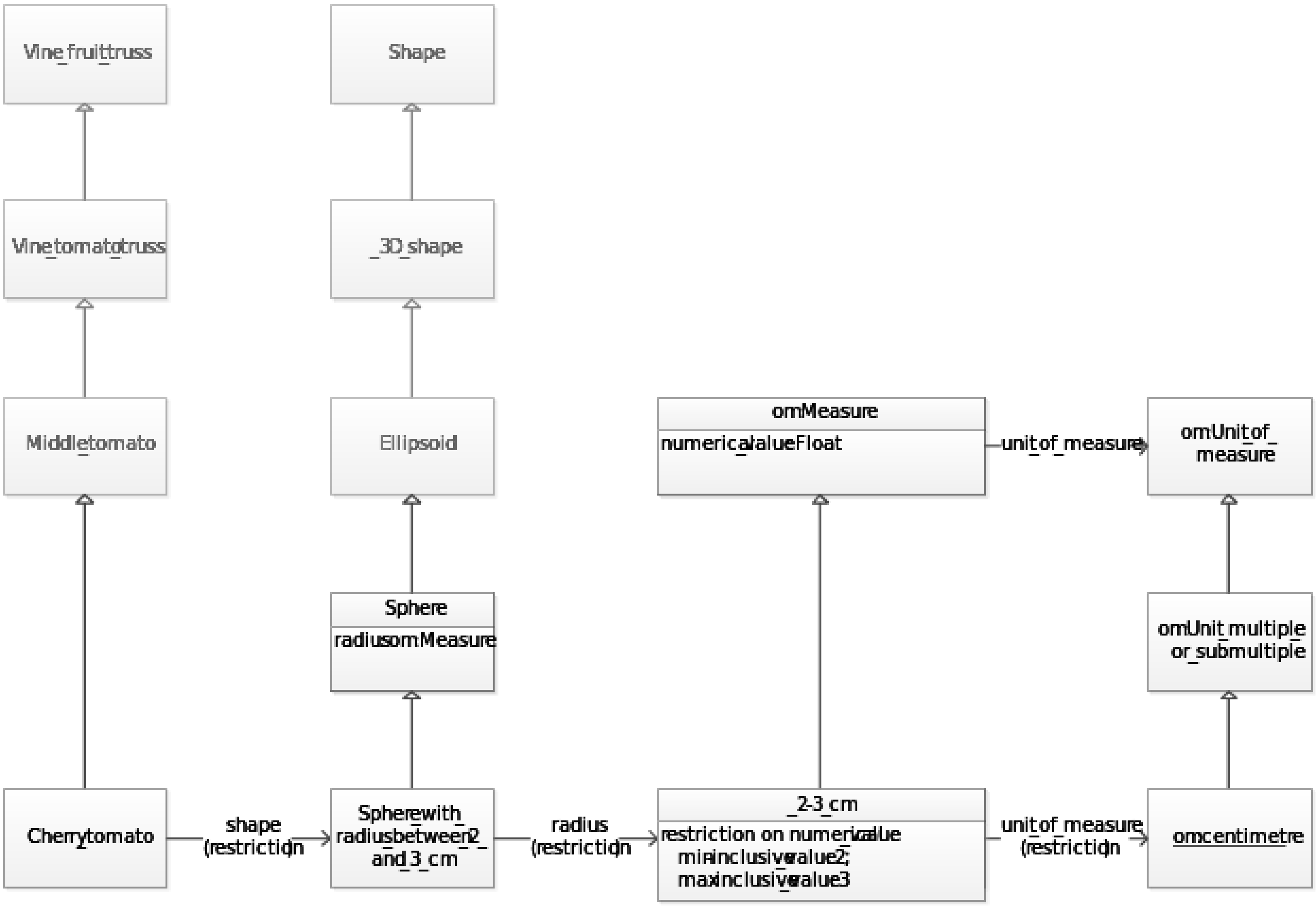
The value of the radius, _2.6_cm, is an instance of the class om:Measure:



Now we would like to postulate at Cherry tomato level that the radius is always between, say, 2 and 3 cm. For this reason it is necessary that the shape of a Cherry tomato refers to a particular subclass of Sphere instead of Sphere itself:



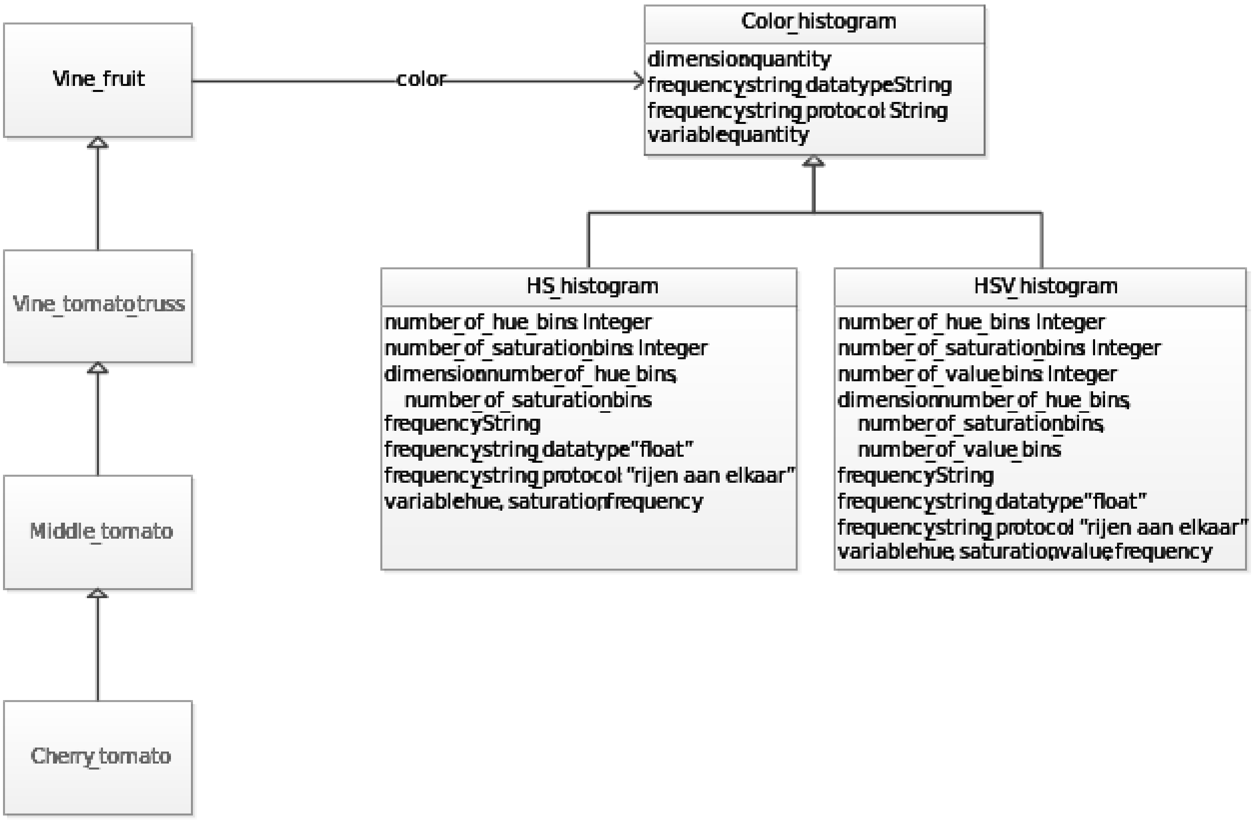
Now we would like to restrict the radius of the new class Sphere_with_radius_between_2_and_3_cm to a measure between 2 and 3 cm. It may look as follows then (don't be afraid):



At the left of the figure we see the same tomato and Sphere classes as in the previous figure. Now we also show the infrastructure for the radius of the class Sphere_with_radius_between_2_and_3_cm. On top we see the class om:Measure again from the previous figure. To the right the class om:Unit_of_measure. This one has got (indirectly, through subclass om:Unit_multiple_or_submultiple, which represents prefixed units such as kilogram and millimeter) the instance om:centimetre.
We have defined the class _2-3_cm_measure (subclass of om:Measure) and have restricted its (datatype) property numerical_value in such way that the minimum and maximum values of the property are specified. We have done this using the XSD properties minInclusive en maxInclusive. XSD is a standard ontology that is being used together with RDFS or OWL.
We have restricted the property unit_of_measure to om:centimetre.

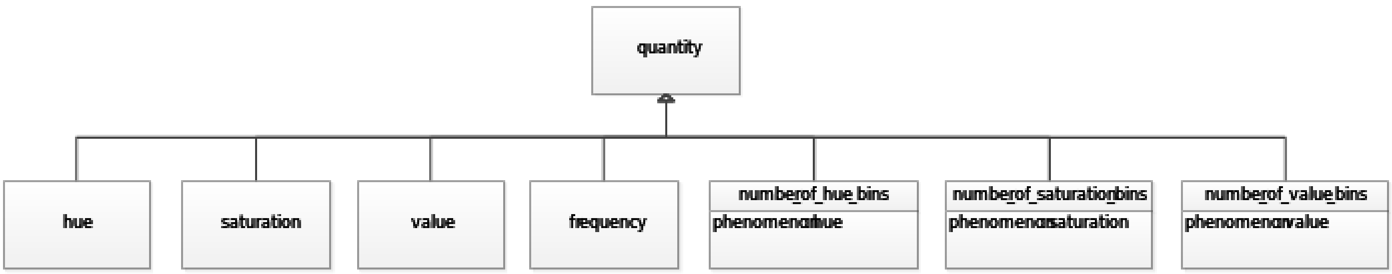
4 Color histogram

And now the component color histogram.



I have tried to follow the idea of NetCDF as much as possible. I'm not sure whether I have succeeded sufficiently. For example, in NetCDF one has to specify explicitly the hue and saturation intervals, where we leave this implicit (we only specify the *numbers* of bins, not exacly *which* bins). Also (for the time being) we leave implicit *how many* hue, saturation, and frequency intervals there are and – above all – what the dimensionality of the variables is (particularly relevant for the frequency variable, which has the dimensions hue and satuariation). Finally, in our approach the structure of the variable values is (for now?) quite flat (particularly relevant for the frequency values): all rows sequenced. In fact all rows should be surrounded by braces, such as in (a human-readable conversion of) NetCDF, as well as the entire dataset.

