RobMoSys

H2020-ICT-732410

RobMoSys

Composable Models and Software for Robotics Systems

Deliverable D3.2:

First draft of software and tools for motion, perception and world-model stacks

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under grant agreement N732410.

Project acronym:	RobMoSys
Project full title:	Composable Models and Software for Robotics Systems
Work Package:	WP 3
Document number:	D3.2
Document title:	First draft of software and tools for motion, perception and world-model stacks
Version:	1.0
Delivery date:	30 June, 2019
Nature:	Report (R)
Dissemination level:	Public (PU)
Editor:	Enea Scioni (KUL)
Authors:	Enea Scioni (KUL), Herman Bruyninckx (KUL), Marco Frigerio (KUL), Nikolaous Tsiogkas (KUL), Filip Reniers (KUL), Matteo Morelli (CEA), Luz Maria Martinez Ramirez (TUM), Dennis Stampfer (HSU), Christian Schlegel (HSU)
Reviewer:	Alessandro di Fava (PAL)

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under grant agreement N°732410 RobMoSys.

Executive summary

The RobMoSys project has three complementary aims: (i) to use formal **models** as the basis for software, (ii) to use the structure provided by the models to improve the **composability** of software components, and (iii) to develop the **tools** to help developers write such software components and compose them together in systems. The aim is *not* to provide a software-only framework for robotics; but the solid foundations behind *any* such software framework.

This Deliverable reports on the status of the development of the core "platform" functionalities of *motion*, *perception* and their interactions via *world models*. At this phase of the project, the **methodology** of how to model these platform software functionalities in a way that achieves the above-mentioned goals has matured, via Work Package 2, whose major influence on the work represented here is:

- the identification of the importance of, and models for, the **horizontal** and **vertical** composition of software components;
- the identification of the importance of, and models for, (i) the **cause-effect chain** constraints in the execution of software components, (ii) the **life cycle state machines** of such components, and (iii) the **runtime (re)configuration** of software functionalities.

The developments of Work Package 3 **apply** these insights in a systematic way, to the developments of the robotics domain-specific "inside" of software components. One of the consequences is that this Deliverable introduces work on the **tools** that support **to model and to compose** the **fine-grained software functions and data structures** that make up the **platform** functionalities of robotic systems.

The following paragraphs summarize the WP3 developments around the above-mentioned WP2 concepts.

Vertical composition: robotics functionality comes at various *levels of abstraction* that must be put together, such as the composition of perception and control of *actuators*, *transmissions*, *joints*, *kinematic chains*, and *tasks*, but also the composition of the many choices on *representations* that have to be made (*mathematical*, *coordinates*, *digital*) and on *physical units*.

Horizontal composition: at any of the above-mentioned levels of abstraction, functionalities must be put together, and the focus of this Deliverable is on such integration between motion, perception and world modelling, for different *tasks* that a robot system must execute (possible concurrently), but also for the different *mechatronic resources* that the system is composed of.

Cause-effect chain: a robotic system comes with a lot of software functionalities deployed in its software architecture, but not all functions run all the time, independently of each other. Instead, the interactions of the robot system with its environment (including its peer robots) imply a high level of reactiveness, where specific software functionalities must be activated, at the right time and in the right order. Any somewhat realistic robot system must take into account several constraints that make this "scheduling" a non-trivial job, which needs methodological modelling and tooling in itself.

Life cycle state machine: the functionality of a software component is only made available to other software components when in the *Running* state. Of course, most of the domain-dependent functionalities have more than one *behaviour*, which results in a vertical composition of functional states into the *Running* state. For example, the robot's motion control can switch between velocity

control and impedance control; or its perception functionality switches between localisation and tracking.

Runtime configuration: not only the scheduling of the execution of the software functionalities must be determined at runtime, but also the choice of the many "magic numbers" that influence their specific behaviour; for example, gains in control loops, model hypotheses in perception, or geometric resolutions in world models. Reconfiguration becomes, in itself, a system level challenge, because the "magic numbers" in many software functionalities depend on each other, and these dependencies must be modelled, and their configuration supported by tooling.

The **impact** of the reported work is, first and foremost, focused on supporting the project's **Pilots**: those are the primary **dissemination** channel, because they allow (i) the Project to showcase system-level composability, vertically as well as horizontally, and supported by model-based tooling, and (ii) to transfer methodology and technology from the "academic" side of the project to the "industrial" side, but also to the "outside" world, represented by the partners in the recently started *Integrated Technical Projects*. The expectations are that this can bring a first round of "TRL boosting", such that the next Call for *Pilot Projects* can be realised on a more solid foundation.

The Pilots are also an excellent way to test the **level of standardization-readiness** of (a selection of) models of RobMoSys data structures and functionalities, and their composition. Because the Project has the ambition to solve, finally, the decades-old problem of robotics, namely the lack of standards for the essential platform-level functionalities of motion, perception and world modelling, formalized in models and supported by software reference implementations and tooling. This Deliverable suggests a concrete standardization, but for now these are just well-motivated and formalized suggestions, that must be confronted with the feedback and priorities of the wider robotics community.

Contents

1. Introduction

The RobMoSys project adopts model-driven development techniques to enable the development of better robotic systems, considering **composition** a necessary first-class primitive for modelling, tools realisation and software development. This means that not only models shall be composed together, defining yet another model, but also tools and the final software product must be composable, enabling **interoperability** and **re-usability**.

RobMoSys adopts the "Unix philosophy", that a tool shall do one thing only, but do it well. The same philosophy is adopted for the design of *models*, and, in the context of software functionalities, the design of the *functions* and *data structures* with which to realise these functionalities.

This approach towards *"minimality"* does not prevent the creation of *"monolithic"* tools or frameworks,[1] that are often expected in specific application domains because they bring the "user friendliness" of *de facto* "standardizations" to that domain. The drive for minimality is expected to help interoperability between such tools, because the development efforts of the common, re-used parts can be shared.

RobMoSys adds a fundamental extension to the Unix philosophy: the latter's *universal interface* boils down to typically not much more than text streams, while RobMoSys' universal interface (the so-called *data sheet* of a software component) can handle **models that conform to a set of meta-models**. In other words, interaction is not just via text, but via text with a formally modelled meaning (see D2.1, D2.2 and RobMoSys Ecosystem Organization[2]). The disadvantage of such "semantically rich" interfaces is the non-trivial effort to design and realise them (hence, one of the goals in RobMoSys), but the advantage is that tool interoperability is facilitated by requiring conformance to those meta-models.

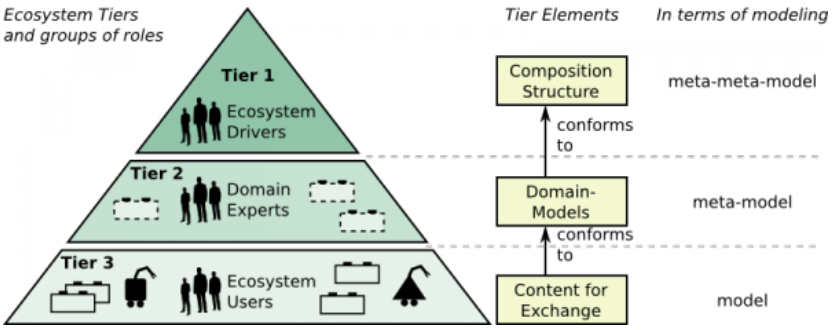


Figure 1.1: The three composition tiers of the RobMoSys Ecosystem [D2.3, D2.6]

Conformance to *Tier-1*[3] models is sufficient for tools that treat *context-free* (i.e., without the inter-dependencies that are essential to take domain knowledge on board) aspects.

For example, the concept of a component with ports is generic and can be found in different modeling languages. However, in order to achieve composition, domain-specific concepts need to be introduced, such as e.g. the definition of services (see Deliverables D2.1, D2.2 and D2.6), and

[1] The software of a robotic solution is already, *de-facto*, an integration of multiple technologies and software libraries. However, most of today's frameworks that claim to address composability enforces the developer to make strong technological bindings, adopting a static workflow with limited interoperability with other tools and approaches, or, worse, leaving the developer the responsibility to address those directly, by manual programming.

[2] https://robmosys.eu/wiki/general_principles:ecosystem:start

[3] see https://robmosys.eu/wiki/modeling:tier1

also how the configurations of these services depend on each other and on the context provided by the current task execution. This document goes a level deeper and addresses functional composition within components in a similar manner as component composition is addressed at system level. This Deliverable documents the design of the *semantic* information on the meaning of the data, their mathematical and numerical representation, their units and dimensions, their inter-dependencies, etc., via the so-called *Tier-2* meta-models of RobMoSys. For now, these domain-specific models describe the core *geometric* primitives and relationships, the *physical dynamics* of mechanical chains of joints and links, and the *information dynamics* of Bayesian information processing.

Robotic software tools and libraries are only truly composable if their functionalities are defined on the basis of the Tier-2 meta-models that represent the knowledge of the robotics domain. Examples of "true" composability are (i) to enable data exchange between software components only if the the semantics of the data is well-defined and consistent, and, (ii) all representation choices are made explicit (e.g., dimensions, units, mathematical representation, etc). Such a tool (see Sec. 2.5) generates *transformations* between those choices automatically, whenever it detects the need. For example, two components can exchange *orientation* values, even if one uses quaternions and the other (one of the many sets of) Euler angles.

Domain-specific knowledge shall be exploited not only for the elements that define a software component's *interface* (e.g., Communication-object meta-model, Service-Definition meta-model, Component-definition meta-model, etc), but also to build and to describe the **internals** of the component. The latter is a core focus for Work Package 3, whose major outcome are concrete models, tools and software to implement the component's functionality via the composition of *functions*, *data structures*, and *control flow*.

The (very) good news is that the vast majority of the "best practices" and "patterns" that hold at component interfacing level, also hold for the interfacing of functions and threads inside one single process. The only difference in design focus is that **realtime performance** must be configured in, without any change in the implementations of the functionalities.

The explicit description of such interaction entry/exit point of an algorithm, its constraints and configuration options, are necessary to ship a software component with a "*datasheet*" that provides all *usage* information and meaning to developers as well as to automated tools, without exposing the implementation internals of the component.

This Deliverable describes the first draft of the software for motion, perception and world model stacks (Sec. 3). The implementation of the concrete functionalities is complemented with the design and implementation of tooling that allows (i) to exploit Tier-2 models, (ii) to specify the composition of computation functions, and (iii) to bridge the gap between the Component-definition meta-model and the computational model of the functionality embedded in it, that is, solving the so-called **horizontal composition**. This focus on "tools first" (Chapter. 2) is motivated by the desire to have (i) an early release date of the first batch of tools, thus (ii) supporting the 2nd wave of ITPs and projects partners (in particular the industrial partners) in the Pilot cases; (iii) having early feedback, hopefully followed by an acceptance of the developed tools in the wider robotics community, and (iv) enabling co-development of more functionalities, models and tools. Finally, also the Pilots give focus (Sec. 4) to the software design of the tools and the choice of concrete implementations.

2. Principles and basic tooling

2.1 About flexibility, usability, optimality and composability

RobMoSys is about composable robotics systems by means of composable models and software. However, when it comes to a concrete *software implementation*, those principles and choices must be better shaped, since the concrete tools and implementations are limited, guaranteeing a certain degree of composability, within a well-defined scope, by means of a balance between flexibility, optimality and usability of the software solution and of the development workflow.

In this document, the term **flexibility** refers to the the capability of a robotics (software) system to adapt to different circumstances or contexts, during a well-defined phase of the development, also called **static** flexibility, or during a well-defined phase of the software component lifecycle, i.e., **dynamic** flexibility.

Optimality of a software solution usually refers to the capability of the tool (or a concrete implementation of an algorithm) to realise a valid solution with an optimal usage of both computational and memory resources. That is, the required resources shall be minimal, and well-balanced between memory and computational requirements. For example, caching mechanisms (e.g., *memoization*) should be employed in those cases where there is a better balance and concrete benefits due to multiple requests of the same information, relieving the computational costs[1] despite the memory usage (in addition to those cases where caching is a mandatory feature for the realisation of the concrete algorithm). Modern compilers already perform different types of *context-free* optimisation over the software implementation of certain algorithms: inline functions, data-flow optimisation (e.g., constant-folding, subexpression elimination), dead code elimination, and loop unrolling to mention a few. Within the scope of the RobMoSys project, the aim is to produce tools to aid the realisation of software implementations that exploits **context-dependent** optimisations, that is, taking into account how higher-order, domain-specific knowledge influences architectural choices.

Usability is only well-defined after the final *user* is identified with one of the RobMoSys roles. In this sense, a concrete software tool or implementation is *usable* if it allows and exposes, in a familiar form, all the necessary functionalities and configuration options that the user may need. Therefore, usability is not an absolute concept, i.e., a tool might be usable for a set of users, and not being usable for another set of users. However, nothing prevents the tool developer to provide different *views* over the same contents in the tool, such that the requirements of a different types of users are met.

Software **composability** is enabled by software design techniques that, starting from the conformity to well-defined meta-models such as the Tier-1 RobMoSys composition structures and the Tier-2 domain-specific models, allows tools synergies, (i.e., complementary tools) tool interoperability (i.e., alternative tools with respect to the functionality offered) and the possibility to use/integrate together multiple software products and implementations. However, software composability is not easy to realise, and pragmatically speaking, it is often bounded to a specific context or domain, in which the meta-models are defined.

During the development of a particular tool, the degree of composability is shaped already at design time, and the relative choices influence (and are influenced by) the final (desired) workflow

[1] Communication costs as well, in case of a local cache on the information consumer with respect to the producer.

about how *to use* the tool. For example, the composability of a code generator tool (that produces some code starting from existing models) comes from those models, allowing *static flexibility*. In fact, artifacts (i.e., generated code) are not composable by themselves. This implies a constraint in the workflow of the user, e.g., a change in the model triggers a refresh of the artifact. Nevertheless, artifacts composability is still possible if (i) the artifacts are models as well, (ii) the artifacts are used by a third tool that provides a certain degree of composability, e.g., by means of a well-established interface, or (iii) if there exists a model of the code-generator tool. For such a tool, the artifact would be *optimal* with respect to the domain-specific information encoded in the models.

Another tool could realise the same functionality, but aiming to *dynamic flexibility*, providing a runtime interface valid within a specific state of the software component lifecycle, to exchange models and to *interpret* them. The runtime overhead of such a solution would be higher, with different (but not necessarily inferior) opportunities of optimisation, since lazy evaluation tecniques can be adopted in some cases.

In the realisation of RobMoSys compliant software tools and relative implementations (within WP3), specific choices are made as a compromise of the elements above, with a concrete development workflow as a target. Those choices are unavoidable during the implementation phase, and this opens up to alternative implementations/tools that adhere to the same models, but with a different target.

The software reported in this deliverable, as well as further development and interactions are *independent* from the concrete tool realizations (e.g., SmartMDSD Toolchain and Papyrus4Robotics), yet adhering to Tier-1 and Tier-2 meta-models. However, special attention to integration issues with WP2 software baselines is given to guarantee the software composability that RobMoSys strive for. Our software implementation tries to cover as much as possible of the current Tier-2 meta-models discussed in D3.1, D3.3 (and further interactions), and explicit implementation choices are discussed where it is possible.

2.2 Horizontal Composition: Software Components as Data Integration Systems

In a component-based software solution, the role of each component is to expose a certain functionality with a dedicated, modelled interface to other components, processing the incoming data provided by other components. In that sense, a software component can be seen as a *Data Integration System* (DIS), a well-known concept in database literature (see Fig. 2.1). A DIS is defined as "an information system that integrates data of different independent data sources and provides the users with a uniform acess to the data by means of a global model" [4]. This definition holds for component-based software solutions as well, since every component acts as a DIS: (i) it collects data from some *source* components, (ii) it consumes the received data, holding an internal state and (iii) it serves the results to some *user* components, providing a well-defined *view* on the *consumer* internal state.

In the specific robotics domain, a typical *source* component "S" is a sensor driver, or any other component that provides *perception* functionalities. Examples of *user* components "U" can be either an User Interface that visualises runtime information about the robotics system, as well as "actuator" components, that realises the robotic motion starting from a computed control action. The *consumer* component "C" refers to the DIS used as reference that elaborates, stores and serves information provided by heterogeneous **producer**/*source* components to **consumer**/*user*

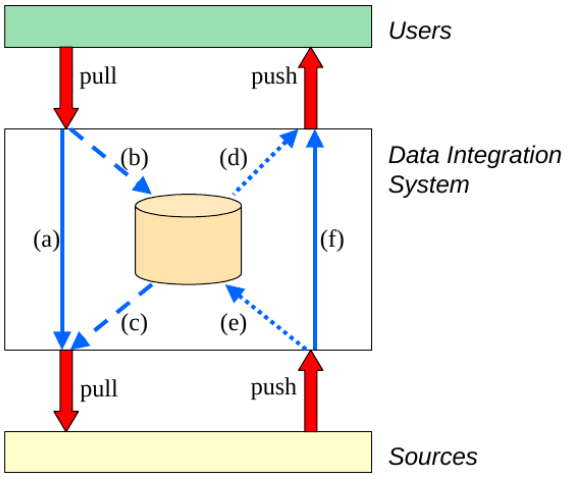


Figure 2.1: Component as Data Integration System (Figure from [4]). A component collects heterogeneous data from multiple sources, and it serves the data to the user components, with different views on the data itself, both synchronously or asynchronously, e.g., caching a result.

components. Obviously, the definition of *source* and *user* component is relative (with respect to the definition of the *consumer* component), and components are usually chained between each others. This established information flow allows to define a cause-effect chain, constraining the component execution order (scheduling of the component's activity) and other non-functional properties of the software architecture. The information flow also fits naturally to a **streams**-based interaction between components; but it fits also very well to the inside of a component, where "components" may be replaced by "threads" or "processes".

Any component contains some aspects of a DIS, however some are more "data-oriented" than others, that is, their main functionality resides on hosting heterogeneous data and serving to other components for different purposes. It is the typical case of a component that hosts *world model* information about the robotics application, or part of it. For those components where the data is dominant, the definition of **quality of service** based on the quality of the data provided is of fundamental relevance: how **fresh** is the data, and how (re)usable can it be served to the next computation step? In database literature [4], different metrics have been proposed, all as specific implementation of the concept of **data freshness**: currency, obsolescence, freshness rate and timeliness. Independently from the metric choice, the concept that *data freshness* is a quality of service is shown in the robotics domain as well. More concretely, with respect to the functionality served by the component `C`, the questions to be answered are:

- **caching on input**: in case the input data is cached, is it possible to (re-)use a cached input value for the execution of a query?[2]
- **caching on output**: if the (final or partial) result of a computation (i.e., answer to a query) is cached, that result is *fresh* enough to be used again?
- **computational time (delay)**: is the computation time short enough to make the result usable by the user?

[2] In this context, "solving a query" means computing a valid result.

In general, data freshness is influenced by caching mechanisms around the functionality embedded in the component. Typically, caching mechanisms are meant to reduce the computational time. Caching the computational state of an algorithm, or part of it, may prevent the execution of a new computation step (when inputs are invariant). Caching on input can prevent to pull (or push) data from the sources again, avoiding indirect costs (of resources and time), e.g., by triggering an activity on a source component, and relative costs to handle the communication.

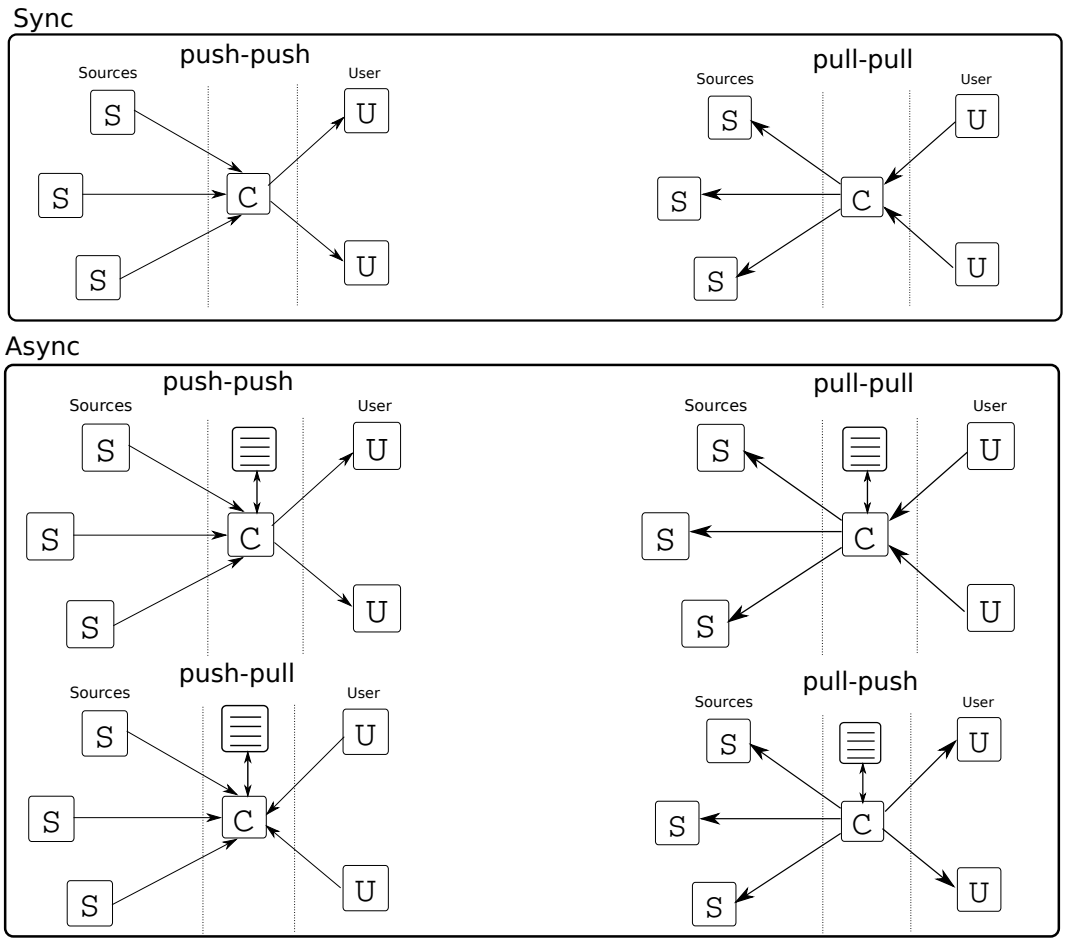


Figure 2.2: Communication patterns between heterogeneous *source* components (S), *user* components (U) and the consumer component C. Asynchronicity is related to the caching mechanism in the C component of inputs and/or outputs, in-memory or in persistent.

Considering a consumer component C as a reference, there are six possible compositions (see Figure 2.2), two synchronous and the other asynchronous. Asynchronicity is due to the presence of a caching mechanisms (input, output, or both) for which, for example, a result can be computed as soon as the inputs are available, and served later, in a second phase, when a pull request from the user component occur; it is the case of asynchronous push-pull pattern. Even if "cold", the data should be served "fresh enough" with respect the domain of the application, that involves both consumer and user components.

In short, data freshness is a quality of service that must be evaluated case by case, and it depends not only on the internals of the component under analysis (or synthesis), but also on the communication pattern in which the component C is employed.

There are multiple **policies** to handle data freshness at runtime; the two extremes are:

- informing about the quality of the result. The consumer component does not hold any decision whether to use a result or not. The data is computed at its best, and shipped with meta information on the quality of the data served. Example of useful data freshness information are related on the freshness of the "ingredients" that allow the computation of the result, thus related to the freshness on the input/output cache of the consumer component: (i) if the data is a result of a fresh computation; (ii) if the data is a result of a computation based on cached intermediate results; (iii) if the data is a result previously cached (and how old it is). Upon these information, the user component is responsible to decide whether to discard the incoming result, or to use it for the user's component purpose.
- The consumer holds the decision on the data freshness, and it only provides fresh data to the user, filtering out old data.

The choice on the policy may vary from case to case. For example, for real-time critical applications, it is preferred to always get a result, even if computed upon old cached values: waiting for an update might take too long or, even worse, with a non-deterministic update time.

Between the two policies reported above, many types of compromise are possible, and a composable design of the component internals is a key-enabler for this. Concrete entry/exit points of the computational model embedded in the component must be modelled, without side-effects, e.g., the functional behaviour is fully modelled, and the caching mechanisms are configurable. The proposed Model-based Function Composition Framework (MFCF) is designed to be such a technological key-enabler, but other solutions are possible as long as they adhere to the RobMoSys principles discussed so far.

A concrete use case regarding the development of world model facilities is discussed in Section 3.1.

2.3 Property Graph

2.3.1 Introduction

The property graph is a generic model for the representation of heterogeneous and linked data (and, hence, it is the core meta model for graph databases, which have known an exponential growth in the last decade). The nodes of the graph host the entities that one wishes to represent, while the edges model the relations between such entities. Both nodes and relations, in general, have a set of key–value properties representing the information encoded in the nodes and relations, and its meaning. The power of the property graph model lies in it being a very generic *mechanism*, as any object can be inserted in the graph and there are no constraints on the relations that can be established among them.

Such flexibility allows to model and interconnect heterogeneous objects, statically and dynamically, in ways that need not be pre-determined. Existing models, data objects, etc., can be *composed* together into a rich knowledge base. The information that can be modeled in the graph is not limited to traditional data-record-like items, and it includes, among other things, (symbolic references to) algorithms; it is then possible to relate, for example, geometric features with the available image processing algorithms for their detection. Also, composability implies de-composability, meaning that different subsets of data can be extracted from the graph to serve algorithms with different inputs; relations (edges) in the graph can always be ignored if not relevant, or exploited otherwise.

Given one ore more domains, it is up to the user to decide what is best represented by a node and what by an edge. A common conceptual operation that is easily performed on a property graph is the transformation of a relation between two entities into a new entity related with the other two, whenever the relation itself must take part into other connections. For example, a "measurement" relation between a quantity like a position vector and its numerical representation, can become a node when further information like the provenance or the coordinate system need to be explicitly represented in the graph.

