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

Enea Scioni1,2, Nico Hübel1, Sebastian Blumenthal1,4, Azamat Shakhimardanov1, Markus Klotzbücher1, Hugo Garcia1, Herman Bruyninckx1,3

1KU Leuven, Belgium
2Università di Ferrara, Italy
3Eindhoven University of Technology, the Netherlands
4Locomotec, Augsburg, Germany

Abstract—Many robotics applications rely on graph models (that is, nodes and edges) 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 the research hypothesis that hierarchical hypergraphs are better models than traditional graph models to represent the structural properties of systems: (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 can in itself be another hierarchical hypergraph. These properties are encoded formally in a Domain Specific Language (or “a meta model of a language”), called “NPC4”, built with node, port, connector, and container as primitives, and contains and connects as relationships. These two relationships are key to the formal description of topology, which complements other key relationships in robotics systems such as “is-a” (behaviour) and “has-a” (composition, aggregation), which are already well covered by modelling languages like UML or AADL. The structural model described in NPC4 form can be further enriched with additional domain-dependent constraints that define a more concrete Domain Specific Language (DSL). NPC4 introduces a particular “contains” primitive, the container, to support overlapping contexts, which is important in knowledge-centric robotics systems to model (i) various levels of abstraction in domains, (ii) “multiple inheritance” from (or rather “conformance to”) different knowledge domains, and (iii) connecting one or more domain DSLs to the same software infrastructure in which they all have to be “activated”.

Index Terms—Domain Specific Language, meta meta model, composability, structural modelling, knowledge representation

1 INTRODUCTION

Everywhere in robotics, graph-based structures (that is, compositions of “nodes” and “edges”) show up as models of concepts, knowledge, software, and systems, to name just a few examples. Graph models are good at separating the structural and behavioural parts of a design, that is, the undirected graph represents which nodes interact with which other nodes, without describing the behaviour inside the nodes, or of the interaction dynamics between nodes. Below is a non-exhaustive list of examples in robotics, where nodes, edges and sometimes ports are the building blocks of the graph-based structural models. The Appendix provides more details about how the structure in each of the specific domains supports the

www.joser.org - © 2016 by E. Scioni, N. Hübel, S. Blumenthal, A. Shakhimardanov, M. Klotzbücher, H. Garcia, H. Bruyninckx
domains' behaviour; the insight that the reader should get from this list of Figures and examples is that a quite limited number of modelling primitives suffice to support all structural aspects of these robotics sub-domains. The objective of the paper is to formalize this insight into a Domain Specific Language for the structural properties of systems; their behavioural properties are then still to be composed onto the structural model, by adding node and edge types, each with many different behavioural semantics. The concrete examples are:

- software architectures, as in Fig. 1;
- kinematics and dynamics of actuated mechanical structures, see Fig. 2.
- Finite State Machines (FSM), as in Fig. 3; a transition is structurally an edge that represents a relationship between two states (i.e., nodes), enriched with domain-specific behaviour dependent on the FSM formalism applied (e.g., guard conditions, events, priorities to which the transition is subject to, etc.);
- probabilistic graphical models such as Bayesian Networks or Factor Graphs, Fig. 4;
- control diagrams, such as the Cartesian position control scheme of Fig. 5, as well as other “data flow” computational models such as Simulink [2], where the nodes host functions (e.g. integrator and gain blocks), and the edge orientation indicates inputs and outputs (e.g. \( \dot{x} = y \) or \( \dot{y} = x \) for an integrator block);
- knowledge representation networks, such as the “semantic web”;
- web applications, in which HTML5 [3] brings a significant change in the way that structure and behaviour are being separated in a clean but composable way.

All graph models 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 [4], or HTML5) as a “view” on a part of the internal state of the node it is connected to, and serving as an explicit “attachment point” for interactions via edge connectors.

The central research hypothesis in this paper is that the concept of the hierarchical hypergraph is a compositional representation to cover all the structures discussed above, particularly, via the containment and connection relationships. The formal version of this structural representation is the NPC4 meta model; it conforms to the meta meta model of the mathematics of hierarchical hypergraphs, adding the semantics of system composition.

A meta model (or, modelling language) is a language with which to create concrete models of a system in a particular application domain or context. A meta meta model is a domain-independent “language” to support the creation of domain-dependent meta model languages, or Domain Specific Languages (“DSLs”). It is beyond the scope of this work to provide details on DSL concepts, therefore the reader can find further information in [5], [6], [7], [8], [9], while some examples of DSLs in robotics are [10], [11], [12], [13], [14], [15].

This paper’s refutable research hypothesis is that NPC4 provides:

- a separation between structure and behaviour;
- a minimal set of language primitives and relationships to describe the structural part of graph-based models that are relevant in the robotic domain;
- a methodology to make a new DSL by only having (i) to specialize the interpretation of NPC4’s primitives (node,
port, connector, container) to the domain, and (ii) to add constraints to the contains and connects relationships.

The minimality concept pertains the expressivity of NPC4: each primitive and relationship does not add expressivity in the structural model, but syntactic sugar to simplify the structural specification in some domains, that is, a derived DSL [16].

The validity of the presented approach is motivated by the large set of relevant examples that conform to the NPC4 meta model; its refutation could come from showing that the above-mentioned goals are not reached (e.g., some primitives are redundant), or are already covered completely by existing modelling languages like UML or AADL, which are very prominent in the industrial practice [17].

The focus on only the generic structural aspects of robotics systems can hopefully lead to a step change in reuse of software:

- **reuse of syntactical parsing code**: the structure of a DSL is explicitly visible through the language’s syntax, and since NPC4 provides a common structural basis to DSL builders, they should be able to reuse a lot of the parsing software;

- **reuse of infrastructure code**: every DSL that is being introduced in a robotics system requires more support from the system’s infrastructure code than only the realisation of the modelled domain functionalities, e.g., logging, messaging, debugging, tracing, and so on. NPC4 provides all the “hooks” to connect these non-nominal software requirements too;

- **reuse of “Model-to-X” transformation tooling**: models are declarative specifications of domain functionalities, and inevitably needs to be transformed into code that supports turning the declarative specifications into procedural code, and basing different DSLs onto the same NPC4 core simplifies reuse of such model transformation tools.

**Overview.** 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. It also creates a fully formal language for hierarchical hypergraphs, in the form of a 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 contains relationship represents hyperedges. Section 3 discusses and formalizes the constraints and properties integrated into the NPC4 language, while Section 4 illustrates its composability features. Section 5 revisits two of the use-cases introduced above in more details, explaining how NPC4 can be used as the basis for their structural models, and the resulting benefits. Finally, Section 6 gives details on prior work and how the proposed NPC4 can be seen as a step-forward regarding models standardisation in robotics.

## 2 HIERARCHICAL HYPERGRAPHS

This section proposes the adoption of hierarchical hypergraphs in the robotics domain, instead of traditional graphs, as its main structural model. The motivation is based on the list of examples in Sec. 2.1 that illustrate various ways in which the use of traditional graphs introduces erroneous ways of representing and 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 then introduced in Sec. 2.4.
2.1 Motivations and bad practices

Traditional graphs have nodes and edges as model primitives, and most practitioners feel very comfortable with using them as graphical primitives for modelling. However, traditional graphs have a rather limited expressivity with respect to modelling the structural properties of a system design. The paragraphs below explain commonly occurring “bad practices” in using traditional graphs.