The different variables can be regarded as quantities:



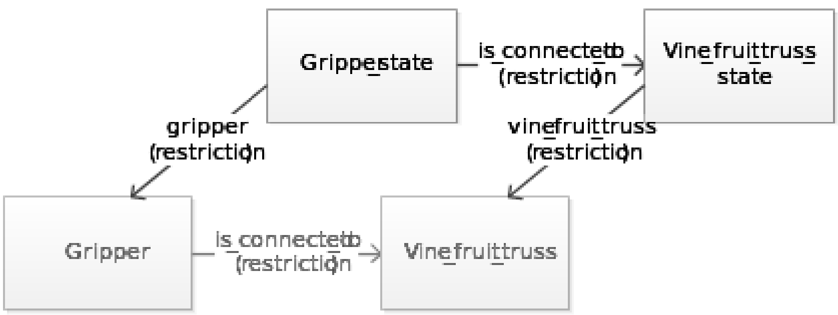
5 State

As to states, the point is that every object can have different states, at different moments in time for example. As a consequence, we can't store these states at the objects as defined in Chapter 1, since these objects are static. Our idea now is to define for each object class a state class, which can be used to store the values of the different states.

I have tried to represent this in the diagram above. I immediately admit it's not a very clear drawing. It is based on the first figure of Chapter 1. In fact I have "doubled" it with state classes (one state class for each object class). Every state class refers to its original object class.
In the upper left class I have included some variable properties, such as time (t), temperature (T), etc. In fact a state as defined in this diagram can be seen as a state *space*. (One state (space) can, by the way, be regarded as a record of a table referring to only one object, but I will leave this sideway for now).

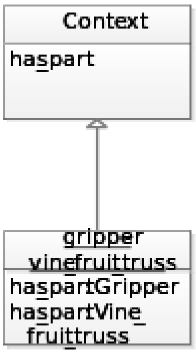
6 Context

A context represents a combination of objects, representing a particular situation, for example a gripper that holds a vine fruit truss, or a vine fruit truss that's in a production line, etc. It is important to be able to store knowledge about such situations.
A gripper that holds a vine fruit truss can be seen as a state, since a gripper does not *always* hold a vine fruit truss. This may be modeled in the below way (similar to the approach in the previous chapter):



However, it is not possible to store the information specific for certain combinations of objects – i.e., contexts. The diagram only indicates which *possible* combinations may be made; it does not represent the combinations as such. So, we define the concept Context for this purpose, with a property has_part using which the desired combinations can be specified:

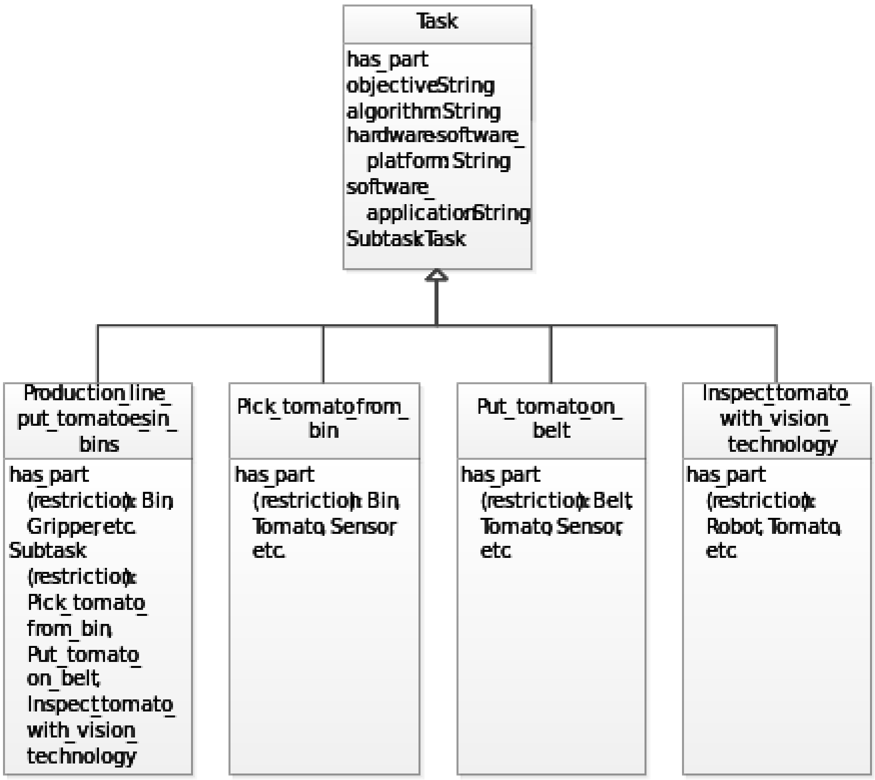
In the example of an instance of such a context gripper-vine_fruit_truss, the property has_part refers to the classes Gripper and Vine_fruit_truss:



Note that information about the realised states appears in the "common" part of the ontology (as described in Chapter 1) and the states part (previous chapter). The concept context is meant to store (generic) knowledge about such situations. I do not think we should define a concept "context state", as we have discussed (Herman, Evert and I).

7 Task

A task can be seen as a process: e.g., pick a tomato from a bin, put a tomato on a belt, inspect a tomato with vision technology, as well as the incorporating task, i.e., the entire production line. Hardware and software specifications are related to these tasks.
Here's a preliminary diagram:



I am thinking about whether these tasks are generic, or specific (realized) information should be stored. Then we should think of state versions of these concepts too. For further discussion. Realizations of these task can be implemented as instances of these classes. Likely, we also need state versions of these classes (like in Chapter 6) to store specific combinations of conditions (times, temperatures, etc.)

8 Future outlook

Especially Chapters 6 and 7 need to more worked out, to my feeling. Let's see the current descriptions as input for discussion.
In the future we would like to focus on how to link histograms to components that can be recognized in images: fruit, shadow, light/lighting, free space, background, etc. The concept image will have to be defined. The concept pixel is probably not required, because that will happen on histogram level. In other words, to a histogram will be linked which type pixel (fruit, stalk, etc.) is represented. In a later stage more about this subject.
Also the concepts that are related to the robot will have to be defined: gripper, arm, production line, belt, etc.

(Draft version. Not for distribution.)

Hierarchical Hypergraphs for Knowledge-centric Robot Systems: a Composable Structural Meta Model and its Domain-Specific Language *NPC4*

Herman Bruyninckx,[1,2] **Azamat Shakhimardanov,**[1]
Markus Klotzbücher,[1] **Hugo Garcia**[1]
[1]University of Leuven, Belgium
[2]Eindhoven University of Technology, the Netherlands

October 4, 2014

Abstract

Many robotics applications rely on *graph models* in one form or another: perception via probabilistic graphical models such as Bayesian Networks or Factor Graphs; control diagrams and other computational "function block" models; software component architectures; Finite State Machines; kinematics and dynamics of actuated mechanical structures; world models and maps; knowledge relationships as "RDF triples"; etc. In traditional graphs, each edge connects just two nodes, and graphs are "flat", that is, a node does not contain other nodes.

This paper advocates *hierarchical hypergraphs* as the more fundamental *structural meta model*: (i) an edge can connect *more* than two nodes, (ii) the attachment between nodes and edges is made explicit in the form of "ports" to provide a *uniquely identifiable* view on a node's internal behaviour, and (iii) every node, edge or port can in itself be another hierarchical hypergraph. These properties are encoded formally in a *Domain Specific Language* (or "meta model"), called "NPC4", built with *node*, *port*, *connector*, and *container* as key language primitives, and *contains* and *connects* as language primitives. The explicit introduction of the four "*c*"-primitives is key to NPC4's *composability* as a modelling language.

NPC4 models only the *structural* aspects of a system, but application-specific connection policies and behaviours can be added systematically. This applies, in particular, to application-specific visualisations, knowledge relationships, and causality definitions, all of which are very centred on *domain knowledge* and show the need for formal ways to represent *context*. Contexts are a challenge for traditional graphs, since they are seldom non-overlapping.

1 Introduction

Everywhere in robotics, graph-based models show up as formal model of concepts, knowledge, software, systems, etc. Graph models are good at separating the *structural* and *behavioural* parts of a design, that is, the graph only represents which nodes interact with which other nodes, without describing the dynamical behaviour *inside* the nodes, or of the interaction dynamics *between* nodes. Below is a non-exhaustive list of examples of graph-based modelling use cases in robotics, where *nodes*, *edges* and (sometimes) *ports* are the building blocks of the graphical models:

- *software architectures*, as in Figs. 1–2. Typically, each node represents an input-output relationship that is dynamic and time-varying, while the structure of the interactions (i.e., the edges and the ports) does not change over time. Some frameworks offer hierarchical composition (e.g., Simulink [46] or Modelica [34]), at least in the *modelling* part of system design.

- *kinematics and dynamics of actuated mechanical structures*, as in Fig. 3. The joint nodes contain actuator dynamics, and the link nodes contain rigid-body inertia dynamics; the edges represent connectivity, modelling which actuators and links are connected, that is, exchanging energy. Hierarchy is

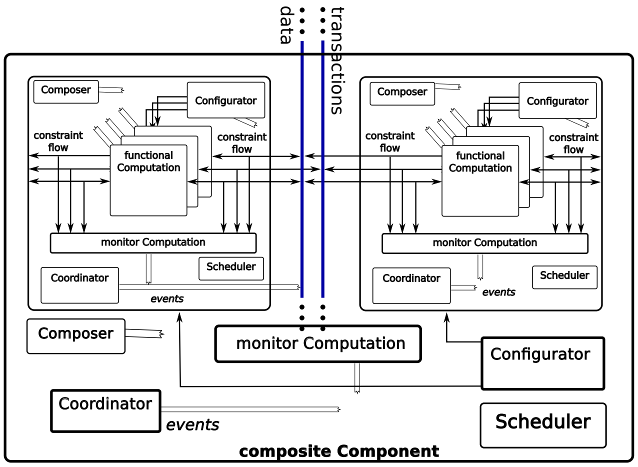


Figure 1: Structural model of a *composite component* software architecture [47]. *Nodes* represent software reponsabilities; *edges* represent data flows.

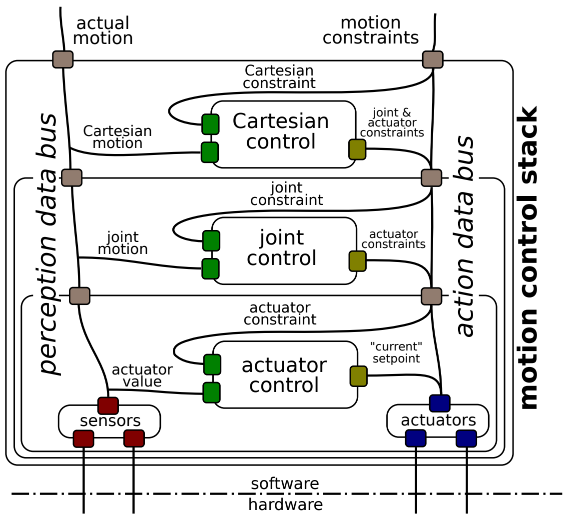


Figure 2: "Reference architecture" for a *motion control* stack of software components. The *nodes* are the rounded rectangles, representing control or input/output activities; the *edges* are lines connecting the coloured *ports*, which give access to variables inside nodes.

possible, e.g., a spherical joint can mechanically be realised by a parallel mechanism.

