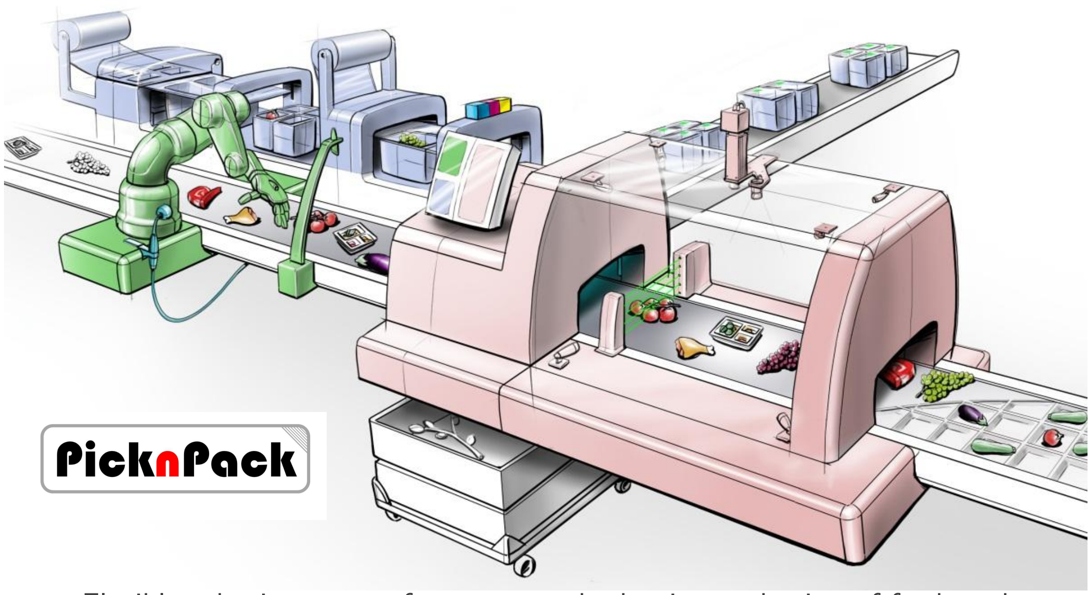
D2.2 — Report on adaptive GUI

Herman Bruyninckx, Nico Hübel, Johan Philips

11 October 2015



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

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 introduces a new approach for developing an adaptive GUI for complex, flexible systems. These days modern systems like the PicknPack line consist of multiple sub-systems that often come from different vendors, run different operating systems, and are programmed in different programming languages. Traditionally, each of the vendors would deliver the sub-system with its own GUI, but usually the interface provided for introspection from outside the sub-system, at the system level, is limited. This hampers flexible solutions that can easily be integrated and adapted to changing and increasingly complex requirements. In Picknpack these information boundaries are opened up to enable easier and more flexible integration at the system level. This is achieved by introducing a common communication model and message or data models that all sub-systems conform to. The details about the used communication infrastructure and protocols are described in 4.1 and the message or data models are described in 3. As long as each sub-systems conforms to this communication model, their implementation details, like the used programming language, are decoupled and the overall integration can be done by defining and using the message/data models. The same is true for the GUIs. Since the information boundary is removed, one GUI at system level can be used to introspect and re-configure any sub-system. While this is sufficient for fully automated factories with centralized control centres, many factories have still workers that need to interact with sub-systems directly. Instead of falling back to sub-system specific GUI implementations, we propose a flexible, configurable GUI that can run on any sub-system as well as at the system level. Such a GUI would need to be

cross-plattform: Each sub-systems can run a different operating systems. The GUI will need to run on all of them.

expandable: Each sub-system can add different data and information that needs to be rendered.

customizable: Each sub-system can require different visualizations of data and information.

We solve these issues by leveraging the following technologies:

Browser based technologies: Modern web browsers are available for practically all operating systems and offer all infrastructure necessary for all kinds of visualizations. Therefore, they satisfy the requirement for cross-platform implementation.

Data models: Data models allow to decouple the visualization infrastructure from the data that can be visualized. This allows for flexibility in the GUI. Extensibility can be achieved by writing a so called widgets for each new data type. This data type comes with a data model and extends the capabilities of the GUI to display data conforming to this data model. At the same time these widgets solve the requirement of customizability by enabling sub-system developers to create their sub-system GUI by loading the required widgets.

In addition, the web community is very active in developing and improving libraries and sometimes even (quasi-)standards that can be leveraged. Most noteworthy are GeoJSON[1] and TopoJSON[2] that come with data specifications (or models) and visualization implementations that can be leveraged to write plugins for the GUI.

It is also important to note that this architecture is designed to support hierarchy, which is important to handle complexity. However, all levels in the hierarchy are treated with the same methodology. In the Picknpack case there are two relevant levels of hierarchy:

[1] http://geojson.org/geojson-spec.html
[2] https://github.com/mbostock/topojson/wiki

1. The module level deals with the functionality of an individual module and handles the integration of the devices (sub-systems). At this level the main use case for a GUI is to show information and data of the module and its devices and to allow interactions with the devices. Such interactions could be the mere visualization of collected data, or the displaying of camera images, but can also entail altering device settings.

2. The line level deals with the integration of the modules to realise the functionality of the production line. At this level the main use case for the GUI is to aggregate and visualize information about the whole line. However, since the information of the modules (sub-systems at this level) is accessible, the GUI can be used to display the same information as the module level GUI.

This report is structured as follows. In section 2 we will explain the mentioned concepts in more details, discuss the underlying architecture, and outline what can be done with these new concepts. Afterwards, we show and explain the used models in section 3 before we discuss the implementation in section 4. Finally, we summarize the results in section 5.2.

2 Concepts

2.1 Explanation of Used Terminology

Some terms used to describe the GUI are explained in the following and the visible elements are shown in Fig. 1:

- *GUI*: The Graphical User Interface presented in this report. It makes use of a *browser* as cross-platform infrastructure and uses *widgets* for visualizing data conforming to defined *data models*.

- *Browser*: The browser is like any web browser commonly used, that is, mainly Firefox and Chrome. They run their own, embedded, javascript interpretor, which provides the ability to define widgets from which cross platform GUIs can be composed.

- *Composer*: The composer is a software module of the GUI that is in charge of deploying *plugins*, *widgets*, and *mediators* and configuring all of them properly.

- *Widget*: A widget is a piece of javascript code that enables the GUI to visualize a certain type of data. The widget registers these visualization capabilities with the *mediator* by informing it of the *data models* its data has to conform to.

- *Mediator*: The mediator is responsible for the communication between all entities inside the GUI and outside the GUI. To be able to mediate between these two systems, it has to know about the capabilities and requirements of both sides of the system boundary. It is thus **the key place for system integration and one of the key elements in the proposed architecture**. The mediator can be composed of several mediators that have different responsibilities (e.g. handling the event stream or setting up a dedicated peer-to-peer session; see communication in section 4.1).

- *Plugin*: A plugin corresponds to an event loop in the browser. Multiple widgets are usually combined into one plugin. The browser supports already built-in plugins in conformance to the HTML5[3] standard.

[3]http://www.w3.org/TR/html5/

- *Data model*: A data model is a formal description that a certain data type conforms to. These models can be used to communicate interfaces or capabilities, verify the structure of incoming data, and are easier to change than their corresponding entities in the source code (if tools are available for interpreting the models instead of hard coding corresponding data structures). Thus, they are a key element for flexible, reusable software that is easy to reconfigure. The models used in PicknPack are discussed in section 3.

2.2 GUI Architecture

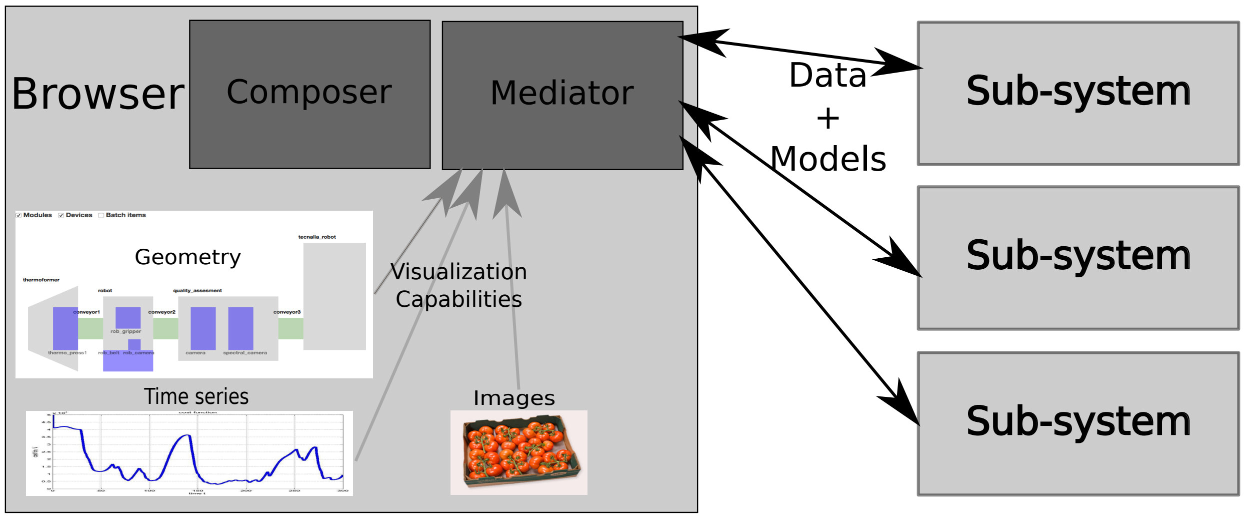


Figure 1: This shows an example for a system (or line) level GUI. The GUI runs in a *browser* and the *composer* has deployed three *widgets*, one showing the geometry of the line, one showing the time series of a value, and one showing an image taken in one of the modules. The mediator knows about the visualization capabilities of the GUI and is responsible for the communication with other entities outside the GUI and makes new data and information available to the responsible widget.

2.2.1 Deployment

When the GUI is loaded in the browser,the composer is launched and receives information about the composition of the GUI, that is, which plugins and widgets to load and how the layout of the window looks like. The plugins register themselves in the browser and run then within its infrastructure. The widgets run within their plugins. In addition, each widget enables the GUI to visualize one or more types of data and it informs the mediator about the data model(s) it can visualize. The mediator registers these visualization capabilities and which widget is responsible for which data model and also sets up the communication with other systems outside of the GUI. How this communication is setup depends on the application. The PicknPack specific communication is briefly explained in section 4.1 and useful communication patterns can be found in appendix B.

2.2.2 Event Based Interaction within the GUI

In principle the whole GUI is based on event streams. This helps decoupling the sub-systems like widgets, plugins, and the mediator. So whenever something happens that needs to be processed or visualized by

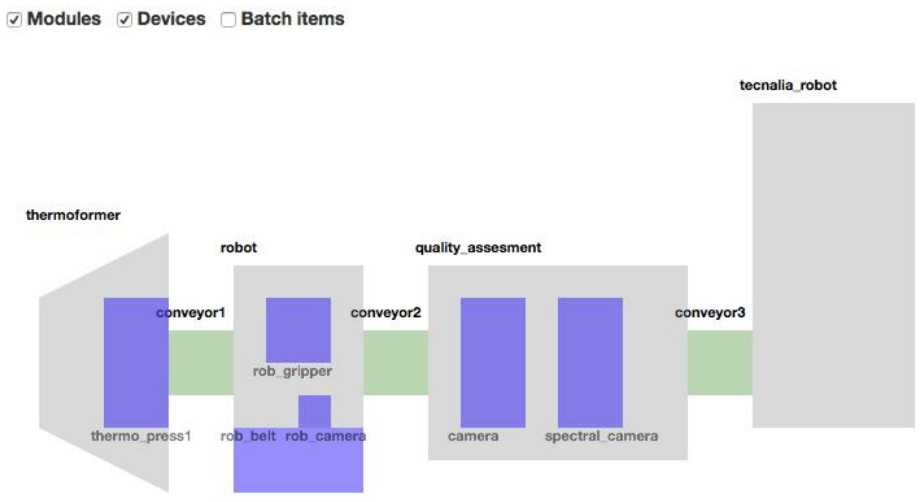


Figure 2: This shows a widget displaying an example geometry of a line with four modules and several devices. It also allows to select what is visualized by using the checkboxes at the top.

another sub-system, an event is raised that then can be processed by that sub-system. For example, if a user clicks on the a module in the geometry widget shown in Fig. 2, this widget can raise an event that is then processed by a widget for displaying data (like configurations of modules or devices). Then this widget can then request the necessary data and visualize the module's configuration.

This is increasing the flexibility in the sense that both widgets can still be used independently, but when used together exhibit additional behaviour. If only the geometry widget is available, it will still raise the event, but since there is no other widget to process the event, it can be dropped. And on the other hand, if only the data widget is available, it will never receive an event to display a module's configuration data from the geometry widget, but can still be triggered from other widgets or the mediator.

Neither of these two cases is creating a problem for the functionality of the system. On the contrary, the architecture supports these kinds of flexibility. However, if one wants to optimize performance, there are existing libraries that help preventing such cases and they can be prevented by explicitly modelling the dependency in the widget's model and checking for them at deploy time.

2.2.3 Mediator: Information Flow across the System Boundary

All data flow that crosses the system boundary is going through the mediator. The mediator knows about the inside of the GUI (visualization capabilities) but also about the outside of the GUI (communication infrastructure and protocols, data and message models, authentication processes, and query end points). The information flow can be unidirectional or bidirectional and it can be initiated from within the GUI or from another system outside the GUI. These four cases are discussed in the following:

- *Unidirectional, GUI initiated*: One of the widgets wants to push information to a connected system. E.g. a data widget showing a module's configuration wants to push changed configuration parameters to that module. For doing so, the widget raises an event that is processed by the mediator. This event contains information on the addressee (e.g. the module) and the data (either directly or a link to where the data can be fetched). The mediator packages this information into a message that conforms to an appropriate message model and puts this message on the communication channel.

- *Unidirectional, externally initiated*: An external system wants the GUI to visualize some data, e.g. a sensor sends new measurements that need to be displayed in a time series widget. This data (directly or in form of a link from which this data can be fetched) is sent to the mediator. It also contains metainformation like to which data model(s) the data conforms to. The mediator uses this metainformation to check if this data can be visualized and by which widget(s). It then makes the data available to these widgets and creates appropriate events that trigger the visualization. If the GUI cannot display data conforming to the data model provided in the metainformation, the mediator can be configured to notifies the sender or simply ignore the data.

- *Bidirectional, GUI initiated*: One of the widgets is triggered by an event but requires data for visualizing, e.g. in the example above, when someone clicks on a module in the geometry widget, triggering the data widget to display the module's configuration. In this case, the widget raises an event that is processed by the mediator. This event contains the addressee (the module) and the requested data model (configuration data). The mediator then takes care of providing this information to the module by either checking its local cache or querying an external system (e.g. the module itself or a world model providing a querying interface). Once the data is available it raises an event for the requesting widget that then can visualize the data.

- *Bidirectional, externally initiated*: An external system requests the GUI to visualize information and requires feedback, e.g. and error messages that needs to be acknowledged or requires action by a user. This information is sent to the mediator. It also contains metainformation like to which data model(s) the data conforms to. The mediator uses this metainformation to check if this data can be visualized and by which widget(s). It then makes the data available to these widgets and creates appropriate events that trigger the visualization. These widgets have to be designed to give a feedback event to the mediator containing the information that it needs to feedback to the external system. E.g. an error could be visualized by a pop-up window that needs acknowledgement or provides a selection of actions. The acknowledgement or selected action is then passed to the mediator, which relays it to the external system. If the GUI cannot display data conforming to the data model provided in the metainformation, the mediator notifies the sender that then has to handle this.

2.3 Unique Identifiers

Unique identifiers (*"UIDs"*) are essential for traceability, but also for keeping track of all entities within the scope of PicknPack. Entities here mean not just physical entities described in the first part of section 2.1 but also every piece of data and information in the PicknPack system. So PicknPack has to come up with a methodological way of assigning and interpreting such UIDs. Instead of reinventing the wheel badly, the suggestion is to use existing standards like

- *GTIN*:[4] the *Global Trade Item Number*, which covers all industries and economical activities, and

- *UUID*:[5] the *Universally Unique IDentifier*, which is widely adopted in distributed software systems.

The GTINs are used within the tracing database to keep track of the food before and after the line. The UUIDs will be used to identify all entities (hardware, software, data, information) within the line. In practice, this poses no problem since all information is linked using these unique IDs, so theoretically any number of UID standards could be used.

[4]http://www.gs1.org/barcodes/technical/idkeys/gtin
[5]http://tools.ietf.org/html/rfc4122

However, to increase the flexibility further in the direction of Plug-n-Play hardware and software, a USB like standard would be required. In order to create a USB like standardised device descriptor, a standardization committee would be necessary that *enumerates* the different *types* of food production lines, modules and devices, and assigns a unique name and ID to them. Each "vendor" than adds its own IDs to identify the specific *instance* of the generic type that has been used for a specific instance of food production. The latter ID is nothing else than the commonly known *"serial number"* that is already in common use.

3 Models

A PicknPack system requires more than a standardized way to design its system architecture, since it produces, consumes and exchanges a lot of data, internally and with other systems in the world. The only way then to realise *flexibility* and *traceability* is to define *standards* that different vendors can interpret unambiguously, and for which "conformance tests" can be designed. The design of such standards is a continuous process, and as a process, can also profit from some "standardization" itself.

In most engineering systems, standards are only being used by the *human* developers who implement a system or subsystems. But because of the high demands for *flexibility* and *traceability* in the PicknPack project, it will also be necessary that software components understand (some of) the standards and can turn them into activities that respect the standards. In software related and other engineering systems, such standards are also called models.

For the GUI the use of such models was already mentioned in the previous section. In order for the widgets to register their visualization capabilities with the mediator, they need to communicate models. And for the mediator such models are important to filter incoming data and map it to the responsible widgets. Mere data types are not sufficient since they can be ambiguous. It starts with the fact that "basic" data types like strings and floating point numbers have different representations and becomes even more confusing for complex data types. Therefore, we explicitly let every piece of data contain a link to its model, which makes the data interpretable for humans and software systems alike.

A well-designed standard does more than formally represent a set of agreements and relationships, but it can also be used to generate visualizations from these formal descriptions. Current activities in WP2 are on creating widgets for visualizing the models that are introduced in the sections following the introduction of the chosen modelling language JSON.

3.1 JSON as the default modelling language

The first realisation in this context is the thorough search for already existing standards that could be adopted, especially if they come with an active and rich ecosystem of software implementations and tools. After an intensive evaluation process, *JSON*[6] has been chosen as the default modelling language, and more precisely, the semantically richer "linked data" version *JSON-LD*[7] is the eventual objective of this work in the project. The motivations for this suggestion are:

- *ecosystem*: since JSON has been adopted as *the* data model for all web applications, a very strong development activity has been set up, resulting in many (open source) tools and reference implementations that PicknPack can profit from.

- *UIDs*: it is common practice in JSON modelling to foresee UID tags for all fields in a model, which conforms nicely with the PicknPack policy in this respect, more in particular the need to introduce

[6] `https://tools.ietf.org/html/rfc7159`, and `json-schema.org`
[7] `http://en.wikipedia.org/wiki/JSON-LD`, and `http://www.w3.org/TR/json-ld/`

such UIDs to support tracing in a methodological way.

- *composition*: JSON in itself supports composition natively, which means that a composition of several JSON models is again a valid JSON model. Every element (or so called object) in a JSON model can either be defined in that model itself or can be a reference to another JSON model. This brings multiple advantages:
  - *Reusability*: Composability allows for models of small granularity that can be easily reused by composing them together using this referencing mechanism.
  - *incremental updates*: while some JSON models can be quite long and complex in their entirety, one often only has to send over small "diffs" between an old and new version of the model; for example, each time the ThermoFormer creates a new trays, it suffices to send over only the models of the newly created packages, and not the whole Web model. Since JSON is designed as a composable language, we will make appropriate use of this opportunity to reduce the amount of communication.
- *existing data models*: for some data models there are already suggested standards. And many of these data models come with software for their interpretation, verification, and even visualization, from which the PicknPack developers and specifically the GUI can directly benefit. Some useful data models that can directly be reused are:
  - *time stamps*: time stamps conforming to ISO 8601[8] will be used in the PicknPack project. This is supported by most JSON interpreters natively.
  - *time series*: There are several definitions of time series in JSON[9], most of them directly conforming to a data base.
  - *geometry models*: There are standards like GeoJSON and TopoJSON that model geometry and are widely used (e.g. in Open Street Map). They come with full support for GUI tooling (see section 3.6.1).
- *ontology*: JSON-LD has been designed with lots of the "lessons learned" from previous efforts to make semantic modelling languages, e.g., OWL, or RDF. Not coincidentally, that means that it has the same focus on *composition over inheritance* as the PicknPack software methodology, and that it has native support to model the complex graph-based interactions between knowledge primitives, relationships and constraints that are needed to model, query, and integrate several *ontologies*. This will hopefully be demonstrated in the Skill that automatically adapts the configuration of the Production Policies in all Modules every time a new type of food is being processed on a Line.
- *queries*: Instead of using a REST interface with fixed APIs, the trend is going towards using queries. Query based interface have multiple advantages, which are discussed in section 5. In short some of the advantages are support of hierarchy and sending models instead of data, moving computation to where the data is, typed but flexible content, and the support to send code. Of course these advantages come with the disadvantages that the software needs to be able to handle flexible queries and to interpret models. For the PicknPack GUI we use queries floowing the GraphQL specification[10] because this allows for flexible, JSON-style queries, which follows the rest of the design choices.

[8] https://en.wikipedia.org/wiki/ISO_8601

[9] To name just a few: http://eagleio.readthedocs.org/en/latest/reference/historic/jts.html, http://square.github.io/cube/, https://www.rethinkdb.com/docs/dates-and-times/python/

[10] https://facebook.github.io/graphql/

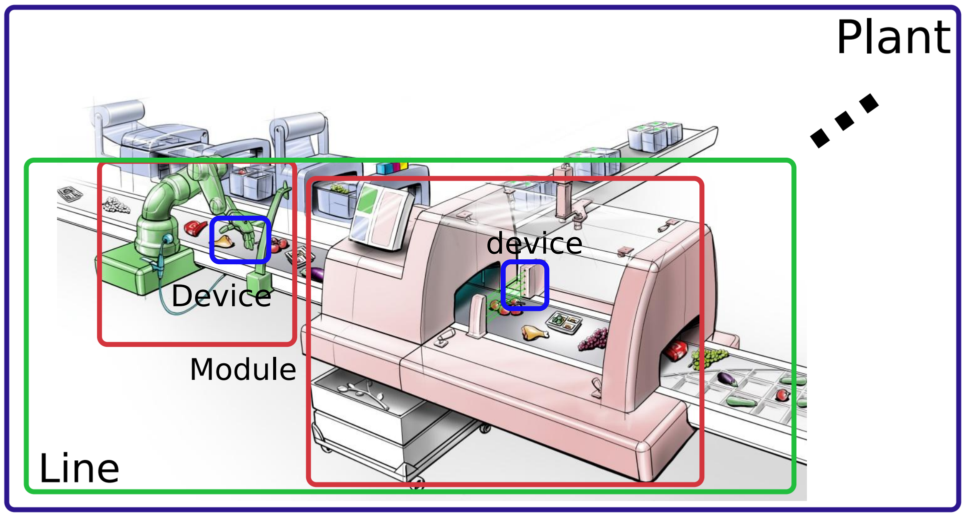


Figure 3: Sketch of the PicknPack production hierarchy of *Device*, *Module*, *Line* and *Plant*.

3.2 Topology Models

The *naming* of the various complementary types of machinery in the PicknPack food processing factory is standardised in the following *hierarchy*, Fig. 3:

- *Plant*: a whole factory, located at a particular place in the world, and containing one or more lines.
- *Line*: a (semi) automated processing infrastructure that takes in boxes of (possibly already processed) food and turns them into *trays*. For example, boxes of green, red, and yellow peppers are input, and the output consists of a packaged trays that contain one pepper of each type. This transformation from the input to the output is done by *modules*.
- *Module*: the top-level processing component in a line, that is provided by one single vendor, and whose production logs must be visible for the tracing. For example, quality inspection, or thermoformer.
- *Device*: a component in a *module*; its production and quality might be kept proprietary to the module vendor. For example, a hyperspectral camera, or a computer-controlled gripper for peppers.
- *Tray*: an output of the line; a tray is formed by the thermoformer module and can have multiple *punnets* for different food.
- *Punnet*: a punnet contains food and is the smallest topological unit. A tray has at least one punnet.

These physical entities in the factory are describing its topology. The individual entities are modelled in separate models and its instances are linked by UUIDs. From the collection of this hierarchical factory model, the containment tree (example shown in figure 4) can be build. Each of these entities contains a geometric model as described in subsection 3.6.2, which is used in the Geometry Widget of the GUI (example shown in figure 2).