An edge can only connect two nodes, while many structural interactions are so-called n-ary relationships, that is, more than two (i.e., “n”) entities interact at the same time, and influence each other’s behaviour.

Obvious examples of n-ary relationships are “knowledge relationships”, such a conditional probability in a Bayesian network Fig. 4. 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 by allowing to interconnect various levels of abstraction when modelling a system.\(^1\)

For example, a kinematic model of a robot structure might be enough for motion planning, but the dynamics of its actuators might be needed to design the robot’s motion controllers. Since the actuators are mechanically connected to the kinematic chain of the robot, a hierarchical structural model would apply perfectly to support the separation between the kinematic and dynamic models of the same robot.

Also 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 which it is used. Context is an indispensable structure in coping with the information in, and about, complex systems; in software implementations, this is most visible in how values of configuration parameters are to be determined.

A third prominent “bad practice” example of (too) “flat” structural models are the popular (open source) robotics software frameworks, like ROS or Orocos: they do not support hierarchical composition of software components, 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. Further details will be presented in Section 5.

Edges have no levels of abstraction, and just serve as topological symbols representing the immutable, logical state of different nodes being “connected” or “not connected”.

However, almost all of the use cases in the introduction have edges that can exhibit dynamics when opened up to a deeper level of abstraction; e.g., the communication channels between software components (time delays, buffering,...), the mechanical dynamics of joints and actuators in robotics hardware, and so on. The structure introduced in this paper advocates a clear and systematic rule: behaviour is only placed in nodes, at any particular level of abstraction of the model. When going to a more detailed level of abstraction, it is possible that behavioural nodes “show up” in a part of the model that was just an edge at a higher level of abstraction. For example, an ideal kinematic joint is a perfect constraint between interconnected links, but when going to a more detailed dynamical model level, behaviour will show up in the form of friction, or energy transmission dynamics inside the electrical actuator.

Interactions are uni-directional. Most modelling approaches use directed edges, that is, the graph assumes that each “partner” in an interaction can influence one or more other “partners”, without ever being influenced itself by those partners in any way. Nevertheless, bi-directional interactions are the obvious physical reality: interactions, including man-machine interactions, exchange energy in both directions.

2.2 Primitives, relationships and their semantics

This section introduces a minimal and complete set of primitives and relationships to describe a semantically consistent structural model. The concepts of hyperedges and hierarchy, as key additions to existing graph modelling traditions, aim to prevent the implicit, domain-specific assumptions discussed in the previous section.

---

\(^1\) This paper deals only with hierarchy in structure; the complementary hierarchies in behaviour or mereology are already well modelled by the “is-a” and “has-a” relationships in languages like UML.
The core of the language are the structural relationships of has-a, connects and contains between the model primitives of Node, Port, Connector and Container. The semantic role of a Node is to host a behaviour, while a Connector describes the interaction relationship between the behaviour inside multiple Nodes by “connecting” them. Formally, a Connector realises an hyperedge, since the relationship is not unary but n-ary, and is un-directional by default (that is, unless explicitly constrained not to be so). In traditional graph modelling, a duality property exists between Node and Connector: both can be seen as vertex or hyperedge.

However, this symmetry disappears as soon as the containment relationship is introduced. In fact, the hierarchy concept is orthogonal with respect to the hyperedge connection concept. Hierarchy is expressed by the relationship contains applied to the Node primitive: a Node can contain a full hierarchical hypergraph in itself. The latter is semantically justified by observing that the hosted behaviour by the Node can be structurally represented as composition of internal Nodes and the interactions between them. Note that composition is a primary design driver of the proposed hierarchical hypergraph approach.

To achieve full expressiveness of the structural model, the Port is formally introduced as the third primitive in the language. A Port offers a specific view of a Node, exposing a specific part of a Node’s internal behaviour, and creates structure in the connects relationships across hierarchy levels. As a consequence, the connects relationship involves directly the Port primitive, and not Nodes, as it will be illustrated in the following section.

Finally, a primitive called Container provides a grouping feature, allowing to add extra semantic knowledge to a selected subset of primitives; such “grouping” is known under various names, such as: “context”, “namespace”, “scope”, etc. In contrast to Nodes, Containers can overlap each other, in non strictly hierarchical ways.

### 2.3 Design drivers

The major design drivers to ground the hierarchical hypergraph concepts as a Domain Specific Language are minimalitiy, explicitness and composability, as suggested in [18]:

- **Minimality.** The model represents only interconnection and containment structure. It serves as a skeleton to represent the information about the structural model, but it does not make any assumption on the behaviour present in such a structure.

- **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.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Graphical Convention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port</td>
<td><img src="image1" alt="Port" /></td>
</tr>
<tr>
<td>Node</td>
<td><img src="image2" alt="Node" /></td>
</tr>
<tr>
<td>Connector</td>
<td><img src="image3" alt="Connector" /></td>
</tr>
<tr>
<td>Container</td>
<td><img src="image4" alt="Container" /></td>
</tr>
<tr>
<td>Connection</td>
<td><img src="image5" alt="Connection" /></td>
</tr>
</tbody>
</table>

![Figure 6](image6)

Figure 6: Graphical conventions to represent hierarchical hypergraphs: (i) Port is a square composed of two rectangles which represent (with respect to the Node to which the Port is attached) the internal (black) and external (white) docks; (ii) a Node is a rounded box; (iii) the Connector is shaped as a filled circle; (iv) the Container is represented as a dashed outline. The bottom row shows an example of two Nodes, namely A and B, connected by the Connector j attached to the external docks of Ports p1 and p2. The “clamps” on the docks appear if the docks have been linked to a connector.

- Container for the concept of packaging a model in an entity that can be referred to in its own right.
- has-a for the mereology relationship, that is, representing the parts present in the system, irrespective of the two structural relationships below.
- contains for the relationship of composition into hierarchies.
- connects for the relationship of composition via interaction.

**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 or other supporting software and tooling).

### 2.4 Formalisation into the NPC4 language

The previous section provided an overview about the role and the motivations of the primitives and relations proposed in this work. This section turns this into a concrete DSL, the NPC4 meta model for hierarchical hypergraphs. A common approach in Model-Driven Engineer (MDE) [5], [9] is based on four different layers of abstractions, from M0 to M3, briefly resumed as follows. The M0 level refers to instances (“implementations”) of a DSL model. The M1 level comprises models that conform to [6] a meta model, defined on the M2 level. Therefore, the meta model on the M2 level specifies the
DSL in a formal way. Typically, each meta model can conform to several meta models at level M3. NPC4 resides on the M2 level, conforming to the (abstract and not yet formalized) M3-level concepts of hierarchical hypergraphs and structural system composition. It is obvious that multiple meta models on M2 can coexist and can be composed\(^2\) into new DSLs. In this cases, a new DSL conforms to one or more other meta models on the same M2 level; a concrete example will be illustrated in Section 5.

In the remainder, the textual formalization of the language is discussed, while Fig. 6 shows the corresponding graphical conventions used in the paper; the latter are introduced only for illustration, and should not be considered as “the” formalization introduced in this paper. Table 1 provides an overview of the language core (which is “the” formalization), and Table 2 illustrates the DSL by means of the concrete example of Fig. 7.