A natural limitation of the flexibility of property graphs is that they require the user to be aware about the *existence* of the relations; in fact, the graph can be explored and searched for connections, but that comes at the price of higher complexity at the user side. Another issue with property graphs pertains to the traversal engines – the implementations allowing to navigate through the graph given some query – which are i) hard to realize and ii) generic like the graph itself, *i.e.* they cannot support directly some domain specific computations. To mitigate the issue and provide more specialized functionality, however, refined mechanisms based on the basic one may be constructed. For example, a DSL (Domain Specific Language) for queries about a specific domain (involving a subset of types of entities/relations) can be defined on top of the native query language for the graph, which exposes only the generic concepts of node, edge, property.

There exist some graph based databases or property graph engines, like Apache TinkerPop[3] and Neo4J[4]. However, these implementations are typically designed for large databases or to interact with existing data system providers. Therefore they do not match the requirements of multi-robot systems about footprint size and performance (latency etc.). Furthermore, it is not trivial to embed these existing databases within a software component with the current RobMoSys software baseline, due to technological incompatibilities; for example, different programming languages, deployment in a stand-alone (Java) virtual machine, and, especially, no focus on providing the

[3] See http://tinkerpop.apache.org/
[4] See https://neo4j.com/

property graph mechanism as a *library* that can be embedded inside the in-process RAM memory of a realtime motion stack component. The designs and the examples illustrated in this document thus refer to a custom, minimal implementation of the property graph concepts, for in-memory storage of data, suitable for usage in robotics applications, *e.g.* in the motion and perception stacks (see chapter 3).

Other related technologies in robotics, which also make use of an underlying graph data structure, include the URDF file format [5] and the tf component [6] from the ROS ecosystem. These solutions, however, almost completely lack composability and are hardly extensible. The URDF, for example, is a monolithic format supporting in *a* way a number of aspects of a robot model: if these aspects (like connectivity, geometry, inertia, attachment of frames/points, sensors, meshes for 3d visualization, etc.) are modeled separately, and corresponding tooling exists, it is much easier to support alternative representations for the same information (*e.g.* how to specify the relative pose of the frames on the model). Models addressing a self contained concern are composable, and composability is reflected in the tooling. Conversely, a monolithic format leads to monolithic tools and larger, harder specifications.

2.3.2 Property graph, Entity-Relation meta model (ER) and Block-Port-Connector meta model (BPC)

Any property graph *conforms to* a core meta model in RobMoSys, namely *Block-Port-Connector*:

- *Block*: every node in a property graph is a Block. Its "behaviour" is represented by its properties.
- *Port*: every Port on a Block is a *view* on a subset of the Block's properties.
- *Connector*: this is the relation that links two or more Ports, and whose own properties represent the meaning of the relation.

Moreover, the property graph serves the purpose of representing *entities* and *relations*, where their role is defined within the modelling domain. As a general modelling approach towards the definition of entities and relations in a specific domain, entities are those that have a meaning by themselves. Instead, relations need to have their "arguments" fullfilled to be well-formed. As illustrated in details in Section 2.3.3, in the geometric domain a relative pose is a geometric *relation* between two frames (*i.e. entities*). In a property graph, entities are always modelled as *nodes*; relations can be modelled as *edges*, but also as *nodes*. The latter is a design choice, which depends on the richness of details of the domain subject of the modelling effort. In case a relation is represented by a node in the graph, then it must have a very well defined set of edges, which must be connected to other nodes (entities): those nodes are the "arguments" of the relation. For a relative pose represented as node in the graph, two edges must connect to other two nodes representing the frames. However, those edges are not interchangeable, but they have a specific role (for a relative pose, one frame is target and the other is reference of the relation). To be able to define this, the concept of port must be introduced. This concludes that a property graph must follow the BPC meta model. Further details can be found in the Deliverable D3.3.

[5] See https://wiki.ros.org/urdf

```
{
  "Pose" : {
     "of"                : "frame-1-id",
     "with_respect_to" : "frame-2-id"
   }
}
```

Listing 2.1: Symbolic representation of Pose relation in JSON format, as described in D3.3 Annex, Ch.3

2.3.3 Embedding knowledge in a property graph: a use case

There is no unique way to embed knowledge in a property graph, and this depends on both the knowledge domain, the preferences and habits of the domain experts, and the context of the application. For example, within the same domain, there are certain details that are non-relevant for a specific application, and it is out-of-interest to represent (and store) those relations, constraints and properties. Nevertheless, one major advantage of the property graph approach is to be able to extend the domain and the concrete knowledge representation starting from an existing property graph, just **by composition** of the extra knowledge as new relations on already existing nodes; such an *extend-by-composition* approach is extremely RobMoSys compliant.

The aim of this section is to present the methodological approach to embed knowledge in a property graph. To this end, this section presents a running example, considering the *geometry* domain as a target of the knowledge representation. In particular, this example discusses how to represent a Pose, a geometry *relation* that involves two frames (*i.e.*, two geometric *entities*); a major use case of the Pose is to represent the relative position and orientation of two rigid bodies.

Recalling from D3.3 (cf. D3.3 annex, Chapter 3), a Pose is a geometric relation expressed between two frames. In details, this relation is ordered, that is, it expresses the concept of relative Pose of a frame with respect to another frame. Listing. 2.1 shows a JSON serialization of the Pose relation that expresses the relation *only* symbolically, without associating any concrete numerical value (*i.e.*, the *measurement* of the Pose relation). Moreover, this relation is a *composite* relation between two other relations: Position and Orientation. Hence, the extra semantics it adds is that of a *compositional constraint* between those two other relations.

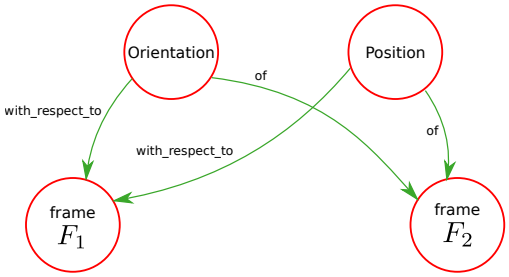
Let us consider the graphical representation of a property graph in Fig. 2.3. In this representation, both geometric entities and relations are represented as nodes of a property graph. To embed the concept of Pose, we assume that two nodes that represents the two frames (F_1 and F_2) which are the arguments of the relation are already instantiated:

1. if the concept of Orientation and Position between those two frames is already embedded by relatives nodes, (Fig. 2.3a), introducing the concept of Pose implies adding a new node in the graph. This is done by indicating that the Pose is a relation composed by the two existing relations (nodes), by means of an edge labeled composed-by.

2. if Orientation and Position nodes are not instantiated, creating a new node Pose will instantiate not one but three nodes, and relative edges to fullfil the definition of the semantics of Pose as a composite relation.

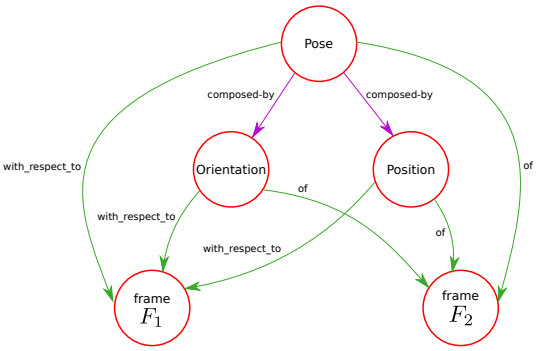
Independently of the initial state of the property graph, the result of adding a Pose in the property graph is the same, as shown in Fig. 2.3b: edges with_respect_to and of must be coherent with

previous knowledge, and the extra edges `composed-by` are instantiated.



(a) A property graph embedding the knowledge of `Orientation` and `Position`

.



(b) `Pose` relation is embedded by adding a new node and edges to express the composition of that relation and its arguments (the two frames F_1 and F_2.

Figure 2.3: Graphical representation of property graph to express `Pose`, `Orientation` and `Position` relations.

Adding measurements to symbolic relations

The property graph in Fig. 2.3 only embeds the knowledge of a `Pose` relation *symbolically*: no numerical values are attached to it, or rather, no *measurement* of the `Pose` is stored in the property graph.

To this end, other nodes in the property graph must be introduced. These nodes are of type *data block* (`dblx`), and their role is to represent and to store a concrete data block in-memory, by means of the *property* data structure of each of the nodes. From the implementation mechanism point of view, it means that a `dblx` node in the property graph represents (*"is a model of"*) the concrete data in-memory, and managing such a node involves operations outside the boundaries of the property graph, such as memory management (*e.g.*, memory allocation, garbage collection, etc) and serialization (*e.g.*, to send the data block over a network connection). The allocated memory serves the purpose of containing the numerical values of a measurement of the relation. The chosen data structure can have many forms, all with the same information meaning, depending on what is most appropriate for the algorithm/application that uses it. Moreover, more `dblx` nodes storing the same measurement can coexist, in order to support different data struc-

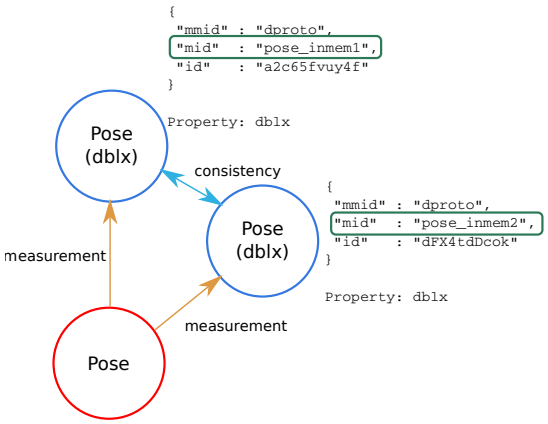


Figure 2.4: Two `dblx` nodes in a property graph to store a measurement of the same relation. The numerical representation choices differ between the two `dblx` (different `mid`, and an extra constraint `consistency` is added to indicate the data stored as property must be consistent.

ture choices, see Fig. 2.4. To represent all the
information needed to convert from one representation to another one with the same meaning, three symbolic identifiers are introduced: the `id` of a node; the *model-id* (`mid`) of the node, that is a reference to the model that fully describe the choices over the measurement value in the node with that `id`; and the *metamodel-id* (`mmid`), that determines which model (or DSL) the `mid` conforms to, and, hence, that has the information to support *model-to-model* transformations. Regardless of the representation choices of the measurement, these `dblx` have the same semantics. However, the measurement value must be kept consistent: a new (bidirectional) edge `consistency` is employed for modelling such a constraint. The underlying implementation that accesses the value must ensure that at every writing of a `dblx` (*e.g.*, a new measurement) all other `dblx` associated with a `consistency` constraint must be updated as well, in a consistent way. In addition, this operation must be *atomic* with respect to any other property graph manipulation that involves the nodes related to the measurement update. The model referenced by the `mid` shall indicate the various choices to be taken into account for the digital representation of the measurement, including (but not limited to) mathematical representation, abstract data type, data structure, or physical units. These modelling elements are illustrated in details in the Annex of D3.3, Chapter 3.

Finally, a DSL has been developed to describe and to associate the semantic value of the measurement (or numerical value of a geometric relation) with its representation choices: the *data prototype* (`dproto`) DSL. Together with the `dproto` DSL, a tool for automatic datatype conversion has been developed. This is further discussed in Sec. 2.3.5.

Independently of the concrete tool realisation, the presented solution based on the property graph concepts, allows to separate fully the description of symbolic "values" (relations) from their measurements, and it separates different concerns (such as mechanisms for memory management, datatype conversions, serialization, etc.). This increases composability with respect to alternative tools, mechanisms and DSLs that tackle each concern separately.

Fig. 2.5 shows the same property graph already described in Fig. 2.3, but with measurements for each relation (`Position`, `Orientation` and `Pose`). The `consistency` constraints are applied to the `dblx` nodes explicitly, instead of being derived implicitly from the knowledge of the relation `Pose` being `composed-by` `Orientation` and `Position`. The consequence is the same as the one discussed previously: on a new measurement (*i.e.*, an update on a `dblx` property), the properties of the constrained nodes must be updated as well. The role of a DSL and an automatic datatype conversion tool is important in this context: to guarantee this consistency, handling knowledge representation in a property graph means including functionalities dependent on the specific domain. For example, the property of the `dblx` `Pose` can be a 4×4 homogeneous transformation matrix, while the property of the `dblx` `Orientation` can be expressed in Quaternions (*i.e.*, 4 numerical values): the consistency can only be maintained if a conversion function is provided.

Finally, it is important to notice that, by means of this modelling effort, it is possible to evaluate possible overhead and computation load of operations over the values of the property graph. This enables the development of update policies that take into account specific requirements with respect to computational resources and real-time execution. For example, some `dblx` may be accessible by software components subject to real-time constraints, thus requiring higher priority on solving consistency with respect to other `dblx` that are less time-critical in the application.

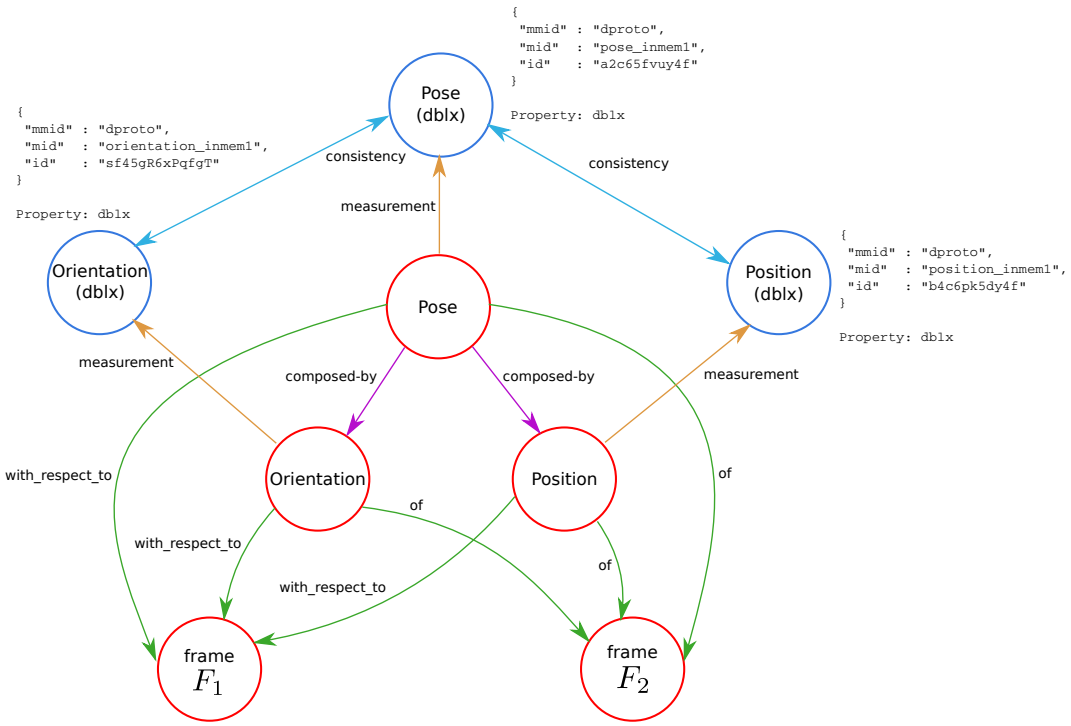


Figure 2.5: Property graph representing a `Pose` relation with measurements. The `consistency` edges model constraints when operating on the `dblx` nodes.

Extending `measurement` domain knowledge

The presented running example encodes some geometry entities and relations, where the "measurement" relation has a fundamental role to link symbolic information with concrete measurement values. So far, the measurement relation is modelled as a simple edge in the property graph. The modelling choice of treating the measurement as a first class citizen is good in those application contexts where no further information is required. However, there are more "properties" of a measure that an algorithm may exploit, to guarantee both functional and non-functional properties of the system. For example, more advanced strategies to guarantee `consistency` constraints can be developed, *e.g.*, to serve differently new incoming measurements (request of a property update), or requests to fetch a property value can be prioritised to realise different quality of service and real-time requirements. This section discusses an alternative modelling choice of the measurement relation that expands the previous domain. The same methodology of this running example can be employed in all other cases, where an edge in the property graph is promoted to become a node in itself, with its own property data structure.

In this extension, the addressed "properties" of the measurement are:

- the coordinate frame (`coord-frame`) from which the measurement takes place;
- `provenance` of the measurement, as a relation between the measurement and the nodes that models the providers of the data (a software component, a device, a user, etc);
- `used-by`, a relation that links the measurement and the "users" that have read access to the data, independently of the mechanism used to retreive the data (e.g., explicit query by polling; push notification; etc.);

- the relation with the `dblx` node that actually models the storage of the data (already modelled previously).

Figure 2.6 shows a graphical representation of this `measurement` extension. The original edge is converted into a node in the graph, still linked by a `measured-by` edge with a node `Position`, which represents a geometric relation. Functional data dependencies (*i.e.*, provenance and data users) are now modelled, linking the symbolic concept of Position, the concrete value instances (`dblx`) and inputs/outputs. At the time of writing, this measurement model is the one adopted in the world model stack, as discussed in Chapter 3.

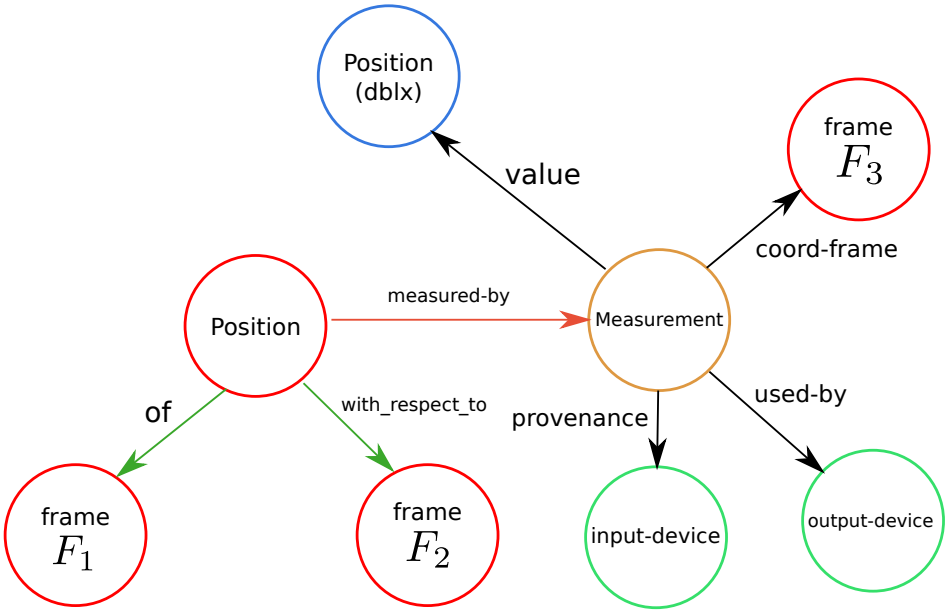


Figure 2.6: Extension of the geometric domain with measurement information (provenance, etc).

Properties and (time) series measurements

Another typical case of knowledge representation is to store properties as time-series, that is, as time-ordered sequences of measurements. Once again, there is no unique or best solution to this problem, as it depends on the concrete mechanisms of memory management that the property graph implementation supports, as well as on the requirements in the application. At the time of writing, we envision two different approaches (with their related implementations, that are under development):

1. at each request to update the measurement, the property of the `dblx` node is not modified but a new `dblx` node is created. This follows the *best practice* of *data immutability*: if all past relations with the `dblx` node are kept alive and are read-only, sharing data between activities becomes very composable. In practise, this strategy requires the implementation of a garbage collector at the property graph mechanism, to detect those nodes and relations that are not required by any "user" in the application.

2. the measurement sequence is still modelled with a single `dblx` node. The node property refers to a data structure (and its mechanism) to store sequences of data, and the memory management problem is delegated to such mechanism.

Summary

By means of a running example, this section explored the property graph approach applied to the geometric domain, which is a core-domain for any robotics applications. The current methodology to embed knowledge in a property graph was described, considering the following modelling choices:

- all nodes and edges have a `semantic-id`, with `id`, `mid` and `mmid` fields.
- nodes represents *entities*, and edges represent `relations`. Both are the "flat" mechanism of the property graph, which implies that "higher-order" meaning and domain-specific interpretation must be added by the application builders, explicitly.
- both *entities* and relations of the geometric domain are represented as `nodes` in the property graph, labelled edges assumes a semantic meaning only if the domain (spatial geometry) is specified.
- the `measurement` relation is a first fundamental building block, to embed knowledge of the property graph: it links symbolic representation, numerical values and non-functional properties, such as data provenance and data dependencies, with *functionalities* that require the data.
- the second fundamental relation is the `consistency` dependency between selected `dblx` nodes.

This property graph embodiment and its implementation are the fundation for the world model, motion and perception stacks, Chap. 3.

2.3.4 Tools and its implementation design

The previous sections describe the property graph and the methodology to embed knowledge on it. By doing so, it also collects a set of specifications that a reference implementation must consider.

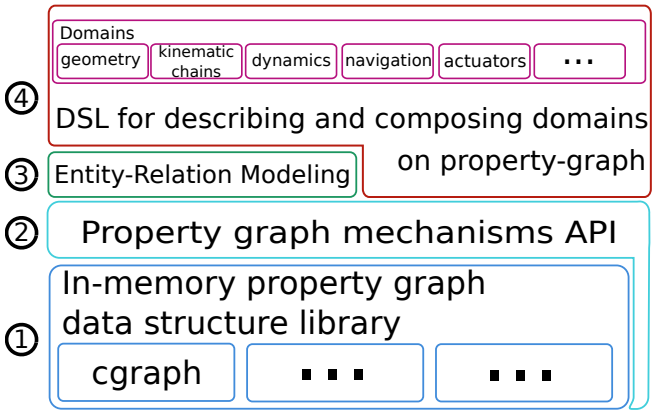


Figure 2.7: Current implementation design of the `property-graph` mechanism.

At M30 of the RobMoSys project, the implementation of the property graph is in alpha phase. This section quickly describes some design choices with reference to Figure 2.7, explained in the following:

1. Layer 4 refers to a Domain Specific Language for the definition of the primitive entities of various domains of interest, *e.g.* `Vector` in the geometry domain. These definitions would serve as the model/type for the entities in the graph, defining the bare minimum properties that must be represented. For example, a 3D vector must have an `origin` and an `end Point`; therefore, any node in the graph of type `Vector` will have at least two edges connecting it with two `Point` nodes.

2. Although a relation can always be promoted to an entity (*i.e.* an edge becomes a node) in order to further qualify it and to express higher order relations, any given domain defines which are the entities and which are the relations, in that domain. For example, in kinematics, the relative velocity between two rigid bodies is a relation, it cannot be defined without any of the two bodies. This layer allows to "view" the objects existing in the property graph in terms of Entities and Relationships for a given domain.

3. Layer 2 is the property-graph-specific C API to be exposed to clients. Establishing the API enables the use of different backend implementations in layer 1. The API shall reflect the main concepts of the property graph, by means of operations to construct, remove, connect nodes and edges, traverse edges, set and get key/value properties.

4. The bottom layer represents a wrapper of an existing, suitable, general-purpose graph library implementing the API described in point 2. Different implementations using different libraries can possibly address different use-cases, like real-time and non-realtime execution constraints. At the moment, our developments are based on the `cgraph` library.[6]

An additional module not shown in the figure is the query engine, which is in fact the primary client of the property graph API. The query engine (or query solver) must be able to interpret a user query (whose format also need to be defined) and perform the corresponding graph traversal required to fetch a desired node, property, or set of those. In fact, a query and therefore a traversal might also be used to trigger specific computations rather than only retrieving data. This is an important requirement to be able to make use of the *structure* of the information in the graph, for certain algorithms. For example, while traversing the robot model (encoded in the graph) from the leaf bodies to the base, the inertia tensor of each body can be composed to compute the composite inertia of the whole structure. Currently, an exploratory prototype query engine has been developed, drawing inspiration from the Gremlin language of Apache TinkerPop, cited before in this document (*e.g.* Section 2.3.1).

2.3.5 Storing values in data blocks and its semantics

Independently from the domain, any property graph must handle some properties that are *numerical* values, often *measurement* of some symbolic relations. These values must be stored in-memory, following the memory layout of a given *data structure*. Those "particular" nodes in a property-graph are very relevant, because they are subject of queries. In fact, the stored values are used by (numerical) algorithms of heterogeneous nature to perform computations for perception and motion control. Therefore, different requirements on the memory layout may occur on the same data, in order to serve better (and optimise) each single algorithm. As a consequence, multiple data blocks (`dblx`) containing the same semantic value (same measurement) but with

[6] See for example https://graphviz.gitlab.io/_pages/pdf/cgraph.pdf and https://www.mankier.com/3/cgraph.

different choices in the data structure layout (but also different choices as units of measurement, etc.) can co-exist. This implies a set of other technical problems, mostly regarding memory management and keeping consistency among the `dblx` containing the same measurement.

To this end, a DSL to annotate each `dblx` with its semantic information and multiple choices has been developed. In this DSL, each `dblx` conforms to a `dproto`, which is the model of that specific `dblx`. This allows to perform automatic data conversion between different data structure layouts. More details on the role of `dproto` and `dblx` can be found in Section 2.4.2 and in the Annex (Chapter 5).

2.4 The Model-based Function Composition Framework (MFCF)

RobMoSys promotes a Model-Driven Engineering (MDE) approach in robotics, starting from the Tier-1 (meta-)models (i.e., the RobMoSys composition structures) and the realisation of software baselines as development tools conformant to these meta-models. These software baselines aid the full development cycle of the component-based software of a robotic application: the creation of a component model (which conforms to the Component-definition meta-model), composing software elements such as services (cf. service-definition meta-model and communication-pattern meta-model), the definition of the component lifecycle and its activities, the component deployment, safety properties and, most important, the concrete functionality implemented within a component.

However, embedding one or more functionalities in a component is still a challenging tasks. For example, for simple domain-specific cases, the "user-code" embedded in a component can be generated, but often only partially, leaving to the component supplier the burder of writing some "glue-code" to connect the generated code to the middleware employed as a backend of the software baseline. In general, code generation is not an option, due to the heterogeneity of the domain and the lack of tools. Therefore, embedding functionalities is still an art, with few generic guidelines (often ditacted by the concrete software baseline), without a methodology or tool that support the component supplier. Functionalities implemented by a functional developer are often shipped in libraries, hard to integrate due differences in the API design, the domain, the programming language, the programming paradigm, etc. The component supplier must find the most appropriate solution to integrate such diverse libraries, dealing with conversion between datatypes, memory management and, sometimes, even bindings from one programming language to another.