3.2.1 Plant

The plant of factory is the highest level of hierarchy in the PicknPack modelling. The model specified in the listing 12 is essentially a container for PicknPack lines (holding their UUIDs) but can be extended with additional information as shown in the example in listing 1 by adding information.

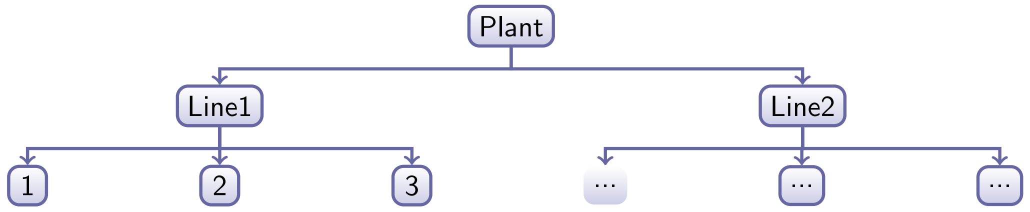


Figure 4: This shows an example for a PicknPack containment tree describing the physical topology of a PicknPack plant. Each node in that tree is an instance conforming to a model and each edge is a link to the UID of that instance. This also conforms to the model described in appendix C
.

```
{
  "uuid": "e8dc7d1f-5b27-414c-8a1e-3a825a7a7a04",
  "name": "Example PnP factory",
  "lines": [
      {
      "model": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/
         raw/json_models/json_models/line_schema.json",
      "line_uuid": "c2c8604d-384e-4308-b869-2e2477b4a1b1"
      },
      {
      "model": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/
         raw/json_models/json_models/line_schema.json",
      "line_uuid": "f43491b2-4c10-4d18-85dd-b66d03790e01"
      }
  ],
  "owner": "EXAMPLE INC",
  "additional_information": {
      "more": "info",
      "even": "more info",
      "money_gained": 200000000
  }
}
```

Listing 1: Example conforming to the PnP Plant Schema specified in listing 12.

3.2.2 Line

Each instance conforming to a line model (listing 13) links to the UUID of the factory it is contained within and holds the UUIDs of the modules and trays that are contained within itself. The examples shown in listing 2 contains three modules and no trays yet. It also has a description of its own shape and position in the topology object.

```
{
  "uuid": "44f2b7c0-cc9b-49f5-83f7-932b545be5f2",
  "description": "This is a line example",
  "name": "Line1",
  "belongs_to": {
      "model": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/raw/json_models/json_models/factory_schema.json",
      "uuid": "98000470-c574-47fd-b3d0-d41f0256270d"
  },
  "modules": [
      {
          "model": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/raw/json_models/json_models/module_schema.json",
          "module_uuid": "9384b059-7fb0-44e7-a4df-cc432edde5ac"
      },
      {
          "model": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/raw/json_models/json_models/module_schema.json",
          "module_uuid": "2a8c6dc0-0f59-4746-bcdd-53909fb96516"
      },
      {
          "model": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/raw/json_models/json_models/module_schema.json",
          "module_uuid": "c7c0c774-a513-4d05-b3da-c2b9c352c926"
      }
  ],
  "web_of_trays": [],
  "topology": {
    "type": "Topology",
    "description":"topology of a line (and position of the line within the factory?); all lengths in m (using the scale parameter in the transform)",
    "transform": {
      "description": "The transform describes the offset of this punnet within the tray coordinate frame",
      "scale": [1000.0, 1000.0],
      "translate": [0.0, 0.0]
    },
    "objects": {
      "shape": {
        "type": "GeometryCollection",
        "description": "The line is approximated with a box here. Multiple ways of description possible, arbitrarily chosen: Geometry collection with a Polygon for each of the six sides. It is possible to approximate a CAD model using TopoJSON and then show the line in 3d if required.",
        "geometries":[
          {"type": "Polygon", "arcs":[[0,1,2,3]],"properties": {"color": "red", "other_property": "random_prop" }}
```

```
            ]
          }
        },
        "arcs": [
          [[0,0],[50,0]],
          [[50,0],[0,50]],
          [[50,50],[-50,0]],
          [[0,50],[0,-50]]
        ],
        "bbox": [[0,0,0],[50,50,1]]
    }
}
```

Listing 2: Example conforming to a PnP Line Schema specified in listing 13.

3.2.3 Module

Each instance conforming to a module model (listing 14) must belong to a fixed set of module types that contain their type specific parameters. This set of module types and their type specific parameters is what would need to be standardized by industry. Each instance also keeps track of the UUID of the line it is contained in as well as of the UUIDs of the devices itself contains and its configurations. It also has a description of its own shape and position in the topology object.

An examples of a thermoformer type module with two devices and one configuration is shown in listing 3.

```
{
  "uuid": "f38a84f6-0d6b-4979-bf41-bf1c982ceb44",
  "description": "This is a module example",
  "name": "Module1",
  "module_type": {
      "type": "thermoformer",
      "specific_propoerties": {
          "batch_type": 1,
          "production_speed": 1.75
      }
  },
  "belongs_to": {
      "model": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/
         raw/json_models/json_models/line_schema.json",
      "uuid": "31327959-4880-45ca-8340-dba6c6c1ccbe"
  },
  "devices": [
      {
          "model": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-
             line/raw/json_models/json_models/device_schema.json",
          "device_uuid": "99a8890c-5bb5-4ff4-975c-592243001781"
      },
      {
          "model": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/raw/json_models/json_models/device_schema.json",
          "device_uuid": "4b894810-01bc-4b55-b9b5-706a0e355137"
      }
  ],
  "configuration": [
      {
          "model": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/raw/json_models/json_models/configuration_schema.json",
          "config_uuid": "cca58211-956a-4a69-a6e8-833092893186"
      }
  ],
  "topology": {
    "type": "Topology",
    "description":"topology of a module; all lengths in m (using the scale parameter in the transform)",
    "transform": {
      "description": "The transform describes the offset of this punnet within the tray coordinate frame",
      "scale": [0.1, 0.1],
      "translate": [0.0, 0.0]
    },
    "objects": {
      "shape": {
        "type": "GeometryCollection",
        "description": "The module is approximated with a box here. Multiple ways of description possible, arbitrarily chosen: Geometry collection with a Polygon for each of the six sides.",
        "geometries":[
          {"type": "Polygon", "arcs":[[0,1,2,3]],"properties": {"color": "green" }}
        ]
      }
    },
    "arcs": [
      [[0,0],[75,0]],
      [[75,0],[0,50]],
      [[75,50],[-75,0]],
      [[0,50],[0,-50]]
    ],
    "bbox": [[0,0,0],[75,50,50]]
  }
}
```

Listing 3: Example conforming to a PnP Module Schema specified in listing 14.

3.2.4 Device

Each instance conforming to a device model (listing 15) must belong to a fixed set of device types that contain their type specific parameters. This set of device types and their type specific parameters is another part of the standardization to be carried out by industry. Each instance also keeps track of the UUID of the module it is contained in as well as of the UUIDs of its configurations. It also has a description of its own shape and position in the topology object.

An examples of a camera type device with two configurations (meaning that the configuration parameters have changed once) is shown in listing 4.

```
{
  "uuid": "e6ede9d6-edb4-468d-8e01-b7f6a140668e",
  "description": "This is a device example",
  "name": "Device1",
  "device_type": {
      "type": "camera",
      "specific_propoerty": {
          "intrinsic_parameters": {
              .
              .
              .
          },
          "extrinsic_parameters": {
              .
              .
              .
          }
      }
  },
  "belongs_to": {
      "uuid": "5dbe2474-620b-4f33-89f8-1ef4ec959a24"
  },
  "configuration": [
      {
          "model": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-
             line/raw/json_models/json_models/configuration_schema.json
             ",
          "config_uuid": "1db83368-cb8c-440a-97cc-62f33a952b97"
      },
      {
          "model": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-
             line/raw/json_models/json_models/configuration_schema.json
             ",
          "config_uuid": "6e99a7a1-d63a-45ff-9edf-f2939c145ffa"
      }
  ],
  "topology": {
    "type": "Topology",
    "description":"topology of a device and position of the device
       within the module; all lengths in m (using the scale parameter
       in the transform)",
    "transform": {
      "description": "The transform describes the offset of this punnet
         within the tray coordinate frame",
      "scale": [0.01, 0.01],
      "translate": [0.0, 0.0]
    },
    "objects": {
      "shape": {
        "type": "GeometryCollection",
        "description": "The device is approximated with a box here.
           Multiple ways of description possible, arbitrarily chosen:
           Geometry collection with a Polygon for each of the six sides
           .",
        "geometries":[
          {"type": "Polygon", "arcs":[[0,1,2,3]],"properties": {"color"
             : "green" }}
        ]
      }
    },
    "arcs": [
      [[0,0],[75,0]],
      [[75,0],[0,50]],
      [[75,50],[-75,0]],
      [[0,50],[0,-50]]
    ],
    "bbox": [[0,0,0],[75,50,10]]
  }
}
```

Listing 4: Example conforming to a PnP Device Schema specified in listing 15.

3.2.5 Tray

Each instance conforming to a tray model (listing 13) links to the UUID of the batch in which it was produced and the UUIDs of the punnets it contains. Furthermore, it keeps a list of features that are attached by the individual modules (see subsection 3.3.2). It also has a description of its own shape and position in the topology object. In addition, it can contain a tray type object, which comes from a fixed set of tray types that have to be defined by the manufacturer depending on the manufactured products. The tray type is an optional key to a list of static tray parameters, like its own shape or number, shape, and location of punnets, that can be stored in a look up table at every entity requiring this information. Thus, the tray type can be used to reduce the amount data to be communicated. However, no flexibility is lost, since all required information is still present in the model itself as well.

An examples of a tray with two punnets an no features yet is shown in listing 5.

```
{
  "uuid": "2f75dd8c-f514-431c-966d-750adce7c813",
  "description": "This is an example of a tray with two punnets",
  "name": "Tray1",
  "belongs_to": {
      "uuid": "2a3f575a-d7f3-410e-ae4c-66d5d449844a"
  },
  "punnets": [
      {
          "model": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-
             line/raw/json_models/json_models/punnet_schema.json",
          "punnet_uuid": "bfa9047b-94c2-4cfd-bbc9-215397f0af36"
      },
      {
          "model": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-
             line/raw/json_models/json_models/punnet_schema.json",
          "punnet_uuid": "bdae0c97-1e1e-4f2e-8aea-b8aec934dd02"
      }
  ],
  "features": [],
  "tray_type": "default",
    "topology": {
                "type": "Topology",
                "description":"topology of the tray and position of the
                    tray within the batch coordinate frame; all lengths
                    in mm",
                "transform": {
                  "description": "The translate in the transform should
                      be used to describe the offset of that tray
                     within the batch.",
                  "scale": [1.0, 1.0],
                  "translate": [0.0, 0.0]
                },
                "objects": {
                  "shape": {
                    "type": "GeometryCollection",
                    "description": "For now it will be approximated
                       with a 2d box for visualization and a 3d
                       bounding box for coordinate calculations",
                    "geometries":[
                      {"type": "Polygon", "arcs":[[1,2,3,4]],"
                         properties": {"color": "green" }}
                    ]
                }
            },
            "arcs": [
                [[0,0],[250,0]],
                [[250,0],[0,250]],
                [[250,250],[-250,0]],
                [[0,250],[0,-250]]
            ],
            "bbox": [[0,0,0],[250,250,100]]
        }
}
```

Listing 5: Example conforming to a PnP Tray Schema specified in listing 16.

3.2.6 Punnet

Each instance conforming to a punnet model (listing 17) links to the UUID of the tray in which it is contained and has a description of its own shape and position in the topology object. It also links to the UUIDs of attached features and can have a punnet type, similar to the tray type above, from a fixed list that can be used to reduce the amount of information that needs to be communicated.

An examples of a punnet of type "punnet_left" and with no features yet is shown in listing 6.

```
{
  "uuid": "89b43197-24d3-43a8-995c-f3772e1908d3",
  "belongs_to": {
      "uuid": "401b1e72-92ba-4d00-b986-03ffa09e288c"
  },
  "name": "Example punnet",
  "punnet_type": "punnet_left",
  "features": [],
  "topology": {
    "type": "Topology",
    "description":"topology of a punnet and position of the punnet
       within the tray; all lengths in m (using the scale parameter in
       the transform)",
    "transform": {
      "description": "The transform describes the offset of this punnet
          within the tray coordinate frame",
      "scale": [0.01, 0.01],
      "translate": [0.0, 0.0]
    },
    "objects": {
      "shape": {
        "type": "GeometryCollection",
        "description": "The punnet is approximated with a box here.
           Multiple ways of description possible, arbitrarily chosen:
           Geometry collection with a Polygon for each of the six sides
           .",
        "geometries":[
          {"type": "Polygon", "arcs":[[0,1,2,3]],"properties": {"color"
             : "green" }}
        ]
      }
    },
    "arcs": [
      [[0,0],[75,0]],
      [[75,0],[0,50]],
      [[75,50],[-75,0]],
      [[0,50],[0,-50]]
    ],
    "bbox": [[0,0,0],[75,50,10]]
  }
}
```

Listing 6: Example conforming to a PnP Punnet Schema specified in listing 17.

3.3 Data Models

Data Models are used to attach additional information to one of the topology models and to cross-cut the hierarchy and link multiple entities together. A brief explanation of the models is given here before describing their models in the following subsections:

- *Batch*: a number of trays produced by the thermoformer in one production cycle. This information is kept to be able to identify problems in the production process of the trays and punnets.
- *Configuration*: a set of parameters and policies influencing the behaviour of modules and devices. Whenever the configuration changes, a new one is added.
- *Feature*: a feature is a generic entity that could greatly benefit from standardization across industry. Every module attaches a specific feature type, like quality information or what food was placed, to a tray or punnet.
- *5P model*: the 5P model (Sec. 3.4) serves the need to make production traceable by linking multiple entities (instances of topology and data models) together in every production step.
- *Data Types*: basic data types like numerical values, strings, lists, tables, etc. that are used to express the data in the data models. They are directly supported by all computer systems and therefore not explicitly modelled.
- *Time Series*: a composed data type that contains a data type recorded and time-stamped over a certain period of time.

In general, data is kept immutable, meaning that instances of data models are never changed but new instances are added (modules might locally discard old ones). That way traceability can be ensured.

3.3.1 Batch

Each instance conforming to a batch model (listing 18) links to the line its trays belong to and to the trays produced in the batch. Also features can be linked from a batch and the geometry of a batch is stored in a topology object. Also a batch type can be assigned that contains the topology and types of the trays within the batch.

An examples of a batch containing four trays and with no features yet is shown in listing 6.

```
{
  "uuid": "6b426916-f8c9-4bcc-ad9c-41b1a4ce6a56",
  "description": "This is an example of a batch with 5 trays",
  "name": "Batch1",
  "belongs_to": {
      "uuid": "77b9b2d7-62df-43d1-ad0e-7bea8d5bfbf0"
  },
  "trays": [
      {
          "model": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/raw/json_models/json_models/tray_schema.json",
          "tray_uuid": "643d7967-3c45-41c8-ab7f-97a348794959"
      },
      {
          "model": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/raw/json_models/json_models/tray_schema.json",
          "tray_uuid": "4469c112-7778-4a5b-85ae-cbaff7adbac2"
      },
      {
          "model": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/raw/json_models/json_models/tray_schema.json",
          "tray_uuid": "3fce4640-4f91-42b0-8d12-8631d5ad17c1"
      },
      {
          "model": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/raw/json_models/json_models/tray_schema.json",
          "tray_uuid": "956d0ddc-7304-4000-9109-c2b965454c25"
      }
  ],
  "features": [],
  "batch_type":{
      "type": "4tray_config_A",
      "traytypes": [["default","default"],["default","default"]]
  }
}
```

Listing 7: Example conforming to a PnP Batch Schema specified in listing 18.

3.3.2 Feature

Each instance conforming to a feature model (listing 19) links to the entity it belongs to. It has a time stamp assigned by the entity by the entity adding the feature and contains metainformation about who assigned that feature using what kind of process and in what configuration. The feature type contains the actual information and is module specific.

An examples of a feature containing of type "quality_grade" is shown in listing 8.

```
{
  "uuid": "8062ef8a-378a-4d6d-b99f-422bb988039d",
  "name": "Example feature",
  "belongs_to": {
      "uuid": "ba87066f-f4b1-45d8-8978-d1fd2854c893"
  },
  "time_stamp": "2015-06-16T18:25:43.511Z",
  "production":{
      "model": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/
         raw/json_models/json_models/production_schema.json",
      "production_uuid": "11e9b92f-dd5e-4590-977d-9c21759516a9"
  },
  "feature_type": {
    "type": "quality_grade",
    "grade": "A"
  }
}
```

Listing 8: Example conforming to a PnP Feature Schema specified in listing 19.

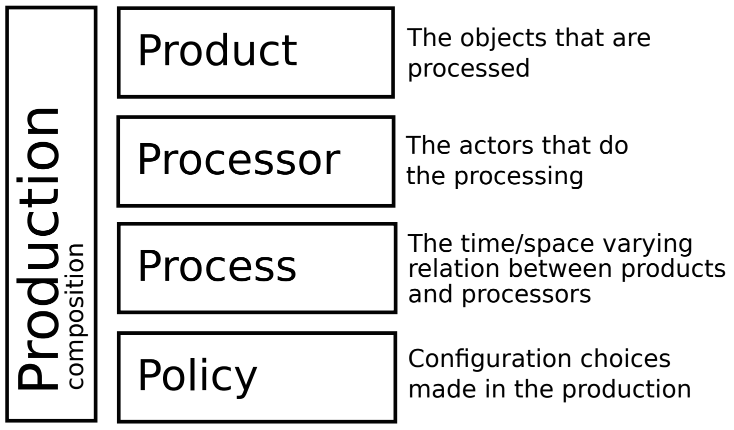

Production
composition
Product
The objects that are processed
Processor
The actors that do the processing
Process
The time/space varying relation between products and processors
Policy
Configuration choices made in the production

Figure 5: The generic "5P" model of "Production".

3.4 5P model of production

The "5P meta model" depicted in Fig. 5 is, again, a structure that serves several complementary design activities:

- first of all, it fills a hole that was still missing in a Picknpack's design "search space", namely the generic model of any *activity*, at any of a system's "levels of abstraction" (Plant, Line, Module, Device).

  With its five constituent components, the 5P meta model is semantically richer than other "Actor models" from the literature (which typically have three), while still being simple enough to be used as a design guideline for all developers, in all applications. "Simple enough" means concretely that the 5P model will help developers identifying whether their software designs are complete, that is, the design has explicit software representations for all relevant bits and pieces.

- it provides the *"provenance" data model*[11] of every manufacturing activity. In the PicknPack context, the *Production* is a batch of produced food packages, where the *Products* are the batches of incoming food and packaging material, the *Producers* are Plants, Lines, Modules or Devices, each having their own production *Process* and production *Policy*.

  It is the project's shared responsibility to provide modelling standards for each of the five constituents of a Production, so the UIDs[12] suffice to serve as keys in a tracing database.

- the two most important consequences of the *5P structure* of a *Production* being the *composition* of the four other parts in the model are:

  - the 5P structure adds extra *Production knowledge* (i.e., relationships, constraints and tolerances) to the models of the four other parts, which implies that any Production must be represented in a system by its own software "agent". The appropriate place to put that software agent is at the highest "level of abstraction" that is involved in the Production.
  - every Production consists of mostly "facts" (i.e., the factual information about the Products, Processors and Process involved in the Production), but its *Policy* is another source of (not necessarily factual) knowledge in the overal system: the rules that determine the configuration of Processors and Process with which the outputs are produced. Again, because it adds knowledge, it must be represented by a software agent, at the appropriate level of abstraction.

3.5 Finite State Machines

Finite State Machines (FSMs) are mainly used for coordination and are crucial for integration. The Life Cycle State Machine (LCSM) and the Stop Light protocol are two examples of "finite state machines" that occur in the PicknPack context. But since Coordination is, in general, present in each and every software module in a PicknPack system, and FSMs are one of the appropriate ways to implement coordination, it makes sense to introduce a standard model for state machines.

A model for so-called "restricted Finite State Machine"[13] is shown in listing 20. The added value of such (huge) modelling efforts is that the *code* to implement FSMs, as well as to communicate about them in a system, can be automatically generated, validated and visualised. A method to to generate code from these models as well as a widget for visualizing them in the GUI have been developed.

```
{
  "type":"rfsm_model",
```

[11]See e.g., http://www.w3.org/2005/Incubator/prov/XGR-prov-20101214/ for description of the term "provenance" and standardisation activities around it.

[12]*Unique IDentifiers*, that is, a code that serves as a unique key to find back all information of the objects involved in a Production.

[13]*M. Klotzbücher and H. Bruyninckx*, Coordinating Robotic Tasks and Systems with rFSM Statecharts. Journal of Software Engineering in Robotics, 3(1):28–56, 2012.

```
  "version":2,
  "rfsm": {
    "type" : "state",
    "transitions" : [
      { "tgt": "inactive", "src": "initial", "events": [] },
      { "tgt": "inactive", "src": "active", "events": ["e_deactivate"]
         },
      { "tgt": "active", "src": "inactive", "events": ["e_activate"] }
    ],
    "containers" : [
      { "id": "inactive",
         "type": "state",
         "transitions": [
          { "tgt": "creating", "src": "initial", "events": [] },
          { "tgt": "configuring_resources", "src":"creating", "events":
              [ "e_done" ] },
          { "tgt": "deleting", "src":"initial", "events": ["
             e_deactivate"] }
         ],
         "containers": [
           { "id": "creating", "type": "state" , "entry": "creating"},
           { "id": "configuring_resources", "type": "state", "entry": "
              configuring_resources" },
           { "id": "deleting", "type": "state", "entry": "deleting" }
         ]
      },
      { "id": "active",
        "type": "state",
        "transitions" : [
          { "tgt": "configuring_capabilities", "src": "initial", "
             events": [] },
          { "tgt": "pausing", "src":"configuring_capabilities", "events
             ": [ "e_done" ] },
          { "tgt": "running", "src":"pausing", "events": ["e_run"] },
          { "tgt": "pausing", "src":"running", "events": ["e_pause"] },
          { "tgt": "configuring_capabilities", "src":"pausing", "events
             ": ["e_configure"] }
        ],
        "containers" :  [
           { "id": "configuring_capabilities", "type": "state", "entry"
              : "configuring_capabilities" },
           { "id": "running", "type": "state", "entry": "running" },
           { "id": "pausing", "type": "state", "entry": "pausing" }
        ]
      }
    ]
  }
}
```

Listing 9: Life Cycle State Machine model as an example conforming to schema in listing 20

3.6 Geometry Models

3.6.1 Existing Standards for Geometry Models

At each of levels of abstraction in PicknPack production units, the need arises to model *where* a particular unit is located, and what its *layout* is. This need has been solved since quite some time in the domain of *Geographical Information Systems* ("GIS"), which has resulted in the GeoJSON[14] and TopoJSON[15] standards. Figure 6 gives an example of a 2D polygonal layout modelled by the TopoJSON code listed below.

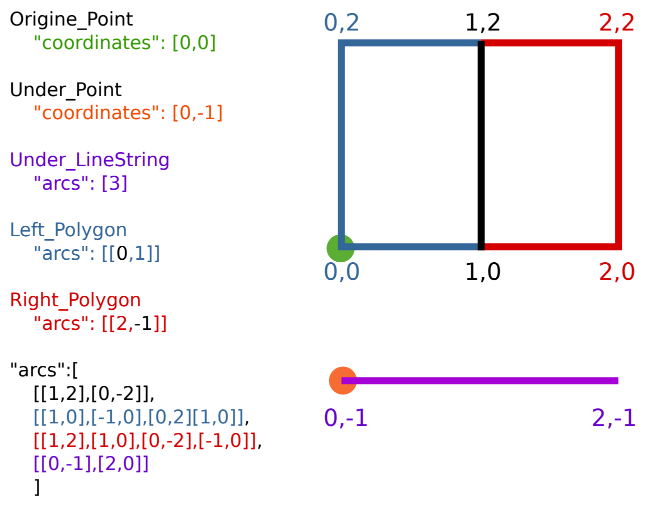


Figure 6: This figure from Wikipedia illustrates the layout modelling primitives of TopoJSON.

```
{
  "type":"Topology",
  "transform":{
    "scale": [1,1],
    "translate": [0,0]
  },
  "objects":{
    "two-squares":{
      "type": "GeometryCollection",
      "geometries":[
        {"type": "Polygon", "arcs":[[0,1]],"properties": {"name": "
           Left_Polygon" }},
```

[14] http://en.wikipedia.org/wiki/GeoJSON, and http://geojson.org/ and http://ld-geojson.org/
[15] http://en.wikipedia.org/wiki/GeoJSON#TopoJSON, and https://github.com/mbostock/topojson/wiki

```
        {"type": "Polygon", "arcs":[[2,-1]],"properties": {"name": "
           Right_Polygon" }}
      ]
    },
    "one-line": {
      "type":"GeometryCollection",
      "geometries":[
        {"type": "LineString", "arcs": [3],"properties":{"name":"
           Under_LineString"}}
      ]
    },
    "two-places":{
      "type":"GeometryCollection",
      "geometries":[
        {"type":"Point","coordinates":[0,0],"properties":{"name":"
           Origine_Point"}},
        {"type":"Point","coordinates":[0,-1],"properties":{"name":"
           Under_Point"}}
      ]
    }
  },
  "arcs": [
    [[1,2],[0,-2]],
    [[1,0],[-1,0],[0,2],[1,0]],
    [[1,2],[1,0],[0,-2],[-1,0]],
    [[0,-1],[2,0]]
  ]
}
```

Listing 10: This is a basic TopoJSON example taken from Wikipedia.