**Identity** is given to all primitives by simple declaration:

- **Node**: node-B, node-X, ...
- **Port**: port-p, port-x, ...
- **Connector**: connct-i, connct-j, ...
- **Container**: cntnr-m, ...

Furthermore, let \{node\}, \{port\}, \{connector\} and \{container\} be the sets of all the declared Nodes, Ports, Connectors and Containers, respectively.

**has-a**: a relationship between a Node and a Port. A Port can exist on itself (e.g., when it is still “floating” during the construction of a graph model in a development tool), but the graph model can only be “well-formed” (see Sec. 3) if every port belongs to exactly one node. Ports are those parts of a node through which (a selected subset of) the latter’s behaviour becomes accessible for interaction to other nodes. So only statements of the following type make sense:

\[
\text{has-a}\text{(node-B, port-p)}, \quad (5)
\]

and statements of the following type do not:

\[
\text{has-a}\text{(connct-i, port-p), has-a}\text{(cntnr-m, port-p)}. \quad (6)
\]

The inverse relationship \text{part-of} could be added to the model language, as syntactic sugar:\(^3\)

\[
\text{part-of}\text{(port-p, node-B)} \leftrightarrow \text{has-a}\text{(node-B, port-p)}. \quad (7)
\]

**contains**: Nodes and Containers can contain other primitives, as represented by containment statements of the following type:

\[
\text{contains}(M, X), \quad (8)
\]

with \(M\) and \(X\) being a Node or a Container. The \text{contains} relationship brings hierarchy in the relations between Node and Container primitives.

Containment is a transitive relationship, so other containment relationships can be derived from the statements above; for

---

\(^1\) In this work the following sentences are equivalent of expressing an has-a relationship: (i) "a port belongs to a node", (ii) "a port is attached to a node".

\(^2\) This composition of DSLs is sometimes denoted as language mixin.

---

The inverse relationship \text{part-of} could be added to the model language, as syntactic sugar:\(^3\)

\[
\text{part-of}\text{(port-p, node-B)} \leftrightarrow \text{has-a}\text{(node-B, port-p)}. \quad (6)
\]

**has-a**: a second relationship of this kind exists between a Port and a dock. The dock is a structural property of the Port that holds at most one connection with a Connector. Each Port has exactly two docks, one internal and one external with respect to the Node which owns the Port. The docks are true Port properties by design, therefore they are not considered as a primitive of the language. To distinguish with respect to the previous has-a relationship, the dock is uniquely referred by a \textit{dot} (.) notation, that is:

\[
\forall \text{port} \in \{\text{port}\}, \exists \text{edock}, \exists \text{idock} \quad (7)
\]

with edock and idock being a port’s external and internal dock, respectively. The dock property will turn out to be important later on, when well-formedness of connectors will be discussed in Sec. 3.

Fig. 6 shows the graphical convention of a Port, visualised as box divided in black and white rectangles; the former represents the internal dock, the latter is the external dock. The has-a relationship between Node and Port is visualised by placing the Port along the Node border.

---

3. Informally, in this work the following sentences are equivalent of expressing an has-a relationship: (i) "a port belongs to a node", (ii) "a port is attached to a node".

---

\(^2\) This composition of DSLs is sometimes denoted as language mixin.
Node: node—A, node—B, node—C, node—D, node—X, node—T
Port: port—q, port—r, port—p, port—n, port—u, port—s
Connector: connector—j, connector—i
Container: container—m

has—a(node—T, port—u)
has—a(node—X, port—s)
has—a(node—B, port—n)
has—a(node—B, port—p)
has—a(node—C, port—q)
has—a(node—D, port—r)

contains(node—A, node—B)
contains(node—A, node—C)
contains(node—A, node—D)
contains(node—T, node—A)
contains(node—T, node—X)
contains(container—m, node—A)
contains(container—m, connector—j)

connects(connector—j, port—q.edock)
connects(connector—j, port—n.edock)
connects(connector—j, port—r.edock)
connects(connector—i, port—p.edock)
connects(connector—i, port—s.edock)
connects(connector—i, port—u.idock)

Table 2: Full NPC4 model of the example shown in Fig. 7.

example:

contains(container—m, node—A),
contains(node—A, node—B)
⇒ contains(container—m, node—B).

connects: a Connects relationship binds two or more nodes together, via an hyperedge (i.e. a Connector) attached to (an internal or external dock on) Ports on these Nodes. So, statements of the following type are semantically valid:

connects(connct—i.port—s.edock),
connects(connct—i.port—u.idock).

2.5 Composition

An extra keyword is introduced to indicate that all primitives in NPC4 can be compositions in themselves:

composite = {node, port, connector, composite}.

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

Secondly, the composition with other, external DSLs is realised via the following fundamental design choice, motivated by the proven way that, for example, XML-based meta models such as XHTML, SVG or JSON use: each primitive in a model must have the following meta data “property tags”, that explicitly indicate in which knowledge context (that is, using which meta models) 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 human readability.

Such a 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 more of its possible composition relationships, namely contains and connects:

compose = {has—a, contains, 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

Figure 7: Generic example of a hierarchical hypergraph model. Node T is at the top of the hierarchy, and allows to refer to the whole model from within other models. Nodes A and X are contained by T, as is Container m; Nodes B, C and D are contained by A. Connectors i and j link Nodes on. All Ports have Connector docks internal and external to the Node they belong to. Container m gives a context to Node A and its internals, but not to Node X or Connector i.

4. This motivation comes from the objective to make the formal models useful not just to human system developers, at design time, but also to robots themselves, at run-time.
to be integrated. Trying to do that with one big modelling language becomes increasingly inflexible,\(^5\) 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.

### 3 NPC4 Language Constraints

The proposed NPC4 language not only introduces *primitives* and *relationships*, but also *constraints* to guarantee both syntactic and semantic correctness. In this section these constraints will be discussed.

#### 3.1 Constraints for Structural Well-Formedness

Some constraints must be satisfied by composition relationships in a graph model to make sure that the model is *well-formed*.

**There must be no “floating” ports:**

\(\forall p \in \{\text{port}\}, \exists N \in \{\text{node}\} : \text{has-a}(N, p).\)  \((11)\)

The reason is that ports get their semantic meaning only from giving access to the behaviour that is contained in the node they belong to, so: without a node, a port has no meaning.

**Contains relationships on Nodes must result in a containment tree.\(^6\)**

A *Node* can contain other *Nodes*, but it must not contain itself. Furthermore, each node has one and only one “direct parent node” in a containment relationship. The reason for this constraint is as follows: since nodes are meant to represent behaviour, and since the containment hierarchy is meant to allow levels of abstraction in a system model, it makes no sense if two nodes that are separated at a higher level of modelling would contain the same behaviour node at a more detailed model level. In other words, behaviour cannot be shared by two nodes with different identity.

---

As an example, Fig. 8 visualizes the node containment tree of Fig. 7. The *node* containment tree is unique for each *hierarchical hypergraph* and plays a relevant role on determining the validity of a *connects* relationship, as it will be discussed in the following paragraphs.

