RobMoSys

H2020—ICT—732410

# RobMoSys

Composable Models and Software for Robotics Systems

# Deliverable D2.5: Modeling Foundation Guidelines and Meta-Meta-Model Structure

Dennis Stampfer (Hochschule Ulm)
Alex Lotz (Hochschule Ulm)
Christian Schlegel (Hochschule Ulm)

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under grant agreement No. 732410

*Project acronym*: RobMoSys

*Project full title*: Composable Models and Software for Robotics Systems

*Work Package*: WP 2

*Document number*: D2.5

*Document title*: Modeling Foundation Guidelines and Meta-Meta-Model Structure

*Version*: 1.0

*Due date:* June 29th, 2018

*Delivery date*: June 29th, 2018

*Nature*: Report (R)

*Dissemination level*: Public (PU)

*Editor:* Dennis Stampfer (HSU), Alex Lotz (HSU), Christian Schlegel (HSU)

*Author(s)*: Dennis Stampfer (HSU), Alex Lotz (HSU), Christian Schlegel (HSU), Enea Scioni (KUL), Nico Huebel (KUL), Herman Bruyninckx (KUL), Matteo Morelli (CEA), Chokri Mraidha (CEA), Sara Tucci (CEA), Huascar Espinoza (CEA)

*Reviewer:* Sergi Garcia (PAL Robotics)

## Executive Summary

This deliverable provides the *updated* version (M18) of the modeling foundation guidelines and the meta-meta-model structures with another update in M30. Such foundations provide the formal basis for model structures and their processing.

Modeling foundation guidelines and meta-meta-model structures provide the means to express the structures to manage the interfaces between different roles (robotics expert, domain expert, component supplier, system builder, installation and deployment, operation), at different levels of abstraction (e.g. from high-level "*move and perceive*" to "*grasp firmly*" to very detailed "*manipulate with pinch grasp and non-slipping prehension pressure*"), and with respect to different concerns (computation, coordination, configuration, communication) in an efficient and systematic way by making the step change to a set of fully model-driven methods and tools for composition-oriented engineering of robotics systems.

That is needed to enable a composition-oriented engineering process for robotic systems where the properties of a system are predictable from the properties of its building blocks and their composition "glue". Designing a robotic system and building a robotic system becomes a process of composition and configuration of building blocks.

The RobMoSys consortium uses a Wiki for the content of this document. This allows for a living document with a continuous publishing process following the principles of composition for its content. While the basic principles expressed in this initial version will remain stable, refinements and extensions as well as improvements will be added continuously.

Thus, this document serves as a guide through that material of the Wiki visible on the RobMoSys website which is relevant to this deliverable. A snapshot of the content of the Wiki at the time of delivery of this document is attached in the appendix.

The presented version of this document is an *update of "Deliverable D2.1: Modeling Foundation Guidelines and Meta-Meta-Model Structures"*. It *extends* the reading guide through the Wiki and highlights the updates since the previous version of this deliverable. Since the initial version of the deliverable, the structures have proven to be suitable and stable. The document lists examples of their realization within RobMoSys tooling and examples where the RobMoSys approach has been applied.

## Summary of Updates to this Document

### D2.1 - June 30th, 2017

Initial version of this deliverable

### D2.5 - June 30th, 2018

This document is an update of the initial deliverable D2.1. This document serves as an updated guide to the RobMoSys wiki. Therefore, the following extensions of the RobMoSys Wiki contribute to this deliverable:

- The RobMoSys Glossary has been *extended*
- The Meta-Model Structures has been *updated* and has *demonstrated its suitability*
    - Minor *updates* have been undertaken: graphical notation and description has been updated.
        - Wiki Page on "**Block-Port-Connector**"
    - The concept of Block-Port-Connector has demonstrated to be suitable through the implementation and use in Papyrus4Robotics.
        - Wiki Page on "**Papyrus4Robotics**"
- The Modeling Foundation Guidelines have been *updated* and *extended*
    - *Extended* the description of the Ecosystem Organization
        - The following new subpages have been added. They now describe the realization in RobMoSys tooling. RobMoSys can now be experienced by using tooling and the existing building blocks or systems.
            - Wiki page on "**Tooling Support for the RobMoSys Ecosystem Organization**"
        - The Ecosystem Organization has been *extended* to cover the evolvement of structures in Tier 1
            - Wiki page on "**Tier 1 in Detail**"
            - Section "Ecosystem Organization in the Industry 4.0 Domain" and Wiki page on "**OPC Unified Architecture (OPC UA)**"
    - *Extending* and *updating* the Architectural Patterns and other related pages around Robotic Behavior. The main pages that have been influenced are:
        - Wiki page on "**Architectural Pattern for Component Coordination**"
        - Wiki page on "**Architectural Pattern for Task-Plot Coordination (Robotic Behaviors)**"
    - *Extending* Roles in the ecosystem
        - Adding the Behavior Developer Role Description and extending the Component Supplier Role
            - Wiki page on "**Behavior Developer**"
            - Wiki page on "**Component Supplier**"
        - *Extending* the description for the Performance Designer
            - Wiki page on "**Performance Designer**"
- The Foundations and Structures have demonstrated feasibility through applying them
    - Whole new section in the wiki with several subpages on "**Composition in an Ecosystem**" to illustrate composition by several examples. These examples also describe how RobMoSys tooling can be used to apply the specified concepts.
        - Wiki page on "**Task-Level Composition for Robotic Behavior**"
        - Wiki page on "**Service-based composition of software components**"

        - Wiki page on “**Managing Cause-Effect Chains in Component Composition**”
        - Wiki page on “**Coordinating Activities and Life Cycle of Software Components**”
    - Examples of Tier 2 domain structures have been extended and its support through RobMoSys tooling has been described
        - Wiki page on “**Flexible Navigation Stack**”
    - Extended the description of how the RobMoSys tooling supports in using the meta-model structures and modeling foundation guidelines
        - Wiki page on "**Papyrus4Robotics**"
        - Wiki page on "**The SmartMDSD Toolchain**"

In addition to the wiki, the following extensions to this document have been made:

- Proven suitability to disseminate the RobMoSys concepts and knowledge through the wiki in a very open and transparent way to engage the robotics community.
- Comparison of the RobMoSys Ecosystem with OPC UA in the industry 4.0 domain
- Description on how the RobMoSys Ecosystem Tier 1 structures evolve over time.
- Description on how RobMoSys realizes the Ecosystem Tiers.
- Description on the Block-Port-Connector realization alternatives.

## Content

# 1 Introduction

RobMoSys is about managing the interfaces between different roles (robotics expert, domain expert, component supplier, system builder, installation, deployment and operation) and separate concerns in an efficient and systematic way by making the step change to a set of fully model-driven methods and tools for composition-oriented engineering of robotics systems.

It is about the first principles of *composition*, *separation of roles* and *explicated models*:

- understanding at design-time
- plausibility at run-time
- justifiability at inspection time

## 1.1 The RobMoSys Wiki

The RobMoSys consortium uses a Wiki for the content of this document. This allows for a living document with a continuous publishing process following the principles of composition for its content. While the basic principles expressed in this initial version will remain stable, refinements and extensions as well as improvements will be added continuously.

Thus, this document serves as a guide through that material of the Wiki visible on the RobMoSys website which is relevant to this deliverable. A snapshot of the content of the Wiki at the time of delivery of this document is attached in the appendix.

This document refers to the RobMoSys wiki. A snapshot is attached in the appendix of this document for simple printing. Additionally, it can be accessed online at

- http://www.robmosys.eu/wiki-sn-02/

We refer to specific wiki pages like this: *Wiki Page on "<Title of wiki page>"*. These wiki pages can be accessed via its title in the appendix and in the RobMoSys Wiki Jump-Page at

- http://www.robmosys.eu/wiki-sn-02/jumppage

The live version of the wiki at http://www.robmosys.eu/wiki also reflects updates and ongoing additions after the submission of this document. An up-to-date jump-page can be found at

- http://www.robmosys.eu/wiki/jumppage

The main philosophy behind the RobMoSys Wiki is to favour early access, openness, and transparency over completeness. This is to support communication of RobMoSys being a community endeavour. During more than one year of maintaining and sharing technical insights through the Wiki, it was confirmed that this approach indeed simplifies and speeds up the communication on one hand. On the other hand, it is very well appreciated by the closer RobMoSys community (e.g. by integrated technical projects / ITPs) and the general robotics community. These claims are supported by many positive feedbacks from the many events such as brokerage days, conference workshops, the European Robotics Forum Workshop, and technical workshops with RobMoSys ITPs.

## 1.2 RobMoSys Ecosystem Organization

An initial version of a **Glossary** provides definitions for the most relevant terms in the context of

RobMoSys. See

- Wiki Page on "**Glossary**"

RobMoSys envisions a robotics business ecosystem in which a large number of loosely interconnected participants depend on each other for their mutual effectiveness and individual success. The modeling foundation guidelines and the meta-meta-model structures are driven by the needs of the typical tiers of an ecosystem and the needs of their stakeholders (see figure 1). The different tiers are arranged along levels of abstractions. Figure 1 also illustrates the amount of experts and people contributing to and using the particular tiers.

**Tier 1** structures the ecosystem in general for robotics. It is shaped by the drivers of the ecosystem that define an overall composition structure which enables composition. Tier 1 contains the main technical structures for composition and the lower tiers conform to Tier 1 (similar to, for example, the ecosystem of the Debian GNU/Linux OS and its structures). Tier 1 is shaped by few representative experts for ecosystems and composition. This is kick-started by the RobMoSys project.

Structures defined on Tier 1 can be compared to structures that are defined for the PC industry. The personal computer market is based on stable interfaces that change only slowly but allow for parts changing rapidly since the way parts interact can last longer than the parts themselves and there is a huge amount of cooperating and competing players involved. This resulted in a tremendous offer of composable systems and components.

**Tier 2** conforms to these foundations, structuring the particular domains within robotics and is shaped by the experts of these domains (representatives of the individual sub-domains in robotics), for example, object recognition, manipulation, or SLAM.

**Tier 3** conforms to the domain-structures of Tier 2 to supply and to use content. Here are the main "users" of the ecosystem, for example component suppliers and system builders. The number of users and contributors is significantly larger than at the above tiers as everyone contributing or using a building block is located at this tier.

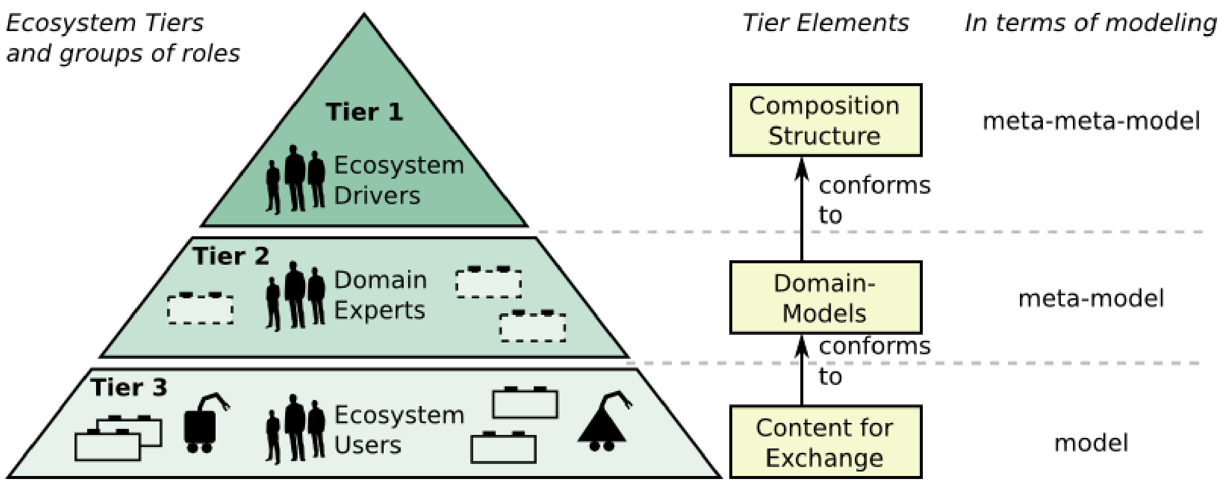


*Figure 1: Tiers of an Ecosystem, their elements and the elements in terms of modeling.*

Tier 1 further distinguishes between generic composition structures (**Modeling Foundation Guidelines and Meta-Meta-Model Structures** such as scientific grounding and block-port-connector concepts), and the RobMoSys composition structures (concepts for robotics building blocks). Deliverable D2.1 focusses on generic composition structures while D2.2 focuses on the RobMoSys composition structures which are both at Tier 1.

See:

- Wiki page on "**Ecosystem Organization**"

## Contents in the RobMoSys Ecosystem

The realization of the **three composition tiers** in tooling validates the feasibility of the overall approach through applying it. The RobMoSys tooling implements a vertical example of the composition tiers. See:

- Wiki page on "**SmartMDSD Toolchain Support for the RobMoSys Ecosystem Organization**"
- Wiki page on "**Getting Started With Papyrus4Robotics**"

The structures of **Tier 1** have been implemented in Papyrus4Robotics to support modeling at Tier 2 and Tier 3. They also have been implemented in the SmartMDSD Toolchain to support modeling, code-generation and use of previously developed software components at Tier 2 and Tier 3. See

- Wiki page on "**Papyrus4Robotics**"
- Wiki page on "**The SmartMDSD Toolchain**"

Examples of **Tier 2** domain models support the feasibility of the RobMoSys approach for domain modeling. A repository of domain models for use with the SmartMDSD Toolchain has been set up at [1].

- Wiki page on "**Flexible Navigation Stack**"

To demonstrate **Tier 3**, a repository of previously developed software components for use with the SmartMDSD Toolchain has been set up[2]. A repository of systems that serve as an example for use with the SmartMDSD Toolchain has been set up as well[3]. Initial pilot skeletons are available that demonstrate running systems. See

- Wiki page on "**The SmartMDSD Toolchain**"
- Wiki page on "**Gazebo/TIAGo/SmartSoft Scenario**"

The RobMoSys Ecosystem, Foundation Guidelines, and Meta-Model Structures tend to be stable: the recent discussions with the running integrated technical projects (ITPs) allow for an early conclusion that the RobMoSys structures allow for integration of the ITP contributions without altering the RobMoSys structures.

[1] https://github.com/Servicerobotics-Ulm/DomainModelsRepositories
[2] https://github.com/Servicerobotics-Ulm/ComponentRepository
[3] https://github.com/Servicerobotics-Ulm/SystemRepository

## Ecosystem Organization in the Industry 4.0 Domain

The organization of an ecosystem in three tiers can also be found in other domains. For example, a significant part of the industry 4.0 domain is shifting towards the OPC Unified Architecture[4] (OPC UA). OPC UA is a standard for machine-to-machine communication comprising communication infrastructure and information models for semantic data exchange. OPC UA is standardizing connectivity of industrial devices and enables the interoperability among products of different vendors. It does not yet address the next level of interoperability which we call "composability".

The OPC UA ecosystem is in its structures exactly conformant to the explicated tiers of the RobMoSys ecosystem approach. The OPC foundation is the driver in tier 1, the companion specifications belong to tier 2 and finally there are the users at tier 3. The strong point about OPC UA is that it is driven by industry in a joint effort and that they successfully manage the ramp up of an ecosystem along these tiers.

A direct comparison of the RobMoSys Ecosystem with OPC UA is given in the figure below.

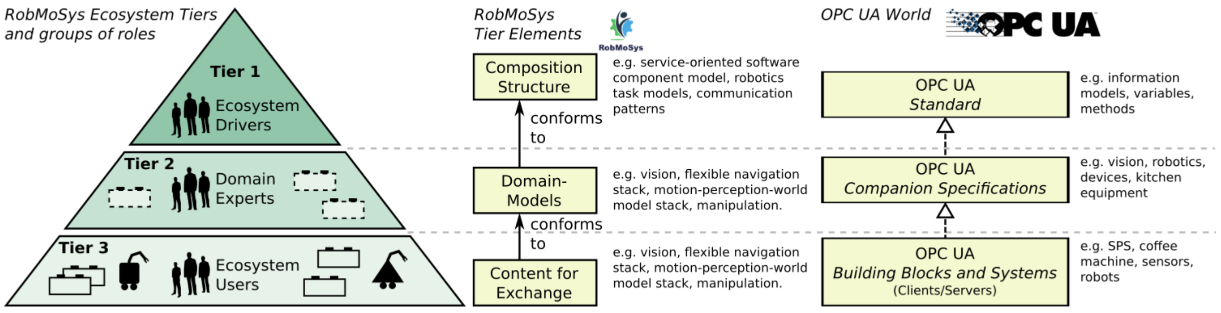

RobMoSys Ecosystem Tiers and groups of roles
Tier 1
Ecosystem Drivers
Tier 2
Domain Experts
Tier 3
Ecosystem Users
RobMoSys Tier Elements
Composition Structure
e.g. service-oriented software component model, robotics task models, communication patterns
conforms to
Domain-Models
e.g. vision, flexible navigation stack, motion-perception-world model stack, manipulation.
conforms to
Content for Exchange
e.g. vision, flexible navigation stack, motion-perception-world model stack, manipulation.
OPC UA World
OPC UA Standard
e.g. information models, variables, methods
OPC UA Companion Specifications
e.g. vision, robotics, devices, kitchen equipment
OPC UA Building Blocks and Systems (Clients/Servers)
e.g. SPS, coffee machine, sensors, robots

*Figure 1b: Direct comparison of the RobMoSys Ecosystem and the OPC UA Ecosystem.*

As prominent example for domain models (companion specifications), VDMA is working on companion specifications for vision[5] and robotics[6]. Companion specifications sometimes contain additional concepts that have evolved in a particular domain, but that are generally applicable. For example, the companion specification for vision foresees a generic state automaton for components with component-specific sub-states---a very similar concept to the RobMoSys component life-cycle and state pattern[7]. In the long-run, they may be adopted by OPC UA itself, thus move from Tier 2 to Tier 3. This movement of structures describes the evolvement of an ecosystem and also has been identified for RobMoSys (see wiki page on „Tier 1 in detail"). OPC UA is actively postulating the creation of companion specifications by providing support and guidance.

OPC UA eases device integration thanks to an overall methodology (Tier 1) and domain-specific standards (composition Tier 2). Device suppliers now can adopt the Tier 2 standards and gain compatibility with users that expect these standards. OPC UA, however, does not specifically aim for composition and is, in fact, less suitable for composition of software components. It misses adequate abstractions and concepts (e.g. such as RobMoSys communication patterns). However, composability starts being addressed in OPC UA as it can be observed in recent developments that are on the way to introduce the concept of skills.

OPC UA can also be used as an underlying communication infrastructure below the RobMoSys structures. In the context of composition, the challenge with OPC UA is to introduce additional structures that enable composition. This is done by, for example, the RobMoSys communication

[4] https://opcfoundation.org
[5] https://opcfoundation.org/markets-collaboration/vdma-machine-vision
[6] https://opcfoundation.org/markets-collaboration/vdma-robotics
[7] Wiki page on "Coordinating Activities and Life Cycle of Software Components"

patterns. This is where the German national BMWi/PAiCE Project "Service Robot Network"[8] (SeRoNet) is adopting parts of the RobMoSys composition structures and provides a mapping to OPC UA. Thereby, SeRoNet can fully benefit from composition as introduced by RobMoSys but also manages the seamless integration with the traditional OPC UA world, for example to use OPC UA powered devices.

In general, the industry 4.0 world based on OPC UA has a fully conformant way of thinking with respect to the overall RobMoSys world. Thus, there is a very good chance to communicate the RobMoSys contributions to that domain and thereby link the robotics domain with the automation domain. While OPC UA and its companion specifications at the moment are at the level of integration with a roadmap towards the next levels which we call composability, RobMoSys already now proposes solutions to address composability. Due to the very same ecosystem structures, there is a very good chance to enable adoption of the RobMoSys outcomes within the industry driven OPC UA automation domain. For RobMoSys, the strength of OPC UA is that it provides standardized and uniform ways to access all kinds of devices like sensors, actuators, machineries, cloud services etc. RobMoSys puts its focus on the software composition for most complex sensori-motor systems which then can get networked with industry 4.0 environments via OPC UA.

See also:

- Wiki page on "**OPC Unified Architecture (OPC UA)**"
- Wiki page on "**Tier 1 in Detail**"

# 2 Approach

## 2.1 Introduction

**Structural models** describe the static aspects of a system, its parts and relationships. It is widely accepted to represent structural models as a set of connected blocks. Blocks encapsulate functionalities and interact with their external world (which can be other blocks) via ports. A port establishes an interface that external elements can use to interact with the block. Connectors connect ports of blocks. Note that a block represents a generic entity. It can for instance represent a logical / functional element (data processing, controlling, actuating), a piece of software (a software component) or even a physical entity (e.g. a sensor or an actuator). A block can encapsulate other blocks for reducing complexity (nesting for information hiding and information abstraction). Blocks can be arranged such that they represent abstraction levels in vertical stacks.

As generic representation behind blocks, ports and connectors, we have chosen **hierarchical hypergraphs** which represent links between different models and different views (whether it is a S/W component model, a H/W component model, a kinematic chain, a model for task plots, etc.). They also allow for sound foundations for managing constraints and partial bindings across different models and provide the basis for navigating through such structures.

**Behavioral models** describe the dynamic aspects of a system, for instance, a chain of actions or system states and associated events / transitions. Again, the port-based modeling paradigm with ports, blocks and connectors can be used to structure the behavioral models. Behavioral models can be composed out of other behavioral models through the port-based modeling paradigm. Behavioral models can range from continuous time phenomena to discrete event systems, from reactive systems to planning systems. They can range from finite-state-automatons (as used for managing the life-cycle inside a component) over different forms of process networks (to model behavioral characteristics of connected components) to robot coordination languages (such as

[8] https://www.seronet-projekt.de

task-nets used for orchestration of the overall robot behavior and subordinated components).

**Knowledge models** are formal relationships between primitives and parameters in the structural and the behavioural models, to encode dependencies between them that hold in particular contexts. In contrast to the structural and the behavioural models, knowledge models are not best served by means of a block-port-connector approach, but by plain formal n-ary (or "graph") relationships; and hierarchy is an extremely important structural property of knowledge relationships and such knowledge hierarchies can overlap, which is a fundamental ("compositional") property of "contexts" or "namespaces".

Finally, a **robotics system** is composed out of building blocks (components and their horizontal and vertical composition). A component is a block which encapsulates functional blocks, behavioural blocks, the inter-relationships between these blocks and their configurations. A component comes with data ports, coordination ports and configuration ports. Again, a component can encapsulate other components with their functionalities, behaviours and configurations. The advantage of encapsulation is that a system can be modeled by composing and connecting the ports of its sub-systems, independently of alternative or future implementations of these sub-systems. Aggregate ports can describe very high-level connections between components which can be refined into various types of interaction models (e.g. patterns for geometrical interaction, patterns for information exchange with quality-of-service).

Both, structural and behavioral models are necessary to define the system, but structural models are necessary to organize the the system in units and to hide complexity. In this way, different roles (software engineer, behavior developer, system builder, etc.) are exposed to different parts of the system model at different levels of granularity. Most importantly, the different roles can add constraints to those parts visible to them at any time refining the problem space (requirements) and / or the solution space for other downstream developer roles.

Thanks to structural modeling, all the different roles can compose different parts of the system at different levels of granularity and for different concerns. For instance, blocks can be composed under the constraints of a specific architectural pattern. Blocks of different nature can also be composed together, for instance blocks representing functionalities can be composed with blocks representing computing resources to describe which resources will be used by a given functionality.

As said, design becomes a process of composition and configuration of components. The target system is designed using predefined, modular components which are selected, configured and assembled in a way that the design specifications are met. Models for system-of-systems comprise structural models and behavioural models for both, the internals of a component and the interaction between components. For example, system-level behavioural models relate to component configurations, the behavioural models of the components and the interaction between the components.

## 2.2 The Details of Tier-1

Figure 2 shows the details of the structure of Tier-1. All the elements in Tier-1 are summarized as meta-meta-models. Moreover, the meta-meta-models within Tier-1 are organized themselves in a hierarchical manner in order to best serve the realization of the RobMoSys objectives. The lowest level within Tier 1 contains the RobMoSys composition structures. Tier-2 then conforms to these composition structures.

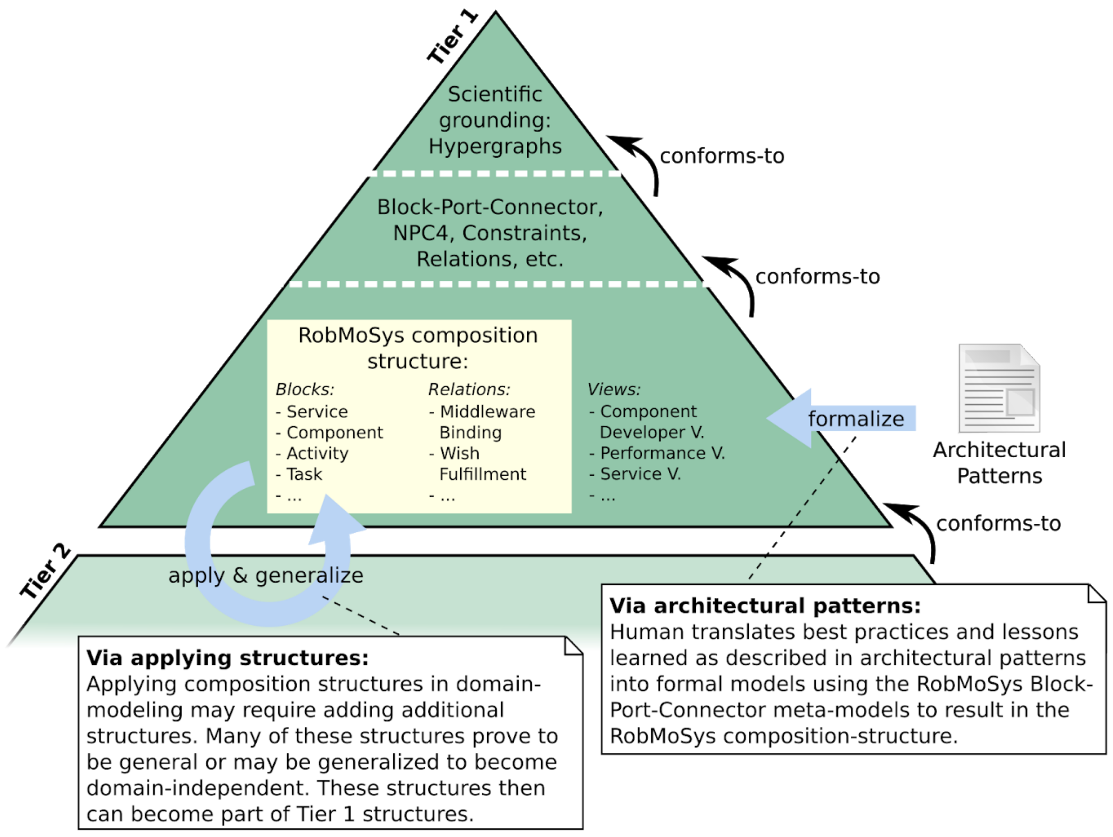

Tier 1
Scientific grounding: Hypergraphs
conforms-to
Block-Port-Connector, NPC4, Constraints, Relations, etc.
conforms-to
RobMoSys composition structure:
Blocks:
- Service
- Component
- Activity
- Task
- ...
Relations:
- Middleware Binding
- Wish Fulfillment
- ...
Views:
- Component Developer V.
- Performance V.
- Service V.
- ...
formalize
Architectural Patterns
conforms-to
Tier 2
apply & generalize
Via architectural patterns:
Human translates best practices and lessons learned as described in architectural patterns into formal models using the RobMoSys Block-Port-Connector meta-models to result in the RobMoSys composition-structure.
Via applying structures:
Applying composition structures in domain-modeling may require adding additional structures. Many of these structures prove to be general or may be generalized to become domain-independent. These structures then can become part of Tier 1 structures.

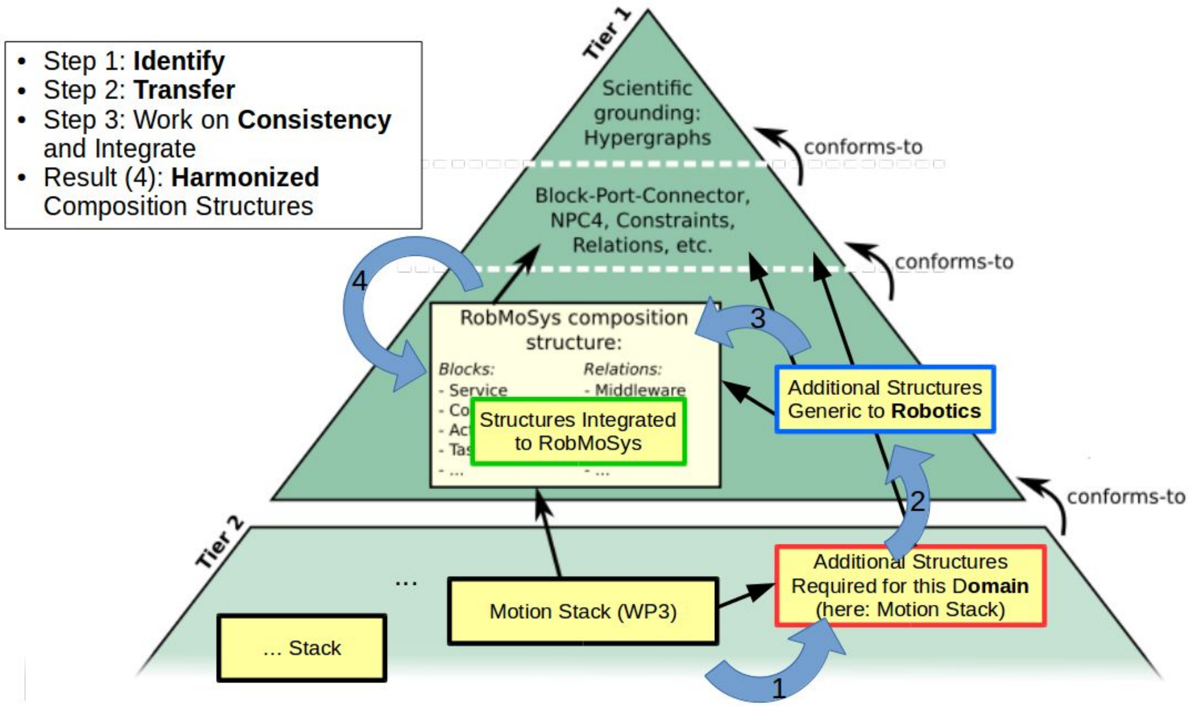
*Figure 2: Details of the structure of Tier-1.*

There are two approaches on how to come up with the composition structures in Tier 1. RobMoSys is a community effort and will guide contributors in one of these approaches such that their knowledge and methodology becomes accessible through the composition structures. For example, the following two approaches have already proven to be successful with respect to the integrated technical projects (ITPs) of RobMoSys.

The first and initial approach to come up with composition structures is to formalize architectural patterns (Fig. 2). See

- Wiki page on "**Architectural Patterns**"

The second approach is to evolve the composition structures over time by generalizing existing domain-specific structures. In some cases, the composition structures of Tier 1 may not be sufficient or not complete for modeling in a particular robotics domain. This situation requires additional structures to be added on Tier 2. However, many of these structures tend to be generally applicable or may be generalized such that they become domain-independent and finally part of the composition structures. This is illustrated in the figure below.



Step 1: Identify
Step 2: Transfer
Step 3: Work on Consistency and Integrate
Result (4): Harmonized Composition Structures
Tier 1
Scientific grounding: Hypergraphs
conforms-to
Block-Port-Connector, NPC4, Constraints, Relations, etc.
conforms-to
RobMoSys composition structure:
Blocks:
- Service
Relations:
- Middleware
Structures Integrated to RobMoSys
Additional Structures Generic to Robotics
conforms-to
Tier 2
...
Motion Stack (WP3)
Additional Structures Required for this Domain (here: Motion Stack)
... Stack
1
2
3
4

*Figure 3: Evolvement of Tier 1 composition structures via applying them on lower tiers.*

The first step (step 1, figure above) is to identify the additional structures that are independent of an application but general to a domain. The second step is to transfer these structures to Tier 1, thereby making them domain independent (step 2, figure above). The final step is to work on the consistency of the newly identified structures with the existing composition structures with the aim to integrate them (step 3, figure above).

For example, it is necessary to shape them to the overall RobMoSys approach, taking separation of roles, composability, etc. into account. This results in the next generation of harmonized composition structures (step 4, figure above).

See:

- Wiki page on "**Tier 1 in Detail**"

## 2.3 Block-Port-Connector Realization Alternatives

RobMoSys describes the Block-Port-Connector concept (including the concepts of topology and mereology) as a generic and recurring mechanism that can be found in different meta-meta-model realization alternatives. Two widely known realization alternatives are Eclipse Ecore and Essential Meta-Object Facility (EMOF). While this level alone is not sufficient to realize composition in robotics, it provides an essential foundation for the RobMoSys composition structures (i.e., the RobMoSys actual meta-models).

Meanwhile, several of the realization-independent RobMoSys composition structures have been realized within the SmartMDSD Toolchain (based on Eclipse Ecore, Xtext and Sirius technologies) and within the Papyrus4Robotics Toolchain (leveraging concepts from https://www.omg.org/spec/MARTE/ based on UML profiles). See

- Wiki page on "**Papyrus4Robotics**"

- Wiki page on "**The SmartMDSD Toolchain**"

These realization alternatives validate and support the feasibility of the specified structures at Tier 1. The existing RobMoSys tooling builds on the composition structures to model Tier 2 / Tier 3. This indirectly validates the concepts of Tier 1.

## 2.4 Modeling Foundation Guidelines

The following list of pages and their subpages provide the set of fundamental principles in RobMoSys.

- Wiki Page on "**General Principles**":
    - Wiki Page on "**Separation of Levels and Separation of Concerns**"
    - Wiki Page on "**Architectural Patterns**"
    - Wiki Page on "**Ecosystem Organization**"
    - Wiki page on "**Tier 1 in Detail**"
    - Wiki Page on "**User-Stories**"
    - Wiki Page on "**PC Analogy: Explaining RobMoSys by the example of the PC domain**"
- Wiki Page on "**Your Role in the RobMoSys Ecosystem**":
    - Wiki Page on "**Roles in the Ecosystem**"

## 2.5 Meta-Meta-Model Structures

The following list of Wiki pages and their subpages provide the first set of Meta-Meta-Model structures. This D2.1 document focuses on the two topmost layers within Tier-1 whereas the document D2.2 (besides other content) covers the lowest layer in Tier-1 namely the RobMoSys composition structures and views.

- Wiki Pages on "**Tier 1 Structure**"
    - Wiki Pages on "**Scientific Grounding: Hypergraph and Entity-Relation model**"
    - Wiki Pages on "**Block-Port-Connector**"

# 3 Appendix

A snapshot as of June 29th, 2018 of the RobMoSys Wiki is attached in the appendix for simple printing. The snapshot can be accessed online via http://robmosys.eu/wiki-sn-02. The live version of the wiki can be found at http://www.robmosys.eu/wiki**.**

# RobMoSys Wiki

RobMoSys enables the composition of robotics applications with managed, assured, and maintained system-level properties via model-driven techniques. It establishes structures that enable the management of the interfaces between different robotics-related domains, different roles in the ecosystem, and different levels of abstractions. Documents that provide an overview and introduction:

- “Section 1 / Excellence”: excerpt of RobMoSys Grant Agreement, Annex 1 (part B) .
- Presentation of the RobMoSys project at European Robotics Forum 2017, Edinburgh .
- Presentation “Modeling Principles and Modeling Foundations” at the RobMoSys Brokerage Day, July 5th 2017, Leuven .

The **RobMoSys Wiki** provides technical details on the RobMoSys approach including examples realizing the RobMoSys structures. The main philosophy behind the RobMoSys Wiki is to favour early access, openness, and transparency over completeness. This is to support communication of RobMoSys being a community endeavour. For general information about the RobMoSys project or its open calls, please refer to the project website [http://www.robmosys.eu].

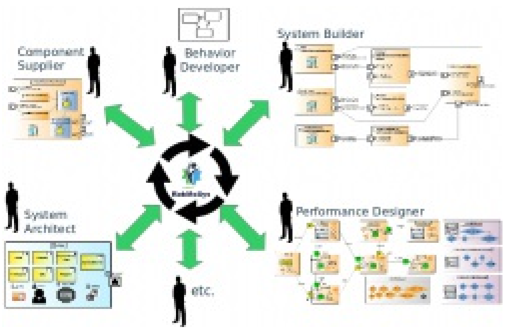
*Please note: The RobMoSys consortium is continuously updating this wiki to provide early insights. See the changelog. If you came here through a RobMoSys document, please see the jumppage to find referred pages. This is a live and evolving wiki, stable snapshots are available.*

## Glossary and FAQ

The glossary contains descriptions of used terms. The technical FAQ provides answers to frequently asked questions.

## Your Role in the RobMoSys Ecosystem

Start reading here to see what your role is in the RobMoSys ecosystem or learn more about Roles in the Ecosystem. Main ecosystem users are:



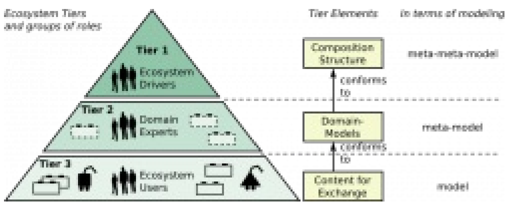
- Behavior Developer
- Component Supplier
- Function Developer
- Performance Designer
- Safety Engineer
- Service Designer
- System Architect
- System Builder

Besides the ecosystem participants, there are also other roles like the Model-Driven Engineering tool developers (see RobMoSys Composition Structures) and framework builders (see Software Baseline). Read a quick introduction to the role of open call applicants in the project-level FAQ [http://robmosys.eu/faq/#1501224896192-8bac1f66-275f].

---

## General Principles

RobMoSys manages the interfaces between different roles and separates concerns in an efficient and systematic way by making the step change to a set of fully model-driven methods and tools for composition-oriented engineering of robotics systems. The following list of pages provide some fundamental principles in RobMoSys.

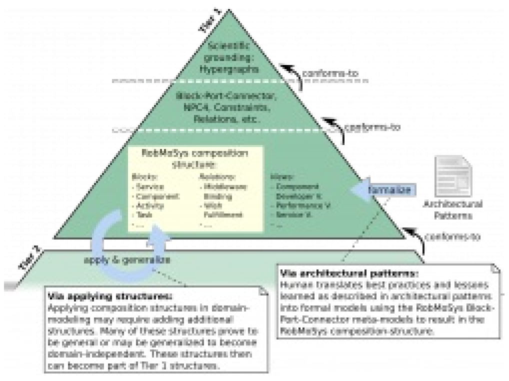


- Separation of Levels and Separation of Concerns
- Architectural Patterns
- Ecosystem Organization and Tiers
- User-Stories
- PC Analogy: Explaining RobMoSys by the example of the PC domain

---

## Tier 1: Modeling Foundations

RobMoSys considers Model-Driven Engineering (MDE) as the main technology to realize the so far independent RobMoSys structures and to implement model-driven tooling. The wiki pages below collect some basic modeling principles related to realizing the RobMoSys structures.


Tier 1
Scientific grounding: Hypergraphs
conforms-to
Block-Port-Connector, NPC4, Constraints, Relations, etc.
conforms-to
RobMoSys composition structure:
Blocks:
- Service
- Component
- Activity
- Task
- ...
Relations:
- Middleware Binding
- Wish Fulfillment
- ...
Views:
- Component Developer V.
- Performance V.
- Service V.
- ...
formalize
Architectural Patterns
conforms-to
Tier 2
apply & generalize
Via applying structures:
Applying composition structures in domain-modeling may require adding additional structures. Many of these structures prove to be general or may be generalized to become domain-independent. These structures then can become part of Tier 1 structures.
Via architectural patterns:
Human translates best practices and lessons learned as described in architectural patterns into formal models using the RobMoSys Block-Port-Connector meta-models to result in the RobMoSys composition-structure.

- Roadmap of MetaModeling
- Modeling Principles
  - Modeling Twin
  - Realization Alternatives
- Tier 1 Structure
  - Scientific Grounding: Hypergraph and Entity-Relation model
  - Block-Port-Connector
  - RobMoSys Composition Structures (and metamodels)
  - Views which are used by roles

---

## Tier 2: Examples of Domain Models

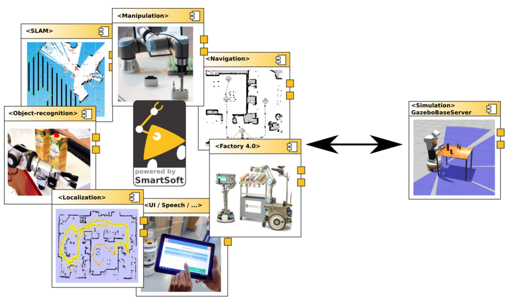
RobMoSys allows the definition of domain-specific models and structures at composition Tier 2. To illustrate this concept, RobMoSys defines the following extendable content for Tier 2.

- Motion, Perception, Worldmodel Stack
- Flexible Navigation Stack
- Active Object Recognition
- etc.

---

## Tools and Software Baseline

RobMoSys provides a set of tools and a software baseline that already conform to the RobMoSys approach. This set can serve as a starting-point for implementations or demonstrations.



### Tooling Baseline

- Roadmap of Tools and Software
- Development Environments and Tools
  - SmartSoft World
  - Papyrus for Robotics
  - to be extended

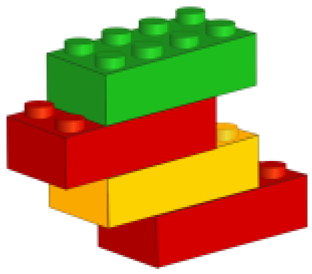
### Tier 3: Existing Building Blocks and Scenarios

- Components
  - SmartSoft Components
- Scenarios and Systems
  - Gazebo/Tiago/SmartSoft Scenario
  - Cause-Effect-Chain Example Scenario

---

## Composition in an Ecosystem

RobMoSys adopts a composition-oriented approach to system integration that manages, maintains and assures system-level properties, while preserving modularity and independence of existing robotics platforms and code bases, yet can build on top of them.



- Introduction to Composition in an Ecosystem
- We illustrate composition by:
  - Task-Level Composition for Robotic Behavior
  - Service-based composition of software components
  - Composition of algorithms
  - Managing Cause-Effect Chains in Component Composition

- Coordinating Activities and Life Cycle of Software Components

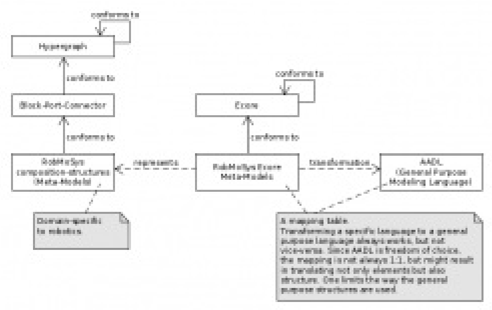
## Pilots: Demonstrating the RobMoSys Approach

RobMoSys uses pilots to demonstrate the use of its approach through the development of full applications with robots. Pilots span different domains and different kind of applications. The pilots can be provided to project contributors to support designing, developing, testing, benchmarking and demonstrating their contribution.

- Goods Transport in a Company:
  - Intralogistics Industry 4.0 Robot Fleet Pilot
- Mobile Manipulation for manufacturing applications on a product line:
  - Flexible Assembly Cell Pilot
  - Human Robot Collaboration for Assembly Pilot
- Mobile manipulation for assistive robotics in a domestic environment or in care institutions:
  - Assistive Mobile Manipulation Pilot
- Modular Educational Robot Pilot

The project is open for constructive suggestions from the community for further pilots or extensions to existing pilots, as long as “platform”, “composability” and “model-tool-code” are first-class citizens of those suggestions.