Once the component is crafted, the component supplier should provide a documentation for the system builder. This "datasheet" of the component should contain everything about its interface, and this is already provided by the concrete component model. However, there is a lack of formalism to document the implementation of the functionality, especially when the finalization is manually realised. Moreover, without a formal model of the internals, it is hard to predict certain non-functional constraints, or to apply new ones. For example, the reaction of the component upon reception of a new input value at one of its port, should be clear without looking into the details of the implementation: will another output be available? or will the component wait for further inputs on other ports before proceeding? By pulling a value on a component's port, that value will be a new one, or a cached value? All of these, and more, can be answered only knowing what the component is actually running internally.

A formal description of the computation will allow the implementation of generic mechanisms to embed the algorithm into activities in a systematic way, for example, enabling to *store* the computational state of the internal algorithm. This is not trivial to be realised without a formal model of the software implementation, but still it is a valuable in many occasions. For example, it would be possible to restore the computational state later in the application, *e.g.* after recovery from a failure; or to get exactly the same initial conditions to each run of a planning algorithm; or for testing and debugging purposes of the library/functionality itself in a real scenario.

All the elements presented above are some of the motivations for a formal model and an implementation mechanism to describe and execute heterogeneous functionalities embedded within a software component. This section introduces the *Model-based Function Composition Framework* (MFCF), a software tool that aids both the function developer and the component supplier on these challenges; it aims to be the skeleton to which the implementation of functions can be

attached, composed and finally shipped within a component.

MFCF proposes a domain-specific language, MFCF-DSL, to describe the composition of a computational algorithm, and its underlying runtime mechanism for the execution. The main features are:

- *interoperability* of heterogeneous implementations, considering a *multi-language* paradigm;
- a formal language to describe the *computational model* of an algorithm, with
- explicit computational state and scheduling policies;
- minimality and usability as design principles, without the need to sacrifice ...
- ... the completeness of formal modelling of each function and data used in the algorithm, with particular attention to enrich the concrete implementations with semantics;
- any primitive of the language has its unique `id` and a reference to its model and meta-model, enabling composability and high-order level of reasoning about the computational structure;
- ready for both offline workflow (*e.g.* injecting generated code in a component) and online workflow, providing high flexibility at runtime;
- a step forward to a formal definition of a "component datasheet".

Another result of this work is a few guidelines that allows composability between (function) implementations, sometimes with the cost of do not exploit the full set of features that the employed programming language provides.

2.4.1 MFCF's User Workflow

As any other tool, MFCF is designed with a target user in mind: the function developer and the component supplier . Figure 2.8 depicts the envisioned workflow from the point of view of the component supplier , when building a new component around existing functionalities, developed by the function developer . The workflow is presented in a linear form, that is, from the creation of a stand-alone functionality to its embedded version within a software component, including the integration of other existing functionalities modelled within the MFCF.

2.4.2 MFCF DSL

This section briefly introduces the MFCF domain specific language (DSL) for functional composition. The purpose is not to provide a complete overview of the language, but to illustrate the main concepts and the features it provides. The language was implemented as an *external* DSL, to be compact and not verbose, to increase *usability*, to be independent from the general purpose programming language or toolchain employed to implement the runtime. However, it is possible to serialise a MFCF script/model to another host language, such as JSON or XML. This feature is foreseen in the development plan, since it allows to share computational models among software components, enabling technological compatibility with the existing RobMoSys baselines and dynamic reconfigurability of the deployed functionalities. Since the grammar and syntax of the language may be subject to major changes, the formalisation of the grammar elements is not reported. Major changes are possible not only about the syntax, but also to better support **composability**, with respect to the current state. Therefore, this document is meant to be purely

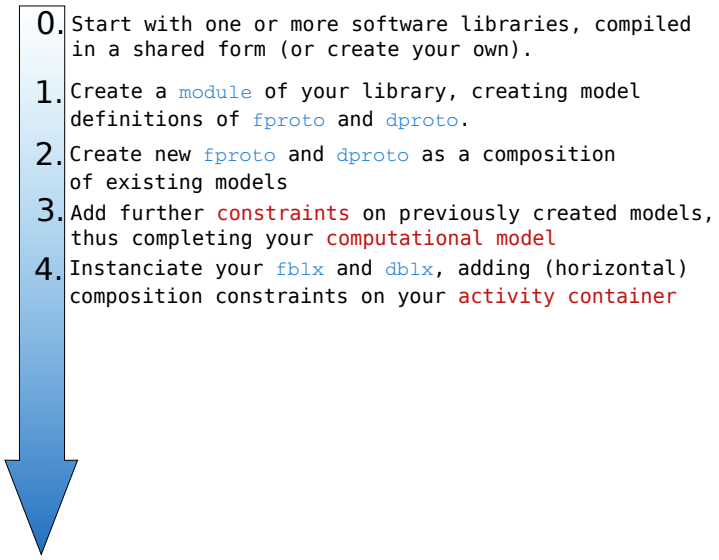


Figure 2.8: The envisioned MFCF workflow from the component builder perspective.

informative, and it shows snippets of the language illustrating the expressiveness with respect to the modelling concepts.

The core primitives of this language, inspired by functional programming, are two:

- **function blocks** (`fblx`) and its model called "function block prototype" (`fproto`): a function block abstract the concept of a function, independently from its implementation;
- **data blocks** (`dblx`): a data block represent a portion of allocated memory, having a specific data structure represented by its model, *i.e.* the "data prototype" (`dproto`).

Function Block (`fblx`) and Function Block Prototype (`fproto`)

Conceptually, a function block prototype (`fproto`) is a model of a *pure function*, an abstraction of an implemented functionality with a input/output model, subject to the constraint of being *stateless*. The property of having no observable *side-effects* promotes composability (and re-usability) of the function, since it enforces the modelling of all function's dependencies, explicitly. Reproducibility of the result is guaranteed, regardless of the internal implementation of the function. Additional data dependency constraints can be applied on the arguments of a `fproto` model, by imposing *causality* constraints that can prevent direct read/write access to the data.

A function block (`fblx`) is a concrete *instance* of a `fproto`, to which is possible to impose further constraints, both between the accessed data and other `fblx`(s). A `fblx` can be executed/invoked by a runtime, in contrast with the `fproto` which is the model of the executable function. The above elements are first class primitives to describe a *computational model*, where the constraints assume a primary role in the scheduling execution of each `fblx`, that is, the order in which the functions are executed must satisfy the imposed constraints.

In MFCF, a function block prototype is expressed as a model of a function prototype, and its implementation is provided by an external library. The model reflects and enhances (in some cases) the information contained in the function declaration, by imposing **causality constraints on the arguments**, *i.e.* whether they are treated as inputs, outputs or both.

```
fproto frame_compose :: c99 {
  library = "kdlwrap",
  fname   = "kdl_frame_compose",
  args    = {
    1 <= s_plain_pose,
    3 => s_plain_pose,
    2 <= s_plain_pose
  }
}
```

Listing 2.2: An example of a `fproto` definition for a composition operation between two poses. The model refers to a function `"kdl_frame_compose"` embedded in the library `"kdlwrap"`. All arguments are of type `s_plain_pose` (which is a reference to a `dproto`), where arguments 1 and 2 are inputs of the function, argument 3 is output.

Listing 2.2 shows an example of `fproto` declaration, which may change depending on the language used to implement the function: to this end, a **context** is indicated. In this example the context is c99, indicating the function has been implemented in ANSI C99. The remaining annotations that define a `fproto` change depending on the context; for c99 the following are specified:

- `library`: indicates the name of the (shared) library in which the implementation of the function is provided;
- `fname`: the name, the function symbol used in the function declaration, as its unique identifier in the indicated library[7];
- `args`: an (unordered) list of argument definitions, each one with (i) an unique, numerical identifier for the position of the argument in the function prototype; the identifiers are sequential numbers starting from 1, since the function return value, if any, is identified with 0, and it must be an output argument; (ii) a model of the datatype, *i.e.*, a data block prototype (`dproto`), as explained in the next section; (iii) the constraint on the data dependency, per argument, that indicates the causality, *i.e.*, the input/output relationship of the argument. The following syntax determines the role and the type of access of the argument: "=>" if it is an output of the function, "<=" if it is an input.

In other programming languages, such as C++, other annotations may be necessary, such as the `class` name and the `namespace` of the function (member).

The structure of the `fproto` declaration describes a *property-graph* (see Figure 2.9) indicating the causality constraints between inputs and outputs of the function.

A function prototype `fblx` is declared in Listing 2.4.2, where `fnc1` is the `fblx` name, an identifier that uniquely defines the `fblx` with the current scope. The visibility of the `fnc1` symbol is defined by the scope of the `module`, which is briefly discussed later in this document.

```
fblx fnc1 :: frame_compose
```

[7] That is, only full function name declaration, or the full mangled symbol from the scope and overload resolution of a function signature.

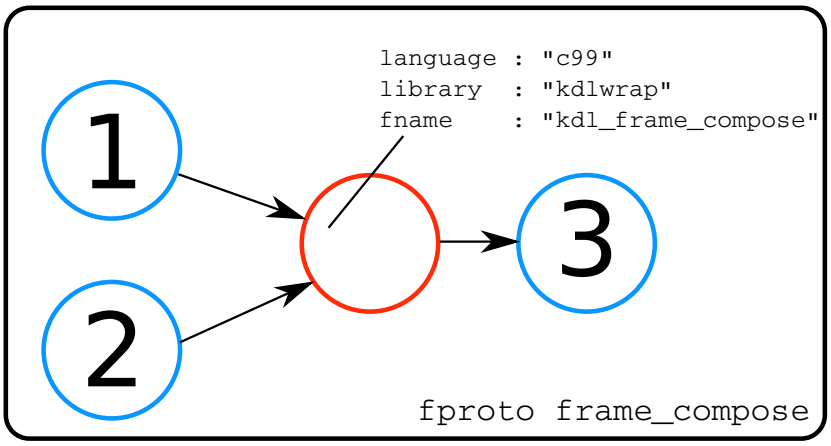


Figure 2.9: Property graph representation of the `fproto frame_compose` reported in Listing 2.2.

Data Block (dblx) and Data Block Prototype (`dproto`)

A data block prototype (`dproto`) is a model that allows to annotate existing datatypes, with the aim to describe its semantics and its representation choices (mathematical, unit of measurements, etc). The following snipped is an example of a declaration of a `dproto`, in the context of `geometry` domain:

```
dproto s_plain_position :: geometry {
  semantic  = Position
  coord     = cartesian
  ddr       = :: c99 { double [3] }
  algebraic = position3
  dr        = {0=0,1=1,2=2}
  units     = position_units
}
```

As for the `fproto`, the `dproto` annotations depends on the specific domain. In the example above:

- Semantics of the data (`semantic`): annotation that indicates the semantic value that the `dproto` represent, in the context of the `geometry` domain;
- Digital Data Representation (`ddr`): the data structure, in-memory, to store the value that the `dproto` represents. In the example, it is described considering c99 data structures definitions;
- Coordinate representation (`coord`): this is specific of the `geometry` domain, and it denotes the choice over the coordinate representation
- Algebraic (`algebraic`): this represents the reference to the *abstract data type* that the `dproto` is using;
- Data representation (`dr`): a mapping between the abstract data type (`algebraic`) and the digital data representation (`ddr`);
- Unit of measurements (`units`): annotation of the units of measurement employed to interpret the numerical values.

Regardless on the domain, it is important to notice that a `dproto` declaration is already a *composition* of annotations, which are defined in the specific domain. To define a new set of annotations, the `domain` keyword is used, establishing a new composition rule that the `dproto` must comply to. For example, the following describes the annotations on the `geometric` domain:

```
domain geometric :: {
  coord, units, algebraic, dr
}
```

In other terms, the `domain` expresses the meta-model of a `dproto`, by means of creating a reference to the domain with the name of the domain (*e.g.*, `geometric`), which represents its `id`. To be noticed that the annotations `semantics` and `ddr` are not specified: this is because these annotations must be always exist in a `dproto` declaration.

At M30, the `domain` has not been implemented yet. This means, a mechanism that fully exploits the domain description has not been included in the current tools. Therefore, the snipped above may change in the future, in order to proper define to user-defined domains.

Nevertheless, the `dproto` declaration as-is already suffice to build `dconv`, an automatic data type conversion tool, in the context of the `geometry` domain. More details on the `dproto` and on `dconv` can be found in the Annex (Chapter 5) of this document.

Finally, a `dblx` is a concrete *instance* of a `dproto`, and it represents the concrete block in memory. The following snipped declares (and instanciate) a `dblx` named `data` (the `id` of the `dblx`), that conforms to the `dproto` named `s_plain_pose`.

```
dblx data1 :: s_plain_pose
```

When interpreted, the MFCF runtime allocates a block of memory of the size of the `ddr` (digital data representation) indicated by its `dproto`.

Composite `fblx` and closures: a computational model definition

The previous sections introduced `fproto` and `dproto` models, as annotations of existing functionalities and data structures, encoded by using different programming languages. On top of these definitions, it is possible to specify a new `fproto` as a *composition* of existing `fproto` (and `dproto`) models.

Listing 2.3 shows a first example of *composite* `fproto`, namely `fnc_composite`:

- `fnc_composite` requires two `dblx` as inputs, namely `d1` and `d2`, and one `dblx` as output, namely `d4`;
- the composite function uses two `fblx`, which as been defined *unbinded* (lines 4-5). That is, in their declaration no instance of `dblx` has been indicated as argument. The binding with `dblx` is defined later (lines 6-7).

The internal structure of `fnc_composite` is illustrated in Figure 2.10 as a property graph. In details, this graph is a *Directed Acyclic Graph* (DAG), where *data dependencies* constraints are explicit by means of an input/output model, *e.g.*, the `dblx` `d2`, output of `fblx` `fnc1`, is in turn input of another `fblx` (*i.e.* `fnc2`). It turns that to compute an output of a composite `fblx`, a sequence of computations must be performed. This chain of computations defines the data dependencies that must be satisfied, a *computational pipeline*, and the outcome is the schedule to obtain a new output value.

Data dependencies are not the only ones that can are applied to this computational model. For example, it is possible to associate a constraint on a `dblx` to determine whether the cached value of an intermediate computation can be reused again. This mechanism is useful in those cases where not all inputs must be "fresh", or better, in those cases where *data freshness* policy is heterogeneous among `dblx`. In short, the component supplier (or the configuration of the component) can decide to perform a chain of computation to obtain an updated output value as soon as a new input has arrived, regardless from the status of the other inputs.

Moreover, in case a composite `fblx` produces multiple outputs, it is possible to impose *priority* constraints, such that the MFCF executive evaluates first the computational pipeline having higher priority. This enables different *quality of service* to different outputs, in case the component configuration and the *cause-effect chain* of the application requirements impose that. Actually, in a large extend, priorites on the computational pipelines conforms to the *cause-effect chain* meta-model.

A way to impose those constraints (which are higher-order constraints with respect to data dependency constraints) is the following:

```
fblx fnc1 :: fproto1
fblx fnc2 :: fproto2

//constraints between fproto
fnc1 > fnc2
```

by indicating to the MFCF executive will take into account that `fnc1` must be executed *after* the execution of `fnc2`.

From a composite `fproto`, a `fblx` can be instantiated as usual, binding the arugments to `dblx` instances (line 16). Instead, Line 19 indicates which functions must be evaluated by a MFCF interpreter and executive, at runtime. The MFCF execution engine must be called within the "even loop" of the *activity* of the software component, as described in Section 2.4.3.

The `fproto` definition in Listing 2.3 does not impose any constraints on the "type" of the `dblx` used as arguments (*i.e.*, the `dproto` model that the `dblx` must conform to) . MFCF language also allows to specify "typed" argument constraints as follows:

```
fproto fnc_composite(d1::dprotoA, d2::dprotoB)
                     => {d3::dprotoB,d4::dprotoC} [ ... ]
```

Finally, `fproto` allows *closures* definitions, both on data (`dblx`) and on functions (`fblx`); the symbol "?" allows late bindings:

```
dblx da, db, dc :: dproto1
fblx fnc3 :: fnc_composite(?,db) => dc
fblx fnc4 :: fnc3(da)

fnc4()
```

Note that in the example above `fnc3` is not *executable*, since the the arguments of the `fblx` model is partially binded to `dblx` instances, while `fnc4` can be evaluated.

Closures on functions (`fblx`) can be useful to define high-level patterns (or structures), by defining generic common constraints but without specifying the concrete `fproto` definition, as shown in Listing 2.4.2.

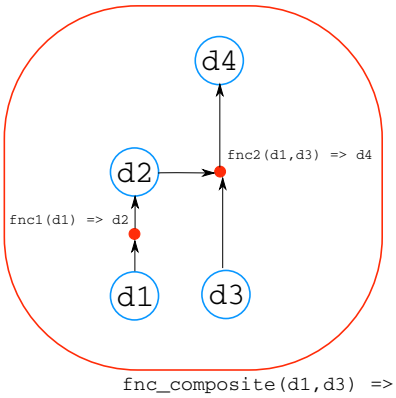
```
// fproto definition
fproto fnc_composite(d1,d3) => d4 ::
[
 fblx fnc1    :: fproto1
 fblx fnc2    :: fproto2
 dblx d2      :: dproto3
 fnc1(d1)     => d2
 fnc2(d2,d3) => d4
]

// usage
dblx da, db     :: dproto1
dblx dc         :: dproto2

// fnc3 is fblx
fblx fnc3 :: fnc_composite(da,db) => dc

// evaluate fnc3
fnc3()
```

Listing (2.3) Example of a composed `fproto` and its usage.



(a) Property graph (DAG) that illustrates the *data dependency* constraints.

Figure 2.10: A example of composite `fproto` (on the left), and the graph-based representation (on the right)

```
dblx da, db, dc :: dproto1
fproto fnc_composite<fncA,fncB>(d1,d3) => d4 ::
[
 dblx d2      :: dproto3
 fncA(d1)     => d2
 fncB(d2,d3) => d4
]

fblx fnc :: fnc_composite<fnc1,fnc2>(da,db) => dc

fnc()
```

Listing 2.4: Example of closure on `fblx`

Modules and Scoping

The MFCF DSL provides two concepts to determine the visibility of block prototypes (`fproto` and `dproto`) and block instances (`fblx` and `dblx`), bounding them to a specific scope:

- `module`: a module is a collection of `fproto` and `dproto` declarations, bundled together to ease their re-use. Once a module is loaded, a namespace is created in the current namespace (or global namespace otherwise). This creates a hierarchical namespace tree, in which the prototype definitions are confined, but accessible from any higher namespace. The "global" namespace refers to the highest level possible in a specific runtime, which is confined within a software component (deployed as a thread or fully fledged OS/process). As a general

practise, for any (shared) library used in MFCF a module should be provided, containing relative `fproto` and `dproto` definitions.;

- scoping is a mechanisms to fetch a MFCF primitive. In MFCF syntax, scoping adopts a dot "`.`" notation. The term "scoping resolution" refers to the operation of solving the (absolute) identifier of a MFCF primitive, starting from its (relative) `id`. Modules are not the only ones that define a scope, but also `fproto` definitions do. Referencing by scoping allows to expose to the outside (with respect to the scope definition) a `dblx` (or a `fblx`), thus creating a new closure. For example, it is possible to refer to the `dblx` "d2" in Listing 2.3 and associate it to another `dblx` defined in an higher scope (upvalue):

```
dblx da    :: dproto1
fnc_composite.d2 = da
```

Conformity to Block-Port-Connector

The definition of a `fproto` reported above is expressed by means of a *usable* DSL, that is, a language that capture the expressivity in a compact form, aimed to be convenient to the "user", i.e., the Function Developer. However, the definition of a `fproto` conforms to the Block-Port-Connector (BPC) meta-model. From the structural point of view, instantiate a `fblx` means: (i) to create a block (instance) (ii) having a number of ports equivalent to the number of arguments expressed in its `fproto`; (iii) such a block conforms to both BPC meta-model and to the `fproto` model (meta-model identifier, `mid`), (iv) while it has an unique `id`, allowing further references to this block. In the concrete MFCF realisation, the `id` is string-based (the name of the `fblx`), unique within the specified `module` (the scope), while the reference to the `fproto` is preserved (thus implementing *reflection* as a property of the language). A graphical representation of a `fblx` is shown in Figure 2.11.

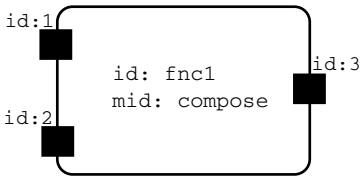


Figure 2.11: A graphical BPC representation of a `fblx` (structural-only), relative to `fnc1::frame_compose`, with `frame_compose` being `fproto` defined in Listing 2.2.

Moreover, in the proposed DSL, a `fproto` declaration adds two constraints:

- restrictions on the port connection, the "datatype" (i.e, `dproto`) that the `fblx` accepts through that port. In the BPC model, this is a property attached to the `port` element;
- Causality, i.e., data access constraints on the data (input/output). In the BPC model, this is a property of the `connector` that models the relationship between the `fblx` and the `dblx`. However, this is expressed in the `fproto` declaration, as a constraint that is expressed in the proposed DSL.

Discussion on MFCF design

This section resumes few design decisions, their rationale and limitations of the current MFCF solution. The status of the framework (DSL and runtime implementation) is in alpha version, so

improvements with respect to the current status are expected. Furthemore, there are few minor discrepancies between the modelling and the reference implementation: some are unavoidable, arbitrary design choices that must be considered as "implementation details"; others are merely related to the level of maturity of the tool, and the implementation "shortcuts" needed to release a first viable product. In both cases, any of these discrepancies influences in any way the general approach.

- **Usability first**. The main drivers of the MFCF DSL are **usabilty** and **minimality**. For example, unique `ids` for `fblx` and `dblx` are scoped user-defined names, instead of being Universally Unique Identifiers (uuid) or a proper Uniform Resource Identifier (URI). However, it is also possible to serialise MFCF models described with MFCF DSL to other hosting formats that aim to **completeness**.

- **No data hiding by design** because it prevents composability. This is unvoidable, since everything that is hidden cannot be referred to, nor it is possible to add constraints on those elements. Instead, the more the functionality is exposed, providing its full set of configuration options, the more is possible to compose imposing

- explicit **causality** is another design driver, in particular regarding the *data dependencies* imposed with a clear input/output model. This is necessary for better functional composability, but also for a better interface with the component model. This is done by exposing the "internals" of a software component by means of the input/output data that the algorithm is using. Policies and configuration regarding the *data freshness* are also related to the cause-effect chain metamodel;

- **Inspired by functional-programming**. The design of the MFCF is strongly inspired by functional-programming, with a clear input/output data dependency model. The `fproto` models allows to "elevate" to pure functional models implementations that are not purely functional, such as implementation of functions that follow object-oriented paradigm. Data protection can be implemented by adding explicit constrants on the model, for example, defining *data access constraints* on top of *data provenance setting*. The advantage is that those constraints can be modified during the runtime, instead of being statically defined at design time/compilation time. This is further elaborated in Section 3.1.

- the current implementation supports the inclusion of functionalities implemented in Lua and C programming languages. Support to C++ is limited for the moment, and embedding header-based libraries, or libraries that makes strong use of meta-programming features (templates, auto-traits, etc.) is not trivial. At the moment, it is possible to declare C++ based `fproto`, introducing the concepts of `class`, `namespace` and accessors (getter and setters). Depending on each single case, it is possible that the C++ objects must be wrapped in a ANSI-C99 compliant form. Support of other scripting languages, such as Python and Julia, has been proven of being technically possible, but no efforts have been allocated to this at the moment. Another interesting language to support is Rust.

2.4.3 Links between computational model and component model

One of the purposes of the MFCF DSL is to specify a computational model to represent the *internal*s of a software component, allowing to embed functionalities developed by the function developer , but how exactly? *How to combine a computational model with a component model?*

The RobMoSys component model is one of the RobMoSys composition structures that defines:

- an interface to interact with other software components, by means of service definitions (horizontal composition);
- models to define *activities*, *execution containers*, and to handle *hardware resources* (vertical composition);
- models (and relative mechanisms) to coordinate all the above, from the "outside" (*e.g.* a coordinator or "master" a component), by means of operational models defined within a component *life-cycle*.

A MFCF model is meant to be deployed in one or more activities: each time that the boundaries of an activity is crossed, a third model that defines the interaction between the activity model and the MFCM model must exist. Considering the case of a single activity in a single execution container, a MFCF runtime[8] should be embedded in that activity, being part of the activity main "event loop" (see Deliverable D3.3). Therefore, the points of interactions between the activity model and the computational model are:

- input and output **data** that is shared between the activity and the MFCF model. Each data incoming (or outgoing) to (from) the activity from (to) the service (and later, shared with other components) must be related to a `dblx` input/output of the computational model. To implement this, the MFCF DSL introduces a keyword `external`, to define that a `dblx` is not handled directly by the MFCF runtime, but from "the outside":

  ```
  external dblx datain :: dblx_model
  ```

  This means that is the component owns the data: it is in the component model that defines the data model (by means of a communication object model), and the memory management of that block of data is delegated to the component's implementation. Obviously, there is a need of a third DSL that configures MFCF DSL and the target DSL that describes the component model (*e.g.*, the SmartMDSD component-model DSL, or the DSL adopted by Papyrus4Robotics);

- **configuration**: services in a component are configured within a certain policy, *e.g.*, if a data is sent sporadically or periodically. Real-time constraints can be imposed to this service policies, to reduce the latency in critical components. These constraints should be reflected in the MFCF model, as a *configuration* of the scheduler policy, *e.g.*, giving priority of computing the most critical output. Other type of policies regards *data freshness*, implementing what has been discussed in Section 2.2. This aims to the configuration and validation of non-functional properties of component-based architectures, as the **cause-effect chain**, to the component's internals, without the need for the component supplier to manual programming.