**Contains relationships of containers must result in a directed acyclic graph.**

That means that a container (or a node) can have multiple “parent containers”, and containers can overlap, but cannot contain themselves. This constraint is weaker than that for nodes, since containers are meant to represent knowledge, and knowledge can be shared indefinitely between nodes with different identities. An example is shown in Fig. 9, where Containers n and p overlap.

---

A *Connector* connects *Ports* on a joint containment tree.

The role of a *Port* is to provide a specific *view* on the *Node* that belongs to. In other words, the effect of the *Port* is to split the containment tree in two sub-trees, considering the *Node* as origin. The Port’s *internal dock* selects the “downward” subtree from that Node, while the external dock selects the “upward” subtree. Establishing a connection with a specific dock means to bound the relationship in the selected subtree, despite the other. For example, if a connector attaches to an internal dock of a port on a *Node*, all its other attachments must be to external port docks of *Nodes* that are contained in the given *Node*, or to other internal port docks of the same *Node*.

For the sake of clarity, Fig. 10 shows different model examples. The procedure to check this constraint is straightforward when starting from the *Node* containment tree: each of the

---

\(^5\)“Bad practice” experiences about relying on ever-growing modelling languages are unfortunately rather common in robotics: CORBA, UML, URDF…, are just some of the better known examples where the initial benefits of “standardization” become hindrance to flexibility in composition, as soon as a couple of dozen “nodes” must be integrated, in ways that were not realised before.

\(^6\)Strictly speaking, a forest, that is a collection of disjuncted trees.
ports involved in a connector prunes the Node containment tree in an downward and upward subtree depending on whether the Connector attaches to the Port’s internal or external dock, and the tree that remains after considering all involved Ports must still be connected.

The semantic meaning of this structural constraint is explained by observing an ill-formed example reported in Fig. 10. For instance, the Connector j in model 5 is semantically not correct, since it relates the node Node E with the whole Node D, but also with a Node D internal (Node A). Of course the Node E can have multiple kind of relationship with the D Node, but these are necessarily different relationships, as showed in the well-formed model 3. Different semantic meaning is represented by the Connectors (j,k) in models 2 and 3. In the former, Node E is in relationship with D, exposing a specific view on it (Nodes A and C). That is, the coupling E-A and E-C is indirect, since it considers explicitly the containment boundary D. In model 3, Connector j relates directly Node E with A and C, while Connector k is a completely unrelated relationship with respect to Connector j.

**Common Ancestor (LCA) of the Nodes involved.**

Considering the example in Fig. 7, a statement of the following type is semantically correct:

\[
\text{contains(node-A, connector-j)}, \quad (12)
\]

since node-A is LCA of Nodes A, B and C. This property is a consequence of a well-formed Connector, and it is not necessarily used to explicitly define a model. In fact, a Connector instance is already fully defined by a list of connects relationship that involves that Connector. However, adding this extra information in a NPC4 model can be useful as “checksum” during the validation phase. Finally, this Connector property helps the rendering of the hierarchical hypergraph layout.

As corollary, that implies that every graph model must have at least one root Node

\[
\forall C \in \{\text{connector}\}, \exists N \in \{\text{node}\} : \text{contains}(N,C).
\]

The reason is that everything inside that root Node must have an identified context.7

---

7. This context need not be unique, since others can be added by composition.
When describing the design decisions behind a formal modelling language, it is not only important to identify and motivate the constraints that compositions in the language must satisfy, but also why some constraints have not been introduced in the language. In this paper, the following “non-constraint” is one of the fundamental design choices: the contains and connects relationships are maximally decoupled, in that one does not depend on the other. For example, even though Nodes “X” and “B” live at two different levels of the containment hierarchy, the connector “i” can still connect both (through a port).

Fig. 7 and Fig. 9 show examples in which a Connector is crossing a containment boundary; even when the boundary is defined by a Node, the Connector can leave without the explicit need of a Port on that Node.

While the decoupling is maximal, it is not total: connectors must take the Node containment hierarchy into account to some extent, that is, as described by the last constraint above.

In summary, 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.

Another adopted “non constraint” design choice regards the direction over a connects relationship: no explicit direction is assumed, thus all the connections are un-directional. The direction is a property which belongs to the behavioural model, and not to the structural one: the constraint will be added in the specific domain of the meta model. A typical case is a FSM meta model, which will discussed in Sec. 5.

### 3.2 Constraints formalization

Section 2.3 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. This section describes some constraints already discussed in the previous section, but striving for formal completeness, by adding some obvious constraint relationships to the core semantics explained above.

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 has-a, contains and connects relationships.

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

\[ \forall (X,Y) \in \{ \text{node, port, connector, contains, connects} \}, \quad X.UID = Y.UID \Rightarrow X = Y. \]

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

**Constraints on has-a.** As mentioned in Sec. 2.4, a Port can be floating during construction time, but a model having a port that is not part-of a Node is an ill-formed model. Furthermore, a Port must be part-of one and only one Node, that is:

\[ \forall P \in \{ \text{port} \}, \forall (N1,N2) \in \{ \text{node} \}, \quad \text{has-a}(N1,P), \text{has-a}(N2,P) \Rightarrow N1 = N2. \]

The previous statements affects other relationships too, as it will be shown in the next paragraph.

**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. Recalling from Section 2.4, the Port has exactly two docks, one internal, and one external. Each such dock is constrained to have only one Connector attached, that is:

\[ \forall (C1,C2) \in \{ \text{connector} \}, \forall P \in \{ \text{port} \} : \]

- \( \text{connects}(C1,P.idock) \)
- \( \text{connects}(C2,P.idock) \)

\[ \Rightarrow C1 = C2 \]

\[ \forall (C1,C2) \in \{ \text{connector} \}, \forall P \in \{ \text{port} \} : \]

- \( \text{connects}(C1,P.edock) \)
- \( \text{connects}(C2,P.edock) \)

\[ \Rightarrow C1 = C2 \]

Furthermore, the well-formedness of the Connector (discussed in Sec. 3.1) can be formally expressed as follows:

- given \( C \) is the Connector to be validated;
- given the sets of internal and external Ports, \( p_{ci} \) and \( p_{ce} \), defined as:

\[ p_{ci} \triangleq \{ p \in \{ \text{port} \} | \text{connects}(C,p.idock) \} \]
\[ p_{ce} \triangleq \{ p \in \{ \text{port} \} | \text{connects}(C,p.edock) \} \]

- then, \( \forall p_i \in p_{ci}, N_i \in \{ \text{node} \} \) s.t. \( \text{has-a}(N_i,p_i) \) holds, \( \forall p_j \in \{ p_{ce} \} - p_i, C \) Connector is valid if \( \text{contains}(N_j,p_j) \) holds, and the following condition holds

- \( \forall p_e \in p_{ce}, N_i \in \{ \text{node} \} \) s.t. \( \text{has-a}(N_i,p_e) \) holds, \( \forall p_j \in \{ p_{ce} \} - p_e, C \) Connector is valid if \( \text{contains}(N_j,p_j) \) does not hold.