- *Finite State Machines*, as in Fig. 4, are often used to model the *discrete* aspects of the behaviour of a robot control system. That is, what activities must be running in the system in concurrent ways, and based on which events the system must switch its overall behaviour to another set of concurrent

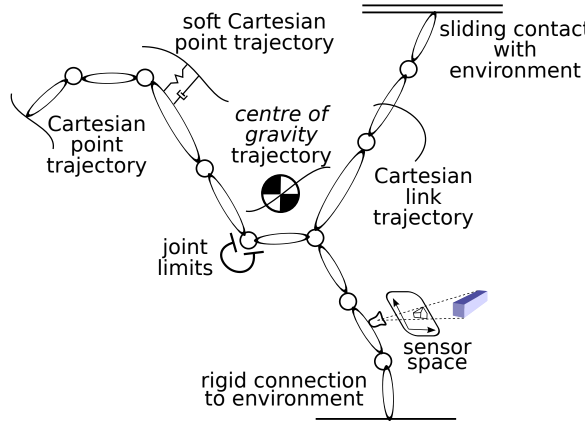


Figure 3: A generic tree-structured kinematic chain with possible task requirements on the chain's joints and links. *Nodes* represent actuated joints and rigid-body links; *edges* represent (dynamics-less) connections between nodes. *Ports* are typically not used, which hinders the methodological extension of the structural chain model with very relevant components such as actuators, transmissions, sensors, geometry, or tasks.

activities. States are connected via so-called "transitions". Structural hierarchy is used to simplify the modelling of the interconnections: all states inside a composite state react to the same event in the same manner.

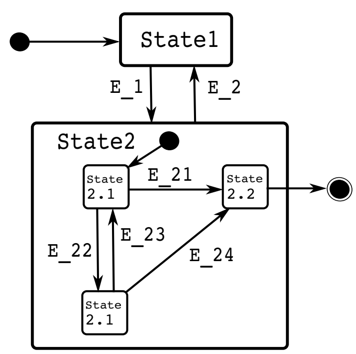


Figure 4: An hierarchical Finite State Machine. *Nodes* represent states, and *edges* represent state transitions; *ports* are typically not used.

- *probabilistic graphical models* such as Bayesian Networks or Factor Graphs, Figs 5–6. *Nodes* represent *information* as captured in "random variables"; *edges* represent probabilistic relationships which govern the interaction between the random variables in the connected nodes. Hierarchy is only part of the textbook vocabulary of probabilistic models in the form of the *plate notation*, Fig. 7.

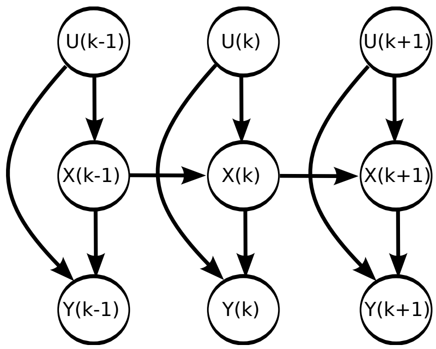


Figure 5: A simple dynamic Bayesian network, representing for example a *Kalman Filter*. The *nodes* contain the random variables in the network, and the *edges* represent probabilistic relationships between random variables; *ports* are typically not represented. However, the model does not allow to indicate which of the random variables in each node are involved in each of the relationships represented by edges; for example, in general, only *some* of the input variables `U(k-1)` influence the output variables `Y(k-1)`.

- *control diagrams* and other computational models, such as the Cartesian position control scheme of Fig. 8; popular instances are *Simulink* [46] diagrams, or *Bond Graph* [1, 8, 24, 38, 39] models in *20Sim* [14]. The separation of structure and behaviour is similar to the above-mentioned cases of software and kinematic models: nodes represent "dynamics", edges represent interaction of information or energy.
- *knowledge representation networks*, such as the "semantic web" (represented often by the RDF, OWL or TopicMap languages) or the robotics *KnowRob* [44] (using also Lisp and Prolog as representation languages). *Nodes* represent facts, data, term, etc., and *edges* represent relationships. RDF and OWL can only represent "triples" relationships; Lisp and Prolog statements have the semantics of *S-expressions* (or "expression trees"). Topic Maps represent more general graphs, but without hierarchy.
- *web applications*: the design behind HTML5 [50] brings a significant change compared to older version of the standard, and most of that change comes from looking at web-based applications as an hierarchical network of interacting components. The *nodes* are HTML5 primitives, such as *Web components* [36], or *HTML templates*; the *edges* represent *bi-directional data binding* supported by JavaScript as in AngularJS [25]; *ports* are the sockets of all kinds that are commonly used in the Web. This evolution of the Web towards separation between structure and behaviour will make it a lot easier to use HTML5 for building graphical user interfaces that match well to the architectures of complex, distributed robot systems.

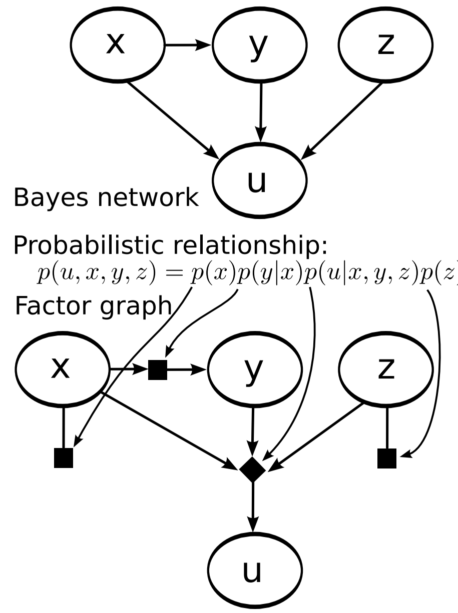


Figure 6: A probabilistic relationship and its corresponding *Bayesian Network* and *Factor Graph* representations. The Factor Graph is one of the few examples where *hyperedges* are *first-class citizens* of the graphical model; the advantage is visible in the figure: the Factor Graph can be linked one-to-one to the semantics it represents (i.e., the probabilistic relationship), while the mainstream Bayesian Network representation can do that only partially.

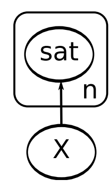


Figure 7: The so-called *plate notation* is one of the few examples where *hierarchy* is a *first-class citizen* of probabilistic graphical models. The plate is the rounded rectangle, and it represents n copies of the graph it contains, in this case, just one single random variable `sat` in the round node.

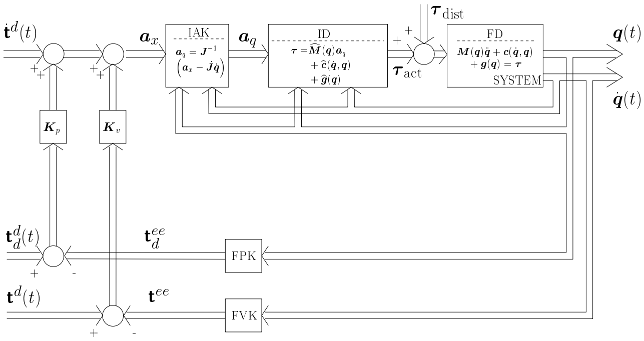


Figure 8: A generic Cartesian control diagram for position controlled robots. The *nodes* contain computations on variables in the diagram, and the *edges* represent (directed) transfer of such variables between nodes; *ports* are typically not represented, as is *hierarchy*. However, the latter *is* present in most controllers, via the implicit structural primitive of *cascaded control loops*, that is, an "outer" control loop around an "inner" loop.

All graph models in the paragraphs above represent the *structure* of the interactions that are represented by their *edges*, and their *nodes* are the containers for the different kinds of *behaviour* that the model represents. Some models support *hierarchy* (i.e., a node can contain a full graph in itself), and some support *hyperedges* (i.e., one edge can link more than two nodes). Some models introduce the concept of a *port* (such as software models, Bond Graphs, or HTML5) as a "view" on part of the internal state of the node it is connected to, and (hence) serving as an explicit "attachment point" for interactions via edge connectors.

A hierarchical hypergraph is a good formal representation to cover *all* the compositional structure discussed above, more particulary, via the *property* (*"has-a"*), *containment* (*"part-of"*) and *connection* (*"interacts-with"*) primitives. (Such formalized structure is called a *mereotopology*, see [7] and references therein.)

Each application domain needs more than a structural model alone, obviously; the approach in this paper makes sure that structure and behaviour are strictly separated, but at the same time *composability* is a first-class design driver, and a systematic method is explained to attach an application domain's own *behavioural* model(s) (its *"is-a"* relationships) to the structural model represented by hierarchical hypergraphs.

Support for *hierarchical hypergraphs*, including *ports*, as *first-class citizens* in the model is a rare exception, e.g., in the examples above, only FSMs, Factor Graphs and HTML5 have them in their models, at least implicitly. Nevertheless, hierarchical, port-based, multi-node interactions are common in all engineering disciplines, as major modelling instruments to deal with complexity. Most practitioners in the field of (robotics) system design are not aware of the fact to what extent their modelling languages and tools restrict their flexibility in modelling the designs of their systems.

In robotics software engineering, most projects[1] *even do not have* explicit structural models, since they provide *only source code*; at best, "models" are only used as informal means of documentation, to be understood by the human developers, but not by the robots themselves during their runtime activities, nor by software tooling to support (semi) automatic code generation. There are a few exceptions that (i) provide explicit formal models (for example, Proteus [27], or OpenRTM [2, 35]), and (ii) support hierarchical hypergraph models implicitly (for example, Mathlab/Simulink or 20Sim, the ROCK toolchain for Orocos [9, 12, 11, 29], or the "plate notation" in probabilistic graphical models, Fig. 7). None of those, however, support the full flexibilty that hierarchical hypergraphs provide to model the structural aspects of complex systems. This restriction becomes a more and more important design bottleneck in robotics, since modern robotic systems are increasingly depending on *runtime use of knowledge*, and the "flat triple spaces" that are standard in common OWL-based [48] semantic web approaches to knowledge representations [3] have proven to be extremely difficult to maintain, adapt, reason with, and compose. The latter problem, more particularly, is caused by the lack of support for hierarchy in OWL or RDF.