## Other Approaches in the RobMoSys Context


conforms to
Hypergraph
conforms to
Block-Port-Connector
conforms to
RobMoSys composition-structures (Meta-Models)
Domain-specific to robotics
conforms to
Ecore
conforms to
represents
RobMoSys Ecore Meta-Models
transformation
AADL (General Purpose Modeling Language)
A mapping table.
Transforming a specific language to a general purpose language always works, but not vice-versa. Since AADL is freedom of choice, the mapping is not always 1:1, but might result in translating not only elements but also structure. One limits the way the general purpose structures are used.

RobMoSys follows a reuse-oriented approach. This means that reinvention should be kept to a minimum and existing approaches should be used wherever possible. The following list provides some common approaches that are considered relevant within the RobMoSys context.

- General Purpose Modeling Languages (SysML/UML) and Dynamic-Realtime-Embedded (DRE) domains (AADL, MARTE, etc.)
- Robotics Approaches (ROS, YARP, RTC, etc.)
- Middlewares (DDS)
- Industry 4.0 domain: OPC UA

start · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/start

# RobMoSys Glossary

The glossary contains descriptions of used terms.

## General Terms

### Ecosystem

A collaboration model (cf. Bosch2010[1], Iansiti2004[2] ), which describes the many ways and advantages in which stakeholders (e.g. experts in various fields or companies) network, collaborate, share efforts and costs around a domain or product.

Robotics is a diverse and interdisciplinary field, and contributors have dedicated experience and can contribute software building blocks using their expertise for use by others and system composition.

Participants in an ecosystem do not necessarily know each other, thus the challenge is to organize the contributions without negotiating technical agreements and without adhering to a synchronized development process to organize the contributions.

See Ecosystem Organization

### Digital Platform

There are two different definitions of digital platforms:

- Economical Definition: Multi-sided market gateways creating value by enabling interaction between two or more complementary customer groups.
- Innovation Definition: Reference architecture/implementation with an innovation ecosystem triggering broad value creation.

Platform is not to be confused with the MDA's [http://www.omg.org/mda/] definition. This definition relates to a concrete technology (in most cases referring to a communication middleware technology such as e.g. CORBA).

The term “Platform” is also used in RobMoSys with respect to the target deployment platform / robot platform. See Platform Metamodel. This is not to be confused with the “Digital Platform”.

### System Composition (Activity)

The action or activity of putting together a service robotics application from existing building blocks (here: software components) in a meaningful way, flexibly combining and re-combining them depending on the application's needs.

See also: System Composition in an Ecosystem

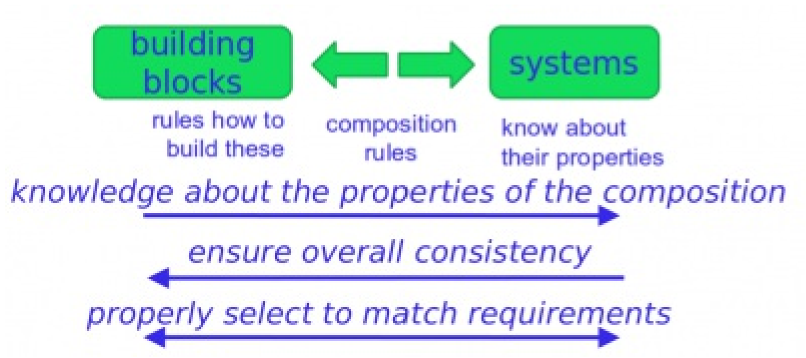
### System Integration (Activity)

The activity that requires effort to combine components, requiring modification or additional action to make them work with others (see Petty2013[3]).

We distinguish integration as an activity and integration as in “integration-centric”.

See also: System Composition in an Ecosystem

## Composability


building blocks
systems
rules how to build these
composition rules
know about their properties
knowledge about the properties of the composition
ensure overall consistency
properly select to match requirements

- The ability to combine and recombine building blocks *as-is* into different systems for different purposes in a meaningful way.

- It is the basic prerequisite for system composition since it is the property that makes *parts* become *building blocks*. Composability has aspects both between components (parts) and the application (whole). Composability comprises syntactic and semantic aspects.

- Composability requires that properties of sub-system are invariant (“remain satisfied”) under composition

- Splittability is “inverse” relationship of composability

## Compositionality

- The ability to compose different modules in a methodological way in order to meet predictable functional and extra-functional requirements.
- Compositionality is a system-level design concern, that reflects the extent to which system designers are able to predict the behaviour of their system on the basis of the formally known behaviour of each of the system’s components.

## Component

A component is the unit of composition that provides functionality to the system through formally defined services at a certain level of abstraction (cf. Szyperski2002[4) ).

A component holds the implementation to bridge between services and functions. A component is defined through a component model and can realize one or more services and interacts with others through services only. When speaking of components, we refer to explicit software components as in the SmartSoft World, in contrast to *component* as a synonym for an arbitrary piece or element of something (as e.g. in AADL [http://www.aadl.info/]).

A component comprises several levels. It is the unit of composition that is being exchanged in the ecosystem.

See also:

- Architectural Pattern Software Components
- Component Metamodel
- Component Supplier role
- Component Development View

## Service

A service can be defined in two different ways:

- a service in the sense of service-oriented architectures (SOA) that provides a self-contained business functionality to a consumer independent of its realization
- one concrete form of a service that is targeted at composition of software components for robotics (see Service Level)

**See also**:

- Communication Pattern

## System

A combination of interacting elements organized to achieve one or more stated purposes. [5]

## System-of-systems

Any system should, in itself, be usable as a building block in a larger system-of-systems. In other words, being a component or a system is not an inherent property of any set of software pieces that are composed together in one way or another.

## Architecture

An organizational structure of a system that describes the relationships and interactions between the system's elements. Architectural aspects can be found at different levels of abstraction.

## Extra-Functional Properties

Extra-functional properties (see Sentilles2012[6] ) are system-level requirements that rule the way in which the system must execute a function, considering physical constraints as time and space. Typical extra-functional properties specify constraints on progress, frequency of execution, maximum time for the execution, mean time between failures, etc.

#### Synonyms

- non-functional properties

## Modeling Twin

A modeling twin describes the packaging of a software/hardware artefact with its model-based representation in order to ship it as a whole (i.e. bundle) to other participants in an ecosystem. The model part of the modling twin is mandatory while the software/hardware part is optional (depending on the current artefact at hand).

See: Modeling Twin

## View

RobMoSys foresees the definition of modeling views that cluster related modeling concerns in one view, while at the same time connecting several views in order to be able to define model-driven tooling that supports in the design of consistent overall models and in communicating the design intents to successive developer roles and successive development phases.

In this sense, a view establishes the link between primitives in the RobMoSys composition structures and the RobMoSys roles. Views enable roles to focus on their responsibility and expertise only. The RobMoSys composition structures ensure composability of building blocks contributed and used by the role.

See: RobMoSys Views

### Engineering Model

In contrast to Scientific Modelling [https://en.wikipedia.org/wiki/Scientific_modelling], engineering models additionally need to be machine-processable in order to enable composition and usage of this model in other models. This is a fundamental feature that improves scalability and modularity of models and model-driven engineering methods. In other words, engineering models always need to provide a benefit and serve a clear purpose with respect to all the other surrounding models of the overall system where this model is part of.

### Activity (in a RobMoSys software component)

The entity that handles the execution of business logic within a component and manages continuous and one-shot operations. In many operating systems activities are mapped to preemptive threads that can be executed concurrently on a CPU core. In some contexts threads are also called tasks, however, this term is to be avoided for this kind of entity within the RobMoSys context as it is reserved for (behavior) tasks (see Task Level).

See Coordinating Activities and Life Cycle of Software Components

### Mission (Level)

See Separation of Levels and Separation of Concerns

### Task (as in task plot for robotic behavior or as in task level)

Is an abstract action (i.e., a job) that a robot is able to perform (see Task Level). Please note, that this term **does not** refer to an operating system thread (which is called **activity** in RobMoSys).

#### Synonyms

- job

### Skill (Level)

See Separation of Levels and Separation of Concerns

### Service (Level)

See Separation of Levels and Separation of Concerns

### Function (Level)

See Separation of Levels and Separation of Concerns

### Execution Container (Level)

See Separation of Levels and Separation of Concerns

### Operating System and Middleware (Level)

See Separation of Levels and Separation of Concerns

### Hardware (Level)

See Separation of Levels and Separation of Concerns

### SmartSoft / The SmartSoft World

An umbrella term for concepts, tools (e.g. the SmartMDSD Toolchain), and content (e.g. software components) that are developed at the Service Robotics Research Center Ulm (Service Robotics Ulm). The latest generation of the SmartSoft world adheres to the RobMoSys structures. See The SmartSoft World.

### Communication Pattern

The semantics in which software components exchange data over component services. RobMoSys adopts a set of few but sufficient communication patterns.

**See also**:

- Service

## General Principles

### Separation of Roles

A principle that enables and supports different groups of stakeholders in playing their role in an overall development workflow without being required to become an expert in every field (in what other roles cover).

A role has a specific view on the system at an adequate abstraction level using relevant elements only.

It is closely related to separation of concerns and a necessary prerequisite for system composition towards an robotics ecosystem.

### Separation of Concerns

A principle in computer science and software engineering that identifies and decouples different problem areas to view and solve them independent from each other (see Dijkstra1982[7]).

It is the basis for separation of roles and a necessary prerequisite for system composition towards an robotics ecosystem.

### Freedom OF choice vs. freedom FROM choice

System development tools generally follow one of the two following approaches:

- One approach is called freedom **of** choice. One tries to support as many different schemes as possible and then leaves it to the user to decide which one best fits his needs. However, that requires huge expertise and discipline at the user side in order to avoid mixing noninteroperable schemes. Typically, academia tends towards preferring this approach since it seems to be as open and flexible as possible. However, the price to pay is high since there is no guidance with respect to ensuring composability and system level conformance.
- Freedom **from** choice (see Lee2010[8]) gives clear guidance with respect to selected structures and can ensure composability and system level conformance. However, there is a high responsibility in coming up with the appropriate structures such that they do not block progress and future designs.

### Architectural Pattern

- A selection of a (sub)set of concerns and levels to fulfill an objective
- An architectural pattern addresses a single level, may connect two related levels or may involve several levels

- See Architectural Patterns
- e.g. extra-functional property

**Objectives for Architectural Patterns**

- Facilitate building systems by composition
- Support Separation of Roles

### Block, Port and Connector

A recurring principle for structuring meta-models at different levels of abstraction. It can be applied on the same level and between different levels.

See Block-Port-Connector

## Concerns

### Computation (Concern)

Computation is related to active system parts that consume CPU time

### Communication (Concern)

Communication concerns the exchange of information between related entities on the same level and also between the levels themselves

### Coordination (Concern)

- Design and modeling of robot behaviors
  - i.e. what happens when and who is involved
- it includes:
  - execution order, (system) states
  - error-handling, resp. error propagation
  - run-time adaptation and (online) reconfiguraiton
  - contingency handling and adaptation rules and strategies

### Configuration (Concern)

- Configuration involves several entities (in contrast to parametrization which typically involves one entity)
  - for example: a set of components (path planning, localization, motion execution) that is configured to work together (move to a destination)
- includes static/dynamic parameter-settings of individual components
- includes static/dynamic wiring between interacting components

### Cross-Cutting Concern

A concern that cannot be separated from others or decomposed and influences or affects multiple properties and areas in a system possibly at different levels of abstraction. For example, security cannot be considered in isolation and cannot be added to a given application by introducing a security-module; it rather has to be considered in all areas of the system.

**Example**

- Non-Functional Properties involve several concerns

## Roles

A certain task or activity with associated concerns that someone (individual, group or organization) takes in the composition-workflow using a view. For example, the Component Supplier role uses the Component Development View view to come up with a component model that conforms to the Component Metamodel.

Someone that takes a particular role typically is an expert in a particular field (e.g. object recognition). A role takes a particular perspective or view on the overall workflow or application. It is associated with certain tasks, duties, rights, and permissions which do not overlap with other roles.

A role has a specific view on the system at an adequate abstraction level using relevant elements only. A role is responsible for supplying a part of the system. “Role” in the sense of a participant of the ecosystem.

See also:

- Roles in the Ecosystem
- RobMoSys Views

## Acknowledgement

This document contains material from:

- Lotz2018 Alex Lotz, "Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System", Dissertation, Technische Universität München, München 2018. [https://mediatum.ub.tum.de/?id=1362587]
- Stampfer2018 Dennis Stampfer, "Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics". Dissertation, Technische Universität München, München, Germany, 2018. [http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2]
- Lutz2017 Matthias Lutz, “Model-Driven Behavior Development for Service Robotic Systems: Bridging the Gap between Software- and Behavior-Models,” 2017. (unpublished work)

## References

1)

Jan Bosch, Petra Bosch-Sijtsema. “From integration to composition: On the impact of software product lines, global development and ecosystems”, in Journal of Systems and Software, Volume 83, Issue 1, January 2010, Pages 67-76, ISSN 0164-1212, DOI: 10.1016/j.jss.2009.06.051 [http://doi.org/10.1016/j.jss.2009.06.051]

2)

Iansiti, Marco, and Roy Levien. “Strategy as Ecology”, in Harvard Business Review 82, no. 3 (March 2004).

3)

Mikel D. Petty and Eric W. Weisel. “A Composability Lexicon”, in Proc. Spring 2003 Simulation Interoperability Workshop, March 2003, Orlando, USA.

4)

Clemens Szyperski. “Component Software: Beyond Object-Oriented Programming (2nd ed.)”. In Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

5)

ISO/IEC 15288:2008 (IEEE Std 15288-2008

6)

Séverine Sentilles. “Managing Extra-Functional Properties in Component-Based Development of Embedded

Systems”. Dissertation. Mälardalen University, Västerås, Sweden, 2012.

7)

E. W. Dijkstra. “On the role of scientific thought”. In Selected Writings on Computing: A Personal Perspective, pages 60–66. Springer-Verlag, 1982.

8)

Edward A. Lee. “Disciplined Heterogeneous Modeling”. In: MODELS 2010. Invited Keynote Talk. Oslo, Norway, 2010.

---

glossary · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/glossary

# Architectural Patterns

## Introduction

Buschmann et. al.[1] provides the following descriptive definition of a pattern in general:

> "A pattern describes a particular recurring design problem that arises in specific design contexts, and presents a solution to it. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate." [2]

Moreover, Buschmann et. al.[3] lists some common properties of a pattern:

- "Patterns document existing, well-proven design experience."
- "Patterns provide a common vocabulary and understanding for design principles."
- "Patterns support the construction of software with defined properties."
- "Patterns help you build complex and heterogeneous software. Patterns help you manage software complexity."

The proposed scheme by Buschmann for describing a software pattern consists of a **Context**, **Problem** and the **Solution**. This triple is used below to also describe individual architectural patterns which analogously address recurring design problems in robotics software development, each occurring in a specific design context, and present a well-proven solution to the design problem. There are two fundamental objectives that drive the design of all presented architectural patterns, namely:

- Facilitate building systems by composition
- Support Separation of Roles

Each architectural pattern needs to contribute towards these two objectives.

## Template for an Architectural Pattern

This is a template for describing an architectural pattern including the required sections that the description must comprise.

### Context

A context describes a situation in which the design problem occurs. Also relate the context to:

- the Levels and Concerns
- involved Roles

### Problem

This part describes a **recurring problem** that repeatedly arises in a given context. This can start with a general, open ended problem and get more concrete with **driving forces** and concrete **requirements** that the solution must fulfill. Also, **constraints to consider** and **desired properties** of the solution can be expressed here.

### Solution

The solution describes how the problem is solved, thereby balancing the driving forces. In some cases, available technologies can be listed here that solve the given problem.

### Optional: Discussion

Any discussion of shortcomings, differences or references to other patterns can be described here.

### Optional: Example(s)

Specific scenarios or technologies that help to understand the problem and/or solution can be listed here.

## List of Architectural Patterns

(alphabetical order)

- Architectural Pattern for Communication
- Architectural Pattern for Component Coordination
- Architectural Pattern for Software Components
- Architectural Pattern for Managing Transitions of System States
- Architectural Pattern for Task-Plot Coordination (Robotic Behaviors)
- Architectural Pattern for Service Definitions
- Architectural Pattern for Stepwise Management of Extra-Functional Properties

### Further Candidates for Architectural Patterns

- Architectural Pattern for Coordination-Frame Transformation
    - Transformation tree (e.g. TF in ROS, Time-Stamps, Pose-Stamps, etc.)
- Subsidiarity Principle
    - at any time a clear control hierarchy
    - delegate decision spaces top-down in the hierarchy
- Knowledge Representation
    - central Knowledge Base
    - synchronize and conflate distributed system-models over global IDs
- Reservation based Resource Management
    - in KB through Tasks and Skills for coordination of Components

1) , 2) , 3)
Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. “Pattern-Oriented Software Architecture, Volume 1, A System of Patterns”. Wiley Press, 1996, ISBN: 978-0-471-95869-7 [http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0471958697.html]

general_principles:architectural_patterns:start · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/general_principles:architectural_patterns:start

# Architectural Pattern for Stepwise Management of Extra-Functional Properties

## Context

Besides of “pure” functions, realistic systems also need to specify and to manage extra-functional properties that might involve different system parts at different levels of abstraction. Extra-functional system properties specify how well a system performs given a certain system configuration.

There are two main developer roles that are involved in the specification of extra-functional properties:

- Component Supplier specifies functional constraints of individual building blocks (i.e. components)
- System Builder defines extra-functional properties within the predefined boundaries by the involved components

Extra-functional properties are cross-cutting in nature (i.e. combining communication, computation and coordination) and relate to several levels of abstraction:

- Task Plot (level) provides the run-time context for the extra-functional properties
- Service (level) link components and is mainly related to the communication concern of extra-functional properties
- Function (level) is related to the computation concern of extra-functional properties
- Execution Container (level) relates to the coordination concern of extra-functional properties
- Hardware (level) finally does both, computation and communication of extra-functional properties

## Problem

- Extra-functional system properties such as e.g. end-to-end response times are cross-cutting in nature and typically involve knowledge and contributions from different developer roles (e.g. component developers and system builders) who are often working independently in different places and at different points in time. This easily leads to inconsistencies in the system. Resolving inconsistencies typically requires expert knowledge and deep insights into all the distributed system parts
- Extra-functional properties bridge between functional constraints in individual building blocks and application-specific system requirements
- Extra-functional properties might be grounded in several system parts that are distributed over several components
- Tracing and assuring extra-functional properties might involve additional (dedicated) analysis tools

## Solution

- The specification of functional aspects of individual building blocks must be linked with the definition of application-specific, extra-functional system aspects on model level
- Individual building blocks specify functional constraints that restrict the remaining design space to be exploited for a later system design
- System-specification allows only those design options that do not conflict with the individual building-block constraints

- Dedicated analysis tools simulate run-time conditions and predict extra-functional system behavior (i.e. the run-time performance quality of a system)
- Optionally: a run-time monitoring mechanism can assure compliance with specified extra-functional properties

## Example

End-to-end response time from sensing until acting in a service robot can be considered as one particular extra-functional property

- this end-to-end response time typically involves several interconnected components forming a data-flow chain of components
- each component in a chain contributes with a certain delay to the overall end-to-end time
- the component's internal delay might be the result of the internally used device driver with certain execution characteristics or otherwise result from the internally configured activities (i.e. tasks/threads)
- individual components should leave as much configuration freedom as possible and only specify really needed functional constraints (such as an unchangeable device driver behavior)
- a specified system-level end-to-end response time needs to be checked with respect to predefined functional constraints in individual components and the overall end-to-end run-time behavior of the entire chain of components
    - for analysing the run-time behavior of the entire chain of components at design-time, dedicated, matured and powerful analysis tools such as SymTA/S can be used
    - run-time behavior can also be directly monitored in an executed robotic system using a dedicated monitoring infrastructure

This example is described with more details in a dedicated wiki page: Managing Cause-Effect Chains in Component Composition.

## Acknowledgement

This document contains material from:

- Lotz2018 Alex Lotz, "Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System", Dissertation, Technische Universität München, München 2018. [https://mediatum.ub.tum.de/?id=1362587]
- Lutz2017 Matthias Lutz, "Model-Driven Behavior Development for Service Robotic Systems: Bridging the Gap between Software- and Behavior-Models," 2017. (unpublished work)
- Stampfer2018 Dennis Stampfer, "Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics". Dissertation, Technische Universität München, München, Germany, 2018. [http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2]

# Architectural Pattern for Software Components

## Context

- A common way to handle system complexity is Component-Based Software Engineering
- Individual components are composable building-blocks that can be (re-)used in different applications (i.e. systems)
- Components in a system are not independent of each other but need to exchange data
- Interconnected components realize (and collaboratively execute) overall system functions (e.g. the navigation stack)

Modeling and developing a software component is the main responsibility of Component Suppliers.

This architectural pattern relates to the following abstraction levels:

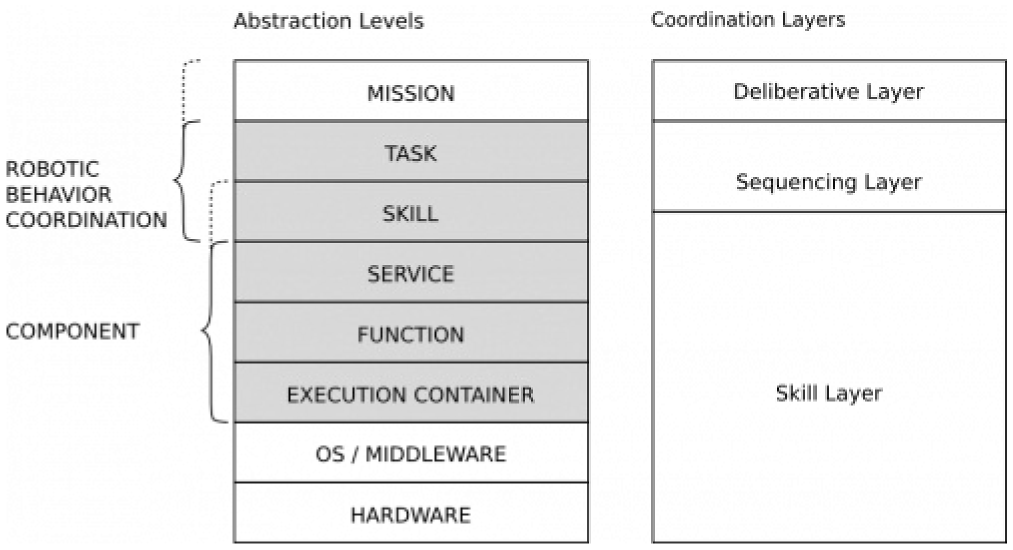
- Skill (level) requires a coordination interface for each component
- Service (level) specifies interaction points to other components (i.e. the communication concern)
- Function (level) realizes the component’s internal functionality
- Execution Container (level) links functionality with the execution platform (i.e. the computation concern)
- Hardware (level) allows to directly interact with sensors and/or actuators within a component

## Problem

- The overall system behavior at run-time is the result of sets of interconnected components that need to be executed in a systematic and deterministic way.
- Real-world environments are open-ended and unpredictable in nature which requires a certain adaptability and flexibility of the robot system behavior.
  - System flexibility in turn requires run-time reconfigurability of individual components. Configuration options of individual components might involve design-time and run-time configurability and depend on the internal (i.e. functional) realization of a component.
- There are cases where several provided services might need to be realized in a single component (e.g. because the used library cannot be separated into several components)
- The overall role of a component is manifold:
  - to realize a coherent set of provided services
  - to specify dependencies to other services (provided by other components)
  - to encapsulate (i.e. decouple) the functional (internal) realization of services from their general representation on system level
  - to specify allowed configuration options and possible run-time modes (i.e. to be used from the skill level)
  - to hide platform-related details such as communication middleware, operating system and internally used device drivers (i.e. mapping to the execution container and interacting with sensors/actuators)

## Solution

The concept of a component spans across several abstraction levels:



From a functional point of view, a component spans over "Execution Container", "Function", "Service" and optionally also the "Skill" levels. From the robotic behavior coordination point of view, a component is on the level of robotic skills[1].

A flexible component model that allows different bundlings of several provided services and that decouples the service definition from its realization within a component:

- a component can realize more than one provided service but a certain provided service is realized by exactly one distinct component
- a component should implement or use a service but not define it (service definition is a separated step)

In addition to the "regular" services a component also implements a generic configuration and coordination interface that provides access to:

- the component's life-cycle state automaton
- admissible run-time modes (i.e. activity states)
- the component's configuration parameters (i.e. allowed parameter sets)
- the coordinated dynamic wiring of component's services (i.e. without conflicting with the component's internal functionality)

See also:

- Component metamodel

## Acknowledgement

This document contains material from:

- Lotz2018 Alex Lotz, "Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System", Dissertation, Technische Universität München, München 2018. [https://mediatum.ub.tum.de/?id=1362587]
- Lutz2017 Matthias Lutz, "Model-Driven Behavior Development for Service Robotic Systems: Bridging the Gap between Software- and Behavior-Models," 2017. (unpublished work)
- Stampfer2018 Dennis Stampfer, "Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics". Dissertation, Technische Universität München, München, Germany, 2018. [http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2]

[1]

R. Peter Bonasso, R. James Firby, Erann Gat, David Kortenkamp, David P. Miller, and Mark G. Slack. "Experiences with an architecture for intelligent, reactive agents". In: *Journal of Experimental & Theoretical Artificial Intelligence*, Volume 9, 1997, DOI:10.1080/095281397147103 [http://dx.doi.org/10.1080/095281397147103].

# Architectural Pattern for Managing Transitions of System States

(To be extended)

- (i.e. System-Mode Transitions)
- synchronize system-modes over shared IDs
- recognize (i.e. awareness about) transitive system-states

## Context

…

## Problem

…

## Solution

…

## Discussion

…

## Acknowledgement

This document contains material from:

- Lotz2018 Alex Lotz, "Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System", Dissertation, Technische Universität München, München 2018. [https://mediatum.ub.tum.de/?id=1362587]
- Lutz2017 Matthias Lutz, “Model-Driven Behavior Development for Service Robotic Systems: Bridging the Gap between Software- and Behavior-Models,” 2017. (unpublished work)
- Stampfer2018 Dennis Stampfer, "Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics". Dissertation, Technische Universität München, München, Germany, 2018. [http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2]

# Architectural Pattern for Communication

## Context

Communication between entities (i.e. exchange of information). Communication is a concern and relates to the following levels:

- Service (level) structures communication
- Execution container (level) provides resources for communication
- Operating System / Middleware (level) realizes communication
- Hardware (level) does communication

This architectural pattern relates to the following roles:

- Service Designer: selects communication pattern (see below)
- System Builder: selects communication middleware

## Problem

- A huge number of communication middlewares
- A huge number of overlapping and conflicting communication schemes
- Requirements that the solution must fulfill:
    - Realize vertical (i.e. layers) and horizontal (e.g. components) exchange of information (with the goal to enable communication, coordination and configuration)
    - Support different schemes for data-flow oriented communication and coordination/configuration concerns
    - At the minimum provide:
        - Publish/Subscribe (i.e. data-flow) communication semantics
        - Request/Response (i.e. on demand) communication semantics
    - Support independence of the underlying middleware solution (i.e. middleware abstraction layer)
    - Reduce the huge variety of overlapping communication semantics in order to improve composability between components
    - Decouple the access to communication within a component (functional-level) from the communication between two interacting components (service-level)

## Solution

An essential set of communication patterns that is rich enough to cover common communication use-cases, yet at the same time reduced enough to support composability.

- CommunicationPatterns (for continuous data transfer)
    - Request/Response
        - e.g. SmartSoft-Query
    - Publish/Subscribe
        - e.g. SmartSoft-Push (sub-variants: PushNewest and PushTimed)
- ConfigurationPattern (for component configuration)
    - Component Parametrization

        - e.g. SmartSoft-Parameter
    - Dynamic Wiring
        - e.g. SmartSoft-Wiring
- CoordinationPattern (for skill realization)
    - Component Lifecycle Automaton
        - e.g. SmartSoft-State (generic lifecycle state automaton)
    - Component (activity) Modes
        - e.g. SmartSoft-State (user-defined states) and SmartSoft-Parameter (trigger)
    - Component Feedback
        - e.g. SmartSoft-Event

See also:

- Communication Patterns

## Discussion

Different middlewares allow for different middleware abstraction levels. For instance, message-based middlewares require a protocol-based abstraction, while e.g. DDS allows for a higher level of abstraction (i.e. directly using the publish/subscribe communication with accordingly preselected QoS attributes). In any case, middleware details should be hidden from both, the component's internal communication access and the communication semantics between components.

The separation of patterns into groups for Communication (i.e. continuous data exchange), Configuration (i.e. parametrization of individual components) and Coordination (i.e. skill-component interaction) provides solutions for recurring communication problems and clarifies the purpose of a particular pattern.

The communication access from within a component (i.e. communication interface access) needs to be as flexible as possible as long as it does not violate with the clearly specified communication semantics outside of the component (resp. between interacting components).

Not every semantic detail needs to be made explicit on model level (some may come from “de-facto standard” implementations). The focus in models need to be on a consistent representation and systematic management of different communication schemes.

## Acknowledgement

This document contains material from:

- Lotz2018 Alex Lotz, "Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System", Dissertation, Technische Universität München, München 2018. [https://mediatum.ub.tum.de/?id=1362587]
- Lutz2017 Matthias Lutz, “Model-Driven Behavior Development for Service Robotic Systems: Bridging the Gap between Software- and Behavior-Models,” 2017. (unpublished work)
- Stampfer2018 Dennis Stampfer, "Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics". Dissertation, Technische Universität München, München, Germany, 2018. [http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2]

# Architectural Pattern for Service Definitions

(To be extended)

- Granularity of components and services
- Abstraction-level of services

## Context

…

## Problem

…

## Solution

…

## Discussion

…

## Acknowledgement

This document contains material from:

- Lotz2018 Alex Lotz, "Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System", Dissertation, Technische Universität München, München 2018. [https://mediatum.ub.tum.de/?id=1362587]
- Lutz2017 Matthias Lutz, "Model-Driven Behavior Development for Service Robotic Systems: Bridging the Gap between Software- and Behavior-Models," 2017. (unpublished work)
- Stampfer2018 Dennis Stampfer, "Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics". Dissertation, Technische Universität München, München, Germany, 2018. [http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2]

# Architectural Pattern for Task-Plot Coordination (Robotic Behaviors)

A description of this architectural pattern can be found here [http://www.servicerobotik-ulm.de/drupal/?q=node/86]. The architecture is a generic architecture for robotics behavior. In terms of the abstraction levels, this pattern addresses task and skill levels; in terms of concerns, it addresses coordination and configuration.

See also:

- Task-Level Composition for Robotic Behavior

## Context

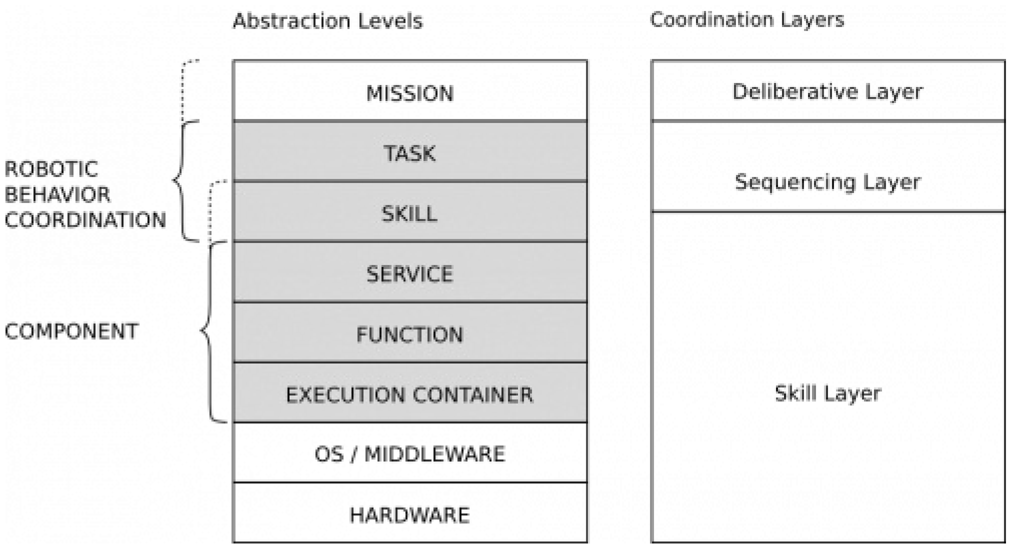
Service robots act in unstructured and open-ended environments that require flexibility and adaptability in execution for the robotic behavior. The basic robot functionality is realized by software components. Software components are typically general software building blocks that are independent of a specific application or scenario. By contrast, the robot’s behavior is highly application-specific and depends on the desired tasks that the robot is supposed to perform and the expected environments where the robot will operate in.

## Problem

- A static sequence of actions is too inflexible for coping with the dynamics of the real world where each single action can fail or can produce unexpected results
    - Robust behaviors require several alternative strategies for performing a task whose combinatorial combination easily explodes if statically designed in advance
- Robot behaviors need to be expressed on different levels of abstraction (i.e. high-level tasks such as e.g. “make coffee” are refined to more specific sub-tasks such as e.g. “approach kitchen”, “operate the coffee machine”, etc.)
- Components are active system parts that continuously exchange data while robot behaviors are event-driven, passive parts that react to events for switching into adequate successive behavior steps (depending on the so far successfully executed actions or failures in execution)
    - A robot behavior bridges between continuous execution in components and event-driven coordination on task plot (level)

## Solution

Robotic Behavior spans across several levels:



- Robotic behavior is about continuous vs. discrete (see here [http://www.servicerobotik-ulm.de/drupal/?q=node/86])
- task-plot description (i.e. hierarchical task-tree)
- using external solvers as experts on demand (i.e. symbolic planer), see deliberative layer here [http://www.servicerobotik-ulm.de/drupal/?q=node/86]

This pattern is supported by the SmartSoft World via SmartTCL [http://www.servicerobotik-ulm.de/drupal/?q=node/84] and Dynamic State Charts [http://www.servicerobotik-ulm.de/drupal/?q=node/87]

## Discussion

…

## Acknowledgement

This document contains material from:

- Lotz2018 Alex Lotz, "Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System", Dissertation, Technische Universität München, München 2018. [https://mediatum.ub.tum.de/?id=1362587]
- Lutz2017 Matthias Lutz, "Model-Driven Behavior Development for Service Robotic Systems: Bridging the Gap between Software- and Behavior-Models," 2017. (unpublished work)
- Stampfer2018 Dennis Stampfer, "Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics". Dissertation, Technische Universität München, München, Germany, 2018. [http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2]

general_principles:architectural_patterns:robotic_behavior · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/general_principles:architectural_patterns:robotic_behavior

# Architectural Pattern for Component Coordination

The here proposed pattern structures and semantically enriches the access of the functionalities within components for coordination by defining a **component coordination interface**. The interface enables the run-time coordination of the components by robotic behavior models on skill and task abstraction level. This interface is the foundation for robotic behavior development and system orchestration (as described in Architectural Pattern for Task-Plot Coordination (Robotic Behaviors)).

## Context

The architectural pattern can be used in the context of coordination of closed software components. The pattern deals only with the concern of coordination and is located at the abstraction level of services, lifting the access to the functionalities within a software component to the skill abstraction level (see Separation of Levels and Separation of Concerns). It involves the roles of theService Designer (Domain Experts), the Component Supplier and the Behavior Developer.

## Problem

Functionalities within closed software components needs to be coordinated to so that the robot as a whole is able to provide a service. The access to the functionalities within the components needs to be on a balanced level to avoid fine grained interaction, so that the user of the software component does not need to know implementation specific details of the component.

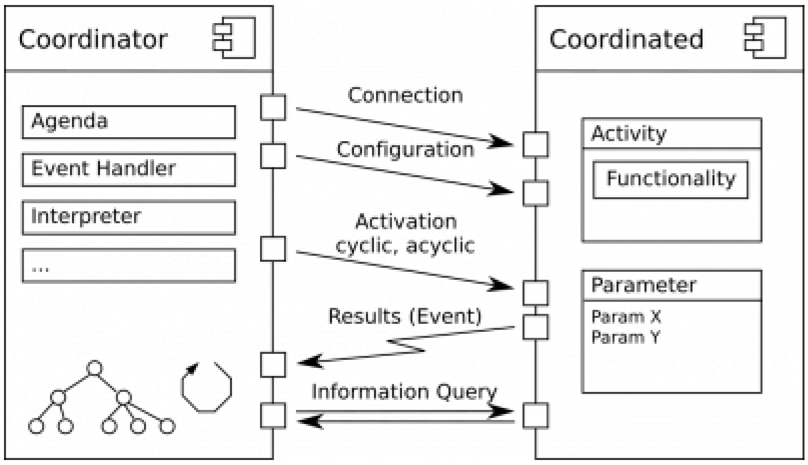
The coordination of the component needs to be possible without binding the behavior models (task level description) to a concrete component.

## Solution

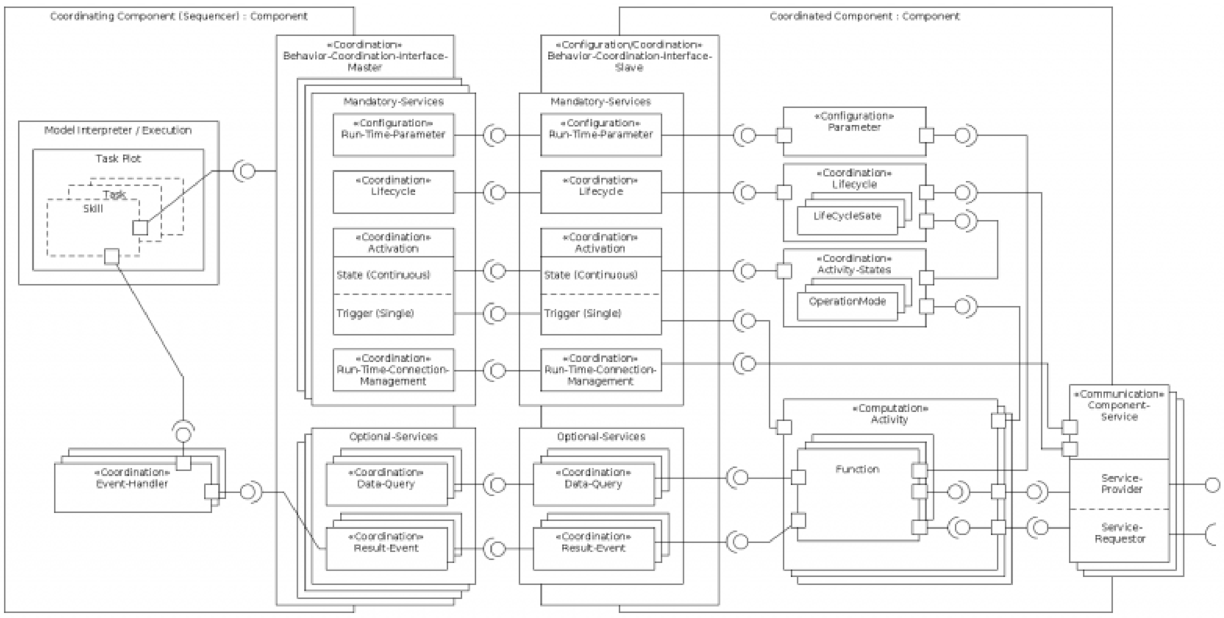
The solution is to define an uniform behavior coordination interface for robotics software components. The interface is two fold: the coordinating component part and the coordinated component part. The coordinating component part is typically realized/implemented by a sequencer component in case of a 3T / three tier architecture (see Architectural Pattern for Task-Plot Coordination (Robotic Behaviors)).

The coordination access to a component via the interface can be grouped into six basic categories, each with a different purpose, semantic and communication mechanism:

- Configuration - Run-Time configuration or parameterization of components, for coordination.
- Activation - Activation of activities and therefor the functionalities within the components.
- Results (Events) - Receiving the results of the activation of the functionalities within the components.
- Connection - Coordination of the inter-component connections and thereby configuring the data flow of the coordinated components.
- Component Life-Cycle - Providing access to components life-cycle e.g. shutdown or error states of the components.
- Information Query - Requesting and receiving information for coordination from components.



The relation of the interface parts to the component parts is shown by the following figure:


Coordinating Component (Sequencer) : Component
Coordinated Component : Component
«Coordination» Behavior-Coordination-Interface-Master
«Configuration/Coordination» Behavior-Coordination-Interface-Slave
Mandatory-Services
Model Interpreter / Execution
Task Plot
Task
Skill
«Configuration» Run-Time-Parameter
«Coordination» Lifecycle
«Coordination» Activation
State (Continuous)
Trigger (Single)
«Coordination» Run-Time-Connection-Management
Optional-Services
«Coordination» Data-Query
«Coordination» Result-Event
«Coordination» Event-Handler
«Configuration» Parameter
«Coordination» Lifecycle
LifeCycleSate
«Coordination» Activity-States
OperationMode
«Computation» Activity
Function
«Communication» Component-Service
Service-Provider
Service-Requestor

The realization of the coordination interface within RobMoSys is done using the Communication/Coordination and Configuration Pattern.

See also:

- Component metamodel

## Discussion

The interface proposed in the pattern harmonizes the coordination access to the components and the functionality encapsulated by them. This allows for the separation of the behavior coordination and behavior models from the functionalities.

## See also

- Architectural Pattern for Task-Plot Coordination (Robotic Behaviors)
- Architectural Pattern for Software Components
- Separation of Levels and Separation of Concerns

## Acknowledgement

This document contains material from:

- Lotz2018 Alex Lotz, "Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System", Dissertation, Technische Universität München, München 2018. [https://mediatum.ub.tum.de/?id=1362587]
- Lutz2017 Matthias Lutz, "Model-Driven Behavior Development for Service Robotic Systems: Bridging the Gap between Software- and Behavior-Models," 2017. (unpublished work)
- Stampfer2018 Dennis Stampfer, "Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics". Dissertation, Technische Universität München, München, Germany, 2018. [http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2]

# Separation of Levels and Separation of Concerns

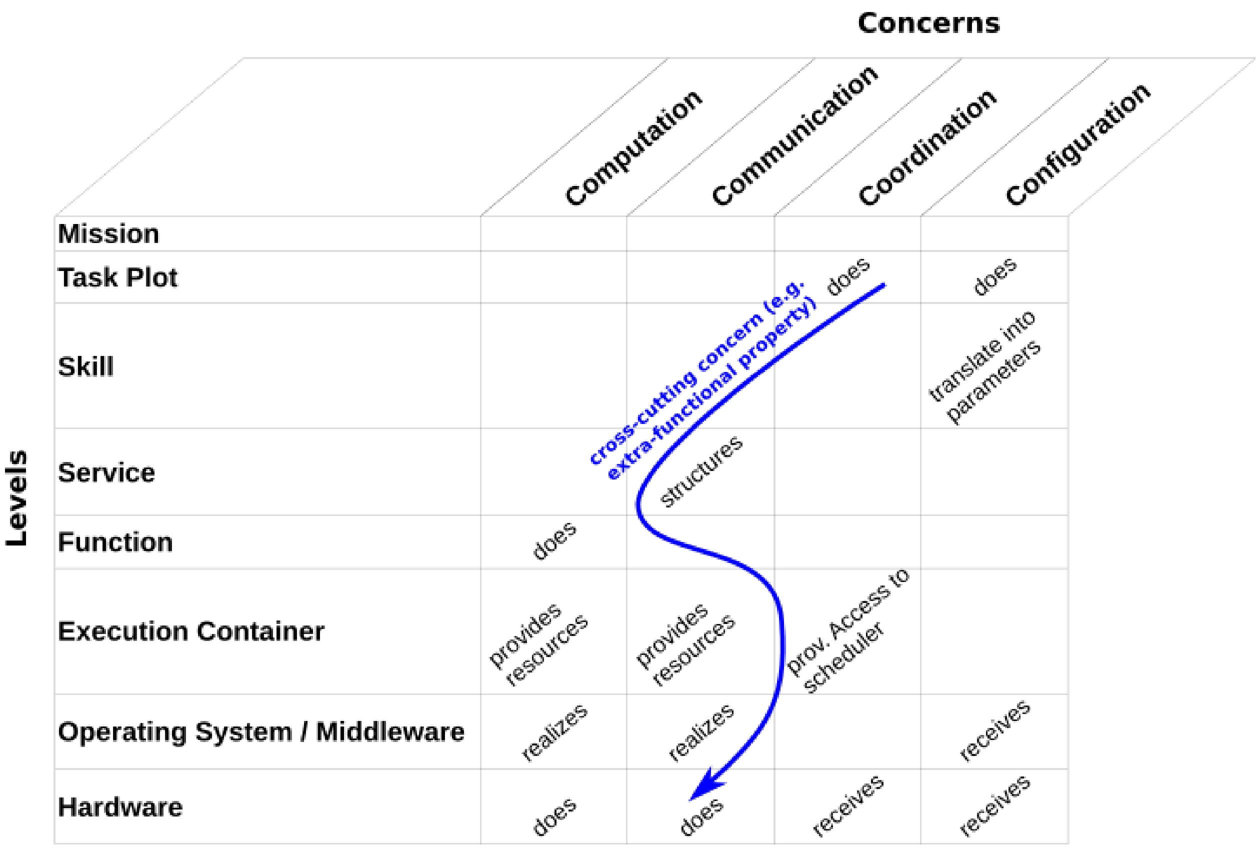
The figure below illustrates the separation of levels and the separation of concerns. Please also refer to the glossary for descriptions of used terms. The levels indicate abstractions in a robotics system.