Adoption of TopoJSON as a PicknPack standards makes sense for several reasons:

- coordinate reference system: the TopoJSON standard follows a nice "composition" methodology, in that it allows to refer to any choice of standardized coordinate reference system ("CRS") to interpret its coordinates in. Those CRSs can be world coordinate systems, in which the TopoJSON coordinates are then interpreted as longitude, latitude and elevation; or they can be "projected", local coordinate systems, in which the TopoJSON coordinates are then interpreted as Cartesian X, Y and$Z$ values. At all levels of abstraction (e.g., the PicknPack production units Plant, Line, etc.) a different CRS can be chosen, and several "lower level" layouts can be composed together in the same "higher level" layout model.

- scale: TopoJSON allows to set the scale of the coordinates, which allows to model the layout as a grid with only integer numbers, and with a constant non-integer conversion to real-world coordinates. This helps readability and also reduces the size of the models.

- integration with GUI tooling, e.g. *Leaflet*[16] and other *OpenStreetMap*[17] compatible "world viewers".

[16] http://leafletjs.com/
[17] http://openstreetmap.org/

An example is given in Fig. 2, which shows the layout of an (imaginary) Line with four Modules and several Devices.

3.6.2 TopoJSON for Location and Layout Widgets

For the above reasons, TopoJSON has been adopted for visualizing location and layout of PicknPack entities. The TopoJSON standard[18] is only human readable and, therefore, a machine readable model (in the form of a JSON Schema) has been created which is publicly available[19] and can be found in appendix A.4. An example for a module layout taken from a PicknPck Module can be found in the following listing 11.

The location of static entities like the production units can be directly encoded in the translate object, while for the moving entities (trays) a time series is introduced that receives new elements (time stamp and location) with each encoder tic.

```
{
    "type": "Topology",
    "description":"topology of a module; all arc lengths in mm (using
       the scale parameter in the transform)",
    "transform": {
      "description": "The transform describes the offset of this module
          within the line coordinate frame",
      "scale": [10.0, 10.0],
      "translate": [0.0, 0.0]
    },
    "objects": {
      "shape": {
                "type": "GeometryCollection",
                "geometries":[
                  {"type": "Polygon", "arcs":[[0,1,2,3]],"properties":
                     {"color": "green" }}
                ]
      }
    },
    "arcs": [
      [[0,0],[75,0]],
      [[75,0],[0,50]],
      [[75,50],[-75,0]],
      [[0,50],[0,-50]]
    ],
    "bbox": [[0,0,0],[75,50,10]]
  }
```

Listing 11: TopoJSON example of a topology object taken from a module model. *Scale* is used to keep the arcs integers, while *translate* is used to express the position within the coordinate frame of the entity one level above in the containment tree. The module shape is expressed as a polygon using four arcs. It was agreed among the PicknPack partners that the Line visualisation is 2D for now. Therefore, the arcs are only 2D. However, the bounding box is 3D because the height is relevant for grasping objects.

[18] https://github.com/mbostock/topojson/wiki
[19] https://github.com/nhuebel/TopoJSON_schema

4 Implementations

4.1 Communication

To boost flexibility, the Pick-n-Pack GUI is implemented as a separate process, communicating to the Pick-n-Pack line through sockets, the same way Modules and the Line controller communicate with each other. Three communication types can be identified: a (virtual) data bus, peer-to-peer connections and transactions.

Data bus

The virtual data bus makes use of the Zyre communication library, built on top of ZeroMQ, an open-source socket library. All Module developers also use this library to communicate data and events between each other and the GUI acts as a virtual module, observing these data. The advantage of using Zyre and ZeroMQ over other communication middleware is the fact it takes care of connection bootstrapping, offers local discovery, provides a fast asynchronous message queuing framework and is very lightweight.

Peer-to-peer

The data bus is mainly used to emit events and small data messages, while large data such as 3D camera images is transmitted over individual peer-to-peer sessions. This type of data is on demand, for instance if the user wants to zoom into a particular module and retrieve large data sets. The peer-to-peer connection can be established with the same networking library, ZeroMQ, and the reason for this separate socket connection is mainly its transient nature and the ability to both on the software level as the hardware level send this large data sets only the a select set of peers. On the hardware level a swith applying a spanning tree protocol can be used to ensure the data is not sent to all peers.

Transactions

In order to maintain data consistency all write operations to the Pick-n-Pack database are performed using transactions. By definition, a transaction is an atomic, consistent, isolated and durable unit of work. Moreover, the policy in Pick-n-Pack is to work with immutable data, meaning that all data stored in the database will never be overridden but only appended with updates.

Transactions are well-suited to provide such increments and since they operated in an all-or-nothing fashion, it is guaranteed that the database remains consistent even if a network failure occurs during one of such transactions.

Two distinct databases are used in Pick-n-Pack, both accessible by transactions. The traceability database, of which more details can be found in the Work Package 3 deliverables, takes care of data about the products before they enter the line, what processes have been applied during the line and what happens to the resulting products after they leave the line.

All data about the line configuration, its modules, their operations and states are stored in the world model database on the line itself. For this world model database RethinkDB is used as it offers a rich query language, supports many programming languages by its different client drivers and allows to execute small aggregation and computation blocks on the database itself.

The communication activity in each process should be able to open different physical communication channels for each of these types of communication, since the synchronisation needs and protocols for all three are quite different, mostly because they require different "blocked while waiting for an answer" behaviours.

The reference implementation that is available at the time of writing this document only considers *event broadcasting* as communication requirement, but that is enough to illustrate how to make complex systems out of lots of different subsystems with different and interacting functionalities, resource requirements, and task objectives. More in particular, the communications involve the events required for the *Life Cycle State Machine* (Sec. 3.5) and the *Stop Light* synchronisation.

4.2 GUI

The Gui is a self contained software that works on Firefox, Safari and Chrome desktop size displays. The GUI has three plugins: the state machine, the map of the line down to the device level, and the dynamic view of batch items coursing through the line. The map and the batch view also have additional click, and drag interactions in order to zoom in or out of a detail in the map. The GUI uses a popular graphing library, D3, and some utility libraries, queue.js, jquery, and bootstrap to manage data aquisition, fixtures, and page layout.

The GUI will soon be connected to the Pick and Pack line database, by which it will be able to both receive live messages and send event messages to the line.

The following steps are foreseen for the following months:

- Making a configuration DSL to automatically visualise line, module and device configuration options. This includes, which configuration parameter is read-only or changeable, given the state of the line.
- Making a configuration plugin that can display and change line, module and device configuration.
- Allowing the plugins to talk to the mediator. Integrating the mediator with reactjs in order to automatically update the plugins with the new data.
- Making a GUI DSL to extend the plugins.

When the GUI starts it expects to communicate with the line immediately. The mediator registers the line's semantic data, including the data models the line, modules and devices will send back to the GUI. The Composer retrieves an initial Visualization configuration file on the local file system and maps the plugins, their layouts and the data queries through the Mediator. The Mediator gets the data from the line and pushes it straight to the visualizations: the GUI is rendered. When an event is triggered in a plugin, the plugin sends the message to the mediator who is in charge of passing this message to the right sub-system in the line. When data changes on the line, the mediator is in charge of pushing the changes back to the plugin that will then render the changes.

5 Discussion and conclusion

5.1 Advantages and Trade-Offs

The PicknPack project balances between two rather antipodal goals: *flexibility* and *performance*. The approach described in this Deliverable has the potential to reconcile both, in the following way:

- the flexibility comes from the consequent focus on the *modelling* of the stable semantic aspects of a food production context, in such a way that the same few concepts come back over and over again. More in particular, (i) *composition* as the key to extending functionalities and adding features, (ii) the *mediator pattern* to connect a variable number of line and software components together, and (iii) the choice for *query* based interaction between modules.

- the performance will have to come from the *implementations* of these models. And when the architecture or requirements of a particular food processing context will change, it is expected that re-implementations will be necessary to achieve a specified performance, but without having to change the models.

Here is a non-exhaustive list of cases that can require such re-implementations:

- *communication speed*: when more modules are placed into a line, or the line can produce faster, the amount of communication will increase. The current draft implementations send around (changes in) models in non-binary formats, or in non-aligned messages, which requires parsing all the time. This parsing can be avoided by introducing aligned, binary message formats. The reduction in flexibility is rather limited, as long as one does not expect the layout and operation of a line to change often.

  An increase in the number of modules in a line might require to split up a line into several sub-modules, in such a way that the communication inside each cluster is an order of magnitude higher than in between clusters. This situation is easy to handle with changes in the communication hardware layout, e.g., by introducing a tree-based layout of the communication hubs, or via the more modern approach of *software-defined networking*.[20]

- *embedded hardware for the GUIs*: the current draft implementation has chosen to use the browser as the infrastructure for the GUI, because one can build upon an extremely active community, which has the same objectives towards flexibility, vendor neutrality and model-based declarative descriptions of a system's activities. However, some module vendors might choose not to foresee the hardware support required to work with browser-based technology. In this case, the communication mechanism and architecture based on the Mediator pattern can remain, but the JavaScript implementation will have to be replaced by non-interpreted alternatives, such as Qt.

Of course, also the models will have to be changed, each time a new type of production module is introduced, or a new type of food.

The use of data models everywhere is not a black-and-white choice, but rather a continuum. On the one end of this continuum, it is most flexible to communicate models all the time, but then software needs ability to interpret them. The other end of the continuum is to make the models completely implicit and to send pure data only. Many solutions exist in between, by letting all modules interpret their communication model once at connection time, and then, during runtime, only to send/receive conforming data in an efficient binary format.

The design represented in this Deliverable tries to be as forward-looking as possible, and one of the major parts in this context is the choice for *query* based interfaces between modules. This is only a very recent trend in the world of "the Web", started by large players such as Facebook[21] and Twitter. The major reason to abandon the more traditional *REST* approach[22] is that clients have more influence on what information they can get in one single interaction, which results in a smaller amount of data sent, and fewer interactions needed. The graphical models underlying the modern query approach fit also very well with the JSON models that are introduced in the project: each JSON model *is* a graph, and the graph query languages all use JSON as their host language. Nevertheless, while rationally being a more flexible and future-proof solution (because of its *modelling* quality), the maturity of *implementations* supporting graph queries and graph data bases is less than that of REST or SQL alternatives.

[20] https://en.wikipedia.org/wiki/Software-defined_networking
[21] https://facebook.github.io/react/blog/2015/05/01/graphql-introduction.html
[22] https://en.wikipedia.org/wiki/Representational_state_transfer

5.2 Conclusions

The key advantages of the designs and technologies described in this Deliverable are:

- browser-based technology is the most flexible, cross-plattform and vendor neutral approach (at the time of writing) to make GUIs.
- the consistent use of the *Mediator pattern* helps developers to couple the “insides” and “outsides” of modules, lines, devices, GUIs,. . . in a systematic way, and to allow the maximum efficiency for each *deployment* of a particular sub-system on concrete hardware and a concrete operating system.
- the consistent focus on *data models* is key for realising flexibility. Not in the least in a context where *knowledge*-based module configuration, and *traceability* are important design requirements. Modelling is not an easy development effort, but once realised it makes implementations more easy, more refactorable, or less programming language-dependent.
- the consistent focus on *event streams* as the lowest level of communication is key for realising the decoupling between the communication proper, and the policies that each module has to access and process the data it needs from other modules.
- the consistent focus on given UIDs to everything (data, models, software, events, etc.) is key to realise traceability.

These advantages are possibly compromised by the following challenges:

- only a minority of programmers are already fully aware of, and trained in, the technical approaches mentioned above.
- the trade-off between flexibility and performance is a very difficult one to get right, and, moreover, it might have to change during the life-time of a production plant.
- there is not yet much software tooling available to help developers with both just-mentioned challenges.

Appendices

A JSON Models

A.1 Topology Models

This section contains the models describing the PicknPack topology as described in section 3.2.

A.1.1 Factory Model

```
{
    "id": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/blob/
       json_models/json_models/factory_schema.json",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "PnP factory",
    "description": "PnP factory model",
    "type": "object",
    "properties": {
        "uuid": {
            "description": "Factory UUID",
            "type": "string",
            "pattern": "^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[
               a-fA-F0-9]{4}-[a-fA-F0-9]{12}$"
        },
        "name": {
            "description": "optional factory name",
            "type": "string"
        },
        "lines": {
            "description": "lines in the factory",
            "type": "array",
            "items": {
                "type": "object",
                "required": ["model","line_uuid"],
                "description": "require linked element to conform to
                   this model and to be a UUID",
                "properties": {
                    "model": {"enum": ["https://gitlab.mech.kuleuven.be
                       /rob-picknpack/pnp-line/raw/json_models/
                       json_models/line_schema.json"] },
                    "line_uuid": {"type": ["string"], "pattern": "^[a-
                       fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-
                       F0-9]{4}-[a-fA-F0-9]{12}$"}
                }
            },
            "minItems": 1,
            "uniqueItems": true,
            "description": "require at least one line in the factory
               and that all lines are unique"
        }
    },
    "required": ["uuid","lines"]
}
```

Listing 12: PnP Plant Schema as described in section 3.2.1.

A.1.2 Line Model

```
{
  "id": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/blob/
     json_models/json_models/line_schema.json",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "PnP model of a line",
  "type": "object",
  "properties": {
    "uuid": {
      "description": "Line UUID",
      "type": "string",
      "pattern": "^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F
         0-9]{4}-[a-fA-F0-9]{12}$"
    },
    "name": {
      "description": "optional line name",
      "type": "string"
    },
    "belongs_to": {
        "type": "object",
        "required": ["uuid"],
        "properties": {
            "uuid": {"type": ["string"],"pattern": "^[a-fA-F0-9]{8}-[a-
               fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12
               }$"}
        }
    },
    "modules": {
      "description": "links to the modules on this line",
      "type": "array",
      "items": {
          "type": "object",
          "required": ["model","module_uuid"],
          "properties": {
              "model": {"enum": ["https://gitlab.mech.kuleuven.be/rob-
                 picknpack/pnp-line/raw/json_models/json_models/
```

```
                module_schema.json"] },
            "module_uuid": {"type": ["string"],"pattern": "^[a-fA-F0-
                9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-
                fA-F0-9]{12}$"}
        }
      },
      "minItems": 1,
      "uniqueItems": false
    },
    "web_of_trays": {
      "description": "links to the trays in this line",
      "type": "array",
      "items": {
          "type": "object",
          "required": ["model","tray_uuid"],
          "properties": {
            "model": {"enum": ["https://gitlab.mech.kuleuven.be/rob-
                picknpack/pnp-line/raw/json_models/json_models/
                tray_schema.json"] },
            "tray_uuid": {"type": ["string"],"pattern": "^[a-fA-F0-9]
                {8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA
                -F0-9]{12}$"}
        }
      },
      "minItems": 0,
      "uniqueItems": false
    },
    "topology": {
      "description": "topology of the line (and position within the
         factory); must be valid TopoJSON",
      "$ref": "https://raw.githubusercontent.com/nhuebel/
         TopoJSON_schema/master/topojson.json"
    }
  },
  "required": ["uuid","belongs_to","modules","web_of_trays","topology"]
}
```

Listing 13: PnP Line Schema as described in section 3.2.2.

A.1.3 Moduel Model

```
{
  "id": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/blob/
     json_models/json_models/module_schema.json",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "PnP model of a module",
  "type": "object",
  "properties": {
    "uuid": {
      "description": "Module UUID",
      "type": "string",
      "pattern": "^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}$"
    },
    "name": {
      "description": "optional module name",
      "type": "string"
    },
    "module_type": {
        "description": "description of the module class (used for constraining the configurations; TODO: later to be replaced by link to ontology); IMPORTANT: We need input from the domain experts here!",
        "type": "object",
        "oneOf": [
        {
            "type": "object",
            "description": "model of a thermoformer",
            "required": ["type","specific_propoerties"],
            "properties": {
                "type": {"type": "string", "enum":["thermoformer"]},
                "specific_propoerties": {"type": "object"},
                "optional_propoerties": {"type": "object"}
            }
        },
        {
            "type": "object",
            "description": "model of a label printer",
            "required": ["type","specific_propoerties"],
            "properties": {
                "type": {"type": "string", "enum":["label_printer"]},
                "specific_propoerties": {
                    "label_queue": {
                        "type": "array",
                        "items": {
                            "type": "object"
                        },
                        "minItems": 0,
                        "uniqueItems": true
                    }
                }
            }
        }
        ]
    },
    "belongs_to": {
        "type": "object",
        "required": ["uuid"],
        "properties": {
            "uuid": {"type": ["string"],"pattern": "^[a-fA-F0-9]{8}-[a-
              fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12
              }$"}
        }
    },
    "devices": {
        "description": "devices within a module",
        "type": "array",
        "items": {
            "type": "object",
            "required": ["model","device_uuid"],
            "properties": {
                "model": {"enum": ["https://gitlab.mech.kuleuven.be/rob
                  -picknpack/pnp-line/raw/json_models/json_models/
                  device_schema.json"] },
                "device_uuid": {"type": ["string"],"pattern": "^[a-fA-F
                  0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}
                  -[a-fA-F0-9]{12}$"}
            }
        },
        "minItems": 0,
        "uniqueItems": true
    },
    "configuration": {
        "description": "settings of the module",
        "type": "array",
        "items": {
            "type": "object",
            "required": ["model","config_uuid"],
            "properties": {
                "model": {"enum": ["https://gitlab.mech.kuleuven.be/rob
                  -picknpack/pnp-line/raw/json_models/json_models/
                  configuration_schema.json"] },
                "config_uuid": {"type": ["string"],"pattern": "^[a-fA-F
                  0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}
                  -[a-fA-F0-9]{12}$"}
            }
        },
        "minItems": 0,
        "uniqueItems": true
    },
    "topology": {
        "description": "topology of the module and position of the
           module within the line; must be valid TopoJSON",
        "$ref": "https://raw.githubusercontent.com/nhuebel/
           TopoJSON_schema/master/topojson.json"
    }
  },
  "required": ["uuid","module_type","belongs_to","devices","
     configuration","topology"]
}
```

Listing 14: PnP Module Schema as described in section 3.2.3.

A.1.4 Device Model

```
{
  "id": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/blob/
     json_models/json_models/device_schema.json",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "PnP model of a device",
  "type": "object",
  "properties": {
    "uuid": {
      "description": "Device UUID",
      "type": "string",
      "pattern": "^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F
         0-9]{4}-[a-fA-F0-9]{12}$"
    },
    "name": {
      "description": "optional device name",
      "type": "string"
    },
    "device_type": {
        "description": "description of the device class (used for
           constraining the configurations; TODO: later to be replaced
           by link to ontology) IMPORTANT: We need input from the
           domain experts here!",
        "type": "object"
    },
    "belongs_to": {
        "type": "object",
        "required": ["uuid"],
        "properties": {
            "uuid": {"type": ["string"],"pattern": "^[a-fA-F0-9]{8}-[a-
               fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12
               }$"}
        }
    },
    "configuration": {
        "description": "settings of the device",
        "type": "array",
        "items": {
            "type": "object",
            "required": ["model","config_uuid"],
            "properties": {
                "model": {"enum": ["https://gitlab.mech.kuleuven.be/rob
                   -picknpack/pnp-line/raw/json_models/json_models/
                   configuration_schema.json"] },
                "config_uuid": {"type": ["string"],"pattern": "^[a-fA-F
                   0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}
                   -[a-fA-F0-9]{12}$"}
            }
        },
        "minItems": 0,
        "uniqueItems": true
    },
    "topology": {
        "description": "topology of the device and position of the
           device within the module; must be valid TopoJSON",
        "$ref": "https://raw.githubusercontent.com/nhuebel/
           TopoJSON_schema/master/topojson.json"
    }
  },
  "required": ["uuid","device_type","belongs_to","configuration","
     topology"]
}
```

Listing 15: PnP Device Schema as described in section 3.2.4.

A.1.5 Tray Model

```
{
  "id": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/blob/
     json_models/json_models/tray_schema.json",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "PnP model of a tray",
  "type": "object",
  "properties": {
    "uuid": {
      "description": "Tray UUID",
      "type": "string",
   "pattern": "^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}$"
  },
  "name": {
   "description": "optional tray name",
   "type": "string"
  },
  "belongs_to": {
    "type": "object",
    "required": ["uuid"],
    "properties": {
      "uuid": {"type": ["string"],"pattern": "^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}$"}
    }
  },
  "punnets": {
   "description": "links to the punnets within this tray (using some form of UID)",
   "type": "array",
   "items": {
     "type": "object",
     "required": ["model","punnet_uuid"],
     "properties": {
       "model": {"enum": ["https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/raw/json_models/json_models/punnet_schema.json"] },
       "punnet_uuid": {"type": ["string"],"pattern": "^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}$"}
     }
   },
   "minItems": 1,
   "uniqueItems": false
  },
  "features": {
   "description": "features of a tray that are added by the modules/devices",
   "type": "array",
    "items": {
     "type": "object",
     "required": ["model","feature_uuid"],
     "properties": {
       "model": {"enum": ["https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/raw/json_models/json_models/feature_schema.json"] },
              "feature_uuid": {"type": ["string"],"pattern": "^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}$"}
          }
      },
      "minItems": 0,
      "uniqueItems": true
    },
    "tray_type": {
      "description": "tray type defines which program is run by modules; contains all static information",
      "enum": [ "default", "big_landscape", "big_portrait","small","tomato"]

    },
    "topology": {
        "description": "topology of the module and position of the module within the line; must be valid TopoJSON",
        "$ref": "https://raw.githubusercontent.com/nhuebel/TopoJSON_schema/master/topojson.json"
    }
  },
  "required": ["uuid","belongs_to","punnets","features","tray_type","topology"]
}
```

Listing 16: PnP Tray Schema as described in section 3.2.5.

A.1.6 Punnet Model

```
{
  "id": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/blob/json_models/json_models/punnet_schema.json",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "PnP model of a punnet within a tray",
  "type": "object",
  "properties": {
    "uuid": {
      "description": "Punnet ID",
      "type": "string",
      "pattern": "^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}$"
    },
    "belongs_to": {
        "type": "object",
        "required": ["uuid"],
        "properties": {
            "uuid": {"type": ["string"],"pattern": "^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}$"}
        }
    },
    "name": {
      "description": "optional name/description of punnet",
      "type": "string"
    },
    "punnet_type": {"type": "string", "enum":["punnet_left","punnet_right"]},
    "features": {
      "description": "features of a batch that are added by the modules/devices",
      "type": "array",
      "items": {
          "type": "object",
          "required": ["model","feature_uuid"],
          "properties": {
              "model": {"enum": ["https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/raw/json_models/json_models/feature_schema.json"] },
              "feature_uuid": {"type": ["string","number"]}
          }
      },
      "minItems": 0,
      "uniqueItems": true
    },
    "topology": {
      "description": "topology of the punnet and position of the pocket relative to the tray; must be valid TopoJSON",
      "$ref": "https://raw.githubusercontent.com/nhuebel/TopoJSON_schema/master/topojson.json"
    }
  },
  "required": ["uuid","belongs_to","topology","features"]
}
```

Listing 17: PnP Punnet Schema as described in section 3.2.6.