Objectives and overview The aim of this paper is to improve the modelling flexibility that robot system developers have in tackling these complexity challenges, by introducing them to an hierarchical hypergraph *meta model*[2] they can use to tackle *all* of the above-mentioned use cases, and many more, in a methodological way. The core idea in the methodology is the insight that all systems have a *structural* part (that is, the model that represents (i) which subsystems interact with which other ones, and (ii) how their internal structure looks like), that can be *fully separated* from their *behavioural* part (that is, the model of the "dynamics" of the subsystems). Being aware of that *separation of concerns*, and having access to a formal modelling language that supports it, is expected to help a lot (i) to let human devel-

[1] Including popular "open source" projects such as ROS, Orocos, OpenCV, PointCloudLibrary, etc.

[2] A *meta model* is a language with which to create concrete *models* of a system in a particular application domain or context, [4, 6, 37].

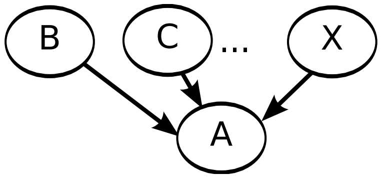


Figure 9: A simple Bayesian network in which the traditional graph structure mis-represents the real interaction between the random variables in the nodes: the network is a graphical representation of the n-ary probabilistic relationship $p(A|B, C, \ldots, X)$, while the arrows suggest only binary interactions. The *Factor Graph* model of Fig. 6 is a better graphical representation of the real n-ary interactions.

opers express their system designs in a more methodological way, and, hence, (ii) to enable a huge gain in development efforts for software representations, toolings and implementations.

Section 2 explains the semantics of what this paper understands under the term "hierarchical hypergraph", since that concept is, surprisingly, not part of the mainstream literature. Section 3 elaborates further on the semi-formally described core conceots, and create a fully formal language for hierarchical hypergraphs, in the form of a *Domain Specific Language* ("DSL", or "meta modelling language"). The language is called *NPC4*, inspired by the first letters of its core *primitives* and *relationships*: `node`, `port`, `connector`, `container`, and, respectively, `contains` and `connects`. The `contain` relationship represents hierarchy, the `connects` relationship represents hyperedges.

Section 4 looks back at the use cases introduced above, and explains how NPC4 can be used as the basis for their structural models.

2 Hierarchical hypergraphs

This Section motivates why the robotics domain has to adopt *hierarchical hypergraphs*, instead of traditional graphs, as its main structural meta model. The motivation is found from a list of examples (Sec. 2.1) that illustrate various ways in which the use of traditional graphs introduces erroneous ways of representing (and hence, "reasoning") about complex systems. The situation is critical since many users of graph models are not aware of these problems, or cannot formulate them by lack of an appropriate and semantically well-defined language; such a language, NPC4, is introduced in Sec. 3.

2.1 Bad practices

Traditional graphs have *nodes* and *edges* as model primitives (such as in, for example, Fig. 5), and most practitioners feel very comfortable with usign them as graphical primitives for modelling. However, traditional graphs have a rather limited expressiveness with respect to *composition*, that is, to model the *structural* properties of a system design. Here is a list of commonly occurring "bad practices" in using traditional graphs to represent the semantics in system models:

- *an edge can only connect two nodes*, while many structural interactions are so-called *n-ary relationships*, that is, more than two (i.c., "n") entities interact at the same time, and influence each other's behaviour.

  Obvious examples of n-ary relationships are "knowledge relationships", such as the (still extremely simple!) Bayesian network of Fig. 9. But also motion controllers of robotics hardware must deal in a coordinated way with all the links, joints, sensors, actuators, *and* their interactions via the robot's kinematic chain.

- *the structural model is flat*, in that all nodes and edges in the model live on the same "layer" of the model. However, *hierarchy* has, since ever, been a primary approach to deal with complexity in design problems.

  Again, *knowledge relationships* are prominent examples of where the problem of flat structural models is very apparent: here, *hierarchy* is equivalent to *context*—that is, the meaning of a concept depends on the context in with it is used—and context is an indispensable structure in coping with the information in, and about, complex systems. Another prominent "bad practice" example are the popular (open source) *robotics software frameworks*, like ROS or Orocos: they do not support hiearchical composition of software nodes, the consequence being that users always see all the dozens, or even hundreds, of nodes at the same time. This makes understanding, analysis and debugging of applications difficult.

- *edges have no behaviour* and just serve as topological symbols representing the logical state of two (or more) nodes to be "connected" or "not connected".

  However, almost all of the use cases in the Introduction have edges that do exhibit dynamics, e.g., the communication channels between software components (time delays, buffering,...), the mechanical dynamics of joints and actuators in robotics hardware, etc.

- *interactions are uni-directional* (in the case of *directed* edges in a graph model), that is, the graph assumes that each "partner" in an interaction can influence one or more other "partners", without ever being influenced by those partners in any way. Nevertheless, *bi-directional* interactions are the obvious reality, in physical interactions (including man-machine interactions), as well as in computational, knowledge and information interactions.

  Again, the recent ROS (and, to a lesser extent) Orocos practice (but also earlier practice in robotics such as [43]), illustrate this problem: software nodes are only exchanging data with each other via so-called *publish-subscribe* protocols, which work only in one direction, namely from the publisher node to the (possibly multiple) subscriber nodes. In addition, publish-subscribe introduces a *policy* (hence, "behaviour") of how messages are being delivered from publisher to subscriber. Few frameworks allow to separate the structure and behaviour of their communication interactions; one of the better examples is ØMQ [28].

  Another "bad practice" are *control diagrams*: the directed edges in, for example, *Simulink* [46] diagrams, can only represent input/output interactions between computational nodes, which prevents a "downstream" computation to influence the behaviour of the "upstream" nodes; saturation of a "block" or "channel" being one of the simplest and common examples of this problem.

  Nevertheless, there are other computational tools, like *20Sim* [14], that do not oblige their users to use only uni-directional interactions, since they offer so-called *Bond Graph*-based modelling primitives [1, 24, 38, 39], that allow to represent the physical bi-directional energy interaction of dynamical nodes.

The opposite of the later problem also occurs: *directed arrows* are used in graphical notations while the represented interaction is really bi-directional, hence resulting in semantically misleading or too constraining models. For example, the probabilistic information in Bayesian networks *does* "flow" in both directions along an edge. Also in this context, *hierarchical*[3] models are (very slowly!) starting to be used [22, 17, 20, 33] because of the complexity of integrating "local" and "global" features in sensor data, and of combining them with the knowledge available about the objects whose sensor features the system can observe.

[3] The hierarchy discussed in this paper is that of nodes and/or edges being compositions of other nodes and edges themselves. This is a semantically different kind of hierarchy then what is called *hierarchical Bayes* models in probabilistic modelling, that indicate models whose topology is a *tree* with the same kind of nodes at each layer of the tree.

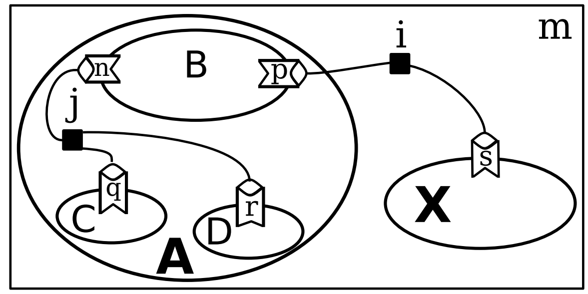


Figure 10: A concrete model of a hierarchical hypergraph relationship. The *nodes* A and X are at the top of the hierarchy. The nodes B, C and D are contained inside node A. The *connectors* "i" and "j" link ports on nodes. The *container* "m" gives the context of the whole composition and allows to refer to it from other models.

2.2 Model primitives in NPC4

This Section defines the complementary modelling concepts of *hyperedges* and *hierarchical graphs*; Sect. 3 later introduces a language to represent them formally. The core of the language are the **structural relationships** between the `Node`, `Port`, `Connector` and `Container` primitives, illustrated in Figure 10:

- `has-a`: a `Port` can exist on itself (e.g., when it is still "floating" during the construction of a model in a graphical tool), but the behaviour of a `Port` is only defined by the `Node` behaviour behind it. So, only statements of the following type are semantically valid:

  $$\texttt{has-a}(\texttt{node-B}, \texttt{port-p}), \tag{1}$$

  whose semantics is that the node with name `node-B` has a port with name `port-p`, and it is through this port that the node can be connected to other nodes, via a connector. So, since ports can belong only to nodes, statements of the following type are *not* semantically valid: `has-a(connector-i, port-p)`, or `has-a(container-m, port-p)`.

- `connect`: a `Connector` forms an hyperedge between (`Ports` on) several `Nodes`. So, statements of the following type are semantically valid:

  $$\texttt{connects}(\texttt{connector-i}, \texttt{node-B}, \texttt{Port-p}). \tag{2}$$

- `contains`: the **containment** hierarchy of `Node`, `Port`, `Connector` and `Container` is represented by statements of the following type:

$$\texttt{contains}(\texttt{M},\texttt{N}), \tag{3}$$

with both `M` and `N` being a `container`, `node`, `port`, or `connector`.

Compared to traditional graphs, the presented model splits the "edge" primitive in two, "port" and "connector", in order to allow "behaviour" not just in the nodes and the edges, but also in the "places" where both are attached to each other. The motivation for this choice is the requirement of *hieararchical composition*: at a certain level of abstraction of a system model, a `Port` might be completely passive, without behaviour, because that behaviour only appears when going to deeper levels of abstraction. A typical example is communication: two nodes connected with communication middleware send and receive data through socket ports, at the application layer, but when going inside such a socket at the level of the operating system, lots of activity becomes visible: packet composition, encoding, timestamping, etc.

The container primitive is an essential and novel addition to graph models, to represent a structural primitive of containment that *carries no behaviour*, but is needed for *information* purposes only. More precisely, the container model primitive is needed *to store meta data*, such as: unique identifiers; references to the modelling languages in which the nodes, ports or connectors inside a container are expressed; references to ontologies that encode the semantic meaning of the model; version numbers; etc. But most importantly, to contain the *model* of the hierarchical hypergraph that is embedded in the container.

The term *composition* is used to denote any combination of the three relationships *has-a*, *connects*, and *contains*.