At M30, a proof-of-concept of the mechanism that allows to embed a MFCF engine within an activity of a RobMoSys baseline (namely SmartMDSD) has been implemented. However, a third model that links the specific component-model DSL adopted by SmartMDSD and a MFCF model has not yet formalised. This activity is foreseen in M30-M40. In order to prove the benefits of the proposed MFCF framework, the same will be realised using Papyrus4Robotics baseline. In this way, it is expected that porting an algorithm from a RobMoSys baseline and another is trivial.

[8] MFCF runtime refers to the executive engine that interpret and perform the computing of a MFCF model

2.5 Horizontal Composition: data-conversion tool

To fully implement horizontal composition, there is the need to convert a `dblx` to a *Communication Object* (and viceversa). To this end, `dproto` annotations suffice to create automatic data-conversion tools. This is further explained in the Annex, Chapter 5.

2.6 Integrated Technical Projects (2nd wave call)

The 2nd open call of RobMoSys ITP included contributions to *functional composition* topic in instrument #2, for which at least 1 ITP project is expected to be approved. Therefore, further contributions and results are expected to be obtained from the RobMoSys community, and not by the sole RobMoSys core consortium. Depending on the approved ITP, the RobMoSys consortium will work in strong collaboration with the ITP partners, by coaching them regarding the principles described in this section. Moreover, interoperability (o complementarity) with the proposed MFCF and the tool developed by the ITP will be investigated. This will allows to strengthen the RobMoSys ecosystem and its adoption.

3. Motion, Perception and World Model Stacks

This Chapter shows how models and concepts described in deliverable D3.3 are applied to create software and tools, as basic building blocks for the integration of motion control, perception and world modeling, for any robotic application. In this context, the term "*stack*" represents the (often huge) set of functionalities that are relevant in a particular domain; for the robotics domain, the **world model** takes the central role, between "task", "control" and "perception". The stack design can exploit the well-known duality between control and perception: the former links the world model to the actuators, the latter links the world model with the sensors. Nevertheless, those stacks are closely-coupled, in the same way as function, data and control flow are, that is the realisation of an application is not possible considering only one "stack". The logical separation in "stacks" only helps as a gentle introduction, and decouple the large number of proposed challenges.

These tools are designed and realised on top of the principles and implementation mechanisms already described in Chapter 2, which realisation was not possible without the continuous formal modeling efforts that started with the deliverable D3.1 and now updated with deliverable D3.3. In particular, the *property graph* concepts and its reference implementation plays a fundamental role in all "stacks". Special care and efforts have been given to the design of the world model stack, with many discussions among the RobMoSys consortium, including industrial partners to fully support the Pilot cases. Chapter 4 summarises some collected requirements. As a remark, the tools described in this sections are developed internally by the RobMoSys consortium, and this document presents their design and the status of their reference implementation at M30. However, these tools are not the only ones representing the set of facilities of the RobMoSys ecosystem. In fact, another set is provided by ITP results (1st call), and it will continue with the second wave of ITP projects. The software here described is planned to be used within the context of the Pilot scenarios and 2nd call ITP projects, depending on the resulting approved projects.

3.1 World Model Stack

The *world model* modelling serves multiple purposes, the major three being:

- *formal representation of meaning* (Sec. 3.1.1): it represents the **knowledge** that one has about how all data about representations of the world should be interpreted in a consistent way, and, similarly, how to make sure that updates in concrete world model data structures are always done consistently?
- *explicit model of abstract data type* (Sec. 3.1.2): to describe which concrete data structures to use in a *concrete* programming language.
- *loose coupling between perception, task and control* (Sec. 3.1.3): all these parts in any robotics system must share information about the world, *at runtime*, and that observation gives world modelling a central place in every robot's software system design.

Geometry is the most important common foundation of all task aspects in a robotic system. Hence, geometry gets a priority treatment, in this document and in the model and software development. A major outcome of the work realised until now is a meta model for geometry to serve many complementary representation purposes:

- **maps**: where in the world are which "features"?
- **kinematic chains**: how are motors, links and joints connected together in the body of a robot? Where are sensors and tools attached to that body?
- **sensor information**: camera images, laser scans, tactile and force sensors, all have the same limited set of geometric primitives in the models that represent the relations between the "state" of the robot in its environment, and the "data" that is produced by the sensors.
- **control information**: the same geometric primitives are a core part of the specification of where the robot should move to, and what are the distance constraints it has to comply with,
- **plan transitions**: many conditions that the robot's plan requires to be monitored make use of geometrically specified areas.

Geometrical information is not always *numerical* and/or directly *observable*. Many geometric relations are **qualitative** (and hence purely *symbolical*), such as:

- *the robot is in front of the table*.
- *move towards the door until it is within arm reach*.
- *hand over the bottle from right to left hand*.
- *there are two cars in front of this car*.
- *move to your left*.

Each of these qualitative expressions requires "higher-order reasoning" to ground them in relations that *can* be quantified, for perception, control of monitoring. A major objective of RobMoSys is to encode a relevant and sufficiently rich set of such qualitative relations, and the relations that allow to ground them.

A world model contains a lot more information than just the geometry for the maps. This Section describes two of them, with very generic "platform centric" importance: *provenance* and *data freshness*.

3.1.1 World Model knowledge as a property-graph

A world model is the set of **knowledge** on top of which a robotic system reasons, reacts and elaborates a control action to perform the desired tasks. The collection of a world model description includes perception inputs and their post-processed data, the state of world object instances, that is, the objects in the environment in which the robotic system operates, and more. A world model shows up in any robotics application, independently from the complexity of the task. For this reason, the world model is considered a basic building block, a skeleton on which the robotic solution is developed on.

Each element represented in a world model is an instance of an entity or relation which conforms to a specific model. The set of these models define an ontology, which is defined within the context of the application. Actually, such an ontology is a composition of taxonomies originated from different domains. The most common domain is the geometric one, the spatial representation of objects as a geometric abstraction and the geometric relations between those objects that conform a *geometric chain*. A world model includes the *robot model* (see Section 3.2.1), *i.e.* the description of the robotic system possibly involving multiple domains (kinematics, dynamics, actuators, robot capabilities, etc.), and *all* of this extra knowledge is *anchored* to geometrical entities anytime. The description of the environment and the robot, together, allows the instiantiation of a task specification, where "actors" and the subject of the action are well-identified. Finally, there are other domains of interest, but those are quite application dependent. For example, it is hard to generalise the properties that describe an object in the environment:

- physical properties can be generalised, but there are several assumptions on the representation choices of those properties, and whether they are invariant or not during the duration of an application. For example, shape or color can be considered invariant properties of an object, if they are assumed to be features of the object itself. In other cases the same properties are *attributes*: they can change during the application.

- non-physical properties are also application dependent. For example, it is possible to describe objects depending on their function or *utility*. However, the same object can be employed for different usages in different applications.

Besides, other properties of the world model can refer to non-functional parts of the application, such as those models and meta-models that describe the software of the application itself (or its configuration).

Therefore, **composability** is a mandatory requirement for a world model, such that properties and descriptions that makes sense only in one domain can be merged, stored and linked with other properties belonging on different domains. This composability can show up at design time, but also at runtime of the application, for example, if a new object instance enters into the environment, and the robotic system is capable of generating a model of it from observations. To support composability, the modelling tool chosen to create a world model is the property graph, as discussed in Section 2.3.

As a illustrative example, let us consider an simple scenario shown in Figure 3.1. In this application, a manipulator has the task to grasp a hose adaptor which is located on the workbench. To locate the hose adaptor, the robotic solution makes use of a camera that provides the pose of the hose with respect to the camera's reference frame. Figure 3.1 highlights the frames involved in this world model description.

Figure 3.2 is a possible property-graph representation of the world model described in Figure 3.1. Recalling the running example in Section 2.3, frames $(F_1, F_2, \dots)$ are represented as nodes, as

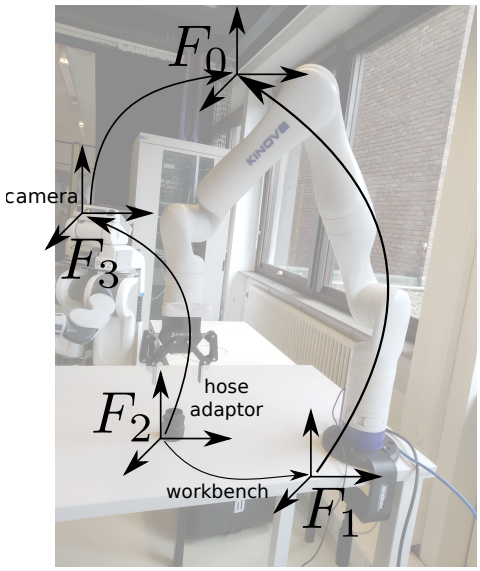


Figure 3.1: A simple "scene" of a robotic application. The scene contains: a camera, a robotic arm, an object subject of the task and a workbench. In the figure, frames attached to each world model elements and their geometric relations (*i.e.* relative poses) are indicated.

well as the geometric relations of type `Pose`. For each `Pose`, a relative measurement node is indicated, linking the "value" ("data block" (`dblx`) node) hosting the measurement. Moreover, the "provenance" and the "life-cycle" of the measurement is explicly defined: the former indicates the origin of the measurement (*e.g.*, a configuration, input from a component, etc); the latter indicates for "how long" the current value of the measurement is valid. Further details about the concept of provenance and life-cycle are described in the following paragraphs.

To complete the scene of Figure 3.1, each world model object (such as the `workbench`, the `hose_adaptor` and the `camera` device) is attached (*i.e.* `attached_to` relation) to one of the frames previously indicated. Thus, the pose of the `camera` object with respect to the `workbench` is determined by the pose measurement of F_2 with respect to F_1. In short, the world model representation in the form of the property graph of Figure 3.2 already separates the *geometric domain*, the measurements and a symbolic level of abstraction of object identifiers. In turn, the nodes that represent the symbolic abstraction of object instances in the world model can be used to express other kinds of relation, such as the *qualitative spatial-topological relation* that the "`hose_adaptor is_on the workbench`", or that the "`robot_endeffector is_above the workbench`".

To conclude, a graph-based representation of the world model is not limited to host *instances* of world model objects, but also their *models*. Even if instances and models are two different (sub-)graphs, they can be hosted together by means of the relation `instance_of`, which can be represented by a simple edge in the graph. Figure 3.3 shows a concrete example regarding the model that describes objects of type `hose_adaptors`. In the example, the node that represents an *instance* of the `hose_adaptor` (see Fig. 3.1) has a *model ID* (`mid`) linking to `hose_adaptor_model`; the latter is also represented in the property graph as a node. The role of the `hose_adaptor_model` node is, obviously, to host the description of such a model, and all instances of such model must conform to it. In fact, this model defines the properties of "hoses": color, shape and other physical properties; the role of an hose in an assembly task, etc. In short,

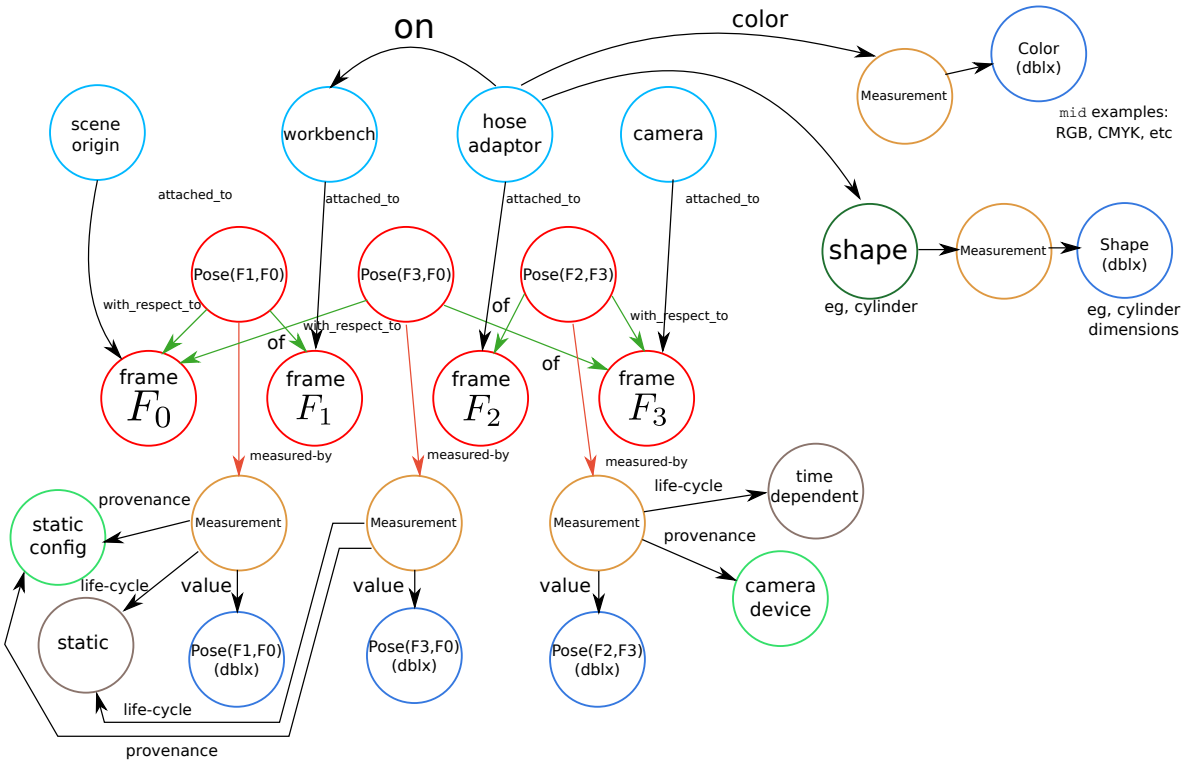


Figure 3.2: A possible property graph representation of the application illustrated in Figure 3.1. For the sake of clarity, this representation is simplified and many properties and relations are omitted. Nevertheless, it should be clear that there are many examples of "higher-order modelling", that is, the relations have certain dependencies that structure them in "hierarchies". For example, measurement data does not make sense if it is not related to models of what is being measured.

a graph-based world model can also host ontologies, each one for a different domain, to which world model object instances conform to.

3.1.2 World Model Runtime

The term *world model runtime* refers to the software tools and implementation responsible to instantiate, store, elaborate, share and distribute world model-related information among other software components.

Any robotic application needs and implements a world model mechanism, implicitly or explicitly, to handle runtime information about the world model. An example of a popular world model runtime is the ROS-TF library [7], which provides an infrastructure to distribute numerical measurements of spatial relations in form of *transform frames*. The ROS-TF is based on ROS publish/subscribe communication mechanism: any ROS node can publish on a specific topic (*i.e.*, `tf`) and any ROS node can subscribe to it. The final responsibility of reconstructing the transform frames graph is up to each consumer node: the hosted code in the consumer node can "query" the local graph for computing relative poses. This allows to distribute transform frames, regardless of the nature of the reference frames, which can be attached to objects, a link of a kinematic chain, etc. In practice, this tool is handy for fast prototyping, and the design has been improved by partial separation between the communication and the computation needed to answer to queries.

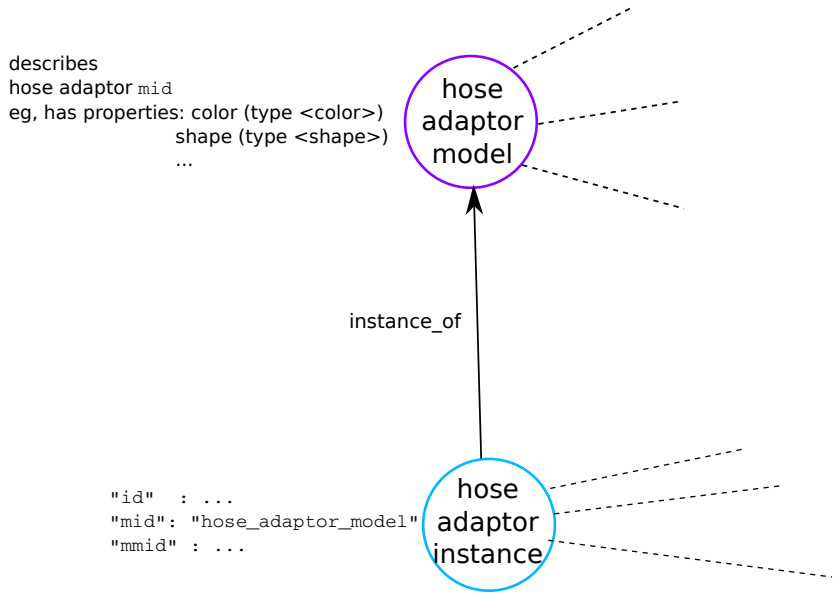


Figure 3.3: An example of a property graph representation of both *instance* and *model*. This particular relation refers to the example of Fig. 3.2

However, this tool has several limitations to truly support the role of a world model runtime; just to mention a few:

- there are multiple assumptions in the numerical choice to represent the transform frame: digital data representation, mathematical choice, etc. The consumer node may rely on different choices, so manual (thus unvalidated) conversion must be developed (also called "glue-code").
- the library supports only the distribution of numerical transformations, without symbolic reference to the relation between frames and their measurements, or to any other symbolic information that is needed to explain the role of these frames in the application.
- frames are only a small subset of the geometric primitives that are relevant and necessary in robotics applications; for example, vectors or polygons.
- the ROS-TF library supports only trees, with no loops.
- other world model information is not considered, and can not be refered to.
- each consumer of TF information needs to reconstruct the full tree; this lead to an unnecessary computational overhead.
- data consistency cannot be checked (let alone guaranteed), as well as other non-functional properties (*e.g.*, latencies, freshness, tolerances). This prevents the usage of the library under deterministic requirements, *e.g.*, to realise a control loop or to base safety-related decisions on.
- every node can publish a transform frame, anytime. On one hand, this allows high flexibility at runtime, ideal for fast-prototyping. On the other hand, this approach is error-prone: there

is no control or access policy to the data, so it is easy to pollute the transform frame tree, including naming clashing situations.

Another popular tool is MoveIt! [5]. MoveIt! makes use of ROS-TF library, but adds more concepts, such as a scene graph, which includes a robot model and some primitives to handle the gripper status (*e.g.*, to establish constant relations between an object frame and an end-effector frame, in case that the end-effector is grasping an object). However, the implementation of these features are driven by the "needs" of the MoveIt! tool, and there is no generic modelling effort or mechanism to support the composition and integration of other entities and their properties that populate the environment.

A runtime to support world model data is already subject of scientific investigations (cf [3, 11]), and a major scientific topic in many European funded projects, such as RoboEarth [17] and Sherpa [16], just to mention a few.

The ambition of the RobMoSys project regarding the world model runtime is:

- to propose models and meta models to describe world model entities, but also to develop an information architecture and a component-based architecture;
- to develop a design of a world model runtime that overcome to the limitations indicated in this section, both on functional and non-functional level;
- and to provide a reference implementation of a world model runtime, suitable in a component-based architecture.

The reference implementation, currently under development, will show the expected benefits in the context of the Pilot cases. Moreover, a world model runtime should be able to perform *computation* over world model data, that is, pre- and post-data processing. This capability enables the execution of complex queries on the world model data, and non-functional operations such as automatic data conversion from one numerical value representation to another.

As discussed in the previous section, a world model runtime can be implemented on top of a *property graph* framework, whether it is an existing NoSQL, graph-based database (such as graphQL[1], Neo4j[2] and TinkerPop[3]), or based on the the property graph in-memory reference implementation described in Section 2.3. In principle, a "world model component" can embed a graph-based database, and its component model will add RobMoSys compliance to the databases software, enabling (horizontal) composition with other software components. However, there is no modern NoSQL graph-based database that suits the requirements of multi-robot system solutions [12, 2], at least when it comes to realtime and low-latency requirements.In fact, those databases are designed for *large, permanent datasets* in the context of enterprises and web technologies, while a world model runtime requires a minimal footprint in terms of memory and computational power, to be used closely with motion and perception algorithms, and not only as knowledge-systems. Besides, it is not trivial to embed these existing databases within a software component with the current RobMoSys software baseline, due to technological incompatibilities (*e.g.*, different programming language target of the component implementation, etc). Therefore, the role of commonly available databases is relegated to store large knowledge-systems, log information, and so on, but not they are not available to be embedded into motion and perception stacks. Some databases *do* have a "realtime" in-memory core, such as RethinkDB, but that core can not be re-used as a stand-alone library in other software systems. The RobMoSys development does reuse many of the "best practices" that these databases rely on too. For example, single

[1] graphQL, <https://graphql.org>
[2] Neo4j, <https://neo4j.com>
[3] Apache TinkerPop, <http://tinkerpop.apache.org>

producer-consumer streams to provide immutable data; possibility to store "callback" functions in the database with user-configurable triggers for their execution; "cursors" to improve iterative requests for the same type of data.

Conversely, the amount of data that a robotic world model runtime must handle is relatively limited, and strong realtime and low-latency requirements apply. There are some robotics use cases, though, such as teaching by demonstration or learning, that do not have the same realtime performance expectations, so the "learning" could take place via the mentioned existing databases.

Also the number and the meaning of queries that the world model runtime must support are limited and well-defined, that is, the query answering can be hardcoded by domain experts (*e.g.*, geometric domain, robot kinematics, scene description, task specification, etc.).

Finally, there are other non-functional properties that are relevant for a runtime of a world model implemented as software component, and those non-functional properties introduce further benefits in the RobMoSys approach. It follows a brief discussion on two of them: **data provenance** and **relations life-cycle**.

Data Provenance

The main purpose of the *data provenance* concept is to indicate which software components are responsible to provide which data, being the components *producers* of (a part of) the world model information[4]. The knowledge about provenance is an enabler of important benefits in a component-based system architecture:

- **logs and debug**: preserving data provenance information in logs allows to better track the source of faulty measurements;
- **reflexion**: the application itself can reason about the source of the data, and implement specific policies possibly related to the Quality of Service; for example, in a given context, data coming from component A can be "preferred" or "trusted" more over data from B.
- **data access rights**: a configuration based on data provenance allows to model the role of each software component regarding world model information.

Since world model data is shared among multiple software components, the latter benefit is very relevant in the design of a component-based software architecture. This enables to define *data access* ***constraints***, for example, by indicating at configuration time which components are allowed to *provide*: i) a measurement update, (*e.g.*, updating a `Pose` measurement), ii) a change over a symbolic relation (*e.g.* setting an edge in the property graph to model the physical constraint of an object grasped by the end-effector), iii) or the modification to the object model itself (*e.g.*, adding a new property, a new node in the property graph, etc). Since world model data can be cached locally in a software component, data access constraints related to provenance can apply **globally**, to the whole component-based architecture or **locally** to a single software component.

To conclude, the *data provenance* is yet another important and necessary meta-model proposed in the RobMoSys approach. Data provenance can be used to configure data access constraints, as a configuration of the system architecture. A brief example, together with the *life-cycle* of relations, is illustrated in Sec. 3.1.3.

[4]The concept of data provenance is not limited to world model information, but to all data shared between software components. However, data provenance impacts the world model capabilities significantly.

Relations life-cycle

Object instances in the world model, represented in a property graph by means of *entities and relations*, may have an additional *life-cycle* property. In general, the definition of life-cycle depends on the domain of the stored knowledge. In the context of the world model, this concept can be concretely defined. As an example, let us consider geometric entities and relations, such as a (relative) `Pose` between two `frames`, which are possibly associated to object instances in the world model. The initial `measurement` of the `Pose` can be known a priori, and updated by perception components at runtime. In other cases, the relation measurement is *invariant*, for example the measurement of the `Pose` between two non-moving objects in a flexible assembly cell, like the workbench and the manipulator base.

This invariant measurement assumption may also be temporary with respect to the runtime of the application, *e.g.*, the `Pose` measurement between an object and an end-effector can be assumed of being constant while the end-effector is grasping the object. Moreover, there is no need to receive the update of a measurement from a perception component, if that measurement can be predicted by a motion model known a priori, *e.g.*, an object placed on a conveyor belt. The above can be generalised to other type of measurements (*e.g.*, the color property of an object) and it is not limited to measurements, but can be applied to symbolic relations as well.

The knowledge about the life-cycle of a relation can be exploited in many ways in a robotic application, with a specific impact on the configuration of its component-based architecture, especially if combined with data provenance. For example, if a software component receives a measurement which is known to be invariant during the overall application, the component can store this measurement internally, without querying for it again in a later phase. To a large extent, caching a measurement or any other relation from the world model really means that the component is building another *local*, partial, world model.

Therefore, the role of relation life-cycle is even more relevant: it allows to establish when an information cached locally needs to be updated.

As a consequence, if the life-cycle of a relation is provided, it must be attached to the inquired world model data (symbolic relation, property or measurement). The following is an enumerative list of the **life-cycle types** modelled so far:

- **static (or constant)**: the shared data (world model relation, measurement, etc) is considered invariant within certain defined conditions, *e.g.*, until a certain event occurs, or for the overall duration of the application.
- **time-dependent**: the shared data (world model relation, measurement, etc) is valid only for a well-defined time-interval. For example, a measurement can be considered valid only within a duration starting from the instant when the measurement was performed (*e.g.*, time-out, see *data freshness*).
- **prediction**: when the evolution of an object property (physical or non-physical) is known, and its measurement can be predicted. Some examples are: if the motion model of an object is known, future measurements (*i.e.* measurements of its `Pose`, `Twist`, etc) can be computed, without the need of perception; the measurement of a `color` as property of a fruit can be predicted over the time if a deteriorating model exists, etc. Less intuitively, the same applies to symbolic relations, such as the relation of an object with respect to others as expected outcome of a specific robot action. Some examples are: the topological relation between objects after an assembly task; the mereological relation between an object and its container after a pick&place task, etc.

The knowledge about the life-cycle can be exploited in multiple ways, depending on the life-cycle type and the degree of adaptability of each single software component implementation:

- **static (or constant)**: when a component `requester` inquiries about world model data[5], the component that provides that data (*i.e.*, the `provider`) can ship the life-cycle information together with the requested data. If so, the `requester` can cache this data without inquire for the same information again, thus avoiding any processing overhead[6]. Moreover, static relations (and their measurements) known at deployment time can be used to initialise the `requester` component, by means of its configuration (component parameters).