A.2 Data Model

A.2.1 Batch Model

```
{
 "id": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/blob/json_models/json_models/batch_schema.json",
 "$schema": "http://json-schema.org/draft-04/schema#",
 "description": "PnP model of a batch",
 "type": "object",
 "properties": {
  "uuid": {
   "description": "Batch ID",
   "type": "string",
   "pattern": "^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}$"
  },
  "name": {
   "description": "optional batch name",
   "type": "string"
  },
  "belongs_to": {
    "type": "object",
    "required": ["uuid"],
    "properties": {
      "uuid": {"type": ["string"],"pattern": "^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}$"}
    }
  },
  "trays": {
   "description": "links to the trays within this batch",
   "type": "array",
   "items": {
     "type": "object",
     "required": ["model","tray_uuid"],
     "properties": {
       "model": {"enum": ["https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/raw/json_models/json_models/tray_schema.json"] },
       "tray_uuid": {"type": ["string"],"pattern": "^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}$"}
     }
   },
   "minItems": 1,
   "uniqueItems": false
  },
  "features": {
   "description": "features of a batch that are added by the modules/devices",
      "type": "array",
      "items": {
          "type": "object",
          "required": ["model","feature_uuid"],
          "properties": {
              "model": {"enum": ["https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/raw/json_models/json_models/feature_schema.json"] },
              "feature_uuid": {"type": ["string"],"pattern": "^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}$"}
          }
      },
      "minItems": 0,
      "uniqueItems": true
    },
    "batch_type": {
      "description": "description of the type of batch; maybe redundant because of #number and size of trays?",
      "oneOf": [
        {
          "type": "object",
          "description": "example1: 4 trays in confiburation A (2-2)",
          "required": ["type","traytypes"],
          "properties": {
              "type": {"type": "string", "enum":["4tray_config_A"]},
              "traytypes": {
                  "type": "array",
                  "items": {
                      "type": "array",
                      "items": {"type": "string", "enum":["default"]},
                      "minItems": 2,
                      "maxItems": 2
                  },
                  "minItems": 2,
                  "maxItems": 2
              }
          }
        },
        {
          "type": "object",
          "description": "example1: 8 trays in confiburation D (3-3-2)",
          "required": ["type","traytypes"],
          "properties": {
              "type": {"type": "string", "enum":["8tray_config_D"]},
                "traytypes": {
                    "type": "array",
                    "items": [
                        {
                        "type": "array",
                        "items": {"type": "string", "enum":["default"]},
                        "minItems": 3,
                        "maxItems": 3
                        },
                         {
                        "type": "array",
                        "items": {"type": "string", "enum":["default"]},
                        "minItems": 3,
                        "maxItems": 3
                        },
                         {
                        "type": "array",
                        "items": {"type": "string", "enum":["
                           large_landscape"]},
                        "minItems": 2,
                        "maxItems": 2
                        }
                    ],
                    "minItems": 2,
                    "maxItems": 2
                }
            }
          }
        ]
      }
    },
  "required": ["uuid","belongs_to","trays","features","batch_type"]
}
```

Listing 18: PnP Batch Schema as described in section 3.3.1.

A.2.2 Feature Model

```
{
  "id": "https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/blob/
     json_models/json_models/feature_schema.json",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "PnP description of features; IMPORTANT: We need input
      from the domain experts here!",
  "type": "object",
  "properties": {
    "uuid": {
      "description": "Feature ID added by module/device",
      "type": "string",
      "pattern": "^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}$"
    },
    "name": {
      "description": "optional name/description of feature",
      "type": "string"
    },
    "belongs_to": {
        "type": "object",
        "required": ["uuid"],
        "properties": {
            "uuid": {"type": ["string"],"pattern": "^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}$"}
        }
    },
    "time_stamp": {
      "description": "UTC time when feature was created/added",
      "type": "string", "format": "date-time"
    },
    "production": {
        "type": "object",
        "required": ["model","production_uuid"],
        "description": "require linked element to conform to this model and to be a UUID",
        "properties": {
            "model": {"enum": ["https://gitlab.mech.kuleuven.be/rob-picknpack/pnp-line/raw/json_models/json_models/production_schema.json"] },
            "production_uuid": {"type": ["string"], "pattern": "^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}$"}
        }
    },
    "feature_type": {
      "description": "description of the feature type; IMPORTANT: This needs input from the domain experts",
      "type": "object",
      "oneOf": [
      {
        "description": "feature attached by thermoformer",
        "properties": {
          "creation_encoder_tick": {"type": "integer"}
        },
        "required" : ["creation_encoder_tick"],
        "additionalProperties":true
      },
      {
        "description": "quality_grade: gives grades from A-F",
        "properties": {
          "type": {"type": "string", "enum":["quality_grade"]},
          "grade": {"type": "string", "enum":["A","B","C","D","E","F"]}
        },
        "required" : ["type", "grade"],
        "additionalProperties":true
      },
      {
        "description": "quality_number: ranks quality on a scale from 1
           -10",
        "properties": {
          "type": {"type": "string", "enum":["quality_number"]},
          "grade": {"type": "number", "minimum":0,"maximum":10}
        },
        "required" : ["type", "grade"],
        "additionalProperties":true
      }
      ]
    }
  },
  "required": ["uuid","belongs_to","time_stamp","production","
     feature_type"]
}
```

Listing 19: PnP Feature Schema as described in section 3.3.2.

A.3 Finite State Machine Model

This section contains the developed rFSM schema (section 3.5) used for the visualization in a GUI widget, for code generation, and for coordination at the line level.

```
{
  "id": "http://people.mech.kuleuven.be/~u0072295/rFSM/rfsm-schema",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
      "type": { "enum": ["rfsm_model"] },
      "version": {
        "type": "number"
      },
      "active_leaf_state" : {
        "type": "boolean"
      },
      "active_leaf": {
        "type": "boolean"
      },
      "rfsmgraph": {
        "$ref" : "#/definitions/rfsmgraph"
      }
  },
  "required" : ["type","version", "active_leaf", "active_leaf_state", "rfsmgraph"],
  "definitions": {
    "rfsmgraph": {
      "type": "object",
      "properties": {
        "id" : {
          "type": "string"
        },
        "type" : {
          "enum" : [ "state","transition"]
        },
        "transitions": { "$ref" : "#/definitions/transitions" },
        "subnodes": { "$ref" : "#/definitions/subnodes" }
      }
    },
    "transitions" : {
      "type": "array",
      "items": { "$ref": "#/definitions/transition" }
    },
    "subnodes" : {
      "type": "array",
      "items": { "$ref" : "#/definitions/subnode" }
    },
    "subnode" : {
      "type": "object",
      "properties": {
        "id": { "type" : "string" },
        "type": { "enum": [ "state", "connector"] }
      }
    },
    "transition": {
      "type": "object",
      "properties": {
        "type" : { "enum": [ "transition" ] },
        "tgt": { "type": "string" },
        "src": { "type": "string" },
        "events": {
          "type": "array",
          "items": { "$ref": "#/definitions/event" }
        }
      }
    },
    "event" : {
      "type": "string"
    }
  }
}
```

Listing 20: rFSM Schema

```
{
  "type":"rfsm_model",
  "version":2,
  "rfsm": {
    "type" : "state",
    "transitions" : [
      { "tgt": "inactive", "src": "initial", "events": [] },
      { "tgt": "inactive", "src": "active", "events": ["e_deactivate"]
         },
      { "tgt": "active", "src": "inactive", "events": ["e_activate"] }
    ],
    "containers" : [
      { "id": "inactive",
          "type": "state",
          "transitions": [
           { "tgt": "creating", "src": "initial", "events": [] },
           { "tgt": "configuring_resources", "src":"creating", "events":
               [ "e_done" ] },
           { "tgt": "deleting", "src":"initial", "events": ["
              e_deactivate"] }
          ],
          "containers": [
            { "id": "creating", "type": "state" , "entry": "creating"},
            { "id": "configuring_resources", "type": "state", "entry": "
               configuring_resources" },
            { "id": "deleting", "type": "state", "entry": "deleting" }
          ]
      },
      { "id": "active",
        "type": "state",
        "transitions" : [
          { "tgt": "configuring_capabilities", "src": "initial", "
              events": [] },
           { "tgt": "pausing", "src":"configuring_capabilities", "events
              ": [ "e_done" ] },
           { "tgt": "running", "src":"pausing", "events": ["e_run"] },
           { "tgt": "pausing", "src":"running", "events": ["e_pause"] },
           { "tgt": "configuring_capabilities", "src":"pausing", "events
              ": ["e_configure"] }
        ],
        "containers" :  [
            { "id": "configuring_capabilities", "type": "state", "entry"
               : "configuring_capabilities" },
            { "id": "running", "type": "state", "entry": "running" },
            { "id": "pausing", "type": "state", "entry": "pausing" }
        ]
      }
    ]
  }
}
```

Listing 21: LCSM model as an example of the schema in listing 20

A.4 TopoJSON Model

This section contains the developed TopoJSON Schema (section 3.6.1) split in multiple files referencing each other (Listing 22, 23, 24, 25). An example taken from a PicknPck Module can be found in Listing 11.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "id": "https://raw.githubusercontent.com/nhuebel/TopoJSON_schema/
       master/topojson.json#",
    "title": "TopoJSON object",
    "description": "Schema for a TopoJSON object",
    "type": "object",
    "required": [ "type" ],
    "properties": {
        "bbox": { "$ref": "https://raw.githubusercontent.com/nhuebel/
           TopoJSON_schema/master/bbox.json" }
    },
    "oneOf": [
        { "$ref": "https://raw.githubusercontent.com/nhuebel/
           TopoJSON_schema/master/topology.json"  },
        { "$ref": "https://raw.githubusercontent.com/nhuebel/
           TopoJSON_schema/master/geometry.json" }
    ]
}
```

Listing 22: TopoJSON Object Schema

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "id": "https://raw.githubusercontent.com/nhuebel/TopoJSON_schema/
       master/topology.json",
    "title": "Topology",
    "description": "A Topology object as defined by TopoJSON",
    "type": "object",
    "required": [ "objects", "arcs" ],
    "properties": {
        "type": { "enum": [ "Topology"] },
        "objects": {
            "type": "object",
            "additionalProperties": {"$ref": "https://raw.
               githubusercontent.com/nhuebel/TopoJSON_schema/master/
               geometry.json"}
        },
        "arcs": {"$ref": "#/definitions/arcs"},
        "transform": {"$ref": "#/definitions/transform"},
        "bbox": { "$ref": "https://raw.githubusercontent.com/nhuebel/
           TopoJSON_schema/master/bbox.json" }
    },
    "definitions": {
        "transform": {
            "type": "object",
            "required": [ "scale", "translate" ],
            "properties": {
                "scale": {
                    "type": "array",
                    "items": { "type": "number"},
                    "minItems": 2
                },
                "translate": {
                    "type": "array",
                    "items": { "type": "number"},
                    "minItems": 2
                }
            }
        },
        "arcs": {
            "type": "array",
            "items": {
                "type": "array",
                "items": {
                    "oneOf": [
                        { "$ref": "#/definitions/position"},
                        { "type": "null" }
                    ]
                    },
                    "minItems": 2
                }
            },
            "position": {
                "type": "array",
                "items": { "type": "number"},
                "minItems": 2
            }
        }
}
```

Listing 23: TopoJSON Topology Schema

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "id": "https://raw.githubusercontent.com/nhuebel/TopoJSON_schema/
       master/geometry.json",
    "title": "Geometry objects",
    "description": "A Geometry object as defined by TopoJSON",
    "type": "object",
    "required": [ "type" ],
    "properties": {
        "id": { "type": [ "string", "integer" ]},
        "properties": { "type": "object"  }
    },
    "oneOf": [
        {
            "title": "Point",
            "description": "A Point Geometry object as defined by
               TopoJSON",
            "required": [ "type","coordinates" ],
            "properties": {
                "type": { "enum": [ "Point" ] },
                "coordinates": { "$ref": "#/definitions/position" }
            }
        },
        {
            "title": "MultiPoint",
            "description": "A MultiPoint Geometry object as defined by
               TopoJSON",
            "required": [ "type","coordinates" ],
            "properties": {
                "type": { "enum": [ "MultiPoint" ] },
                "coordinates": {
                    "type": "array",
                    "items": { "$ref": "#/definitions/position" }
                }
            }
        },
        {
            "title": "LineString",
            "description": "A LineString Geometry object as defined by
               TopoJSON",
            "required": [ "type","arcs" ],
            "properties": {
                "type": { "enum": [ "LineString" ] },
                "arcs": {
                    "type": "array",
                    "items": { "type": "integer" }
                }
            }
        },
        {
            "title": "MultiLineString",
            "description": "A MultiLineString Geometry object as
               defined by TopoJSON",
            "required": [ "type","arcs" ],
            "properties": {
                "type": { "enum": [ "MultiLineString" ] },
                "arcs": {
                    "type": "array",
                    "items": {
                        "type": "array",
                        "items": {"type": "integer"}
                    }
                }
            }
        },
        {
            "title": "Polygon",
            "description": "A Polygon Geometry object as defined by
               TopoJSON",
            "required": [ "type","arcs" ],
            "properties": {
                "type": { "enum": [ "Polygon" ] },
                "arcs": {
                    "TODO": "Check if arcs refer to valid LinearRings",
                    "type": "array",
                    "items": {
                        "type": "array",
                        "items": {"type": "integer"}
                    }
                }
            }
        },
        {
            "title": "MultiPolygon",
            "description": "A MultiPolygon Geometry object as defined by TopoJSON",
            "required": [ "type","arcs" ],
            "properties": {
                "type": { "enum": [ "MultiPolygon" ] },
                "arcs": {
                    "type": "array",
                    "items": {
                        "type": "array",
                        "items": {
                            "type": "array",
                            "items": {"type": "integer"}
                        }
                    }
                }
            }
        },
        {
            "title": "GeometryCollection",
            "description": "A MultiPolygon Geometry object as defined by TopoJSON",
            "required": [ "type","geometries" ],
            "properties": {
                "type": { "enum": [ "GeometryCollection" ] },
                "geometries": {
                    "type": "array",
                    "items": { "$ref": "https://raw.githubusercontent.com/nhuebel/TopoJSON_schema/master/geometry.json" }
                }
            }
        }
    ],
    "definitions": {
        "position": {
            "type": "array",
            "items": { "type": "number"},
            "minItems": 2
        }
    }
}
```

Listing 24: TopoJSON Geometry Schema

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "id": "https://raw.githubusercontent.com/nhuebel/TopoJSON_schema/
       master/bbox.json",
    "title": "TopoJSON bounding box",
    "description": "A bounding box as defined by TopoJSON",
    "type": "array",
    "items": { "$ref": "#/definitions/dimension" },
    "minItems": 2,
    "maxItems": 2,
    "definitions": {
      "dimension": {
          "type": "array",
          "description": "This array should have an entry per dimension
              in the geometries",
          "items": {"type": "number"}
      }
    },
    "TODO": "check number of dimensions (2*n), n being the number of
       dimensions represented in the contained geometries), with the
       lowest values for all axes followed by the highest values "
}
```

Listing 25: TopoJSON Bounding Box Schema

B Communication

This section contains a paper submitted to the Journal of Software Engineering for Robotics. It contains a discussion of communication middlewares and mechanisms. Then it introduces communication patterns together with example use cases, some of which were developed for PicknPack.

Communication Patterns in Robotics

Johan Philips[1] Nico Huebel[1] Herman Bruyninckx[1,2]
[1] Department of Mechanical Engineering, KU Leuven, Leuven, Belgium
[2] Department of Mechanical Engineering, Eindhoven University of Technology, Eindhoven, the Netherlands

Abstract—In almost all robotics applications nowadays, communication plays a prominent role, especially when multiple robot systems are involved. Nevertheless, it receives little attention during development. The notion of a communication stack has been abstracted away by robotics middlewares, trying to solve all use cases by a one-size-fits-all implementation. Although the intention of middleware developers is to simplify application building, it usually also takes away flexibility. Therefore, an application dependent trade-off needs to be found that allows customizing communication to specific applications. We argue that communication should be modelled first using communication patterns that are independent of a specific communication middleware. Afterwards, the communication model can be implemented using an existing middleware that supports the used communication mechanisms required by the chosen patterns. In this paper, we introduce a set of communication mechanisms and patterns, built on top of them, which have emerged in our recent multi-robot projects. Among other things, they solve typical communication issues like failure detection, discovery, communication bootstrapping, and task coordination. In addition, we give use cases for the presented patterns and refer to their formalizations done by domain experts. Another, often overlooked, fact is that communication happens at different abstraction levels within one application. Communication can refer to information exchange between robots, between (local or networked) software components or devices, and even between computations within the same algorithm. However, similar patterns re-occur on multiple levels of abstraction, e.g., for multi-robot oordination and task coordination or (distributed) algorithm execution. Communication patterns which haven been modelled and formalised in a DSL and/or protocol potentially result in re-usable implementations by mature communication libraries. Separating communication mechanisms from communication middleware and from patterns built from these mechanisms, is an important step towards a model-driven engineering approach that includes flexible communication.

Index Terms—Communication Patterns; Networked Robots; Multi-Robot Systems; System architectures, integration and modeling; Network Architecture and Design

1 Introduction

In almost all robotics applications nowadays, communication plays a prominent role, especially when multiple robot systems are involved. Nevertheless, it receives little attention during development and, typically, is taken for granted: unlimited bandwidth is assumed, perfect uptime is expected and little thought goes into customising communication to the specific application at hand.

Regular paper – *Manuscript received September 1, 2015; revise, 2015.*

- *This work was supported by the University of Leuven Geconcerteerde Onderzoeks-Acties* Model based intelligent robot systems *and* Global real-time optimal control of autonomous robots and mechatronic systems, *the University of Leuven IOF Kennisplatform* Transition, *and from the European Union's 7th Framework Programme (FP7/2007–2013) projects* BRICS *(FP7-231940,* Best Practice in Robotics*),* SHERPA *(FP7-600958,* Smart collaboration between Humans and ground-aErial Robots for imProving rescuing activities in Alpine environments*), and* PicknPack *(FP7-311987,* Flexible robotic systems for automated adaptive packaging of fresh and processed food products.

On the contrary, since the introduction of ROS [1] into robotics, the majority of robotics application developers typically use only one type of communication anymore: pub-sub over a *ROS Master*. The notion of a communication stack has been abstracted away, too much so, trying to solve all use cases by a *one-size-fits-all* solution. Also the upcoming *ROS* 2.0 seems to follow the same path by choosing only one type of middleware, DDS [2], and preconfiguring it as much as possible beforehand, with the intention *to ease* the design of communication needs for the robotics application developer. However, such a focus on "ease of use" is also taking away *flexibility*, which can lead to unexpected limitations, or require clumsy workarounds. An application dependent trade-off between the "Freedom from Choice vs. Freedom of Choice" [3] has to be found.

Moreover, many people in robotics suffer from the "Not Invented Here Syndrome" and instead of making use of existing communication middlewares, they keep reinventing the wheel without getting close to the mature, existing communication middlewares.

We argue that communication should be *modelled* first using communication mechanisms that are independent of a specific communication middleware. Afterwards, the communication model can be *implemented* using an *existing* middleware that supports the used communication mechanisms and fits the use case. In addition, certain communication patterns, which make use of the communication mechanisms, have been useful in our recent projects. Therefore, this paper focuses on introducing communication mechanisms as well as communication patterns which proved useful in multi-robot systems and have been formalised in a DSL and/or protocol in some way.

The separation of the communication mechanisms from the communication middleware as well as the patterns built from these mechanisms are important steps towards an model-driven engineering approach that includes flexible communication.

Section 2 will review existing communication libraries and middlewares both within and outside the robotics field. In Section 3 communication mechanisms are introduced, followed, in Section 4, by useful communication patterns which build upon the introduced mechanisms, and are explained together with potential robotics use cases. In the discussion section (5) trade-offs and guidelines are presented to help application developers. Finally, section 6 concludes the paper.

2 State of the art

This section reviews several mature communication libraries and middlewares, data representation and serialisation libraries, developed outside robotics and compares them to specific robotics middlewares.

2.1 Communication Middlewares

All introduced libraries and middlewares here are considered *mature*, which means they are well documented, are formalised in a DSL and/or protocol, are adopted by industry and/or a large user community, and are accompanied with detailed examples and recommended use cases.

Enterprise messaging systems, such as [4], [5], [6], focus on offering a rich set of pre-defined policies, based on a set of enterprise-wide standards.

The Apache Software Foundation provides numerous projects that are useful with respect to communication. Apache Kafka [7] processes messages at scale. Kafka is a distributed pub-sub real-time messaging system that provides strong durability and fault tolerance guarantees. Apache Camel [8] is an integration framework providing routing and mediation rules based on enterprise integration patterns [9]. Apache Qpid [6] is a set of messaging tools based on the Advanced Message Queuing Protocol (AMQP) [10]. Apache Avro [11] is Apache's data serialisation system, offering a compact, fast, binary data format, container files to store persistent data, remote procedure calls and integration with dynamic languages. Moreover, is relies on schemas when reading or writing data and during RPCs.

Data Distribution Service (DDS) [2] for real-time systems, is a communication standard from the Object Management Group (OMG) aiming at distributing data in real-time between applications, using the publish-subscribe paradigm. OpenDDS [12] is a C++ implementation of this OMG standard, sponsored by Object Computing, Inc. (OCI), who have also been involved in the development of the ADAPTIVE Communication Environment (ACE) [13]. OpenDDS makes use of ACE to provide a cross platform environment. Data Distribution and Storage Service (DDSS) [14] makes use of DDS's asynchronous interaction and its dynamic discovery to support mobile and pervasive systems.

The Extensible Messaging and Presence Protocol (XMPP) is an XML-based communication protocol that enables the near-real-time exchange of structured yet extensible data [15]. The core protocol contains methods to open and close XML streams over TCP connections including channel encryption, authentication, and error handling as well as communication primitives for messaging and request-response interactions. Among other things, experimental extensions for sensors, control, and discovery are developed within the Internet of Things community [16].

ZeroMQ [17], [18] is a lightweight socket library, offering several *socket types* for different deployment levels and various communication patterns on top of TCP sockets, as well as discovery over UDP. It is known for its high performant asynchronous communication and provides bindings to many programming languages. The ZeroMQ Guide [18] helped in the search for better communication patterns for robotics.

Common Object Request Broker Architecture (CORBA) [19] is a standard that was created with interoperability of operating systems and programming languages in mind. CORBA specifies a mapping from a so called interface description language to operating system and language specific implementations. While the basic idea of CORBA is good, it suffers from its extensive, monolithic structure that is hard to implement as a whole.

2.2 Data Representation and Serialization

There are multiple ways to represent data structures and to serialise them for communication. Here, commonly used methods to represent and serialise data are presented.

Comma Separated Values (CSV) [20] are mainly storing tabular data in plain text and are mainly used to exchange such data between databases and programs that share no other exchange format. Despite its structure and the fact that data is in plain text, large data sets are not easily human-readable, and this format usually lacks metadata to interpret data. Therefore, it is not very useful in robotic applications.

The Extensible Markup Language (XML) [21] is a standardised, widely-adopted, human and machine-readable format designed for use across the Internet. Many data formats like SVG, URDF, OWL, or formats of common office software have been derived from XML. XML documents can be validated by defining schemas and multiple schema definition languages are available. Even though XML was designed to be human readable, complex XML documents are generally more difficult to interpret than JSON or YAML documents.

JavaScript Object Notation (JSON) [22], [23] is a standardised human and machine-readable format to transmit data objects. Contrary to its name, it is language independent. JSON Schemas [24], [25] can be used for validation and documentation. JSON's big advantage is its wide adoption and, therefore, existence of many tools and composable domain specific extensions like GeoJSON [26]. It is also a native format of modern browsers, which enables easy integration with browser based GUIs.

YAML (YAML Ain't Markup Language) [27] is a human and machine-readable data format, which is relying on indentation to structure data instead of delimiters such as brackets or tags. This makes it especially easy to read and write. However, it can also be error prone due to mixing white spaces and tabs and the fact that a single additional white space in a line is hard to spot but still results in a valid YAML document but with a different meaning. YAML can be validated using YAML's type declarations in the document itself or using externally defined schema description languages.

The Open Data Description Language (OpenDDL) [28] is a human-readable format that requires each unit of data to be explicitly typed. It is composable but there is no validation for the core format available.

Other widely used serialisation libraries, known for their speed and compactness, are Protocol Buffers [29], Thrift [30] and MessagePack [31].

2.3 Robotic Middlewares

Some popular robotic middlewares are reviewed here with respect to their flexibility in choosing and configuring the communication.

ROS [1] offers essentially a message based publish subscriber system and a request reply system called services. In both cases connections are setup by a central broker, *ROS Master*, that negotiates and appropriate transport protocol and sets up a TCP or UDP connection using XMLRPC. This central master is a single point of failure and also makes it difficult to distribute the system. Choosing other modes of communication requires implementing specialised nodes providing such functionalities. In [32], the message system of ROS is compared to ZeroMQ's message queues with respect to space and time scalability and results were very much in favour of ZeroMQ. Latency tests showed more than a tenfold difference between both implementations at a high number of subscribers.

For the upcoming *ROS* 2.0 a switch to DDS was announced. While DDS provides a flexible but complex system, it is planned to hide most of this complexity, and thus also flexibility, from its users [33]. Developers are, however, foreseeing access to DDS's API for specialised users, who require full flexibility, DDS offers.

Mechanism	Immediacy	Cardinality	Direction
Request-reply	Sync	1-to-1	2-way
Publish-subscribe	Async	1-to-many	1-way
Push-Pull	Sync	many-to-1	1-way
Dealer-Router	Async	many-to-many	2-way

TABLE 1
Communication mechanisms and their characteristics (policies).

Orocos [34], [35], [36], [37] offers lock-free data ports implemented using Posix MQueue or a communication transport conforming to CORBA. Orocos was designed to be independent of underlying communication mechanisms, however in practice, only few bindings to additional transport mechanisms have been developed.