**Constraints on contains.** The constraints in this paragraph 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:

\[
\text{composite} = \{\text{node, port, connector, composite}\},
\]

\[
\forall C \in \{\text{composite}\}:
\exists n \in \{\text{node}\} \lor \exists c \in \{\text{connector}\} \lor \exists d \in \{\text{composite}\}:
\text{contains}(C, n) \lor \text{contains}(C, c) \lor \text{contains}(C, d).
\]

Every contains relationship can only be defined on existing Nodes and containers:

\[
\forall C \in \{\text{contains}\},
\exists (X, Y) \in \{\text{node, container}\}:
\text{contains}(C, X) \lor \text{contains}(C, Y).
\]

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

\[
\forall X \in \{\text{node, connector, composite}\},
\exists T \in \{\text{Node}\} : \text{contains}(T, X) \lor T = X.
\]

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: several context containers can be put around any existing model, and/or a composite can conform to more than one meta model. The top Node need not have any Port attached to it, so that it reduces to just a container of meta data.

4 MODELLING WITH NPC4

This section briefly discusses some structural features of the proposed solution.

4.1 Structure for supporting software

Many domain models use only traditional graphs, with Nodes and edges, while this paper’s hierarchical hypergraph model splits the “edge” primitive in two new first-class primitives: “port” and “connector”. The motivation for this choice is to allow not only more precise domain semantics if needed, but also a more flexible infrastructure to support a domain model with software. For example, by using ports to log and visualise data exchange between Nodes, or to count the number of interactions (statically as well as during run-time), or to make graphical development tools in which selections have to be made on which internal behaviour of Nodes to connect to, and so on.

Recall also the other motivation of this paper with respect to hierarchical composition: at a certain level of abstraction of a system model, a port might be a completely passive part of a system model, that is, without behaviour of its own, while more behaviour appears when going to a deeper level of abstraction in the system part represented by that port. 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, and so on. Much of that activity is “infrastructure” code for the higher level of abstraction, but this papers approach allows to connect all these things together, over different levels of abstraction.

The third software-centric motivation for the presented model pertains to the introduction of the container primitive: it carries no behaviour, but is used to model information influence of “higher” contexts on “lower” Nodes, ports and connectors. 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 (hence indicating, among other things, which configuration values to use for all model parameters); version numbers; etc. One particularly useful case is to introduce containers to store the composition model of the sub-system that is embedded within its internal context.

4.2 Behaviour on deeper levels of abstractions

In the proposed structural meta model, the hierarchy concept is applied to Node and Container primitives only. Allowing Ports and Connectors being hierarchies on themselves would violate the design choice that only Nodes carry behaviour.

However, in practical cases Ports and Connectors may manifest behaviour, if a deeper level of abstraction is considered. A concrete example arises in the attempt of model a software system involving 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 (i.e., a composition of Nodes, Connectors and Ports). In short, modelling the distribution explicitly boils down in introducing a deeper levels of abstraction.

In such cases, it is possible to apply a systematic model-to-model transformation to obtain an alternative model, as a composition of the original NPC4 model and a separate NPC4 model of the Port (or Connector) internals. Fig. 11 illustrates two examples which expands Port and Connector respectively. In both cases Port and Connector have been modelled as a simple Node, which already enables the

8. Or scope, namespace, domain, or whatever terminology has been used to represent this container concept.
hierarchy feature. Furthermore, a Container may be added to preserve the knowledge over the original model. As a remark, this property is offered by the composability feature of NPC4, considered as one of the primary design drivers of the language.

In conclusion, modeling a deeper level of abstraction is always possible, but the described structural models differs.

5 Examples

This section gives concrete examples on how the meta model language NPC4 can be used in existing and new meta models for robotic DSLs. These models must conform to the NPC4 meta model from the structural part. Thus, the following rules to design the DSLs are used, in accordance with the “composability” design driver (Sec. 2.3) behind NPC4:

• give new, domain specific names to the NPC4 primitives and/or relationships.
• add domain specific extra semantics to the NPC4 primitives, relationships and constraints.

The provided example DSLs focus on the NPC4-related structural part, rather than defining full fledged DSLs. As examples finite state machines and component architecture models are chosen, since both models are present in many robotics applications. The examples illustrate that both models use the hierarchical hypergraph as core compositional structure. Furthermore, the DSLs are exemplary designed to be internal DSLs embedded into JSON [19]. Thus, they have to conform to the syntax requirements of JSON. A JSON Schema file is used to check if the textual description of a model is syntactically correct or not. However, the methodology on how to make a DSL based on the NPC4 meta model does not depend on JSON or any other tool chain. Additional examples on how to adapt the the NPC4 methodology to other domains are briefly discussed in the Appendix.

5.1 Finite State Machines

FSMs are this paper’s primary example, because they have simple and familiar semantics, with a big part of it reflected in their structural model. There are many different FSM “dialects”, because, despite a rather large harmonization in the structural models, the behavioural parts of the various FSM DSLs do still differ. From a structural point of view, FSMs are defined as a set of states linked with transitions, with the constraint that each transition connects only two states.

This section discusses how an FSM DSL can be built with NPC4, and uses a so-called Life Cycle State Machine (“LCSM”) as a concrete illustration. Fig. 12 gives a graphical picture, Tables 3 and 5 give textual JSON [19] versions of the domain-centric model, without and with behaviour respectively, while Table 4 gives the NPC4-centric “full” version of it.

LCSMs are common software components to coordinate the “life cycle” of other software component instances, from when they created from their “platform resources” (memory, CPU, I/O), till they are ready to provide their “capabilities” to other components; the configuration of, both, resources and capabilities is a major “behaviour” of a LCSM. A real-world example of such a LCSM is the motion control of all the joint actuators in a robot: only when, both, the platform resources and the capabilities have been properly configured, the component is ready to “run”, that is, to actuate the robot’s motors based on commands from a control component.

The component may be paused from its running state, that is, it is fully ready to provide its service (immediately, without
any further configuration), but for one or another application-dependent reason, the service is not actually delivered, yet. In the above-mentioned example of motion control, the operator might have pressed a “motion freeze” button.

It is beyond the scope of this paper to model “the” correct LCSM, but it has just been introduced in this text to serve as a familiar example of a commonly occurring FSM model.

The whole FSMs structural part of a LCSM can be modeled straightforwardly by the above-mentioned approach as follows:

- states are represented by nodes in NPC4;
- states can be hierarchical, with a strict tree structure constraint in the contains relationship;
- transitions are represented by Connector relationships, with the extra constraints that the transitions are always (i) directed, and (ii) only connecting exactly two states. The latter constraint is fully structural, while the direction constraint belongs to the behavioural part of the DSL.
- transitions between hierarchical levels are allowed, so this structural property of FSMs requires no extra NPC4 constraint.

There is no explicitly visible concept of Port in FSM DSLs, but ports are needed nevertheless, in all software infrastructure with which an FSM model is stored and executed.

Fig. 12 gives, on the left, the traditional “domain-only” view on a LCSM, while the part on the right gives some examples of how ports are needed to attach the mentioned “infrastructure” structure and behaviour:

- Ports model the structural crossing of a transition across each level of hierarchical depth of states, Fig. 12. Such Ports are connected only to one connector in the same hierarchical level. Since the port belongs to a Node, the port can be attached to two connections, one internal and one external. For each hierarchical level crossed, a connector contained by the crossed hierarchical scope must be defined. The latter is due to the constraint that connects relationship is applied only between a Connector and a Port, and not between Ports directly;
- entry and exit functions of states are behavioural primitives of a state, which are “pointed to” from the ports where that behaviour is structurally located in the FSM, that is, there were the corresponding “incoming” and “outgoing” transitions connect with a state.