The levels can be seen as an analogy to “ISO/OSI model” for robotics that addresses additional concerns beyond communication. The analogy is interesting, because ISO/OSI partitions the communication aspect in different levels of abstraction that then help to discuss and locate contributions. The ISO/OSI separations in levels allows to develop efficient solutions for each level. Establishing such levels for robotics would clearly help to communicate between robotics experts–as ISO/OSI does in computer science.

The levels and concerns can be used to identify and illustrate architectural patterns. The blue line in the figure is an abstract example. An architectural pattern combines several levels and several concerns. For example, the architectural pattern for a software component spans across the levels of service, function and execution container.

See also

- Architectural Patterns



## About the Levels

- The lower levels address more concerns and are more cross-cutting in their nature
- The higher levels are more abstract and address less concerns / individual concerns. They thus allow a

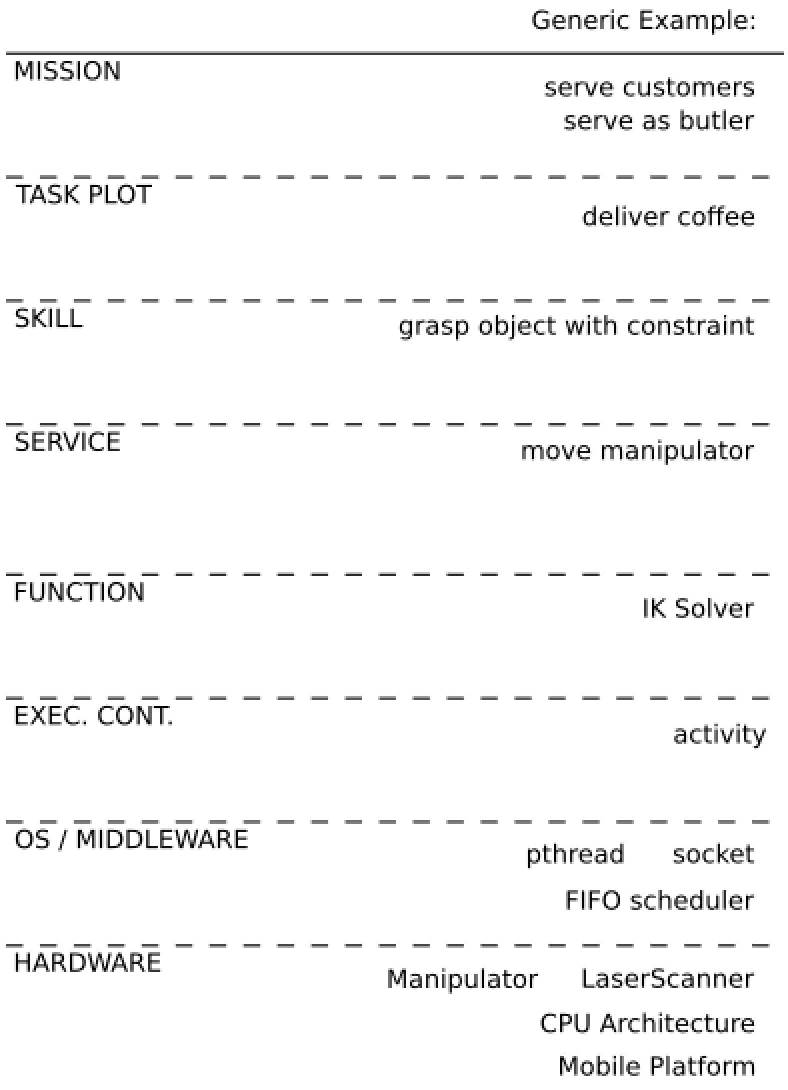
better separation of concerns and separation of roles.
- By definition, a level can not be defined on its own, since its semantics is the relationships between the items at this “level” and those at the other levels. This exercise to get these relationships well-defined is a tough one, this is of high priority though, since “level”/“layer” is one of the most often used term in (software) architecture.
- A layer is on top of another, because it depends on it. Every layer can exist without the layers above it, and requires the layers below it to function. A layer encapsulates and addresses a different part of the needs of much robotic systems, thereby reducing the complexity of the associated engineering solutions.
- A good layering goes for abstraction layers. Otherwise, different layers just go for another level of indirection. An abstraction layer is a way of hiding that allows the separation of concerns and facilitates interoperability and platform independence.

## On the number and separation of levels

- Individual levels always exist but are not always explicitly visible.
- Transition between layers can be fluent
- There are single layer approaches (clear separation between layers offering full flexibility in composition) but also hybrid ones (combining several adjacent layers into one loosing flexibility). For example, ROS1 implemented both the middleware and execution container while in ROS2, the middleware level is planned to be separated.
- Different levels might require different technologies
- Individual levels may also be separated horizontally (e.g. fleet of robots vs. an individual robot, or group of components vs. an individual component)

## Example: Levels

- Below are examples for each of the levels.
- They demonstrate the level of abstraction that can be found in each layer.



## The individual Levels

### Mission (Level)

- A higher level objective/goal for the robot to achieve.
- At run-time, a robot might need to prioritize one mission over another in order to rise the probability of success and/or to increase the overall quality of service

Examples

- In logistics: do order picking for order 45
- serve customer
- serve as butler

Synonyms

- goal
- objective

### Task (Level)

- A task (on the Task level) is a symbolic representation of what and how a robot is able to do something,

independent of the realization.
- A job that is described independent of the functional realization.
- Includes explicit or implicit constraints.
- tasks might be executed in sequence or in parallel
- task-sets might be predefined statically (at design-time) or dynamically generated (e.g. using a symbolic planner)
- tasks might need to be refined hierarchically (i.e. from a high-level task down to a set of low-level tasks)
- not to be confused with tasks in the sense of processes/threads (see Execution Container)
- see also: Task-Level Composition for Robotic Behavior

Examples

- Move to room nr. 26
- Grasp blue cup
- Get a cup from the kitchen
- deliver coffee

Synonyms

- job

See also:

- Architectural Pattern for Task-Plot Coordination (Robotic Behaviors)

### Skill (Level)

Defines basic capabilities of a robot. The area of transition between high-level tasks and concrete configurations and parameterizations of components on the service-level.

Skills enable tasks to become independent of the actual realization in components.

A collection of skills is required for the robot to do a certain task. For example, a butler robot requires skills for navigation, object recognition, mobile manipulation, speaking, etc. A component often implements a certain skill, but skills might also be realized by multiple components.

Skill-level often interfaces between symbolic and subsymbolic representations.

Examples

- An abstract high level task (e.g. move-to kitchen) is mapped to concrete configurations and services that components offer (e.g. parameterize path planning, localization and motion execution components with destination set to kitchen).
- grasp object with constraint

Synonyms

- capability
- system-function

See also:

- Architectural Pattern for Component Coordination

### Service (Level)

A service is a system-level entity that serves as the only access point between components to exchange information at a proper level of abstraction.

Services follow a service contract and separate the internal and external view of a component. They describe the functional boundaries between components. Services consist of communication semantics, data structure and additional properties.

Components realize services and might depend on existence of a certain type of service(s) in a later system.

See also: Service-based Composition

## Function (Level)

- a coherent set of algorithms, for example implemented in libraries, that serve a unique functional purpose
- a piece of software that performs a specific action when invoked using a certain set of inputs to achieve a desired outcome.[1)]

Example

- A function implemented in an library, e.g. OpenCV Blob Finder
- An implemented algorithm, e.g. PID-controller
- Functions developed or modeled in Matlab, Simulink, etc.
- Inverse kinematics (IK) solver

Synonyms

- functional block

## Execution Container (Level)

- provides the infrastructure and resources for the functional level
- provides mappings towards the underlying infrastructure (e.g. operating system, communication middleware).

Examples

- Access to scheduler
- Threads, eventually processes

## Operating System and Middleware (Level)

Example elements on this level: e.g. phread, socket, FIFO scheduler

An Operating System is, for example, responsible for:

- Memory management
- Inter-Process-Communication
- Networking-Stack, e.g. TCP
- Hardware Abstraction Layer

Examples for Operating System

- Linux, Windows
- FreeRTOS, QNX, vxWorks

A (communication) middleware is a software layer between the application and network stack of the operating system. Communication middlewares are very common in distributed systems, but also for local communication between applications. They provide an abstract interface for communication independent of the operating system and network stack.

There are many distributed middleware systems available. However, they are designed to support as many different styles of programming and as many use-cases as possible. They focus on freedom of choice and, as result, there is an overwhelming number of ways on how to implement even a simple two-way communication using one of these general purpose middleware solutions. These various options might result in non-interoperable behaviors at the system architecture level.

For a component model as a common basis, it is therefore necessary to be independent of a certain middleware.

Examples

- OMG CORBA
- OMG DDS
- ACE

### Hardware (Level)

Solid pieces of bare metal that the robot is built of and uses to interact with the physical environment. It includes actors/sensors and processing unit.

Examples

- Sensors: laser scanner, camera
- Actuators: manipulator, robot base/mobile platform
- Processing units: embedded computer, cpu architecture

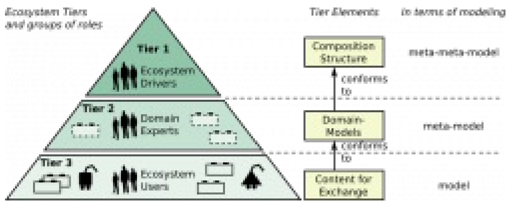
## Acknowledgement

This document contains material from:

- Lotz2018 Alex Lotz, "Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System", Dissertation, Technische Universität München, München 2018. [https://mediatum.ub.tum.de/?id=1362587]
- Lutz2017 Matthias Lutz, “Model-Driven Behavior Development for Service Robotic Systems: Bridging the Gap between Software- and Behavior-Models,” 2017. (unpublished work)
- Stampfer2018 Dennis Stampfer, "Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics". Dissertation, Technische Universität München, München, Germany, 2018. [http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2]

1)
“Systems and software engineering – Vocabulary,” in ISO/IEC/IEEE 24765:2010(E) , vol., no., pp.1-418, Dec. 15 2010 DOI: 10.1109/IEEESTD.2010.5733835https://doi.org/10.1109/IEEESTD.2010.5733835 [https://doi.org/10.1109/IEEESTD.2010.5733835]

# General Principles



RobMoSys manages the interfaces between different roles and separates concerns in an efficient and systematic way by making the step change to a set of fully model-driven methods and tools for composition-oriented engineering of robotics systems. The following list of pages provide some fundamental principles in RobMoSys.

- Separation of Levels and Separation of Concerns
- Architectural Patterns
- Ecosystem Organization and Tiers
- User-Stories
- PC Analogy: Explaining RobMoSys by the example of the PC domain

# Analogy: The PC Domain

We use the analogy of hardware in the PC Domain to illustrate concepts of RobMoSys. Using an analogy, we can describe particular concepts in a given context (the pc domain), which is easier to understand since the context of the PC domain is widely known. One can then transfer information given to the robotics domain. The PC domain is only an analogy that helps to illustrate concepts; the PC domain is different than robotics, so do not read too much into the examples given here.

## Configuration, Composition, and Integration

Using the PC Domain, we illustrate the terms Configuration, Composition, and Integration.

### Configuration

Configuration is lik going to a retail store that is specialized in a certain range of products, e.g. Dell or Apple, and as for a computer. What you get is a list of possible configurations of a computer where you can select its components from a list of predefined components. This means going through a product configurator, selecting the base product and selecting some extra options, e.g. hard drive capacity.

This essentially is a product line approach where parts of the product line and its variants is even visible to the customer.

### Composition

Composition is like going to a computer retail store and buying and assembling the parts in an assisted way: for example, based on the items in the shopping cart, let the customer know:

- that the five PCIe cards will not fit the mainboard with only 4 slots
- that the power supply is not sufficient to power the system
- that the graphics card has an additional power socket which is not provided by the power supply

There are some online computer retailers that provide this kind of features. All this information is available in data sheets, but not all customers have the knowledge and experience to understand it. They need the support described above. Even experts are lost in case there is no data sheet.

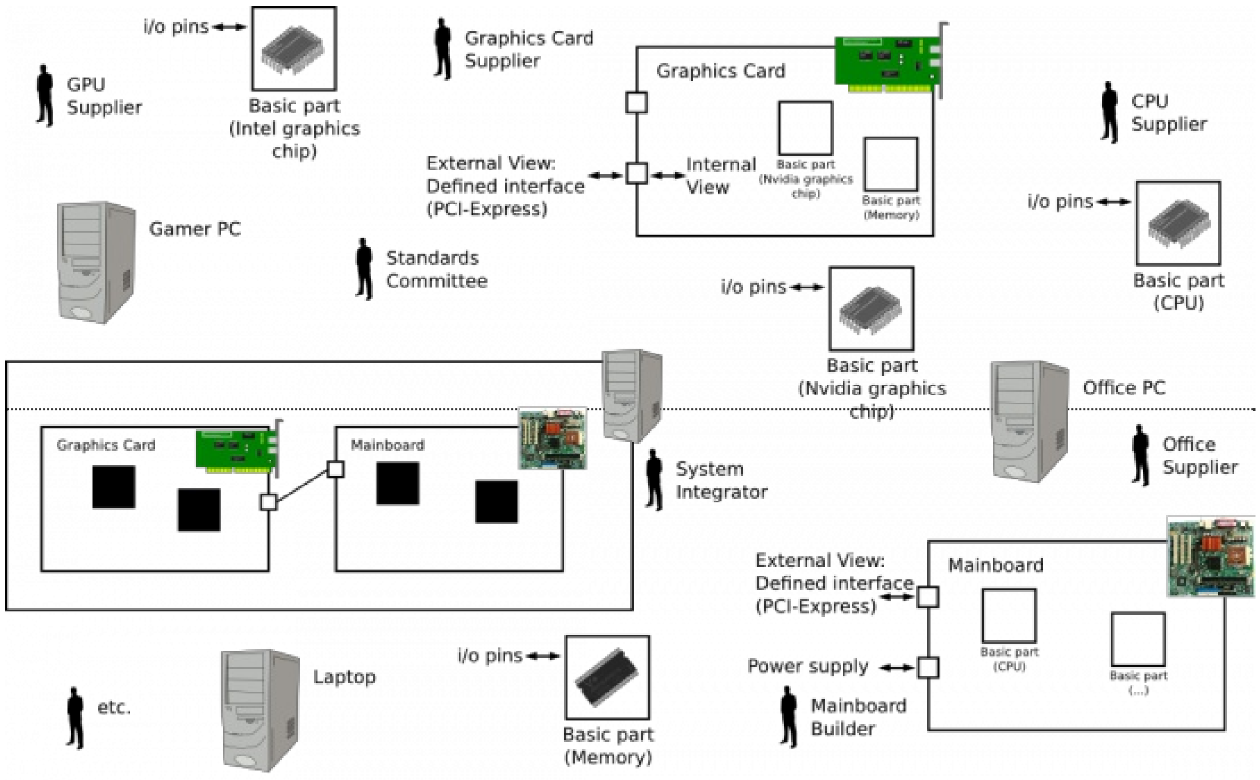
In robotics, there is neither a superordinate structure such as PCIe, no data-sheets for components, and no support for selecting components.

### Integration (in contrast to composition)

Integration is like assembling parts with non-standard interfaces that do not allow to separate and exchange parts afterwards, for example, a battery that is soldered inside a laptop. Even after ripping out the battery, it cannot be used as there is no knowledge about the battery, no data sheet: How much power? How about electrical polarity/pin assignments? One starts to reverse-engineer to discover the properties using a voltmeter and other tools.

## Ecosystem Example: Graphics Cards

In the PC industry, different ecosystem participants can supply and use building blocks to flexibly compose systems based on their needs. There are graphics card suppliers that do not know where their product is being used or for what purpose. They supply their graphics card and adhere to an specified interface (e.g. PCI express) to make sure it can be used with any mainboard. They can build their graphics card using off-the-shelf building blocks (e.g. Nvidia graphics chip and standard memory). They provide data sheets for the graphics card that specifies the properties of the product which are necessary to use it. The data sheet does not need to expose internal details or layouts (protected IP) of the graphics card.



Suppliers and Users collaborate and exchange building blocks in an ecosystem to flexibly compose systems based on their need.

## What Enables Composition in the PC Domain?

Enablers of composability in the PC domain are:

- Building blocks **adhere to superordinate structures** (e.g. PCIe)
- Building blocks **explicate properties in data sheets** (e.g. power supply, form factor, thermal information)

Thanks to this enablers, the following is possible in the PC domain and RobMoSys aims at the same for robotics:

### Views

Thanks to explicated properties in data sheets, specific views on a system can be taken. They are independent and each address concerns of the system. For example:

- A form factor view: will everything fit into the case? Are there enough slots in the casing for assembling the hard discs?
- A thermal view: how is heat flowing through the system and is the ventilation sufficient?
- A power supply view:

- General layout view: are there enough slots in the casing to access the PCI cards from the outside? Are there enough slots PCIe slots on the mainboard?

RobMoSys uses Views to group elements of the composition structure which are addressed by one role.

### Decoupling supply and use

Thanks to data sheets, one can plan a system and come up with a blueprint for later assembly since data sheets contain all necessary information. The physical devices do not need to be present at that stage and can be assembled by someone else based on the blueprint. The blueprint can be used to verify the system: for example the performance might not be sufficient for the intended application.

### IP is still flexible

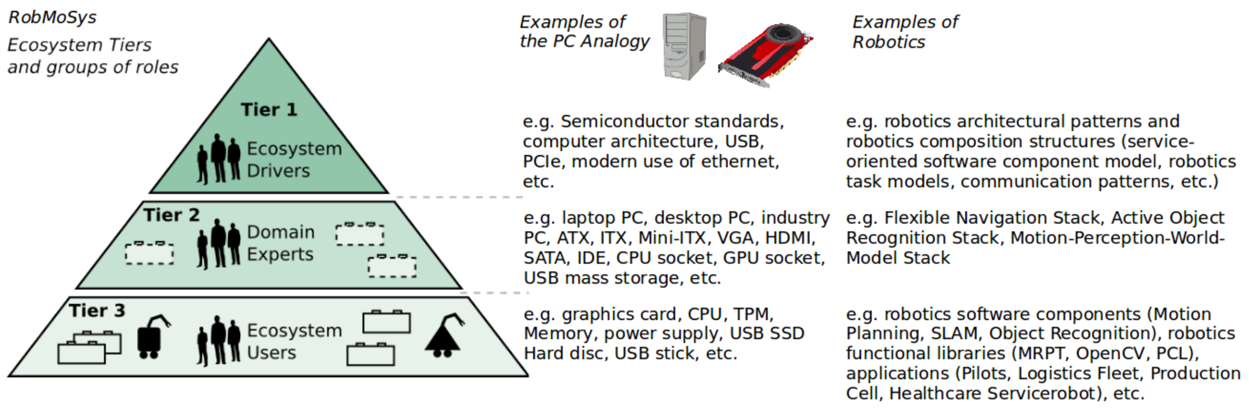
Exposing properties in a data sheet does not mean to expose intellectual property (IP). It is only about exposing the information that is relevant to use it (e.g. external view / interface), size of the device, power supply, etc. Information about the internals of the building block (circuit layout, chipset used, capacitors used, etc.)

### Flexible composition Combinations and alternatives

Adhering to superordinate structures means gaining access to all other building blocks that adhere to the same structure. This gives high flexibility in composing parts.

## RobMoSys Composition Tiers in the PC Domain

The below picture illustrated the Ecosystem Organization in composition Tiers using examples of the PC domain.



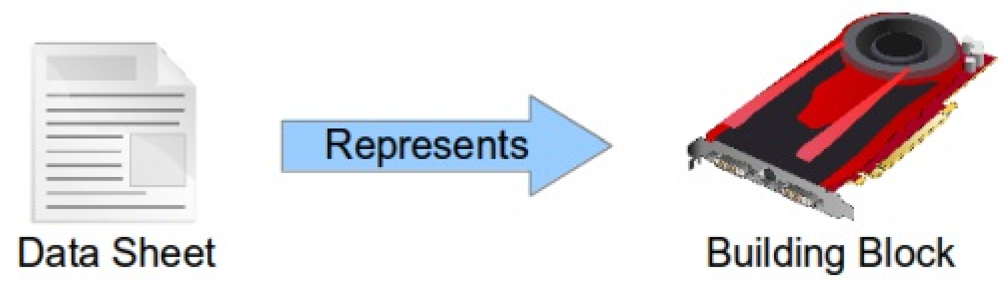
| Examples of the PC Analogy | Examples of Robotics |
|---|---|
| e.g. Semiconductor standards, computer architecture, USB, PCIe, modern use of ethernet, etc. | e.g. robotics architectural patterns and robotics composition structures (service-oriented software component model, robotics task models, communication patterns, etc.) |
| e.g. laptop PC, desktop PC, industry PC, ATX, ITX, Mini-ITX, VGA, HDMI, SATA, IDE, CPU socket, GPU socket, USB mass storage, etc. | e.g. Flexible Navigation Stack, Active Object Recognition Stack, Motion-Perception-World-Model Stack |
| e.g. graphics card, CPU, TPM, Memory, power supply, USB SSD Hard disc, USB stick, etc. | e.g. robotics software components (Motion Planning, SLAM, Object Recognition), robotics functional libraries (MRPT, OpenCV, PCL), applications (Pilots, Logistics Fleet, Production Cell, Healthcare Servicerobot), etc. |

The RobMoSys composition Tiers illustrated with examples of the PC domain.

General-purpose standards for the pc domain are located at Tier 1. USB for example can be used to connect almost any device. Every computer has a need for storage capacity. Within this domain, Universal Mass Storage (UMS, also known as “USB mass storage”) is based on USB and makes USB devices accessible as a hard disk to enable file transfer (Tier 2 in this analogy). Hardware vendors and users can offer or use any particular device with storage capacity that supports UMS on “Tier 3”. With the intention to connect a portable device for the sake of transfering files, any of these devices that supports UMS may be suitable: a particular USB stick, portable SSD Harddisk, Digital Camera, or mobile phone. Additional modeled descriptions must then support the system integrator in choosing the right building block: digital camera might be used to transfer documents, but the USB stick or SSD harddisk is probably the first choice depending on the file's size and other factors.

## Data Sheets and The Modeling Twin

Data sheets in the PC domain are comparable to the Modeling Twin in RobMoSys. Data sheets represent a physical building block. See What Enables Composition in the PC Domainto learn about the benefits.



general_principles:pc_analogy:start · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/general_principles:pc_analogy:start

# Ecosystem Organization

## Composition Tiers

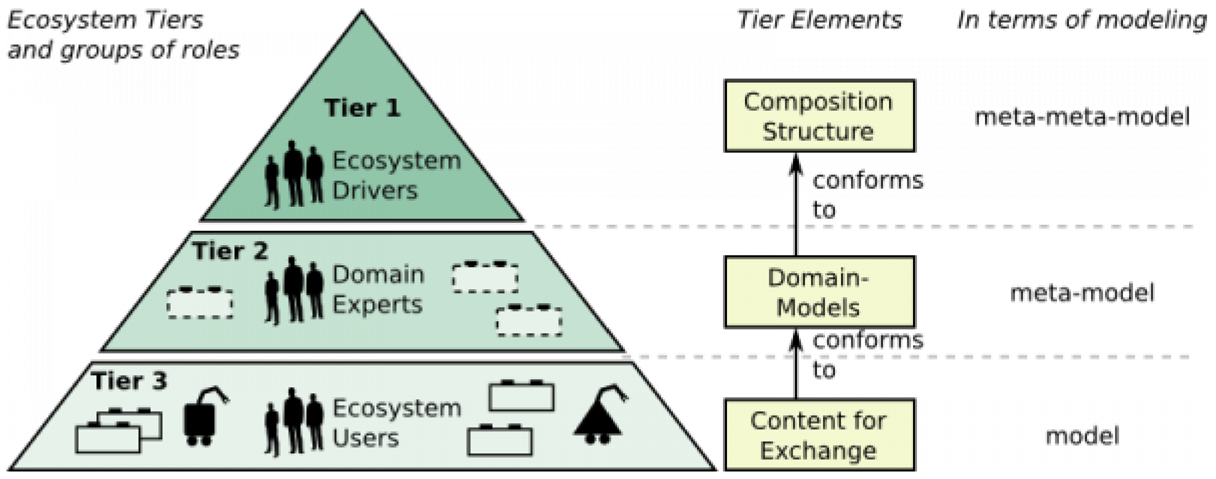
The general composition structure distinguishes three tiers.

RobMoSys envisions a robotics business ecosystem in which a large number of loosely interconnected participants depend on each other for their mutual effectiveness and individual success. The modeling foundation guidelines and the meta-meta-model structures are driven by the needs of the typical tiers of an ecosystem and the needs of their stakeholders (see figure 1). The different tiers are arranged along levels of abstractions. Figure 1 also illustrates the amount of experts or people contributing or using the particular tiers.

Tier 1 structures the ecosystem in general for robotics. It is shaped by the drivers of the ecosystem that define an overall composition structure which enables composition and which the lower tiers conform to (similar to, for example, the ecosystem of the Debian GNU/Linux OS and its structures). Tier 1 is shaped by few representative experts for ecosystems and composition. This is kick-started by the RobMoSys project. Structures defined on Tier 1 can be compared to structures that are defined for the PC industry. The personal computer market is based on stable interfaces that change only slowly but allow for parts changing rapidly since the way parts interact can last longer than the parts themselves and there is a huge amount of cooperating and competing players involved. This resulted in a tremendous offer of composable systems and components.

Tier 2 conforms to these foundations, structuring the particular domains within robotics and is shaped by the experts of these domains, for example, object recognition, manipulation, or SLAM. Tier 2 is shaped by representatives of the individual sub-domains in robotics.

Tier 3 conforms to the domain-structures of Tier 2 to supply and to use content. Here are the main “users” of the ecosystem, for example component suppliers and system builders. The number of users and contributors is significantly larger than on the above tiers as everyone contributing or using a building block is located at this tier.



### Tier 1: Composition-Structure – Meta-Structure

Tier 1 structures the ecosystem in general for robotics, independent of the sub-domains. It is shaped by the drivers of the ecosystem that define an overall structure which enables composition and which is to be filled by the lower tiers. Tier 1 defines general concepts and models for system composition such as the concept of service definitions, concept of components, and the composition-workflow that is tailored to service robotics. See Tier 1 Details for more information.

In terms of meta-modeling, elements of this tier correspond to/are meta-meta-models

#### Elements on this tier

RobMoSys Composition Structures, e.g.

- concept of service definitions
- concept of components, i.e. the Component Metamodel
- a set of communication semantics to choose from

#### Examples of roles on this tier

Content on this tier is defined by the ecosystem drivers, i.e. the RobMoSys community with moderation of the RobMoSys consortium.

#### See also

- Tier 1 Details

### Tier 2: Robotics-Domain-Specific Structures – Robotics Domain Models

Tier 2 structures the particular domains within service robotics. It is shaped by the experts of these domains, for example experts from object recognition, from manipulation, or from SLAM. This is a community effort which structures each robotics domain by creating domain-models. Experts working at this level define concrete service definition models, for example a service definition for robot localization.

Domain-models, for example, are “Service Definitions” that cover data structure, communication semantics and additional properties for specific services such as “robot localization”. To find such a service definition, domain experts of each particular domain discuss how to represent the location/position of a robot and what additional attributes are required and how they are represented (e.g. how the accuracy is represented).

In terms of meta-modeling, elements of this tier correspond to/are meta-models

#### Examples of elements on this tier

- service definitions for localization
- definition of how a robot pose with uncertainty is represented

#### Examples of roles on this tier

- These are experts in the particular domain (SLAM, object recognition, manipulation), for example the manipulation domain to come up with domain-models for a composable motion stack based on the RobMoSys composition structures on Tier 1.
- Service Designer role

### Tier 3: Ecosystem Content

Tier 3 uses the domain-structures from Tier 2 to fill them with content: to supply or to use content. It is shaped by the users of the ecosystem, for example component suppliers and system builders. They use the domain-models to create models as actual “content” of the ecosystem to be supplied and used. On this tier, for example, concrete Gmapping component for SLAM that provides a localization service is supplied to a system builder to compose a delivery robot.

In terms of meta-modeling, elements of this tier correspond to/are models (of components/systems)

**Examples of elements on this tier**

- Components for AMCL localization, Gmapping, etc. providing a localization service
- Task plot: how to make coffee
- Composed applications: A restaurant butler robot
- Component model based on the Component Metamodel

**Examples of roles on this tier**

- Component Supplier
- System Architect
- System Builder

## RobMoSys Modeling Support

- See the various meta-models of the RobMoSys composition structures.

## RobMoSys Tooling Support

- See how the SmartMDSD Toolchain supports the RobMoSys Ecosystem Organization in three composition tiers
- See how Papyrus4Robotics supports the three composition tiers

## See also

- Analogy: The PC Domain
- Roles in the Ecosystem
- Tier 1 Details
- Composition in an Ecosystem

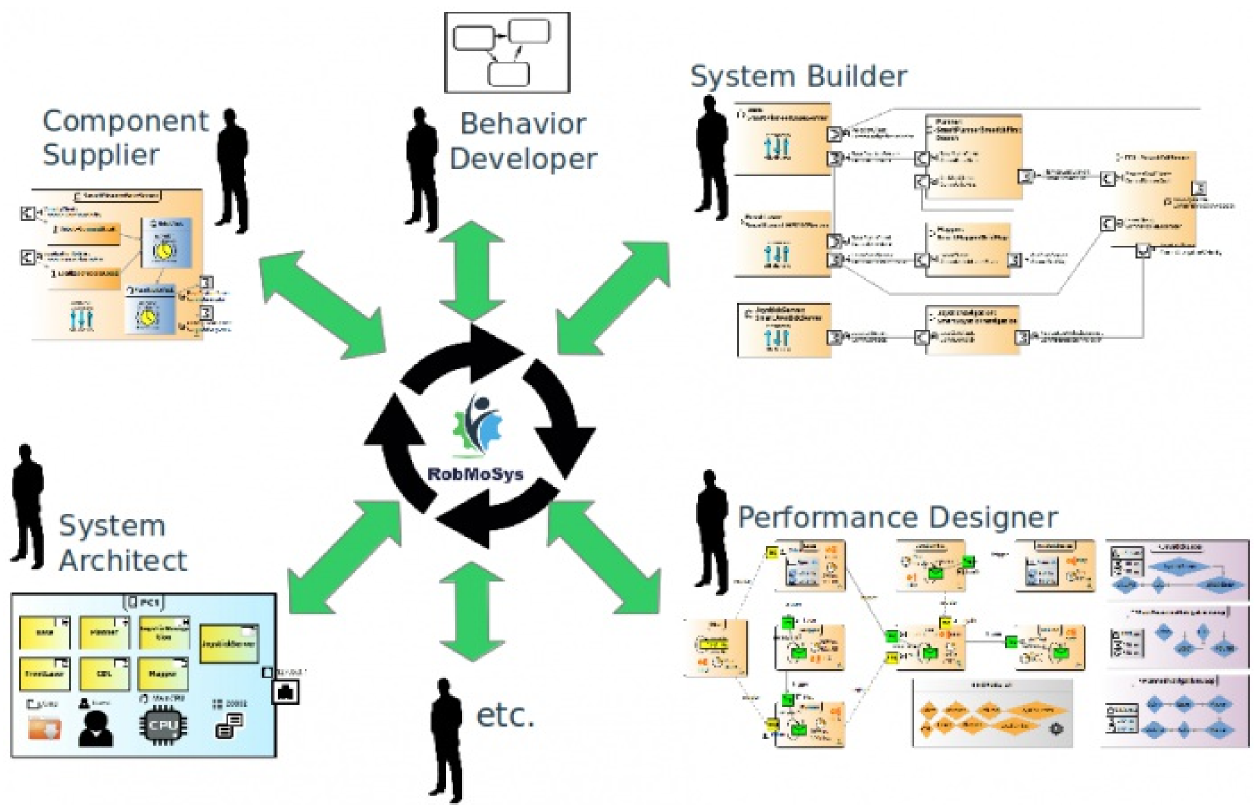
## Acknowledgement

This document contains material from:

- Stampfer2018 Dennis Stampfer, "Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics". Dissertation, Technische Universität München, München, Germany, 2018. [http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2]

# Roles in the Ecosystem

The participants in the ecosystem (see Ecosystem Organization) take one or several “roles” to use and supply building blocks. The RobMoSys composition structures define which parts are variable and which parts are fixed, i.e. guided by the structures to ensure composability. Each role uses dedicated views to work on models and Modeling Twin



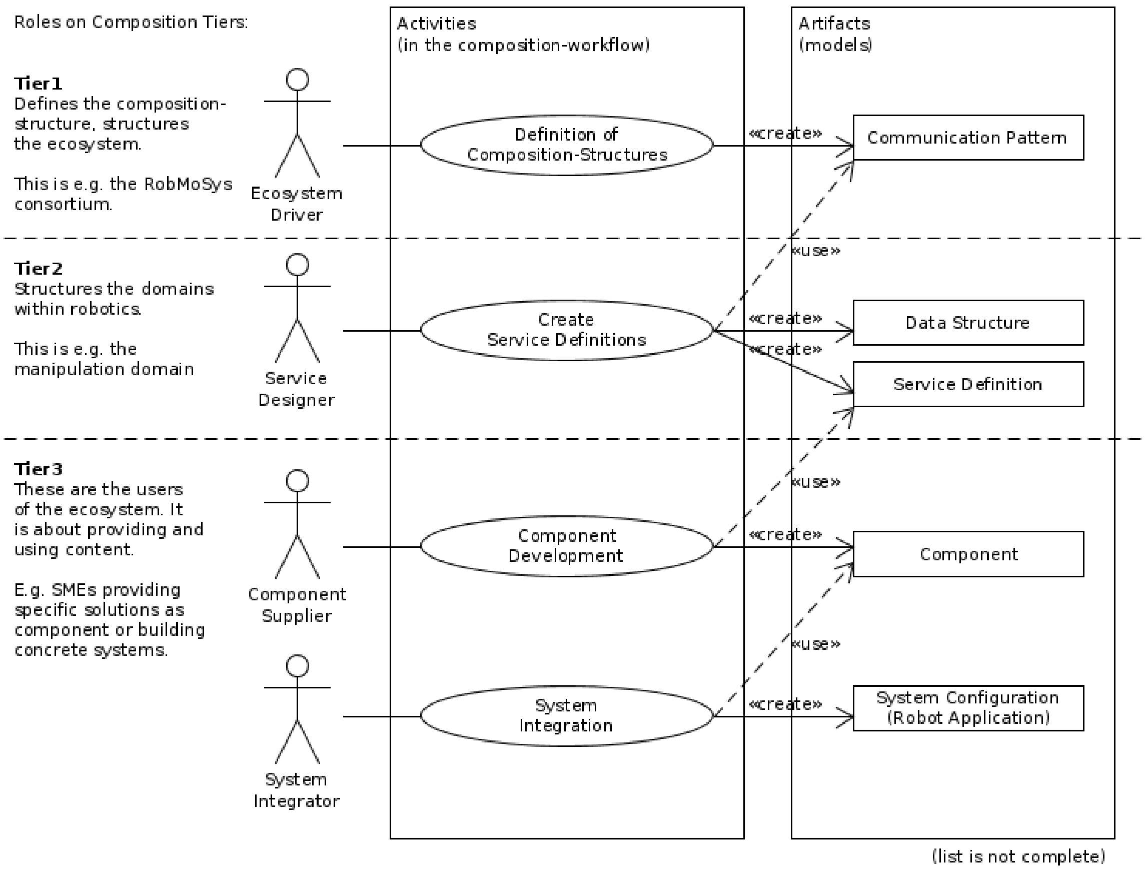
## List of Roles

(alphabetical order)

- Behavior Developer
- Component Supplier
- Function Developer
- Performance Designer
- Safety Engineer
- Service Designer
- System Architect
- System Builder

## Roles in Context of Composition Tiers

The figure below illustrates the roles and their corresponding activities that use or create models on each composition tier.


Roles on Composition Tiers:
Tier1
Defines the composition-structure, structures the ecosystem.
This is e.g. the RobMoSys consortium.
Ecosystem Driver
Tier2
Structures the domains within robotics.
This is e.g. the manipulation domain
Service Designer
Tier3
These are the users of the ecosystem. It is about providing and using content.
E.g. SMEs providing specific solutions as component or building concrete systems.
Component Supplier
System Integrator
Activities (in the composition-workflow)
Definition of Composition-Structures
Create Service Definitions
Component Development
System Integration
Artifacts (models)
Communication Pattern
Data Structure
Service Definition
Component
System Configuration (Robot Application)
«create»
«use»
(list is not complete)

## See also

- Ecosystem Organization to learn about Ecosystem and its Composition Tiers
- RobMoSys Views to learn about the concept of views that roles use
- Modeling Twin

# System Builder

This role on Tier 3 puts together systems from building blocks (i.e. software components). Based on a system architecture from a system architect, the system builder selects components (provided bycomponent suppliers) from the ecosystem that realize the needed services. Matchmaking must be made on the basis of offered services and on other properties, e.g. the required accuracy. Another concern of system builders is to package everything together such as e.g. also the robotic behavior models from behavior developers and making the system ready for deployment.

Synonym:

- Within the literature, this role is sometimes called “system integrator” which is considered inappropriate within the RobMoSys context, because of its close relation to “system integration” which contrasts to system composition (see glossary).

Related views and models:

- System Component Architecture Metamodel

See also:

- System Architect
- Component Supplier
- User Stories including this role
- Roles in the Ecosystem

# Function Developer

Provides content on function-level to be used by component suppliers.

Synonym:

- none

Related views and models:

- Fix Me! to be defined

See also:

- Component Supplier
- User Stories including this role
- Roles in the Ecosystem

# Service Designer

These are the domain experts on Tier 2 that design individual service definitions for use byTier 3 roles component supplier and system architect. This enables the definition of “de-facto” standard service definitions within a specific robotics sub-domain such as “object recognition”, “mobile manipulation”, “SLAM”, etc. For example, they can define what is a common (good) representation for a “localization” service that should be used (and shared) within the “SLAM” domain.

Synonym:

- none

Related views and models:

- Service Design View
- Service-Definition Metamodel

See also:

- Component Supplier
- System Architect
- User Stories including this role
- Roles in the Ecosystem

## Acknowledgement

This document contains material from:

- Stampfer2018 Dennis Stampfer, "Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics". Dissertation, Technische Universität München, München, Germany, 2018. [http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2]

# Performance Designer

A performance designer is a role at Tier 3 and is responsible to configure performance-related system properties. Therefore, predefined Activities within components are configured exploiting their left-open variability such that several Activities form trigger chains and thus realize application-specific end-to-end timings. Based on a performance model, a Compositional Performance Analysis (CPA) can be automatically triggered to simulate and validate the envisioned run-time performance of a system. Moreover, a performance model can be used by the System Builder role to refine the instantiated components of a given system. Further details can be found in:

- Alex Lotz, Arne Hamann, Ralph Lange, Christian Heinzemann, Jan Staschulat, Vincent Kesel, Dennis Stampfer, Matthias Lutz, and Christian Schlegel. “Combining Robotics Component-Based Model-Driven Development with a Model-Based Performance Analysis.” In: IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR). San Francisco, CA, USA, Dec. 2016, pp. 170–176. LINK [http://dx.doi.org/10.1109/SIMPAR.2016.7862392]
- Alex Lotz, "Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System", Dissertation, Technische Universität München, München, Germany, 2018. [https://mediatum.ub.tum.de/?id=1362587]

Synonym:

- none

Related views and models:

- Performance Metamodel

See also:

- User Stories including this role
- Roles in the Ecosystem

# Component Supplier

A component supplier is a role on Tier 3 that offer software components as units of composition that provide or require services (service-level) and contain functions. He/she models the component by using existing service definitions and functions. He/she therefore uses models from the roles service designer and function developer.

One of the tasks of the component supplier is also to implement a skill that lifts the abstraction of a component from the service level to the task level (see Separation of Levels and Separation of Concerns). These skills are then used be the behavior developer to orchestrate components.

Synonym:

- component developer

Related views and models:

- Component Development View
- Component-Definition Metamodel

See also:

- Service Designer
- Function Developer
- User Stories including this role
- Roles in the Ecosystem

general_principles:ecosystem:roles:component_supplier · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/general_principles:ecosystem:roles:component_supplier

# Behavior Developer

The role of the Behavior Developer is responsible for developing tasks or task-plots (composition of tasks) modeling how a robotics system, consisting of software components, is orchestrated at run-time to provide a service as a whole system. The role models robot behavior through the tasks at the according abstraction level of tasks.

The tasks that the behavior developer models make use of the functionalities provided by the components. Functionalities that are implemented within software components become accessible through skills (skill behavior model). Skills lift the abstraction level of components to use them on a task level (see Separation of Levels and Separation of Concerns). Thereby the tasks itself are independent of any component and can be reused with a robotics system consisting of different software components. To connect tasks to components the role uses the skill definitions (Domain Experts, Tier 2), as interface to the skills.

Skills are defined at Tier 2 and are implemented in Tier 3 by the component supplier role.

The resulting tasks are used by the System Builder to compose a run-able system including the behavior models. Thereby the component independent tasks are linked with skills provided by the selected components, according to the skill definitions used by the tasks.

The role of the Behavior Developer is driven by the needs of an application or a service a robotic system has to provide. It realizes variability at a task level, thereby using and fixing some of the variability provided either by skills or by other reused tasks. The role may also introduce additional variability at the task level and specify rules and policies how this variability will be bound at run-time, using the then available information (context).

Synonym:

- none

Related views and models:

- Robotic Behavior Metamodel

See also:

- User Stories including this role
- Roles in the Ecosystem
- Architectural Pattern for Task-Plot Coordination (Robotic Behaviors)
- Task-Level Composition for Robotic Behavior
- Separation of Levels and Separation of Concerns
- Component Supplier

# Safety Engineer

The Safety Engineer is responsible to define safety-related system aspects and closely interacts with system builders.

Synonym:

- none

Related views and models:

- … link view (to be defined)
- … link model (to be defined)

See also:

- User Stories including this role
- Roles in the Ecosystem

# System Architect

This role on Tier 3 designs a system architecture based on existing service definitions fromservice designers. The resulting system architecture is independent of specific components and can be used by system builders to select according components for realizing this system architecture. In other words, a system architect provides a kind of “system blueprint” for system builders who can realize this system by selecting appropriate components. For example, a system architect might design a robot navigation stack based on mapping, localization, and motion-execution services.

Synonym:

- none

Related views and models:

- System Service Architecture Metamodel

See also:

- Service Designer
- System Builder
- User Stories including this role
- Roles in the Ecosystem

# User Stories

The following user-stories provide more detailed examples of the primary user-stories [http://robmosys.eu/user-stories/] and the user-stories presented at theERF 2017 [http://robmosys.eu/download/sara-tucci-cea-christian-schlegel-hs-ulm-presentation-of-the-robmosys-project/]. The user-stories are supposed to guide RobMoSys consortium to provide the structures and the open call third party partners to apply.