To the best of our knowledge the SmartSoft framework is the only robotic middleware that explicitly defines a set of communication patterns. While they aim for a set of minimal but usable patterns [38], we explicitly decouple mechanisms and provide patterns for a set of more specific robotics use cases. Therefore, both approaches are complementary.

[39] contains a good overview of robotic frameworks and how they share information between different software modules. However, the proposed OpenRDK framework only introduces blackboards and data-ports as means of communication. It makes communication between threads, processes, and over the network transparent for users by automatically establishing communication via shared memory or a configurable TCP/UDP connection.

Microsoft Robotics Studio uses the Decentralized Software Services Protocol (DSSP). It distinguishes communication between processes on the same processing unit, different processing units, and over the network and minimises overhead accordingly [40]. It uses the SOAP protocol over TCP/IP, if network communication is required. Since it is limited to the .NET environment, it does not satisfy interoperability requirements and, in addition, its development is suspended.

3 Mechanisms

This section describes a set of mechanisms, listed in TABLE 1, which are used to formulate the communication patterns in Section 4.

Firstly, a distinction between synchronous and asynchronous communication is made. The former implies blocking calls and ordering of data flow, the latter implies immediately returning from a receive or send call, which implies polling is required. Secondly, cardinality changes between mechanisms, from 1-to-1 communication, over 1-to-many or many-to-1, to many-to-many. Thirdly, data flow direction is defined as either unidirectional (1-way) or bidirectional (2-way).

Each of these mechanisms and policies have their merit in particular use cases, and application developers should take this into account to design their application accordingly.

3.1 Request-reply

Request-reply, depicted in Figure 1, is a communication mechanism that is commonly well-understood and characterised by *bidirectional, synchronous communication*. A client connects to a server, which is offering a set of services, sends a particular request and waits until it receives a reply. The server, on the other hand, listens for incoming connections, accepts the client connection, receives the request and sends a reply to the client. The reply can, for instance, be the result of some computation that runs in the background.

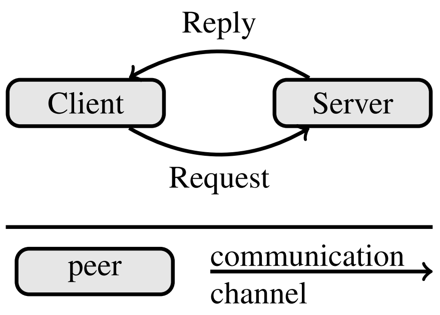


Fig. 1. Conceptual drawing of Request-reply. In this bidirectional synchronised communication mechanism, the client sends a request first and blocks until it receives a reply, while the server waits for a client connection and then receives a request and responds with a reply.

In a typical request-reply (or client-server) architecture, communication is 1-to-1 and *synchronous*. The client blocks until it receives a reply, i.e. it cannot send multiple requests without having received any replies. Also the server is blocking until it receives a request. A server can only send a reply when it is offered a request. It cannot, for example, broadcast messages to all its connected clients at a preconfigured frequency.

Some communicaiton libraries, such as ZeroMQ, transparantly add a many-to-1 request-reply mechanism between clients and a server. Opening one reply socket at the server-side is sufficient to accommodate as many request sockets at client-side as needed.

Robotics use case

An example use case for request-reply would be to decouple components running with a different time horizon, e.g. a real-time robot controller and a non-real-time motion generator. Whenever the robot controller requires new input, a request is sent to the motion generator to receive a new set point. The motion generator guarantees a reply within a deterministic time and this new set point is fed back to the robot controller. If a non-real-time client requests an update of the whole trajectory, this can be handled in background by only replacing the trajectory once the new one is computed.

Synchronous request-reply is also useful to imitate remote procedure calls (RPC) over a message-oriented technology.

3.2 Publish-subscribe

This mechanism, depicted in Figure 2, is dominating the robotics community since the introduction of ROS, overshadowing other useful mechanisms and patterns.

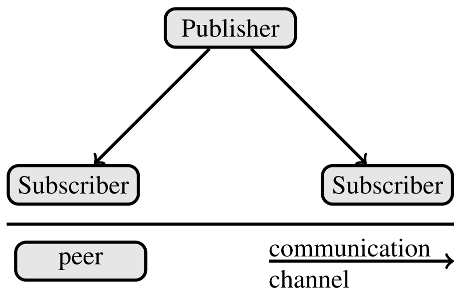


Fig. 2. Conceptual drawing of the unidirectional asynchronous communication mechanism Publish-Subscribe. This example shows one publisher sending data to all its subscribers, either periodically or whenever new data is available.

It is characterised by *unidirectional, asynchronous communication*. A publisher publishes data on a particular channel, whereas a subscriber, subscribes to this channel and receives all data put on that channel after its subscription.

Publishers typically don't need to know about their subscribers, which is both the strength and weakness of this pattern. It offers scalability, since the publisher's tasks is to just publish data, but it also causes *slow joiners* to miss (possibly important) data transmitted before they subscribed.

Moreover, publish-subscribe is not suited for *reliable multicast*, since by design it kills back-chatter (subscribers are not allowed to send data to the publisher). This improves scalability but does not allow coordination between publishers and subscribers, because publishers are not aware when subscribers have connected, disconnected, disappeared or crashed and subscribers cannot ask publishers to limit their publishing rate.

Robotics use case

Many robotics software applications use publish-subscribe to distribute data between software components. Typical examples are sensor measurements which are published at a particular frequency or a localisation component broadcasting the estimated robot pose to all subscribers.

3.3 Push-pull

This mechanism, depicted in Figure 3, is characterised by *unidirectional, synchronous communication* and is used to synchronise work flow of different computations in different processes. Typically, a *ventilator* process divides work (i.e. computations) in an evenly fashion among a set of *worker* processes. After a worker finishes its computations it pushes

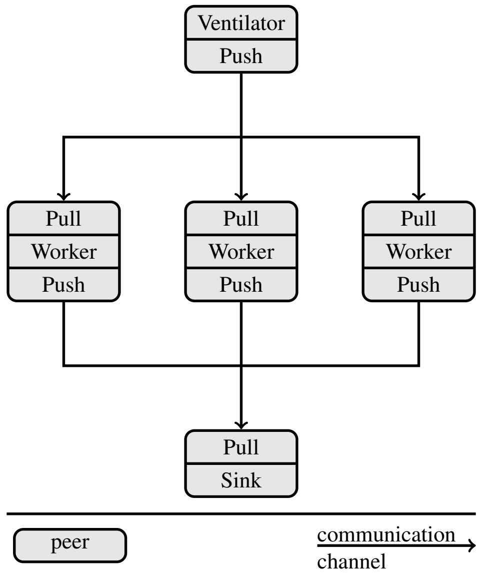


Fig. 3. Conceptual drawing of a pipeline using push-pull with a ventilator, a set of workers and a sink process. In this unidirectional synchronous communication mechanism, pushed data is distributed evenly to pull sockets.

the result to a *sink* process, which pulls all results from all workers using fair queueing.

This mechanism also suffers from *slow joiners*, meaning that fast connecting pull sockets receive more tasks and the distribution of tasks might not be evenly, which can result in longer execution time. This issue can be solved by introducing load balancing algorithms, which require additional communication.

The main difference between push-pull and request-reply mechanism is that communication is unidirectional, meaning replies do not necessarily go the requesting process (Ventilator and Sink do not have to be the same process).

Compared to publish-subscribe, push sockets have a *one of many* policy, while publish sockets have a *all of many* policy, i.e. a push socket sends a message to only one connected pull sockets while a publish socket sends it to all connected subscribers.

Robotics use case

The pipeline depicted in Figure 3 is useful in multi-robot applications in which homogeneous robots have equal capabilities and have to perform a set of tasks in a synchronised way. A coordinator (ventilator in the figure) can push a new task to one robot (worker in the figure). Whenever a robot has finished, it pushes the result to a component responsible for aggregation (sink in the figure). Another example could be the analysis of a large point cloud that is split into separate pieces, each analysed by a different component and the results are merged afterwards.

3.4 Dealer-Router

This mechanism, depicted in Figure 4, is characterised by *bidirectional, asynchronous communication*. While request-reply is considered *one to one* and publish-subscribe or push-pull are *one to many*, some use cases also require a *many to many* connection where back-chatter is possible. This bidirectional support allows to build more reliable communication between peers with faster failure detection.

A Router is basically an asynchronous version of the Reply socket used in the request-reply pattern, while a Dealer socket is an asynchronous version of the Request socket. Because of the asynchronicity, different combinations are possible: Reply-Router, Dealer-Reply, Router-Dealer, Dealer-Dealer, Router-Router. The last three offer full asynchronous many to many communication and differ in how much bookkeeping is done behind the scenes. A Router socket keeps track of which clients sent which requests and automatically adds correct message envelopes, while a Dealer socket does not.

More concretely, routers are extremely useful in network configurations with many intermediate points. A message header will be added by each Router socket in between source and destination, identifying the sender. This results in a complete trace of where a message has been and how a reply could eventually be routed back to the source. All routing information is added to the response such that each router on the way back knows how to propagate the message by popping off one piece of routing information.

In Figure 4, a broker example is shown. The Router socket ensures replies are forwarded back to the right clients and the Dealer socket distributes requests to workers offering a particular service. Using a broker improves scalability, since clients are not aware of workers and vice versa.

Robotics use case

Router and Dealer sockets are most useful in multi-robot systems with multi-hop networks to abstract away message routing or in setups with reliable asynchronous communication requirements. In a search-and-rescue mission, for example, each robot could maintain a Router socket for incoming communication and a Dealer socket for each peer robot, which connects to that peer robot's Router socket. The Router socket ensures replies are propagated back correctly over the Dealer socket and if communication coverage is limited, one of the robots can act as relay between two robot teams.

3.5 Communication DSL

All the above-described mechanisms have been formalised in a set of RFCs and the wire protocol referencing these mechanisms, ZeroMQ Message Transfer Protocol (ZMTP), is formalised in [41]. This *is* already a DSL, mature and time proven, and hence ready to be exploited in a robotics context, to a much larger extent than what is currently the case.

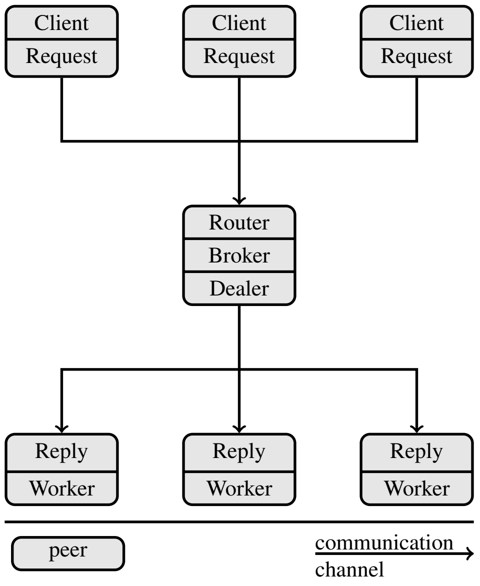


Fig. 4. Conceptual drawing of a broker using Dealer and Router sockets. In this bidirectional asynchronised communication mechanism, the Router socket ensures replies are forwarded back to the right clients and the Dealer socket distributes requests to workers offering a particular service.

4 Communication Patterns

In this section we will highlight a set of communication patterns, which are relevant for robot software design, especially in multi-robot systems. The purpose is not to provide an exhaustive list, but to clearly indicate that the typical *Publish-Subscribe* pattern, which is adopted by many robot system developers nowadays, can impede your possibilities with respect to communication while better alternatives exist.

The presented patterns have been useful in our recent projects and can be implemented using the ZeroMQ socket library and/or the Zyre library, built on top of ZeroMQ. But as mentioned already, the validity of the presented patterns depends nowhere on this particular implementation.

4.1 Heartbeating

A basic prerequisite in reliable communication is to be able to detect connection issues, disconnects, or peer failures. This can be easily achieved by *heartbeating* between peers in the network. Peers will send, at a particular frequency a small *alive* message to each other, indicating the connection is still up and running. Important to note is that, typically, every data received from one of the peers, is also considered as a heartbeat message and heartbeats are sent on the same socket as data, to act as a *keep-alive* for the connection.

Figure 5 depicts a conceptual drawing of a heartbeating scenario between three peers connected to each other with dealer sockets.

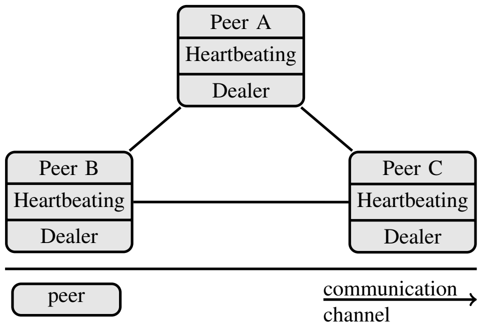


Fig. 5. Conceptual drawing of heartbeating scenario between three peers connected to each other with dealer sockets. If no data is exchanged between two peers for a configurable time period, a heartbeat signal is sent to notify others the peer is still alive. If no heartbeats (or data) is received after a expiry time, the peer will be assumed dead and a particular recovery policy can be applied.

A more advanced way of heartbeating is a two-way ping, where each peer sends a *ping* on inactive channels and checks if it receives a *ping ok* back from the peers on these channels.

Robotics use case

In a multi-robot application with uncertain network quality, e.g. in search and rescue missions, heartbeating is essential to detect poor communication channel quality or peer failures.

4.2 Shared state

In many multi-robot applications it is required to maintain a single, consistent state which is shared among different robots. There are different scenarios that require a various amount of communication. We will discuss some examples in the use case section below. The most complex shared state example with respect to communication is shown in Figure 6.

Robotics use case

The simplest use case for a shared state can be applied if robots or processes can join at any time but require a common configuration. After joining the network, they request their configuration from a central server. This simple case would only require a request-reply mechanism.

The next use case is a so-called *blackboard* which multiple entities can use to share data [42]. The ROS parameter server is an example of a blackboard. A blackboard requires data management to prevent different entities from reading and writing to the same data at the same time. This use case corresponds to having the *state update* and *state request* arrows in Figure 6.

If multiple robots require the same, consistent state, the previous use case is not sufficient since all robots require an update of this state whenever it is changed. An example

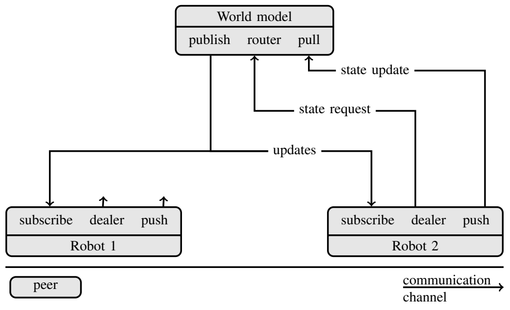


Fig. 6. Conceptual drawing of shared state pattern between a world model peer and two connected robot peers. The goal of this pattern is to maintain a consistent state between several peers in a network. The world model publishes new updates to its peers, while each robot peer can push their update to the world model. Whenever a new robot peer joins the network it is able to request the full state through a dealer-router connection.

for this use case is a shared world model, which aggregates information from multiple robots. To achieve a consistent shared state between robots, a combination of communication mechanisms can be used, as depicted in Figure 6:

- Publish-Subscribe to publish updates from the world model to its peers whenever the world model (or a relevant part of it) has changed.
- Push-Pull to allow peers to push their update to the world model.
- Dealer-Router to allow peers to send state requests to the world model, whenever they join the network or when an inconsistency due to package loss was detected.

A further example is a collaborate task that requires multiple robots *to execute sub-tasks in a coordinated way*. The state of the task execution is shared among all robots. Each robot pushes changes related to its own execution to the peer managing the overall task state, which notifies the relevant other robots. This allows them to react accordingly (e.g. by stopping or starting their execution). If a new robot connects, it can request the execution state to check if it can start with the execution of its own subtask.

4.3 True peer connectivity & local discovery

In some use cases, where communication quality is uncertain and dropouts might occur frequently, a centralised solution is not an option. For example, search-and-rescue missions cannot rely on communication to be flawless and if one node in the network acts as a central connection point, all communication is lost if this node fails or drops out. Moreover, in situations where a lot of nodes are present, a centralised solution would introduce a bottleneck since it has to process data from all peers.

Typically, setting up such a peer-to-peer network requires configuration files which are synchronised among all peers, which is cumbersome and becomes unsustainable in large networks, especially if in a heterogeneous group of robots with many different development teams involved.

True peer connectivity exemplifies brokerless messaging in a many-to-many network where all peers can act as clients or servers and connections are setup properly whenever new peers arrive. It is, in fact, solving a bootstrapping problem caused by the asymmetry of TCP (one side binds, while the other connects): peer A can talk to peer B after peer B has talked to peer A, but peer B can't talk to peer A until peer A talked to peer B.

This problem is solved by broadcasting UDP beacons containing connection information. Next an asynchronous request socket (dealer) is setup by each peer receiving the beacon, which connects to that peer's asynchrouns reply socket (router), found in the beacon. To ensure discovery is synchronised, each peer sends a hello message to every newly connected other peer. It could happen that the first beacon from a peer is received after you receive messages from that peers.

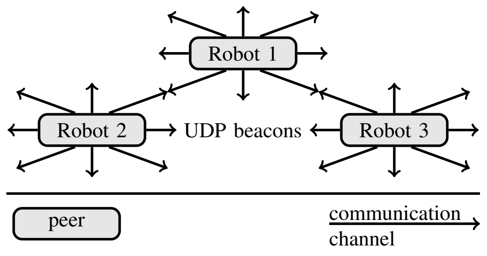


Fig. 7. Conceptual drawing of discovery using UDP beacons. Each robots broadcasts UDP packets containing its inbox endpoint, which is a TCP socket, the other can connect to. These beacons bootstrap connections between all robot peers, which could be very dynamic (coming and going) and creates a full mesh between peers.

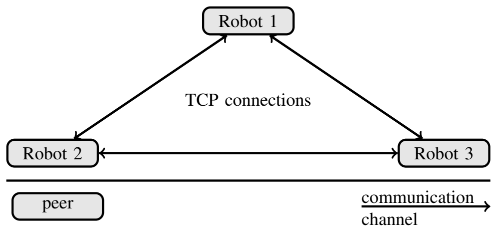


Fig. 8. Conceptual drawing of true peers connectivity. After each robot has received TCP endpoints from its peers, it opens a dealer-router connection to them, effectively creating a mesh.

Figures 7 and 8 demonstrate the UDP beacons and mesh creation to reach true peer to peer connectivity.

Robotics use case

Bootstrapping a multi-robot application is a common problem especially if multiple development teams are involved and need to sync configuration information. Local discovery and the true peer pattern alleviate application developers from this issue.

An important robotic use case are search-and-rescue missions. Due to unreliable communication, robots can lose connection and re-connect or new robots arrive and need to be integrated into the existing group of peers.

(Mobile) Sensor Networks are another application for this pattern. Sensors run out of energy and switch themselves off or new sensors arrive. Or in some scenarios sensors switch themselves on and off to conserve energy. All this an be handled with this pattern.

4.4 Census

This pattern is used whenever one node in a network is monitoring or querying a set of other nodes to determine how to proceed. It is a *group request-reply* pattern, where replies are dealt with according to the use case. In multi-robot systems, a coordinator or mission planner, typically needs to query a set of robots for their capabilities and availability to establish how a mission can be executed.

Such a coordinator might know in advance how many robots are in the network, but these robots can come and go, and the goal of Census is to determine a sensible answer based on the replies from those robots that are able to contribute.

The pattern is achieved by a router socket at the coordinator side, i.e. the node sending the query, and dealer socket at the target nodes. It exploits the two-way many-to-one dialog of the router-dealer mechanism. Figure 9 depicts an example with one coordinator and three participants.

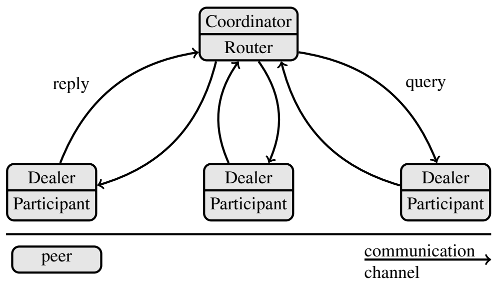


Fig. 9. Conceptual drawing of Census pattern. The coordinator node in the network queries a set of participant nodes to determine their availability, for instance, in a mission planning scenario. Each participant can answer with a reply, and a particular census policy in the coordinator will determine how many replies are required to proceed.

Robotics use case

This pattern fits perfectly in a multi-robot mission planning application. The planner queries a set of robots for their availability and capability and as soon as enough robots have replied, the mission can start.

4.5 Coordinated publish-subscribe

This pattern addresses the slow joiner problem, publish-subscribe suffers from. In a typical pub-sub setup, a publisher dumps data on its output channel without having to care about how many subscribers are listening. As a result, subscribers arriving late or requiring some time to setup their subscription, will not receive the first messages the publisher sends. This makes coordination difficult with this mechanism.

As a work-around, a separate request-reply channel can be setup between the publisher and each subscriber to which subscribers can send a *ready* signal, when they successfully setup their subscription. The publisher, on the other hand, will wait with outputting data until all subscribers have sent this signal. As a compromise, the publisher needs to know how many subscribers require its data.

Robotics use case

Whenever critical information is sent over a publish-subscribe network, this pattern should be considered.

4.6 Distributed logging

A final pattern is common in applications that need extensive logging of several nodes in a network. One use case is traceability, for example. It tackles the problem of combining information from many sources and reduces thereby the workload on the other nodes.

A passive solution would involve the definition of an *observer* and a shared communication channel, which picks up all chatter and creates a huge file based on that information. This creates two issues:

- How does the observer distinguish between valuable data to log and other data flow?
- How do we avoid that all nodes in the network receive all information that needs logging.

The solution is an *active* approach, where one node introduces itself as a *logger* when it joins the network and supplies and endpoint to which all other nodes that want to delegate their logging can connect. This endpoint could be added as a key-value pair in a header in a hello message when the logger enters.

The only thing other nodes need to do is check this header for a shared key defined loggers and connect to that endpoint. Again the router-dealer mechanism is well-suited for this type of connection.

Robotics use case

In a multi-robot application with limited communication bandwidth or robots with limited processing power, this pattern can be used to offload data to a more powerful logger node which can do data postprocessing. Setting up a separate connection to the logger limits interference to other peer robots on the network.

5 Discussion

In this discussion section, we provide some guidelines on how proposed patterns could be used, which trade-offs developers should make and we address composability and deployment.

Addressed issues

The patterns and mechanism, introduced in this paper, serve as examples how to solve various issues encountered in multi-robot systems deployed in the field. More concretely, following functionality is addressed:

- dealing with **uncertain** communication networks (Heartbeating (4.1), True Peers (4.3))
- dynamically (re)configuring **multiple** robots (Discovery & True Peers (4.3))
- **synchronising** multi-robot mission execution (Census (4.4))
- bootstrapping **initialisation** of coooperative robot applications (Discovery (4.3))
- **tracing** of data and intermediate (mission) states (Shared State (4.2, Distributed Logging (4.6))
- detecting failures or connection drop-outs (Heartbeating (4.1), Coordinated Pub-Sub (4.5))
- **(re)configuring** communication protocols and infrastructure on the fly (Discovery & True Peers (4.3))

Trade-offs

For many communication scenarios in robotic applications, there are many decisions and trade-offs during system design. Often there is not just one solution but the scenario can be solved in different ways and the best choice depends on the available communication infrastructure. Although, the pattern should not change, implementations might depend on available infrastructure: what network connections are available, is broadcasting enabled, which transports are possible (TCP, UDP, ...), is it a *multi-hop* network?

Some questions that help selecting patterns, mechanisms, and libraries to implement are:

- How much bandwidth is available?
- How reliable does communication have to be?
- Which communication patterns are most useful for my use case?
- Does data pose additional requirements for the communication?

The first two questions are at opposite ends of the design spectrum. TCP guarantees delivery but uses more bandwidth, while UDP is more suited for low bandwidth scenarios, but is unreliable. So if losing few data packages is affordable, and no delivery guarantee is necessary, UDP is preferable. If bandwidth varies, as it does in many search and rescue scenarios, patterns for discovery and re-establishing connections might be required. And if large data, such as point clouds or video streams, have to be communicated, reconfigurable communication is desirable, e.g. to lower publishing/pushing frequency or filter data based on the available bandwidth.

Composability

Similar to how the presented communication patterns are built upon the presented mechanisms, patterns themselves can be building blocks to solve more complex scenarios. For example, the collaborative robot task execution use case of the shared state pattern (4.2) could require a redistribution of sub-tasks if robots disconnect. In that case, this pattern would be composed with the heartbeating pattern from section 4.1, meaning that robots send heartbeats to the entity that manages the common state. If heartbeats of a robot fail to arrive for some time, this entity can redistribute its sub-task to another one.