Since the FSM meta model conforms to [6] the hierarchical hypergraph meta model, the resulting concrete structure of the LCSM is described with NPC4 primitives (Fig. 12, on the right).

- states and the hierarchical relationship is directly preserved into a Node hierarchical structure (tree) (see example Table. 3);
- transition as port-connector-port pattern: a simple (not inter-level) transition is structurally equivalent of a composition of two connects relationship (see Tab. 4);
- entry and exit points as connection between one parent and some of its child nodes, making use of the same port-connector-port pattern.

Note that FSMs have different types of inter-level transitions, with possibly different behaviour semantics, while being indistinguishable from a structural point of view. For instance, the
transition indicated with tr2 in Fig. 12 (left) connects a state 
from an “deeper” level of containment to a state at a “higher”
level, while the transition tr3 does the opposite. Both have
have analogue structures, i.e., chains of the port-connector-
port pattern, defined as \{c4, c16, c10\} and \{c5, c19, c9\},
respectively. However, for tr3 the structure is fully defined 
by the transition itself, while tr2 only defines \{c4, c16\}: the
connector \{c10\} is given by the entry point defined in the
active state. The latter observation confirms a major invariant
design decision of this paper, that the structure of the transition 
is decoupled from its behavioural meaning in the particular 
domain meta model.

In summary, NPC4 provides a set of primitives to describe 
the graph representation of an FSM DSL, simply by adding 
domain-specific constraints. The structure provides the neces-
sary attachment points to host behavioural policies of various 
FSM dialects [20]. A similar procedure can be applied to other 
domains already mentioned in Section 1.

Table 4: On top, a snippet of a FSM model taken from the LCSM 
example (see Fig. 12), with JSON support. The model conforms to a 
FSM meta model, which it conforms to NPC4. On bottom, a NPC4 
code snippet which describes the structure of the FSM model above, 
with emphasis on the non-interlevel transition between the two states.

```
{  "states" : [  "active" , "creating" , "configuring_resources"  ],
    "contains" : [  { "parent" : "active" ,  "children" : [ "createing" , "configuring_resources" ] }  ],
    "transitions" : [  { "type" : "transition" , "id" : "tr1" ,  "src" : "creating" ,  "tgt" : "configuring_resources" }  ]
}
```

Node: active, creating, configuring_resources
Port: p7, p8
Connector: c3
contains(active, creating)
contains(active, configuring_resources)
has-a(creating, p7)
has-a(inactive, p2)
connects(c3, p7.edock)
connects(c3, p8.edock)

Table 5: An extended version of the FSM model snippet in Table 3. 
Below, its relative structure described with NPC4. The emphasis 
is on the definition of the entry point of the composite state and 
the reflected changes on the graph structure. Changes has been 
highlighted. The full visual representation is shown in Fig. 12.

```
{  "states" : [  "lcsm" , "active" , "inactive" ] ,
    "contains" : [  { "parent" : "lcsm" ,  "children" : [ "lcsm", "active" ] }  ],
    "entry" : "inactive" }
```

Node: lcsm, active, inactive
Port: p1, p2
Connector: c1
contains(lcsm, active)
contains(lcsm, inactive)
has-a(lcsm, p1)
has-a(inactive, p2)
connects(c1, p1.idock)
connects(c1, p2.edock)

5.2 Component architecture models

Several benefits are possible from the adoption of a hierar-
chical hypergraph structure to describe a component-based 
software architecture. In the robotics domain, there is a large 
number of functionalities that must be integrated in an overall 
architecture, often deployed into reconfigurable components 
connected each others. Several middlewares offer such a 
capability, among which ROS, Orocos, YARP, CLARAty [21], 
[22], [23], [24]. These frameworks offer complementary (and 
sometime alternative) features regarding different aspects of 
the robotic system, such as real-time components, commu-
nication services, run-time configuration, deployment system 
and others. It is a common practice to build a software 
infrastructure choosing not one, but several middlewares to 
better adapt to specific needs. For instance, Orocos enables 
real-time activity containers, while ROS is more popular 
among the robotics community when different functionalities 
are to be deployed in multiple machines. Therefore, a robotic 
application is often a heterogeneous integrated system.

To show the potential of the proposed NPC4 language, this 
section considers a concrete software application. The chosen 
use-case pertains to an application where a fleet of mobile 
platforms, equipped with camera, are exploring an unknown 
environment automatically, storing the collected information 
in a centralised database. The functionalities required for such 
applications are developed in a component-based fashion using 
ROS and Orocos middleware; some components are deployed 
on a centralised server, others are deployed locally with respect 
to the mobile agents. Among the latter, some components have
real-time requirements, such as motion control functionalities. Thus, Orocos middleware is preferred, otherwise ROS is chosen. The architectural structure expressed in NPC4 is shown in Fig. 13. Discussion of the “optimality” of the given architecture is beyond the scope of the example; the focus is on presenting the mapping between structural primitives and behaviour (or functionalities), reported in Table 6.

The role of the Containers is fundamental and non-traditional in this context, since it allows to attach additional information, such as grouping components in accordance with the hardware that is hosting the functionality, or the grouping of components that have to be configured together on the basis of the same system knowledge. Therefore, a first benefit given by a structural description is to provide different context-dependent views of the same architecture, concerning both off-line and run-time information. This enables, for example, the usage of common tools to visualize the architecture graphically. Tools provided in the middleware are not sufficient: for instance, ROS rqt_graph can only represent ROS-nodes, but not components from other frameworks, mainly due to the lack of a formal model description.

A second benefit is given by the possibility to solve queries over the software infrastructure, decoupled from the implementation of the middleware adopted. Relevant queries are the ones that validate the correctness of the model, for instance to verify a proper component connectivity. The previous can be done by checking the Connector IDs, which represent a specific data-stream (e.g., ROS-topics) and the extra directivity constraint given by the Ports connected to it. Only the extra constraint is necessary to accomplish a complete query that crosses the boundaries of the frameworks. In short, it is possible to build test tools on the NPC4 structural model, avoiding solutions based on a single framework functionalities. Obviously, a necessary requirement is the existence of the integration functionality between the middleware, which must be modeled as well. In fact, the software integration often regards only the functionalities; the components are hidden behind an interface layer. Separating and modeling the structural parts avoids information hiding that regards only the functionalities. Furthermore, it is possible to link and chain these queries to the existing tools. As an example, information on the nodes and connectors contained in a ROS container can be retrieved online through the commands rostopic and rosnod e.

A third advantage regards the code generation required for the deployment of the component into activity containers (OS processes or threads) provided by the different middleware. This is a popular issue tackled by the community so far, and toolchains to support it already exist. [25], [26], [27]. An approach based on NPC4 is the following: from a structural NPC4 model and additional knowledge associated to it, the
Table 6: NPC4 primitives applied to ROS and Orocos functionalities, applied in the example of Fig. 13. A structural primitive in NPC4 offers an attachment for multiple functionalities or behaviours, that must be interpreted accordingly the extra constraints imposed by different framework domains. For instance, in both middleware a port item has directivity as extra constraint related to the behaviour that represent (e.g., ROS: pub/sub, Orocos: input/output ports). As a remark, the proposed mapping describes up to a certain level of abstraction. For instance, Ports can be further expanded to represent a buffer policy on the incoming data.