User-stories are described in the *As user, I want*-style:

- As a (role), I want (goal, objective, wish), so that (benefit)
- As a (role), I can (perform some action), so that (some goal is achieved)

Some user-stories are described in context of a specific ecosystem participant or role. Some are not described in a specific context and can apply to multiple roles. For example what is of interest to an integrator can be of interest to a supplier since the integrator might also supply a system (see system-of-system).

See also:

- Roles in the Ecosystem

## Composable commodities for robot navigation with traceable and assured properties

A very generic but extremely important user story illustrating the full scope of RobMoSys by a single example: Based on model-driven tools, develop and provide composable navigation components with all their explicated properties, variation points, resource requirements etc. (the modeling twin / data sheet). Become able to compose your navigation system out of these readily available commodity building blocks according to your needs and be sure that your needs are being matched, that the properties become traceable etc.

- I, as system builder, just want to become able to compose robotics navigation out of commodity building blocks according to my needs with predictable properties, assured matching with my requirements, free from interference. It is just astonishing that this is not yet possible in robotics. (with MoveBase being exactly an example of 1how it should not be)

## Description of building blocks via model-based data sheets

RobMoSys achieves a specific level of quality and traceability in building blocks, their composition and the applications.

as a **component supplier**

- I want my component to become part of as many systems as possible to ensure return-of-investment for development costs and to make profit.
- I need to offer my software component (building block) such that others can easily decide whether it fits their needs and how they can use it.
- I want to offer my software component with a data sheet in form of a digital model (see modeling-twin). A data sheet contains everything you need to know to become able to use that software component in a proper way (interface between the component and its environment) while protecting intellectual

property. It contains information about the internals of the software component only as long as this is needed for a proper use.

as a **system builder**

- I want to select from available components the one which best fits my requirements and expectations (provided quality, required resources, offered configurability, price and licensing, etc.)
- I want to check via the data sheet (in form of a digital model) whether that building block with all its strings attached fits into my system given the constraints of my system and given the variation points of the building block. Thereto, I want to be able to import it into my system design to perform e.g. a what-if analysis etc.
- I want to extract from my system design the specification of a missing building block such that someone else can apply for providing a tailored software component according to my needs
- I want to use components as grey-box, use them “as-is” and only adjust them within the variation points expressed in the data-sheet without the need to examine or modify source code.

## Replacement of component(s)

A hardware device is broken and the identical device is not available anymore (deprecated, discontinued, only next version available). As a system builder,

- I want to check whether all my relevant system level properties and constraints are matched when I use the new device.
- I also want to know how I need to configure it for that.

The very same holds true for software components where a software library used is not available anymore with updates of other libraries etc.:

- As a system builder, when I remove a software component from a system, I want to know which constraints define the now white spot in my design in order to fill in another one with the proper configuration to again match the system level properties.

Example:

- From laser-based localization to visual localization
- Replacing a 6 DOF manipulator with a 5 DOF manipulator

## Composition of components

I want to be able to predict selected properties of the composition of various software components given their individual properties, their configurations, their composition. For example, I want to know about the required resources, whether there are bottlenecks somewhere, whether there are no unnecessarily high update rates without consumers requiring them etc.

I want to know about the consistency of the overall settings in order to increase the trust into the system. I want to know that critical paths are transformed from design-time into run-time monitors and sanity checks, e.g.

## Quality of Service

I would like to know whether the amount of resources and the achieved performance (in general, quality of task achievement) is adequate. I want to know what kind of impact a decrease in resource assignment has on the performance of the functionalities of the robot.

I want to make sure that properties are traceable through the system and are managed through the development and composition steps. For example

- qualities at service ports of components are linked with component configurations which are linked with configurations of the execution container and the underlying OS and middleware
- at deployment time (system builder), reservation based resource management should be tool supported

## Determinism, e.g. for robot navigation

As system builder, I want my system (e.g. navigation system on a mobile robot) to work exactly the same way again when I change the platform (e.g. change the mobile base or the laser ranger or the computing platform in a mobile robot).

- I want to know that the intended functional dependencies and intended processing chains are finally realized within my system composition
- I want to know that relevant functional dependencies are still valid even after replacing one of my onboard computers by a different one

## Free from hidden interference

- When extending a system, I want to know that I do not interfere with the already setup components, already used resource shares etc.
- I want to be sure that deploying further components onto my system is free from hidden interference or hidden side-effects.

## Management of Non-Functional Properties

As system builder,

- I want to be able to adhere to functional and, in particular, to non-functional properties when composing software components.
- I want to re-use software components as black (gray) boxes with explicated variation points such that application-specific system-level attributes can be matched without going into the internals of the building blocks.
- I want to be able to work on explicated system level properties: allow to design system properties such as end-to-end latencies and explicit data-propagation semantics during system composition without breaking component encapsulation.
- I want to be able to match / check / validate / guarantee required properties via proper configurations of variation points, via sound deployments etc.

Separation of roles (in particular, between component providers (driven by technology) and system builders (driven by the application domain) is considered a basic prerequisite towards the next level of market maturity for software in robotics, and thus towards a software business ecosystem. Support for the system builder is needed in order to know about the properties of resulting systems instead of wondering whether they match the requirements or whether they are resource-adequate etc.

## Gap between design-time assumptions and run-time situation

When a system is deployed, design-time assumptions might not hold. For many systems it is difficult to know when the system fails during operation.

- As a system builder, I want to generate sanity checks, monitors and watchdogs from my design-time models to be able to detect unwanted behavior and to detect operation outside of specified ranges.

## System analysis tools

There are analysis tools in related domains not yet accessible to robotics as they are complex to use. I would like to have support from these tools during the design of components, their selection and composition etc. I want to better address what-if questions, to perform trade-off analysis etc. These tools should be attached to robotics via dedicated model transformations without requiring me to get into them.

## Task modeling for task-oriented robot programming

- Reusable and composable task blocks which express knowledge about how to execute tasks (action plot) and what are good ways to execute tasks (qualities).
- Management of the constraints such that composition for parallel and nested execution is free of conflicts and that open variation points can be bound at run-time according to the given situation ways to link generic task descriptions (with all their constraints and resource requirements) with software components (with all their configurations etc.)

## Safety

- As safety engineer, I want to model limits for critical properties like the maximum speed when carrying around a hot coffee, when maneuvering in a crowded environment, the maximum speed dependent on visibility ranges etc.
- As safety engineer, I model constraints for particular applications and environments.
- As system builder, I want to be able to import these constraints such that tools help me to ensure design-time consistency and run-time conformance with them (via generated hard-coded limits, via monitors, via sanity checks etc.)

It is important to highlight what we are trying to say about system safety (not necessarily to prove), because systems are safe in a particular context under a particular set of assumptions (e.g. by run-time monitors etc.). The focus is possibly shifted from fail-safe to safe-operational, which may include some liveness in it. It is about efficient falsification (the following things cannot happen) rather than costly verification (it always behaves only like that).

general_principles:user_stories · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/general_principles:user_stories

# Tier 2: Examples of Domain Models

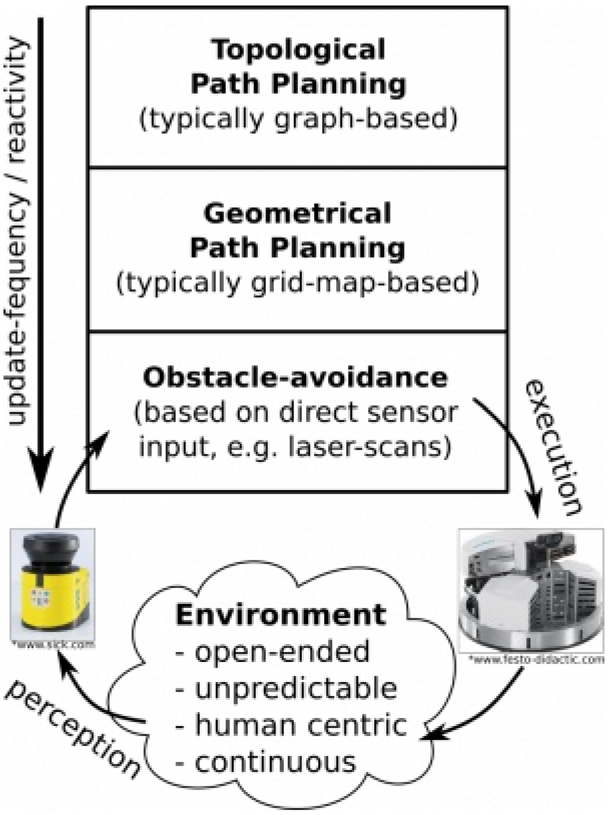
RobMoSys allows the definition of domain-specific models and structures at composition Tier 2. To illustrate this concept, RobMoSys defines the following extendable content for Tier 2.

- Motion, Perception, Worldmodel Stack
- Flexible Navigation Stack
- Active Object Recognition
- etc.

# Flexible Navigation Stack

The flexible navigation stack is a set of components that realize specific navigation services to provide a flexibly applicable navigation capability for a service robot. The services defined (service definitions) for the navigation stack are a typical example of the Composition Tier 2 contents of the RobMoSys Ecosystem. This navigation stack can be used with various robot platforms and different kinds of sensors. Moreover, it is able to deal with unstructured and dynamic environments of variable scale. The focus hereinafter is to emphasize the general design choices and architectural decisions of the navigation stack components. After that, the following section provides some technical details and references for the concrete open-source components that can be used already now, e.g. with the Robotino3 platform.

The figure on the right illustrates the three main levels of the navigation stack. These levels describe the shared responsibilities between different parts of the navigation stack. These responsibilities are assigned top down according to the subsidiarity principle (as explained next).


update-fequency / reactivity
Topological Path Planning (typically graph-based)
Geometrical Path Planning (typically grid-map-based)
Obstacle-avoidance (based on direct sensor input, e.g. laser-scans)
execution
*www.sick.com
*www.festo-didactic.com
Environment
- open-ended
- unpredictable
- human centric
- continuous
perception

## Obstacle Avoidance Level

The bottom level defines components (a full list is provided further below) related to the fast and reactive obstacle-avoidance navigation loop. This loop ensures that regardless of where the robot has to move next, this movement will not cause any collisions and the robot will not be commanded to execute a physically invalid movement considering the robot's kinematic and dynamic constraints. Therefore this loop will only command navigation values that never lead to a collision even if these commands might not directly lead toward the next goal (e.g. because of the need to avoid a suddenly appeared obstacle in between). Consequently, this loop might lead to a globally non-optimal, yet collision-free, navigation.

## Geometrical Path Planning Level

At the middle level, a geometric path planner calculates intermediate way-points based on a grid-map of the current environment. The planner relies on this map, which is updated during the navigation to accommodate for changes in the environment. A localization component estimates the current position of the robot within that maps. Several existing path-planning algorithms (using A* for example) allow the generation of intermediate way-points to be individually approached by the lower obstacle-avoidance level. In contrast to the lower obstacle-avoidance level, this intermediate geometric path planing level has a global view on the mapped environment. This is useful to e.g. avoid local minima (by generating intermediate way-points around them). It is worth mentioning that this intermediate level typically does not generate full trajectories (to be exactly executed by the lower level), but sparse intermediate way-points. These way-points are within a direct line of sight, which allows approaching them individually by the lower level without requiring a map. Overall, this enables a clear separation of concerns between the two lower levels and avoids several disadvantages with respect to wasting resources (due to e.g. too frequent need for path re-planning) continuous velocity changes and too tight (i.e., inflexible and hardly exchangeable) coupling with the lower level.

## Topological Path Planning Level

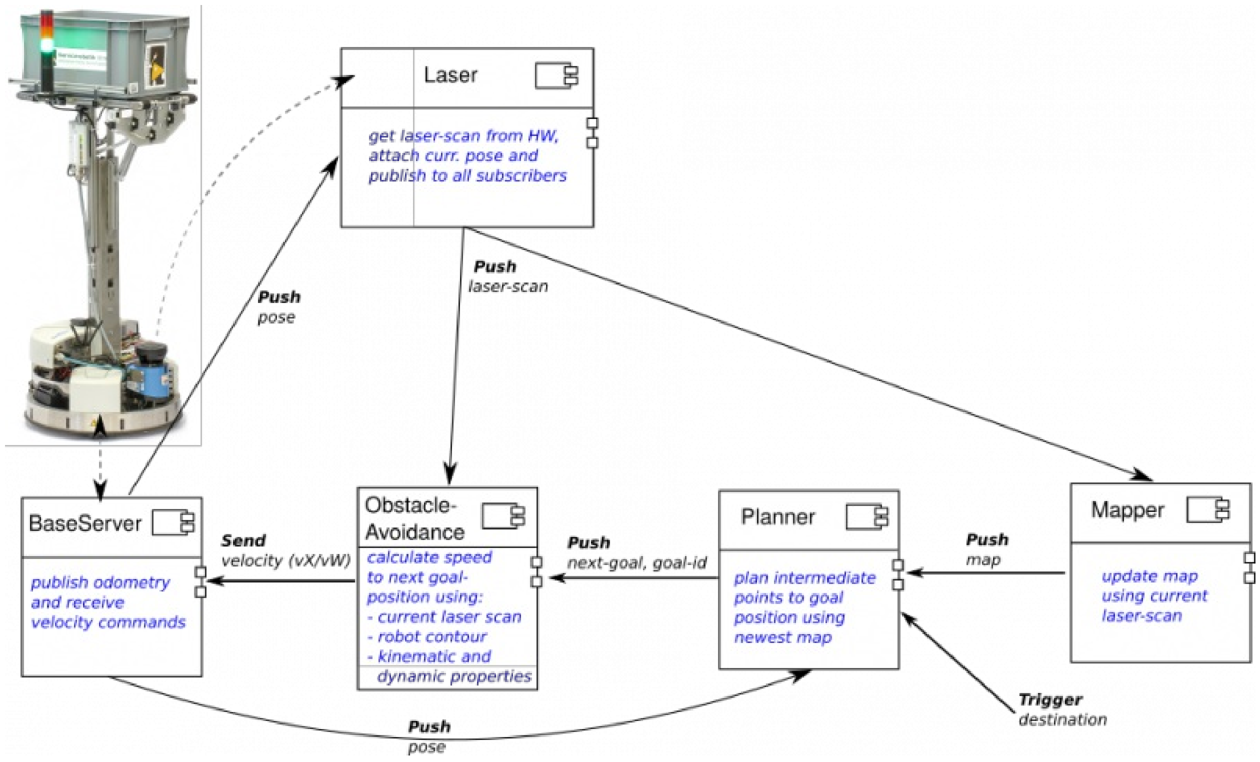
In some cases, even the intermediate level is not sufficient. For instance, if a robot needs to navigate in an entire building consisting of several floors, maybe connected over elevators, then building a single huge grid map becomes complicated, too inefficient and too resource consuming. In these cases, it is rather reasonable to calculate several smaller grid-maps (e.g. one for each level or room in the building) and to concatenate these grid-maps in a topological map (which is typically a graph). The responsibility of this top level is to provide a logical plan how to navigate through the separated maps, e.g. through levels or rooms of a building.

## Flexibility in the Navigation Stack

The separation of the navigation components into these three levels has several advantages. The levels can be composed to individual navigation solutions best fitting the needs of the application or the current environment a robot is navigating in. According to these needs the size of the stack can be changed, with the bottom level being the most versatile and configurable one. For instance, some scenarios might require to manually command a robot using a joystick. In that case, both upper levels would be replaced by a simple joystick driver component, while the collision avoidance level still validates the navigation commands. In other scenarios, a robot might always navigate in a single map only. For that the geometrical path-planner on the middle level (without the topological path planner on top) is fully sufficient. Of course, there are also scenarios where all three levels are needed. Even in these latter cases, components on the individual levels can be flexibly exchanged (even at run-time, while moving) by alternatives because of a clear separation of responsibilities on each level and due to the clear interfaces between the levels. For example, it is possible to exchange free navigation with corridor-based navigation[1] to make the robot move only within predefined tracks.

## The navigation stack components and services

The figure below illustrates the interaction of the navigation components over generic navigation services. While the navigation services are always stable, there are several alternatives for each of the fife navigation components (see below) that realize the same services but internally implement different algorithms. This decoupling between a component's internal implementation and the component's service-based interaction is a fundamental principle in RobMoSys that enables a flexible reuse (i.e., exchange) of components by alternatives with unique selling points and thus makes the navigation stack flexibly usable in different applications with different requirements with respect to envisioned environments and the used robot platforms.


Laser
get laser-scan from HW, attach curr. pose and publish to all subscribers
Push pose
Push laser-scan
BaseServer
publish odometry and receive velocity commands
Send velocity (vX/vW)
Obstacle-Avoidance
calculate speed to next goal-position using:
- current laser scan
- robot contour
- kinematic and dynamic properties
Push next-goal, goal-id
Planner
plan intermediate points to goal position using newest map
Push map
Mapper
update map using current laser-scan
Trigger destination
Push pose

The navigation stack consists of two hardware-related components, namely **Laser** and **BaseServer**. These two components abstract away the used hardware. While the components themselves are specific to a particular platform (e.g. Robotino3, PAL Tiago, etc.), they implement the following services that are platform-independent:

- **BaseServer**
  - The BaseServer acts as a hybrid component, it is both a **scanner** component in the sense that it provides updated odometry values, and as a **actuator** component in the sense that it receives navigation commands to be executed by the base platform.
  - *provides **BaseStateService***: **PushPattern<DataType=CommBasicObjects.CommBaseState>**: This service continuously provides the current geometric position (i.e., odomentry) of the base platform.
  - *provides **NavigationVelocityService***: **SendPattern <DataType=CommBasicObjects.CommNavigationVelocity>**: This service receives navigation-velocity command values which are executed by the base platform. The base platform executes the latest available navigation command until a new value arrives and overrides the previous value.
  - *provides **LocalizationUpdateService***: **SendPattern < DataType = CommBasicObjects.CommBasePositionUpdate >**: This is an optional service that allows correcting the robot's pose (i.e., its odometry) from a localization component (see below).
- **Laser**
  - The Laser component receives odometry updates and publishes new laser-scans together with the latest available odometry value. This component is one classical type of a **scanner** component.
  - *requires **BaseStateService*** (see explanation above)
  - *provides **LaserService***: **PushPattern <DataType=CommBasicObjects.CommMobileLaserScan>**: This service continuously provides the current laser-scan including the *CommBaseState* (as the geometric frame) from the time when the laser-scan has been recorded.

The other three navigation components implement the different capabilities of the navigation stack, namely (1) obstacle avoidance, (2) mapping, and (3) path-planning. Again, similar to the two hardware-related components above, the three components internally implement a specific algorithm and are exchangeable due to the following algorithm-independent service definitions that they individually implement:

- **Mapper**
  - This component receives a current laser-scan and accumulates the information from this scan into a locally maintained grid-map.
  - *requires* ***LaserService*** (see explanation above)
  - *provides* ***CurrGridMapPushService***: **PushPattern <DataType=CommNavigationObjects.CommGridMap>**: This is an updated grid-map.
- **Planner**
  - This component takes a current grid-map and the current destination location[2] as input and calculates a path (consisting of intermediate way-points) to reach that destination.
  - *requires* ***CurrGridMapPushService*** (see explanation above)
  - *provides* ***PlannerGoalService***: **PushPattern < DataType = CommNavigationObjects.CommPlannerGoal >**: This is the next intermediate way-point for the platform to approach.
- **ObstacleAvoidance**
  - This component implements an obstacle-avoidance algorithm, such as e.g. the Curvature Distance Lookup (CDL) [http://ieeexplore.ieee.org/document/724683/][3] approach. This components takes two inputs, namely the current laser-scan and the next way-point to approach and calculates a navigation command that approaches the next way-point on the as direct curvature as possible avoiding any collisions.
  - *requires* ***LaserService*** (see explanation above)
  - *requires* ***PlannerGoalService*** (see explanation above)
  - *requires* ***NavigationVelocityService***: provides navigation-velocity commands to be executed by the base platform, thus closing the loop back to the BaseServer (see explanation above).
- *optional* **Localization**
  - This component implements a localization algorithm (such as e.g. AMCL [https://www.ri.cmu.edu/publications/monte-carlo-localization-for-mobile-robots/]) based on the current laser-scan to calculate a current actual position of the robot within the environment. This position is communicated through the LocalizationUpdateService (see below) to correct the robot's odomentry (i.e., to improve the accuracy).
  - *requires* ***LaserService*** (see explanation above)
  - *requires* ***LocalizationUpdateService***: This service provides a pose update for the robot's odometry.

Overall, the three navigation components **BaseServer**, **Laser** and **ObstacleAviodance** together realize the lowest **obstacle avoidance level** (see above). The **Mapper**, the **Planner** and optionally the **Localization** components realize the middle **geometric path planning** level. Finally, the upper **topological path planning** level is realized by a symbolic planner component.

- **SymbolicPlanner**
  - This is a generic component that is able to find solutions for a given problem domain. Internally, this component might implement a symbolic planner algorithm like metric-ff or lama.
  - *provides* ***SymbolicPlan***: **QueryPattern<Request=CommSymbolicPlannerRequest, Answer=CommSymbolicPlannerPlan>**: This query service allows querying for a solution for a given problem domain. The problem domain is transferred within the Request object and the solution is replied within the Answer object.

The symbolic planner component is not only used for geometric path planning but is a generic component that is used for all kinds of combinatoric problems. This component typically directly interacts with the Task Coordination Level.

## RobMoSys Modeling Support

The following composition structures are directly related to the realization of the navigation stack:

- ComponentDefinition Metamodel
- Service-Definition Metamodel
- Communication-Pattern Metamodel
- System Component Architecture Metamodel

## RobMoSys Tooling Support

- The following page discusses the concrete models of this example using the SmartMDSD Toolchain: Support for the Flexible Navigation Stack

1)

Matthias Lutz, Christian Verbeek and Christian Schlegel. “Towards a Robot Fleet for Intra-Logistic Tasks: Combining Free Robot Navigation with Multi-Robot Coordination at Bottlenecks”. In Proc. of the 21th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Berlin, September 6-9, 2016. Electronic ISBN: 978-1-5090-1314-2, DOI: 10.1109/ETFA.2016.7733602. Link [https://doi.org/10.1109/ETFA.2016.7733602]

2)

The next destination is commanded from the behavior-coordination component (seeRobotic Behavior Metamodel for further details).

3)

Christian Schlegel. “Fast local obstacle avoidance under kinematic and dynamic constraints for a mobile robot”. In *IEEE International Conference on Intelligent Robots and Systems (IROS)* Victoria, Canada, 1998. DOI: 10.1109/IROS.1998.724683 [https://doi.org/10.1109/IROS.1998.724683].

domain_models:navigation-stack:start · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/domain_models:navigation-stack:start

# Pilots: Demonstrating the RobMoSys Approach

RobMoSys uses pilots to demonstrate the use of its approach through the development of full applications with robots. Pilots span different domains and different kind of applications. The pilots can be provided to project contributors to support designing, developing, testing, benchmarking and demonstrating their contribution.

- Goods Transport in a Company:
    - Intralogistics Industry 4.0 Robot Fleet Pilot
- Mobile Manipulation for manufacturing applications on a product line:
    - Flexible Assembly Cell Pilot
    - Human Robot Collaboration for Assembly Pilot
- Mobile manipulation for assistive robotics in a domestic environment or in care institutions:
    - Assistive Mobile Manipulation Pilot
- Modular Educational Robot Pilot

The project is open for constructive suggestions from the community for further pilots or extensions to existing pilots, as long as “platform”, “composability” and “model-tool-code” are first-class citizens of those suggestions.

# Modular Education Robot Pilot

This Pilot aims at validating the RobMoSys methodology by applying it in an educational scenario. The general idea is to enable School and University teachers and students to access robot technology without any technical knowledge of robotics, in order to design novel application and educational activities that involve a robot system.

Hence, the main objectives of the Pilot are the following:

- Enable users to easily design new application with a simple user interface.
- Enable users to easily design new end-effector for the robot arm.
- Enable users to easily integrate the robot with web interface.

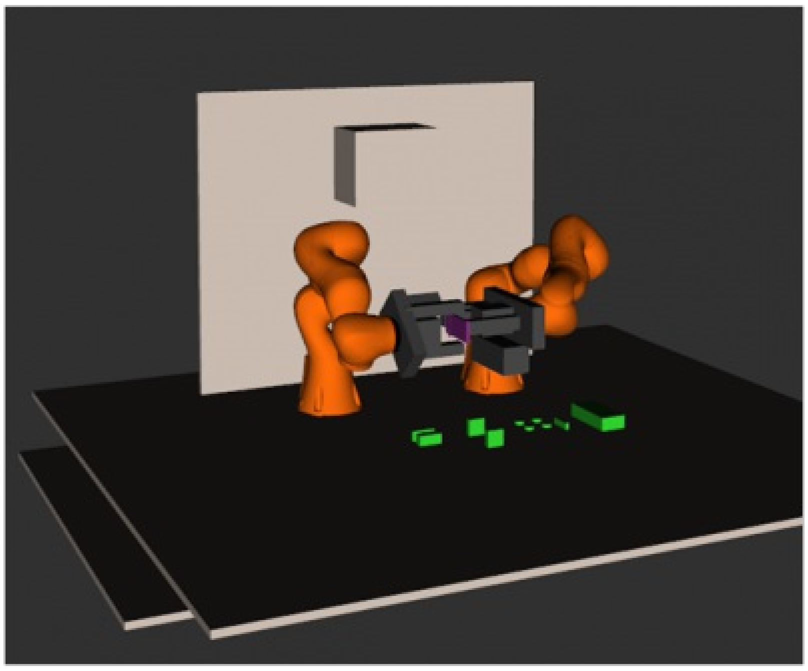
The pilot demonstrates the following user stories:

- Free from hidden interface / Replacement of components(s): the user wants to create a new interface for an existing eDO robot without interfering with the existing components (e.g., a robot hand designed for enabling the robot to communicate with tactile Sign Language tSL) or replace joints with custom object designed by the user without interfering with the functions of the robot.
- Safety: The teachers want the robot to limit critical properties and add working constraints when robot are used by children and underage student
- Quality of Service: the user doesn’t want waste time to configure and setup the robot. The robot need auto-configurable its system and its interface with the educational environment.

The Pilot will use the e.DO Robot from Comau.

# Flexible Assembly Cell Pilot

The objective of this Pilot is to validate the RobMoSys methodology by applying it on a discrete manufacturing task within a highly-flexible assembly cell. The pilot will validate the methodology through all stages, from design to task execution. Some of the performance indicators that will be considered include robustness, ease of integration and monitoring.



The assembly cell has a high degree of autonomy and does not rely on special-purpose tools or sensors. It consists of two robotic arms in a shared workspace, each equipped with a 2D or 3D camera for perception and a gripper for object manipulation.

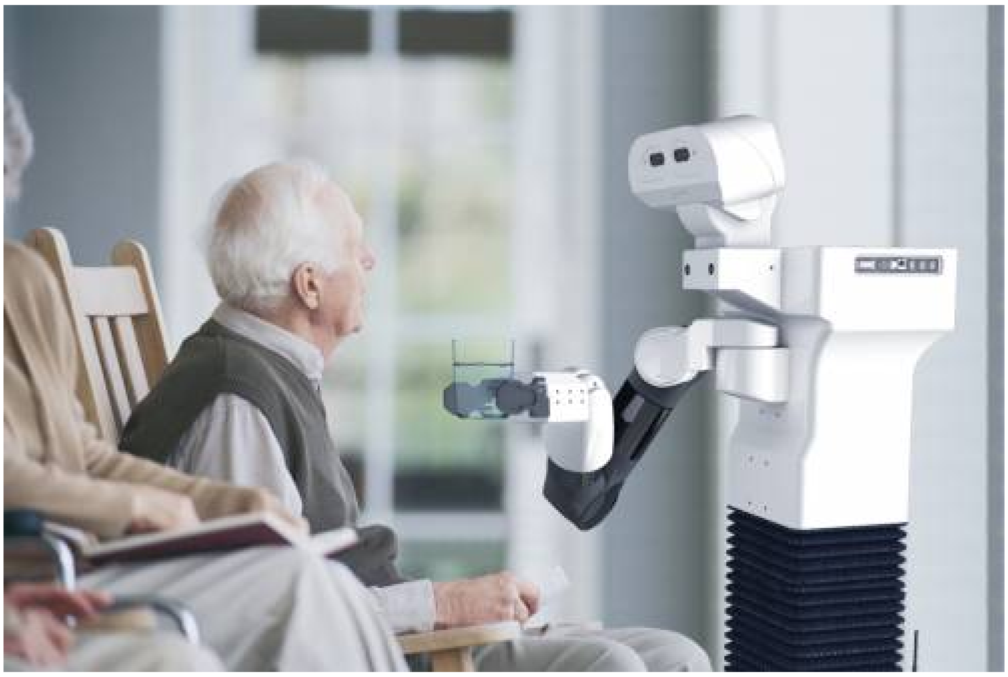
This pilot demonstrates:

- Modeling of a discrete assembly task: the cell operator should be able to specify different assembly tasks using reusable and composable task blocks without having to know the details of the software and hardware that will be ultimately realizing the task.
- Replacing a hardware component: the system builder should be able to replace a hardware component and check whether the system can still perform all the required tasks.

# Assistive Mobile Manipulation Pilot

The objective of this Pilot is to validate the RobMoSys methodology by applying it to an assistive robotics scenario in a domestic environment. The pilot will validate the methodology across all stages from design to task execution.

Some of the performance indicators that will be considered include ease of integration, flexibility when adapting to a new customer's needs (e.g. a person with a specific physical constraint, such as blindness) and effortless comparison between different alternatives using metrics.



Stage: apartment of a person with some physical constraints. The TIAGo mobile manipulator as an assistant for the person.

This pilot demonstrates:

- Replacement of component(s): the System Builder wants to select and replace from available robot end-effectors the one which best fits the requirements and expectations of the person with physical constraints, taking into account specific metrics (provided quality, offered configurability, provided skills, price and licensing, etc.).
- Free from hidden interference: the Component Developer wants to create a new interface for an existing TIAGo robot without interfering with the existing components (e.g a tablet for the hard of hearing person or an audio interface that uses a microphone and a speaker for a blind person).
- Composition of components: the System Builder wants to create a new TIAGo robot check via the data sheet (in the form of a digital model) whether the new building block (the interface) fits into the system given the constraints of the system and the variation points of the building block.

Available RobMoSys Software Baseline:

- The pilot is related to the Gazebo/TIAGo/SmartSoft Scenario. It runs the TIAGo platform with the flexible navigation stack in the SmartSoft World.

- The pilot uses the TIAGo [http://tiago.pal-robotics.com] robot from PAL Robotics.

---

pilots:assistive-manipulation · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/pilots:assistive-manipulation

# Intralogistics Industry 4.0 Robot Fleet Pilot

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. It can be used to showcase the performance of goods delivery and according non-functional requirements.

- Video of the pilot in action [https://www.youtube.com/watch?v=qRSDxBOUVx0]

This pilot demonstrates:

- Task level composition
- Service-based composition of software components

## Available RobMoSys Software Baseline

- The pilot is built using the SmartMDSD Toolchain by composing SmartSoft components
- The pilot features the flexible navigation stack.
- Components are coordinated using Robotics behavior coordination in SmartSoft: SmartTCL
- The pilot uses a fleet of Robotino3 robots. A packaged set of several components for immediate use, including those from the navigation stack with the Robotino3 platform can be downloaded from openrobotino.org [http://wiki.openrobotino.org/index.php?title=Smartsoft]. The navigation stack is also usable with the Gazebo/Tiago/SmartSoft Scenario.

## Pilot Roadmap

The SmartMDSD Toolchain v3 is currently being extended with a focus on conformance to the RobMoSys composition structures. A stable and feature-complete version is expected for release end of 2017. By 1st of

March 2018, the here described pilot will be supported by the SmartMDSD Toolchain v3. This includes software components with full support of:

- Gazebo/TIAGo/SmartSoft Scenario in simulation using the Gazebo simulator.
- Navigation Stack using FESTO Robotino3 and Pioneer P3DX.

More software components and support for fleet coordination to follow. Further development steps and future roadmap of this Pilot in the course of the RobMoSys project will follow with the publication of the second open call.

---

# Human Robot Collaboration for Assembly Pilot

The objective of this Pilot is to validate the RobMoSys methodology in the context of advanced manufacturing where humans and robots are working together in the same production site. This pilot has 2 main objectives:

- Safety certification of the production site based on model-based risk analysis.
- Modeling once, using everywhere: reusing task description for several robots

This pilot demonstrates:

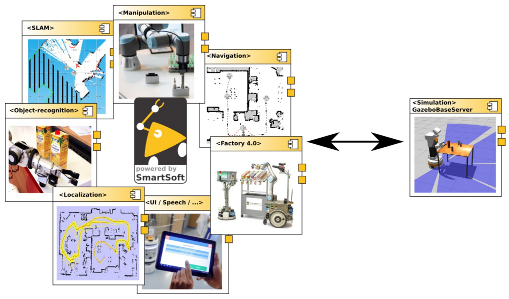
- Safety certification: The system integrators/the safety experts should be guided to set up a new production site or to evaluate an existing one through the RobMoSys tools. Those tools should assist the users to choose the appropriate configuration of the production site in order to be conformant to safety norms.
- Easing the development of robotics systems: The components offered by the RobMoSys ecosystem should be composable and easy to configure. The design and deployment tasks should be in the reach of non-expert users.
- Flexibility and resistance to low-level changes: The system builder and the integrators should be able to design their task and to deploy it on different robots having the same capabilities.

RobMoSys Wiki

http://www.robmosys.eu

# Tools and Software Baseline

RobMoSys provides a set of tools and a software baseline that already conform to the RobMoSys approach. This set can serve as a starting-point for implementations or demonstrations.



## Tooling Baseline

- Roadmap of Tools and Software
- Development Environments and Tools
    - SmartSoft World
    - Papyrus for Robotics
    - to be extended

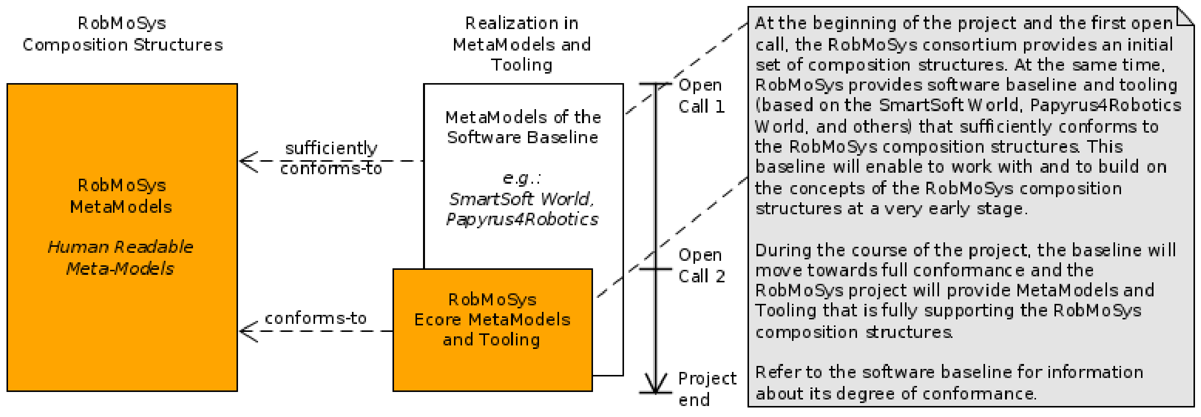
## Tier 3: Existing Building Blocks and Scenarios

- Components
    - SmartSoft Components
- Scenarios and Systems
    - Gazebo/Tiago/SmartSoft Scenario
    - Cause-Effect-Chain Example Scenario

baseline:start · Last modified: 2018/06/29 17:54
http://www.robmosys.eu/wiki-sn-02/baseline:start

# Roadmap of Tools and Software

The RobMoSys project makes a software baseline available to early work with concepts of RobMoSys composition structures. This includes already existing metamodels and tooling, for example from the The SmartSoft World and Papyrus4Robotics World.


RobMoSys Composition Structures
Realization in MetaModels and Tooling
RobMoSys MetaModels
Human Readable Meta-Models
sufficiently conforms-to
conforms-to
MetaModels of the Software Baseline
e.g.: SmartSoft World, Papyrus4Robotics
RobMoSys Ecore MetaModels and Tooling
Open Call 1
Open Call 2
Project end

At the beginning of the project and the first open call, the RobMoSys consortium provides an initial set of composition structures. At the same time, RobMoSys provides software baseline and tooling (based on the SmartSoft World, Papyrus4Robotics World, and others) that sufficiently conforms to the RobMoSys composition structures. This baseline will enable to work with and to build on the concepts of the RobMoSys composition structures at a very early stage.

During the course of the project, the baseline will move towards full conformance and the RobMoSys project will provide MetaModels and Tooling that is fully supporting the RobMoSys composition structures.

Refer to the software baseline for information about its degree of conformance.

## See also

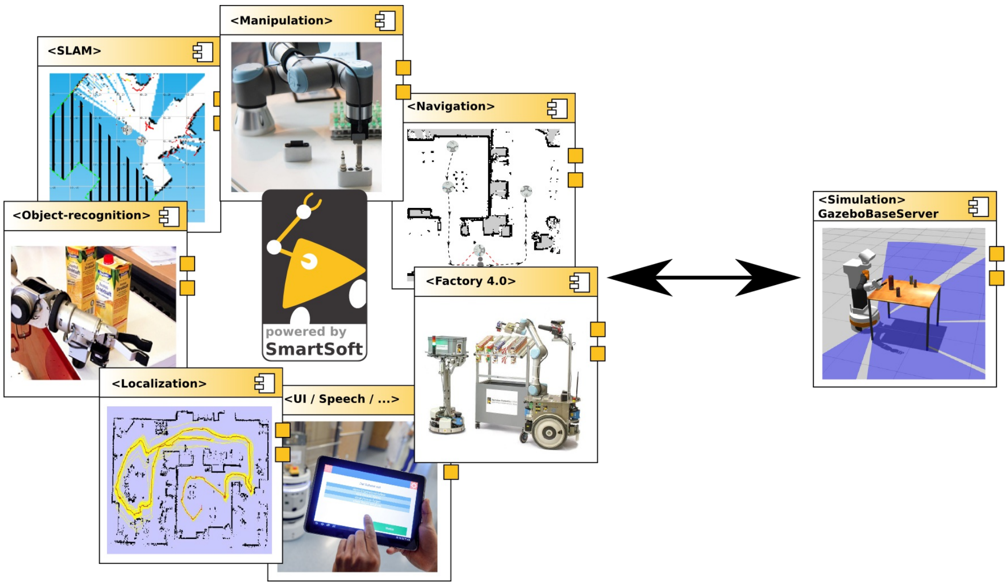
- Roadmap of MetaModeling
- Conformance of SmartSoft to RobMoSys composition structures

baseline:roadmap · Last modified: 2018/06/29 17:54
http://www.robmosys.eu/wiki-sn-02/baseline:roadmap

# Gazebo/TIAGo/SmartSoft Scenario

This scenario contributes to the Pilot mobile manipulation for assistive robotics in a domestic environment or in care institutions and Intralogistics Industry 4.0 Robot Fleet Pilot

The robot platform TIAGo from Pal-Robotics is accessible in the SmartSoft World. A scenario was set up in which you can use the SmartSoft navigation stack and SmartTCL for behaviour coordination to move TIAGo around in the Gazebo simulator.



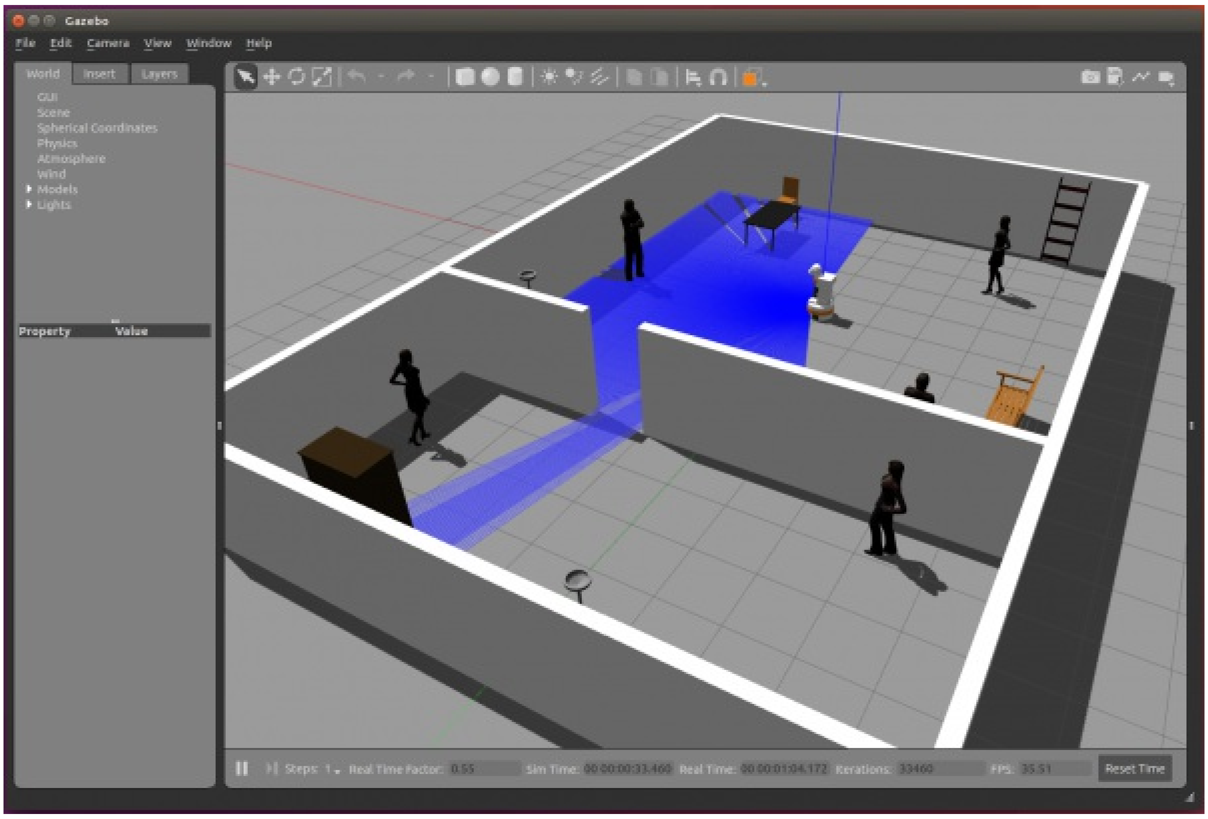
The TIAGo robot platform in simulation can be used with the SmartMDSD Toolchain as available software for the open calls where we emphasize: “do not re-invent in open call projects but build on existing technologies and tools”.