Deployment

The described communication mechanisms and patterns occur on multiple levels of abstraction. Altough, in this paper, emphasis was on multi-robot communication over a network, the presented communication patterns are independent of how the robot system (of systems) is *deployed*, i.e. communication can occur

- between threads **within** one process,
- between processes on the **same machine**,
- between processes on **different machines** in the **same network**,
- between processes on **different machines** in **different networks**.

If communication is within a machine and does not go over the network, data should be shared and not sent to avoid overhead of serialisation and wire protocols. Discussing methods for efficient communication on the same machine [43] are beyond the scope of this paper. Some frameworks [40], [39] configure communication automatically with respect to how communicating entities are deployed.

6 Conclusion

This paper presented a list of communication mechanisms and patterns to solve typical communication issues encountered in multi-robot systems. It is argued that communication should deserve more attention in robot software design and should be modelled first and separated from the choice of communication middleware. In multiple robotics research projects, in which our lab is involved, these patterns have proved useful.

In the discussion section, a motivated list of the trade-offs, developers have to think about, has been provided, and these trade offs have been linked to the mechanisms and patterns presented before. Also, the “deployment context” of the software components has been addressed and reusability examples by composing patterns were presented.

In summary, the emphasis of this paper is to provide a larger but still manageable toolbox to robotics developers to cover many use cases for communication support in complex robotics systems.

References

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009, p. 5. 1, 2.3

[2] Object Management Group, "Data Distribution Service (DDS)," http://www.omgwiki.org/dds/, http://www.omg.org/technology/documents/formal/data_distribution.htm, [Online; accessed 31-August-2015]. 1, 2.1

[3] M. Lutz, D. Stampfer, A. Lotz, and C. Schlegel, "Service robot control architectures for flexible and robust real-world task execution: Best practices and patterns," in *Workshop Roboter-Kontrollarchitekturen*, 2014. 1

[4] "Tibco enterprise message service," http://www.tibco.be/products/automation/enterprise-messaging/enterprise-message-service, [Online; accessed 1-September-2015]. 2.1

[5] "Rabbitmq," http://www.rabbitmq.com/features.html, [Online; accessed 1-September-2015]. 2.1

[6] Apache Software Foundation, "Apache Qpid, messaging built on AMQP," http://qpid.apache.org/, [Online; accessed 1-September-2015]. 2.1

[7] ——, "Apache Kafka, a high-throughput distributed messaging system," http://kafka.apache.org, [Online; accessed 31-August-2015]. 2.1

[8] ——, "Apache Camel, a versatile open-source integration framework," http://camel.apache.org/, [Online; accessed 1-September-2015]. 2.1

[9] G. Hohpe and B. Woolf, *Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004. 2.1

[10] "Advanced message queuing protocol (AMQP)," http://www.amqp.org, [Online; accessed 1-September-2015]. 2.1

[11] Apache Software Foundation, "Apache Avro, a data serialization system," http://avro.apache.org, [Online; accessed 31-August-2015]. 2.1

[12] Object Computing, Inc (OCI), "Opendds," http://www.opendds.org/Article-Intro.html, [Online; accessed 31-August-2015]. 2.1

[13] ——, "ADAPTIVE Communication Environment (ACE)," http://www.theaceorb.com/product/aboutace.html, [Online; accessed 31-August-2015]. 2.1

[14] K.-J. Kwon, C.-B. Park, and H. Choi, "Ddss: A communication middleware based on the dds for mobile and pervasive systems," in *Advanced Communication Technology, 2008. ICACT 2008. 10th International Conference on*, vol. 2, Feb 2008, pp. 1364–1369. 2.1

[15] "Extensible messaging and presence protocol (XMPP): Core," https://tools.ietf.org/html/rfc6120, [Online; accessed 13-July-2015]. 2.1

[16] "Tech pages/IoT systems," http://wiki.xmpp.org/web/Tech_pages/IoT_systems, [Online; accessed 13-July-2015]. 2.1

[17] iMatix Corporation, "Zeromq," http://zeromq.org/, 2007-2014, [Online; accessed 1-September-2015]. 2.1

[18] ——, "Zeromq, the guide," http://zguide.zeromq.org/page:all, 2007-2014, [Online; accessed 31-August-2015]. 2.1

[19] "Documents associated with CORBA," http://www.omg.org/spec/CORBA/3.3/, [Online; accessed 13-July-2015]. 2.1

[20] "Common format and mime type for comma-separated values (CSV) files," https://www.ietf.org/rfc/rfc4180.txt, [Online; accessed 13-July-2015]. 2.2

[21] "Extensible markup language (XML) 1.0 (fifth edition)," http://www.w3.org/TR/REC-xml/, [Online; accessed 13-July-2015]. 2.2

[22] "The javascript object notation (JSON) data interchange format," https://tools.ietf.org/html/rfc7159, [Online; accessed 13-July-2015]. 2.2

[23] "Introducing JSON," http://json.org/, [Online; accessed 13-July-2015]. 2.2

[24] "JSON schema: core definitions and terminology," https://tools.ietf.org/html/draft-zyp-json-schema-04, [Online; accessed 13-July-2015]. 2.2

[25] "json-schema.org - the home of JSON schema," http://json-schema.org/, [Online; accessed 13-July-2015]. 2.2

[26] "The GeoJSON format specification," http://geojson.org/geojson-spec.html, [Online; accessed 13-July-2015]. 2.2

[27] "YAML: YAML Ain't Markup Language," http://yaml.org/, [Online; accessed 13-July-2015]. 2.2

[28] "Open data description language (OpenDDL)," http://openddl.org/, [Online; accessed 13-July-2015]. 2.2

[29] "Google protocol buffers," https://developers.google.com/protocol-buffers/docs/overview/, [Online; accessed 31-August-2015]. 2.2

[30] Apache Software Foundation, "Apache Thrift, scalable cross-language services implementation," http://thrift.apache.org/, [Online; accessed 31-August-2015]. 2.2

[31] "Messagepack)," http://msgpack.org/, [Online; accessed 31-August-2015]. 2.2

[32] A. Shakhimardanov, N. Hochgeschwender, M. Reckhaus, and G. K. Kraetzschmar, "Analysis of software connectors in robotics," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, Sept 2011, pp. 1030–1035. 2.3

[33] "ROS on DDS," http://design.ros2.org/articles/ros_on_dds.html, [Online; accessed 13-July-2015]. 2.3

[34] P. Soetens, "RTT: Real-Time Toolkit," http://www.orocos.org/rtt, [Online; accessed 13-July-2015]. 2.3

[35] H. Bruyninckx, P. Soetens, and B. Koninckx, "The real-time motion control core of the Orocos project," in *IEEE International Conference on Robotics and Automation*, 2003, pp. 2766–2771. 2.3

[36] P. Soetens and H. Bruyninckx, "Realtime hybrid task-based control for robots and machine tools," in *IEEE International Conference on Robotics and Automation*, 2005, pp. 260–265. 2.3

[37] P. Soetens, "A software framework for real-time and distributed robot and machine control," Ph.D. dissertation, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, May 2006, http://www.mech.kuleuven.be/dept/resources/docs/soetens.pdf. 2.3

[38] C. Schlegel, "Communication patterns as key towards component-based robotics," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, pp. 49–54, 2006. 2.3

[39] D. Calisi, A. Censi, L. Iocchi, and D. Nardi, "Design choices for modular and flexible robotic software development: the openrdk viewpoint," *Journal of Software Engineering for Robotics*, vol. 3, no. 1, pp. 13–27, 2012. 2.3, 5

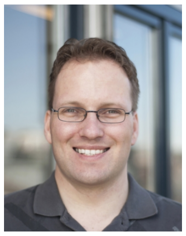
[40] J. Jackson, "Microsoft robotics studio: A technical introduction," *Robotics & Automation Magazine, IEEE*, vol. 14, no. 4, pp. 82–87, 2007. 2.3, 5

[41] iMatix Corporation, "Zmtp: Zeromq message transport protocol," http://rfc.zeromq.org/spec:37/ZMTP, 2015, [Online; accessed 31-August-2015]. 3.5

[42] B. Hayes-Roth, "A blackboard architecture for control," *Artificial Intelligence*, vol. 26, no. 3, pp. 251–0–321, 1985. 4.2

[43] K. Robbins and S. Robbins, *UNIX Systems Programming: Communication, Concurrency, and Threads*. Prentice Hall PTR, 2003. 5

Johan Philips received his Licentiaat in de Informatica (M. Sc. in Computer Science) in 2005, his Master of Artificial Intelligence in 2006, and the Ph. D. degree in Robotics in 2012, all from the University of Leuven. Currently, he is a postdoctoral research associate in the Department of Mechanical Engineering at the University of Leuven, conducting research on communication and coordination in multi-robot systems in half a dozen European research projects. His research interest covers robot software design, distributed systems and multi-agent coordination and control.



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 the Institute for Dynamics and Control, ETH Zurich, Switzerland. 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.

Herman Bruyninckx obtained the Masters degrees in Mathematics (Licentiate, 1984), Computer Science (Burgerlijk Ingenieur, 1987) and Mechatronics (1988), all from the KU Leuven, Belgium. In 1995 he obtained his Doctoral Degree in Engineering from the same university, with a thesis entitled “Kinematic Models for Robot Compliant Motion with Identification of Uncertainties.”

He is full-time Professor at the KU Leuven, and held visiting research positions at the Grasp Lab of the University of Pennsylvania, Philadelphia (1996), the Robotics Lab of Stanford University (1999), and the Kungl Tekniska Hogskolan, Stockholm (2002). Since 2014, he has a partime affiliation with the Eindhoven University of Technology.

His current research interests are on-line Bayesian estimation of model uncertainties in sensor-based robot tasks, kinematics and dynamics of robots and humans, and the software engineering of large-scale robot control systems. In 2001, he started the Free Software (”open source”) project Orocos (http://www.orocos.org), to support his research interests, and to facilitate their industrial exploitation.

He participated in about a dozen European research projects on robotics, with a focus in the recent years on the software engineering aspects. In October 2014, he received an honorary doctorate from the University of Southern Denmark, for his leading role in software development and research in robotics.

C Hierarchical Hypergraphs

This section contains a paper submitted to the special issue on Domain-Specific Languages and Models for Robotic Systems of the Journal of Software Engineering for Robotics. While the presented domain specific language was not used to explicitly model the PicknPack use case, the models presented in section 3 do confirm to the meta model introduced in the paper.

Journal of Software Engineering for Robotics X(X), XX 2015, 1-100
ISSN: 2035-3928

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

Enea Scioni,[1,2] Azamat Shakhimardanov,[1] Nico Hübel,[1] Markus Klotzbücher,[1]
Hugo Garcia,[1] Sebastian Blumenthal,[1,4] Herman Bruyninckx[1,3]

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

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 the research hypothesis that *hierarchical hypergraphs* are a better *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 can in itself be another hierarchical hypergraph. These properties are encoded formally in a *Domain Specific Language* (or “meta meta model”), called “*NPC4*”, built with *n̲ode*, *p̲ort*, *c̲onnector*, and *c̲ontainer* as primitives, and *c̲ontains* and *c̲onnects* as relationships. The formal model of *NPC4* is designed to maximally support its *composability* as a meta modelling language, for both the *structural* and *behavioural* parts of more concrete DSLs that can be built on top of it, each in a specific domain *context*.

NPC4 introduces a particular primitive, the *container*, to support *overlapping contexts*. It targets the following major targets in *knowledge-centric* robotics systems: (i) various *levels of abstraction* in domain models, (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 show up as (sometimes formal) model of concepts, knowledge, software, systems, and so on. 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-

• *This work was supported by the University of Leuven Geconcerteerde Onderzoeks-Acties* Model based intelligent robot systems *and* Global real-time optimal control of autonomous robots and mechatronic systems*, the University of Leuven IOF Kennisplatform* Transition*, and from the European Union’s 7th Framework Programme (FP7/2007–2013) projects* BRICS *(FP7-231940,* Best Practice in Robotics*),* ROSETTA *(FP7-230902,* Robot control for skilled execution of tasks in natural interaction with humans; based on autonomy, cumulative knowledge and learning*),* RoboHow.Cog *(FP7-288533,* Web-enabled and experience-based cognitive robots that learn complex everyday manipulation tasks*),* SHERPA *(FP7-600958,* Smart collaboration between Humans and ground-aErial Robots for imProving rescuing activities in Alpine environments*), and* PicknPack *(FP7-311987,* Flexible robotic systems for automated adaptive packaging of fresh and processed food products.

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 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* of these robotics sub-domains. And the objective of the paper is to formalize this insight into a pragmatically reusable *Domain Specific Model* for the structural properties of all aspects of robotics systems. The concrete examples are:

- *software architectures*, as in Figs. 1–2.

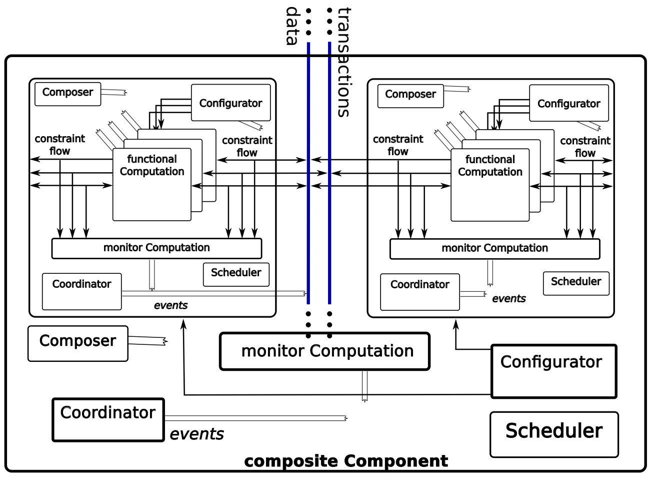


Figure 1. Structural model of a *composite component* software architecture [1]. *Nodes* represent software reponsabilities; *edges* represent data flow.

- *kinematics and dynamics of actuated mechanical structures*, as in Fig. 3.
- *Finite State Machines*, as in Fig. 4.
- *probabilistic graphical models* such as Bayesian Networks or Factor Graphs, Figs 5–6.
- *control diagrams* and other "data flow" computational models, such as the Cartesian position control scheme of Fig. 8.
- *knowledge representation networks*, such as the "semantic web".
- *web applications*, in which HTML5 [2] 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, or HTML5) as a "view" on a part of

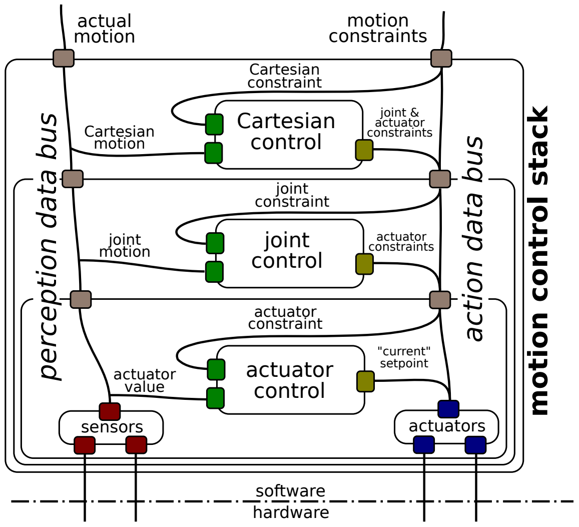


Figure 2. "Reference architecture" for a *motion control* stack of software components. The *nodes* represent control or input/output activities; the *edges* are lines connecting the *ports*, which give access to variables inside nodes.

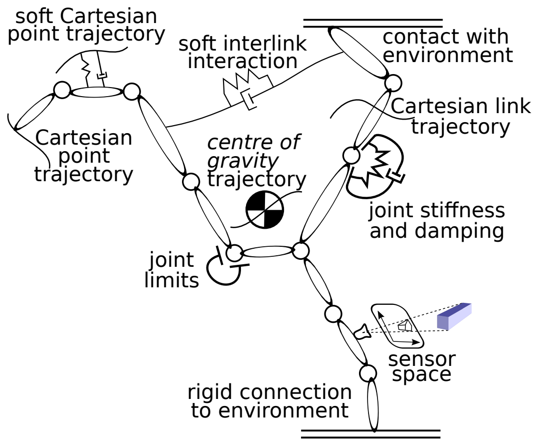

soft Cartesian point trajectory
soft interlink interaction
contact with environment
Cartesian link trajectory
Cartesian point trajectory
centre of gravity trajectory
joint stiffness and damping
joint limits
sensor space
rigid connection to environment

Figure 3. A generic tree-structured kinematic chain with possible task requirements on the chain's joints and links. *Nodes* represent actuated joints, rigid-body links, or a task's interaction dynamics (hard motion constraints , or soft "impedances"); *edges* represent (dynamics-less) connections between nodes.

the internal state of the node it is connected to, and (hence) 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 good formal representation to cover *all* the compositional structures discussed above, more particulary, via the *property* (*"has-a"*), *containment* and *connection* (*"interacts-with"*) primitives. (This formalized structure is a special case of a *mereotopology*, see [3] and references therein.)

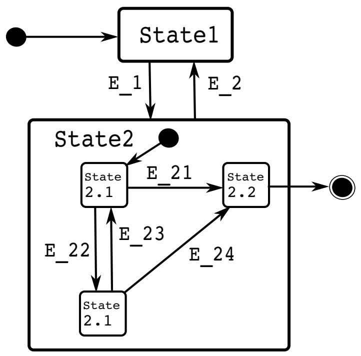


Figure 4. An hierarchical Finite State Machine. *Nodes* represent states, and *edges* represent state transitions.

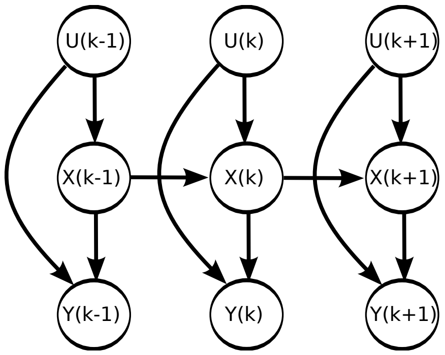


Figure 5. A simple dynamic Bayesian network, representing for example a *Kalman Filter*. The *nodes* contain the random variables in the network, and the *edges* represent probabilistic relationships between random variables.

Obviously, each application domain needs more than only a *structural* model; 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 often not *aware* of the extent to which their modelling languages and tools restrict their flexibility in modelling the designs of their systems.

Even robotics projects with a high software engineering focus[1] *do not have* explicit structural *models*, since they provide *only source code*. At best, "models" are only used as non-formalized 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 [4], or OpenRTM [5], [6]), and (ii) support hierarchical hypergraph models implicitly (for example, Matlab/Simulink or 20Sim, the ROCK toolchain for Orocos [7], [8], [9], [10], or the "plate notation" in probabilistic graphical models, Fig. 7).

1. Including popular open source projects such as ROS, Orocos, OpenCV, PointCloudLibrary, MoveIt, and so on.

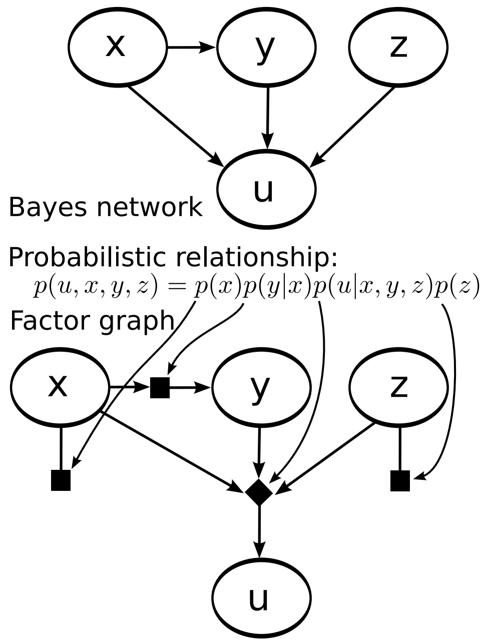

x
y
z
u
Bayes network
Probabilistic relationship:
$p(u,x,y,z) = p(x)p(y|x)p(u|x,y,z)p(z)$
Factor graph
x
y
z
u

Figure 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 not.

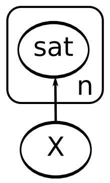


Figure 7. The so-called *plate notation* is one of the few examples where *hierarchy* is a *first-class citizen* of probabilistic graphical models. The plate is the rounded rectangle, and it represents n copies of the graph it contains, in this case, just one single random variable `sat` in the round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

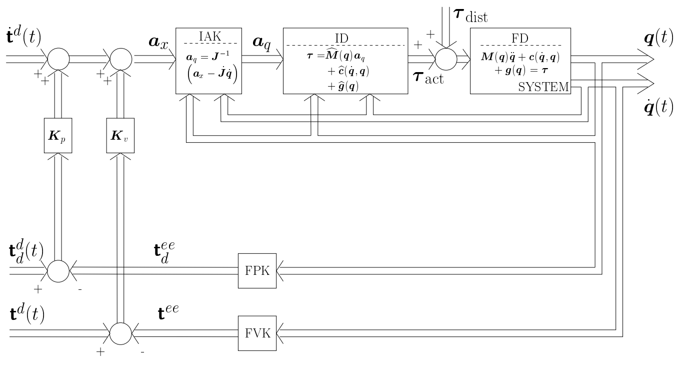

$\dot{\mathbf{t}}^d(t)$
$\boldsymbol{a}_x$
IAK
$\boldsymbol{a}_q = \boldsymbol{J}^{-1}(\boldsymbol{a}_x - \dot{\boldsymbol{J}}\dot{\boldsymbol{q}})$
$\boldsymbol{a}_q$
ID
$\boldsymbol{\tau} = \widehat{\boldsymbol{M}}(\boldsymbol{q})\boldsymbol{a}_q + \widehat{\boldsymbol{c}}(\dot{\boldsymbol{q}},\boldsymbol{q}) + \widehat{\boldsymbol{g}}(\boldsymbol{q})$
$\boldsymbol{\tau}_{\text{dist}}$
$\boldsymbol{\tau}_{\text{act}}$
FD
$\boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{c}(\dot{\boldsymbol{q}},\boldsymbol{q}) + \boldsymbol{g}(\boldsymbol{q}) = \boldsymbol{\tau}$
SYSTEM
$\boldsymbol{q}(t)$
$\dot{\boldsymbol{q}}(t)$
$\boldsymbol{K}_p$
$\boldsymbol{K}_v$
$\mathbf{t}^d_d(t)$
$\mathbf{t}^{ee}_d$
FPK
$\mathbf{t}^d(t)$
$\mathbf{t}^{ee}$
FVK

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; *Hierarchy* is typically not represented explicitly, but *is* present in many control schemes, via the implicit structural primitive of *cascaded control loops*, that is, an "outer" control loop around an "inner" loop.

None of those, however, support the full flexibilty that the hierarchical hypergraph DSL of this paper provides, 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 RDF or OWL based [11] semantic web approaches to knowledge representations [12] 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* as a first-class primitive in OWL or RDF, which, amongst other things, makes it hard to formalize knowledge about relationships or constraints.

Core idea and objectives. The aim of this paper is to provide better modelling flexibility and methodology to robot system developers, by introducing them to an hierarchical hypergraph *meta meta model*, The *NPC4* meta model represents the *structural* properties—*hierarchical hypergraphs*— of all the use cases introduced before, in a fully formal, computer-processable way, and with a clean *separation between structure and behaviour*. *Structure* models which subsystems interact with which *other* ones, and how their *internal* structure looks like; and *behaviour* models the "dynamics" of each of interconnected subsystems, and how the interconnections influence those subsystem dynamics.

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. See [13], [14], [15], [16], [17] for more details on DSL in general; some examples of DSLs in robotics are [18], [19], [20], [21], [22], [23].

This paper's **refutable research hypothesis** is that *NPC4* provides:

- the optimal *separation* between structure and behaviour;
- the optimal methodology of making 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 minimal set of language primitives and relationships that supports all DSLs.

In addition to the envisaged optimal *reuse of modelling concepts*, the systematic approach is also expected to create a step change in *reuse of software*:

- *reuse of syntactical parsing code*: the structure of a DSL is 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-functional software requirements to.
- *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 *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 `contains` relationship represents hierarchy, the `connects` relationship represents hyperedges. Section 3 presents constraints and properties over the NPC4 language, while Section 4 discusses about composability features of the language. Finally, Section 5 revisits some use cases introduced above in more detail, and explains how *NPC4* can be used as the basis for their structural models.

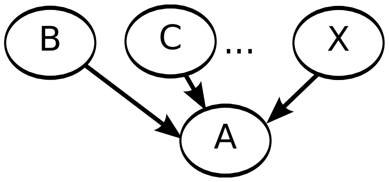


Figure 9. A simple Bayesian network in which the traditional graph structure mis-represents the probabilistic relationship between the random variables in the nodes: the network is a graphical representation of the n-ary probabilistic relationship $p(A|B, C, \ldots, X)$, while the arrows suggest only binary interactions of the form $p(A|B)$. In other words, the structural model is not a faithful representation of the system, and does not support *composability*. The *Factor Graph* of Fig. 6 provides a graphical model that is a better representation of the real n-ary probabilistic relationship.