- **time-dependent**: relations and measurements with a time-dependent life-cycle allows the `requester` to evaluate the *freshness* of the obtained data. For example, a `timestamp` can be attached to the data, and the `requester` evaluates whether to inquire for a new measurement (or to wait for a new update if the cached data is too old – depending on the communication pattern employed). As an alternative policy, the data `provider` component can provide, together with a `timestamp`, an `expiration-date` of the data, suggesting when the data is no longer valid for computation.

- **prediction**: this enables the `requester` component to inquiry prediction on object's properties in different conditions than the current one, *e.g.* further in time. Moreover, instead of inquiring for the measurement directly, the `requester` can request the prediction model itself (formally, or in its executable form). This enables a form of *functional delegation*: the `requester` can embed and execute the prediction by itself, executing a foreign function *locally* (*i.e.*, in the computational resources handled by the `requester` component). The latter mechanism can be used for real-time control: the prediction functionalities are embedded and executed directly in the `requester` component, avoiding any communication overhead.

Finally, life-cycle knowledge can be specified in the initial configuration of the world model runtime: the configuration describes the initial state of the scene, defining all the entities, relations and their initial measurements. An advanced deployment tool can also make use this configuration, *e.g.*, by initialising the components with initial measurements provided with the configuration, if that is supported by the implementation of the target components. Life-cycle knowledge can be exploited to validate data provenance settings, *e.g.*, defining which measurements can be updated at runtime. For example, if a measurement is invariant during the overall duration of an application, and a component declares to update that measurement, then there is a conflict in the configuration model.

To conclude, *relations life-cycle* knowledge, combined with *data provenance* information, enable non-functional benefits when it comes to compose and distribute world model data among software components.

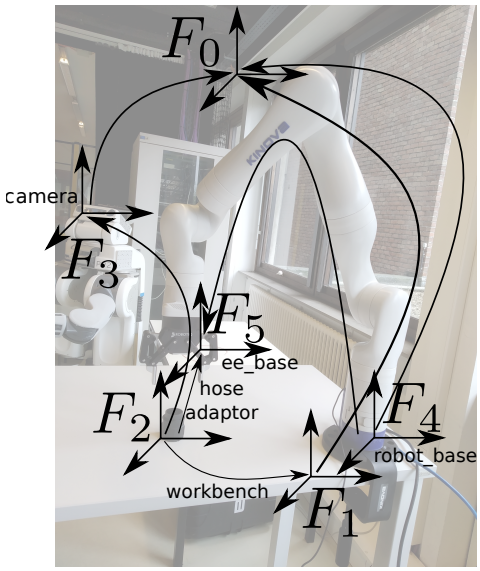


Figure 3.4: A "scene" from a robotic pick & place application. Revised from Figure 3.1.

3.1.3 World Model as a configuration of the information architecture

In this Section, the example of Figure 3.1 is extended (see Figure 3.4) to illustrate an example about how the world model can be used, together with the concepts of data provenance and life-cycle of relations, to define the *information architecture* of the robotic application.

For the sake of clarity, it is important to denote the difference between an *information architecture* and the *component-based software architecture* of an application. The former regards the design of the information flow, considering the *semantics* of the data, at a more abstract level. The latter is a refined version of a information architecture, where concrete choices have been made regarding the component models and related aspects, such as the communication object models and the service definitions. To a large extent, a component-based software architecture is a more concrete implementation of an information architecture, and of course multiple component-based architectures can implement the same information architecture.

The example of Figure 3.4 considers the manipulator and two new frames, F_4 and F_5, attached to the `robot_base` and to the `end_effector`. The robotic solution allows to perform pick&place operations, and in this example the picking phase is considered. Figure 3.5 illustrates the updated version of the world model representation as a property graph. The current task is also represented in the property graph[7], by a node `picking` that establishes a relation between the nodes `hose_adaptor` and the `end_effector`: a controller must compute the control action such that the relative pose between the two is minimised to perform the grasping action.

The relative pose between the `end-effector` and the `robot base` is modelled as well, and its measure is provided by the robot controller. This holds only under the assumption that the robot controller provides that measurement, by implementing internally a forward kinematic algorithm.

[5] this is valid also for any components sharing data, not only for world model data.

[6] In many applications based on ROS-TF, static transformations are distributed continuously (at a fixed rate) by means of "static_transform" nodes. This solution requires (i) extra processes (nodes), (ii) pollution on `tf` topic and (iii) computation overhead at the consumer side (subscribers to `tf` topic). Applying life-cycle knowledge on distributed transformations allows to avoid this runtime overhead.

[7] For the sake of brevity, in this example the task representation is rather simplistic.

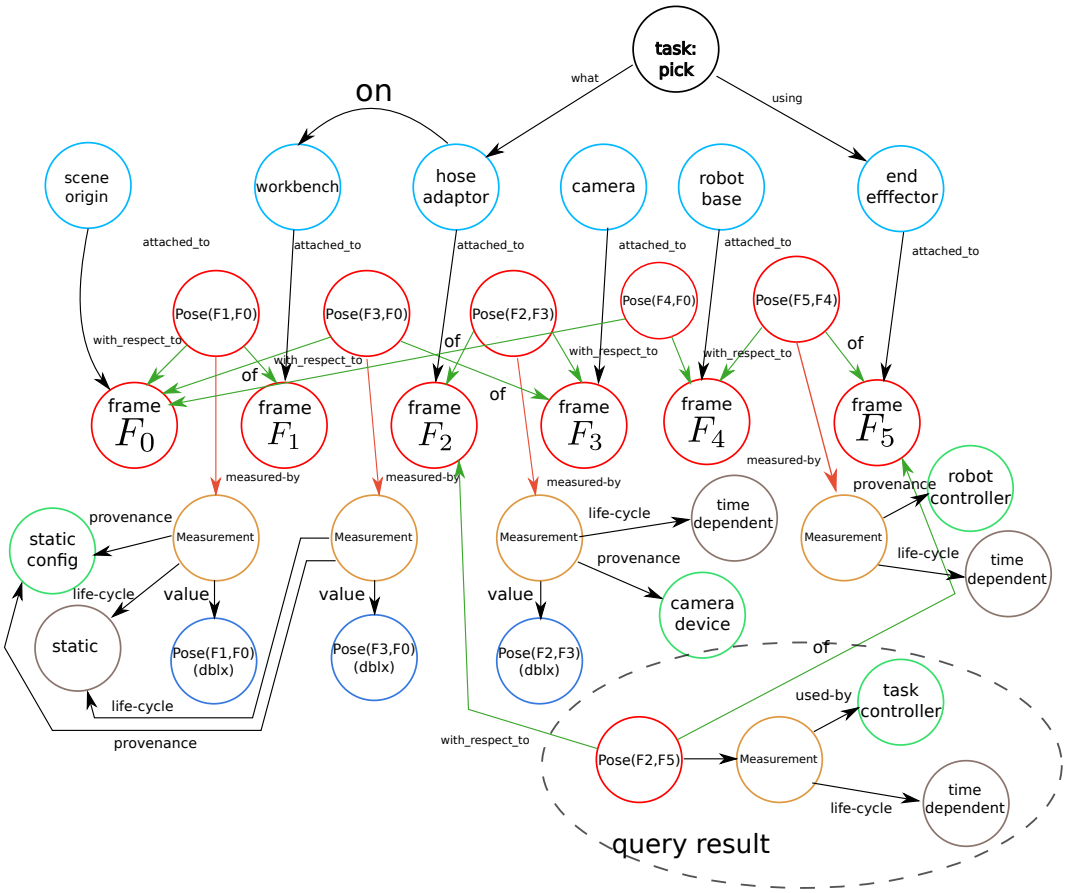


Figure 3.5: World model representation as a property graph. This is an extension of Figure 3.2, with the notion of task and other elements to perform the picking action.

This is a first point of variation of the information architecture. Figure 3.6 illustrates an alternative representation, where the world model runtime is responsible to perform the forward kinematics. In this alternative, a node `forward kinematic` represents the computational function that must be executed, its configuration (*i.e.*, reference to a robot model, also in the property graph) and its input (*i.e.*, the joint positions measurements coming from the `robot controller`).

In order to perform the approaching phase of the picking action, the relative pose $\texttt{Pose}(F_2, F_5)$ must be computed, stored and served to the `task controller`. In this example, the assumption that the world model runtime is capable of computing queries on geometric chains is taken. This means the world model runtime is providing that functionality, embedding an algorithm on top of the property graph mechanisms. Moreover, it is possible to extract only the list of symbolic operations to perform the computation of the transformation, thus delegating the numerical computation only (but not the resolution of the geometric path). If this assumption does not hold, the computation must be delegated to external components, and the measurements of $\texttt{Pose}(F_2, F_5)$ is provided as input of the algorithm.

From the world model information related to the information architecture, and the Table 3.1 that resumes the provenance and life-cycle of measurements relations, it is possible to determine the requirements and the configuration that the component-based solution must comply:

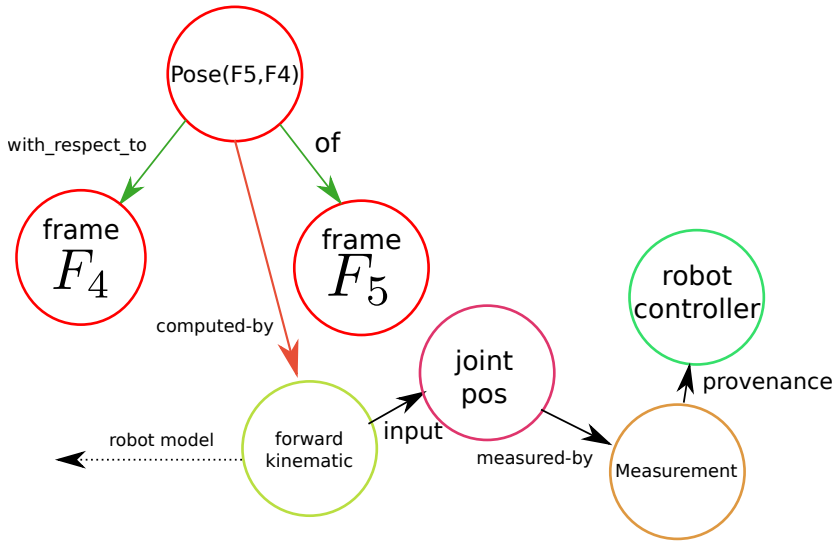


Figure 3.6: An alternative property graph representation for computing the measurement of $\texttt{Pose}(F_5, F_4)$. In this alternative, a node explicitly represents the computation (`forward kinematic`), using measurements of joint positions as inputs. For the sake of clarity, details regarding the robot model and other information are omitted.

Table 3.1: Life-cycle and provenance attributes for the measurement nodes in Figure 3.5.

measurement of	life-cycle	provenance
$\texttt{Pose}(F_1, F_0)$	`static`	`configuration`
$\texttt{Pose}(F_3, F_0)$	`static`	`configuration`
$\texttt{Pose}(F_4, F_0)$	`static`	`configuration`
$\texttt{Pose}(F_2, F_3)$	`time-dependent`	`camera-device`
$\texttt{Pose}(F_5, F_4)$	`time-dependent`	`robot controller`

- a camera component, that provides the measurement of $\texttt{Pose}(F_2, F_3)$
- a robot controller component, that provides the measurements of $\texttt{Pose}(F_5, F_4)$
- a world model runtime and its configuration (the static poses indicated in Table 3.1);
- the world model runtime must provide a service to inform a task controller component about the $\texttt{Pose}(F_2, F_5)$;
- if data access constraints are not respected, the update of the measurement must be rejected. For example, if the runtime of the world model receives the measurement of $\texttt{Pose}(F_5, F_4)$ from the camera and not the robot controller.

As a final remark, note that these requirements do not make any choice about the representation of the numerical values, how the component interacts, etc. These choices will define the concrete component-based solution (and they can be represented in the property graph as well, if necessary).

3.1.4 World Model Mediator Component Design (WMMC)

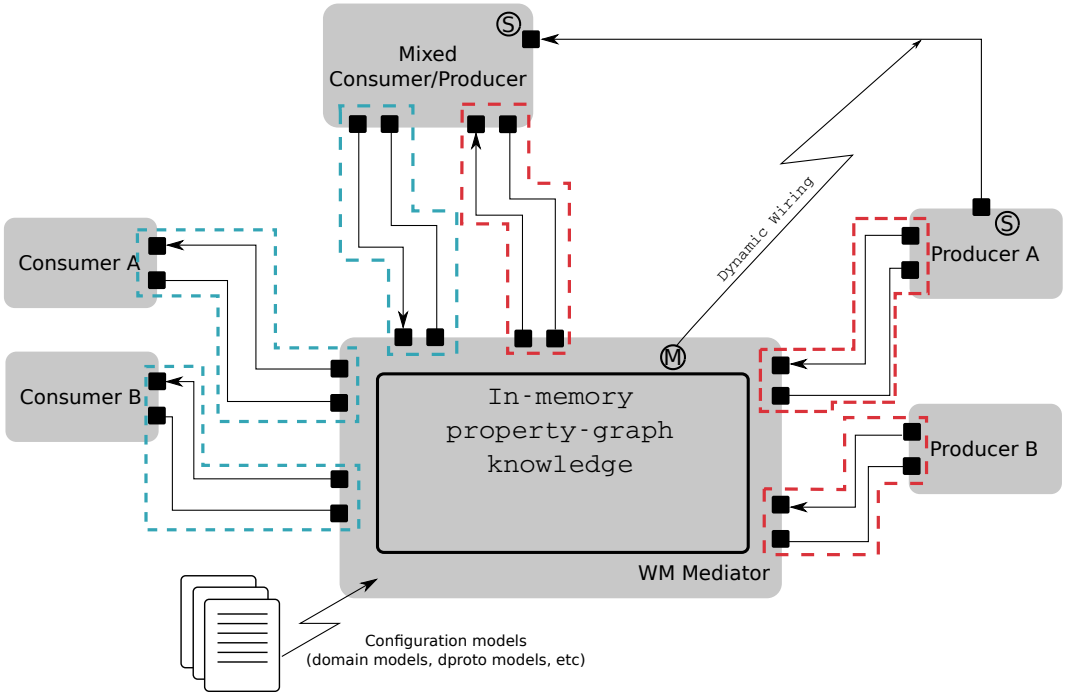


Figure 3.7: World Model Mediator Component Design.

Fig. 3.7 shows the envisioned *World Model Mediator Component* (WMMC). This component serves the role of central data mediator, a single point of exchange and distribution of world model information. The WMMC is responsible to keep in-memory a world model instance for the overall runtime of the application, which contains both object instances of the scene and their models. From the component architecture point of view, it fully implements a DIS (Data Integration System – see Section 2.2) software component, and it manages the interaction between those components that are users of the world model. Moreover, the design of this component fully conforms to the patterns described in the Deliverable D3.3[8].

In Fig. 3.7,consumer component and producer component indicate, respectively: i) a component that performs queries and fetch information from the world model mediator, and ii) a component that updates the world model (measurements, adding/removing world model relations or object instances, etc). Obviously, many software components are both consumer and producer of world model information. For the sake of clarity, in this section consumer or producer refer to a pure-consumer or pure producer component, or the part of a mixed producer/consumer component that covers a specific role only.

Each component interacting with the WMMC makes use of a specific *World Model Communication Pattern* and *World Model Protocol*, which includes a collection of dedicated Communication Objects Models.

[8] Cf. annex document of D3.3, chapters "meta models for information architecture" and "meta models for software architecture". In particular, see the pattern "mediator in peer-to-peer" activity interactions.

World Model Communication Patterns

The current design of the WMMC includes three specific communication patterns, which are built upon the RobMoSys core communication patterns and they are described as follows:

- **world model producer communication pattern** (bundled in red in Fig. 3.7): this *composite* communication pattern applies to producer components and it contains:
  - a Query Pattern that is used by the producer (Query Client/Service Requestor) to perform requests to the WMMC (Query Server/Service Provider): update of a certain measurement, property, creation or modification of instances of the world model, but also to initialise (and terminate) a new stream of data to be processed (by the WMMC), by means of the following communication channel. In addition, the producer can request the WMMC to perform a post-processing computation, after receiving a new measurement, if the requested operation is supported by the WMMC.
  - a Send Pattern, where the role of Send Client is reserved to the producer. This pattern is used to deliver new measurements values or updates to world model instances (entities, relations or properties) directly, for fast processing or specific requirements. However, enabling this feature requires a prior handshaking by means of the Query Pattern described above. During the handshaking phase, the producer informs the WMMC about the expected priority, quality of service, the model of the incoming data and which elements in the world model are subject of the update.
- **world model consumer communication pattern**(bundled in blue in Fig. 3.7): this *composite* communication pattern applies to consumer components and it comprises of:
  - a Query Pattern that is used by the consumer (Query Client/Service Requestor) to perform requests to the WMMC (Query Server/Service Provider), such as: fetching a specific entity, relation or property stored in the world model; establish a stable communication channel over some specific elements in the world model. In addition, the consumer can request the result of some processing of the world model information, if the WMMC supports it.
  - a Send Pattern, where the role of Send Client is reserved to the WMMC. This pattern is used to deliver *notifications* on updates of world model instances (or their properties), avoiding the need to query the WMMC each time. For example, a consumer component can subscribe to receive notifications each time the measurement of a relative pose is changed. To this end, the consumer subscribes itself, by means of a request performed on the query interface (cf. Query Pattern). In this handshaking phase, the consumer has the opportunity to request: the model on which the received data will conform to, sending policies (*e.g.*, sporadic or cyclic), and other non-functional properties of the interaction between the two components.

  In short, the distinction between this communication pattern and the previous regards the *direction* of the Send Pattern, that reflects the role of the producer and consumer of world model information, and the services that the WMMC offers through the query channel.
- **Direct data stream for real-time communication**. By means of the previous world model communication patterns, the WMMC is always *proxy* of world model information, from the producer to the consumer. This mediation can introduce overheads and delays that are

non-desired in real-time applications, *e.g.*, when world model information is used for closing a real-time control loop. To this end, the WMMC can establish a direct communication channel between the producer and the consumer (in addition to the update request of world model data coming from the producer). The consumer is responsible to request the WMMC for the initialisation of a dedicated communication channel, informing the WMMC about the non-functional requirements that the channel must comply to. Of course, this is possible only under the constraint that the service definition of both consumer and producer is compatible. To this end, it is important to notice that the WMMC can reject the request of the consumer component: it is the WMMC that manages that direct communication channel, by initialising or terminating it. This communication pattern makes use of:

- a Coordination pattern, namely the Dynamic Wiring pattern: the WMMC acts as *master* of the dynamic wiring pattern, while each software component user of the WMMC must register itself as a *slave*. The latter is a requirement on producer/consumer components to enable such a feature. In short, the WMMC is responsible to establish and to manage the communication between producer and consumer components by "wiring".
- a Query pattern, with consumer component being a Client Requester. A query can be performed on an already existing query interface, or on a dedicated one.
- (optional) the WMMC can still be updated with new measurements coming from the producer component, to keep track of the status of the world model. In this case, it is possible to re-use the Send communication channel between the producer and the WMMC. Otherwise, the world model elements (entities, relations and properties) that are not updated are marked as such in the internal (in-memory) representation of the WMMC.

To fully implement the World Model Communication Patterns described above, the definition of a *World Model Protocol* is necessary.

3.1.5 World Model Protocol

Software variability is one of the biggest technological barriers when it comes to sharing data among software components, and it is not an exception for the WMMC implementation. In fact, the shared world model data changes among different applications, and in particular the communication object models employed in the various software components (users of the WMMC) may vary to represent the same semantic value.

To overcome this limitation, the communication object models used to define the services employed in the world model communication patterns are of type string, delegating the concrete definition to a specific protocol called *world model protocol*. This approach is analogous to the one used to implement the skill definition metamodel, and it is briefly described as follows.

Currently, communication objects can be defined as in Listing 3.1 and Figure 3.8, where two equivalent representations are given based on the DSLs of the two alternative RobMoSys baselines, SmartMDSD and Papyrus4Robotics, respectively. Each communication object model is meant to be used in the service definition for the Send Pattern (*i.e.*, WM_Notification) and for the Query Pattern (*i.e.*, WM_Request, WM_Reply), in conformance with what defined in Sec. 3.1.4. Each model allows to ship data of type string, which is a valid JSON document in the world model protocol, respecting the following specification.

```
{
CommObjectsRepository WMCommObjects version 0.1.0 {

  CommObject WM_Notification {
     data : String
  }

  CommObject WM_Request {
     data : String
  }

  CommObject WM_Reply {
     data : String
  }
}
}
```

Listing 3.1: Communication Object models employed in the SmartMDSD implementation.

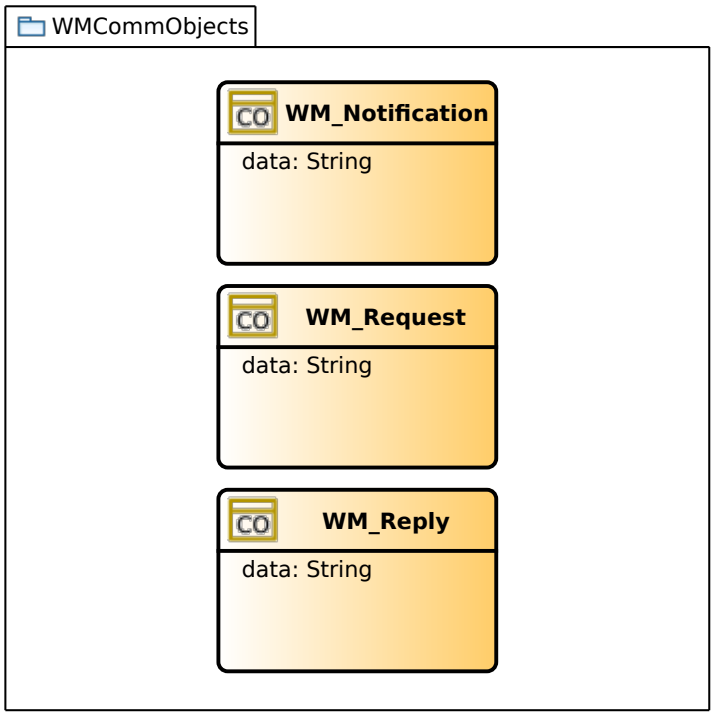


Figure 3.8: Communication object models employed in Papyrus4Robotics implementation, analogous of Listing 3.1.

World Model Protocol Query Service

As briefly mentioned before the interaction between the `WMMC` and any other component is defined by a "world model protocol", hosted in the JSON format. The aim of this section is not to provide a complete formalisation of that protocol (which is still in alpha phase, thus subject to major changes), but to illustrate it with a running example: the request of a `producer` component

to continuosly update a measurement related to a (geometric) relation in the world model. In this example, the `producer` component provides an estimation of the pose between a mobile base and a reference frame. This pose expresses the relation of a frame defined in the centre of the mobile base, named `base_link` for this example, and a global frame, named `map`, that is defined at the origin of the map the robot is used to localise into. An example of the interaction between the two components, including the handshaking phase can be seen in Fig. 3.9.

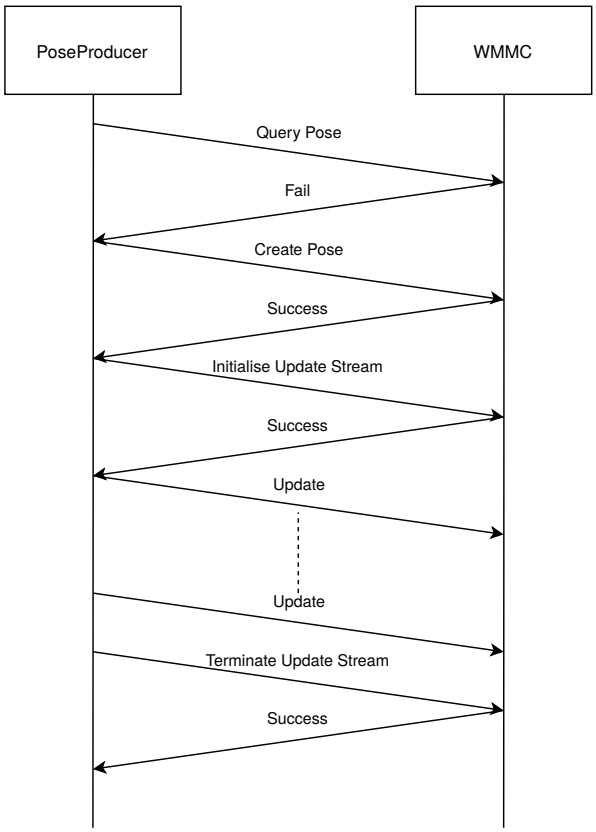


Figure 3.9: Procedure used to establish the continuous update of the pose of a mobile base in the `WMMC` by a pose estimation component.

Initially, the `producer` component queries the `WMMC` to fetch the `id` of the node in the property graph that represents the measurement of the pose between the two indicated frames. Currently, such a query is implemented as a JSON-RPC call (see Listing 3.2), where the remote procedure to call is a query in the property-graph hosted in the `WMMC`. It can be seen from the RPC call that the `producer` requests for the `id` of a node conforming to the `Pose` model (`mid`) of the `Geometry` meta-model (`mmid`). The node is required to be connected with two nodes conforming to the `Frame` model (`mid`), having the names `base_link` and `map` respectively. Once received, the `WMMC` translate the call into a graph traversal that perform the query itself.

As shown in Fig. 3.9 the query fails, and the `WMMC` replies with a failure indication, as showned in Listing 3.3. This can happen for various reasons. In the scope of this example, the query fails because there is no node that represents a pose between the two indicated frames.

Therefore, the `producer` must create such a node, to host the new geometric relation and its measurement. This request is performed by another call, illustrated in Listing 3.4.