The scenario includes:

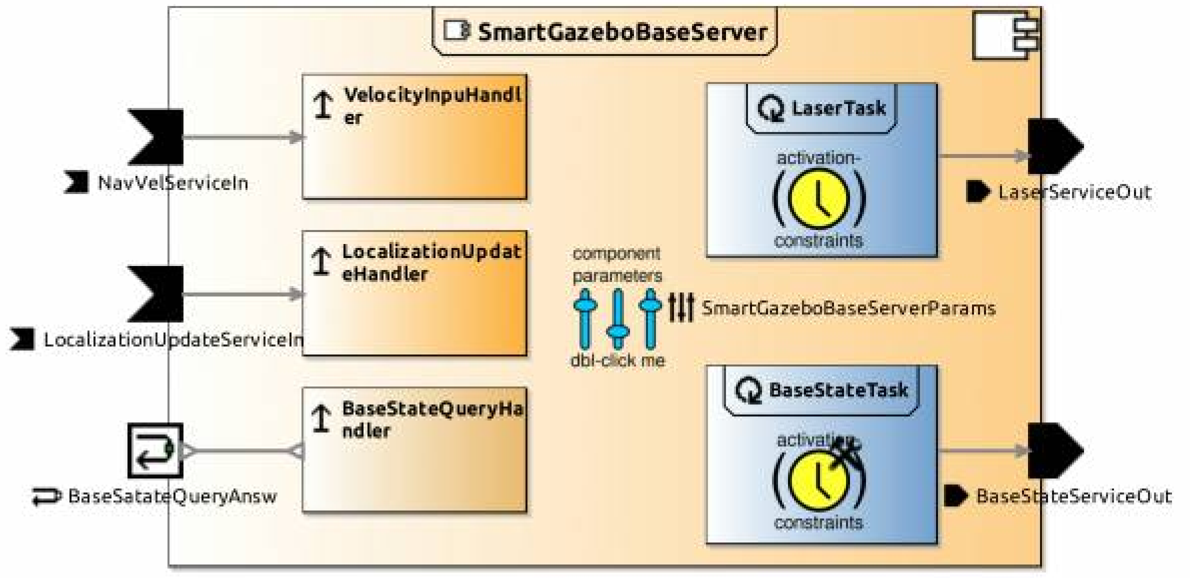
- Navigation Stack: obstacle avoidance (CDL), recording maps with Gmapping, localization, path planning
- SmartTCL for behavior coordination to move TIAGo around in the gazebo simulator

## Available Baseline: Gazebo/TIAGo with the SmartMDSD Toolchain v3

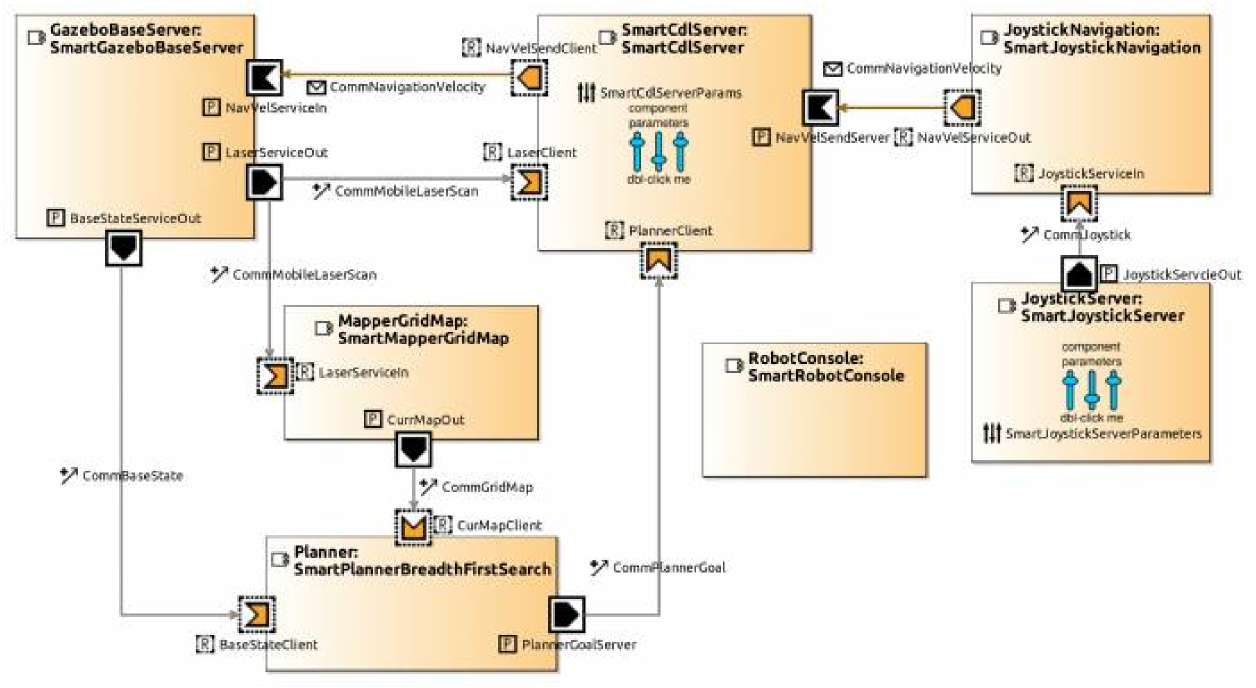
The models and components to run the Pal-Robotics TIAGo using SmartSoft/SmartMDSD Toolchain within Gazebo are available in the SmartMDSD Toolchain v3 Virtual Machine as described here. If you are interested in trying out the scenario with the SmartMDSD Toolchain v2, please refer to http://www.servicerobotik-ulm.de/drupal/?q=node/91 [http://www.servicerobotik-ulm.de/drupal/?q=node/91].



Open the SmartMDSD Toolchain in the virtual machine and take a look at the components. The main software component that interacts with the Gazebo Simulation [http://gazebosim.org/] environment is the SmartGazeboBaseServer [https://github.com/Servicerobotics-Ulm/ComponentRepository/tree/master/SmartGazeboBaseServer] component.



This component internally communicates with the Gazebo Simulation and provides communication-services that are used by the other navigation components [https://github.com/Servicerobotics-Ulm/ComponentRepository] (as shown in the figure below).



The easiest way to test the components is to use the fully configured Virtualbox image with precompiled component binaries and configured Gazebo Simulation environment with preloaded TIAGo models.

You will find a **Readme.txt** on the desktop within the virtual machine providing step-by-step instructions to run the scenario. In short, to run the full scenario, open a Terminal within the virtual machine and type in:

```
# ./Desktop/start-tiago-deployment.sh start
```

Wait until the Gazebo simulation starts, loads the Tiago models and all the navigation componets start within individual XTerms. Select the XTerm with the title “SmartRobotConsole” (be aware that some XTerms might start on top of other XTerms thus hiding them).

- Within SmartRobotConsole XTerm type in the menu number: **99** (for selecting the Demos)
- Within the next menu, type in the number **2** (for the Planner-CDL Goto demonstration)
- Now the menu should ask to give in a new goal coordinate x/y in mm for the robot to drive to. As an example type in:
    - **(-3000)(8000)**

This coordinate should command the robot to drive to a neigbour room on the right.

Enjoy!

In order to stop the scenario, select your first Terminal window and type in:

```
# ./Desktop/start-tiago-deployment.sh stop
```

# IDEs & Toolchains

Placeholder page. Please refer to Tools and Software Baseline.

# The SmartSoft World

SmartSoft is an umbrella term for concepts and tools to build robotics systems. The SmartSoft approach [http://www.servicerobotik-ulm.de/drupal/?q=node/19] defines a systematic component-based robotics software development methodology and according model-driven tools [http://www.servicerobotik-ulm.de/drupal/?q=node/20] that support different developer roles in a collaborative design and development of robotic software systems. The SmartSoft World includes (a non-complete list):

- The **SmartMDSD Toolchain**: an Integrated Development Environment (IDE) for robotics software development using model-driven software development.
- The **SmartMARS Meta-Model**: It defines the structures behind the service-oriented and component-based approach.
- The **SmartSoft Framework and implementation**: two exchangeable reference implementations (current: **ACE** middleware, former: **CORBA** middleware) and execution containers for several platforms and operating systems.
- A **repository with open source software components** for immediate reuse to compose new applications (sensor access, skills, task sequencing, knowledge representation, etc.). They have been built with the SmartSoft technologies and tools.

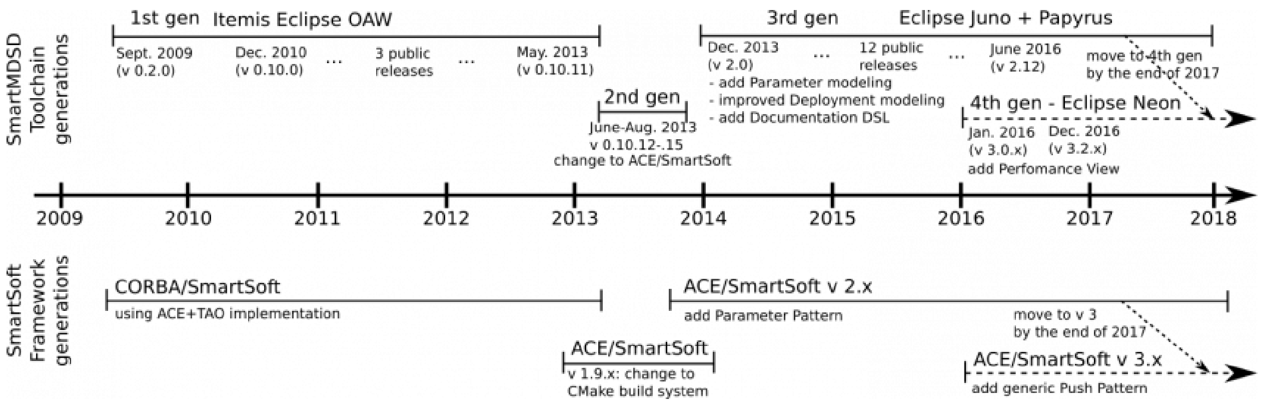
There are two main technology clusters in SmartSoft that adhere to the RobMoSys structures. One is the SmartSoft robotics framework that provides a C++ library for programming robotics software components independent of the underlying communication middleware. The other technology is the SmartMDSD Toolchain that directly implements the RobMoSys metamodels and conforms to the RobMoSys structures. It serves as a baseline for model-driven tooling.

SmartSoft is officially supported by FESTO Robotino [http://www.festo-didactic.com/int-en/learning-systems/education-and-research-robots-robotino/robotino-for-research-and-education-premium-edition-and-basic-edition.htm] (see also Robotino Wiki [http://wiki.openrobotino.org/index.php?title=Smartsoft]).

See: Getting started with the SmartSoft World [http://www.servicerobotik-ulm.de/drupal/?q=node/7]

## SmartMDSD Toolchain and the SmartSoft Framework

The SmartMDSD Toolchain has been introduced in 2009 and has been continuously refined and extended in various public releases and three generations since then. The figure below shows the main generations of the SmartMDSD Toolchain and the SmartSoft robotics framework.



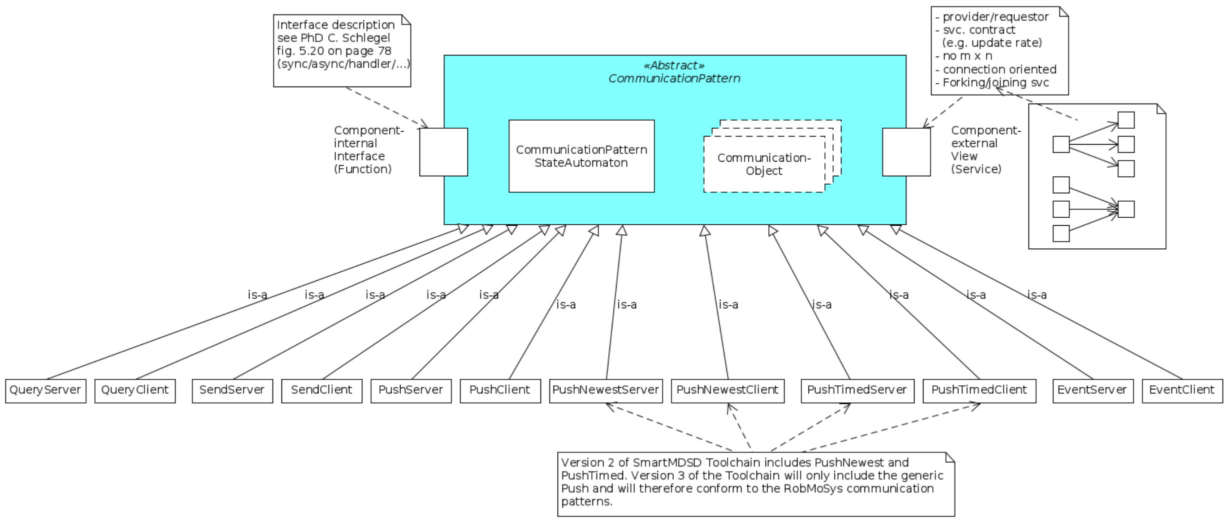
## Productive Releases

We encourage to use the latest stable release of the v3 toolchain. See The SmartMDSD Toolchain

The SmartMDSD Toolchain (version 2.x) and the SmartSoft framework (version 2.x) are very matured (TRL 6) are – among others – used by FESTO Robotino. They will be supported for a while but are not fully conform to RobMoSys. The RobMoSys staff is happy to support you in choosing the right version depending on your needs. (To all RobMoSys Integrated Projects: approach your coaches for help!)

## Conformance to RobMoSys Composition Structures

The SmartSoft software baseline is continuously evolving to match the latest developments in robotics software engineering methods. While many current SmartSoft structures already now fully conform to the RobMoSys definitions, there are some necessary refinements that are summarized below.


Interface description see PhD C. Schlegel fig. 5.20 on page 78 (sync/async/handler/...)
«Abstract» CommunicationPattern
Component-internal Interface (Function)
CommunicationPattern StateAutomaton
Communication-Object
Component-external View (Service)
- provider/requestor
- svc. contract (e.g. update rate)
- no m x n
- connection oriented
- Forking/joining svc
is-a
QueryServer
QueryClient
SendServer
SendClient
PushServer
PushClient
PushNewestServer
PushNewestClient
PushTimedServer
PushTimedClient
EventServer
EventClient
Version 2 of SmartMDSD Toolchain includes PushNewest and PushTimed. Version 3 of the Toolchain will only include the generic Push and will therefore conform to the RobMoSys communication patterns.

Further differences between the current SmartMARS Metamodel and the RobMoSys composition structures will be described in the same way here.

## Licenses: SmartSoft is open source

The ACE/SmartSoft framework version 3 is licensed under the LGPL v3 license. The SmartMDSD Toolchain v2.x uses the LGPL v2.1 license. The SmartMDSD Toolchain v3.x is licensed under 3-clause BSD license. The SmartSoft components come in various open-source licenses (e.g. GPL/LGPL, see individual component).

## Separation of Levels and Concerns in SmartSoft

SmartSoft provides implementations for the individual levels listed in Separation of Levels and Separation of Concerns:

| Level | Available/Accessible in the SmartSoft World |
|---|---|
| Mission | SmartTCL HL Interface |
| Task Plot | SmartTCL Task Block |
| Skill | SmartTCL Skill Block |
| Service | Service Definitions:<br>- Communication Object (data structure)<br>- Communication Patterns (comm. semantics)<br>SmartSoft Components |
| Function | C++ Library (libOpenRave) |
| Execution Container | SmartTask |
| OS/Middleware | ACE, CORBA, DDS, Linux, Windows, iOS |
| Hardware | UR5, Sick, ARM, x86, Robotino, Segway, MARS |

## Robotics Behavior in SmartSoft

SmartTCL [http://www.servicerobotik-ulm.de/drupal/?q=node/84] (and the concept of Dynamic State Charts [http://www.servicerobotik-ulm.de/drupal/?q=node/87]) are realizations of the Architectural Pattern for Task-Plot Coordination (Robotic Behaviors)

## SmartSoft Terminology

To be extended

### Communication Object

- A self-contained entity to hold and access information that is being exchanged via services between components in SmartSoft.
- Communication objects are ordinary C++-like objects that define the data structure and implement middleware-specific access methods and optional user access methods (getter and setter) for convenient access.
- See also the RobMoSys definition for Communication Objects

### Communication Pattern

Communication Patterns are a set of few but sufficient characteristics for the exchange of information over services for component interaction in SmartSoft. Communication patterns are fix set of software patterns defining recurring communication solutions for robotics software components. SmartSoft provides communication patterns for the sake of composability, for example *send*, two-way *request-response*, and *publish/subscribe* mechanisms on a timely or availability basis. SmartSoft communication patterns are an implementation of the Architectural Pattern for Communication

### Framework

Abstracts away platform-specific details such as independence of a particular operating-system (OS) and communication middleware by providing a unified and platform independent API.

### Quality of Service

Quality of Service (QoS) defines the ability of a system to meet application-specific customer needs and expectations while remaining economically competitive. (see Wikipedia service-quality)

## Further Resources

All about the SmartSoft World can be found at http://www.servicerobotik-ulm.de [http://www.servicerobotik-ulm.de]. Selected links:

- Getting started with SmartSoft [http://www.servicerobotik-ulm.de/drupal/?q=node/7] provides an overview and starting point
- Use SmartSoft and Gazebo to run the PAL robotics Tiago[http://www.servicerobotik-ulm.de/drupal/?q=node/91] in simulation

### Selected Publications

- Dennis Stampfer, Alex Lotz, Matthias Lutz, and Christian Schlegel. "The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software." In: Journal of Software Engineering for Robotics (JOSER): Special Issue on Domain-Specific Languages and Models in Robotics (DSLRob) 7.1 (2016). ISSN 2035-3928, pp. 3–19. Link [http://joser.unibg.it/index.php/joser/article/view/91]
- Alex Lotz, Arne Hamann, Ralph Lange, Christian Heinzemann, Jan Staschulat, Vincent Kesel, Dennis Stampfer, Matthias Lutz, and Christian Schlegel. "Combining Robotics Component-Based Model-Driven Development with a Model-Based Performance Analysis." In: IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR). San Francisco, CA, USA, Dec. 2016, pp. 170–176. LINK [http://dx.doi.org/10.1109/SIMPAR.2016.7862392]
- Matthias Lutz, Dennis Stampfer, Alex Lotz, and Christian Schlegel. "Service Robot Control Architectures for Flexible and Robust Real-World Task Execution: Best Practices and Patterns." In: Workshop Roboter-Kontrollarchitekturen, co-located with Informatik 2014. Vol. P-232. GI-Edition: Lecture Notes in Informatics (LNI). ISBN: 978-3-88579-626-8. Stuttgart: Bonner Köllen Verlag, 2014. LINK [https://www.gi.de/service/publikationen/lni/gi-edition- proceedings- 2014/gi-edition-lecture-notes-in-informatics-lni-p-232.html]

See also: Further Publications [http://www.servicerobotik-ulm.de/drupal/?q=node/15] and Technical Reports [http://www.servicerobotik-ulm.de/drupal/?q=node/18] in context of SmartSoft.

# Support for Service-based Composition

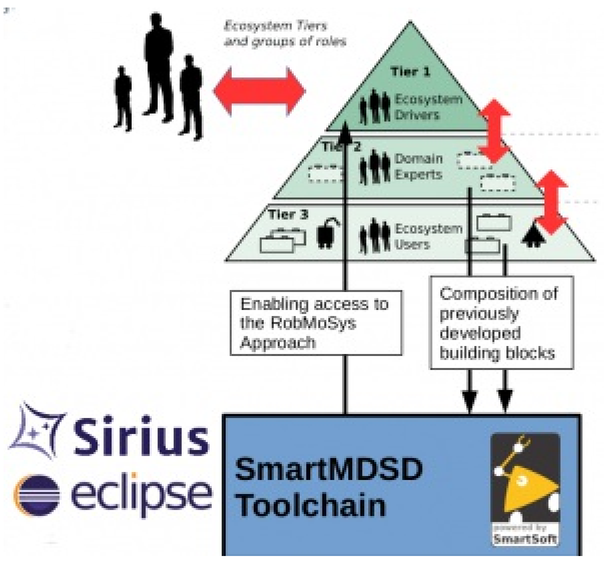
This page uses the SmartMDSD Toolchain to illustrate the support forService-based Composition. Therefore, the Gazebo/TIAGo/SmartSoft Scenario is used as an example.

This page is a placeholder. Please refer to the Gazebo/TIAGo/SmartSoft Scenario that already uses the principles of service-based composition.

# SmartMDSD Toolchain Support for the RobMoSys Ecosystem Organization

This page describes how the SmartSoft World and the SmartMDSD Toolchain supports the three composition tiers in the RobMoSys Ecosystem.

The SmartMDSD Toolchain is an Integrated Development Environment (IDE) for robotics software to support system composition by realizing the RobMoSys composition structures (i.e., the RobMoSys meta-models) for the three composition tiers in the RobMoSys Ecosystem.

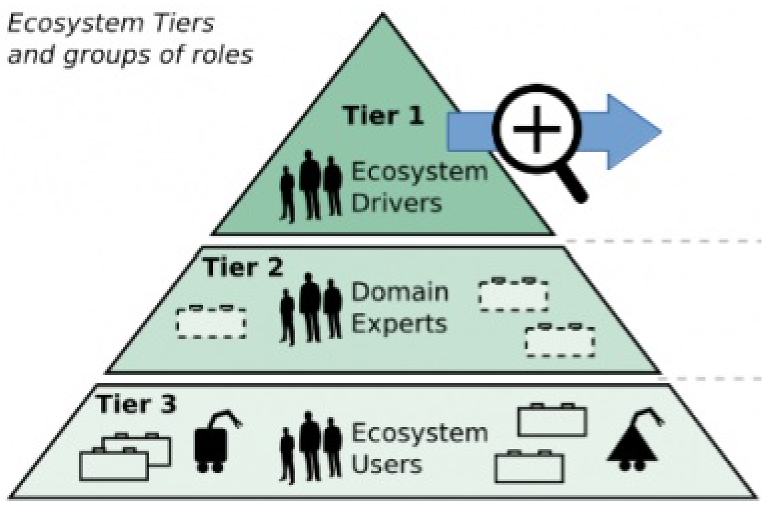

Ecosystem Tiers and groups of roles
Tier 1
Ecosystem Drivers
Tier 2
Domain Experts
Tier 3
Ecosystem Users
Enabling access to the RobMoSys Approach
Composition of previously developed building blocks
Sirius
eclipse
SmartMDSD Toolchain
powered by SmartSoft

Therefore, SmartMDSD Toolchain provides textual and graphical model editors, and implements a fully fledged code-generator that generates C++ code for SmartSoft Software Components. Moreover, dedicated model editors of the SmartMDSD Toolchain support the different developer roles in their individual responsibilities according to their respective modeling view. Existing content, such as the Flexible Navigation Stack developed with the SmartMDSD Toolchain demonstrates the usability of the modeling tools and provides initial content to be used and extended by external parties.

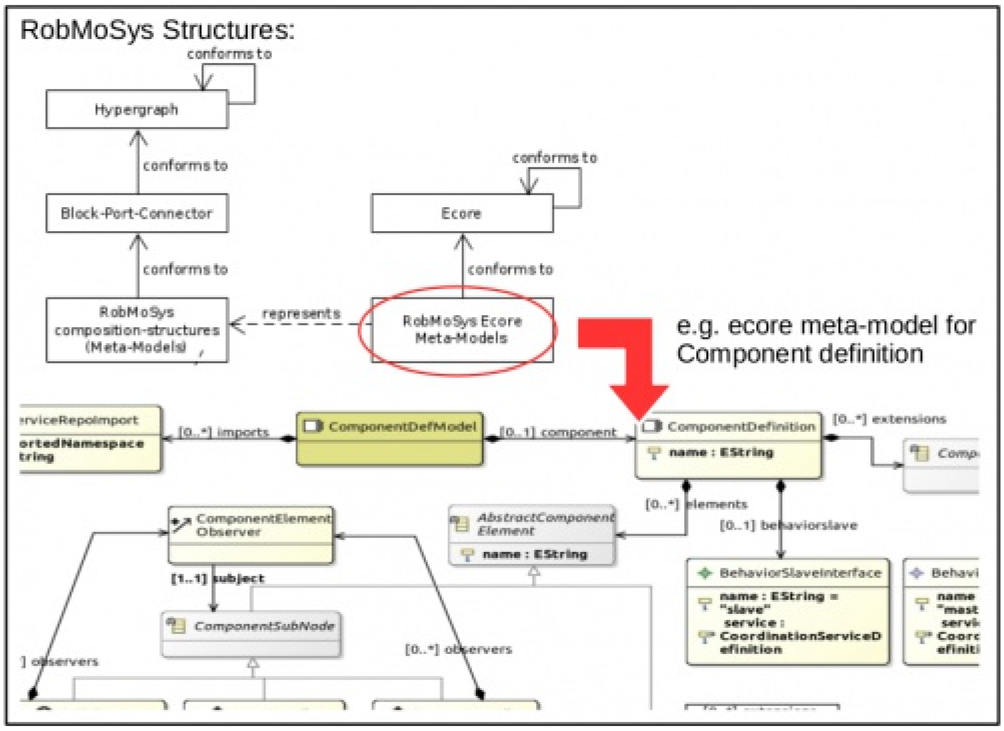
The SmartMDSD Toolchain is available as a standalone installation [http://www.servicerobotik-ulm.de/files/SmartMDSD_Toolchain/releases/] and as a virtual machine image [http://web2.servicerobotik-ulm.de/files/virtual-machine/] that includes a fully configured SmartSoft installation and the components of the Navigation Stack.

## Support for Composition Tier 1

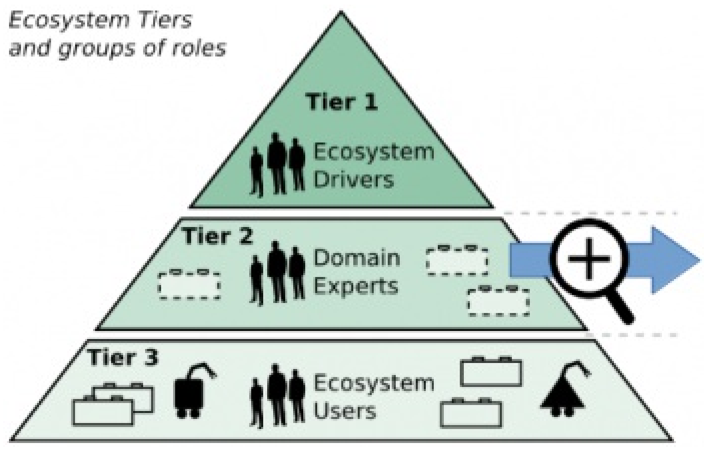
The SmartMDSD Toolchain implements the RobMoSys composition structures using Eclipse Ecore.



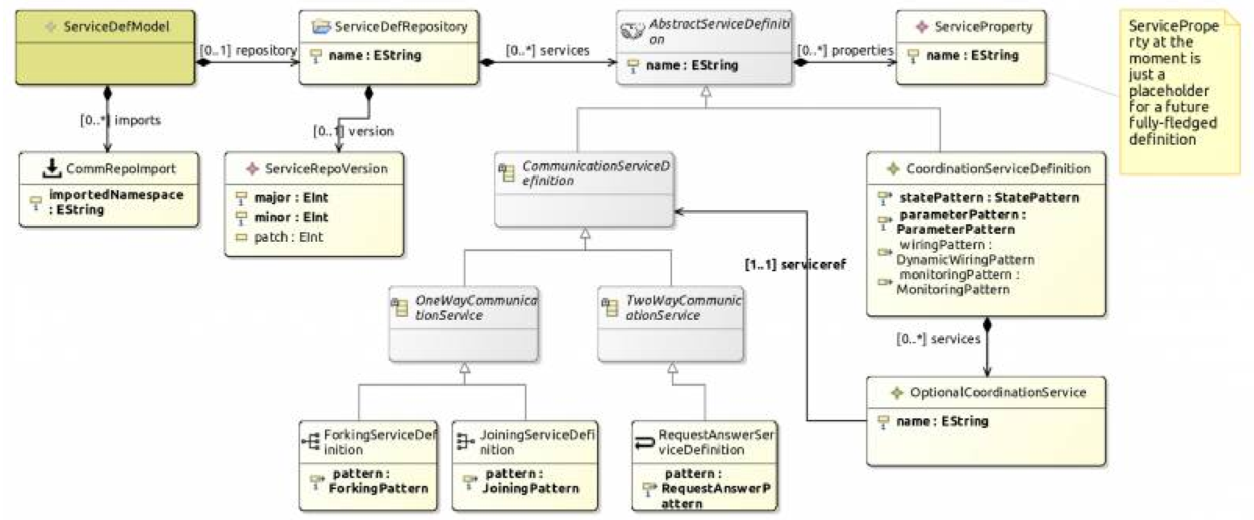
The figure on the right illustrates by the example of the component meta-model how the RobMoSys composition structures are realized based on Eclipse Ecore. This and many other meta-models are implemented within the SmartMDSD Toolchain and are used to provide dedicated model editors for specific roles at the lower Tiers 2 and 3.



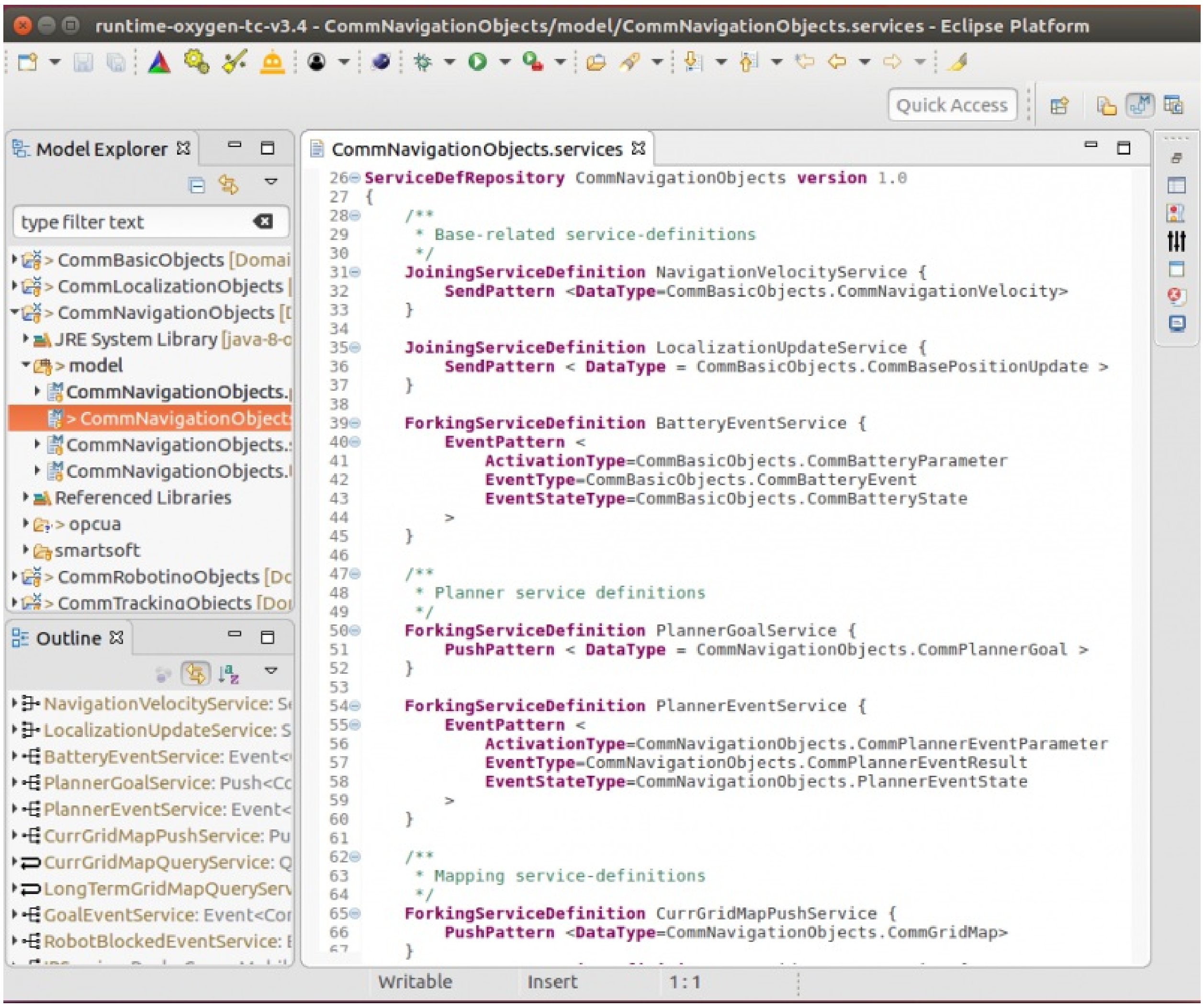
## Support for Composition Tier 2



The SmartMDSD Toolchain supports in modeling **domain structures** (i.e., domain models) that conform to the RobMoSys composition structures defined at Tier 1 (see above). On the one hand, this means that the Toolchain internally implements the related Service-Definition Metamodel (see the Ecore diagram below) as part of Tier 1, and, on the other hand, the Toolchain provides relevant model editor (see the Eclipse screenshot below) to support the involved Service Designer role in the definition of**domain models** (i.e., service definitions). These **domain models** are used at the next Tier 3 to (a) implement components that realize specific services and to (b) compose systems by interconnecting required and provided services of related components.


ServiceDefModel
[0..1] repository
ServiceDefRepository
name : EString
[0..*] services
AbstractServiceDefinition
name : EString
[0..*] properties
ServiceProperty
name : EString
ServiceProperty at the moment is just a placeholder for a future fully-fledged definition
[0..*] imports
CommRepoImport
importedNamespace : EString
[0..1] version
ServiceRepoVersion
major : EInt
minor : EInt
patch : EInt
CommunicationServiceDefinition
CoordinationServiceDefinition
statePattern : StatePattern
parameterPattern : ParameterPattern
wiringPattern : DynamicWiringPattern
monitoringPattern : MonitoringPattern
[1..1] serviceref
OneWayCommunicationService
TwoWayCommunicationService
[0..*] services
OptionalCoordinationService
name : EString
ForkingServiceDefinition
pattern : ForkingPattern
JoiningServiceDefinition
pattern : JoiningPattern
RequestAnswerServiceDefinition
pattern : RequestAnswerPattern

The SmartMDSD Toolchain screenshot below shows an excerpt of the **domain models** of the Flexible Navigation Stack.


runtime-oxygen-tc-v3.4 - CommNavigationObjects/model/CommNavigationObjects.services - Eclipse Platform
Quick Access
Model Explorer
type filter text
> CommBasicObjects [Domai
> CommLocalizationObjects
> CommNavigationObjects [I
JRE System Library [java-8-o
> model
CommNavigationObjects.
> CommNavigationObject
CommNavigationObjects.
CommNavigationObjects.
Referenced Libraries
> opcua
smartsoft
> CommRobotinoObjects [Dc
> CommTrackingObjects [Do
Outline
NavigationVelocityService: S
LocalizationUpdateService: S
BatteryEventService: Event<
PlannerGoalService: Push<Cc
PlannerEventService: Event<
CurrGridMapPushService: Pu
CurrGridMapQueryService: Q
LongTermGridMapQuerySen
GoalEventService: Event<Cor
RobotBlockedEventService: E
CommNavigationObjects.services
Writable
Insert
1:1

```
ServiceDefRepository CommNavigationObjects version 1.0
{
    /**
     * Base-related service-definitions
     */
    JoiningServiceDefinition NavigationVelocityService {
        SendPattern <DataType=CommBasicObjects.CommNavigationVelocity>
    }

    JoiningServiceDefinition LocalizationUpdateService {
        SendPattern < DataType = CommBasicObjects.CommBasePositionUpdate >
    }

    ForkingServiceDefinition BatteryEventService {
        EventPattern <
            ActivationType=CommBasicObjects.CommBatteryParameter
            EventType=CommBasicObjects.CommBatteryEvent
            EventStateType=CommBasicObjects.CommBatteryState
        >
    }

    /**
     * Planner service definitions
     */
    ForkingServiceDefinition PlannerGoalService {
        PushPattern < DataType = CommNavigationObjects.CommPlannerGoal >
    }

    ForkingServiceDefinition PlannerEventService {
        EventPattern <
            ActivationType=CommNavigationObjects.CommPlannerEventParameter
            EventType=CommNavigationObjects.CommPlannerEventResult
            EventStateType=CommNavigationObjects.PlannerEventState
        >
    }

    /**
     * Mapping service-definitions
     */
    ForkingServiceDefinition CurrGridMapPushService {
        PushPattern <DataType=CommNavigationObjects.CommGridMap>
    }
```

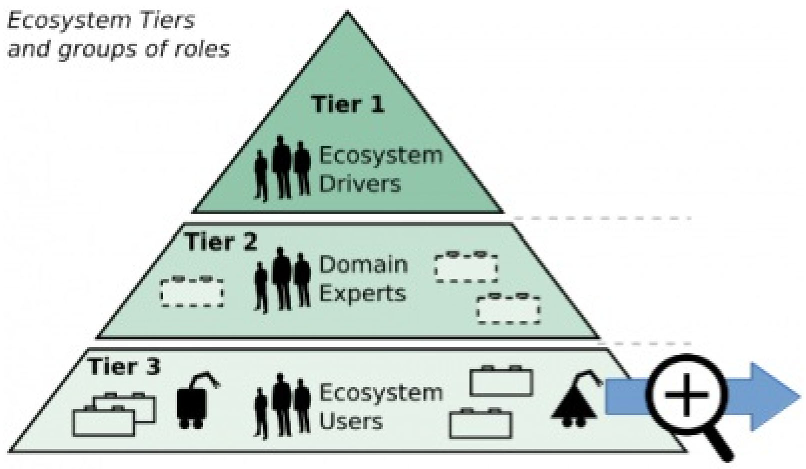
All the domain models of the Flexible Navigation Stack and other stacks are publicly available for immediate use in the Github repository:

- DomainModelsRepositories [https://github.com/Servicerobotics-Ulm/DomainModelsRepositories]

The main developer role at this Tier 2 is the Service Designer.

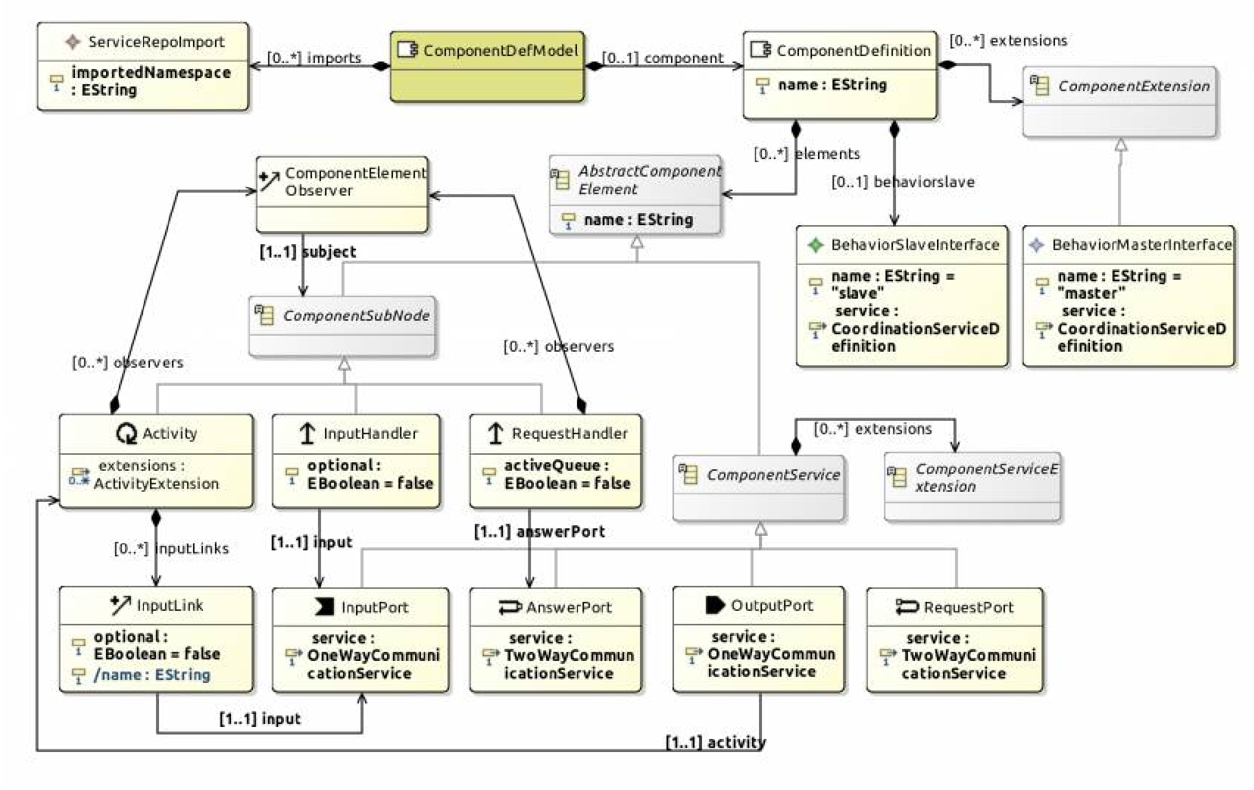
## Support for Composition Tier 3

The Tier 3 is about adding content to the Ecosystem in the form of reusable software components and systems. The SmartMDSD Toolchain supports in **developing components** and in **composing** previously developed components to **systems**, as well as **deploying** systems to robotic **target platforms**.

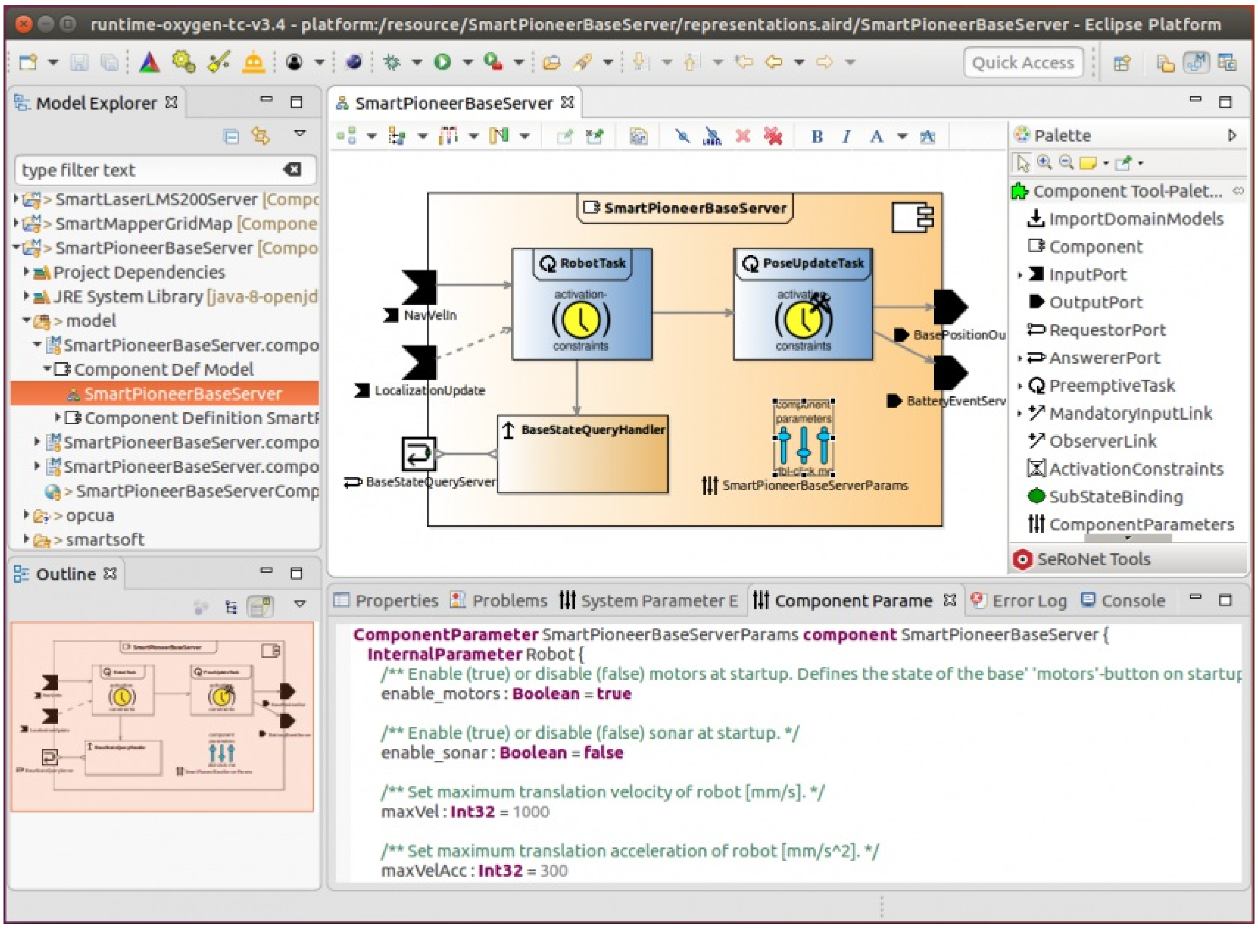


Similar to Tier 2, the SmartMDSD Toolchain implements several RobMoSys composition structures (i.e., Ecore-based meta-models) for Tier 3 related to component development, system composition, and deployment. On the one hand, at this Tier 3, Component Suppliers can develop individual components that realize selected service definitions (i.e., domain models from Tier 2). On the other hand, System Builders can compose components to new systems. Other roles, such as Performance Designer, Safety Engineer, and Behavior Developer cooperatively contribute to a system from different modeling viewpoints. This Tier 3 consists of the majority of Toolchain users as these are all the Ecosystem participants who provide concrete content and who compete with building block alternatives with unique selling points thus altogether realizing a robotics component and system market.