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.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 by allowing to interconnect various *levels of abstraction* when modelling a system.

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.

Note that the kinematic chain example above is about modelling *behaviour*. This paper’s research hypothesis considers context and behaviour as the two *only* modelling aspects that require *hierarchy*; as the other parts of the paper’s research hypothesis, this one could be easily refuted by giving an example of a third necessary structural modelling aspect that requires hierarchy.

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. (This software architecture use case has parts that fall in the “context” as well as in the “behaviour” categories of hierarchical structure.)

Edges have no levels of abstraction, and just serve as topological symbols representing the immutable, 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 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.

Again, the recent ROS (and, to a lesser extent) Orocos practice (but also earlier practice in robotics such as [24]), 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 [25].

Another "bad practice" are *control diagrams*: the directed edges in, for example, *Simulink* [26] 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* [27], 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 [28], [29], [30], [31], 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*[2] models are (very slowly!) starting to be used [32], [33], [34], [35] 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.

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.

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 dynamics inside *multiple* `Nodes` by "connecting" them. Formally, a `Connector` realises an *hyperedge*, since the relationship is not *unary* but n*-ary*, and is bi-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.

2.3 Design drivers

The major design drivers to ground the *hierarchical hypergraph* concepts as a *Domain Specific Language* are *minimality*, *explicitness* and *composability*:

Minimality. The model represents only *interconnection and containment structure*. It serves as skeleton to represent the informations about the structural model, but it does not make any assumtion on the behaviour of such 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.

2. 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.

Primitive \ Primitive	Node	Port	Connector	Container
Node	contains	has-a	contains†	contains
Port	part-of	-	connects	is-contained*
Connector	is-connected (port)$^{+}$	connects	-	is-contained*
Container	contains	contains	contains	contains

Table 1
Overview of the primitives introduced by *NPC4* and the relative structural relationship allowed between them. The table reads has {primitive-row} {relationship} {primitive-column}, e.g. "a Node (can) contains a Node".
Note: (i) * it is not a relationship in *NPC4*, passive form; (ii) $^{+}$ it is not a formal relationship in *NPC4*, but informally a Connector is *indirectly* connected to a Node through a port; (iii) † as property of a *well-formed* Connector, see Sec. 3.1.

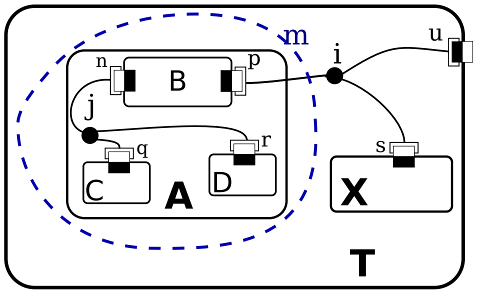


Figure 10. Generic example of a hierarchical hypergraph model. Node T is at the top of the hierarchy, and allows to refer to the whole model from within other models. Nodes A and X are contained by T, as is Container m; Nodes B, C and D are contained by A. Connectors i and j link Ports on Nodes. All 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.

- 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*.

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 meta model for hierarchical hypergraph*. It describes the *textual* formalization of the language, while

Node: node−A, node−B, node−C, node−A, 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. 10.

Fig. 11 shows the corresponding *graphical* conventions used in the paper. Table 1 provides an overview on the language core, and Table 2 illustrates the DSL by means of the concrete example of Fig. 10.

Identity is given to all primitives by simple declaration:

$$\texttt{Node}: \quad \texttt{node-B}, \texttt{node-X}, \ldots \tag{1}$$

$$\texttt{Port}: \quad \texttt{port-p}, \texttt{port-x}, \ldots \tag{2}$$

$$\texttt{Connector}: \quad \texttt{connct-i}, \texttt{connct-j}, \ldots \tag{3}$$

$$\texttt{Container}: \quad \texttt{cntnr-m}, \ldots \tag{4}$$

has-a: a relationship between a Node and a Port. A Port

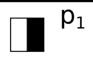
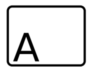
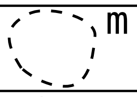
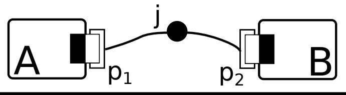
Primitive	Graphical Convention
Port	
Node	
Connector	
Container	
Connection Node-A Node-B	

Figure 11. 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 `Node`s, namely *A* and *B*, connected by the `Connector` *j* attached to the external docks of `Port`s p_1 and p_2.

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:

$$\texttt{has-a}(\texttt{node-B}, \texttt{port-p}), \qquad (5)$$

and statements of the following type *do not*:

$$\texttt{has-a}(\texttt{connct-i}, \texttt{port-p}), \texttt{has-a}(\texttt{cntnr-m}, \texttt{port-p}).$$

The inverse relationship `part-of` could be added to the model language, as syntactic sugar:[3]

$$\begin{aligned} & \texttt{part-of}(\texttt{port-p}, \texttt{node-B}) \\ \Leftrightarrow\ & \texttt{has-a}(\texttt{node-B}, \texttt{port-p}). \end{aligned} \qquad (6)$$

`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 *dot* (`.`) notation, that is:

$$\forall \texttt{P} \in \{\texttt{ports}\}, \exists! \texttt{P.edock}, \exists! \texttt{P.idock} \qquad (7)$$

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. 11 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.

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

$$\texttt{contains}(\texttt{M}, \texttt{X}), \qquad (8)$$

with `M` and `X` being a `Node` or a `Container`. The `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 example:

$$\begin{aligned} & \texttt{contains(container-m, node-A)}, \\ & \texttt{contains(node-A, node-B)} \\ \Rightarrow\ & \texttt{contains(container-m, node-B)}. \end{aligned} \qquad (9)$$

`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:

$$\begin{aligned} & \texttt{connects}(\texttt{connct-i}, \texttt{port-s.edock}), \\ & \texttt{connects}(\texttt{connct-i}, \texttt{port-u.idock}). \end{aligned} \qquad (10)$$

2.5 Composition

An extra keyword is introduced to indicate that all primitives *in NPC4* 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 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:

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".

- `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 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` [14]; 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`:

$$\texttt{compose} = \{\texttt{has-a}, \texttt{contains}, \texttt{connects}\}.$$

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*[4] 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,[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.

4. This motivation comes from the objective to make the formal models useful not just to human system delveopers, at design time, but also to *robots themselves*, at *runtime*!

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.

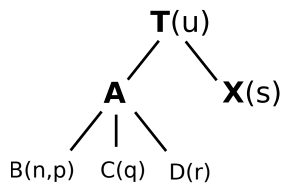


Figure 12. The containment tree of the nodes in Fig. 10. Each node carries its ports as arguments, since this information is required to check the well-formedness of `Connector`s.

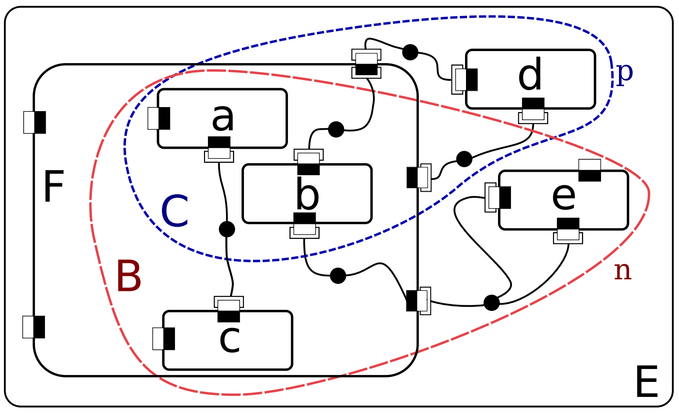


Figure 13. An example of an hierarchical composition in which *containment* does not follow a strict *tree* hierarchy *for containers*: the containers "p" (small blue dashes) and "n" (long red dashes) have some internal `Node`s in common, with each other and with `Node` "A"; the containers "p" and "n" do not have ports themselves, in contrast to the `Node` "A". The *nodes* and their *connectors* do satisfy the *node containment tree* constraint.

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:

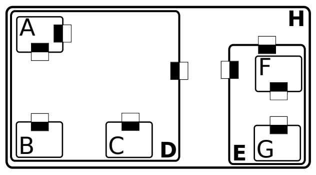
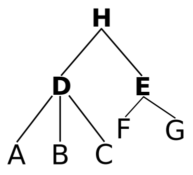
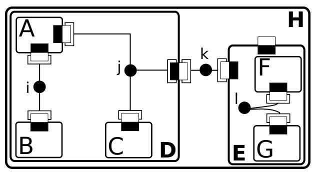
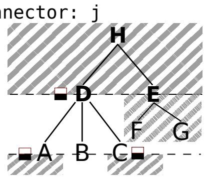
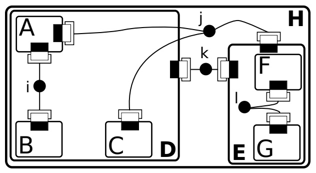
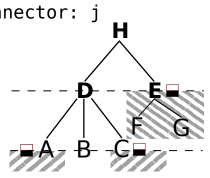
$$\forall \texttt{P} \in \{\texttt{ports}\}, \exists! \texttt{N} \in \{\texttt{nodes}\} : \texttt{has-a(N,P)}. \quad (11)$$

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 `Node`s must result in a *containment tree*.[6]

A `Node` can contain other `Node`s, 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

6. Strictly speaking, a *forest*, that is a collection of disjuncted trees.

n°	Hierarchical Hypergraph Layout	Node Containment Tree
1		
2		connector: j 
3		connector: j 

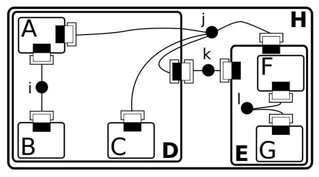
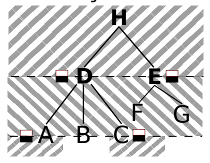
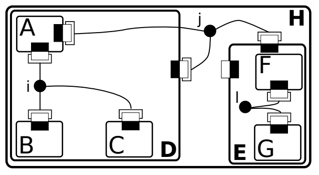
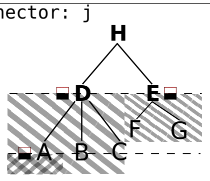
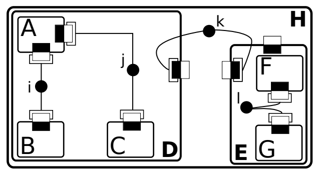
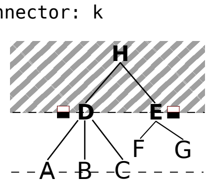
n°	Hierarchical Hypergraph Layout	Node Containment Tree
4		connector: j 
5		connector: j 
6		connector: k 

Figure 14. Different abstract examples of structural models defined with NPC4. Both graphical layout and relative containment tree have been reported. All the examples are based on the first model, which defines `contains` relationships only. The models differs on the `connects` relationship, and the containment tree is not affected by these changes. Examples (2) and (3) show well-formed models. In the associated containment tree, the `Connector` *j* is considered and the `Port`s involved in the relationship are indicated. In both cases, the resulting sub-tree obtained by pruning portions discriminated by the `Port`s is valid. The models (4), (5) and (6) are *ill-formed* because of the presence of a wrong `Connector`. In detail, in Model (4) the relation `connects(j,port-d.idock)` invalidates connections with `node-d` internals, thus the connection is not feasible. In example (5), `port-D.edock` excludes possible connections with `Node`s `A`,`B` and `C`; since a `Port` in `A` is connected, the `Connector j` is not correct. The latter case (6) shows an intuitive case of connecting two `Node`s through two wrong `dock`s (`Connector k`).

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. 12 visualizes the node containment tree of Fig. 10. 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. 13, 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 sake of clarity, Fig. 14 shows different model examples. The procedure to check this constraint is straightforward when starting from the `Node` containment tree: each of the 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. 14. 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`.

A *well-formed* `Connector` is contained in the *Lowest Common Ancestor* (LCA) of the `Nodes` involved.
Considering the example in Fig. 10, a statement of the following type is semantically correct:

$$\texttt{contains(node-A,connector-j)}, \qquad (12)$$

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 explicity 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`

$$\forall \texttt{C} \in \{\texttt{connectors}\}, \exists \texttt{N} \in \{\texttt{nodes}\} : \texttt{contains(N,C)}.$$

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

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. 13 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.

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 *bi-directional*. The direction is a property which belongs to the *behavioral* model, and not to the *structural* one: the constraint will be added in the specific domain of the *metamodel*. A typical case is a FSM metamodel, 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 semantic 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`, `connects` and `contains` relationships.

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

$$\begin{aligned}\forall \texttt{X}, \texttt{Y} \in \{&\texttt{Node}, \texttt{Port}, \texttt{Connector},\\ &\texttt{contains}, \texttt{connects}\},\\ \texttt{X.UID} = \texttt{Y.UID} &\Rightarrow \texttt{X} = \texttt{Y}.\end{aligned}$$

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

Constraints on `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:

$$\begin{aligned}&\forall \texttt{P} \in \{\texttt{Port}\}, \forall \texttt{N1}, \texttt{N2} \in \{\texttt{Node}\},\\ &\quad \texttt{has-a(N1,P)}, \texttt{has-a(N2,P)} \Rightarrow \texttt{N1} = \texttt{N2}.\end{aligned}$$

7. This context need not be *unique*, since others can be added by composition.

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:

$$\begin{aligned} & \forall \texttt{C1}, \texttt{C2} \in \{\texttt{connectors}\}, \forall \texttt{P} \in \{\texttt{ports}\}: \\ & \texttt{connects(C1,P.idock)}, \\ & \texttt{connects(C2,P.idock)} \\ \Rightarrow\ & \texttt{C1} = \texttt{C2} \end{aligned}$$

$$\begin{aligned} & \forall \texttt{C1}, \texttt{C2} \in \{\texttt{connectors}\}, \forall \texttt{P} \in \{\texttt{ports}\}: \\ & \texttt{connects(C1,P.edock)}, \\ & \texttt{connects(C2,P.edock)} \\ \Rightarrow\ & \texttt{C1} = \texttt{C2} \end{aligned}$$

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

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

$$p_{ci} \triangleq \{p \in \{\texttt{ports}\} \,|\, \texttt{connects(C,}p\texttt{.idock)}\}$$
$$p_{ce} \triangleq \{p \in \{\texttt{ports}\} \,|\, \texttt{connects(C,}p\texttt{.edock)}\}$$

- then, $\forall p_i \in p_{ci}$, $N_i \in \{\texttt{nodes}\}$ s.t. `has-a`(N_p, p_i) holds, $\forall p_j \in \{p_{ci}\} - p_i$, `C Connector` is valid if `contains`$(N_p, p_j)$ holds, and the following condition holds
- $\forall p_e \in p_{ce}$, $N_i \in \{\texttt{nodes}\}$ s.t. `has-a`(N_p, p_e) holds, $\forall p_j \in \{p_{ce}\} - p_e$, `C Connector` is valid if `contains`$(N_p, p_j)$ 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:

$$\begin{aligned} & \texttt{composite} = \{\texttt{node,port,connector,composite}\}, \\ & \quad \forall \texttt{C} \in \{\texttt{composite}\}: \\ & \quad\quad \exists \texttt{n} \in \{\texttt{Node}\} \vee \exists \texttt{c} \in \{\texttt{Connector}\} \\ & \quad\quad \vee \exists \texttt{d} \in \{\texttt{composite}\}: \\ & \quad\quad\quad \texttt{contains(C,n)} \vee \texttt{contains(C,c)} \\ & \quad\quad\quad \vee \texttt{contains(C,d)}. \end{aligned}$$

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

$$\begin{aligned} & \forall \texttt{c} \in \{\texttt{contains}\}, \\ & \quad \exists \texttt{X}, \texttt{Y} \in \{\texttt{node,container}\}: \\ & \quad\quad \texttt{c(X,Y)}. \end{aligned}$$

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

$$\begin{aligned} & \forall \texttt{X} \in \{\texttt{node,connector,composite}\}, \\ & \quad \exists \texttt{T} \in \{\texttt{Node}\}: \texttt{contains(T,X)} \vee \texttt{T=X}. \end{aligned}$$

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 discuss 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 runtime), 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[8] 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 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. 15 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 *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 possibile, but the described structural models differs.

5 Examples

This Section gives some concrete examples about how the meta meta model language *NPC4* can be used to make DSLs (meta model languages) for the *structural* parts of systems in the various robotics sub-domains introduced in Sec. 1. The examples show the two complementary ways in which such *domain specific extensions* are made, in accordance with the "composability" design driver (Sec. 2.3) behind *NPC4*:

8. Or scope, namespace, domain, or whatever terminology has been used to represent this container concept.

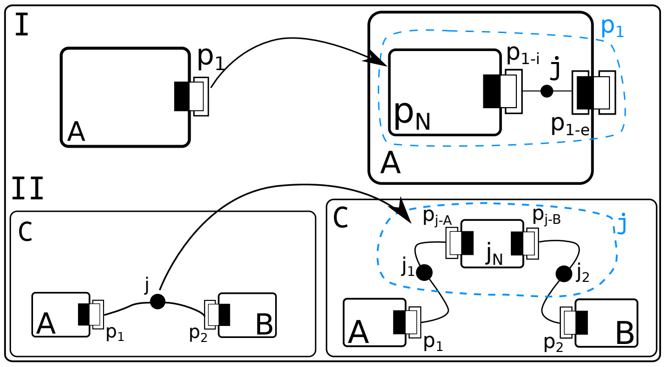


Figure 15. Examples of possible model-to-model transformation to describe a deeper level of abstraction. The first example (I-right) shows a solution to model behaviour on a `Port` primitive (I-left): `Port` p_1 is expanded in a `Node` p_N contained in `A` (original owner of p_1), while an internal `Connector` establishes a *view* over the same `Node`. The containment tree changes only internally to `A`, thus the change is compliant with the original model. Not necessarily but useful, p_1 refers to a `Container` in the transformated model, such that the original semantic information is preserved (and it is possible to retrieve the original model). The second example (`II`) shows a similar case, but considering the `Connector j` as target of the transformation.

- 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-related* structural part is explained, but not the concrete DSLs, containing structure and behaviour. However, some suggestions will be provided about the directions to take to realise full DSLs.

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* as a concrete illustration. Fig. 16 gives a graphical picture, Tables 3 and 5 give textual JSON [36] versions of the *domain*-centric model, without and with behaviour respectively, while Table 4 gives the *NPC4*-centric "full" version of it.

A LCSM are common software components to coordinate the "life cycle" of other software component instances, from

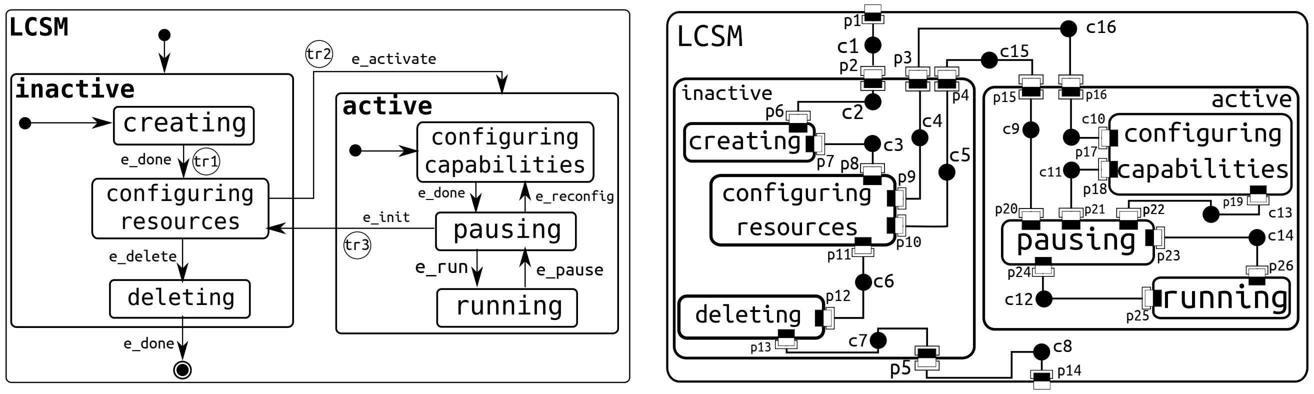


Figure 16. Life Cycle State Machine model (LCSM) of a generic software entity as an FSM model (UML flavor) on the left, and its Structural Model as Hierarchical Hypergraph on the right. `state`s are represented as `Node`s, while `transition`s as *Port-Connector-Port* patterns formed by two `connects` *NPC4* relationships. cX and pX are *instance_uid* of `Connector` and `Port`, respectively.

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 an LCSM. A real-world example of such an 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 occuring FSM model.

The whole FSMs structural part of an 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.

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

```
Node: lcsm, active, inactive

contains(lcsm, active)
contains(lcsm, inactive)
```

Table 3
A snippet of a possible DSL for FSM, hosted by a JSON document. Below, the same hierarchical structure described with *NPC4*.