The respective reply for the creation request can be see in Listing 3.5. The result of the response

```
{
  "jsonrpc": "2.0",
  "id": "1",
  "method": "query",
  "params": {
    "Id"       : "?",
    "nodeMId" : "Pose",
    "nodeMMId": "Geometry",
    "nodeProperties": [
      { "edgeName": "of", "direction": "out", "nodeMId": "Frame"
          , "nodeName": "base_link"},
      { "edgeName": "with_respect_to, "direction": "out", "
         nodeMId": "Frame", "nodeName": "map"}
    ]
  }
}
```

Listing 3.2: Example of query about the existence of a pose between two frames.

```
{
  "jsonrpc": "2.0",
  "id": "1",
  "result": ["Fail", "node not found"]
}
```

Listing 3.3: Example of query response about the existence of a pose between two frames.

includes the status of creating the new node, along with and object containing its `id` in a Base64 encoding and a property defining what the string represents.

Finally, to establish a continuous update using the send pattern, another RPC must be performed. Listing 3.6 shows an example of such a request. In this way, the `producer` is allowed to register updates of the measurement.

If the initialisation is completed successfully, a response as the one in Listing 3.7 is sent to the `producer`. The `producer` can then start updating the pose measurement using the send service described in the next section. In an equivalent manner, the `producer` component can inform the `WMMC` about the termination of that stream update by means of another RPC request. In addition, similar requests can be used to initialise direct connections between components for real-time cases, avoding the overhead caused by the mediation of the `WMMC`.

World Model Protocol Send Service

As it can be seen in Figure 3.9 after establishing a stream update the producer component can send updates to the `WMMC` in a direct fashion. This is also achieved by using JSON enconding on top of the `WM_Notification` communication object. An example of the notification message used to update the pose measurement can be seen in Listing 3.8.

```
{
  "jsonrpc": "2.0",
  "id": "2",
  "method": "create",
  "params": {
    "nodeType": "node",
    "nodeMId":  "Pose",
    "nodeMMId": "Geometry",
    "nodeProperties": [
      { "propType": "edge", "name": "with_respect_to", "
        propProperties": {"propType": "node", "propMId": "Frame
        ", "propMMId": "Geometry", "propName": "map"}},
      { "propType": "edge", "name": "of", "propProperties": {"
        propType" : "node", "propMId": "Frame", "propMMId": "
        Geometry", "propName": "base_link"}}
    ]
  }
}
```

Listing 3.4: Example of creation request for a pose between two frames

```
{
  "jsonrpc": "2.0",
  "id": "2",
  "result": ["Success", {"bin": "YAX==", "repr": "nodeUId"}]
}
```

Listing 3.5: Example of creation request for a pose between two frames

WMMC is not a coordinator component

To the eyes of a careful reader, the role of the WMMC can appear similar to the role of the coordination component of the application. In fact, there are some technological commonalities, such as their role as a *master* in a dynamic wiring communication pattern. However, the WMMC has a very different role than the coordination component:

- a coordination component embeds a "task specification", or better, a *behavioural model* of the application; the WMMC has no relations with the behavioural model, but it embeds the runtime state of the overall application;
- a coordination component *coordinates* other components by changing their operational mode (*i.e.*, the State pattern), and it can perform *monitoring* and (re-)configuration (Parameter pattern); instead the WMMC has no access to the (internal) state (*i.e.* mode) of a third component;
- a coordination component is responsible to take the initiative to perform a dynamic wiring connection, depending on the skill realisation; this does not hold for the WMMC, since the initialisation of a direct data stream is up to the components that are users of the WMMC.

```
{
  "jsonrpc": "2.0",
  "id": "3",
  "method": "init",
  "params": {
        "serviceType": "StreamUpdate",
        "source": "PoseProducer",
        "target": {
          "targetMId": "Pose",
          "targetMMId": "Geometry",
          "targetId": {
                "bin": "YAX==",
                "repr": "vertexUId"
      }
    }
  }
}
```

Listing 3.6: Example of initialisation request for a stream update of an object in the WMMC.

```
{
  "jsonrpc": "2.0",
  "id": "3",
  "result": ["Success"]
}
```

Listing 3.7: Example of initialisation response for a stream update of an object in the WMMC

Therefore, the WMMC only performs the dynamic wiring operation, but it does not decide when that is necessary. Nevertheless, the request can be rejected, *e.g.* due to an incompatibility between the pairing services;

- on one hand, a coordinator component manages other software components, *including* the WMMC; on the other hand, the coordinator component is user (specifically, it is consumer) of the WMMC to perform its coordination role;
- a coordination component can reconfigure the settings over data provenance during the runtime of the application; the WMMC *applies* these data access constraints.

3.1.6 WMMC implementation status

At the time of writing, the WMMC described in this section is under heavy development. Currently, the World Model Communication Patterns are implemented on top of the SmartMDSD software baseline RobMoSys toolchain. In the future, a similar solution can be developed using Papyrus4Robotics toolchain. While supported middlewares (such as SmartACE, OPC/UA and ROS) allows to establish the creation of new service instances *dynamically* at runtime, the SmartMDSD toolchain requires the component model being defined "statically" defined at design

```
{
  "message": {
    "source": "PoseProducer",
    "target": {
      "targetMId": "Pose",
      "targetMMId": "Geometry",
      "targetId": {
        "bin": "YAX==",
        "repr": "nodeUId"
      }
    },
    "data": {
      "bin": "YWJjZA==",
      "representation": "dproto-model"
    },
    "command": "update"
  }
}
```

Listing 3.8: Example of pose update measurement notification.

phase. Therefore, the reference implementation of a `WMMC` must know, at design time, the number of software components (`producer` or `consumer`) that will interact with the `WMMC`. This can be set at *system building time* by means of an appropriate configuration of the `WMMC`. Alternative solutions are possible and they may be investigated during the last phase of the project.

The "internals" of the `WMMC` are based on the property graph concepts and relative reference implementation, as described in Section 2.3.

3.2 Motion Stack

3.2.1 Kinematic trees and motion solvers

The property graph (Section 2.3) is a suitable format also to represent robot models, and queries on the graph can be used to perform some kinematics and dynamics computations. The mechanical structure of an articulated robot has itself the topology of a graph, which makes the representation more intuitive; however, the potential of the property graph lies in the complete flexibility of adding, removing and exploring connections that go beyond the pure representation of the static structure of the robot. For example, by relating some sensors mounted somewhere on the structure to a particular perception algorithm relevant in a given runtime context.

Of course, the extent of the computations doable by traversing the robot model graph depends on the graph itself and on the capabilities of the query engine. On the other hand, for specific execution constraints such as hard real time control loops at high frequencies (e.g. in the order of $10^2/10^3$ Hz), the runtime traversal of the property graph is simply not a viable option. In this case other policies must be employed. It is plausible to imagine, for example, a flexible combination with a code generation approach: the same mechanism of query resolution and traversal of the graph could be used to generate code rather than perform the actual computation; then, the non–composable and very specifically tailored code could be deployed and executed in a control loop.

With reference to the previous figures in Section 2.3, Figure 3.11 illustrates a possible layout of a property graph modeling a *kinematic pair*, that is, a pair of rigid bodies connected by a joint. The variations of the basic layout (on the left) are discussed in the following paragraphs. This pattern can be applied recursively to model kinematic trees and even loops. The entities

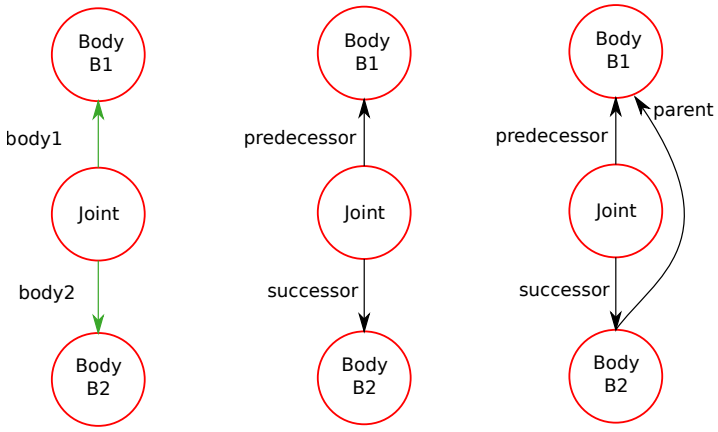


Figure 3.10: A simple property graph modeling a kinematic pair. Depending on the context, different relations can be represented, in addition to the defining ones prescribed by the model of an entity (leftmost figure, for the case of the `Joint` kind). The `parent` (and therefore `child`, not shown here) relation only makes sense in the case of kinematic trees.

involved in this model are rigid bodies (or links) and a joint. Joints are actually immaterial motion constraints between bodies, and a such they would be well represented as *relations* between the bodies[9]; however they do possess properties and must also be involved in other relations, therefore

[9] Also, any joint ceases to exist as soon as either of the two bodies does so; this is the distinctive feature of relations over primitive entities.

we represent them as entities (graph nodes). Viewing a given element of a domain as a relation or as an entity is very much dependent on the context and the scope of the modeling, and it is always possible in a property graph (see Section 2.3.1).

Typically, the `MID` property of any joint (see Deliverable D3.3) would reveal its concrete type (*e.g.* revolute, prismatic, ball-socket, etc.), otherwise a semantic tag in the form of a property would serve the same purpose. In this figure we assumed a model for joints that prescribes only two properties of type rigid body, with no ordering nor specific semantic for neither of the two; probably the bare minimum required to define the type. As mentioned before, however, the user is free to apply specific polices and therefore further refine and/or constrain the basic relations. For example, given a numbering scheme on the kinematic tree (an ordering among the links) and a joint polarity, the relations with the bodies might become the richer "predecessor" and "successor", which are meaningful also in the case of kinematic loops. When the robot is known to be a tree (no loops), then some even more restrictive relations can be introduced, like the parent-child relation between links.

Figure 3.11 extends the previous example with a few more entities. We attached a Cartesian

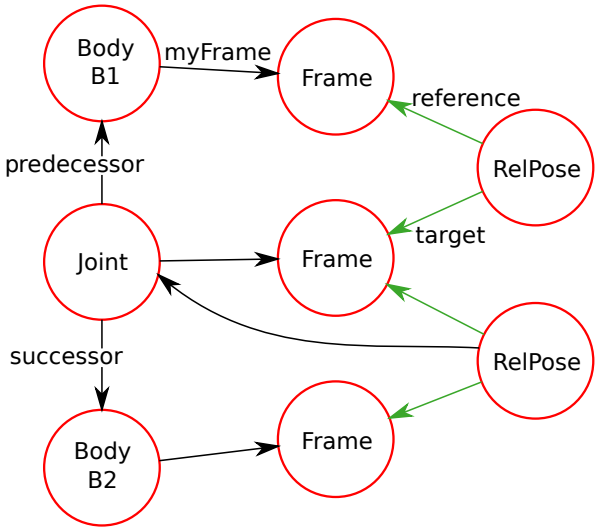


Figure 3.11: Property graph for a kinematic pair with attached Cartesian frames. Any `RelativePose` models the pose of `target` with respect to `reference`. Measurements of such relations are required to compute the forward kinematics of the mechanism. The second pose depends on the kind of joint and also on its position status.

frame to each body and to the joint, and we can assume other policies have been implicitly adopted by the user of the property graph; for example, assuming the z axis of the joint frame coincides with the joint axis itself (for one dof joints). Different layouts are of course possible, for example, the user might be interested only in a particular *point* on the second body, rather than a full frame.

A relative pose relation exists between frames (see also Figure 2.5), and once again it is represented as an entity (a graph node). Observe how the relative pose between the successor frame and the joint frame has a further relation with the joint itself, as it depends on the nature of the joint and on the numerical value of the joint position status (which is conventional). If the actual numerical value of the joint status is not available, the actual computation of B2 relative to B1 cannot be performed. However, the same traversal could be used to construct a function that

potentially does the same, when given the joint status.

Queries

Doing some computations on the robot model, *e.g.* finding the relative pose between two bodies, requires in general multiple graph traversals. In a kinematic tree, for example, the kinematic subchain connecting the two bodies of interest (like the two hands of a humanoid) must first be determined, and then traversed again to perform the actual calculus. In this example, a sequence of compositions of relative poses would yield the desired result.

Therefore, the query language (and obviously the corresponding interpreter, or "solver") must support the backtracking of a previously determined path. On the other hand, it is also possible to augment the property graph with further, custom relations for the sole purpose of easing the formulation of subsequent queries. Most importantly, we must be able to associate some specific computation (like a pre–determined function) to a specific step of the traversal.

Assuming to have established the desired path between a pair of bodies, the traversal must be configured so as to compose the current relative pose with the intermediate result. The "current" relative pose is the one between the frame of the currently visited entity (either a joint or a body), and the frame of the previous entity. Even this simple processing is not trivially specified as a query. In the Gremlin language of TinkerPop[10] it may look similar to this:

```
joint.as("joint").out("myFrame").in("target").as("theRightPose")
     .out("reference").in("myFrame")
     .in("predecessor").where(eq("joint"))
     .select("theRightPose")
```

Assuming `joint` is the joint node, this query simply fetches the relative pose that exists between the joint frame, and the frame of the body which is the predecessor of the joint. In particular:

- The first line reaches all the relative poses whose `target` is the frame of the joint, and labels them as `theRightPose`; it is irrelevant that in our example we only have one; that is not the case in general.
- The second line reaches the entities whose frame is the `reference` of the poses reached before. The traversal agents are now in nodes that do have an outgoing `myFrame` edge (*i.e.* bodies and joints, in our context).
- The third line follows an incoming edge `predecessor` (if available) and selects only the node which is equal to the starting joint. At this point, the only traversal agent that "survived" is the one that touched the pose we are actually interested in.
- The last line simply returns the pose node.

Although the queries easily get very complicated, because the underlying traversal mechanism is very *generic*, it is always possible to construct any number of DSL (Domain Specific Languages) on top of it, directly exposing concepts of the domain (such as "the reference frame of the predecessor") with a simpler syntax. For example, it would be easy to introduce an higher level command like

```
joint.poseWRTPredecessor()
```

doing the same thing as the example above.

[10] Apache TinkerPop is a property–graph computing framework. See http://tinkerpop.apache.org/.

Available software

Currently, at M30, a tool for code generation for kinematics solvers, developed in the context of RobMoSys and another H2020 project, ESROCOS[11], is available. At the current stage, the tool is tailored for code generation (rather than online computations) and hard real–time execution in control loops.

It perfectly conforms to the concepts of property graph and graph traversal – especially because a kinematic tree is topologically a graph, as said before – but at the moment it is not based on a generic property graph mechanism as illustrated in the previous sections. The queries, the traversal and the associated computations (which in this case are ultimately about generating code) are performed by a tailored implementation, *i.e.* in a traditional way. The tool ("KinGen", from now on) is described in detail in an academic publication [8] and on a RobMoSys wiki page[12], therefore here we only provide an overview of its main features.

The original ambition of KinGen is to explicitly expose all the choices that uniquely identify the semantics of an implementation of a solver, by requiring the user to provide them. The declarative specification of the solver (*e.g.* end-effector forward kinematics), the programming language, the mathematical representation of rotations, etc., should all be composed symbolically to let the tool generate code exactly matching the configuration.

To achieve this goal, it is effective to introduce intermediate models of the solver, lying between an abstract, purely functional declarative specification on one hand, and the very concrete, "grounded" computer code on the other. Intermediate models allow to fix some choices leaving others open for subsequent configuration. KinGen is relying on a format for one of such intermediate representations, an imperative model where the sequence of *primitive* operations is explicitly represented, with no other details. Figure 3.12 illustrates that the tool is in fact composed of two programs, a generator and a compiler. The generator takes a robot model and an imperative request, and generates the imperative model of the requested solver. The compiler can then take this model, applies some grounding choices and emit actual computer code.

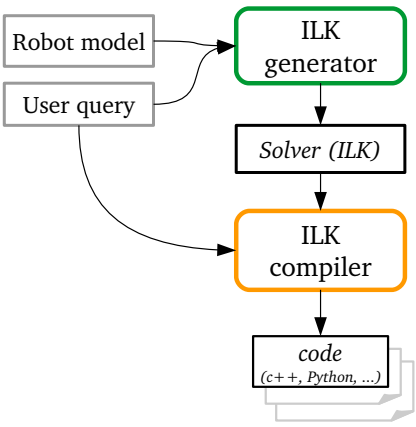


Figure 3.12: Overview of the KinGen tool. ILK is the name of the format to host imperative models of robotics solvers.

The ILK-Generator is the component that embeds robotics specific knowledge and transforms an imperative request (*e.g.* find the end–effector position relative to the base) to a sequence of semantic operations that model the right computations. To do so, the generator maintains a local representation of the robot model, in the form of a graph, and traverses it a few times to accumulate the information needed to generate the solver model. For example, to determine the right sequence of Cartesian frames whose relative poses have to be composed.

It is possible to reimplement a similar functionality based on a generic property graph traversal engine, where the computations associated with the traversal would be about constructing the imperative solver model (this model could then be compiled and deployed, but also interpreted online – but this aspect is outside the scope of this document). Such an implementation is certainly more complex than a dedicated solution like the ILK-Compiler, but on the other hand

[11] https://www.h2020-esrocos.eu/
[12] https://robmosys.eu/wiki/baseline:environment_tools:kin-gen

it can exploit the flexibility of a property–graph–based robot model; that is, the capability of attaching dynamically further information to the model, establish new relations, and be able to refer to them in subsequent queries.

For example, we might have a manipulator controller using an inverse dynamics solver for the feedforward control commands; once the manipulator grasps an object, the user might want to reconfigure the solver so as to return also the position of its end–effector (in addition to the joint forces), as that data can be used to estimate the position of the object. In a more complex scenario, with two manipulators grasping a heavy tool, for some kind of task, the property graph representing the two robots could be manipulated to represent a new, virtual individual robot with a kinematic loop; then, specific solvers for this particular case could be constructed, and then used for the control of the motion of the tool.

3.2.2 Other existing components and tools

A wikipage keep track of "ready to be used" component models, existing at M30, which some of them were already available in Y1. This list is expected to grow in the incoming months, to support both ITP and Pilot cases.

3.2.3 1st call of Integrated Technical Projects Results

Contributions and additions to motion stack functionalities and tools are not solely developed by the RobMoSys consortium. Among the 1st wave of ITPs, the project "Extension of Intrinsically Passive Control model and integration in the RobMoSys ecosystem" (**EG-IPC**) also contributed to the motion stack development. In particular, the project developed and demonstrated the RobMoSys approach in the context of teleoperation with haptic feedback. As a result, the ITP has proposed a relation between the RobMoSys meta-models and energy-based control, which is a model-driven loop control strategy, including bond-graphs and passity layer concepts. This ITP project was executed in strong collaboration with the coach from KUL, leader of WP3 on basic building blocks for motion, perception and world model stacks. Further details can be found in the EG-IPC deliverables, and in the wikipage of the project. In the call for 2nd wave of ITPs, instrument #2 proposed two topics to futher contribute to the motion stack, namely mobile navigation and manipulation topics. Therefore, the RobMoSys consortium expects to keep the development of basic building blocks in strong collaboration with the RobMoSys community.

```
CommObject PositionMeasurementXYZ {
   x : Double
   y : Double
   z : Double
}

CommObject PositionMeasurement3 {
   data : Double[3]
}
```

Listing 3.9: Examples of communication objects models for `Position` measurement with SmartMDSD DSL.

3.3 Perception Stack

3.3.1 Perception components and functionalities

In this section, developed software components for perception (or under development) are briefly listed, with few extra details on the conceptual design and motivations. Where possible, the description of the components is independent from the component-based tool used. The current development targets to the SmartMDSD toolchain, which is a RobMoSys software baseline. However, analogous components are expected to be developed with Papyrus4Robotics, which is also a RobMoSys software baseline. For those components originally developed using MFCF (see Section 2.4), the porting from one baseline to another is expected to be trivial for the component supplier.

Sensor fusion component

Sensor fusion consists in combining data from multiple sensors to reduce uncertainties on a specific measurement with respect to an individual measurement. Therefore, sensor fusion is to be considered another fundamental perception building block of any robotic application. On a component-based architecture, the source of a sensor data is always modelled as a software component, and the data is provided by means of a service. The service model is the first point of *software variability*, since a different combination of i) a communication pattern choice and ii) a communication object model choice can be employed to share the semantically equivalent data. As an example, a measurement of a `Position` can be encoded in many ways. Listing 3.9 illustrates an example of enconding expressed with the Communication Object DSL of SmartMDSD toolchain, while Figure 3.13 shows an alternative representation by means of the Papyrus4Robotics tool. The latter also allows to attach units of measurement to the values.

In the same way, another point of *software variability* regards the service models employed in the software components that receive the result of the sensor fusion algorithm.

Software variability is a major technological barrier to achieve multiple software components to interact between each other (*i.e.* horizontal composition). This is typically solved by means of:

- definition of **standards** and their wide adoption; RobMoSys is elaborating those standards (service models, communication object models and their semantics) to that aim;
- **reconfigurability** of the software component to handle different service model choices.

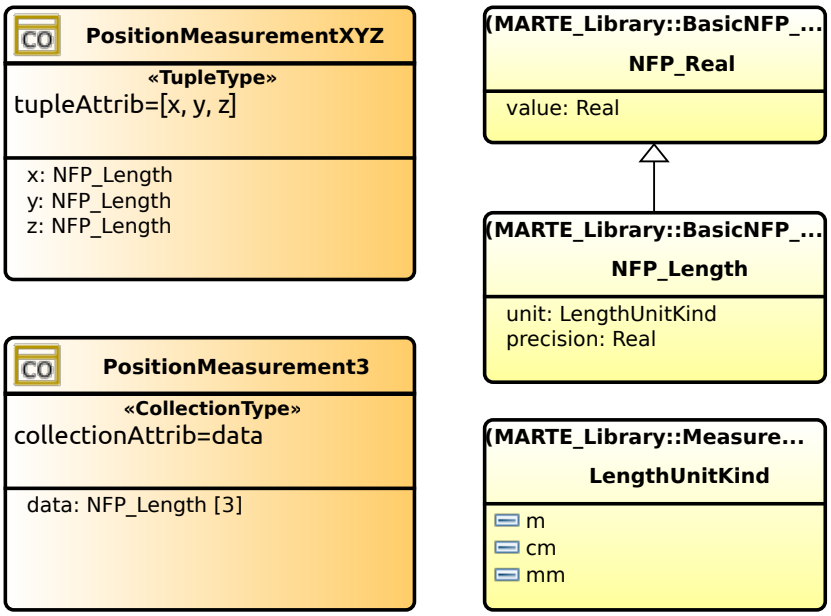


Figure 3.13: Example of communication object models for `Position` measurement with Papyrus4Robotics. On the left, the communication object definitions; on the right, unit of measurements of the value expressed with MARTE library (OMG standard).

Nevertheless it is not trivial to develop a generic *sensor fusion component*, due to the *software variability* problem. In fact, the sensor fusion component model cannot be designed at priori, since it depends on the components that will interact with it.

Therefore, the role of the sensor fusion component is two-fold:

- **functional**: to perform a sensor fusion algorithm, thus embedding an algorithm, and representing the managed resources needed for the computation;
- **non-functional**: semantically speaking, all incoming data and outcoming data of a sensor fusion component must have the same *semantic meaning*. However, the software variability related to the adoption of different standards limits the software composability. Therefore, a sensor fusion component acts as an **adaptor**, to mitigate the difference between service model choices.

Figure 3.14 shows and describes the conceptual design of the envisioned sensor fusion component (toolchain-agnostic), which peforms the sensor fusion algorithm, dispatching the result on different services while the data semantics is conserved. In this way, the system builder will be able to easily re-use this sensor fusion component in different applications. This influences also the re-usability of "sensor" components, which can now be used as long as the data they provide has the expected semantic meaning.

Figure 3.16 and Figure 3.16 illustrate the SmartMDSD version and the Papyrus4Robotics version of such a component, respectively. Currently, the number of inputs, outputs and their services is configured at design time of the component. This is due to a current technical limitation of the tool, since it is not possible to reconfigure the component model at runtime or at deployment time. This may be improved in the final phase of the project, aiming to the original conceptual

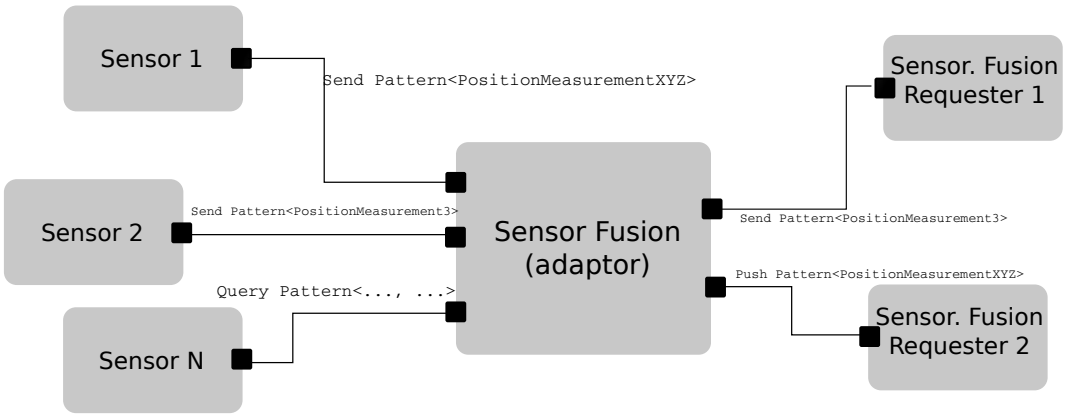


Figure 3.14: Concept of the sensor fusion component as "adaptor", in a "toolchain-agnostic" form. Multiple components are source of data having same semantics (*i.e.* measurement of a `Position`), but with different service choices (*variability point*). The sensor fusion component computes a refined measurement, which is then served to component "requester" of the data, maintaining the same semantics but with the desired service choice.

design of Figure 3.14. Nevertheless, it is not to be intended as a limitation with respect to the RobMoSys approach, as long as a *component data sheet* is provided with the component model and its implementation.