### 6 RELATED WORK

Support for hierarchical hypergraphs, including ports, as first-class citizens in the model is a rare exception. Among the examples discussed in Section 1, only FSMs, Factor Graphs and HTML5 have them in their models, and then only 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 often not aware of the extent to which their modelling languages and tools restrict their flexibility in modelling the designs of their systems.

An intrinsic limitation is present in control diagrams: the directed edges in, for example, Simulink [2] 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 [28], that do not oblige their users to use only uni-directional interactions, since they are based on the so-called Bond Graph-based modelling primitives [4], [29], [30], [31], that allow to represent the physical un-directional (“non-causal”) 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

9. They do support “is-a” behavioural hierarchy.
and the behaviour (or functionality). In literature, several works tackled the deployment of complex and heterogeneous system architectures. The ROCK toolchain [25] is a practical effort to bridge MDE techniques and Orocos functionalities. A similar effort has been done in [26], having OpenRTM-aist [37] as backend middleware. The RobotML [27] DSL extends these principles, providing support to multiple middlewares in its toolchain. These solutions are based on existing modeling frameworks, such as the Eclipse Modeling Project [43]. However, the presented work differs from the mainstream approaches: instead of providing a solution that binds to a specific M3 tool, a mathematical generalisation is provided such that the same structural model can be used for different purposes. Concretely, the Component-Port-Connector (M2 model) of the BRICS Component Model [44] conforms to the NPC4 meta model. However, the versatility of NPC4 has been shown in Section 5: the structure of a component-based model (Section 5.2) and a FSM model (Section 5.1) are the same. Such a structural equality can be exploited, because of the existence of common operations; for example, in the rFSM [14] implementation [45], over the 60% of the code pertains to managing nodes and connections (states and transitions), while only the remaining 40% implements the concrete behavioural engine.

This work also contributes to language-oriented software development [46], [47] by means of composition of multiple DSLs. In detail, [48] provides a terminology to distinguish between different forms of language composition, and the derivation of a new DSL that structurally conforms to NPC4 can be described by using that terminology (Section 5.1); firstly the NPC4 primitives and relationships are specialized with additional constraints, and this step is called language restriction or specialization; secondly, a behavioral model is attached accordingly, leading to a language unification. Therefore, the NPC4 promotes a development process based on extension composition, that is the specialization of multiple DSLs that work together, each one addressing a different aspect of the system (structural, behavioural, functional, etc.).

Also the different design phases of a DSL development (decision, analysis, design, implementation, deployment) are
discussed in literature; NPC4 provides the structural primitives that are required to support the development of DSLs [16].

In summary, separating and standardizing the hierarchical hypergraph structure from the attached behaviour has a promising impact on the development of systems in the robotics domain.

7 CONCLUSIONS

What is the minimal set of primitives and relationships, to cover all use cases of structural composition in robotics applications?

This was the main research question that the authors tackled for almost a decade, motivated by the drive to realise a step change in the reuse of “infrastructure code”. Several robotic frameworks have been developed in the last years, and all of them have quite overlapping needs with respect to the structural composition of the functional primitives they offer. Yet no common designs or models are shared, let alone code.

This paper advocates the use of the NPC4 language, as the meta model to represent port-based and container-based composition, for both interconnection of behaviour and containment of knowledge, and in a domain-independent way.

The minimal set of primitives adopted are commonly used elements: Nodes, Ports and Connectors (or semantically equivalent concepts) have been used in several contexts, in one form or another. The real challenge was to identify the minimal set of constraints that govern all structural compositions: the lesson learned is that developers tend to be not very aware of such constraints, and the more expert one is in a certain domain, the more obvious and implicit such constraints appear.

The objectives behind 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 potential benefits of having a common structure are manifold, such as common tools for storing, querying and composing heterogeneous systems, as well as easily create new functionalities or DSLs based on the graph structure. Those benefits related to reuse of both modelling concepts and software is discussed by means of two examples; however, only further adoption of the proposed NPC4 model can validate the research hypotheses, and this adoption depends strongly on the further development of tools that compose the NPC4 with other DSLs, exploiting the common structure description.

The behaviour attached to the structural primitives 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 run-time.

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.

Although presented in a robotics context, nothing in NPC4 depends on this specific robotics domain. NPC4 can also serve the goals of related to other application domains such as 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.

APPENDIX

This section gives further domain specific explanations for each of the graph structures that where listed in Section 1.

- **software architectures**, as in Fig. 1. Typically, each Node represents an input-output relationship that has dynamic and time-varying behaviour, 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 [2] or Modelica [49]), at least in the modelling part of system design.
- **kinematics and dynamics of actuated mechanical structures**, as in Fig. 2. The joint nodes contain actuator dynamics, and the link nodes contain rigid-body inertia dynamics; the edges represent structural connectivity, modelling which actuators and links are exchanging energy, that is, exhibit behaviour. Hierarchy is possible, e.g., a spherical joint can mechanically be realised by a parallel mechanism.
- **Finite State Machines**, as in Fig. 3, model the discrete 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 activities. The structure of these switches is modelled by the states being connected via so-called “transitions”. Structural hierarchy abstracts away how the system reacts to a set of events.
• probabilistic graphical models such as Factor Graphs, Fig. 4. Nodes represent time-varying (“behavioural”) information as captured in “random variables”; edges represent (“structural”) probabilistic relationships which govern the interaction between the random variables in the connected nodes. Ports are typically not represented, such that the graphical model does not allow to indicate which of the random variables in each node are involved in each of the relationships represented by edges.

• control diagrams and other “data flow” computational models, such as the control scheme of Fig. 5; popular instances are Simulink [2] diagrams, or Bond Graph [4], [50], [29], [30], [31] models in 20Sim [28]. The separation of structure and behaviour is similar to the above-mentioned cases of software and kinematic models: nodes represent “dynamics”, edges represent exchange of information or energy.

• knowledge representation networks, such as the “semantic web” (represented often by the RDF or OWL) or the robotics KnowRob [51]. 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. Surprisingly, none of the mainstream approaches support hierarchy as a top-level modelling primitive, although it is needed to give structure to the concept of various “levels of abstraction” in a knowledge representation of a system.

REFERENCES


[28] ControlLab Products B.V., “20sim-,” http://www.20sim.com/, accessed online 2 August 2013. 6, A


Languages to realize advanced robotic applications.


Enea Scioni received his B.Sc. and M.Sc degrees in Computer Science and Automation Control from University of Ferrara in 2007 and 2010, respectively. Since 2011, he is a PhD candidate at both University of Ferrara and KU Leuven. His current research interests are on formal specification and scheduling of constraint-based tasks, optimization-based robot control and coordination of complex robotic systems. His research also concerns on developing software middleware tools and Domain Specific Languages to realize advanced robotic applications.

Nico Hübel received his Dipl.-Ing. (M.Sc.) in Engineering Cybernetics from the University of Stuttgart, Germany, in 2010. From 2010 to 2014 he was research assistant at ETH Zurich. Since 2014 he is a Research Scientist in the Robotics Research Group of KU Leuven. He was a research scholar at Tokyo Institute of Technology and a member of R&D at KUKA Robotics. His research interests are in the area of autonomous robotics, learning, control systems theory, and software engineering for these areas.