The figure below shows the Component-Definition Metamodel based on Ecore as an example. Several other meta-models are realized within the SmartMDSD Toolchain as well. A most recent version of the meta-model realizations can be found in the SmartMDSD Toolchain sources (which are open-source using the BSD3 License).



Based on the Component-Definition Metamodel (shown in the Ecore diagram above), the SmartMDSD Toolchain implements a graphical Component-Definition model editor, that allows modeling components such as e.g. the *PioneerBaseServer* component shown in the screenshot below.



```
ComponentParameter SmartPioneerBaseServerParams component SmartPioneerBaseServer {
 InternalParameter Robot {
  /** Enable (true) or disable (false) motors at startup. Defines the state of the base' 'motors'-button on startup
  enable_motors : Boolean = true

  /** Enable (true) or disable (false) sonar at startup. */
  enable_sonar : Boolean = false

  /** Set maximum translation velocity of robot [mm/s]. */
  maxVel : Int32 = 1000

  /** Set maximum translation acceleration of robot [mm/s^2]. */
  maxVelAcc : Int32 = 300
```

Several fully implement components based on the SmartMDSD Toolchain and the SmartSoft framework can be found in this Github repository:

- https://github.com/Servicerobotics-Ulm/ComponentRepository [https://github.com/Servicerobotics-Ulm/ComponentRepository]

Besides of the component-development view (that is used for illustration above), the SmartMDSD Toolchain implements several other system-related modeling views that enable the related developer roles to define relevant system models.

See next:

- Flexible Navigation Stack
- Gazebo/TIAGo/SmartSoft Scenario
- The SmartMDSD Toolchain

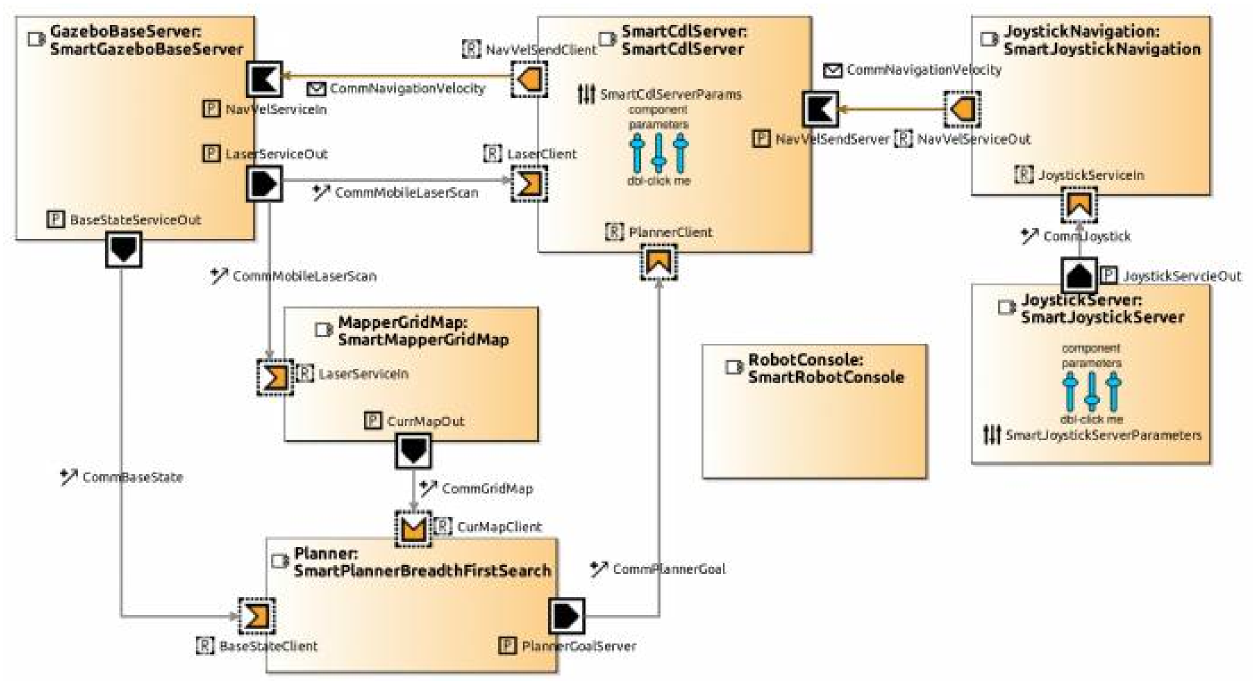
baseline:environment_tools:smartsoft:smartmdsd-toolchain:ecosystem-tiers:start · Last modified: 2018/06/29 17:54
http://www.robmosys.eu/wiki-sn-02/baseline:environment_tools:smartsoft:smartmdsd-toolchain:ecosystem-tiers:start

# Support for the Flexible Navigation Stack

This page describes how the SmartMDSD Toolchain and the SmartSoft World supports theFlexible Navigation Stack.

## Ready-to-run Example: Tiago

As one of the further baselines in RobMoSys, the SmartSoft navigation components can be used with the PAL Robotics Tiago platform within the Gazebo simulation. It features PAL Robotics Tiago: see SmartGazeboBaseServer [https://github.com/Servicerobotics-Ulm/ComponentRepository/tree/master/SmartGazeboBaseServer] as virtual robot base. This example is available "ready-to-go" in the virtual machine image. A screenshot of the SmartMDSD Toolchain displaying the flexible navigation stack:



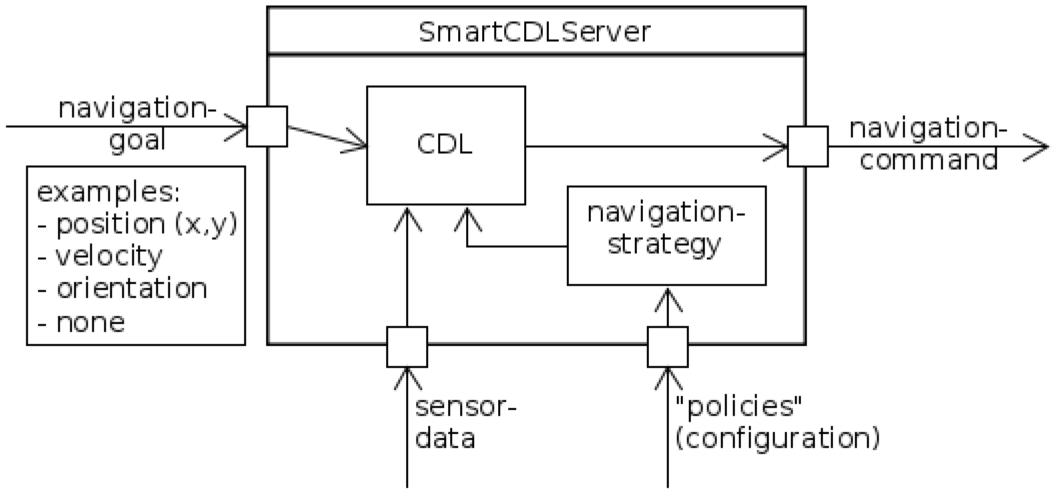
## Available Software Components in the SmartSoft World

The fife ready-to-use navigation components of the navigation stack can be downloaded from the SmartSoft Github component repository [https://github.com/Servicerobotics-Ulm/ComponentRepository]. The following list of references provides documentation for the fife navigation components:

- SmartCdlServer [https://github.com/Servicerobotics-Ulm/ComponentRepository/tree/master/SmartCdlServer]: this is the main obstacle-avoidance component that uses the Curvature Distance Lookup (CDL) [http://ieeexplore.ieee.org/document/724683/][1) approach in its core
- SmartPlannerBreadthFirstSearch [https://github.com/Servicerobotics-Ulm/ComponentRepository/tree/master/SmartPlannerBreadthFirstSearch]: this is geometrical path-planning component using a breadth-first-search algorithm
- SmartMapperGridMap [https://github.com/Servicerobotics-

Ulm/ComponentRepository/tree/master/SmartMapperGridMap]: this component calculates up to date occupancy grid maps
- SmartAmcl [https://github.com/Servicerobotics-Ulm/ComponentRepository/tree/master/SmartAmcl]: this is a localization component internally using the Adaptive Monte Carlo Localization (AMCL) [http://wiki.ros.org/amcl] algorithm.

The SmartCdlServer [https://github.com/Servicerobotics-Ulm/ComponentRepository/tree/master/SmartCdlServer] component (see figure below) deserves some further explanations. In a nutshell, this component receives laser-scans and next goals (which can be either a position, velocity, orientation or even undefined). Based on these inputs, the internal CDL algorithm calculates a set of collision-free navigation-commands. Each of these navigation-commands is equally valid, the selection of one "appropriate" one is performed upon a configurable navigation-strategy. For example, one strategy might try to maximize the overall velocity, another might try to stay in the middle of a hallway, yet another strategy might try reaching the next goal closest possible (often the default strategy). This separation between the general obstacle-avoidance and the definition of different strategies adds flexibility with respect to applicability of this component in different scenarios.



There is a list of further components related to different sensor types and robot platforms as alternatives to the above list of components: More precisely, the following two to use robot platforms are supported directly:

- Pioneer P3DX: SmartPioneerBaseServer [https://github.com/Servicerobotics-Ulm/ComponentRepository/tree/master/SmartPioneerBaseServer]

The following sensor component provides updated laser-scans using the SICK LMS200 laser scanner:

- SmartLaserLMS200Server [https://github.com/Servicerobotics-Ulm/ComponentRepository/tree/master/SmartLaserLMS200Server]: provides lase-scans.

## The Flexible Navigation Stack with FESTO Robotino3

Note: all components and links in this section refer to the v2-generation of the SmartMDSD Toolchain:

- SmartRobotionBaseServer: see the Robotino3 Wiki [http://wiki.openrobotino.org/index.php?title=Smartsoft]
- A packaged set of several components for immediate use, including those from the navigation stack with the Robotino3 platform can be downloaded from here [http://wiki.openrobotino.org/index.php?title=Smartsoft].

Another application that uses this navigation stack in a structured and coordinated fleet environment using e.g. Robotino3 robots is described in the ETFA2016 paper [http://ieeexplore.ieee.org/document/7733602/][2].

[1]

Christian Schlegel. "Fast local obstacle avoidance under kinematic and dynamic constraints for a mobile robot". In *IEEE International Conference on Intelligent Robots and Systems (IROS)* Victoria, Canada, 1998. DOI: 10.1109/IROS.1998.724683 [https://doi.org/10.1109/IROS.1998.724683].

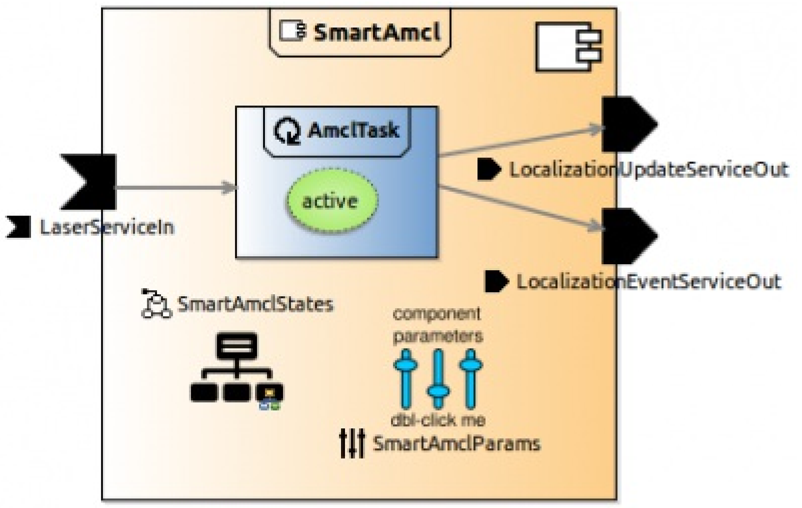
[2]

Matthias Lutz, Christian Verbeek and Christian Schlegel. "Towards a Robot Fleet for Intra-Logistic Tasks: Combining Free Robot Navigation with Multi-Robot Coordination at Bottlenecks". In *Proc. of the 21th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Berlin, September 6-9, 2016. Electronic ISBN: 978-1-5090-1314-2, DOI: 10.1109/ETFA.2016.7733602 [https://doi.org/10.1109/ETFA.2016.7733602]

# Support for Coordinating Activities and Life Cycle of Software Components

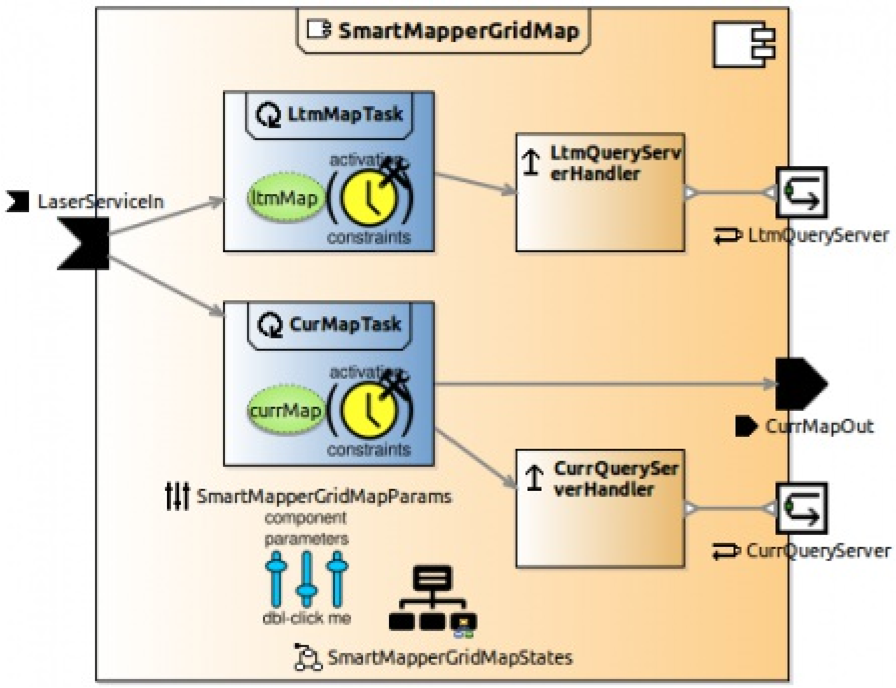
This page describes how the SmartMDSD Toolchain supports Coordinating Activities and Life Cycle of Software Components.

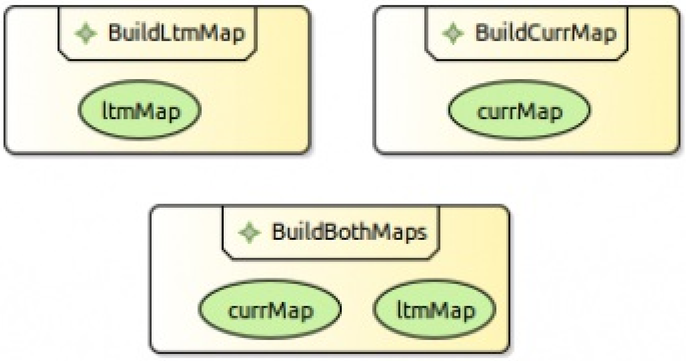
## Example Use-Cases for Component Operation Modes



As an example, the figure on the right shows the model of the “SmartAmcl” component (i.e., a component providing a localization service based on the “Adaptive Monte Carlo Localization” approach). This component internally specifies a single **activity** called “AmclTask”. Moreover, the “AmclTask” is mapped to the component's **operation mode** called “active” (see green ellipse in the figure). As stated above, the component's lifecycle does not need to be explicitly modeled as it is implicitly available for each component by default. Additionally, the component's lifecycle provides two default **operation modes** called “active” and “neutral” (as part of the “Alive” submachine within the component's lifecycle). That is, if the “active” **operation mode** is activated, then the referenced**activity** “AmclTask” is activated thus consuming the relevant resources. By contrast, switching into the “neutral” **operation mode** implicitly deactivates the**operation mode** “active” and thus the referenced**activity** “AmclTask”. In other words, the component is conveyed into a “standby” mode thus releasing the relevant resources.

The two default **operation modes** “active” and “neutral” cover the majority of simple software components that provide a single service based on a single activity with a functional block. However, more complex components allow the definition of multiple provided services and several activities within a single component. For such cases, a more detailed model of the component's **operation modes** is required.





As an example for a more complex component, the figure above provides the model of the “SmartMapper” component. This component provides three **services**, namely “LtmQueryServer”, “CurrQueryServer” and “CurrMapOut”. The first **service** provides a long-term map while the other two **services** provide access to the current map (i.e., a grid-map of a local section from the long-term map). The component internally maintains and updates both map types. There are different situations at runtime, where either one of the map types is needed, or both map types are used, or none of the map types is currently needed. The model of the component's **operation modes** (see figure on the right) supports all these cases. As can be further seen in the component model (in the figure above) the “LtmMapTask” **activity** is only active if one of the**operation modes** “BuildLtmMap” or “BuildBothMaps” is active. Respectively, the “CurrMapTask”**activity** is only active if one of the **operation modes** “BuildCurrMap” or “BuildBothMaps” is active. Please note that the “neutral” **operation mode** is not explicitly modeled as it implicitly exists for every component by default.

See also:

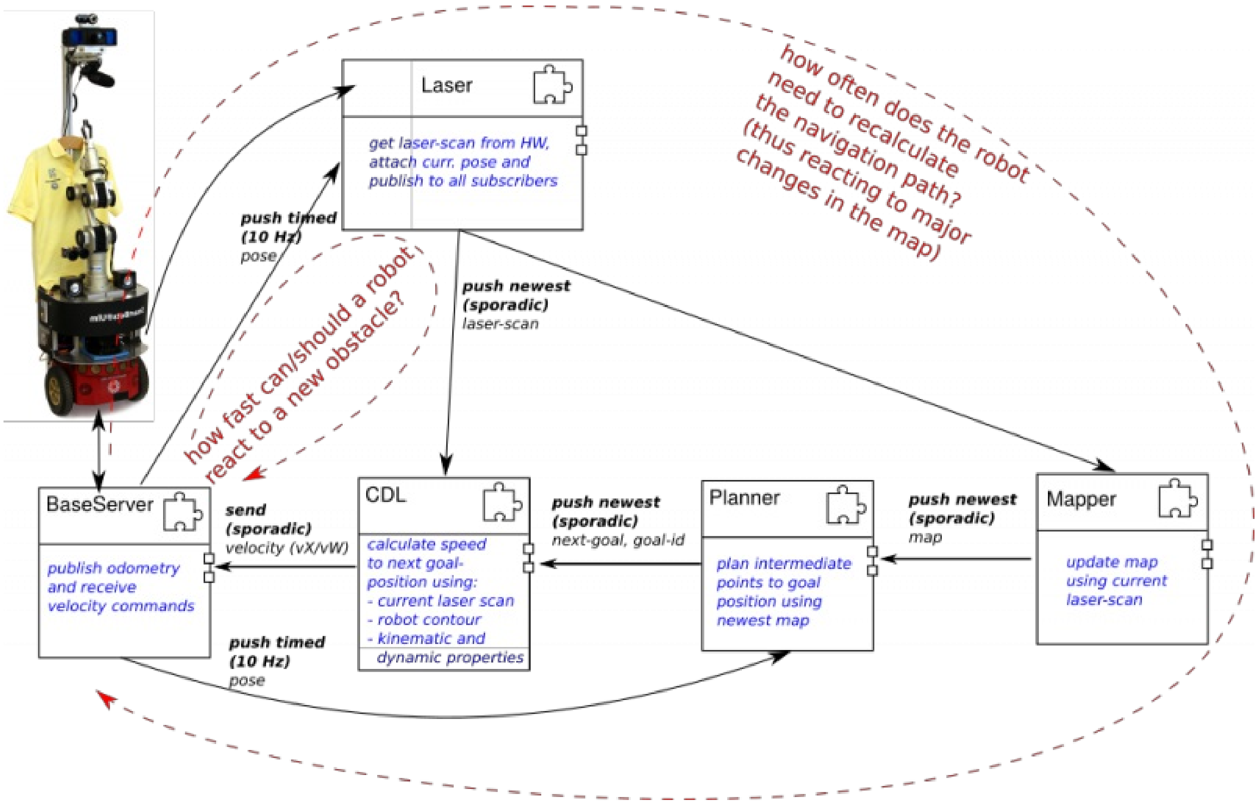
- Christian Schlegel, Alex Lotz and Andreas Steck, “SmartSoft - The State Management of a Component”, in *Technical Report 2011/01*, Hochschule Ulm, Germany, ISSN 1868-3452, 2011.PDF [http://www.zafh-servicerobotik.de/dokumente/ZAFH-TR-01-2011-ISSN-1868-3452.pdf]
- Component Development View
- Component-Definition Metamodel

# Support for Managing Cause-Effect Chains in Component Composition

This page uses the SmartMDSD Toolchain to illustrate the Management of Cause-Effect Chains in Component Composition. Therefore, the Gazebo/TIAGo/SmartSoft Scenario is used as an example.

## Example Use-Case for Managing Cause-Effect Chains

The figure below shows a schematic illustration of the Gazebo/TIAGo/SmartSoft Scenario consisting of navigation components altogether providing collision-avoidance and path-planning navigation functionality. This example is used in the following to discuss different aspects related to managing cause-effect chains which are again related to managing performance-related system aspects.

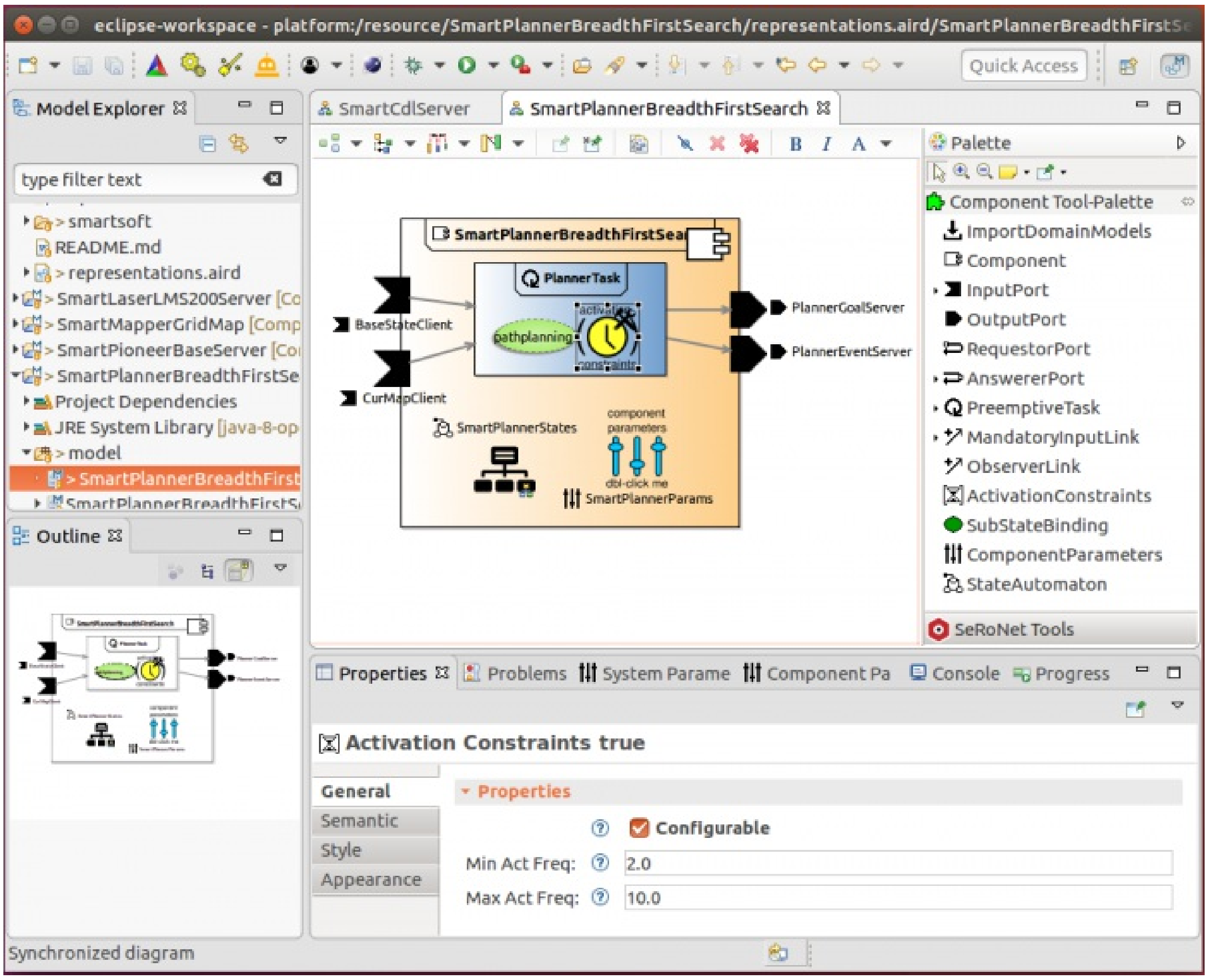

Laser
get laser-scan from HW, attach curr. pose and publish to all subscribers
how often does the robot need to recalculate the navigation path? (thus reacting to major changes in the map)
push timed (10 Hz) pose
push newest (sporadic) laser-scan
how fast can/should a robot react to a new obstacle?
BaseServer
publish odometry and receive velocity commands
send (sporadic) velocity (vX/vW)
CDL
calculate speed to next goal-position using: - current laser scan - robot contour - kinematic and dynamic properties
push newest (sporadic) next-goal, goal-id
Planner
plan intermediate points to goal position using newest map
push newest (sporadic) map
Mapper
update map using current laser-scan
push timed (10 Hz) pose

The example system in the figure above consists of five navigation components, from which two are related to hardware devices (i.e., the Pioneer Base and the SICK Laser) and the other three components respectively implementing collision-avoidance (i.e., the CDL component), mapping and path-planning. As an example, two performance-related design questions are introduced in the following with the focus on discussing the architectural choices and the relevant modeling options:

1. How fast can a robot react to sudden obstacles taking the current components into account?
2. How often does the robot need to recalculate the path to its current destination (thus reacting to major map changes)?

## The component development view

The design and management of performance-related system aspects can be approached from two different viewpoints. On the one hand, individual components can specify implementation-specific configuration boundaries (as shown in the example below). On the other hand, a system that instantiates relevant navigation components can refine their configurations (within the predefined configuration boundaries) to meet application-specific performance requirements (see next section).



Activation Constraints true

Properties

✓ Configurable

Min Act Freq: 2.0

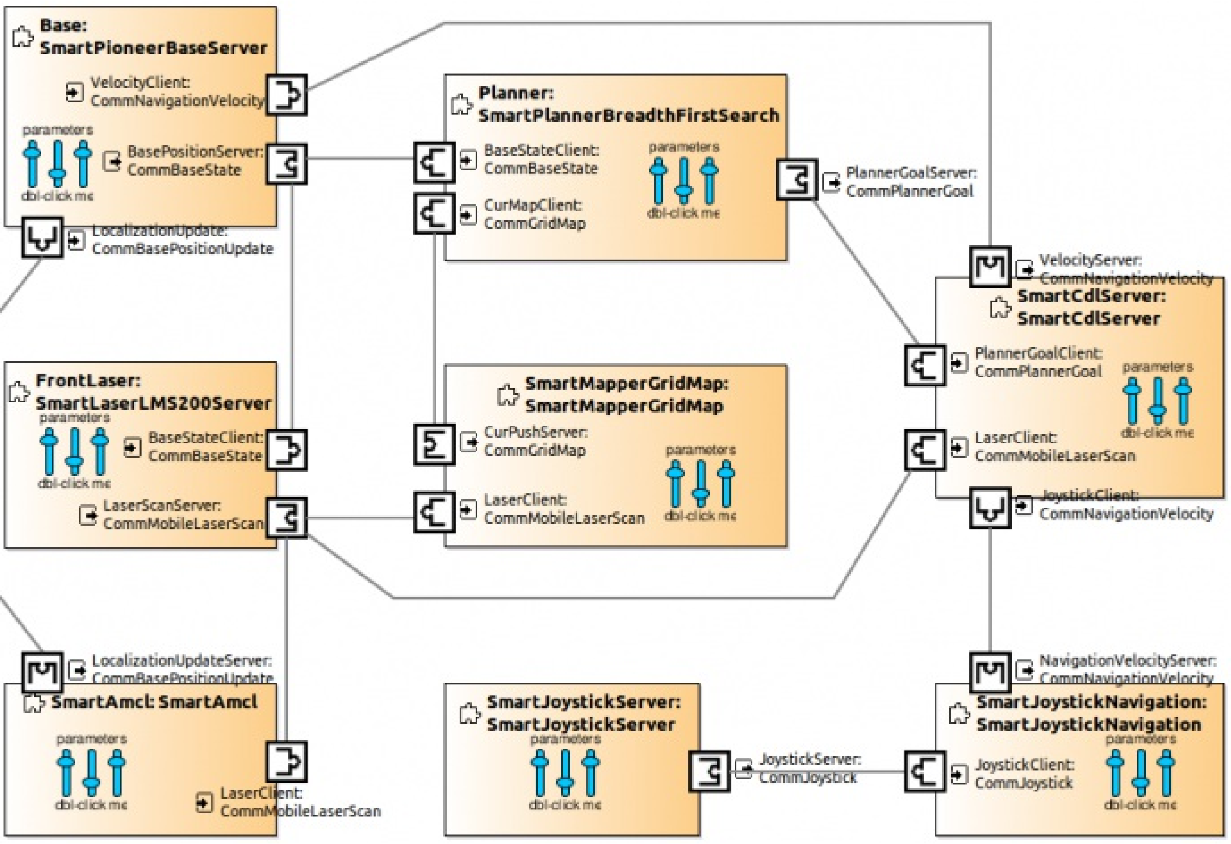
Max Act Freq: 10.0

The figure above shows the model of the "SmartPlannerBreadthFirstSearch" component as a representative example for demonstrating the role of the Component Supplier. The responsibility of this role is to define and to implement a component so that it can be (re-)used in different systems. Among other things, the component supplier also is responsible to define component-specific, performance-related constraints (if the internal business logic of this component requires specific execution characteristics). For example, the planner component (in the figure above) specifies that the "PlannerTask" should be executed with an update frequency within the boundaries from 2.0 to 10.0 Hertz and that the actual update frequency can be configured within these boundaries during a later system configuration phase.

- Component Development View
- Component Supplier Role
- Component Definition Metamodel

## The system-configuration view

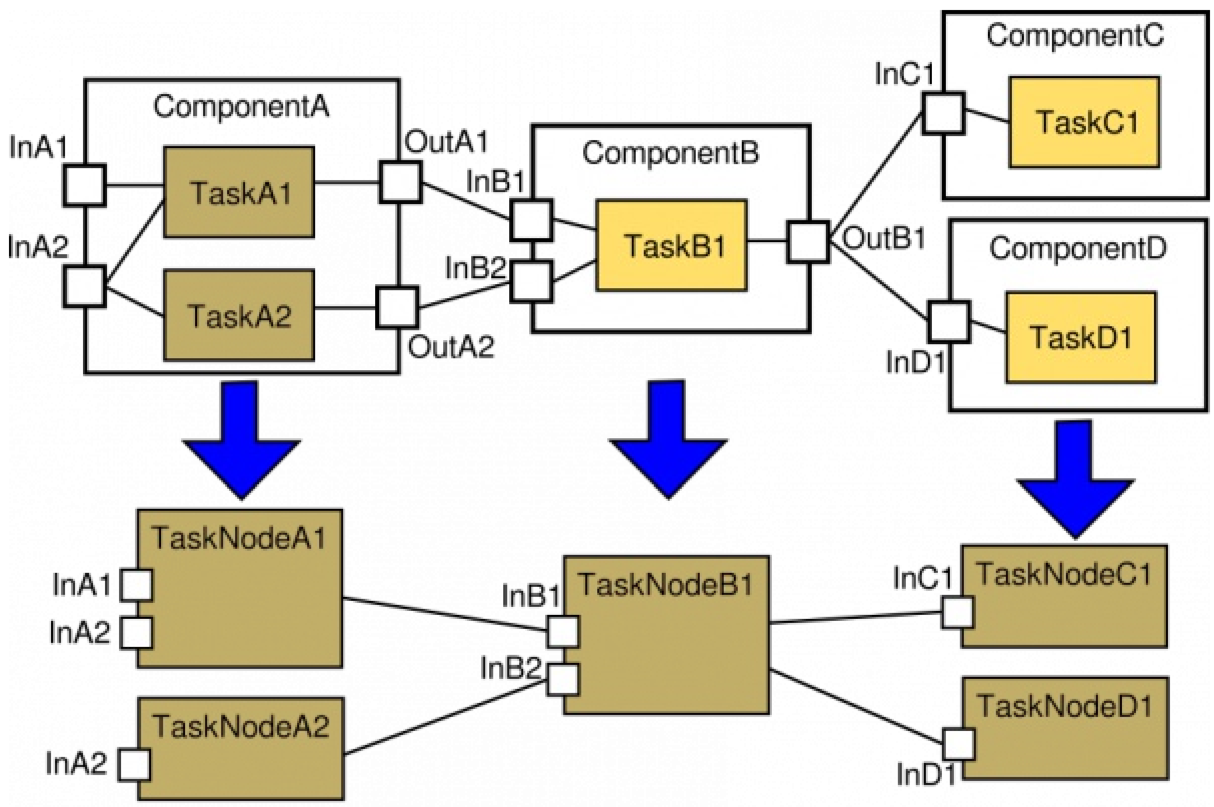
The figure below shows an example model of the navigation scenario. This model enables System Builders to instantiate and compose components to a system and to specify initial wiring as well as initial configurations of these components.


Base: SmartPioneerBaseServer
VelocityClient: CommNavigationVelocity
parameters
dbl-click me
BasePositionServer: CommBaseState
LocalizationUpdate: CommBasePositionUpdate
Planner: SmartPlannerBreadthFirstSearch
BaseStateClient: CommBaseState
CurMapClient: CommGridMap
PlannerGoalServer: CommPlannerGoal
VelocityServer: CommNavigationVelocity
SmartCdlServer: SmartCdlServer
PlannerGoalClient: CommPlannerGoal
LaserClient: CommMobileLaserScan
JoystickClient: CommNavigationVelocity
FrontLaser: SmartLaserLMS200Server
BaseStateClient: CommBaseState
LaserScanServer: CommMobileLaserScan
SmartMapperGridMap: SmartMapperGridMap
CurPushServer: CommGridMap
LaserClient: CommMobileLaserScan
LocalizationUpdateServer: CommBasePositionUpdate
SmartAmcl: SmartAmcl
LaserClient: CommMobileLaserScan
SmartJoystickServer: SmartJoystickServer
JoystickServer: CommJoystick
NavigationVelocityServer: CommNavigationVelocity
SmartJoystickNavigation: SmartJoystickNavigation
JoystickClient: CommJoystick

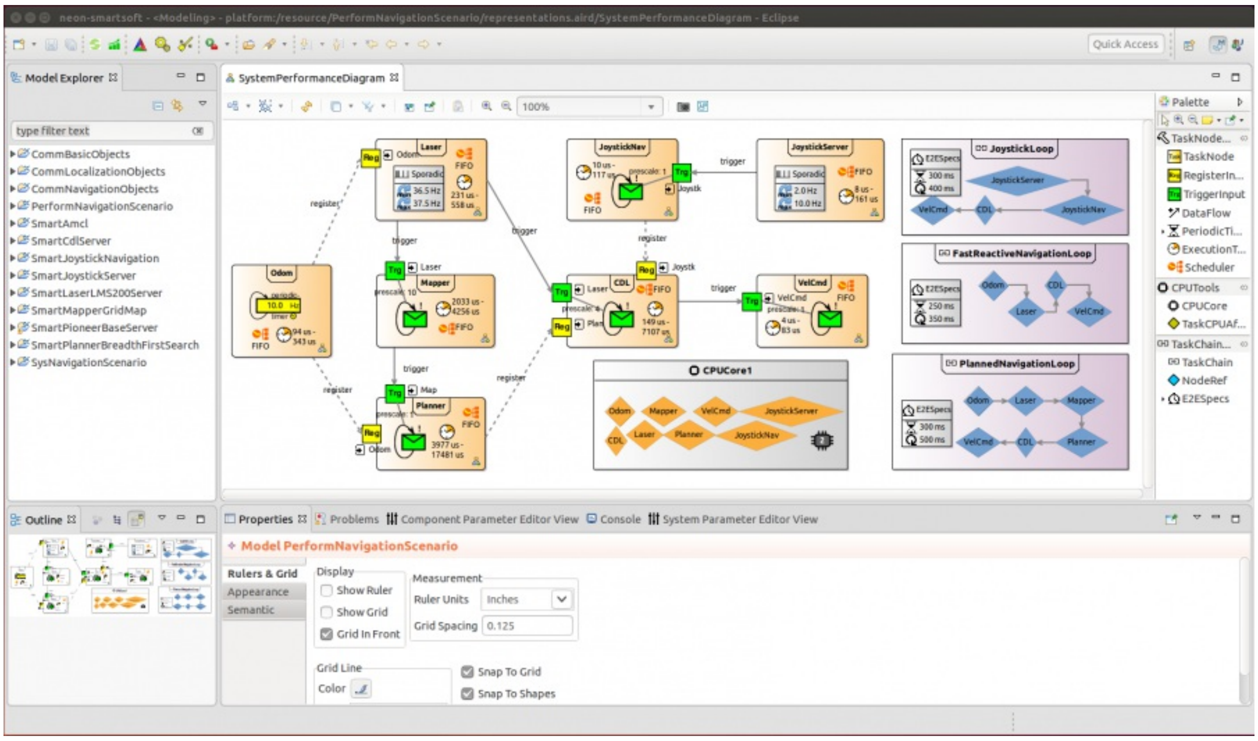
- System Configuration View
- System Builder Role
- System Component Architecture Metamodel

## The performance view

A given system (as e.g. shown in the previous section) can be refined so that performance-related configurations are designed in combination, which is the main responsibility of the Performance Designer (as discussed next).



A performance designer refines the configurations of **activity** models from the selected components of the system configuration view (see preceding section). Therefore, several **activities** are considered in combination and the component shells are blended out (as they are not relevant for this performance view). The figure above illustrates the transformation from a system-configuration model to an activity-net.



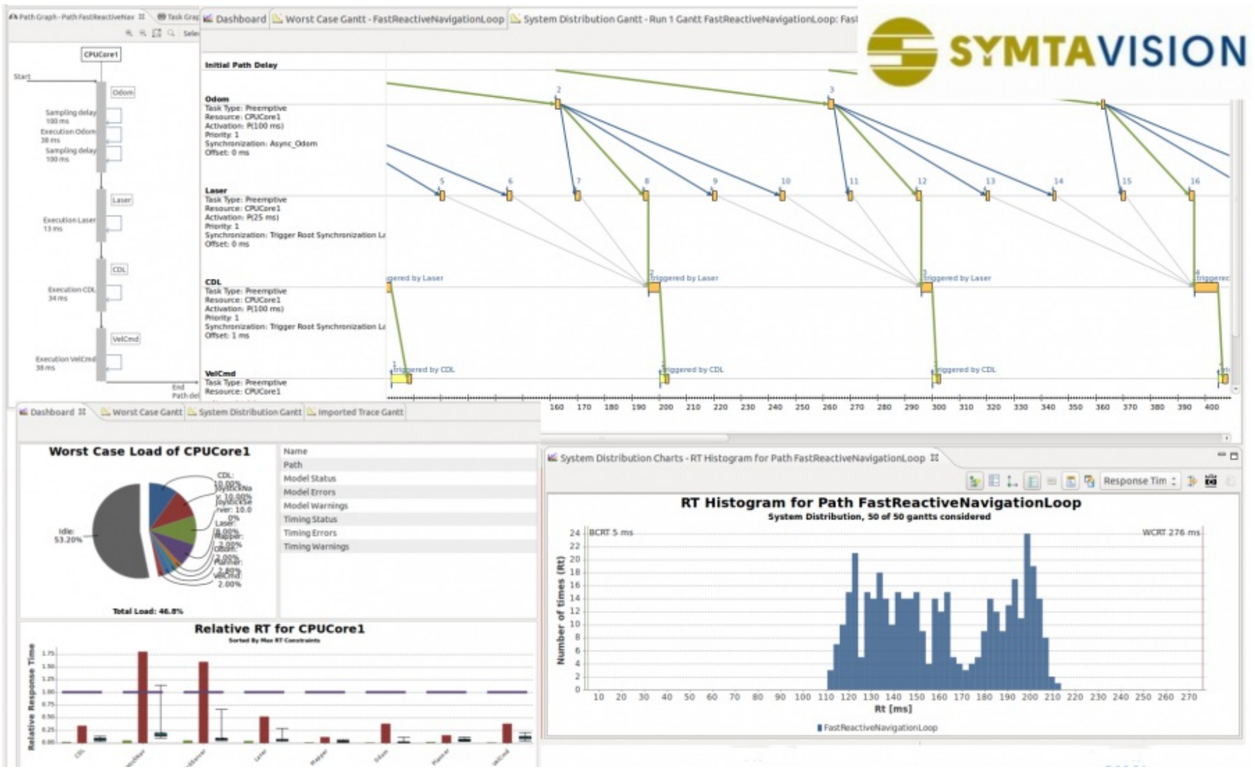
The transformation of a system-configuration model of the navigation scenario into an activity-net results in the model shown in the above figure. In this model individual **activity nodes** (orange blocks in the figure) can be refined by selecting reasonable activation semantics (i.e., selecting a DataTrigger or a PeriodicTimer as an activation-source for an **activity node**). Overall, an activity-net forms a directed graph with several paths sometimes crossing the same **activity nodes**.

In order to specify end-to-end delays, individual (acyclic) paths of the overall activity-net need to be selected. Such paths are called cause-effect chains and are visualized by the three rectangles in the above figure on the right. For each of these cause-effect chains individual end-to-end delay requirements can be specified. These end-to-end delay specifications can be now easily verified by triggering an automated performance analysis (see next).

- Performance View
- Performance Designer Role
- Cause-Effect-Chain and its Analysis Metamodels

## Performance Analysis based on SymTA/S

Based on the performance model (from the preceding section) a compositional performance analysis can be automatically triggered which simulates different run-time conditions including scheduling and sampling effects. This analysis allows verifying the specified end-to-end delays and based on the results to refine the performance model.


SYMTAVISION
Initial Path Delay
Odom
Laser
CDL
VelCmd
Worst Case Load of CPUCore1
Total Load: 46.8%
Relative RT for CPUCore1
RT Histogram for Path FastReactiveNavigationLoop
System Distribution, 50 of 50 gantts considered
BCRT 5 ms
WCRT 276 ms
Number of times (Rt)
Rt [ms]
FastReactiveNavigationLoop

As an example, the figure above shows the results of the compositional performance analysis which is calculated using the SymTA/S timing analysis tool from Luxoft [https://auto.luxoft.com/uth/timing-analysis-tools/] (formerly Symtavision). The results show for the cause-effect chain called “FastReactiveNavigationLoop” that the distribution of the overall end-to-end delays is within the specified requirements defined in the performance model.

See also:

- Managing Cause-Effect Chains in Component Composition
- Architectural Pattern for Stepwise Management of Extra-Functional Properties

- Cause-Effect-Chain and its Analysis Metamodels

## Acknowledgement

This document contains material from:

- Lotz2018 Alex Lotz, "Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System", Dissertation, Technische Universität München, München 2018. [https://mediatum.ub.tum.de/?id=1362587]

# The SmartMDSD Toolchain

The SmartMDSD Toolchain is an Integrated Development Environment (IDE) for robotics software to support system composition according to the structures of RobMoSys. It supports in applying the RobMoSys approach with the SmartSoft World.