2.3 Examples of NPC4 models

Figure 10 illustrates the *has-a* and *connects* relationships on a simplistic, artificial example of a container "m":

```
has-a(node-B,port-n),
has-a(node-B,port-p),
has-a(node-C,port-q),
has-a(node-D,port-r),
has-a(node-X,port-s),                        (4)
connects(connector-i, node-B, port-p),
connects(connector-i, node-X, port-s),
connects(connector-j, node-B, port-n),
connects(connector-j, node-C, port-q),
connects(connector-j, node-D, port-r).
```

The Figure also illustrates the *hierarchical composition* (or "*contains*") relationship; e.g., node "A" is the *composition* of nodes "B", "C" and "D", so:

```
contains(node-A, node-B),
contains(node-A, node-C),          (5)
contains(node-A, node-D).
```

The container "m" contains all nodes and connectors:

```
contains(container-m, node-A),
contains(container-m, node-B),
contains(container-m, node-C),
contains(container-m, node-D),          (6)
contains(container-m, node-X),
contains(container-m, connector-i),
contains(container-m, connector-j).
```

Containment is a *transitive* relationship, for example:

```
   contains(container-m, node-A),
   contains(node-A, node-B)             (7)
⇒  contains(container-m, node-B).
```

The hyperedge and hierarchy relationships are *decoupled*, in that one does not imply anything about the other. For example, even though nodes "X" and "B" live at two different levels of the containment hierarchy, the edge "i" can still connect both. The FSM in Fig. 4 shows a real-world example of such hierarchy crossing edge, in the transition out of `State2.2` towards the end state.

2.4 Constraints in NPC4

The `has-a`, `connects` and `contains` relationships typically come with *constraints*, that is, not all syntactically possible relationships are also semantically meaningful. The constraints on `has-a` and `connects` are simple: only nodes can have ports, and connections can only connect to ports. The `contains` relationship is more complex, in that its transitivity property, Eq. (7), implies relationships between more than just the two modelling primitives involved in one single `contains` statement. The constraint imposed by NPC4 is that any composite containment relationships should always represent a **partial order**:

- no node, port or connector *should* be contained in itself, via one or more levels in the containment hierarchy. This would destroy the structural order in the model, while such ordering is exactly the strongest feature in the authors' ambition to (semi)automtic tool chain support for the NPC4 language.
- every node, port and connector *must* be fully contained in another one, or in a container. The motivations are:
  - *pragmatic*: one of the ambitions of NPC is to provide a modelling approach that is infinitely composable, every model needs "something" that other models can *refer to* when they want to include that model in their own model as a sub-system. For example, a model of the kinematics of a robot device combines coordinate representations, physical units, and geometric shapes, but to represent all these different aspects in one single big DSL leads to *"one size fits all"* maintainance and implemention difficulties.
  - *ontological*: every model has a specific *semantic meaning*, and it must be possible to identify explicitly the (possible multiple!) "knowledge contexts" of that meaning. For example, robot motion controllers combine concept and knowledge from the complementary domains of (i) the kinematics and dynamics of robot devices, (ii) linear control theory, and (iii) motion trajectory tasks.
- any node, port or connector *can* be contained in more than one container, but not in more than one other node, port or connector. This constraint reflects the important semantic difference between, on the one hand, the node, port and connector primitives, and, on the other hand, the container primitive: the former are intended *to contain behaviour*, the latter *to represent knowledge*. This fundamental difference in modelling is explained in [6].

Figure 11 shows an example in which a connector is crossing a containment boundary, or, in other words, connectors can leave a container without the *explicit* need for a port on that container. Indeed, NPC4 does *not* introduce the (most often implicit!) constraint of interpreting a *containment boundary* also as a *connection boundary*, since this should only be decided (explicitly!) when domain-specific semantics is being added to the domain-independent semantics provided by NPC4. The case in which the containment constraint also implies a connector constraint is sometimes called a *strict* hierarchical composition, or a "nested" graph; the containment relationship in that case reduces to a tree.

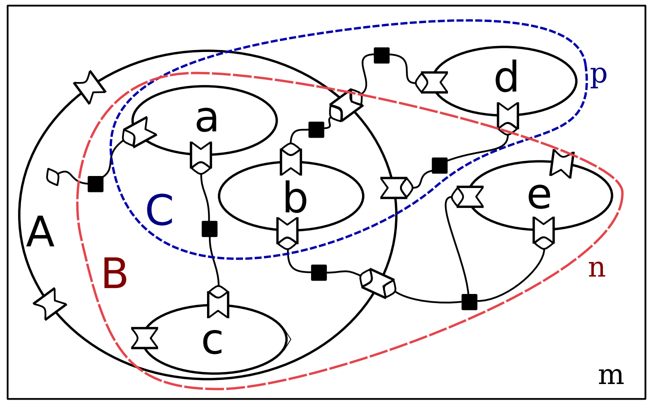


Figure 11: An example of an hierarchical composition in which *containment* does not follow a strict *tree* hierarchy: the containers "p" (small blue dashes) and "n" (long red dashes) have some internal nodes in common, with each other and with node "A"; the containers "p" and "n" do not have ports themselves, in contrast to the node "A".

2.5 Hierarchy — Behaviour

The paragraphs above showed examples of *hierarchy* for *nodes* and containers, but this paper uses the term *hierarchical graph* for a graph in which also ports and connectors can be hierarchies in themselves, Figure 12. A concrete example arises when one decides to distribute a software system over two computers: what was first a simple shared data structure (i.e., a "`connector`") in the centralized version now becomes a full set of cooperating "middleware" software components in itself in the distributed version (i.e., a composition of `nodes`, `connectors` and `ports`).

`Nodes` and `connectors` are both *hyperedges*, in the sense that they both connect zero, one or more `ports`. The `ports` themselves are *not* hyperedges, since in our language, one single `Port` is always connecting one single `node` to one single `connector`. So, as far as *structural* properties are concerned, there is not yet a se-

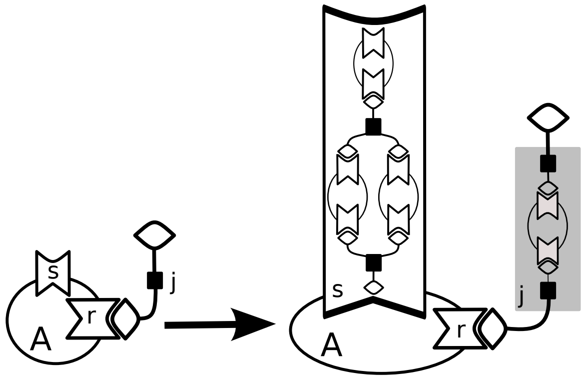


Figure 12: Examples of possible hierarchy in Ports and Connectors: the Port "s" and the Connector "j" of the left-hand model are hierarchically expanded in the right-hand model.

mantic reason to introduce both `nodes` and `connectors` in the language, since they are acting 100% symmetrically in the structural relationships. (This property is sometimes refered to as the *duality* between `nodes` and `connectors`.)

The reason why two different model primitives, `nodes` and `connectors`, are necessary, becomes clear as soon as the *structural* ("composition") model of an application is composed with the *behavioural* ("interaction") models that come from a particular application domain: such behaviours are (typically) put inside `nodes`, while `connectors` are (typically) meant to represent behaviour-free interconnection relations, and `ports` to represent behaviour-free "access" of the connector to the behaviour inside one specific `node`.

However, NPC4 does not want *to impose* in advance the (arbitrary!) choice of where an application will see behaviour fit best, in the structural primitive that it calls a `node`, or in the structural primitive that it calls a `connector`, or even in both or in the `port`. Hence, nothing is put in the NPC4 language that can bias this choice.

Anyway, the (*fundamental*) asymmetry between behaviour-carrying and behaviour-free structural primitives is just a matter of an *arbitrary* selection of the *names* "`node`" and "`connector`". And, moreover, while `connectors` and `ports` are *typically* the behaviour-free parts of a structural model, the NPC4 meta model *allows* both to `contain` sub-models with `nodes`, `connectors` and `ports` (Fig. 12); and hence, if the `nodes` in that composition can have behaviour, also the containing `connectors` and `ports` have behaviour.

Note that the `container` was not part of the hierarchy and behaviour discussion above, for the simple reason that `containers` are not meant to represent behaviour, but only information ("meta data", "knowledge contexts"). `Containers` are also allowed *to overlap*, which is not allowed for `nodes`, `connectors` or `ports`.

3 The NPC4 Domain-Specific Language

The intention of the previous Section was to introduce the concepts to human readers, and now this Section turns these informally introduced concepts into a *formal model* that computers can parse and reason upon. Such a formal language model (which is given the name *NPC4*) is often called a *"meta model"*, or *"modelling language"*, or *"Domain Specific Language"*, or *DSL* for short [21, 26]. (Some other examples of robotics DSLs are [10, 15, 16, 23, 30, 42].) The NPC4 meta model represents the *structural* properties—*hierarchical hypergraphs*— of all the use cases introduced before, in a fully formal, computer-processable way.

3.1 Design drivers

The major design drivers behind the presented NPC4 language are semantic *minimality*, *explicitness* and *composability*:

Minimality. The model represents *interconnection and containment structure*, and only that. No behavioural, visual, software, process,... information is represented.

Explicitness. Every concept, and every relationship between concepts, gets its own explicit keyword:

- `node` for the concept of behaviour *encapsulation.*
- `connector` for the concept of behaviour *interconnection.*
- `port` for the concept of *access* between encapsulated behaviour and each of its interconnections.
- `container` for the concept of *packaging* a model in an entity that can be refered to in its own right.
- `contains` for the relationship of composition into *hierarchies.*
- `connects` for the relationship of composition via *interaction.*

The Eqs. (2)–(3) introduced "informally" in the previous Section are already sufficiently formal to serve as part of the NPC4 DSL. But in addition to these obvious language primitives, extra `has-a` relationships are introduced for `attachment point` primitives, on `nodes` and `connectors`:

- a `node-attachment-point` belongs to a `node`, via an explicit

  `has-a(node,node-attachment-point)`

  relationship, and is meant to receive a `connects` relation with a `port`.

- a `connector-attachment-point` belongs to a `connector`, via an explicit

  `has-a(connector,`
  `connector-attachment-point)`

  relationship, and is also meant to receive a `connects` relation with a `port`.

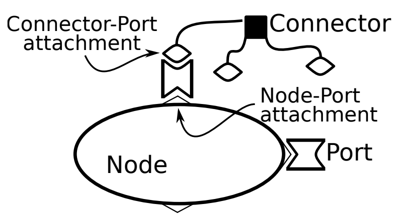


Figure 13: A Connector is an hyperedge that links Ports together, and a Port links a Connector to a Node, via Connector-Port and Node-Port attachment objects.