Fig. 16 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. 16. Such `Ports` are connected only to one connector in the same hierarchical level. Since the port belongs to a `Node` (it represents an internal view of the `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

```
{
  "states" : [
    "active", "creating", "
        configuring_resources"
  ],
  "contains" : [
    { "parent" : "active",
      "children" : [ "creating",
        "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**(configuring_resources, p8)
connects(c3, p7.edock)
connects(c3, p8.edock)

Table 4

On top, a snippet of a FSM model taken from the LCSM example (see Fig. 16), with JSON support. The model which conforms to a FSM metamodel conform to *NPC4* meta-metamodel. 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.

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.
- the initial and terminal state primitives of FSM DSLS are just other cases of states, hence requiring nothing more than the addition of a new *named primitive* in the FSM DSL. The only difference is that the encompassing state must hold the function ("behaviour") that transitions to and from those states "at the right moment"; the latter behaviour is often a "semantic deviation point" between different FSM DSLs.

Since the FSM meta model conforms to [14] to *NPC4*'s hierarchical hypergraph meta-metamodel, the resulting concrete structure of the LCSM is reported in Fig. 16 (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.
- *inter-level transitions* have a similar *structure*, but maybe a different *behaviour*, in various FSM DSLs,

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

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. 16.

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. 16 (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 the minimal set of elements to describe the whole structure of a FSM DSL, simply by adding some constraints and domain specific names. The structure provides all the attachment points required to host all possible behavioural policies of various FSM dialects.

As illustrated in the following paragraphs, similar reasoning is possible in other domains, by simply adding extra structural constraints of the domain to the generic ones in *NPC4*.

5.2 Petri nets

While Finite State Machines are models to coordinate the different possible activities of *one single* "agent", Petri nets [37] are used to coordinate the activities of *several* agents. This is relevant in areas such as discrete process control and simulation, concurrent computing, or workflow management. Fig. 18 shows that the structural model of a Petri net is a bipartite graph consisting of places p, transitions t, and flows f. In addition, each place node has a number of so called tokens, and each flow has a weight (or multiplicity). In terms of *NPC4*, these are the *structural* parts of a Petri net nodel;

- there are two types of the `Node` primitive, namely *places* and *transitions*.
- the `Connector` primitive represent *flows* between nodes, with as structural constraints that (i) they are *directed*, (ii) they only connect *exactly two nodes*, and (iii) connected nodes must *come from a different class*, i.e., a connector is only allowed to connect places and transitions or transitions and places.
- that means that the model contains `Ports` that have a type, but that are not explicitly visible in mainstream graphical representations.

Other properties and constraints belong to the behavioural model of Petri nets:

- the *place nodes* `contain` a number of so called *tokens*.
- connectors have an `has-a` property that represents their multiplicity, which is used in representing behaviour.
- a transition can only be activated if all place nodes connected with incoming flows have at least an amount of tokens equal to the multiplicity of the connecting flow's multiplicity.
- if a transition is activated, all place nodes connected with outgoing flows receive an amount of tokens according to the multiplicity of their connecting flow's multiplicity.
- *implementations* of the models also require a behavioural policy for *when* a transition is activated.

Hierarchy can be added in various ways and even the tokens can *contain* Petri nets [38]; these can be modelled in *NPC4* by specifying these tokens as another class of nodes and hierarchically composing the Petri net from the states, transitions, and flows, where the states are a composition from their basic structure with a number of token nodes. Note that the tokens always are contained in the overall Petri net, while their `contains` property with respect to the place nodes changes at runtime.

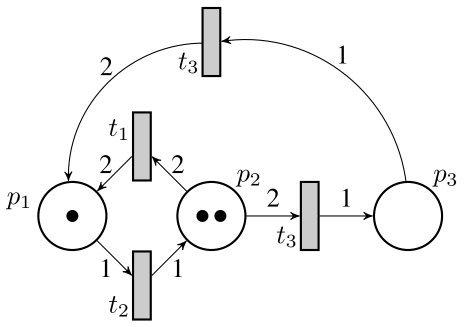


Figure 18. A basic Petri net with three place nodes and four transition nodes that are connected to a bipartite graph using directed connectors.

5.3 Bayesian Networks

The domain of *traditional* probabistic graph models, i.e., Bayesian networks, 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 making, or for scheduling of the computational execution.

Fig. 17 does show an example in the domain of probabilistic graphs where hierarchical `contains` is introduced explicitly. The sequence of model transformations, represents changes in the "level of abstraction" of the same probabilistic model, from its original Factor Graph representation to the final Junction Tree representation. The model transformations change only change the structure of the model, not its behaviour, and are introduced with the sole purpose to support the software infrastructure needed to compute the model's behaviour: the transformations end up with a *tree* structure that is better suited for iterative computations than the original *graph*. It is clear that all models at all these levels of abstraction are supported by the *NPC4* meta meta model.

5.4 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".

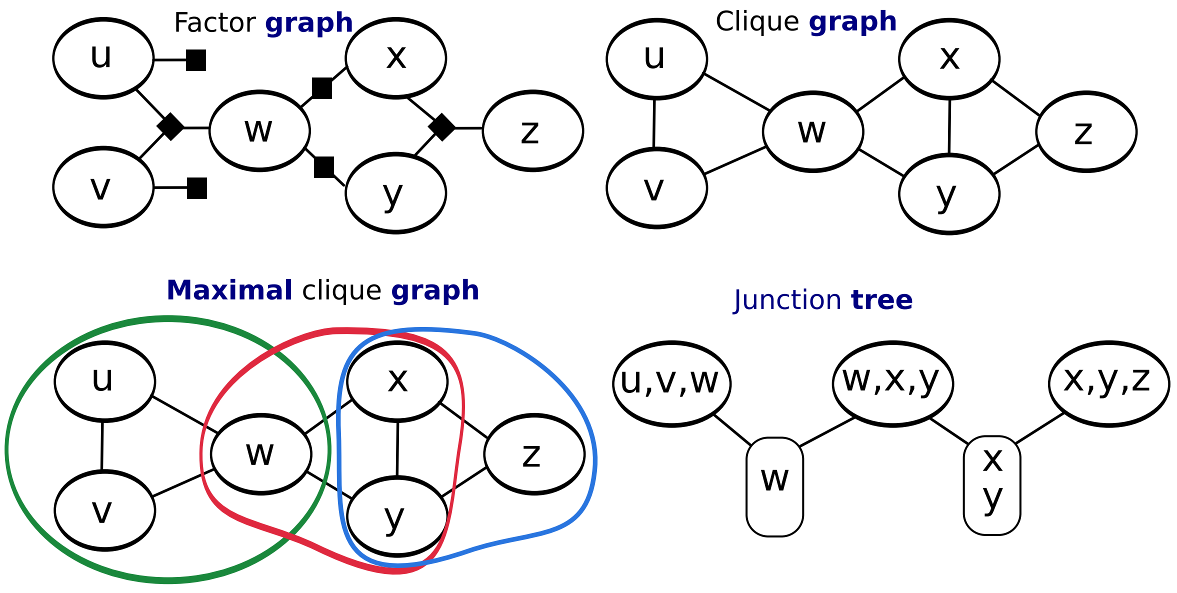


Figure 17. The same probabilistic graphical model represented at the different "levels of abstraction" that are being used in the traditional *junction tree* algorithm.

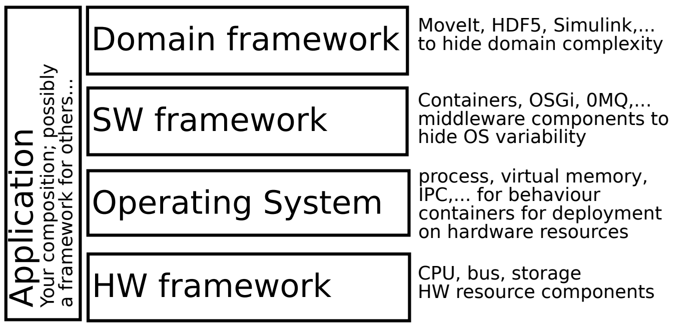

Application
Your composition; possibly a framework for others...
Domain framework
MoveIt, HDF5, Simulink,... to hide domain complexity
SW framework
Containers, OSGi, 0MQ,... middleware components to hide OS variability
Operating System
process, virtual memory, IPC,... for behaviour containers for deployment on hardware resources
HW framework
CPU, bus, storage HW resource components

Figure 19. The four natural levels of abstraction generally present in "software stacks", plus the concept of an "application" which composes them all together in one executable software system.

5.5 Software architecture models

The authors' recent publication [1] 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. 19.

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.

5.6 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, [39]; 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. 20:

- 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. (The JSON-LD format [40] has all primitives on board to represent such inter-DSL cross-linking.)
- similarly, each level *composes* its domain DSL, with `has-a` relationships, with a DSL that represents *physical units*, [41].
- *composition* into a *chain* can only work if the four other levels compose themselves with the *same* DSL on *geometric frames*, [20], 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).

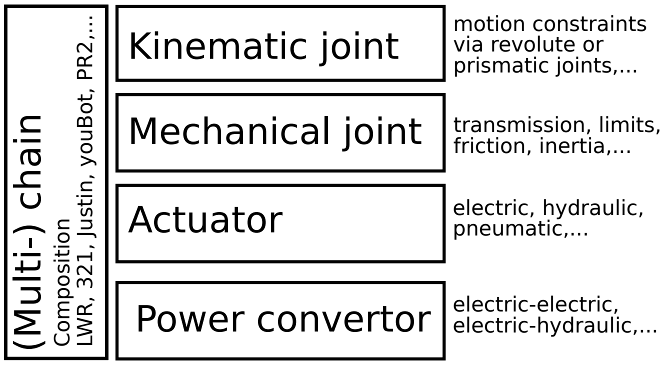

(Multi-) chain
Composition
LWR, 321, Justin, youBot, PR2,...
Kinematic joint
motion constraints via revolute or prismatic joints,...
Mechanical joint
transmission, limits, friction, inertia,...
Actuator
electric, hydraulic, pneumatic,...
Power convertor
electric-electric, electric-hydraulic,...

Figure 20. The four natural levels of abstraction 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”.

6 Conclusions

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

This was the big question that kept the authors busy for almost a decade, motivated by the drive to realise a step change in the reuse of “infrastucture code”. Indeed, lots of frameworks has seen the light in robotics the last ten 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 composition*, for both *interconnection of behaviour* and *containment of knowledge*, and in a domain-independent way. More in particular, the language targets *all* man-made engineering systems based on *lumped parameter* models.

Identifying a minimal set of *primitives* was not so difficult: nodes, ports and connectors (or semantically equivalent concepts) were in use everywhere, in one form or another. What proved *really* difficult was to identify the minimal set of *constraints* that govern all structural compositions: the lesson learned is that humans 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 *NPC4* are already covered, partially, in engineering languages such as Modelica [42], 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 effciency 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.

APPENDIX

This Section gives some more domain specific explanations for each of the graph structures that where listed in the Introduction.

- *software architectures*, as in Figs. 1–2. 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 [26] or Modelica [42]), 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 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. 4, 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 is used to abstract away the details of how the system reacts to one particular set of events.
- *probabilistic graphical models* such as Bayesian Networks or Factor Graphs, Figs 5–6. *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. For example in Fig. 5, only *some* of the input variables `U(k-1)` influence the output variables `Y(k-1)`.
  Hierarchy is used to model the *plate notation*, Fig. 7, or the reduction from a full graph network to a tree structure in the so-called *junction tree* algorithm, [43].
- *control diagrams* and other "data flow" computational models, such as the Cartesian position control scheme of Fig. 8; popular instances are *Simulink* [26] diagrams, or *Bond Graph* [28], [44], [29], [30], [31] models in *20Sim* [27]. 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, OWL or TopicMap languages) or the robotics *KnowRob* [45] (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. 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.
- *web applications*: the design behind HTML5 [2] 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* [46], or *HTML templates*; the *edges* represent *bi-directional data binding* supported by JavaScript as in AngularJS [47]; `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.

Life Cycle State Machine Example

For sake of completeness, Tab. 6 reports the full LCSM model described in the example of Sec. 5.1, which is conform to the

Node: lcsm, active, inactive, creating, deleting, configuring_resources, configuring_capabilities, pausing, running

Connector: c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15, c16

Port: p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12, p13, p14, p15, p16, p17, p18, p19, p20, p21, p22, p23, p24, p25, p26

has−a(lcsm, p1)
has−a(lcsm, p14)

has−a(inactive, {p2, p3, p4, p5})
has−a(active, {p15, p16})
has−a(creating, {p6, p7})
has−a(deleting, {p12, p3})
has−a(configuring_resources, {p8, p9, p10, p11})
has−a(configuring_capabilities, {p17, p18, p19})
has−a(running, {p25, p26})
has−a(pausing, {p20, p21, p22, p23})

contains(lcsm, {inactive,active})
contains(inactive, {creating, deleting, configuring_resources})
contains(active, {pausing, running, configuring_capabilities})

connects(c1, {p1.idock, p2.edock})
connects(c2, {p2.idock, p6.edock})
connects(c3, {p7.edock, p8.edock})
connects(c4, {p3.edock, p9.edock})
connects(c5, {p4.idock, p10.edock})
connects(c6, {p11.edock, p12.edock})
connects(c7, {p13.edock, p5.idock})
connects(c8, {p5.edock, p14.idock})
connects(c9, {p15.idock, p20.edock})
connects(c10, {p16.idock, p17.edock})
connects(c11, {p21.edock, p18.edock})
connects(c12, {p25.edock, p24.edock})
connects(c13, {p19.edock, p22.edock})
connects(c14, {p23.edock, p26.edock})
connects(c15, {p15.idock, p4.idock})
connects(c16, {p3.edock, p16.edock})

Table 8
Full *NPC4* model of the Life Cycle State Machine, described in Sec. 5. It describes the structure obtained automatically from the proposed FSM model, fully reported in Table 6. For compactness, only the first two *has-a* relationships are regular *NPC4* statements. The other statements has been defined as syntactic sugar over the regular *NPC4* relationships. That is, the relationship indicated is unrolled considering each primitive in the second argument list.

meta model FSM proposed in Tab. 7. The proposed FSM meta model exploit *NPC4* language as structural model, and Tab. 8 illustrates the concrete instance of the LCSM example.

References

[1] 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*, vol. 5, no. 1, pp. 17–35, 2014. 1, 5.5

[2] World Wide Web Consortium, “HTML5,” http://www.w3.org, last visited September 2014. 1, A

[3] S. Borgo and C. Masolo, “Full mereogeometries,” *The Review of Symbolic Logic*, vol. 3, no. 4, pp. 521–567, 2010. 1

[4] Groupe de Recherche en Robotique, “Proteus: Platform for RObotic modeling and Transformations for End-Users and Scientific communities,” http://www.anr-proteus.fr/. 1

[5] 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*, Venice, Italia, 2008, pp. 87–98. 1

[6] National Institute of Advanced Industrial Science and Technology, Intelligent Systems Research Institute, “OpenRTM-Aist,” http://www.openrtm.org, last visited August 2013. 1

[7] H. Bruyninckx, “Open robot control software: the OROCOS project,” in *Int. Conf. Robotics and Automation*, Seoul, Korea, 2001, pp. 2523–2528. 1

[8] H. Bruyninckx, P. Soetens, and B. Koninckx, “The real-time motion control core of the Orocos project,” in *Int. Conf. Robotics and Automation*, Taipeh, Taiwan, 2003, pp. 2766–2771. 1

[9] H. Bruyninckx and P. Soetens, “Open RObot COntrol Software (OROCOS),” http://www.orocos.org/, 2001, last visited March 2013. 1

[10] S. Joyeux, “ROCK: the RObot Construction Kit,” http://www.rock-robotics.org, 2010, last visited November 2013. 1

[11] W3C, “Owl,” http://www.w3.org/TR/owl-ref/. 1

[12] G. Antoniou and F. van Harmelen, *A Semantic Web Primer*, 2nd ed. MIT Press, 2008. 1

[13] C. Atkinson and T. Kühne, “Model-driven development: a metamodeling foundation,” *IEEE software*, vol. 20, no. 5, pp. 36–41, 2003. 1

[14] J. Bézivin, “On the unification power of models,” *Software and Systems Modeling*, vol. 4, no. 2, pp. 171–188, 2005. 1, 2.5, 5.1

[15] M. Fowler, *Domain Specific Languages*. Addison-Wesley Professional, 2010. 1

[16] J. Greenfield, K. Short, S. Cook, and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley and Sons, 2004. 1

[17] Object Management Group, “Meta Object Facility (MOF) core specification,” http://www.omg.org/technology/documents/formal/data_distribution.htm, 2006. 1

[18] H. Bruyninckx and J. De Schutter, “Specification of force-controlled actions in the “Task Frame Formalism”: A survey,” *IEEE Trans. Rob. Automation*, vol. 12, no. 5, pp. 581–589, 1996. 1

[19] 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*, Grenoble, France, 1997, pp. 836–841. 1

[20] 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.*, vol. 20, no. 1, pp. 84–93, 2013. 1, 5.6

[21] E. Gat, “ALFA: a language for programming reactive robotic control systems,” in *Int. Conf. Robotics and Automation*, Sacramento, CA, 1991, pp. 1116–1121. 1

[22] M. Klotzbücher and H. Bruyninckx, “Coordinating robotic tasks and systems with rFSM Statecharts,” *J. Softw. Eng. in Robotics*, vol. 3, no. 1, pp. 28–56, 2012. 1

[23] R. Simmons and D. Apfelbaum, “A task description language for robot control,” in *Proc. IEEE/RSJ Int. Conf. Int. Robots and Systems*, Vancouver, British Columbia, Canada, 1998, pp. 1931–1937. 1

[24] 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*, vol. 23, no. 12, pp. 759–776, 1997. 2.1

[25] P. Hintjens, “ØMQ—The guide,” http://zguide.zeromq.org, 2013, last visited July 2014. 2.1

[26] The MathWorks, “Simulation and model-based design by The MathWorks,” http://www.mathworks.com/products/simulink/. 2.1, A

```
{
  "metamodel" : "http://people.mech.kuleuven.be/~u0072295/fsm-dsl",
  "states" : [
    { "type" : "state", "id" : "lcsm" },
    { "type" : "state", "id" : "inactive" },
    { "type" : "state", "id" : "active" },
    { "type" : "state", "id" : "creating" },
    { "type" : "state", "id" : "configuring_resources" },
    { "type" : "state", "id" : "deleting" },
    { "type" : "state", "id" : "configuring_capabilities" },
    { "type" : "state", "id" : "pausing" },
    { "type" : "state", "id" : "running" },
    { "type" : "state", "id" : "exit-lcsm" }
  ],
  "contains" : [
    {
      "parent" : "lcsm",
      "children" : [ "inactive", "active", "exit-lcsm" ],
      "entry" : "inactive",
      "exit" : "exit-lcsm"
    },
    {
      "parent" : "inactive",
      "children" : [ "creating", "deleting", "config_resources" ],
      "entry" : "creating"
    },
    {
      "parent" : "active" ,
      "children" : [ "config_capabilities", "pausing", "running" ],
      "entry" : "config_capabilities"
    }
  ],
  "transitions" : [
    { "type" : "transition", "id" : "tr1", "src": "creating", "tgt": "configuring_resources" },
    { "type" : "transition", "id" : "tr2", "src": "configuring_resources", "tgt": "active" },
    { "type" : "transition", "id" : "tr3", "src": "pausing", "tgt" : "configuring_resources" },
    { "type" : "transition", "id" : "tr4", "src": "configuring_resources", "tgt": "deleting" },
    { "type" : "transition", "id" : "tr5", "src": "configuring_capabilities", "tgt" : "pausing" },
    { "type" : "transition", "id" : "tr6", "src": "pausing", "tgt" : "configuring_capabilities" },
    { "type" : "transition", "id" : "tr7", "src": "pausing", "tgt" : "run" },
    { "type" : "transition", "id" : "tr8" , "src": "run", "tgt" : "pausing" },
    { "type" : "transition", "id" : "tr9" , "src": "deleting", "tgt" : "exit-lcsm" },
  ],
  "events" : [
    { "type" : "event", "id": "e_done", "transition": [ "tr1", "tr5", "tr9" ] },
    { "type" : "event", "id": "e_delete", "transition": [ "tr4" ] },
    { "type" : "event", "id": "e_init", "transition": [ "tr3" ] },
    { "type" : "event", "id": "e_activate", "transition": [ "tr2" ] },
    { "type" : "event", "id": "e_reconfig", "transition": [ "tr6" ] },
    { "type" : "event", "id": "e_run", "transition": [ "tr7" ] },
    { "type" : "event", "id": "e_pause", "transition": [ "tr8" ] }
  ]
}
```

Table 6
Full model of the Life Cycle State Machine discussed in Sec. 5.1. The model is defined through an external DSL, defined in JSON-schema and defined in Table 7. For convenience, the model is described as JSON document too. The keywords “transitions”, “states” and “contains” represents the structural part of the model, while “events” (partially) define the behaviour. Many behavioural parts are not reported, such as “entry/exit” functionalities from a state and so on. It is not the aim of this work to discuss the validity of such DSL. However, it is author’s wish to report a concrete usage of the *NPC4* language to develop metamodels.

```
{
  "id": "http://people.mech.kuleuven.be/~u0072295/fsm-dsl",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type" : "object",
  "properties" : {
    "metamodel" : { "enum" : ["http://people.mech.kuleuven.be/~u0072295/fsm-dsl"] },
    "states" : {
      "type" : "array",
      "minItems" : 1,
      "uniqueItems" : true,
       "items" : { "$ref" : "#/definitions/state" }
    },
    "contains" : {
      "type" : "array",
      "uniqueItems" : true,
      "items" : { "$ref" : "#/definitions/container" }
    },
    "transitions" :  { "$ref" : "#/definitions/transitions" },
    "events" : {
      "type" : "array",
      "uniqueItems" : true,
      "items" : { "$ref" : "#/definitions/event" }
    }
  },
  "required" : [ "metamodel", "states", "transitions" ],
  "additionalProperties" : false,
  "definitions" : {
    "state" : {
      "type" : "object",
      "properties" : {
        "type" : { "enum" : ["state"] },
        "id"   : { "type" : "string" }
      },
      "additionalProperties" : false,
      "required" : [ "type", "id" ]
    },
    "transition" : {
      "type" : "object",
      "properties" : {
        "type" : { "enum" : ["transition"] },
        "id" : { "type" : "string" },
        "src" : { "type" : "string" },
        "tgt" : { "type" : "string" }
      },
      "required" : [ "type", "id", "tgt", "src" ],
      "additionalProperties" : false
    },
    "transitions" : {
      "type" : "array",
      "uniqueItems" : true,
      "items" : { "$ref" : "#/definitions/transition" }
    },
    "container" : {
      "type" : "object",
      "properties" : {
        "parent" : { "type" : "string" },
        "children" : { "type" : "array", "items" : { "type" : "string"}, "minItems" : 1, "uniqueItems" : true },
        "entry" : { "type" : "string" },
        "exit" : { "type" : "string" }
      },
      "additionalProperties" : false,
      "required" : [ "parent", "children", "entry" ]
    },
    "event"  : {
      "type" : "object",
      "properties" : {
        "type" : { "enum" : ["event"] },
        "id" : { "type" : "string" },
        "transitions" : { "type" : "array", "items" : { "type" : "string"}, "uniqueItems" : true, "minItems" : 1 }
      },
      "required" : [ "type", "id", "transitions" ],
      "additionalProperties" : false
    }
  }
}
```

Table 7
JSON-schema metamodel of the proposed FSM DSL in Sec. 5.1. The json-schema used to is the draft04. This metamodel conforms to the *NPC4* meta-metamodel.

[27] Controllab Products B.V., "20-sim," http://www.20sim.com/, accessed online 2 August 2013. 2.1, A
[28] R. R. Allen and S. Dubowsky, "Mechanisms as components of dynamic systems: A Bond Graph approach," *J. of Elec. Imag.*, pp. 104–111, 1977. 2.1, A
[29] P. Gawthrop and L. Smith, *Metamodelling: Bond Graphs and Dynamic Systems*. Prentice Hall, 1996. 2.1, A
[30] H. M. Paynter, *Analysis and design of engineering systems*. MIT Press, 1961. 2.1, A
[31] ——, "An epistemic prehistory of Bond Graphs," in *Bond Graphs for Engineers*, P. Breedveld and G. Dauphin-Tanguy, Eds., 1992. 2.1, A
[32] F. Francis Colas, J. Diard, and P. Bessière, "Common Bayesian models for common cognitive issues," *Acta Biotheoretica*, vol. 58, no. 2–3, pp. 191–216, 2010. 2.1
[33] 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.*, vol. 33, no. 12, pp. 2477–2491, 2011. 2.1
[34] J. F. Ferreira, M. Castelo-Branco, and J. Dias, "A hierarchical Bayesian framework for multimodal active perception," *Adaptive Behavior*, vol. 20, no. 3, pp. 172–190, 2012. 2.1
[35] 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*, ser. Lecture Notes in Computer Science. Springer, 2010, vol. 6314, pp. 424–437. 2.1
[36] D. Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)," http://tools.ietf.org/html/rfc4627, 2006. 5.1
[37] T. Murata, "Petri nets: properties, analysis and applications," *Proc. of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989. 5.2
[38] R. Valk, "Object petri nets," in *Lectures on Concurrency and Petri Nets*, ser. Lecture Notes in Computer Science, J. Desel, W. Reisig, and G. Rozenberg, Eds. Springer Berlin Heidelberg, 2004, vol. 3098, pp. 819–848. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-27755-2_23 5.2
[39] Willow Garage, "Universal Robot Description Format (URDF)," http://www.ros.org/wiki/urdf, 2009. 5.6
[40] M. Sporny, G. Longley, Dave Kellogg, M. Lanthaler, and N. Lindström, "A JSON-based serialization for Linked Data," http://www.w3.org/TR/json-ld/, 2014. 5.6
[41] H. Rijgersberg, M. F. J. van Assem, and J. L. Top, "Ontology of units of measure and related concepts," *Semantic Web*, vol. 4, no. 1, pp. 3–13, 2013. 5.6
[42] Modelica Association, "Modelica: Language design for multi-domain modeling," http://www.modelica.org/, last visited September 2014. 6, A
[43] S. L. Lauritzen and D. J. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems (with discussion)," *J. R. Statist. Soc. B*, vol. 50, no. 2, pp. 157–224, 1988, reprinted in [48, p. 415]. A
[44] F. T. Brown, *Engineering System Dynamics, a Unified Graph-Centered Approach*, 2nd ed. CRC Press, 2006. A
[45] M. Tenorth and M. Beetz, "KnowRob—A knowledge processing infrastructure for cognition-enabled robots," *Int. J. Robotics Research*, vol. 32, no. 5, pp. 566–590, 2013. A
[46] NN., "Web components," http://webcomponents.org, last visited September 2014. A
[47] Google Inc., "Angular JS," http://angularjs.org, last visited September 2014. A
[48] G. Shafer and J. Pearl, Eds., *Readings in Uncertain Reasoning*. San Mateo, CA: Morgan Kaufmann, 1990. 43

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 University of 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 concern on developing software middleware tools and Domain Specific Languages to realize advanced robotic applications.

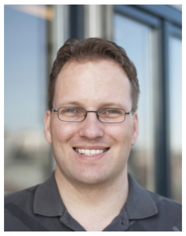
Herman Bruyninckx obtained the Masters degrees in Mathematics (Licentiate, 1984), Computer Science (Burgerlijk Ingenieur, 1987) and Mechatronics (1988), all from the KU Leuven, Belgium. In 1995 he obtained his Doctoral Degree in Engineering from the same university, with a thesis entitled "Kinematic Models for Robot Compliant Motion with Identification of Uncertainties."

He is full-time Professor at the KU Leuven, and held visiting research positions at the Grasp Lab of the University of Pennsylvania, Philadelphia (1996), the Robotics Lab of Stanford University (1999), and the Kungl Tekniska Hogskolan, Stockholm (2002). Since 2014, he has a partime affiliation with the Eindhoven University of Technology.

His current research interests are on-line Bayesian estimation of model uncertainties in sensor-based robot tasks, kinematics and dynamics of robots and humans, and the software engineering of large-scale robot control systems. In 2001, he started the Free Software ("open source") project Orocos (http://www.orocos.org), to support his research interests, and to facilitate their industrial exploitation.

He participated in about a dozen European research projects on robotics, with a focus in the recent years on the software engineering aspects. In October 2014, he received an honorary doctorate from the University of Southern Denmark, for his leading role in software development and research in robotics.

Azamat Shakhimardanov received his B. Sc. degree in control engineering from the Tashkent 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. In 2010 he started his Ph. D. degree in mechanical engineering at the 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.



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 the Institute for Dynamics and Control, ETH Zurich, Switzerland. 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.

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.

Sebastian Blumenthal received his B. Sc. and M. Sc. degrees in 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.