## Download

- SmartMDSD Toolchain **standalone** download:
  - http://www.servicerobotik-ulm.de/files/SmartMDSD_Toolchain/releases/ [http://www.servicerobotik-ulm.de/files/SmartMDSD_Toolchain/releases/]
- SmartMDSD Toolchain **VirtualBox virtual machine image** (development environment preinstalled):
  - http://web2.servicerobotik-ulm.de/files/virtual-machine/ [http://web2.servicerobotik-ulm.de/files/virtual-machine/]
  - Please note: Virtualbox in Ubuntu 16.04 is experiencing some difficulties with kernel 4.13. If that is the case, select the previous LTS-kernel 4.4 at boot-time, or install the latest version 5.2 of Virtualbox from www.virtualbox.org [https://www.virtualbox.org/]

## Available documentation

A lot of documentation exists for the SmartMDSD Toolchain v2 [http://www.servicerobotik-ulm.de/drupal/?q=node/7]. As the SmartMDSD Toolchain v3 has just been released in March 2018, the documentation is currently being adapted and updated to v3. In the meantime, please refer to:

- See readme.txt on the VM guest OS desktop for further instructions how to use the virtual machine and preinstalled SmartMDSD Toolchain and SmartSoft Components
- PDF Readme [http://www.servicerobotik-ulm.de/files/SmartMDSD_Toolchain/releases/v3.4/Readme.pdf] accompanying the SmartMDSD Toolchain release
- Instructions how to use/run the Gazebo/TIAGo/SmartSoft Scenario that is included in the virtual machine image.

If you want to use the SmartMDSD Toolchain v3 you need the ACE/SmartSoft Framework v3 [https://github.com/Servicerobotics-Ulm/AceSmartSoftFramework] installed.

Documentation is currently being migrated from v2 to v3. Most parts work very similar. You might want to browse through:

- Video tutorials to be available shortly. In the meantime, refer to v2 video tutorials [https://www.youtube.com/playlist?list=PLJxdA4EZjZiWSlC4R_ChwH_UIcWXWJ8Te]
- A user manual will be available. In the meantime, refer to the User Manual for v2 [http://www.servicerobotik-ulm.de/toolchain-manual/html/]
- A screencast demonstrates the look and feel of the v3-generation of the toolchain: Screencast [https://www.youtube.com/watch?v=JIYPJXmop3U]. Please note: this screencast shows an outdated technology preview. The current v3 stable release works “similar” as shown in the technology preview.

## RobMoSys Support

This section contains specific examples (non-complete list) of how the SmartMDSD Toolchain supports the RobMoSys composition structures:

- Support for the RobMoSys Ecosystem Organization
- Support for Managing Cause-Effect Chains in Component Composition
- Support for Coordinating Activities and Life Cycle of Software Components
- Support for the Flexible Navigation Stack
- Support for Service-based Composition

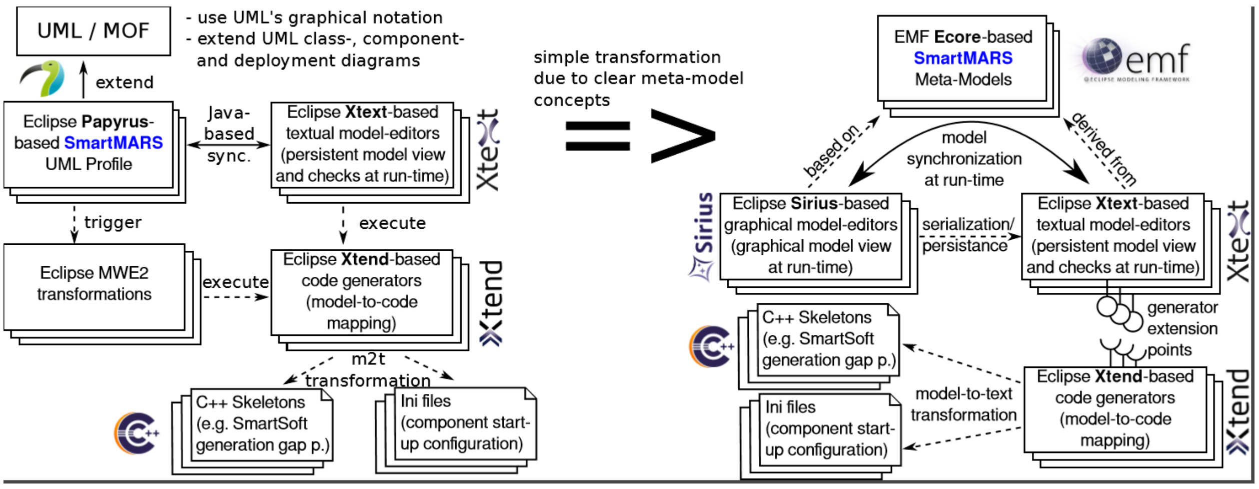
## Available Building Blocks

The following previously developed/modeled building blocks and scenarios are available for immediate use:

- Domain Models Repositories [https://github.com/Servicerobotics-Ulm/DomainModelsRepositories]: These are examples of RobMoSys Composition Tier 2
- Component Repository [https://github.com/Servicerobotics-Ulm/ComponentRepository]: These are examples of previously developed building blocks for Tier 3
- System Repository [https://github.com/Servicerobotics-Ulm/SystemRepository]: These are examples of systems and applications on RobMoSys Composition Tier 3 that are composed from the building blocks
- The SmartMDSD Toolchain features the Gazebo/TIAGo/SmartSoft Scenario, another example of a robot application on Tier 3

## Eclipse Modeling Tools

The SmartMDSD Toolchain has been using various Eclipse Modeling technologies. It started in 2009 with the Itemis Open-Architecture Ware (OAW), then between 2013 and 2016 used Xtext, Xtend and UML Papyrus and is currently moving towards using the latest Eclipse Modeling technologies based on latest Xtext, Xtend and Sirius plugins. The figure below provides a schematic overview of the Eclipse technologies used for version 2.x and the transformation with the recent Eclipse technologies for version 3.x.



UML / MOF
- use UML's graphical notation
- extend UML class-, component- and deployment diagrams
extend
Eclipse Papyrus-based SmartMARS UML Profile
Java-based sync.
Eclipse Xtext-based textual model-editors (persistent model view and checks at run-time)
trigger
execute
Eclipse MWE2 transformations
execute
Eclipse Xtend-based code generators (model-to-code mapping)
m2t transformation
C++ Skeletons (e.g. SmartSoft generation gap p.)
Ini files (component start-up configuration)
simple transformation due to clear meta-model concepts
EMF Ecore-based SmartMARS Meta-Models
emf
based on
derived from
model synchronization at run-time
Sirius
Eclipse Sirius-based graphical model-editors (graphical model view at run-time)
serialization/ persistance
Eclipse Xtext-based textual model-editors (persistent model view and checks at run-time)
generator extension points
C++ Skeletons (e.g. SmartSoft generation gap p.)
Ini files (component start-up configuration)
model-to-text transformation
Eclipse Xtend-based code generators (model-to-code mapping)
Xtext
Xtend

Overall, the SmartMDSD Toolchain provides various textual and graphical model editors as well as code generators to generate glue-logic for the SmartSoft framework and to generate configuration files.

# Papyrus4Robotics

If you already know what Papyrus4Robotics is and you just want to get started with it, see Getting Started With Papyrus4Robotics.

Otherwise, read the sections below to learn more Papyrus4Robotics.

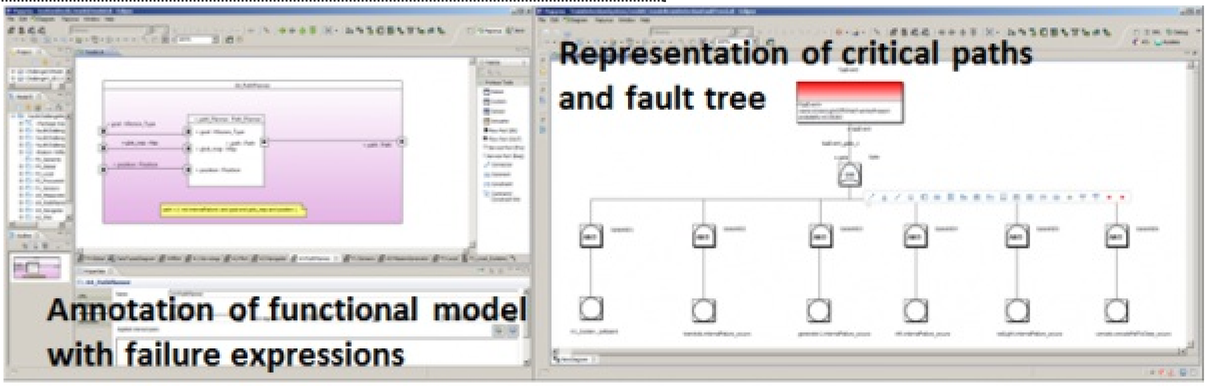
## Presentation

Papyrus is an industrial-grade open source Model-Based Engineering tool. It is based on standards and supports Model-Based Design in UML, SysML, MARTE, fUML, PSCS/SM, FMI 2.0 and many more. Papyrus has been used successfuly in industrial projects and is the base platform for several industrial modeling tools —read more about Papyrus Use Case Stories[https://eclipse.org/papyrus/testimonials.html].

To address the robotics domain according to the RobMoSys methodology and structures, a set of Papyrus-based DSLs and tools are being collected under the Papyrus4Robotics umbrella.

It is important to emphasize that RobMoSys-compliant software baselines are not in competition. Indeed, RobMoSys aims, as one of its primary goals, at the realization of a virtual integration platform built upon existing tools and standards for the development of robotic systems.

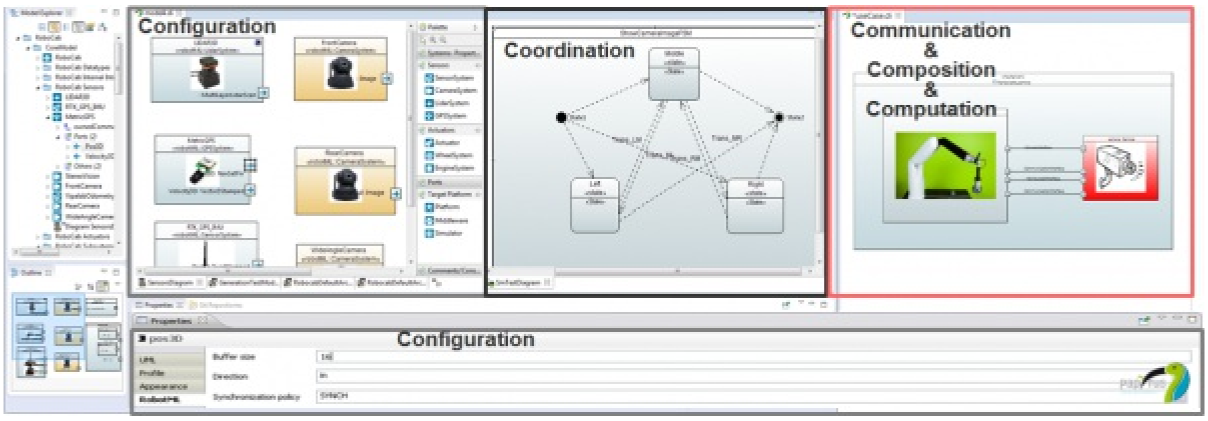
Concretely, this means that the RobMoSys approach and structures can enable model exchange and collaborative development between, e.g., safety engineers and system integrators who use different RobMoSys-compliant software baselines. As an example, SmartSoft and its large set of software components can be used to define the system's functional architecture. Then, a safety module in Papyrus4Robotics can be used to perform dysfunctional analysis on the architecture's key components, including Hazard Analysis and Risk Assessment (HARA), Failure Mode and Effects Analysis (FMEA) and Fault Tree Analysis (FTA). Model-based safety analysis would be enabled by the following components. A dedicated modeling view; a DSL with the main safety concepts for robotics, e.g., various hazards and safety requirements as specified by ISO standards 10218-1/2 (industrial robots), 15066 (collaborative industrial robots) and 13482 (personal care robots); a set of analysis and report generation modules. Read the Aldebaran's use case story [https://eclipse.org/papyrus/resources/aldebaran-usecasestory.pdf] to find out more on this subject.


Annotation of functional model with failure expressions
Representation of critical paths and fault tree

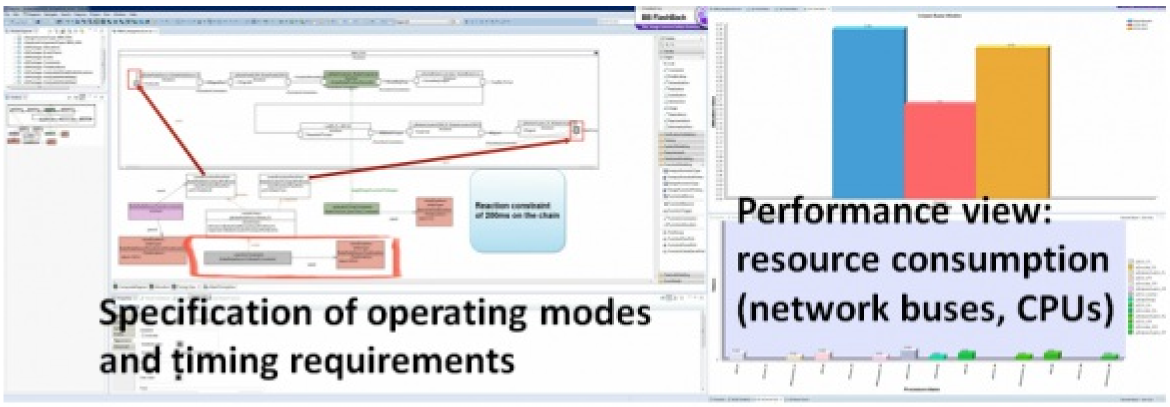
## Realization and tools

Papyrus4Robotics uses UML/SysML as underlying realization technology. The platform uses the UML profile mechanism to enable the implementation of Domain-Specific Languages (DSLs) that assist RobMoSys's ecosystem users in designing robotics systems.

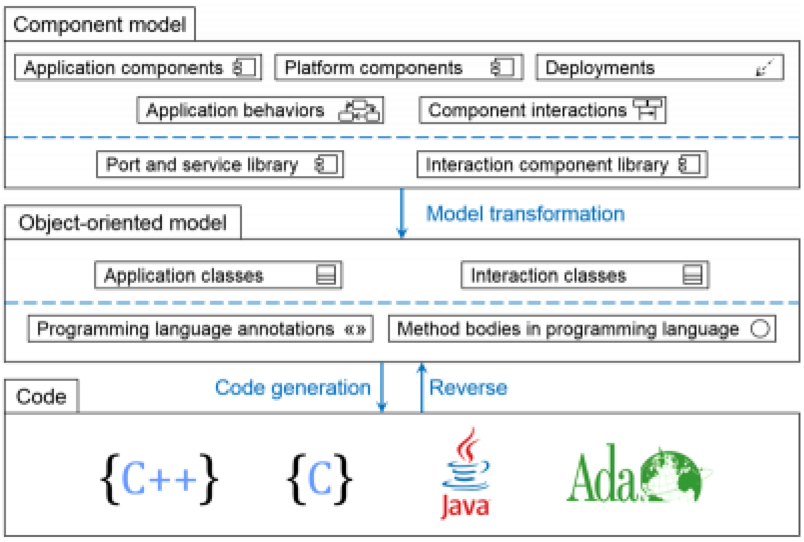
RobotML is a DSL specifically oriented to modeling and design of mobile manipulation robotic systems. **RobotML** conforms to RobMoSys's foundational principles of separation of roles and concerns. It provides several **view points**, including (but not limited to) those for the definition of State Machines, Hardware and Software components , Controllers and Environment. RobotML domain models allow for the representation of the system's architecture, control and communication aspects and span across all 5C concerns of Computation, Coordination, Coordination, Configuration and Composition.



Further modeling views are provided by additional components of Papyrus4Robotics. For example, the **performance view** is featured by Papyrus Architect, a Papyrus4Robotics module dedicated to explore quality attributes of architectures, with a focus on timing properties in real-time applications of embedded (robotic) systems. It leverages the MARTE (Modeling and Analysis of Real-Time Embedded systems) DSL for the specification of system architecture (functional/physical) and of timing properties. The performance view addresses the problem of evaluating the performance of candidate architectures with respect to attributes like hardware resource utilization.


Reaction constraint of 200ms on the chain
Specification of operating modes and timing requirements
Performance view: resource consumption (network buses, CPUs)

In addition to DSLs and modeling, Papyrus4Robotics also features code-generation capabilities. Papyrus Designer [https://wiki.eclipse.org/Papyrus_Software_Designer] supports code generation from models of SW including embedded and real-time and DDS-based distributed systems as potential targets. In Designer, the generation starts from a model that includes the definition of software components, hardware nodes and deployment information. The latter consists of a definition of the instances of components and nodes and an allocation between these. Code generation is done by a sequence of transformations steps. The model transformation takes care of some platform specific aspects (e.g. communication mechanisms or thread allocation), based on non-functional properties.



RobotML includes generators that transform RobotML-compliant models into code for robotic middlewares (e.g., Orocos-RTT [http://www.orocos.org/rtt]) or simulators (e.g., MORSE [https://www.openrobots.org/wiki/morse/]).

## Conformance to the RobMoSys structures

Some modeling concepts in Papyrus4Robotics are already aligned with the RobMoSys definitions. However, further refinement and alignment of meta-models is in process and scheduled to be released and productively used by the end of 2017.

## Separation of Levels and tool coverage

Papyrus4Robotics provides implementations for the individual levels listed in Separation of Levels and Separation of Concerns

| **Level** | **Corresponding DSL or Tool in Papyrus4Robotics** |
|---|---|
| Task Plot | RobotML State Machine |
| Skill | RobotML Inteface |
| Service | RobotML operation (defined in the Skill interface)<br>Software Component representation in Papyrus Designer |
| Function | C++ library (e.g., libOpenRave, etc.) |
| Execution Container | Task and resource representation in Papyrus Designer |
| OS/Middleware | DRM::SRM in UML MARTE |
| Hardware | DRM::HRM in UML MARTE, RobotML's sensors and actuators |

## Platform workbenches in the context of RobMoSys

One major project's focus is on models, software and tools that are generically useful for all possible robotic systems and applications. This includes systems and applications that can, e.g., pass certification, monitor their resource usage at runtime, or form systems-of-systems with just a reconfiguration of the available models.

Building such systems and applications require multi-disciplinary competences (beyond robotics) and sets of platform tools that support best-practices established in near and mature engineering-centric domains, such as automotive or aerospace.

Possible modeling workbenches enabled by the RobMoSys's software baselines are for example SmartMDSD Toolchain, the Papyrus4Robotics set of modeling tools. There are many more existing modeling tools that can be made conformal to the RobMoSys's baseline. In a robotics ecosystem multiple users provide models by using these workbenches and these models are interfaced over the RobMoSys's baseline.

Some workbenches allow for many different kinds of analysis that are strongly related to good practices to employ during the development process—as recommended by **experts** in the complex and critical systems design domain (read Annex 1 of D5.1 to find out more). This includes (and is not limited to):

- verification and co-simulation activities (e.g., based on the FMI 2.0 standard) during early stage of design, thanks to the definition of a model of computation (MoC) on system level;
- handling safety and security aspects as soon as possible and not as an afterthought;
- checking whether the amount of reserved resources (hardware/software) is adequate to meet given performance criteria (e.g., respect of time constraints on end-to-end latencies)

It is unrewarding to define one single modeling workbench that covers all aspects of design, analysis and synthesis (i.e. code-generation). Instead, because platform tools conform to the RobMoSys structures, models can be exchanged from one modleing workbench to another to cover all the design needs of the ecosystem users at all the phases of development.

## Resources

- Installation procedure
  - Papyrus [https://eclipse.org/papyrus/]
  - Papyrus RobotML [https://eclipse.org/papyrus/components/robotml/1.2.0/]
  - Papyrus Software Designer [https://wiki.eclipse.org/Papyrus/Designer/getting-started]
- Documentation and tutorials
  - Papyrus Documentation [http://www.eclipse.org/papyrus/documentation.html]
  - Papyrus RobotML Documentation [https://eclipse.org/papyrus/components/robotml/1.2.0/]
  - Papyrus Software Designer User Guide[https://wiki.eclipse.org/index.php?title=Papyrus_Software_Designer&redirect=no]
- Videos
  - Model driven safety assessment for robotics[https://www.youtube.com/watch?v=CnklgQ7tWns]
  - Modeling and safety assessment for Nao[https://www.youtube.com/watch?v=-k1xWJr4wg0]
  - More videos on Papyrus Companions [https://www.youtube.com/channel/UCxyPoBlZc_rKLS7_K2dtwYA]
- Selected publications
  - Selma Kchir, Saadia Dhouib, Jérémie Tatibouet, Baptiste Gradoussoff, Max Da Silva Simoes, RobotML for industrial robots: Design and simulation of manipulation scenarios. ETFA 2016: 1-8
  - Nataliya Yakymets, S. Dhouib, Hadi Jaber, Agnes Lanusse, Model-driven safety assessment of robotic systems. IROS 2013: 1137-1142
  - Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, Mikal Ziane, RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications. SIMPAR 2012: 149-160

# Getting Started With Papyrus4Robotics

## Installation

Papyrus4Robotics is distributed as a self-contained Eclipse RCP [https://wiki.eclipse.org/Rich_Client_Platform].

The RCP for your OS (64bits) is available from the links below:

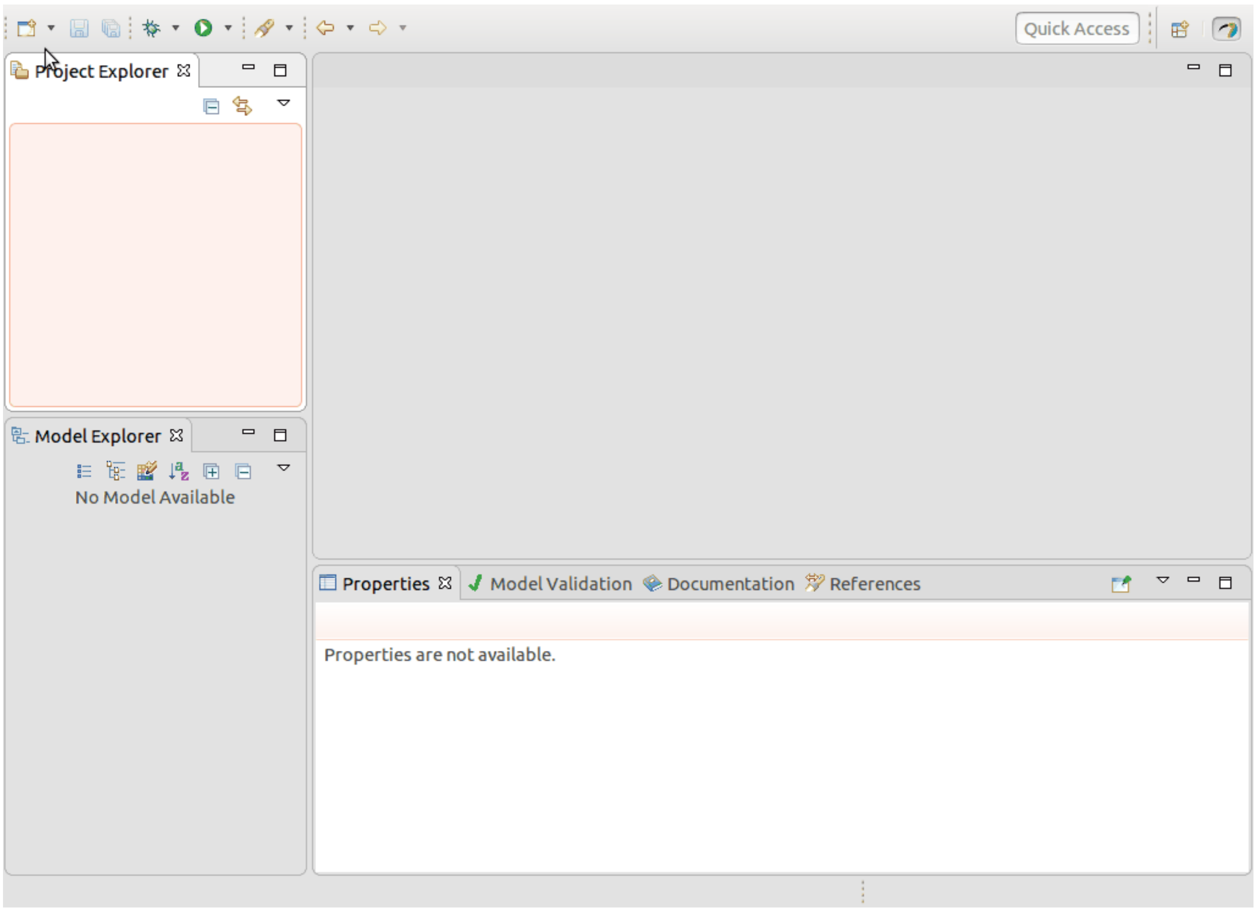
- Windows [ftp://ftp.cea.fr/pub/lise/robmosys/org.eclipse.papyrus.robmosys.product-win32.win32.x86_64.zip]
- Linux [ftp://ftp.cea.fr/pub/lise/robmosys/org.eclipse.papyrus.robmosys.product-linux.gtk.x86_64.zip]
- MacOS [ftp://ftp.cea.fr/pub/lise/robmosys/org.eclipse.papyrus.robmosys.product-macosx.cocoa.x86_64.tar.gz]

To install it, just unpack the RCP archive in a directory of choice.

To run it, make sure that you have a Java 8 or newer JRE/JDK. Then just launch the `papyrus-robmosys` executable.

## Running an Example

Papyrus4Robotics comes with an installed example to get you started. To run the example, just click `New→Example` and accept the default propositions in the wizard.



No Model Available

Properties are not available.

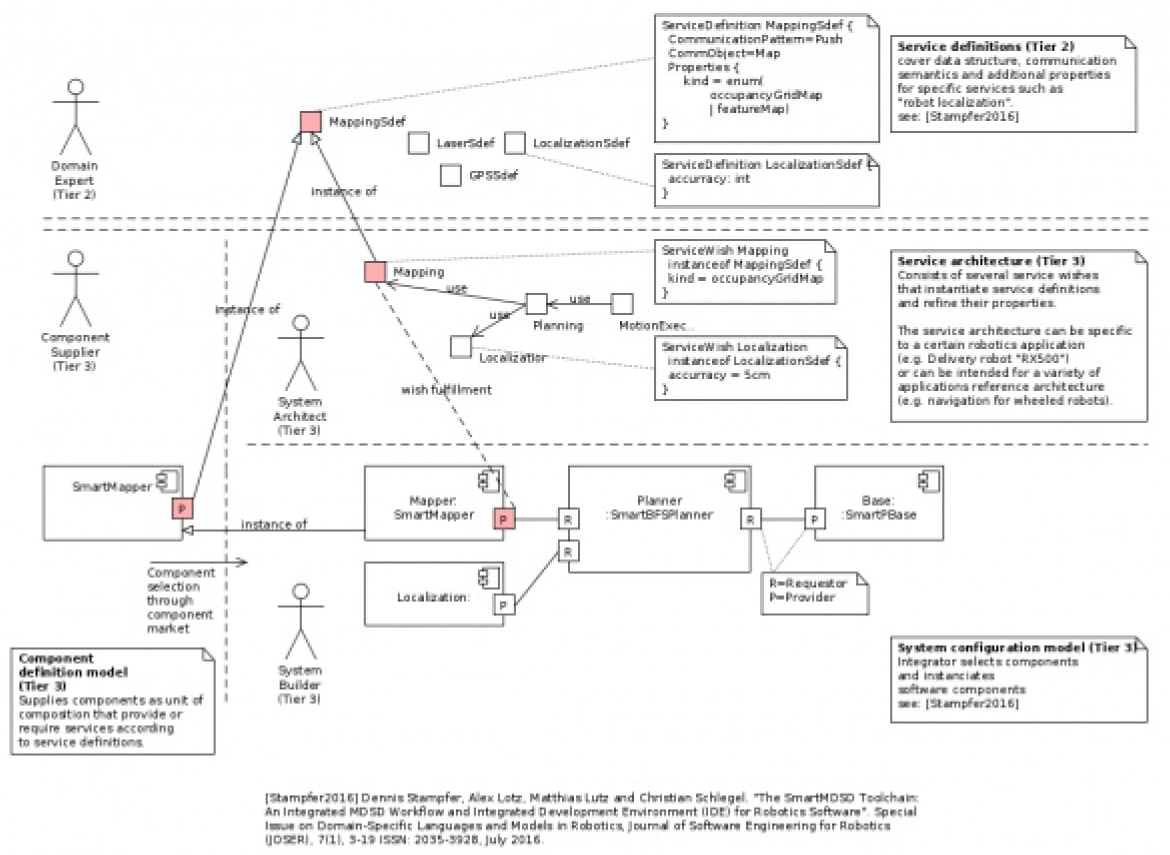
# The Example Explained

## Introduction

This example shows how Papyrus4Robotics supports the service-based approach for the composition of software components.

Composition in an Ecosystem organized in tiersis the approach adopted by RobMoSys to system integration. Next sections discuss how tier 2 and tier 3 participants use Papyrus4Robotics to model a simple service-based composition of a mapper component and a planner component.

The illustration below corresponds to the role descriptions, as taken from the RobMoSys wiki.

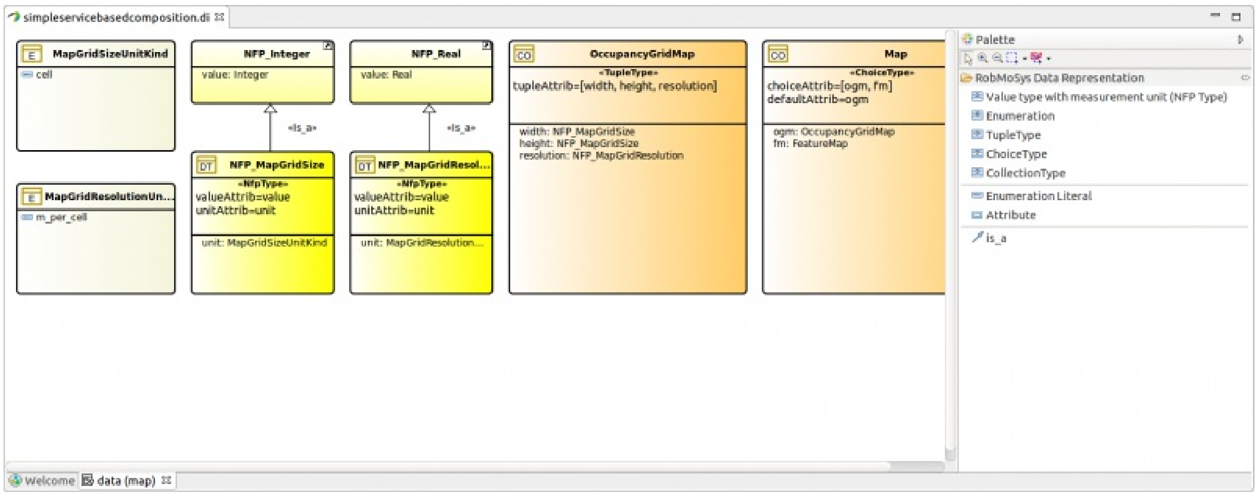


Domain Expert (Tier 2)
MappingSdef
LaserSdef
LocalizationSdef
GPSSdef
instance of
ServiceDefinition MappingSdef { CommunicationPattern=Push CommObject=Map Properties { kind = enum( occupancyGridMap | featureMap) }
Service definitions (Tier 2) cover data structure, communication semantics and additional properties for specific services such as "robot localization". see: [Stampfer2016]
ServiceDefinition LocalizationSdef { accurracy: int }
Component Supplier (Tier 3)
Mapping
use
Planning
MotionExec...
Localization
instance of
System Architect (Tier 3)
wish fulfillment
ServiceWish Mapping instanceof MappingSdef { kind = occupancyGridMap }
ServiceWish Localization instanceof LocalizationSdef { accurracy = 5cm }
Service architecture (Tier 3) Consists of several service wishes that instantiate service definitions and refine their properties.
The service architecture can be specific to a certain robotics application (e.g. Delivery robot "RX500") or can be intended for a variety of applications reference architecture (e.g. navigation for wheeled robots).
SmartMapper
P
instance of
Mapper: SmartMapper
Planner :SmartBFSPlanner
Base: :SmartPBase
R
Localization:
R=Requestor P=Provider
Component selection through component market
System Builder (Tier 3)
Component definition model (Tier 3) Supplies components as unit of composition that provide or require services according to service definitions.
System configuration model (Tier 3) Integrator selects components and instanciates software components see: [Stampfer2016]
[Stampfer2016] Dennis Stampfer, Alex Lotz, Matthias Lutz and Christian Schlegel. "The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software". Special Issue on Domain-Specific Languages and Models in Robotics, Journal of Software Engineering for Robotics (JOSER), 7(1), 3-19 ISSN: 2035-3928, July 2016.

## Domain Expert (Tier 2)

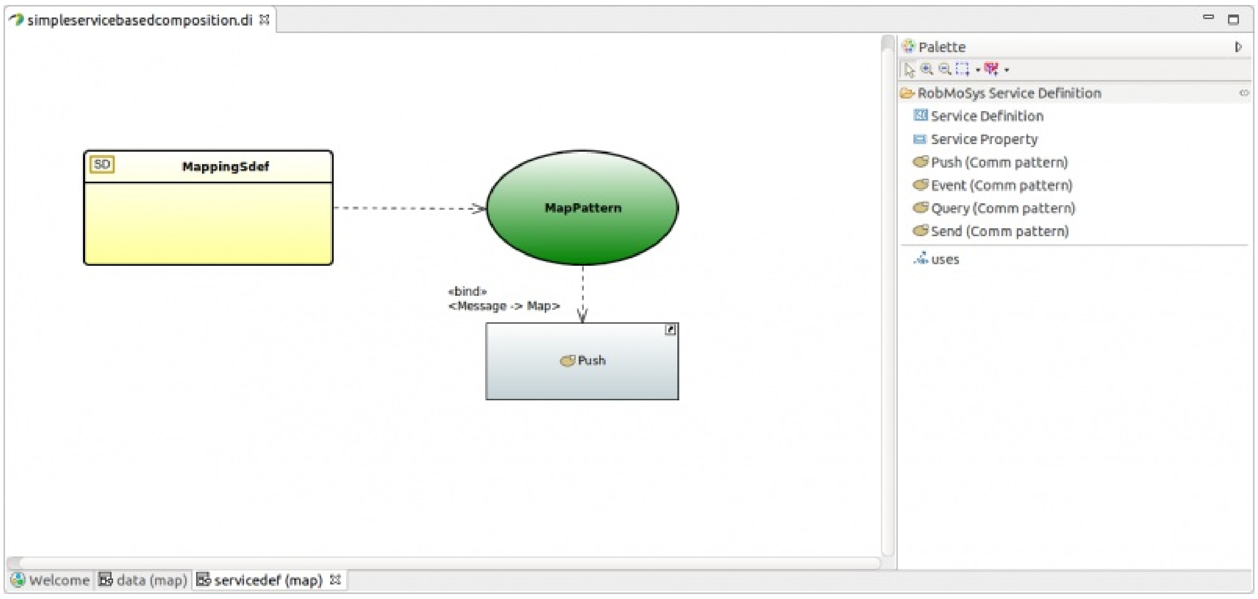
Tier 2 are robotics experts who define a *complete* characterization of services in robotics domains, e.g., mapping, planning, localization, etc. Tier 2 structures each robotics domain by creating domain-models that cover a number of aspects, including data structures and communication semantics. Service designers are the domain experts on Tier 2 that design individual service definitions for use by Tier 3.

The picture below shows a portion of data structures defined for the mapping domain in this example.



Papyrus4Robotics leverages concepts from OMG's MARTE [https://www.omg.org/spec/MARTE/] NFP and VSL profiles to comply with RobMoSys' specifications on digital data representation. Built-in type definitions can be imported from the `BasicNFP_Types` MARTE library and specialized for a specific domain by using a dedicated palette (right side of the picture). Leveraging on MARTE, **Papyrus4Robotics supports physical units descriptions** to formally define unambiguous semantics of units of measurements in data types.

Once the communicated data structures (Communication Objects, identified with the `CO` icon on the top left corner) are defined, the communication pattern usage can be formalized. The next picture shows the model that describes the `MappingSdef` service. In this example,`MappingSdef uses` the `Push` pattern and selects the`Map` data type as communicated data structure.

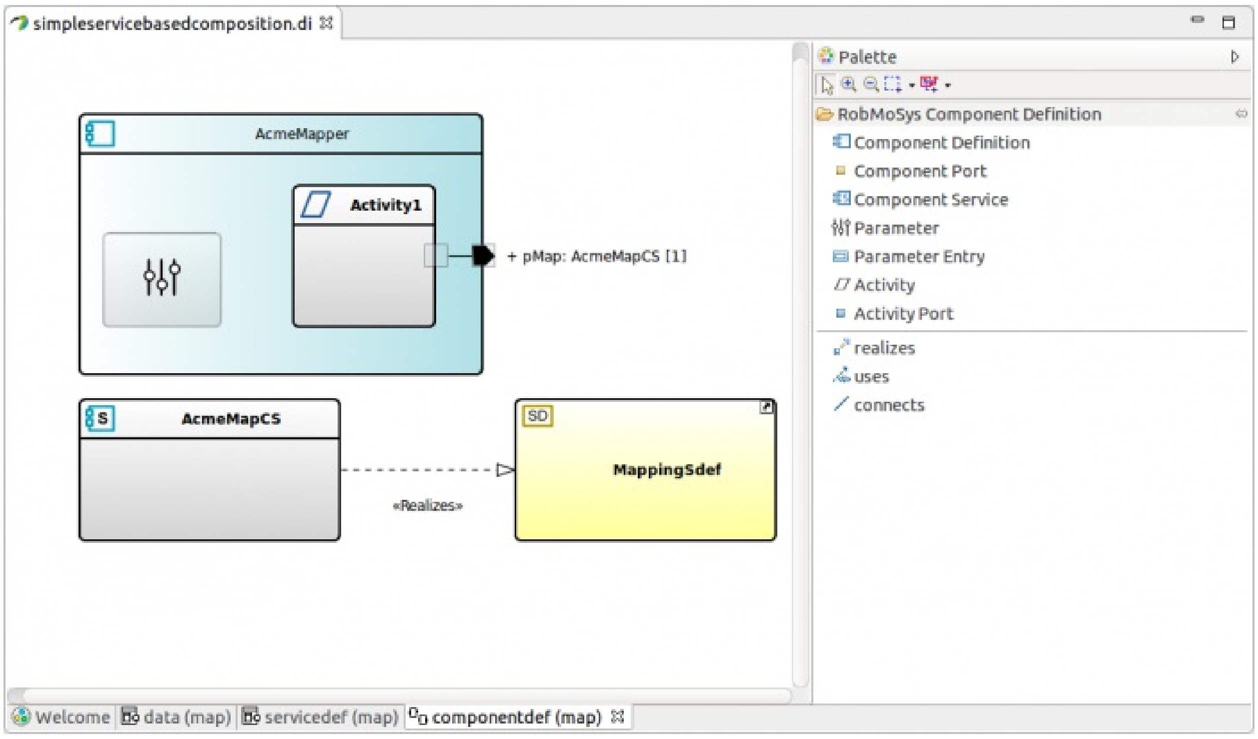


## Component Suppliers (Tier 3)

Component suppliers at Tier 3 provide models of software component definitions.

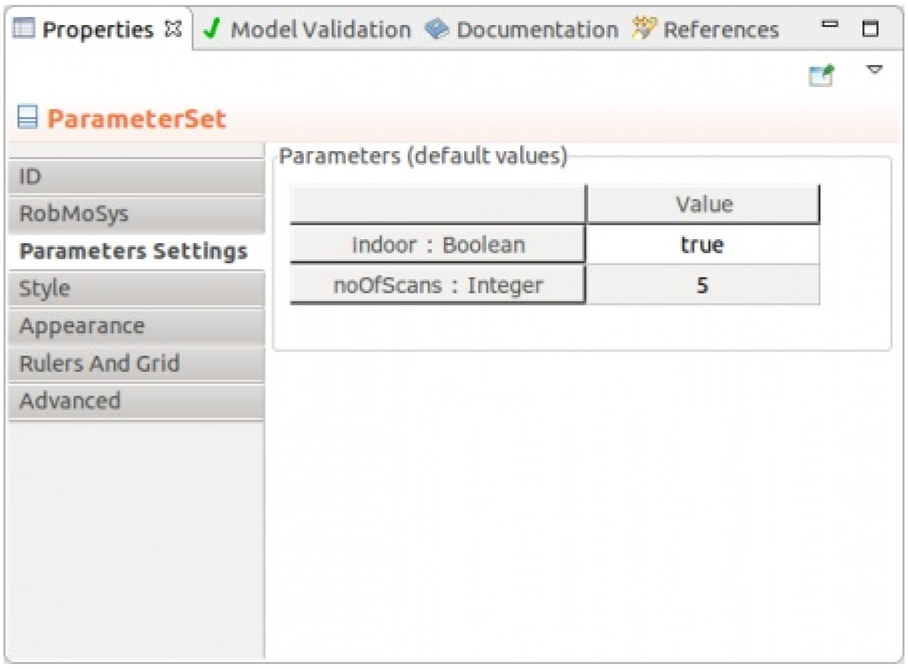
### AcmeMapper

The model below shows a mapper component developed by a company called AcmeCorp (hence `AcmeMapper`), which provides a fully compliant implementation of the`MappingSdef` service definition above.



The example focuses on the modeling of a single component port (`pMap`) providing the mapping service. Aligned with the standard UML rules of interface realization, this is achieved by assigning the port a `ComponentService` item as a type (`AcmeMapCS`) that `realizes` `MappingSdef`.

`AcmeMapper` contains one `Parameter` structure, that represents a set of parameters that make the component configurable for reuse in different scenarios by the system builder or even at run-time. The `Parameter` structure content is visualized in the model editor by selecting the `Paramter` icon and the `Parameters Settings` tab in the property view (see below).



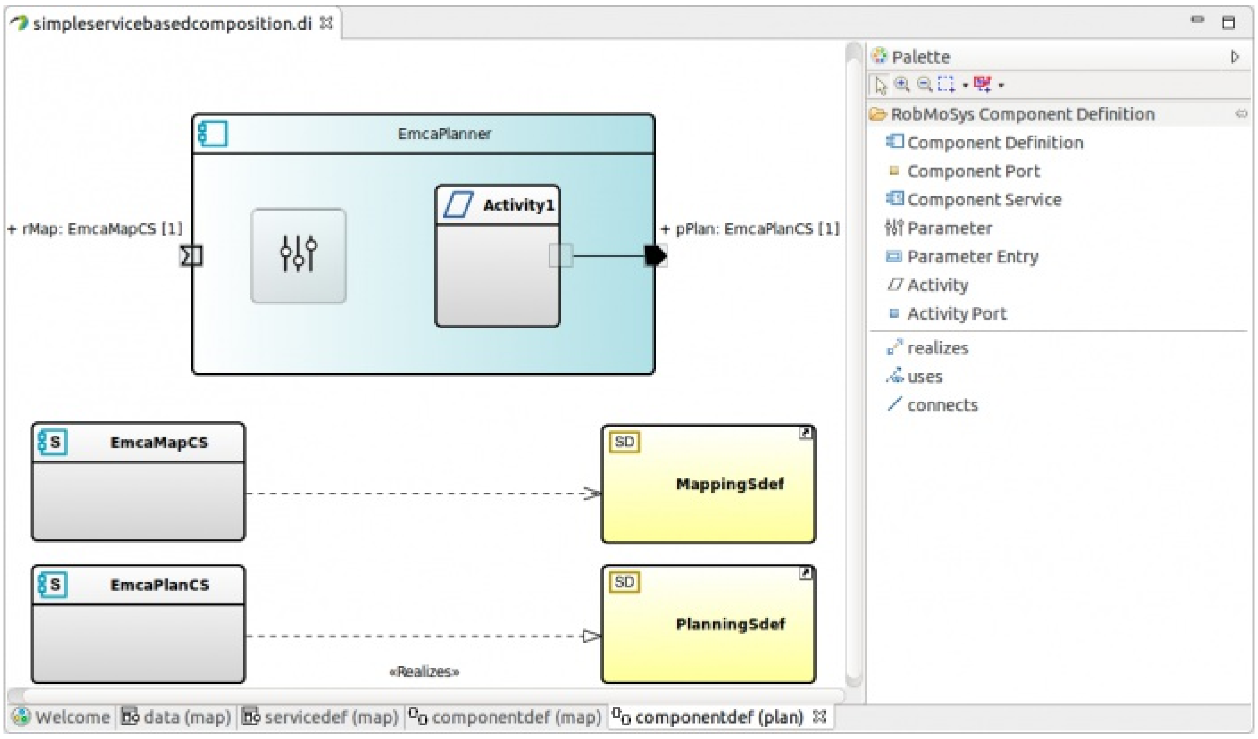
| | Value |
|---|---|
| indoor : Boolean | true |
| noOfScans : Integer | 5 |

In this simple example, `AcmeMapper` has 2 configurable parameters with built-in types. However composite value specifications (collection, tuple, choice, etc.) can be specified as well, using the MARTE VSL syntax.