These primitives allow each node or connector to indicate (explicitly and without needing the other primitives) (i) *how many* interactions it offers, and (ii) to *identify* each of these in a unique way. This is a necessary (but not sufficient) condition for formal reasoning on the *semantic correctness* of interconnections, because the attachment objects are indispensible for A useful side-effect of the explicit introduction of te attachment points primitives shows up in their graphical representation inside software tools: the attachment points are first-class properties of the structural model in itself, but also of every particular graphical visualistion of the model. (This discussion about graphical tooling is beyond the scope of this document.)

Note that Eq. (1) is *not* kept in the formal version of the DSL, but it is replaced by the combination of (i) a "has-a" for Node and its port-attachment points, and (ii) the "connects" of a Port and a port-attachment point.

Composability. The DSL is intended to represent only *structure*, and is, hence, *designed* to be extended (or *composed*) with *behavioural* models: it allows to connect other models to *any* of its own language primitives and relationships, *without* having to change the definition of the language (and hence also its parsers).

So, first of all, an extra keyword is introduced to indicate that all primitives *in NPC4 itself* can be compositions in themselves:

$$\texttt{composite} = \{\texttt{node},\texttt{port},\texttt{connector},\texttt{composite}\}.$$

The recursion in this definition reflects the *hierarchical* property of containment in a natural way.

Secondly, the composition with *other, external* DSLs can be done (in the simple and proven way that, for example, XML-based meta models such as XHTML or SVG use), by providing each primitive in a system with the following meta data that explicitly indicate by which meta model they have to be interpreted:

- `instance_UID`: a *Unique IDentifier* of any instantiation of the primitive concept;

- `model_UID`: a unique pointer to the model that contains the definition of the semantics of the primitive;

- `meta_model_UID`: a unique pointer to the meta model that describes the language in which the primitive's model is written;

- `name`: a *string* that is only meant to increase readability by humans.

Such generic property meta data allows to compose structural model information with domain knowledge by letting each primitive in a composite domain model *refer* to (only!) the structural model that it *"conforms to"* [6]; such composition-by-referencing is a key property of a language to allow for composability.

Finally, since NPC4 is a language for structural composition, it deserves a separate keyword `compose` to refer to one or both of its two possible composition relationships, namely `contains` and `connects`:

$$\texttt{compose} = \{\texttt{contains}, \texttt{connects}\}.$$

The motivation for the *explicitness* design driver is that (i) *each* of the language primitives can be given its own properties and, more importantly, its own extensions, independently of the others, (ii) it facilitates *automatic reasoning* about a given model because all information is in the keywords (and, hence, none is hidden implicitly in the syntax), and (iii) it facilitates *automatic transformation* of the same semantic information between different formal representations. Such *model-to-model* transformations become steadily more relevant in robotics because applications become more complex, and hence lots of different components and knowledge have to be integrated. Trying to do that with one big modelling language becomes increasingly inflexible, because it will be impossible to avoid (partial) overlaps of the many DSLs that robotics applications will eventually have to use in an integrated way.

In the same context, *composability* can only be achieved if none of the DSLs puts any restrictions on any of the other ones; and, even better, that each language is *designed* to be integrated with *any* other language (as long as that other language is also designed for composablity).

The NPC4 meta model is, in itself, already a language that extends that of traditional graph theory. Traditional graphs, offering only vertices and edges as primitives, are the meta meta model of NPC4: the `node`, `port` and `connector` primitives in NPC4 are *extensions* of the traditional vertex, and their interconnections are *extensions* of the traditional edge. In other words, NPC4 composes the DSL of traditonal graphs with the *node-port-connector* structural semantics. Hence, traditional graph relationships and properties hold for NPC4 too, for example: *adjacency*, *incidence* and *paths* of connected primitives in a graph; the *diameter* of a graph; or *directed* edges. NPC4 adds extra semantics to these concepts by restricting defining them to `node`s only.

3.2 Constraints

Section 3.1 introduced the *primitives* of the NPC4 language, and the `contains` and `connects` *relationships* that can exist between these primitives. However, not all relationships that can be formed syntactically also have semantic meaning. So, some *constraints* must be added, as explained in the following paragraphs. Note that no `connects` relationships appear anywhere in the constraints on the `contains` relationships, and the other way around, which reflects the above-mentioned *orthogonality* of both relationships. Of course, when application developers add behaviour to a structural model of their system, they may introduce *extra* structural constraints, even between `connects` and `contains` relationships.

Constraints on primitives. The `UID` of every primitive must be unique:

$$\begin{aligned}\forall \texttt{X},\texttt{Y} \in \{&\texttt{node}, \texttt{port}, \texttt{connector},\\ &\texttt{node-attachment-point},\\ &\texttt{connector-attachment-point},\\ &\texttt{contains}, \texttt{connects}\},\\ \texttt{X.UID} = \texttt{Y.UID} &\Rightarrow \texttt{X} = \texttt{Y}.\end{aligned}$$

Of course, these constraints hold for all three UIDs in the meta data of each NPC4 primitive.

Constraints on `connects`. The constraints in this Section realise the *well-formedness* of the connection relationships, that is, about which kind of structural interconnections are possible.

Every attachment point *must* be connected to either a `node` or a `connector`:

$$\begin{aligned}&\forall \texttt{Y} \in \{\texttt{node-attachment-point}\},\\ &\quad \exists ! \texttt{N} \in \{\texttt{node}\}\ \texttt{and connects(N,Y)}\\ &\forall \texttt{Y} \in \{\texttt{connector-attachment-point}\},\\ &\quad \exists ! \texttt{C} \in \{\texttt{connector}\}\ \texttt{and connects(C,Y)}.\end{aligned}$$

If a `port` is connected, it *must* be connected to one and only one `node`, and/or to one and only one `connector`:

$$\begin{aligned}&\forall \texttt{P} \in \{\texttt{port}\}, \forall \texttt{N1},\texttt{N2} \in \{\texttt{node}\}, \forall \texttt{C1},\texttt{C2} \in \{\texttt{connector}\},\\ &\quad \texttt{connects(N1,P)}, \texttt{connects(N2,P)} \Rightarrow \texttt{N1} = \texttt{N2}\\ &\quad \texttt{connects(C1,P)}, \texttt{connects(C2,P)} \Rightarrow \texttt{C1} = \texttt{C2}.\end{aligned}$$

A `node` and a `port` can only be connected through a `node-attachment-point`:

$$\begin{aligned}&\forall \texttt{N} \in \{\texttt{node}\}, \forall \texttt{P} \in \{\texttt{port}\} : \texttt{connects(N,P)}\\ &\Leftrightarrow \exists \texttt{X} \in \{\texttt{node-attachment-point}\} :\\ &\qquad \texttt{connects(N,X)}, \texttt{connects(X,P)}.\end{aligned}$$

Similarly for `connector`s and `port`s:

$$\begin{aligned}&\forall \texttt{C} \in \{\texttt{connector}\}, \forall \texttt{P} \in \{\texttt{port}\} : \texttt{connects(C,P)}\\ &\Leftrightarrow \exists \texttt{X} \in \{\texttt{connector-attachment-point}\} :\\ &\quad \texttt{connects(C,X)}, \texttt{connects(X,P)}.\end{aligned}$$

A `node` and a `connector` can only be connected through a `port`:

$$\begin{aligned}&\forall \texttt{N} \in \{\texttt{node}\}, \forall \texttt{C} \in \{\texttt{connector}\} : \texttt{connects(N,C)}\\ &\Leftrightarrow \exists \texttt{P} \in \{\texttt{port}\} : \texttt{connects(N,P)}, \texttt{connects(C,P)}.\end{aligned}$$

Of course, more than one such `node-connector` connection can exist.

A `connect` relationship can only be defined on *existing* primitives:

$$\forall \texttt{c} \in \{\texttt{connects}\},\\ \exists \texttt{X,Y} \in \{\texttt{node,port,connector,}\\ \texttt{node-attachment-point,}\\ \texttt{connector-attachment-point}\} :\\ \texttt{c(X,Y).}$$

The `connects` relationship is not *commutative* or not *transitive*, but it is always *symmetric*:

- a node can *only* be connected to itself via a connector (or a more complex composition) that connects two or more of the node's ports, but it can not connect to itself directly.
- if `node_A` is connected to `node_B`, and `node_B` is connected to `node_C`, it is *possible* that `node_A` is connected to `node_C`, but not necessarily so.
- if `node_A` is connected to `node_B`, then `node_B` is connected to `node_A`.

3.2.1 Constraints on `contains`

The constraints in this Section realise the *well-formedness* of the containment relationships of nodes, that is, about which kind of hierarchies, or "composites" are possible.

First, the fact that *every* primitive *can* be a `composite` in itself is expressed:

$$\texttt{composite} = \{\texttt{node,port,connector,composite}\},\\ \forall \texttt{C} \in \{\texttt{composite}\} :\\ \exists \texttt{n} \in \{\texttt{node}\} \vee \exists \texttt{c} \in \{\texttt{connector}\}\\ \vee \exists \texttt{d} \in \{\texttt{composite}\} :\\ \texttt{contains(C,n)} \vee \texttt{contains(C,c)}\\ \vee \texttt{contains(C,d).}$$

Every `contains` relationship can only be defined on *existing* primitives:

$$\forall \texttt{c} \in \{\texttt{contains}\},\\ \exists \texttt{X,Y} \in \{\texttt{node,port, connector,composite}\} :\\ \texttt{c(X,Y).}$$

And finally, there *always* exists at least one `node` at the top of a `contains` hierarchy:

$$\forall \texttt{X} \in \{\texttt{node,port,connector,composite}\},\\ \exists \texttt{T} \in \{\texttt{node}\} : \texttt{contains(T,X)} \vee \texttt{T=X}.$$

This latter constraint is a *very strong* one, that is imposed for one and only one reason: every structural model should have an explicitly identified *context*. In other words, the meta data of the top node must be made rich enough to understand the semantics of *everything* it `contains`, even when the model is deployed in a running system. There can be *more than one context* for each composition, which is in agreement with the design objective of composability: a composite can *conform to* more than one meta model. The top `node` need not have any `port` attached to it, so that it is a *container of meta data* only.

3.3 Host DSL languages

The previous Sections introduces a formal representation of the *semantics* of the NPC4 language, using first order logic. However, such a *declarative* definition is seldom the most compact, human-readable or user-friendly way to let practiners in a particular domain *apply* the meta model effectively. Such application efficiency is determined by many factors, that have less to do with the semantics than with pragmatic motivations within in each user community. For example, users are already familiar with particular formal languages, and prefer not to have to learn new editors, tools or syntax. Because of the familiarity and tooling arguments, XML, Lisp, or Prolog are primary candidates "to host" DSLs. Another popular approach is to provide the DSL in the form of a library in a general-purpose programming language such as C++, Java or Lua, as a so-called *internal DSL*; e.g., [30, 31]. Whatever choice is being made, *enforcing* the *semantics* of the DSL in a host language almost invariably requires the development of a "runtime" that checks all the constraints of the DSL; of course, the *implementation* of that runtime should be checked for its conformance with the DSL [6]. Such a check, fortunately, has to be done only once.