Sebastian Blumenthal received his B.Sc. and M.Sc degrees in computer science from the University Bonn-Rhein-Sieg in 2007 and 2009. In 2013 he started his Ph.D. degree at the KU Leuven, focusing on the software engineering aspects of world modelling in large-scale robot control systems.

Azamat Shakimardonov received his B.Sc degree in control engineering from the Taskent State Technical University, in 2004, and his M.Sc degree in computer science from the University Bonn-Rhein-Sieg in 2007. Since then he has worked in many EU robotics projects as a technical engineer, and in 2015 he obtained his Ph.D degree in mechanical engineering from KU Leuven. His current research interests are robot motion control, robot dynamics, constraint based task specification and software engineering of large-scale robot control systems.

Markus Klotzbücher received his PhD from the KU Leuven, on Domain Specific Languages for Hard Real-Time Safe Coordination of Robot and Machine Tool Systems in 2013, and is now lead engineer embedded software at Kistler Instrumente AG in Winterthur.

Hugo Garcia is research engineer at KU Leuven, and the author of the BRIDE software tool for component-based robotics software.

Enea Scioni received his B.Sc. and M.Sc degrees in Computer Science and Automation Control from University of Ferrara in 2007 and 2010, respectively. Since 2011, he is a PhD candidate at both University of Ferrara and KU Leuven. His current research interests are on formal specification and scheduling of constraint-based tasks, optimization-based robot control and coordination of complex robotic systems. His research also concerns on developing software middleware tools and Domain Specific Languages to realize advanced robotic applications.

Nico Hübel received his Dipl.-Ing. (M.Sc.) in Engineering Cybernetics from the University of Stuttgart, Germany, in 2010. From 2010 to 2014 he was research assistant at ETH Zurich. Since 2014 he is a Research Scientist in the Robotics Research Group of KU Leuven. He was a research scholar at Tokyo Institute of Technology and a member of R&D at KUKA Robotics. His research interests are in the area of autonomous robotics, learning, control systems theory, and software engineering for these areas.

Sebastian Blumenthal received his B.Sc. and M.Sc degrees in computer science from the University Bonn-Rhein-Sieg in 2007 and 2009. In 2013 he started his Ph.D. degree at the KU Leuven, focusing on the software engineering aspects of world modelling in large-scale robot control systems.

Azamat Shakimardonov received his B.Sc degree in control engineering from the Taskent State Technical University, in 2004, and his M.Sc degree in computer science from the University Bonn-Rhein-Sieg in 2007. Since then he has worked in many EU robotics projects as a technical engineer, and in 2015 he obtained his Ph.D degree in mechanical engineering from KU Leuven. His current research interests are robot motion control, robot dynamics, constraint based task specification and software engineering of large-scale robot control systems.

Markus Klotzbücher received his PhD from the KU Leuven, on Domain Specific Languages for Hard Real-Time Safe Coordination of Robot and Machine Tool Systems in 2013, and is now lead engineer embedded software at Kistler Instrumente AG in Winterthur.

Hugo Garcia is research engineer at KU Leuven, and the author of the BRIDE software tool for component-based robotics software.

Enea Scioni received his B.Sc. and M.Sc degrees in Computer Science and Automation Control from University of Ferrara in 2007 and 2010, respectively. Since 2011, he is a PhD candidate at both University of Ferrara and KU Leuven. His current research interests are on formal specification and scheduling of constraint-based tasks, optimization-based robot control and coordination of complex robotic systems. His research also concerns on developing software middleware tools and Domain Specific Languages to realize advanced robotic applications.

Nico Hübel received his Dipl.-Ing. (M.Sc.) in Engineering Cybernetics from the University of Stuttgart, Germany, in 2010. From 2010 to 2014 he was research assistant at ETH Zurich. Since 2014 he is a Research Scientist in the Robotics Research Group of KU Leuven. He was a research scholar at Tokyo Institute of Technology and a member of R&D at KUKA Robotics. His research interests are in the area of autonomous robotics, learning, control systems theory, and software engineering for these areas.

Sebastian Blumenthal received his B.Sc. and M.Sc degrees in computer science from the University Bonn-Rhein-Sieg in 2007 and 2009. In 2013 he started his Ph.D. degree at the KU Leuven, focusing on the software engineering aspects of world modelling in large-scale robot control systems.

Azamat Shakimardonov received his B.Sc degree in control engineering from the Taskent State Technical University, in 2004, and his M.Sc degree in computer science from the University Bonn-Rhein-Sieg in 2007. Since then he has worked in many EU robotics projects as a technical engineer, and in 2015 he obtained his Ph.D degree in mechanical engineering from KU Leuven. His current research interests are robot motion control, robot dynamics, constraint based task specification and software engineering of large-scale robot control systems.

Markus Klotzbücher received his PhD from the KU Leuven, on Domain Specific Languages for Hard Real-Time Safe Coordination of Robot and Machine Tool Systems in 2013, and is now lead engineer embedded software at Kistler Instrumente AG in Winterthur.

Hugo Garcia is research engineer at KU Leuven, and the author of the BRIDE software tool for component-based robotics software.

Enea Scioni received his B.Sc. and M.Sc degrees in Computer Science and Automation Control from University of Ferrara in 2007 and 2010, respectively. Since 2011, he is a PhD candidate at both University of Ferrara and KU Leuven. His current research interests are on formal specification and scheduling of constraint-based tasks, optimization-based robot control and coordination of complex robotic systems. His research also concerns on developing software middleware tools and Domain Specific Languages to realize advanced robotic applications.

Nico Hübel received his Dipl.-Ing. (M.Sc.) in Engineering Cybernetics from the University of Stuttgart, Germany, in 2010. From 2010 to 2014 he was research assistant at ETH Zurich. Since 2014 he is a Research Scientist in the Robotics Research Group of KU Leuven. He was a research scholar at Tokyo Institute of Technology and a member of R&D at KUKA Robotics. His research interests are in the area of autonomous robotics, learning, control systems theory, and software engineering for these areas.

Sebastian Blumenthal received his B.Sc. and M.Sc degrees in computer science from the University Bonn-Rhein-Sieg in 2007 and 2009. In 2013 he started his Ph.D. degree at the KU Leuven, focusing on the software engineering aspects of world modelling in large-scale robot control systems.

Azamat Shakimardonov received his B.Sc degree in control engineering from the Taskent State Technical University, in 2004, and his M.Sc degree in computer science from the University Bonn-Rhein-Sieg in 2007. Since then he has worked in many EU robotics projects as a technical engineer, and in 2015 he obtained his Ph.D degree in mechanical engineering from KU Leuven. His current research interests are robot motion control, robot dynamics, constraint based task specification and software engineering of large-scale robot control systems.

Markus Klotzbücher received his PhD from the KU Leuven, on Domain Specific Languages for Hard Real-Time Safe Coordination of Robot and Machine Tool Systems in 2013, and is now lead engineer embedded software at Kistler Instrumente AG in Winterthur.

Hugo Garcia is research engineer at KU Leuven, and the author of the BRIDE software tool for component-based robotics software.