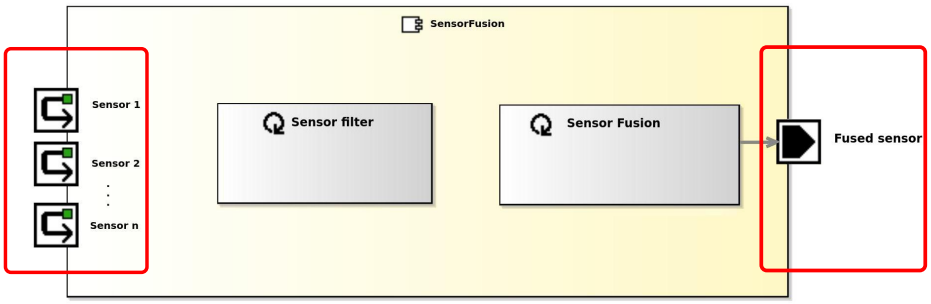


Figure 3.15: SmartMDSD graphical illustration of the proposed sensor fusion component model conforming to the concept of Figure 3.14. The red boxes highlight list of services being available post-configuration.

As a case of study for sensor fusion, estimation of `Pose` and `Twist` geometric relations among rigid bodies (*e.g.*, an object and a reference object, both in the scene).

The sensor fusion component embedding a classical Kalman filter algorithm [14] is considered. Since this algorithm is well-known, this sensor fusion component serves as a practical "hands-on" example, as a basis for dissemination and development of more complex sensor fusion algorithms.

The datasheet of this component must include sensor confidence and the sensor threshold time parameters. The sensor confidence parameter defined for each sensor, to indicate which sensor data is more reliable. The sensor threshold time denotes the time the pose/twist is still confident.

Relation between sensor fusion component and the World Model Mediator Component

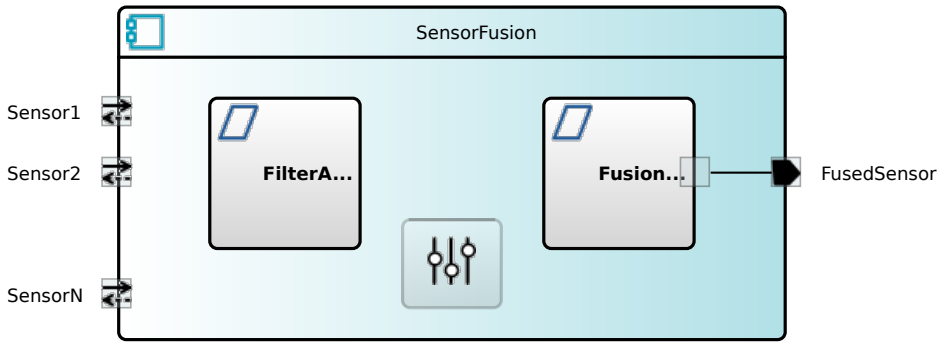


Figure 3.16: Papyrus4Robotics graphical model of the proposed sensor fusion component model, conforming to the concept of Figure 3.14.

The adaptability to multiple services definition, requested by the sensor fusion component previously described, is analogous to the adaptability requested by the `WMMC` (see Section 3.1.4). Generalising, this service adaptability is a desired feature in RobMoSys and it can be applied to other software components as well. However, this adaptability recommended for those software components designed for being *context-independent*; the others can profit from it indirectly, that is, by means of the adaptability offered by the "core" ones.

Moreover, it is important to notice that sensor fusion algorithm does not have to be hosted necessarily in a separated component. In fact, the algorithm can be loaded as post-process service into the `WMMC`, where are component that feed the sensor fusion component are now `producer` components of the `WMMC`. How to separate the algorithmic part and being able to decide in which component to host a specific computation, at design time of the system architecture (or at deployment time), is still under investigation.

Model object features for enabling "fast" object recognition

Object recognition is one of those functionalities that are very common in robotics applications. The robotic system must be able to detect an object in the environment and select those that are subject to the interaction in the described task specification.

To this end, the robot must have a valid model of the object, and such knowledge must be stored and shared among multiple perception components. This is the role of the *world model* and its runtime component (*i.e.*, `WMMC`) as described in Section 3.1.4. However, *how is the object model defined? which is the "minimal" set of properties that an object model should contain? and how object features can be exploited for object detection and recognition?* During Y1/Y2 of the RobMoSys project, the RobMoSys consortium struggled to find a generic answer to these questions, which is not a trivial task, since the answers are very dependent on the context and the domain of the robotic application. With reference to Section 2.3, the property graph approach and tool helps to extend and to compose existing domains, relieving the need of finding *"the"* object model to be conformant to. As a consequence, in this section we limit the focus on modelling those features that enables the implementation of "fast" object recognition algorithms.

In this scope, the envisioned object model must be composed by at least the following elements:

- **state**: refers to the geometrical spatial relation of the object with respect to other objects in the scene, such as `Pose`, `Twist`, but also discrete spatial representations and their topological relations (*i.e.*, object located in a room, a box, etc; object *on top of* a box, etc);

- (invariant) **properties**: properties refer to physical characteristics that describe the object, *e.g.* shape of an object, and they can be variant or invariant during the runtime of the application. For example, in some applications a `color` can be considered an invariant property of a physical object; in other applications a `color` may vary, *e.g.* during a painting task;

- **features**: features of an object are those specific elements that have been observed from the object being in a specific state.

In literature [9][10], object recognition algorithms based on properties are typically faster than those that exploit features, but less accurate. Therefore, properties such as `color` and `shape` are considered to accelerate or improve the object detection and recognition process.

However, there is no generic algorithm that suits the detection of any properties, and the properties definition is strongly application-dependent. To this end, the design of object recognition component must ensure a certain degree of re-configurability: the application builder shall be able to configure the algorithm being in use, by selecting specialised functionalities and combine them in a proper computing pipeline. To this end, this component would be a test case for the MFCF framework (see Section 2.4.3).

Together with properties detection, also features detection algorithms are considered. Object features are typically divided into 4 types: 2D/3D, and local/global features.

The envisioned object recognition component will support at least the following algorithms:

- LBP (2D global feature);

- SURF [1](2D local feature);

- FPFH [13](3D global feature).

The reference implementation of those algorithms is under development, and it makes use of several well-known third party libraries, such as OpenCV, PCL and MRTP libraries.

To validate the object detection and recognition component prior to its usage within the Pilot cases, the system depicted in Figure 3.17 was developed, with the support of the SmartMDSD toolchain.

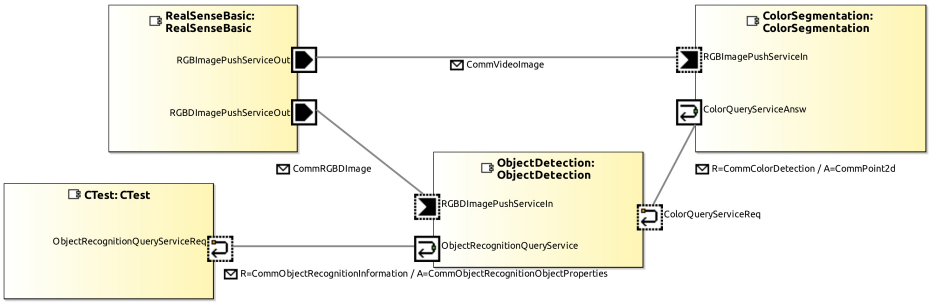


Figure 3.17: System architecture of the object detection and recognition benchmark.

This system can be used as a basic template for enabling "fast" object recognition features, and it consists of:

- Driver component: a component "RealSenseBasic" that implements the drivers to interact with the sensor "Intel RealSense SR300";

- Perception components:
  - Object Detection
  - Color segmentation
  - Shape Detector
- A test component that acts as *coordinator* to activate the object detection and recognition system.

Among the Pilot cases in development, the first target of the object recognition component is the Flexible Assembly Pilot (Siemens). In this pilot, the manipulation activity strongly requires fast object and recognition functionalities. In this scenario, color segmentation will be applied, together with a classificator exploiting user-defined features.

Object tracking

Another relevant perception building block is object tracking. Object tracking consists to locate the state of the object (*e.g.*, its relative `Pose`, `Twist`, etc) while moving. Tracking objects is very relevant for visual servoing applications, to quickly react to moving obstacles, or to fetch objects in motion.

A component embedding object tracking functionalities is planend to be developed. As a prerequisite, this component will need to store previous object detection information: this can be done by interacting with another component, *e.g.*, with the `WMMC`. In fact, there are numerous information that can be used to perform an accurate and fast tracking: kinematic information, images, point clouds and previous relative poses. Internally, the object tracking will make use of a Kalman filter algorithm. Moreover, this component will not be only dedicated to object tracking, but it will compute as well:

- prediction of object's future location;
- correction of the prediction based on new measurements;
- reduction of noise introduced by inaccurate detections;
- data association problem of multiple objects.

Object prediction

Along with the object tracking component, an object prediction component will be developed. Object prediction consists to predict the state of a moving object. The prediction of an object is useful for many applications and can provide information to the motion stack with to work with Model Predictive Control(MPC) applications. Also, it can be useful for some pilots in applications of the avoidance of mobile objects.

4. Interaction with the pilots

4.1 Introduction

The goal of this last chapter is to highlight the role of the ongoing efforts (modelling, tool design and implementations) to support concrete, industrial-relevant cases, starting from the defined pilot cases. In this perspective, design and priorities have been driven by Pilot cases needs, considering the previous Deliverable D4.1 and its extension (Deliverable D4.2).

During Y1 and Y2, the RobMoSys consortium had many interactions and technical discussions regarding the requirement and the design of each stack. For example, the role of the world model (see Section 3.1) has been defined as one of the most important in every Pilot case.

Finally, this chapter aims to show the requirements for the motion, perception and world model stacks, and the expected benefits from the current approach related to the exploitation of the modeling efforts (knowledge) by means of the concrete tools under development and illustrated in this Deliverable. These requirements can be considered as well for the definition of the KPI, and as concrete suggestions for standardisation for the Robotics Community,

For the sake of brevity, only the requirements and the impact on the current development on two pilot cases are discussed in the following pages. More details can be found in the deliverables regarding the pilot-cases (D4.2).

4.2 Flexible Assembly Cell (Siemens)

During assembly within an assembly cell, workpieces are manipulated and fitted together to ultimately produce a desired configuration of the workpieces. In a classical assembly cell, the pose of each workpiece is accurately specified a priory during cell commissioning and task programming, jigs and fixtures specially constructed for the workpieces and the task are utilized to guarantee that the wokpieces are precisely located at the specified poses, and the motions of the mechanical parts of the cell needed to fit workpieces together are specified a priory based on the specified poses of the relevant workpieces. One of the challenges in flexible assembly is to relax the requirement that workpieces have to be precisely positioned and that these poses have to be accurately known in advanced before task execution.

Our flexible assembly cell uses 2D and 3D cameras to determine the pose of the workpieces within the cell during task execution. And the motions of the robot arms and grippers are computed online based on the actual pose of the workpieces. To be able to compute collision free motions, the system needs to have an accurate and consistent geometrical model of itself and of the workpieces including their actual poses. The following list describes some of the most important features needed from the world model required for manipulating workpieces:

- The world model needs to represent the state of the robot and collision objects. The state consists of the pose of the objects plus some appropriate probability distribution describing the uncertainty associated to the pose. Furthermore, different uncertainty models should be represented, such as uni- or multimodal Gaussian or particle representations. The state should also include some meta-data such as the time stamp of the last state update, and the information source. For example, for calibration and sensor fusion, relevant properties of the information source such as the calibration parameters of a camera, should be an (optional) part of the state.

- The world model should be updatable by multiple instances of different modules at different rates. The robot controller should be able to update the state of the robot with very high frequency while at the same time the perception system should be able to update the state of the collision objects at a rather lower frequency.

- The world model should be accessible by multiple instances of different modules at different rates.

- Modules should be able to retrieve the state of objects based specific state or combination of state properties. For example, modules should be able to retrieve all collision objects within a specified region in space.

- The world model should be able to interpolate the state of the objects in time. For example, when querying the pose of an object for a specific time where no data existed, the world model should be able to estimate the pose of the object for that specific time from the existing data. Ideally, the world model should consider information about the velocity, acceleration, etc. of the object for interpolation and maybe even extrapolation.

- It should be possible to clone the world model at a given state and update and query the cloned instance without interfering with the original world model. For example, for planning, the system should be able to take the current state of the world model and simulate changes in it to determine the best next action.

- The number and type of modules that will be updating and querying the world model, is not known in advanced and can change during task execution. For example, for detecting a certain type of objects, a different perception pipeline might be needed. The required modules are started and stopped on demand.

- It should be possible to update and query the world model using different reference frames. The perception system should be able to update the pose of a detected object in its relative reference frame, without having to know about kinematic of the rest of the system. It should also be possible to switch between different coordinate systems, such as polar and Cartesian.

- It should be possible to dynamically add and remove spacial constraints between spatial objects. For example, once an object is attached to the robotic manipulator (e.g. the object is grasped), moving the manipulator will also move the attached object. Once the object is released, the spacial constraint is eliminated.

- The world model should implement a signaling mechanism to notify registered modules about specific changes in the state of the world model. A module may hook a trigger to a certain variable and the world model will send and update / interrupt to the module upon change. Ideally this data channel is real time capable and deterministic, such that safety critical information (e.g. human enters workspace) can be communicated via the world model (component).

In addition to the above listed requirements for general object manipulation, the following features are required for task specification and execution:

- It should be possible to represent different abstraction levels for the data in the world models. For example, it should be possible to query the world model for all objects on a previously specified area such as "storage magazine" or "working surface".
- Process image:

Finally, regaring the deployment of the system components:

- The world model should be accessible from multiple disfferent components distributed over different devices. For example, the world model should be accesible from the preception subsystem running on a PC other than the one where the motion planner and robot controller components are running. Both PCs are connected over standard ethernet network.
- Communication: not every communication pattern possible (PLC)

4.3 Human-robot collaboration for assembly (CEA)

CEA pilot aims at realising a collaborative pick and place task with an human operator. The human operator teaches the task to perform and how to approach the objects, such that the robot is able to perform the task in fully autonomy, or in the presence of a human operator.

The goal of CEA pilot is to ensure system robustness, both at design time and at run-time. To do so, the knowledge from the environment, the agents capabilities and their affordances are modelled in RobMoSys:

- at design time, the knowledge is exploited to add constraints over world model properties. For example: to allow or to forbid the actions execution on the world object instances; to check task consistency; and most importantly, to validate safety properties. In practise, the world model at design-time is comparable to an ontology, where all the relevant features of all the environment elements are modelled;
- at run-time, to monitor the system execution and to ensure that the rules defined at design-time are not violated. At run-time, a world model component acts as a mediator, responsible for collecting data from all the environment objects (perception), then used for executing the task.

In the following, most of the important features needed from the world model are enumerated, as requirements to ensure system robustness:

- It should be possible to represent the objects properties and the agents skills in the world model;
- It should be possible to represent the effect of an action on the executing agent (*e.g.* gripper not empty and on the manipulated object (*e.g.* object in gripper);
- The world model should be able to indicate the logical state of all the objects and the agents at a given time;
- The world model should allow to enable or disable affordances' interfaces attached to the objects, based on the context;

- The world model should be able to take into account the safety constraints that has to be checked when the robot manipulates an object.
- The world model should implement rules that are conformant to the robot capabilites. For example, a robot cannot manipulate an object with a higher weight than its payload;
- The world model should be aware of all the interactions in the system.

4.4 Intralogistics Industry 4.0 Robot Fleet Pilot (HSU)

This pilot is about goods transport in a company, such as factory intra-logistics. It showcases the ease of system integration via composition of software components to a complete robotics application. A particular focus is put not only on the functional scenario of this pilot, but on easing its development and maintenance through its lifecycle. More information on this pilot can be found in D4.1 [15] and in the RobMoSys Wiki[1].

Figure 4.1: The Intralogistics Industry 4.0 Robot Fleet Pilot

4.4.1 Requirements on RobMoSys Methodology

The pilot is intended to demonstrate several benefits of the RobMoSys methodology. This section highlights those that are specific to this deliverable. It elaborates on the requirements and impact that can be demonstrated with this pilot. Among other goals, the pilot aims to:

- Demonstrate the ease of system integration via composition of previously developed building blocks.
- Demonstrate the exchange of software components to address new needs and e.g., add capabilites to robots.
- Demonstrating the adaptation of the production flow at run-time by changing the software configuration only.

[1] https://robmosys.eu/wiki/pilots:intralogistics

- Demonstrate the use of the RobMoSys flexible navigation stack, as an example of elements in composition Tier 2.

Already now, the pilot successfully demonstrates the easy composition of previously developed software components (see service-based composition[2]). While this significantly reduces the time on system-level development, it assumes the existence of software components. Going down to this level, the pilot stresses the need for providing software components and also composing their internals by the means of functional composition as described in this deliverable. A lot of best-practices from service-based composition of software components can be applied to functional composition as well. Improving the functional composition to come up with software components will thus directly improve the composition of software-components on the system-level.

The initial composition and especially the run-time adaptation of components in systems (and even more: robots in fleets) triggers a huge need for managing world model knowledge. Several components and robots act within the same world and share efforts. For example, recognition and manipulation as well as handling of items between two robots requires a common understanding of the world model. Especially in run-time adaptation to change the production flow, it is impossible to manually change the configurations or even re-program, for example, the world-model expectations of a component or whole robot to align one world-model to another.

4.4.2 Current state

The basic pilot scenario is already available[3]. The pilot is built using the SmartMDSD Toolchain[4] by composing previously developed software components from the RobMoSys Software Baseline. It already now uses the RobMoSys composition structures which formalize a lot of Architectural Patterns[5]. More architectural patterns and best practices are already applied within the pilots, but are manually encoded and thus not formally structured. They are therefore not accessible in a systematic way to improve composability. For example, the management of distributed world models and the synchronization over global IDs in the flexible navigation stack[6]. RobMoSys is about to generalize and formalize these concepts to make them accessible through the RobMoSys methodlogy. The insights on the manually applied methodology has already contributed to this deliverable and will contribute to future work around the RobMoSys motion, perception and world-model stacks.

[2] https://robmosys.eu/wiki/composition:service-based-composition:start
[3] Videos at https://robmosys.eu/wiki/pilots:intralogistics
[4] https://robmosys.eu/wiki/baseline:environment_tools:smartsoft:smartmdsd-toolchain:start
[5] https://robmosys.eu/wiki/general_principles:architectural_patterns:start
[6] http://www.servicerobotik-ulm.de/drupal/sites/default/files/2012-WillowGarage-public.pdf, slide 29

Bibliography

[1] Bay, H., Ess, A., Tuytelaars, T., and Van Gool, L. Speeded-up robust features (surf). *Comput. Vis. Image Underst. 110*, 3 (June 2008), 346–359.

[2] Blumenthal, S., Bruyninckx, H., Nowak, W., and Prassler, E. A scene graph based shared 3d world model for robotic applications. In *2013 IEEE International Conference on Robotics and Automation* (May 2013), pp. 453–460.

[3] Blumenthal, S., Hochgeschwender, N., Prassler, E., Voos, H., and Bruyninckx, H. An approach for a distributed world model with qos-based perception algorithm adaptation. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Sep. 2015), pp. 1806–1811.

[4] Bouzeghoub, M. A framework for analysis of data freshness. In *Proceedings of the 2004 International Workshop on Information Quality in Information Systems* (New York, NY, USA, 2004), IQIS '04, ACM, pp. 59–67.

[5] Chitta, S., Sucan, I., and Cousins, S. Moveit![ros topics]. *IEEE Robotics & Automation Magazine - IEEE ROBOT AUTOMAT 19* (03 2012), 18–19.

[6] Foote, T. tf: The transform library. In *IEEE International Conference on Technologies for Practical Robot Applications (TePRA)* (April 2013), pp. 1–6.

[7] Foote, T. tf: The transform library. In *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on* (April 2013), Open-Source Software workshop, pp. 1–6.

[8] Frigerio, M., Scioni, E., Pazderski, P. P., and Bruyninckx, H. Code generation from declarative models of robotics solvers. In *Third IEEE International Conference on Robotic Computing (IRC)* (Feb 2019), pp. 369–372.

[9] Koschan, A., Bericht, T., and Algorithmen Fur, U. Colour image segmentation: A survey.

[10] Martínez, L., Loncomilla, P., and Ruiz-del Solar, J. Object recognition for manipulation tasks in real domestic settings: A comparative study. In *RoboCup 2014: Robot World Cup XVIII* (Cham, 2015), R. A. C. Bianchi, H. L. Akin, S. Ramamoorthy, and K. Sugiura, Eds., Springer International Publishing, pp. 207–219.

[11] Niemueller, T., Lakemeyer, G., and Srinivasa, S. S. A generic robot database and its application in fault analysis and performance evaluation. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2012), IEEE, pp. 364–369.

[12] Ravichandran, R., Prassler, E., Huebel, N., and Blumenthal, S. A workbench for quantitative comparison of databases in multi-robot applications. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Oct 2018), pp. 3744–3750.

[13] Rusu, R. B., Blodow, N., and Beetz, M. Fast point feature histograms (fpfh) for 3d registration. In *Proceedings of the 2009 IEEE International Conference on Robotics and Automation* (Piscataway, NJ, USA, 2009), ICRA'09, IEEE Press, pp. 1848–1853.

[14] Sasiadek, J., and Hartana, P. Sensor data fusion using kalman filter. vol. 2, pp. WED5/19 – WED5/25 vol.2.

[15] Scioni, E., and Meyer-Delius Di Vasto, D. Deliverable 4.1: First report on pilot cases, 2017.

[16] SHERPA. Smart collaboration between humans and ground-aerial robots for improving rescuing activities in alpine environments. http://www.sherpa-project.eu, 2014. Last visited May 2014.

[17] van de Molengraft, M. The RoboEarth project. http://www.roboearth.org/, 2011.

5. Annex

The annex section of this Deliverable presents a pre-print version of a submitted academic publication to illustrate further details on the `dproto` DSL and the automatic datatype conversion tool "`dconv`". The paper describes the status of the tool at M24.

Semantic Annotations for Geometric Relations Supported by a Model-based Datatype Conversion Tool

Enea Scioni*,†, Marco Frigerio†, Herman Bruyninckx†,‡

***Abstract*— This paper advocates the research hypothesis that robotics software should be developed by introducing formal semantic representations of all data and operations already in the design phase. Hence, Interface Description Languages (IDL) alone do not suffice, but models and tools are needed to let developers semantically annotate software interfaces, starting from the data types. The first contribution of this work is a semantic annotation language for data types and their relations, including references to existing domain ontologies. A second contribution applies the language to the core geometrical entities and relations for robotics, as a first step towards standardisation. The last contribution introduces tooling software, `dconv`, to help software developers to introduce semantic annotation in their coding workflow. `dconv` can generate static conversion code between data types with the same semantics but different digital encodings, and it can also perform these data type conversions dynamically, at runtime.**

I. Introduction

Software design and development of robotics applications consists of integrating functionalities of heterogeneous nature, often shipped as a set of libraries or software components. Each implementation uses its own internal data types, often optimised for a specific purpose (*e.g.,* computing, storing, communicating). During the integration process, *data type conversion* is one of the unavoidable pieces of "glue-code" that must be provided. For example, embedding a functionality in a component-based middleware requires the conversion from a communication object employed by the middleware to an internal data type used in the embedded library (and *vice versa*). This applies also within the same software component, when integrating libraries with different *Application Programming Interface* (API). To ensure that the semantics associated with the values remains unaltered is a tedious and error-prone process, whenever developers have only access to informal documentation of the code that provides incomplete descriptions of all choices that have been made: physical units, ordering of variables in a composite data structure, selection of any of the 24 different Euler angle representations, etc.

A common solution is to establish a *"standard"* choice for the data types, expressed by an *Interface Description Language* (IDL) provided by a middleware framework. The most prominent example in robotics is the ROS [1] middleware, with ROS-IDL definitions contained in `geometry_msgs`, `sensor_msgs`, etc. [2]. However, this approach has several shortcomings:

- an IDL describes only the *digital data representation* of the value, that is, how the values are represented digitally, with the purpose of describing the knowledge required for the serialisation process or storage of the values in-memory. An IDL does not express the *meaning* of the values, their constraints[1] and how those are interpreted (*e.g.,* physical units and other choices of representation). This additional *domain-dependent* knowledge lies outside the modelling capabilities of an IDL, and it is mostly relegated to the (often incomplete) documentation;
- it is a common practise to associate a mnemonic identifier to the IDL model (*e.g.,* the data type name). In this case, the only means of checking the compatibility of a pair of values is by matching the identifiers of the corresponding IDL models: the semantics are assumed compatible if the values of the two identifiers are the same. There is no opportunity to establish true semantic equivalence, which might hold even if the identifiers do not match. This is the case for an orientation: if a different coordinate representation choice is used (*e.g.,* quaternion and RPY-angles), the identifiers do not match but the values are compatible. In addition, the chosen identifier is not always formally correct, or it does not fully express the semantics of the represented value, leading to ambiguities in the semantic interpretation;[2]
- almost every middleware in robotics uses its own IDL. A standard based on an IDL works only in a homogeneous system. Composition with other software components is limited to the middleware being used.

In short, an IDL is not suited for capturing the semantic meaning of values, and it should not be used to define standards outside its domain, *i.e.,* the *digital data representation*.

This paper advocates the research hypothesis that the semantics of the data handled in a robotics application must be a first class element of the software design. This is the first step towards formal models of the semantic of software

* Corresponding author, `enea.scioni@kuleuven.be`
†Dept. of Mechanical Engineering, KU Leuven, Belgium.
‡Dept. of Mechanical Engineering, TU/e Eindhoven, the Netherlands.

[1]Most IDLs can constrain a single value, *e.g.,* by bounding it between two values. However, they don't describe the constraints among multiple values. For example, the values of a tuple (x, y, z) representing a *versor* are subject to the unitary norm constraint.

[2] For example, the popular standard `geometry_msgs` in ROS represents a point with a tuple of values (x, y, z) (`geometry_msgs/Point`). However, a point is a geometric *entity* and it cannot have a tuple of this kind as attribute. Instead, the same tuple represents a geometric *relation*, namely a position, which is a relative concept that requires a reference point (or frame) to be defined [3]. Finally, the tuple is valid only under a particular coordinate representation choice (*i.e.,* Cartesian).

interfaces, and all protocols and dialogues between robot systems that make use of such data.