NPC4 is about the structure of hierarchical hypergraphs, and exactly this semantics is something for which the above-mentioned popular host languages have little to no "native" support. On the contrary, in Lisp or XML hierarchy is most often not represented explicitly, but as a result of the *syntactic* structure of the language: the "nesting" in Lisp represents hierarchy (more in particular, expression *trees*, not *graphs*) via matching parentheses (Table 1), while XML achieves the same goal via matching nested tags as in Table 2. The non-intended, but often occurring, result is that

- reasoning or transformations on compositions can only take place after parsing;
- the decision about which structures in the parsed

```
(tag
  (tag1 (id "abc") ...)
  (tag2 (id "xyz"} ...)
)
```

Table 1: Example of an *operational* representation, in Lisp, of a hierarchical composition.

```
<tag>
  <tag1 id="abc"> ... </tag1>
  <tag2 id="xyz"> ... </tag2>
</tag>
```

Table 2: Example of an *operational* representation, in XML, of the same hierarchical composition as in Table 1.

"abstract syntax tree" have semantic meaning and which don't, is not represented explicitly but hidden in the implementation of the parser;

- the hierarchical composition itself can not be given properties (such as a specific visual icon in a graphical programming tool), or be extended with other "behaviour" itself, because there is no language primitive to compose such properties or extensions with.

Of course, nothing in Lisp or XML prevents DSL designers to represent the `contains` or `connects` relationships explicitly, so both languages are definitely valid candidates to host NPC4.

The example of using Lisp or XML to host the same DSL also illustrates what *model-to-model transformation* means: the exact same semantics *can* be represented in a Lisp model and in an XML model, so a model in one language should be transformable into a model in the other language. However, in order for such a transformation to be done correctly, both languages *must refer to* the same external DSL that defines the meaning of the keyword. The state of the art still misses three important things: (i) the discipline of language designers to use such external DSL semantics whenever that makes sense, (ii) the mere existence of formal and composable DSLs, and (iii) the software tooling to support *correct-by-construction* editors of the DSLs and the *Triple Graph Grammar* [18, 19, 41] *model-to-model transformations* required by DSLs with hierarchical hypergraph relationships.

Two (rather composable) examples of XML-hosted DSLs in other domains than robotics are *Xcore* [45] from the Eclipse eco-system, and *Collada* [5] from the computer animation domain. Both DSLs have explicit tags *to refer* to external, application-specific DSLs; Xcore also explicitly supports hierarchy, via it `contains` and `container` keywords.

URDF [49] or *SRDF* [13, 32] are examples created in the robotics community, but, unfortunately, they go against the design principles advocated in this paper, by following the *extension by inheritance* approach instead of the *extension by composition*: if the language designers want to model a new feature, they add a new keyword to the URDF language. In the medium term, this will lead to an overloaden, huge and not semantically consistent or complete "standard", such as is the case with UML or CORBA. Because of its *composability* and *minimality* design drivers, the DSL approach suggested in this paper has a higher chance (but no guarantee!) of leading to a lot of small modelling languages that are semantically correct, and can (hence) be integrated via small, application-specific DSLs without the application developers having to spend time on making their own big languages.

4 Examples

This Section gives some concrete examples about how NPC4 can be used to represent the *structural* parts of systems in the various application domains introduced in Sec. 1. The examples show the two complementary ways in which such *domain-specific extensions* are made:

- give new, domain-specific *names* to the NPC4 primitives and/or relationships.

- add domain-specific *extra semantics* to the NPC4 primitives, relationships and constraints.

For all examples, only the *NPC4-inspired approach* is explained, but not the concrete DSLs that could result from it, since each of them requires a significant extra effort to be developed.

4.1 Bayesian Networks

The domain of *traditional* Bayesian network only uses `node` (for "random variables") and `connect` (for "directed edges"), and no `contains`. The more recent Factor Graph extension was introduced needed to represent explicitly the *hyperedge* connectivity that has been part of the domain since the beginning. For one reason or another, *hierarchy* has never been introduced to the full extent, such that the domain can still not model complex Bayesian networks in which various sub-networks can be given other contexts, for example, for decision

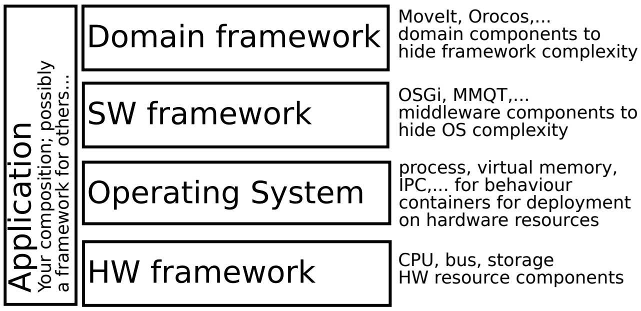


Figure 14: The four natural levels of hierarchy generally present in "software stacks", plus the concept of an "application" which composes them all together in one executable software system.

making, or for scheduling of the computational execution.

4.2 Finite State Machines

This domain has been making use of the full and correct semantics of hierarchical hypergraphs since the beginning [30]:

- `nodes` are used to represent states in the FSMs.
- `connects` are used to represent two concepts: (i) the transitions between states, and (ii) the events that are fired or handled in a state. A transition *must* connect only two states; an event *can* be reacted to by multiple states, and to model that is the use case of the `contains` hierarchy.

4.3 Control diagrams

This domain applies the hierarchical hypergraphs meta model as follows:

- `nodes` are used to represent *function blocks*.
- `connects` are used to represent *data flows*, also with hyperedge semantics.

Hierarchy is used to model context ("plant", "controller", etc.) and to cope with complexity of composition, by introducing `nodes` with various "levels of detail".

4.4 Software architecture models

The authors' recent publication [47] provides an extensive overview of how the hierarchical hypergraph meta model can be applied to the modelling and composition of software systems. (Even without a formally specified DSL, but relying on discipline of the developers.) Roughly speaking, the mapping from NPC4 to the domain of software engineering is very similar to that for control diagrams; the major semantic difference being that the `nodes` represent also software activities (processes, threats, computing nodes,...) and not just computational function blocks. The resulting data flow between such `nodes` typically involves *communication middleware*, whose models (structural and behavioural) are typically "hidden" in a multi-layer hierarchical structure of the system architecture, as illustrated in Fig. 14.

A major use case for an NPC4-based DSL in software architectures will be *deployment models*, that is, to represent the dependencies between the software modules that determine their relative order of creation, configuration, and activation.

4.5 Robot kinematics and dynamics

This Section gives a brief overview of how the meta model of hierarchical hypergraphs can provide a more composable DSL than the mainstream URDF format, [49]; a much more elaborate document on this particular topic is currently under development, which has also the explicit aim to be able to serve as a very flexible and composable family of modelling standards. The core idea behind it is illustrated in Fig. 15:

- the family has five members, each one being a natural hierarchical `contains` context of another one.
- each level has *creates* a DSL of its own, with several new semantic primitives, relationships and constraints.
- each level *composes* a specific subset of the possibly multiple DSLs at the lower levels, not by adding as *properties* in its own DSL primitives, but as `connects` *references* to the DSLs below it.
- similarly, each level *composes* its domain DSL, with `has-a` relationships, with a DSL that represents *physical units*, [40].
- *composition* into a *chain* can only work if the four other levels compose themselves with the *same* DSL on *geometric frames*, [16], as the fundamental `connects` primitive of electro-mechanical systems.

For example, the approach introduced above allows to make a DSL for humanoid robots with only electrical actuators acting at each individual joint, but also for real human musculoskeletal models with muscles connected over multiple joints. When done wisely, both DSLs will share most of their semantic primitives (which supports the objectives of minimality and efficiency), but still be able to provide only those semantic primitives that are really needed (which support the objective of user-friendliness).

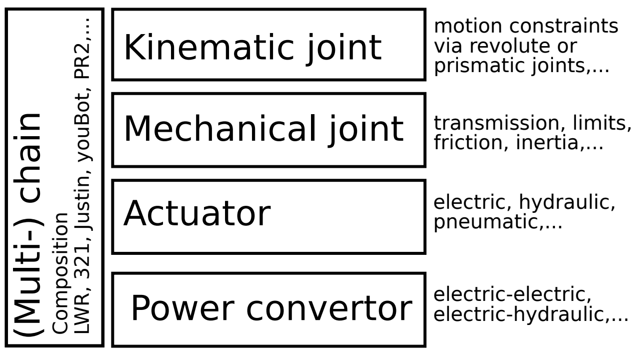


Figure 15: The four natural levels of hierarchy needed to describe the mechanical structure of all kinds of robots, plus the concept of a "kinematic chain" which composes them all together in one "robot system".

4.6 Visualisation with the Model-View-ViewModel pattern

The increasing complexity of robotic systems also increases the complexity of presenting all the generated data to the users in an intelligible way. Again, best practice in this domain uses de-composition and hierarchy to tackle this problem, for example, by providing:

- different *views* on the same data: table form, 3D visualisation of the moving parts, layered maps, etc.
- different visualisations for each knowledge *context* of the system, the various *levels of abstraction* in its models, or the various *levels of detail* in the data.

For all of these, the need for *hyperedge* `connects` relationships, and hierarchical `contains` relationships is obvious. To support that need, the *Model-View-ViewModel* pattern for the design of graphical user interfaces is more and more replacing the more traditional *Model-View-Control* paradigm; the "web app' framework AngularJS [25] is a key example of this evolution. Unfortunately, the terminology it uses is *very* different from the "port-based" terminology of this paper, which complicates the semantic mapping between NPC4 and AngularJS primitives, and, hence, the efficiency of leveraging the large momentum in building "apps" to the more narrow context of robotics.

5 Conclusions

This paper advocates the use of the NPC4 language, as *the* meta model to represent *port-based composition*, for both interconnection and containment, and in a domain-independent way. More in particular, the language targets *all* man-made engineering systems based on *lumped parameter* models.

The objectives behind NCP4 are already covered, partially, in engineering languages such as Modelica [34], but the new contributions of this paper are: (i) to separate strictly the structural and behavioural aspects, and (ii) to make *all* structural relationships *explicit* in a formal language, based on *hierarchical hypergraphs*.

The motivation for this paper is that all current practice relies on many *implicit* specifications of, especially, their structural relationships, and more in particular the "`contains`" and "`connects`" relationships. Only an explicit representation of both will allow an engineering systems *to reason* about its own structure, at runtime, and *by itself*.