The semantic of the data must be identified within a reference domain (*e.g.,* relative position in the geometry domain), independently from any existing, concrete, digital data representation. Conversely, existing digital data representations (possibly IDL-based) shall be *annotated* with a reference to the semantic specification. Such explicit specifications, in a computer-processable format, not only provide better documentation but also enable the development of automatic validations and conversions, directly in the user software or by means of dedicated tools; they ultimately lead to higher software quality.

The concrete outcome of the approach described in this work is "`dconv`" [4], an automatic data type conversion tool that exploits the semantic annotations and their relations. This tool is designed to support both *i)* the generation of static code that performs the conversion, to be included in the user-code at development time, and *ii)* an engine to perform the automatic conversion dynamically, at runtime. Summarising, this work proposes:

- a minimalistic, user-friendly domain-specific language (DSL) to manually specify semantic annotations on top of existing digital data representations (see Sec. IV);
- an annotation schema in the context of the *geometry* domain (see Sec. III), which is at the core of any robotics application;
- a reference implementation of a data type conversion tool that exploits the annotations encoded in the proposed language (see Sec. V).

II. Related Work

Any middleware framework in robotics allows custom data type definitions to share data among software components, which are often described by an IDL. Having this is not only a technical requirement, but also a primary language to design an application within the middleware ecosystem. Noticeable examples are: *ROS* [1] (which uses its own *ROS-IDL*); *SmartSoft* [5], [6], and its *Communication Object DSL* [7]. The Architecture Analysis and Design Language (AADL) [8] also allows to define data types as part of a component interface, while toolchains like *TASTE* [9] relies on *Abstract Syntax Notation One* (ASN.1) [10] as IDL. There are also several IDLs available outside of the robotics domain, with focus on different technical aspects and purposes. The Google's Protocol Buffer [11] and Flat Buffers [12] introduce a DSL to describe a digital data representation, while the implementation is focused on data serialisation and in-memory storage, respectively. Another related project is *HDF5* [13], a filesystem for scientific computing dedicated to store large, distributed datasets. HDF5 allows to include meta-data and attach semantic information on each entry of the dataset. However, all of the above focus on modelling a digital data representation, independently from an application domain, while the semantic value attached to the data is not considered as a primary scope.

The role of ontologies [14], [15] is to provide those semantics descriptions, and the relations between them. In literature, ontologies to define quantities, dimensions and physical units have been investigated already, *e.g.,* the *QUDT* ontology from NASA [16] and the *Ontology of units of Measure and related concepts* (OM) [17]. This work does not propose a novel ontology in this direction, but it aims to link existing ontologies with existing digital data representations. The DSL proposed in Sec. IV is an effort in this direction, but alternatives to encode the same concepts exist. For example, JSON-LD [18] is a general-purpose W3C recommendation widely used in *web* technologies, that aims to link (JSON) data and its meta-data together. However, interoperability with JSON-LD is not subject of investigation in this work.

This paper focuses on the geometric domain, and its modelling of geometric entities and relations as in [3].

Finally, this work also concerns the data representation synthesis problem, a well-known problem in computation and data structures theory [19], [20], [21]. In particular, this regards the realisation of a reference implementation. As shown in Sec. V, in this work concrete data structures and relative conversion operations are generated from the analysis of a graph-based knowledge space, which is defined from a set of semantic annotations (see Sec. IV).

III. The Geometric Domain

To annotate the semantics of a set of values, the first step is to define a domain of reference that provides contextual information. In general, a domain model consists of a list of valid *entities* that belong to the domain, and their *relations*. Without loss of generality, this work considers a simple model of the geometric domain, at the core of any robotics application, as a case of study.

The entities and relations of the geometric domain correspond to the mathematical properties of the Euclidean spaces E^3, $SO(3)$ and $SE(3)$: entities like (spatial) point, vector, orientation frame, displacement frame, etc.; and relations like position, orientation, pose, velocities, etc. *Composition* rules and further constraints must be modelled as well. For example, a displacement frame can be expressed as a composition of a point and an orientation frame. In the same way, a pose can be expressed as a composition of a position and an orientation relations. Since the purpose is to annotate semantically the (numerical) attributes of those geometric relations, a choice of coordinate representation must be made. This choice implies another set of constraints and a specific interpretation of the numerical attribute. For example, a quaternion as a representation choice of an orientation implies specific algebraic properties on the numerical tuple of four elements.

Modelling the geometric domain in a exhaustive way is out of scope of this work, which instead relies on the results described in [3].

IV. A DSL for Data Prototypes

This section proposes a DSL to add semantic annotations to existing data types. For the sake of clarity and brevity, the

```
dproto semantic_ros_position :: geometry {
  semantic   = Position
  coord      = cartesian
  ddr        = ros_position
  algebraic  = position_named
  dr         = {x=X, y=Y, z=Z}
  units      = position_units
}
```

domain

Listing 1. Example of a `dproto` that represents a `Position` in the `geometry` domain. This definition provides semantic annotations to the `ddr` expressed in Listing 2.

```
ddr ros_position :: ROS { geometry_msgs/Point }
```

Listing 2. Example of a `ddr` definition.

```
algebraic position_named  :: Scalar{X,Y,Z}
algebraic position3       :: Vector{3}
algebraic ht_matrix       :: Matrix{4,4}
```

Listing 3. Examples of `algebraic` definitions used in the running example.

section is driven by a set of small running examples, also used in later sections. The formal grammar definition is not reported in this paper, but it can be found in the software documentation. The design choices (grammar and syntax) of the proposed DSL strive for clarity and user friendliness, but alternative encondings of the same concepts are possible. Moreover, grammar and syntax might be subject to changes in future versions, allowing other modelling features not yet considered. Nevertheless, this does not change the concepts introduced in this work.

A. Data Prototype, `dproto`

A data prototype,[3] `dproto`, is a formal collection of semantic annotations that describe the interpretation of the values represented in a data type. An example is shown in Listing 1. The definition of a `dproto` must have an *identifier* (*e.g.,* `semantic_ros_position`) and a *domain* of reference of the `dproto` definition (*e.g.,* `geometry`). Other annotations are described as follows.

Semantic (`semantic`): a symbolic annotation to represent the meaning of the modelled data in the domain of reference. For example, in the domain `geometry`, a tag representing a geometric *relation* is expected, such as `Position`, `Orientation`, `Pose`, `LinearVelocity`, `AngularVelocity`, `Twist`, `Torque`, `Force`, and `Wrench`.

Coordinate Representation (`coord`): the choice of a coordinate representation, valid with respect to the `semantic` value of the `dproto`, previously defined in the domain `geometry`. For example, valid coordinate representations for `Position` are `cartesian`, `polar`, `cylindrical`, etc. More choices are available for an `Orientation`, such as `quaternion`, rotation matrix (`rot_matrix`), Euler angles, RPY-angles, etc. A survey on the coordinate representation choices and their properties can be found in [3].

Digital Data Representation (`ddr`): a reference to a model that describes how the data is managed in-memory, for storing or serializaton purposes. This can be expressed by a concrete data type used in a specific general purpose programming language, or expressed by an IDL. Seen from a *bottom-up* modelling approach, the `ddr` is the data type subject of the annotation. An example of `ddr` definition is shown in Listing 2, and it is composed of:

- a meta-model identifier (*e.g.,* `ROS` in the example) that expresses which IDL (or any other model) has been used to describe the digital data representation;
- a model (or an identifier to a model) that describes the digital data representation, conforming to the meta-model identifier.

Algebraic (`algebraic`): a choice of the abstract data type suitable for the given semantics; each choice corresponds to a syntax for the symbolic reference to the values of the data type. Algebraic choices are:

- `Scalar{...}`, an unordered sequence of scalar values. A different symbol (identifier) is used to refer to each concrete value;
- `Vector{N}`, an ordered sequence of N elements. An integer value (index) is used as an accessor to the concrete values;
- `Matrix{M,N}`, a matrix $(M \times N)$; a pair of integer values (*i.e.,* row and colums, (r, c)) is used to refer to the concrete values.

This abstraction is important as primary mean to establish semantic relations between the digital data representation (`ddr`) and other annotations. In short, `algebraic` defines a data type abstraction of the digital data representation. Listing 3 illustrates some `algebraic` definitions.

Data Representation (`dr`): a relation between the `ddr` and the `algebraic` choices, within the same `dproto`. It relates any symbol or accessor used in the actual digital data representation with the corresponding symbol/accessor defined in the `algebraic` representation. In the example of Listing 1, the values x, y and z of the ROS-IDL `geometry_msgs/Point` are linked to the `algebraic` accessor symbols X, Y and Z, respectively. In this way, it is possible to refer to the concrete values by means of the `algebraic` data type abstraction. This also means that multiple `dproto` definitions with same semantic annotations but different `ddr`, *e.g.,* the concrete implementation of the data type, are possible.

Physical Units (`units`): a *relation* between an `algebraic` definition and the physical units used to interpret the numerical values associated with the abstract

[3] The name *prototype* is inspired by the prototype pattern, which favours *composition* over *inheritance*, and that composability is a major design driver behind the presented work.

```
dproto s_plain_position :: geometry {
  semantic  = Position
  coord     = cartesian
  ddr       = :: c99 { double[3] }
  algebraic = position3
  dr        = {0=0, 1=1, 2=2}
  units     = position_units
}
```

Listing 4. An alternative representation of a `Position`, already modelled in Listing 1, based on different `ddr` and `algebraic`. The `ddr` is defined in compact (anonymous) form.

```
units position_units ::
        qudt, position_named {m = X, m = Y, m = Z}
```

Listing 5. Example of `units` definition, which relates the symbol of meters `m`, unit in the QUDT ontology, and the symbols expressed in the `position_named` `algebraic` definition.

data type. An example of unit definition is shown in Listing 5. Each symbol defined in the `algebraic` definition is associated with a physical unit (if applicable). This relation must also indicate which ontology is used (*e.g., QUDT*). In a `dproto` definition, the physical units, together with the `semantic` value and the context provided by the domain (*e.g.,* `geometry`), already suffice to uniquely define both *dimension* and *quantity kind* represented by the numerical value.

B. Composite Data Prototypes

There are two form of data prototype composition:

1) Composition of `dprotos` (type I): a `dproto` definition can be expressed as a composition of other `dprotos`, as shown in Listing 6. The `semantic` assigned to it must be in turn a composite concept in the modelled domain; *e.g.,* a `Pose` or a `Twist` in the `geometry` domain. The `dproto` is annotated with a `composes` relation, which associates a member of the composed semantic concept (left-hand side) with an identifier of the `dproto` used to represent it (right-hand side); obviously, the `semantic` must match. A `dproto` defined in this way creates an hierarchical dependency (a tree) among `dprotos`, where the composition inherits the annotations from its members. The data representation (`dr`) assigns symbolic identifiers to refer to the different parts of the semantic composition.

2) Composition by semantics (type II): in this case, the `dproto` is a stand-alone definition. However, the `semantic` annotation is a composite in the modelled domain. This is possible because, as opposed to the previous form of composition, the choice of coordinate representation `coord` (*i.e.,* a homogeneous transformation) fully describe the `semantic` concept of `Pose`. Listing 7 shows another example with `Pose` `semantic`, as alternative of Listing 6. However, it is possible to define one or more `view` of `dproto` definitions in this category, to refer to *part of* the composition.

```
dproto s_plain_pose :: geometry {
  semantic  = Pose
  composes = {
    Position    = s_plain_position
    Orientation = s_plain_rotation
  }
  dr = {position = Position,
              orientation = Orientation}
}
```

Listing 6. `s_plain_pose`, a composed `dproto` of `dprotos` (type I). The `semantic` of the `dproto` is `Pose`, which is a composite concept of the domain `geometry`, formed by `Position` and `Orientation`. `composes` relates the members of the semantic composition with other `dproto` definitions.

```
dproto s_plain_pose2 :: geometry {
  semantic  = Pose
  coord     = homogeneous_transformation
  algebraic = ht_matrix
  ddr       = :: c99 { double[16] }
  dr        =
   { 0={0,0},  1={0,1},  2={0,2},  3={0,3}, ... }
}
```

Listing 7. A snippet of a `dproto` "`s_plain_pose2`", composite by semantics (type II).

View (`view`): a relation that applies to a `dproto` composed by semantics (type II). A `view` refers to a part of the composite `dproto`, whose `semantic` value matches another `dproto`. Listing 8 illustrates two `view` definitions, all based on the `dproto` model in Listing 7: the left-hand side of the relation denotes a source, that is, the composite `dproto` and the corresponding semantic value subject of the view; the right-hand side of the relation denotes a target, that is a `dproto` semantically equivalent to the selected part; finally, the definition concludes with the mapping relation based on `algebraic` accessors.

C. Relations

The previous session introduced few relations already, which are essential for a minimal `dproto` definition. In addition, there are other *inter*-`dproto` relations; the one proposed by the current version of the DSL follows.

Alias (`alias`): a bidirectional equivalence relation between `algebraic` definitions. This relation maps each different symbol used in the abstract data types having the exact meaning. Listing 9 shows an example, where the `position_3` and `position_named` are interchangeable, from named-based accessor (`Scalar`) to an index-based accessor (`Vector`), and *vice versa*. This creates a relation between `dprotos`, *e.g.,* between `semantic_ros_position` (Listing 1) and `s_plain_position` (Listing 4), which are now equivalent even if the `algebraic` used is not the same.

Conversion (`conversion`): a *unidirectional* relation between two `dproto` definitions, to declare the existence of an external function implementation that perform the conversion between the `dproto` indicated. This allows to delegate (part of) the conversion, which is useful to overcome

```
view s_plain_pose2.Position -> s_plain_position {
  {3,0} = Z, {3,1} = Y, {3,2} = Z
}
view s_plain_pose2.Orientation->s_plain_rotation
   { {0,0}={0,0},  {0,1}={0,1}, ..., {2,2}={2,2} }
```

Listing 8. `view` definitions on the `dproto` `s_plain_pose2` of Listing 7.

```
alias position3 <-> position_named {
  0 = X, 1 = Y, 2 = Z
}
conversion semantic_ros_quaternion ->
                  s_plain_rotation = quat2rot
```

Listing 9. Example of an `alias` and a `conversion` inter-`dproto` relations.

to unmodelled aspects of the domain, or any limitation of the underlying reference implementation. A usage example is provided in Listing 9, where the conversion between `dproto` is delegated to an external function that implements a change between different coordinate representations.

V. Automatic Type Conversions

This section describes automatic data prototype conversion, which is an example of exploitation of the semantic annotations modelled by `dproto` definitions and their relations. Strictly speaking, the subject of the conversion is not the data prototype `dproto`, but an instance of it, called *data block* (`dblx`). The minimal definition of a data block is given by the combination of the data instance with its metadata, which must contain at least a reference to its metamodel (*i.e.*,, its `dproto`). In fact, it is not necessary to pack the whole `dproto` definition with the data, as long as the implementation allows to retrieve that information from an unique identifier. A *conversion* from a `dblx` (the *source*) and another (the *target*) consists in computing the values of the target `dblx` from the values of the source, such that they have the same information content. The conversion is possible only if the `dblx` are *semantically equivalent*, which is verifiable from their `dproto` definitions, and any additional relation previously modelled; Table I briefly summarises these convertibility requirements. The design of a reference implementation that performs such conversions should consider a two-step procedure: *i)* assess the feasibility of the conversion, by checking the semantic equivalence of the source and the target `dblx`; *ii)* compute the conversion. The first step allows to compute the *distance* of the conversion, defined as the number and the type of conversions steps necessary to perform the conversion. For those cases in which each conversion step is bounded in terms of number and type of operations to perform, the execution of the conversion is deterministic. This is guaranteed for the conversion type 1 and 2, while it depends on the external function implementation for conversions of type 3. From this type of analysis is possible to establish the overhead caused by the data type conversions, which is relevant in real-time applications.

```
algebraic quat              :: Vector{4}
algebraic quat_named        :: Scalar{x,y,z,w}
algebraic orient_rot_mx     :: Matrix{3,3}

alias quat_named  <-> quat { x=0, y=1, z=3, w=4 }

units quat_units :: qudt, quat {
  unitless=0, unitless=1 unitless=2, unitless=3
}

...

dproto asn_quaternion :: geometry {
  semantic  = Orientation
  coord     = quaternion
  algebraic = quat_named
  ddr       = :: ASN1 {
    Base-Types.Wrappers-Quaterniond
  }
  dr        = {re=w, im.0=x, im.1=y, im.2=z}
  units     = ...
}

dproto e_rotation :: geometry {
  semantic  = Orientation
  coord     = rot_matrix
  algebraic = orient_rot_mx
  ddr       = :: Eigen { Matrix<double,3,3> }
  dr        = { ... }
  units     = ...
}

dproto ROT :: geometry {
  semantic  = Orientation
  coord     = rot_matrix
  algebraic = orient_rot_mx
  ddr       = :: c99 { double[9] }
  dr        = { ... }
  units     = ...
}

dproto QUAT :: geometric {
  semantic  = Orientation
  coord     = quaternion
  algebraic = quat
  ddr       = :: c99 { double[4] }
  dr        = { 0=0, 1=1, 2=2, 3=3 }
  units     = quat_units
}

conversion QUAT -> ROT = quat2rot
```

Listing 10. A sample of `dproto` models that form the graph-based knowledge of Fig. 1. A conversion from `asn_quaternion` to `e_rotation` is performed, generating the static code shown in Listing 11.

A. Reference Implementation

An automated data type conversion tool (`dconv` [4]) has been realised to support the research hypothesis of this paper. The software is provided in form of a Lua module and a command-line tool. The choice of the Lua programming language [22] is made to guarantee easy interoperability with C/C++ and other languages, and to have a minimal impact on the software component that uses it. The same library provides: *i)* a code generator that emit the body of a requested conversion function, to be embedded directly in the user-code; *ii)* a runtime converter, that perform the conversion directly on the data allocated by the application. The current implementation is at prototypal stage, and it only targets to

TABLE I

SUMMARY OF THE CONVERSION TYPES SUPPORTED AND THEIR REQUIREMENTS.

n. type	Conversion Type	Requirements for Convertibility
1	direct plain conversion	same `semantic`, `coord` and `algebraic`.
2	direct conversion by `alias`	same `semantic` and `coord`, different `algebraic` but `alias` known.
3	direct conversion by `conversion`	same `semantic`, different `coord` and `conversion` relation known: conversion delegated to the external function.
4	direct composite conversion	both are composition, same `semantic`, members convertible.
5	indirect conversion	`semantic` must be compatible. Conversion performed by an ordered sequence of direct conversions, through other `dprotos` as intermediate steps.
6	indirect composite conversion	`semantic` must be compatible. A composite conversion with at least one member that requires an indirect conversion.

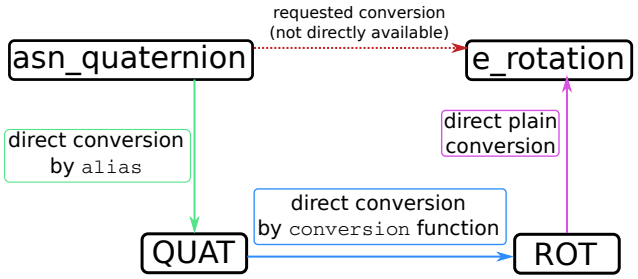


Fig. 1. Example of indirect conversion (type 5) with distance 3. The data type associated with the `dproto` `asn_quaternion` is converted to a data type associated with the `dproto` `e_rotation`.

```
double QUAT_tmp_2[4];
double ROT_tmp_3[9];
QUAT_tmp_2[0] = src.im.arr[0];
QUAT_tmp_2[1] = src.im.arr[1];
QUAT_tmp_2[2] = src.im.arr[2];
QUAT_tmp_2[3] = src.re;
quat2rot(QUAT_tmp_2,ROT_tmp_3);
tgt(0,0) = ROT_tmp_3[0];
tgt(0,1) = ROT_tmp_3[1];
tgt(0,2) = ROT_tmp_3[2];
tgt(1,0) = ROT_tmp_3[3];
tgt(1,1) = ROT_tmp_3[4];
tgt(1,2) = ROT_tmp_3[5];
tgt(2,0) = ROT_tmp_3[6];
tgt(2,1) = ROT_tmp_3[7];
tgt(2,2) = ROT_tmp_3[8];
```

Listing 11. Example of statically generated body function (C code). This converts an orientation, from a `dblx` (*src*) with `coord` quaternion and `ddr ASN.1`, to a `dblx` *tgt* with `coord` rotation matrix and `ddr Eigen`, by means of a composite indirect conversion (type 5).

C/C++ code. Moreover, the support is limited to the geometric domain described in this paper. As a part of the future work, a schema would be added to allow the definition of customised domains. Nevertheless, `dconv` already supports several digital data representation widely used in robotics, like existing IDLs (*e.g.,* ROS-IDL, ASN.1, SmartSoft Communication Object DSL), and also Eigen [23] classes and C99-compatible Plain Old Data (POD). `dconv` generates an uniform accessor schema based on the `algebraic` information provided by the `dproto` definitions. To each assessor is associated an operation, specific to the static code generator or to the runtime conversion engine. The set of `dproto` definitions and relations form a graph-based knowledge space, upon which conversion queries are performed. If a valid path from a source `dproto` to a target `dproto` is found, a conversion function is generated, chaining the previously defined operations.

B. Examples

Static code generation: the Listing 11 shows the generated body function to perform the conversion from `asn_conversion` and `e_rotation`, from the model illustrated in Listing 10.In this case, a direct conversion is not possible, but from the graph-based knowledge defined by the `dproto` models (see Fig. 1), a path is found to realise an indirect conversion (type 5). The generated code uses the accessors defined by the indicated IDL (`ASN.1` and `Eigen Matrix` class in this example).

Runtime conversion: it performs the conversion directly, on the allocated data provided by the application. In this case, the conversion engine must be embedded in the software component. Hence, this tool is a first step towards software components that the can adapt themselves at runtime, and to realise *proxies* between different middleware. TO this end, a simple ROS publish/subscribe example of this is provided along the sources of the tool. More details, conversion and usage examples can be found online.

VI. CONCLUSIONS

The proposed DSL and the `dconv` reference implementation facilitates the creation of multi-"vendor" robotics software systems, by allowing developers to create or to reuse formalizations of the complete semantic meaning of geometric data structures they must introduce in their software. The short-term ambition of the `dconv` development is to let it mature via implementations in multi-partner robotics research projects, with a focus on "realtime" control and sensor fusion components. These efforts have started already, and they are expected to lead to a draft standard candidate for the robotics community, following the proven approach of the IETF (Internet Engineering Task Force) in standardisation via small composable standards that come with reference implementations from the start.

ACKNOWLEDGMENT

The authors gratefully acknowledge the financial support by the European Community's Horizon 2020 Programme projects RobMoSys (H2020-ICT-732410), and ESROCOS (H2020-ICT-730080). The KU Leuven Robotics Research Group is a core lab of Flanders Make (Strategic Research Centre for the Manufacturing Industry in Flanders).

REFERENCES

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. B. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.

[2] Open Source Robotics Foundation, "ROS2 common interfaces," https://github.com/ros2/common_interfaces.

[3] 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 Robotics and Automation Magazine*, vol. 20, no. 1, pp. 84–93, 2013.

[4] E. Scioni, "`dconv`, an Automatic Datatype Cnversion Tool," Software repository, 2018, https://github.com/haianos/dconv-tool.

[5] D. Stampfer, A. Lotz, M. Lutz, and C. Schlegen, "The SmartMDSD Toolchain: an Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software," *Journal of Software Engineering in Robotics*, vol. 7, no. 1, pp. 3–19, 2016.

[6] C. Schlegel, "SmartSoft: Components and toolchain for robotics," http://smart-robotics.sourceforge.net/.

[7] ——, "SmartSoft: Components and toolchain for robotics," http://www.servicerobotik-ulm.de/toolchain-manual/html/ch02s02s02.html.

[8] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The Architecture Analysis & Design Language (AADL): An introduction," Tech. Rep., 2006.

[9] M. Perrotin, E. Conquet, J. Delange, A. Schiele, and T. Tsiodras, "Taste: A real-time software engineering tool-chain overview, status, and future," in *SDL 2011: Integrating System and Software Modeling*, I. Ober and I. Ober, Eds. Springer Berlin Heidelberg, 2012, pp. 26–37.

[10] J. Larmouth, *ASN.1 Complete*. Academic Press, OSS (Open Systems Solutions), 1999.

[11] "Google protocol buffers: Google's data interchange format," https://developers.google.com/protocol-buffers, 2018.

[12] G. LLC, "Google flatbuffers: a memory efficient serialization library," https://google.github.io/flatbuffers/, 2018.

[13] The HDF Group, "Hierarchical Data Format, version 5," https://portal.hdfgroup.org/display/HDF5/HDF5.

[14] T. R. Gruber, "Toward principles for the design of ontologies used for knowledge sharing," *International Journal of Human-Computer Studies*, vol. 43, no. 5–6, pp. 907–928, 1995.

[15] T. R. Gruber and G. R. Olsen, "An ontology for engineering mathematics," in *Fourth International Conference on Principles of Knowledge Representation and Reasoning*, 1994, pp. 258–269.

[16] W3C, "QUDT (Quantities, Units, Dimensions, and Types)," http://www.qudt.org.

[17] H. Rijgersberg, M. F. J. van Assem, and J. L. Top, "Ontology of units of measure and related concepts," *Semantic Web — Interoperability, Usability, Applicability*, vol. 4, no. 1, pp. 3–13, 2013.

[18] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and N. Lindström, "A JSON-based serialization for Linked Data," http://www.w3.org/TR/json-ld/, 2014.

[19] J. Earley, "Relational level data structures for programming languages," *Acta Informatica*, vol. 2, no. 4, pp. 293–309, Dec 1973. [Online]. Available: https://doi.org/10.1007/BF00289502

[20] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv, "Data representation synthesis," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 38–49. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993504

[21] C. Loncaric, M. D. Ernst, and E. Torlak, "Generalized data structure synthesis," in *International Conference on Software Engineering*, 2018.

[22] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho, "Lua—an extensible extension language," *Softw. Pract. Exper.*, vol. 26, no. 6, pp. 635–652, 1996.

[23] G. Guennebaud, B. Jacob, *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.