This requirement of being able to reason on "`contains`" and "`connects`" relationships becomes *mandatory* to deal with *knowledge-centric* systems: their behaviour always depends on the specific *context* in which various pieces of the knowledge integrated in the system are valid or not. Hence, it is important to have an explicit computer-readable representation of the *structural knowledge contexts* in which a system is contained; most often, there are many overlapping contexts active at the same time. Hence, the hierarchical hypergraph meta model is highly relevant to make the step from traditional engineering systems to *knowledge-aware* engineering systems, that is, systems that can *use the knowledge themselves at runtime*.

In the above-mentioned context, the aspect of *composability* of structural models is an important design focus; NPC4 advocates that extra "features" (such as behaviour or visualisation) should not be added "by inheritance" (that is, by adding attributes or properties to already existing primitives), but "by composition", that is, a *new* DSL is made, that imports already existing DSLs and adds *only the new* relationships and/or properties as *first-class* and *explicit* language primitives. The many examples of graphical models taken from the robotics domain, and especially their high degree of non-composability, should be sufficient motivation for practitioners in the field to start adopting NPC4's composability approach.

Although presented in a robotics context, nothing in NPC4 depends on this specific robotics domain, and NPC4 can also serve the goals of related research and application domains such as *Cyber-Physical Systems* or the *Internet of Things*. However, the advantages of the NPC4 meta model pay off most in robotics, because of (i) the large demand for *knowledge-aware* systems, (ii) the online efficiency and (re)configuration flexibility of such robotics systems, and (iii) their need for the online reasoning about—and eventually the online adaptation of—their own structural architectures.

Finally, the authors suggest the NPC4 language for adoption as an *application-neutral standard*, since standardizing the structural part of components, knowledge, or systems, is a long-overdue step towards higher efficiency and reuse in robotics system modelling design, and in the development of reusable tooling and (meta) algorithms.

The hope is that NPC4 is simple, neutral, versatile, customizable and semantically clear and complete enough to stimulate educators, researchers and software developers to pay more attention to modelling, and—not in the least!—*to standardize* their structural modelling approaches.

Unfortunately, even after 50 years of disappointing experiences with respect to standardization in the domain of robotics, many practitioners are not motivated to help create and accept standardization efforts. It is beyond the scope of this paper to explain why and how well-designed and neutral standards are indispensable for the domain of robotics to transition from small-scale academic or industrial development groups to a large-scale, multi-vendor industry. However, the major design principles behind this paper (*minimality*, *explicitness* and *composability* of the DSL) have been strongly motivated by the just-mentioned unfortunate situation of lack of standards in robotics: it is the authors' believe that the high complexity and variability of robotics as a scientific and engineering discipline is exactly due to the pressure of the *open world assumption*: no model of the world is ever complete, or has the right level of detail for the many different use cases that the domain has to support. So, starting with first separating out the simplest part of complex systems—namely its *structural model*—from their more complex behavioural aspects, provides the path of least effort to reach the stated long-term goal.

References

[1] R. R. Allen and S. Dubowsky. Mechanisms as components of dynamic systems: A Bond Graph approach. *J. of Elec. Imag.*, pages 104–111, 1977.

[2] N. Ando, T. Suehiro, and T. Kotoku. A software platform for component based RT-system development: OpenRTM-Aist. In *Conf. Simulation, Modeling, and Programming of Autonomous Robots*, pages 87–98, Venice, Italia, 2008.

[3] G. Antoniou and F. van Harmelen. *A Semantic Web Primer*. MIT Press, 2nd edition, 2008.

[4] C. Atkinson and T. Kühne. Model-driven development: a metamodeling foundation. *IEEE software*, 20(5):36–41, 2003.

[5] M. Barnes and E. L. Finch. COLLADA—Digital Asset Schema Release 1.5.0. http://www.collada.org, 2008. Last visited August 2013.

[6] J. Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171–188, 2005.

[7] S. Borgo and C. Masolo. Full mereogeometries. *The Review of Symbolic Logic*, 3(4):521–567, 2010.

[8] F. T. Brown. *Engineering System Dynamics, a Unified Graph-Centered Approach*. CRC Press, 2nd edition, 2006.

[9] H. Bruyninckx. Open robot control software: the OROCOS project. In *Int. Conf. Robotics and Automation*, pages 2523–2528, Seoul, Korea, 2001.

[10] H. Bruyninckx and J. De Schutter. Specification of force-controlled actions in the "Task Frame Formalism": A survey. *IEEE Trans. Rob. Automation*, 12(5):581–589, 1996.

[11] H. Bruyninckx and P. Soetens. Open RObot COntrol Software (OROCOS). http://www.orocos.org/, 2001. Last visited March 2013.

[12] H. Bruyninckx, P. Soetens, and B. Koninckx. The real-time motion control core of the Orocos project. In *Int. Conf. Robotics and Automation*, pages 2766–2771, Taipeh, Taiwan, 2003.

[13] S. Chitta, K. Hsiao, G. Jones, I. Sucan, and J. Hsu. Semantic Robot Description Format (SRDF). http://www.ros.org/wiki/srdf, 2012.

[14] Controllab Products B.V. 20-sim. http://www.20sim.com/. Accessed online 2 August 2013.

[15] E. Coste-Maniere and N. Turro. The MAESTRO language and its environment: specification, validation and control of robotic missions. In *Proc. IEEE/RSJ Int. Conf. Int. Robots and Systems*, pages 836–841, Grenoble, France, 1997.

[16] T. De Laet, S. Bellens, R. Smits, E. Aertbeliën, H. Bruyninckx, and J. De Schutter. Geometric relations between rigid bodies (Part 1): Semantics for standardization. *IEEE Rob. Autom. Mag.*, 20(1):84–93, 2013.

[17] T. De Laet, H. Bruyninckx, and J. De Schutter. Shape-based online multitarget tracking and detection algorithm for targets causing multiple measurements: Variational Bayesian clustering and lossless data association. *IEEE Trans. Pattern Anal. Machine Intell.*, 33(12):2477–2491, 2011.

[18] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer, 2006.

[19] G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, and B. Westfechtel, editors. *Graph Transformations and Model-Driven Engineering*. Springer, 2010.

[20] J. F. Ferreira, M. Castelo-Branco, and J. Dias. A hierarchical Bayesian framework for multimodal active perception. *Adaptive Behavior*, 20(3):172–190, 2012.

[21] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 2010.

[22] F. Francis Colas, J. Diard, and P. Bessière. Common Bayesian models for common cognitive issues. *Acta Biotheoretica*, 58(2–3):191–216, 2010.

[23] E. Gat. ALFA: a language for programming reactive robotic control systems. In *Int. Conf. Robotics and Automation*, pages 1116–1121, Sacramento, CA, 1991.

[24] P. Gawthrop and L. Smith. *Metamodelling: Bond Graphs and Dynamic Systems*. Prentice Hall, 1996.

[25] Google Inc. Angular JS. http://angularjs.org. Last visited September 2014.

[26] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley and Sons, 2004.

[27] Groupe de Recherche en Robotique. Proteus: Platform for RObotic modeling and Transformations for End-Users and Scientific communities. http://www.anr-proteus.fr/.

[28] P. Hintjens. ØMQ—The guide. http://zguide.zeromq.org, 2013. Last visited July 2014.

[29] S. Joyeux. ROCK: the RObot Construction Kit. http://www.rock-robotics.org, 2010. Last visited November 2013.

[30] M. Klotzbücher and H. Bruyninckx. Coordinating robotic tasks and systems with rFSM Statecharts. *J. Softw. Eng. in Robotics*, 3(1):28–56, 2012.

[31] M. Klotzbücher and H. Bruyninckx. A lightweight, composable metamodelling language for specification and validation of internal domain specific languages. In *Proceedings of the 8th International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, volume 7706 of *Lecture Notes Comp. Science*, pages 58–68. 2012.

[32] L. Kunze, T. Roehm, and M. Beetz. Towards semantic robot description languages. In *Int. Conf. Robotics and Automation*, pages 5589–5595, Shangai, China, 2011.

[33] L. Ladický, P. Sturgess, K. Alahari, C. Russell, and P. H. S. Torr. What, where and how many? Combining object detectors and CRFs. In *2010 European Conference on Computer Vision*, volume 6314 of *Lecture Notes in Computer Science*, pages 424–437. Springer, 2010.

[34] Modelica Association. Modelica: Language design for multi-domain modeling. http://www.modelica.org/. Last visited September 2014.

[35] National Institute of Advanced Industrial Science and Technology, Intelligent Systems Research Institute. OpenRTM-Aist. http://www.openrtm.org. Last visited August 2013.

[36] NN. Web components. http://webcomponents.org. Last visited September 2014.

[37] Object Management Group. Meta Object Facility (MOF) core specification. http://www.omg.org/technology/documents/formal/data_distribution.htm, 2006.

[38] H. M. Paynter. *Analysis and design of engineering systems*. MIT Press, 1961.

[39] H. M. Paynter. An epistemic prehistory of Bond Graphs. In P. Breedveld and G. Dauphin-Tanguy, editors, *Bond Graphs for Engineers*. 1992.

[40] H. Rijgersberg, M. F. J. van Assem, and J. L. Top. Ontology of units of measure and related concepts. *Semantic Web*, 4(1):3–13, 2013.

[41] A. Schuürr. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes Comp. Science*, pages 151–163. 1995.

[42] R. Simmons and D. Apfelbaum. A task description language for robot control. In *Proc. IEEE/RSJ Int. Conf. Int. Robots and Systems*, pages 1931–1937, Vancouver, British Columbia, Canada, 1998.

[43] D. B. Stewart, R. A. Volpe, and P. K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Trans. Software Engineering*, 23(12):759–776, 1997.

[44] M. Tenorth and M. Beetz. KnowRob—A knowledge processing infrastructure for cognition-enabled robots. *Int. J. Robotics Research*, 32(5):566–590, 2013.

[45] The Eclipse Foundation. Xcore. http://wiki.eclipse.org/Xcore, 2013.

[46] The MathWorks. Simulation and model-based design by The MathWorks. http://www.mathworks.com/products/simulink/.

[47] D. Vanthienen, M. Klotzbücher, and H. Bruyninckx. The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming. *J. Softw. Eng. in Robotics*, 5(1):17–35, 2014.

[48] W3C. Owl. http://www.w3.org/TR/owl-ref/.

[49] Willow Garage. Universal Robot Description Format (URDF). http://www.ros.org/wiki/urdf, 2009.

[50] World Wide Web Consortium. HTML5. http://www.w3.org. Last visited September 2014.