`AcmeMapper` defines one activity (it could define more), which is a OS-agnostic representation of a thread. Activities provide wrappers for functions (algorithms). Activities do have set of parameters for configuration (e.g., interarrival range, that is max and min activation frequencies). Similarly to component parameters, activity paramters can be viewed and set through the `Parameters Settings` tab in the property view.

**EmcaPlanner**

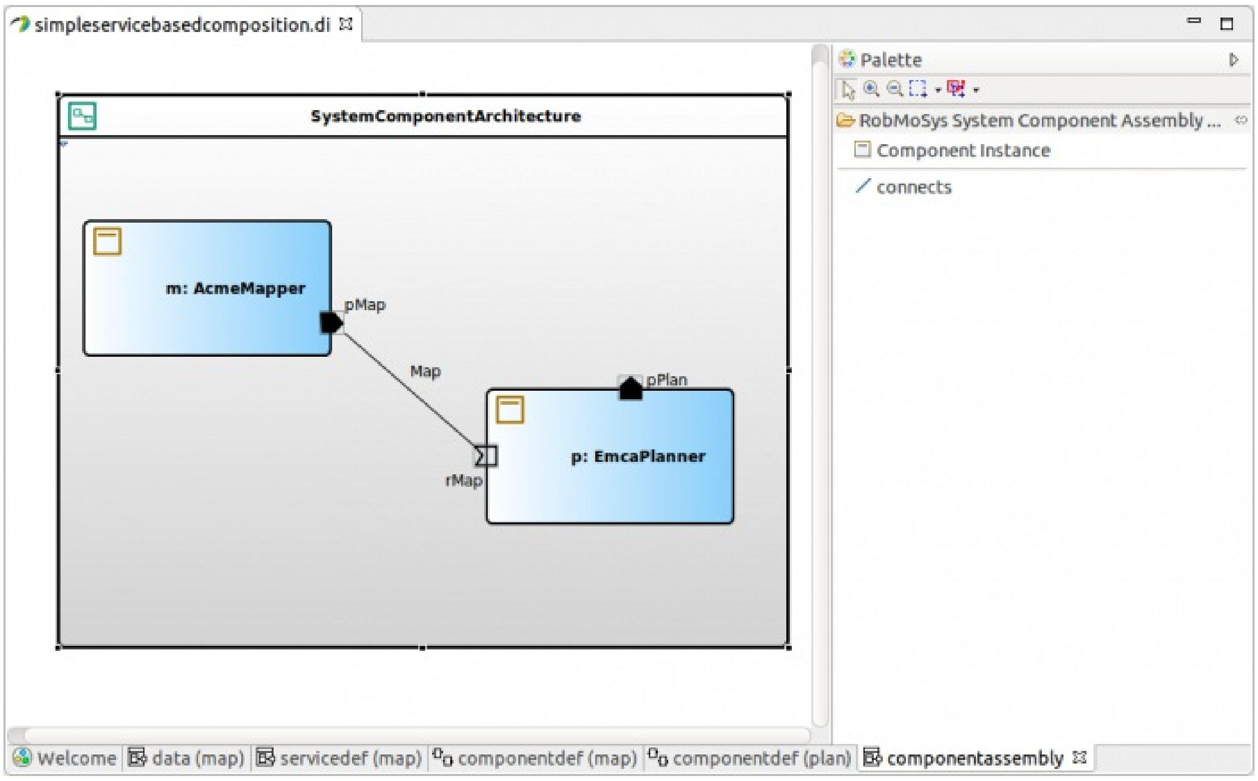
The model below shows a planner component developed by a company called EmcaInc (hence `EmcaPlanner`).



The example focuses on the definition of two component ports. `pPlan` **provides** a implementation of `PlanningSdef` (a planning service definition model not discussed in this document).`rMap` **requires** a fully compliant implementation of `MappingSdef`. To model the service requirement, in agreement with the standard UML rules, the `usage` item is used to create a dependency between the`EmcaMapCS` and `MappingSdef`.

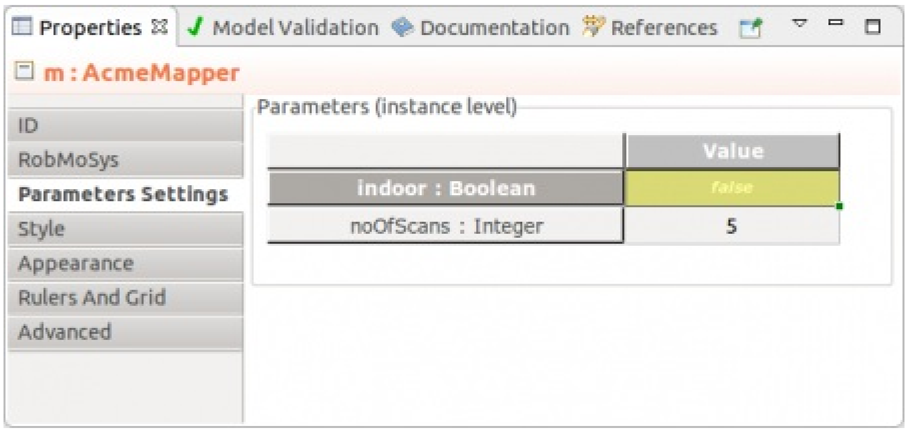
## System Builder (Tier 3)

System builders instantiate component definitions to provide a platform-independent specification of a software system. The model below shows the instantiation and connection of one instance of `AcmeMapper` and one instance of `EmcaPlanner`.



It is now assumed that the mapper component instance must work outdoor. The default configuration of `AcmeMapper` component definition was indoor, so the component instance `m` must be re-configured by the system builder.

For a component instance the parameter set is accessible by clicking on the instance itself and selecting the `Parameters Settings` tab in the property view. The next picture shows the value of `indoor` parameter is set to `false`. Yellow highlighting visually enforces the message that the parameter value is now different from the default one.



Parameters (instance level)

| | Value |
|---|---|
| indoor : Boolean | false |
| noOfScans : Integer | 5 |

## Conclusions

This example shows a structural model in the context of composition of software components. It shows how different tiers contribute models to achieve composition of software components, using service-definitions as central architectural element for it. Then it focuses on one instance of the mapper component and shows a simple reconfiguration of one of its parameters.

# Do It Yourself

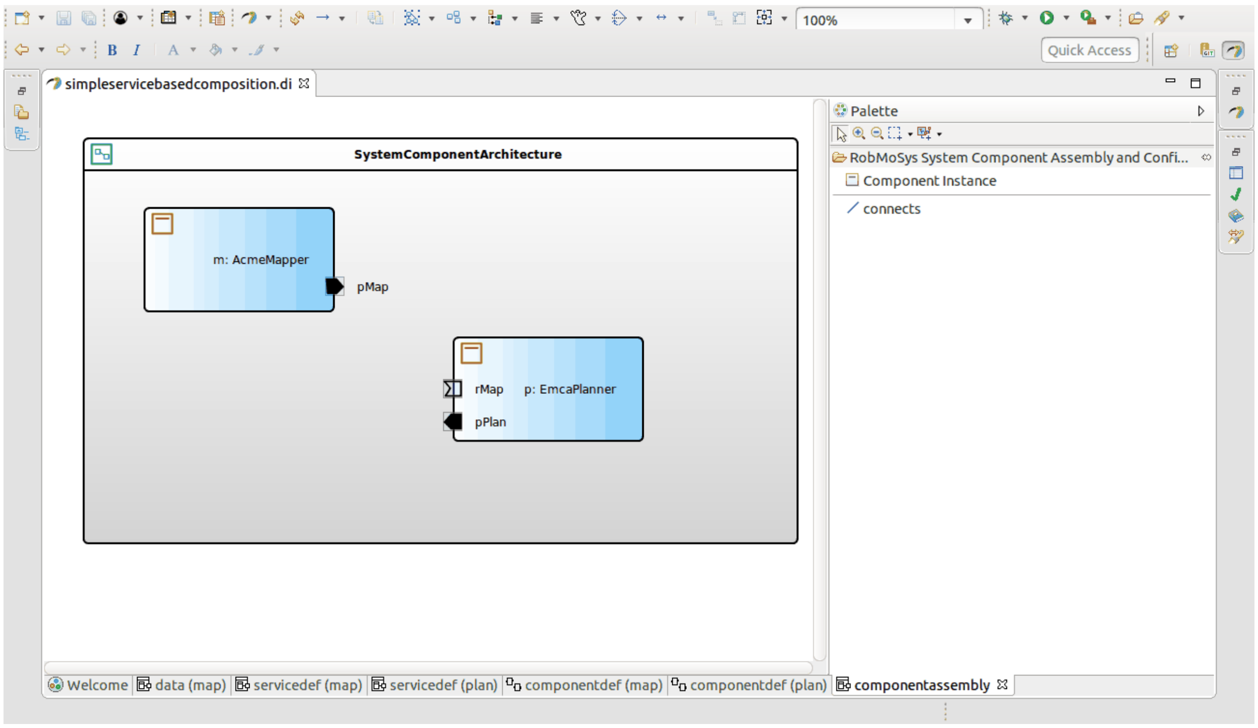
This section is in progress. More content will be added shortly.

### Connect ComponentInstance Items

Connections between component instances can be drawn by using the `connects` item from the system component architecture palette.

First, select `connects` from the palette. Then point and click the source and target component ports to be connected. Note that the connection is only possible if

- both source and target elements are component ports
- both ports provide/require compatible services. In other words, for 2 connected (component) ports, the type represented by a component service must be *instance-of* the same service definition.



# SmartSoft Components

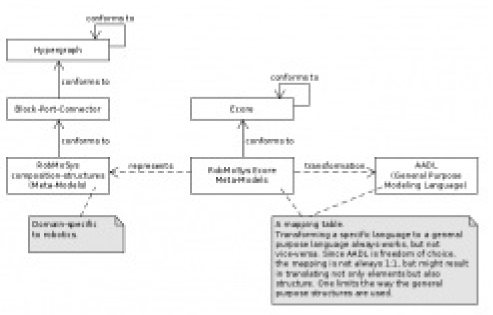
A collection of SmartSoft components is readily available under Open Source Licenses. They have been developed using the SmartMDSD Toolchain and are available for immediate reuse.

- For use with the SmartMDSD Toolchain v2: List of available components [http://www.servicerobotik-ulm.de/drupal/doxygen/components_commrep/group__componentGroup.html]
- For use with the SmartMDSD Toolchain v3: List of available components [https://github.com/Servicerobotics-Ulm/ComponentRepository]

baseline:components:smartsoft · Last modified: 2018/06/29 17:54
http://www.robmosys.eu/wiki-sn-02/baseline:components:smartsoft

# Other Approaches in the RobMoSys Context

RobMoSys follows a reuse-oriented approach. This means that reinvention should be kept to a minimum and existing approaches should be used wherever possible. The following list provides some common approaches that are considered relevant within the RobMoSys context.


conforms to
Hypergraph
conforms to
Block-Port-Connector
conforms to
RobMoSys composition-structures (Meta-Models)
Domain-specific to robotics
represents
conforms to
Ecore
conforms to
RobMoSys Ecore Meta-Models
transformation
AADL (General Purpose Modeling Language)
A mapping table. Transforming a specific language to a general purpose language always works, but not vice-versa. Since AADL is freedom of choice, the mapping is not always 1:1, but might result in translating not only elements but also structure. One limits the way the general purpose structures are used.

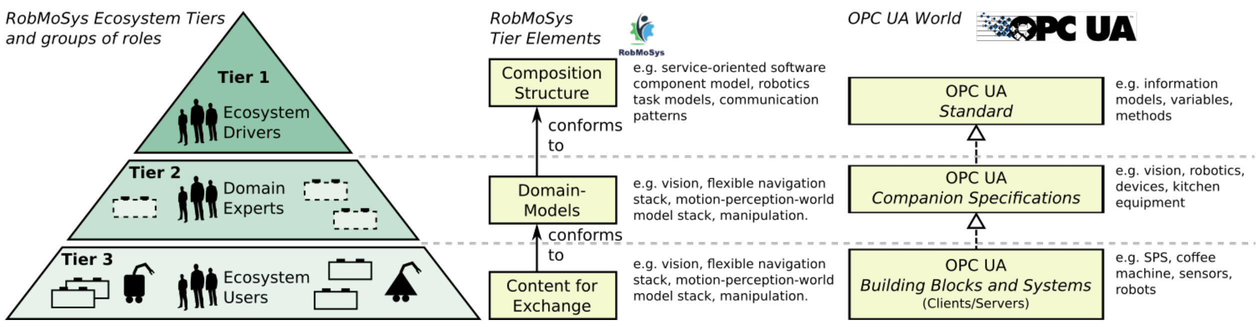
- General Purpose Modeling Languages (SysML/UML) and Dynamic-Realtime-Embedded (DRE) domains (AADL, MARTE, etc.)
- Robotics Approaches (ROS, YARP, RTC, etc.)
- Middlewares (DDS)
- Industry 4.0 domain: OPC UA

# OPC Unified Architecture (OPC UA)

The organization of an ecosystem in three tiers can also be found in other domains. For example, a significant part of the industry 4.0 domain is shifting towards the OPC Unified Architecture (OPC UA) [https://opcfoundation.org/]. OPC UA is a standard for machine-to-machine communication comprising communication infrastructure and information models for semantic data exchange. OPC UA is standardizing connectivity of industrial devices and enables the interoperability among products of different vendors. It does not yet address the next level of interoperability which we call “composability”.

The OPC UA ecosystem is in its structures exactly conformant to the explicated tiers of the RobMoSys ecosystem approach. The OPC foundation is the driver in tier 1, the companion specifications belong to tier 2 and finally there are the users at tier 3. The strong point about OPC UA is that it is driven by industry in a joint effort and that they successfully manage the ramp up of an ecosystem along these tiers.

A direct comparison of the RobMoSys Ecosystem with OPC UA is given in the figure below.


RobMoSys Ecosystem Tiers and groups of roles
Tier 1
Ecosystem Drivers
Tier 2
Domain Experts
Tier 3
Ecosystem Users
RobMoSys Tier Elements
Composition Structure
e.g. service-oriented software component model, robotics task models, communication patterns
conforms to
Domain-Models
e.g. vision, flexible navigation stack, motion-perception-world model stack, manipulation.
conforms to
Content for Exchange
e.g. vision, flexible navigation stack, motion-perception-world model stack, manipulation.
OPC UA World
OPC UA Standard
e.g. information models, variables, methods
OPC UA Companion Specifications
e.g. vision, robotics, devices, kitchen equipment
OPC UA Building Blocks and Systems (Clients/Servers)
e.g. SPS, coffee machine, sensors, robots

As prominent example for domain models (companion specifications), VDMA is working on companion specifications for vision and robotics. Companion specifications sometimes contain additional concepts that have evolved in a particular domain, but that are generally applicable. For example, the companion specification for vision foresees a generic state automaton for components with component-specific sub-states —a very similar concept to the RobMoSys component life-cycle and communication pattern "state pattern". In the long-run, they may be adopted by OPC UA itself, thus move from Tier 2 to Tier 3. This movement of structures describes the evolvement of an ecosystem and also has been identified for RobMoSys (see wiki page on „Tier 1 in detail“). OPC UA is actively postulating the creation of companion specifications by providing support and guidance.

OPC UA eases device integration thanks to an overall methodology (Tier 1) and domain-specific standards (composition Tier 2). Device suppliers now can adopt the Tier 2 standards and gain compatibility with users that expect these standards. OPC UA, however, does not specifically aim for composition and is, in fact, less suitable for composition of software components. It misses adequate abstractions and concepts (e.g. such as RobMoSys communication patterns). However, composability starts being addressed in OPC UA as it can be observed in recent developments that are on the way to introduce the concept of skills.

OPC UA can also be used as an underlying communication infrastructure below the RobMoSys structures. In the context of composition, the challenge with OPC UA is to introduce additional structures that enable composition. This is done by, for example, the RobMoSys communication patterns. This is where the German national BMWi/PAiCE Project “Service Robot Network” (SeRoNet) is adopting parts of the RobMoSys composition structures and provides a mapping to OPC UA. Thereby, SeRoNet can fully benefit from composition as introduced by RobMoSys but also manages the seamless integration with the traditional OPC

UA world, for example to use OPC UA powered devices.

In general, the industry 4.0 world based on OPC UA has a fully conformant way of thinking with respect to the overall RobMoSys world. Thus, there is a very good chance to communicate the RobMoSys contributions to that domain and thereby link the robotics domain with the automation domain. While OPC UA and its companion specifications at the moment are at the level of integration with a roadmap towards the next levels which we call composability, RobMoSys already now proposes solutions to address composability. Due to the very same ecosystem structures, there is a very good chance to enable adoption of the RobMoSys outcomes within the industry driven OPC UA automation domain. For RobMoSys, the strength of OPC UA is that it provides standardized and uniform ways to access all kinds of devices like sensors, actuators, machineries, cloud services etc. RobMoSys puts its focus on the software composition for most complex sensori-motor systems which then can get networked with industry 4.0 environments via OPC UA.

## See also

- Ecosystem Organization
- Tier 1 in Detail
- OPC UA Vision Companion Specification https://opcfoundation.org/markets-collaboration/vdma-machine-vision [https://opcfoundation.org/markets-collaboration/vdma-machine-vision]
- OPC UA Robotics Companion Specification: https://opcfoundation.org/markets-collaboration/vdma-robotics [https://opcfoundation.org/markets-collaboration/vdma-robotics]
- BMWi/PAiCE Project “Service Robot Network”: https://www.seronet-projekt.de [https://www.seronet-projekt.de]
- Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. OPC Unified Architecture. 1st ed. Springer-Verlag Berlin Heidelberg, 2009. ISBN: 978-3-540-68898-3. DOI: 10.1007/978-3-540-68899-0.

## Acknowledgement

This document contains material from:

- [Stampfer2018] Dennis Stampfer, "Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics". Dissertation, Technische Universität München, München, Germany, 2018. [http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2], especially Section “2.5.3 Industrial Automation and Industry 4.0”

---

other_approaches:opc-ua · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/other_approaches:opc-ua

# General Purpose Modeling Languages and Dynamic-Realtime-Embedded domains

SysML, SoaML, AADL, MARTE and others are flexible general purpose modeling approaches for systems. They favor freedom of choice. While they often provide different modeling views, these views are not connected such that overall system consistency can be ensured throughout all potential development phases. This hinders separation of roles that is required for successful system composition and therefore is in contrast with the overall needs for modeling in RobMoSys.

The focus of RobMoSys is on composability and consistency of the different views such that the different roles contribute in a consistent and composable way to the system under specification and development. This requires more elaborate structures to connect the different views in a consistent way. This can be achieved via superordinated meta-model structures and via model-to-model transformations.

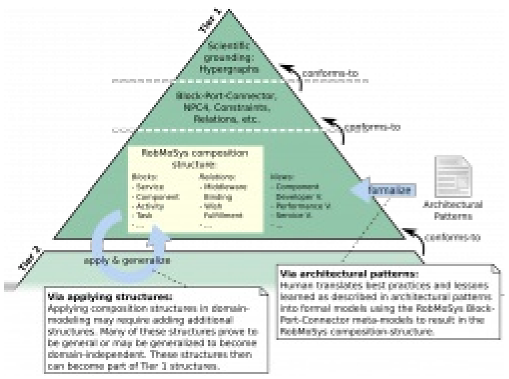
Of course, the structures of RobMoSys will be inspired by, for example, the above approaches wherever appropriate. The RobMoSys structures might enable linking the different modeling views of the mentioned modeling approaches.

For example, AADL requires more abstract, yet consistent, modeling views on top, while other approaches such as SysML might be subprofiled, thus providing more detailed, yet again consistent, robotic-specific views underneath. Many of the (meta-model) structures and abstractions in RobMoSys focus on transformations (and exchange of knowledge) between well known and widely accepted modeling views.

Within the context of UML the term *“semantic variation point”* has been coined to express the purposeful semantic ambiguity for certain UML elements. Because UML is a general purpose modeling language, this semantic ambiguity makes sense and can be narrowed within the derived domain-specific models using e.g. the UML profile mechanism. Moreover, even the domain-specific models can still expose some semantic variability that is closed within concrete realizations (e.g. through code generation or reference implementations). In this sense, RobMoSys as well offers different levels of abstraction for modeling where the higher levels (such as e.g. the block-port-connector) are more general purpose (leaving open some semantic variability) and lower (i.e. domain-specific) abstraction levels (such as e.g. the RobMoSys composition structures) that narrow this semantic variability.

other_approaches:modeling_languages · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/other_approaches:modeling_languages

# Tier 1: Modeling Foundations

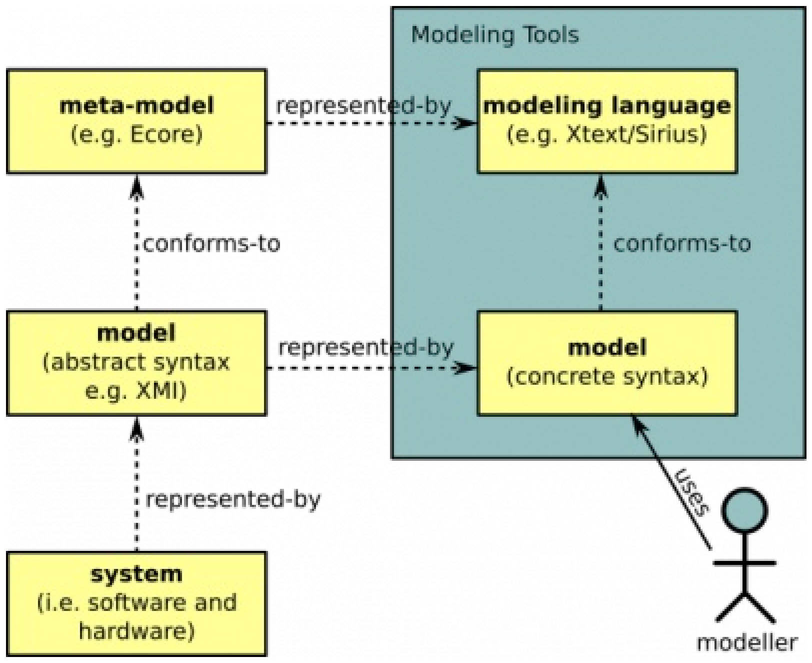

Tier 1
Scientific grounding: Hypergraphs
conforms-to
Block-Port-Connector, NPC4, Constraints, Relations, etc.
conforms-to
RobMoSys composition structure:
Blocks:
- Service
- Component
- Activity
- Task
- ...
Relations:
- Middleware Binding
- Wish Fulfillment
- ...
Views:
- Component Developer V.
- Performance V.
- Service V.
- ...
formalize
Architectural Patterns
conforms-to
Tier 2
apply & generalize
Via architectural patterns:
Human translates best practices and lessons learned as described in architectural patterns into formal models using the RobMoSys Block-Port-Connector meta-models to result in the RobMoSys composition-structure.
Via applying structures:
Applying composition structures in domain-modeling may require adding additional structures. Many of these structures prove to be general or may be generalized to become domain-independent. These structures then can become part of Tier 1 structures.

RobMoSys considers Model-Driven Engineering (MDE) as the main technology to realize the so far independent RobMoSys structures and to implement model-driven tooling. The wiki pages below collect some basic modeling principles related to realizing the RobMoSys structures.

- Roadmap of MetaModeling
- Modeling Principles
    - Modeling Twin
    - Realization Alternatives
- Tier 1 Structure
    - Scientific Grounding: Hypergraph and Entity-Relation model
    - Block-Port-Connector
    - RobMoSys Composition Structures (and metamodels)
    - Views which are used by roles

# Basic Modeling Principles

There is a subtle relationship between the (meta-)models, the actual modeling languages and the concrete models. This relationship is depicted in the figure below.



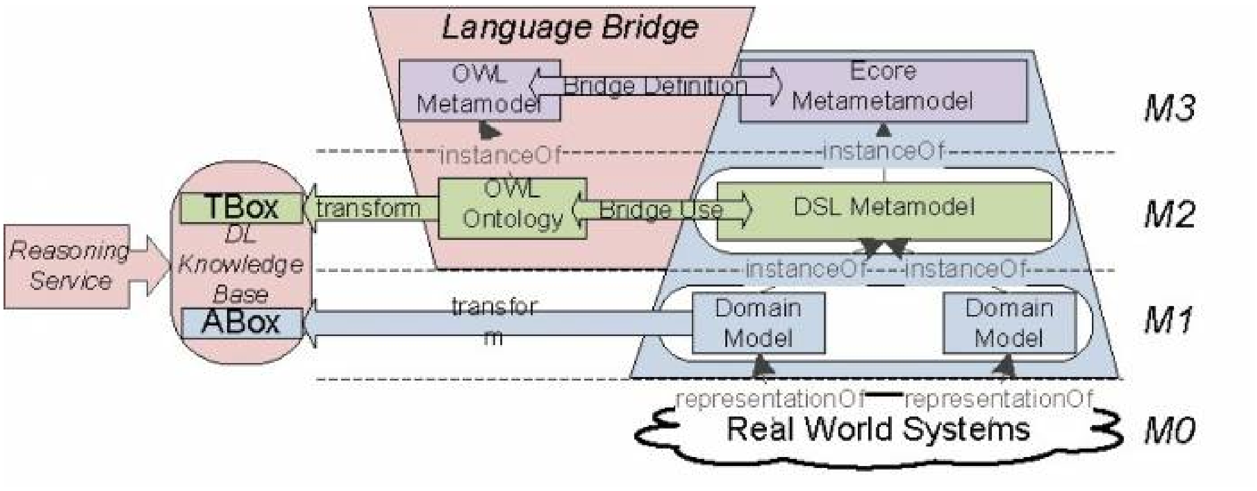
A modeller (i.e. a modeling-tool user who creates models) always works with a concrete syntax. This syntax can be textual, graphical, tabular or any combination thereof. The concrete syntax (sometimes also called notation) is defined by (i.e. it conforms to) the **modeling language**. The concrete syntax of a modeling language is independent of the abstract syntax of an actual **meta-model**. However, the structure of the **modeling language** must adhere to the structures defined in a**meta-model**. In most cases, it makes sense to first specify the meta-model, then to generate a modeling language out of the meta-model and then to adjust only the syntax of the modeling language (without affecting the structure). A model created by the modeller is typically only a representation for the in-memory model that uses the abstract syntax. The abstract syntax is also used to serialize the models in order to make them persistent.

Finally, the model itself is an abstract representation of the actual system (which can be either software, hardware or any combination thereof). Often, it makes sense to package the model with the related software/hardware parts and to ship them together as a so called modeling twin.

Are you new to model-driven engineering? Find introduction literature in the FAQ.

## Ecore-OWL language-bridge

There is a relation between meta-models and ontologies that can be bridged as described here [http://twouse.blogspot.de/2010/08/owl-ecore-language-bridges.html].



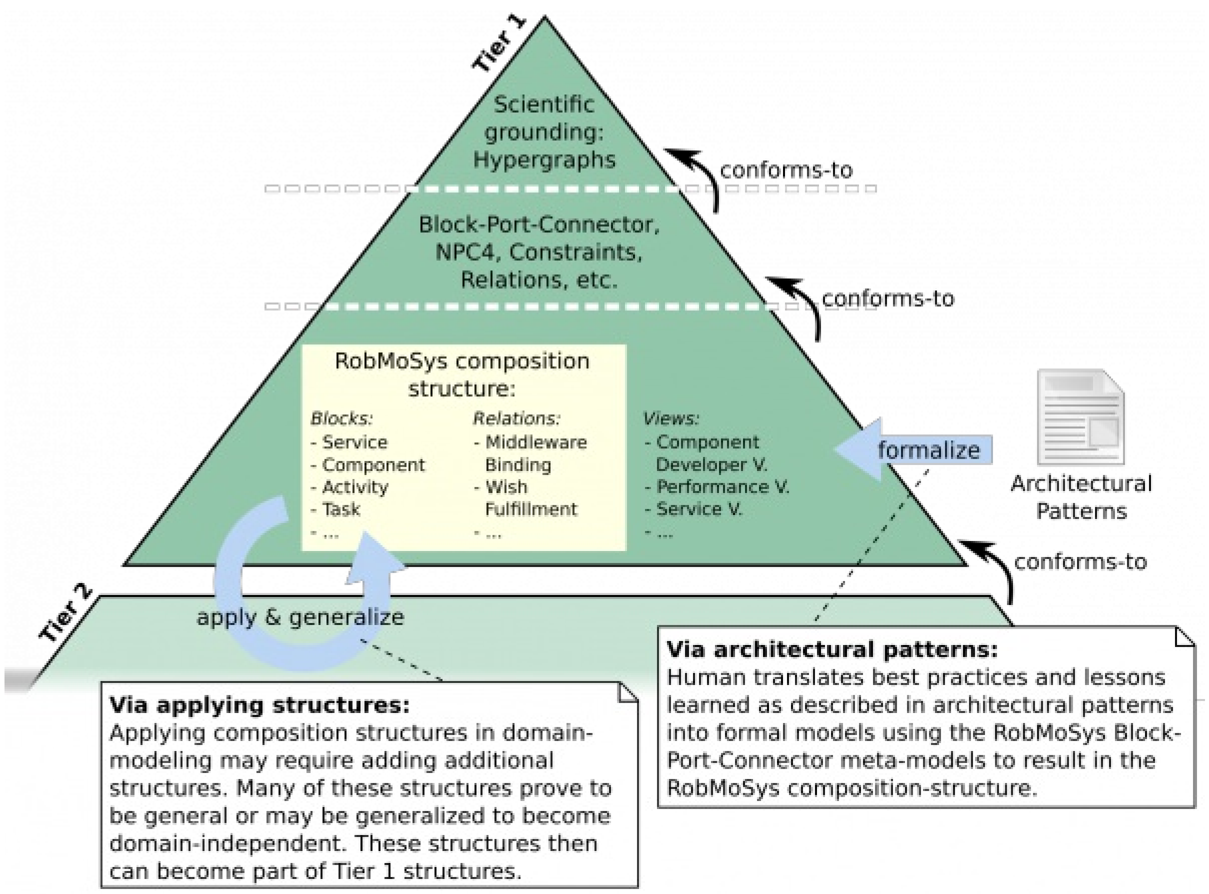
This image is borrowed from twouse.blogspot.de [http://twouse.blogspot.de/2010/08/owl-ecore-language-bridges.html]

The strength of ontologies is the representation of knowledge with extensible structures. Moreover, ontologies allow reasoning on knowledge and the inference of further knowledge. The strength of meta-models is the definition of clear and unambiguous structures. This is particularly useful to represent physical entities and physical properties of the real-world. There are robotics use-cases where in some cases ontologies and in other cases meta-models can be preferred. Therefore it is reasonable to allow using both of them in combination, rather than restricting the usage of only one of them in isolation.

modeling:principles · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/modeling:principles

## Tier 1 in Detail

Tier 1 provides the general structures for composition. The figure below shows the details of the structure of Tier-1 that refines into three levels. All the elements in Tier-1 are summarized as meta-meta-models. Moreover, the meta-meta-models within Tier-1 are organized themselves in a hierarchical manner in order to best serve the realization of the RobMoSys objectives. The lowest level within Tier 1 contains the RobMoSys composition structures. Tier-2 then conforms to these composition structures.



**Via applying structures:** Applying composition structures in domain-modeling may require adding additional structures. Many of these structures prove to be general or may be generalized to become domain-independent. These structures then can become part of Tier 1 structures.

**Via architectural patterns:** Human translates best practices and lessons learned as described in architectural patterns into formal models using the RobMoSys Block-Port-Connector meta-models to result in the RobMoSys composition-structure.

### The levels of Tier 1

#### Hierarchical Hypergraphs and Entity-Relation Model

Hierarchical Hypergraphs can be considered as the topmost abstraction level within Tier 1. It allows definition of a sound scientific grounding and a formalization in a most flexible model. Any modeling structure can be represented by a Hypergraph. The specific structures on the levels below are always specializations (i.e. refinements) of a Hypergraph.

The Hypergraph and Entity-Relation Model page provides additional details.

#### Block-Port-Connector

The next level on Tier 1 is the definition of blocks, ports and connectors as a general meta-level that allows definition of any domain-specific meta-model such as e.g. the RobMoSys composition structure (see below).

The Block-Port-Connector page provides a more detailed description.

**RobMoSys Composition Structure**

RobMoSys composition structures provide domain-specific meta-structures that are used on the lower Tier 2 and Tier 3 to design robotics models in specific robotics subdomains.

The RobMoSys Composition Structures page provides further details.

The RobMoSys views are a complementary technique to the RobMoSys composition structures. This technique supports definition of role-specific modeling views that allow modification and refinement of specific primitives without breaking the overall structures. This is a useful technique that directly supports separation of roles and at the same time allows realization of model-driven tooling that ensures overall system consistency.

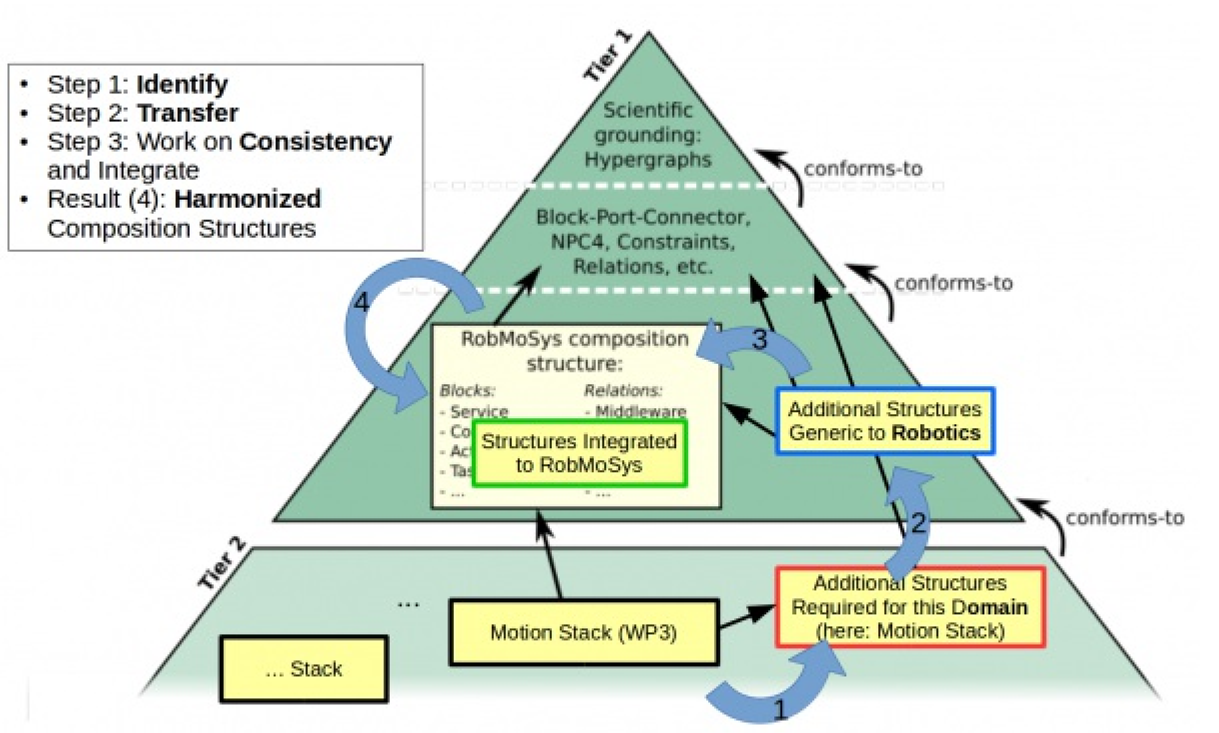
The RobMoSys Views page provides further details.

## Initial Structures and Evolvement of Tier 1

There are two approaches on how to come up with the composition structures in Tier 1. RobMoSys is a community effort and will guide contributors in one of these approaches such that their knowledge and methodology becomes accessible through the composition structures. For example, the following two approaches have already proven to be successful with respect to the integrated technical projects (ITPs) of RobMoSys.

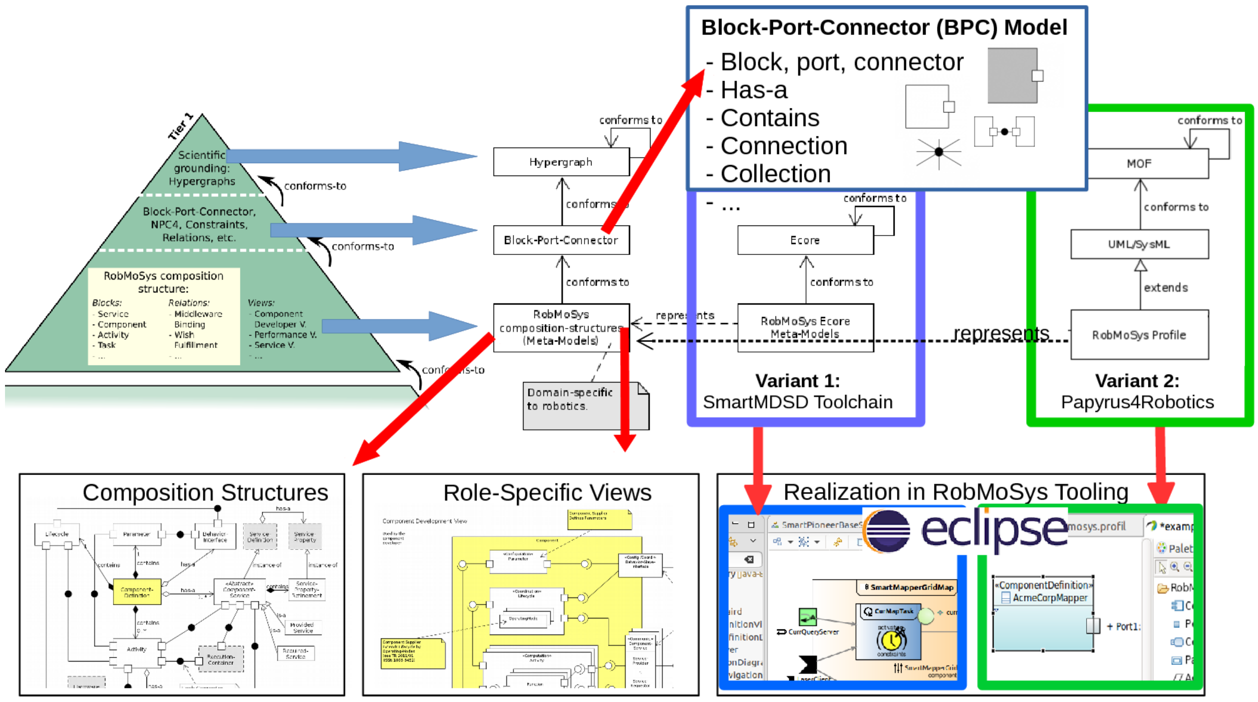
The first and initial approach to come up with composition structures is to formalize architectural patterns.

The second approach is to evolve the composition structures over time by generalizing existing domain-specific structures. In some cases, the composition structures of Tier 1 may not be sufficient or not complete for modeling in a particular robotics domain. This situation requires additional structures to be added on Tier 2. However, many of these structures tend to be generally applicable or may be generalized such that they become domain-independent and finally part of the composition structures. This is illustrated in the figure below.


Step 1: Identify
Step 2: Transfer
Step 3: Work on Consistency and Integrate
Result (4): Harmonized Composition Structures
Tier 1
Scientific grounding: Hypergraphs
conforms-to
Block-Port-Connector, NPC4, Constraints, Relations, etc.
conforms-to
RobMoSys composition structure:
Blocks:
- Service
Relations:
- Middleware
Structures Integrated to RobMoSys
Additional Structures Generic to Robotics
conforms-to
Tier 2
...
Motion Stack (WP3)
Additional Structures Required for this Domain (here: Motion Stack)
... Stack
1
2
3
4

The first step (step 1, figure above) is to identify the additional structures that are independent of an application but general to a domain. The second step is to transfer these structures to Tier 1, thereby making them domain independent (step 2, figure above). The final step is to work on the consistency of the newly identified structures with the existing composition structures with the aim to integrate them (step 3, figure above).

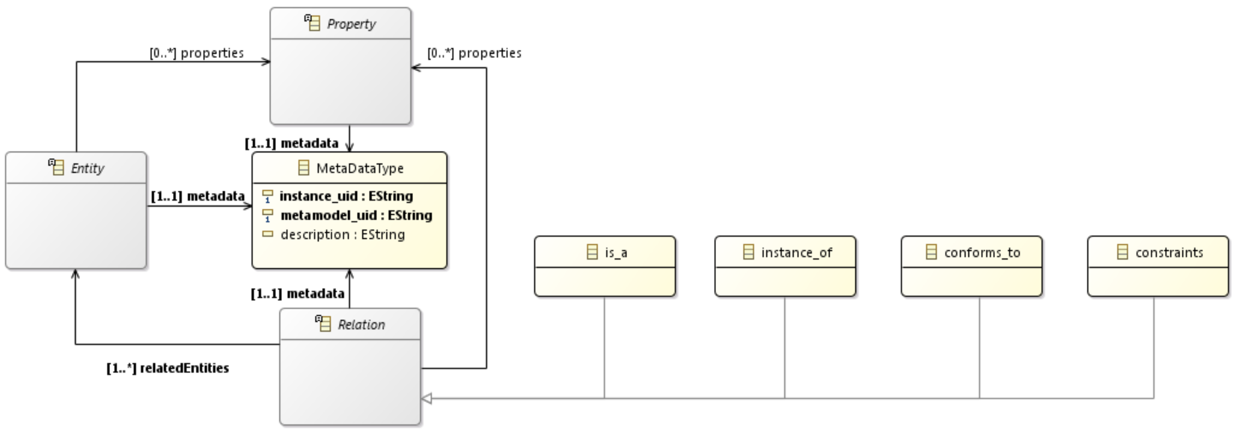
For example, it is necessary to shape them to the overall RobMoSys approach, taking separation of roles, composability, etc. into account. This results in the next generation of harmonized composition structures (step 4, figure above).


Block-Port-Connector (BPC) Model
- Block, port, connector
- Has-a
- Contains
- Connection
- Collection
- ...
Tier 1
Scientific grounding: Hypergraphs
Block-Port-Connector, NPC4, Constraints, Relations, etc.
RobMoSys composition structure:
conforms-to
Hypergraph
Block-Port-Connector
RobMoSys composition-structures (Meta-Models)
Domain-specific to robotics
represents
Ecore
RobMoSys Ecore Meta-Models
MOF
UML/SysML
extends
RobMoSys Profile
Variant 1: SmartMDSD Toolchain
Variant 2: Papyrus4Robotics
Composition Structures
Role-Specific Views
Realization in RobMoSys Tooling
eclipse

# Preliminary Ecore implementation of ER and BPC meta-models

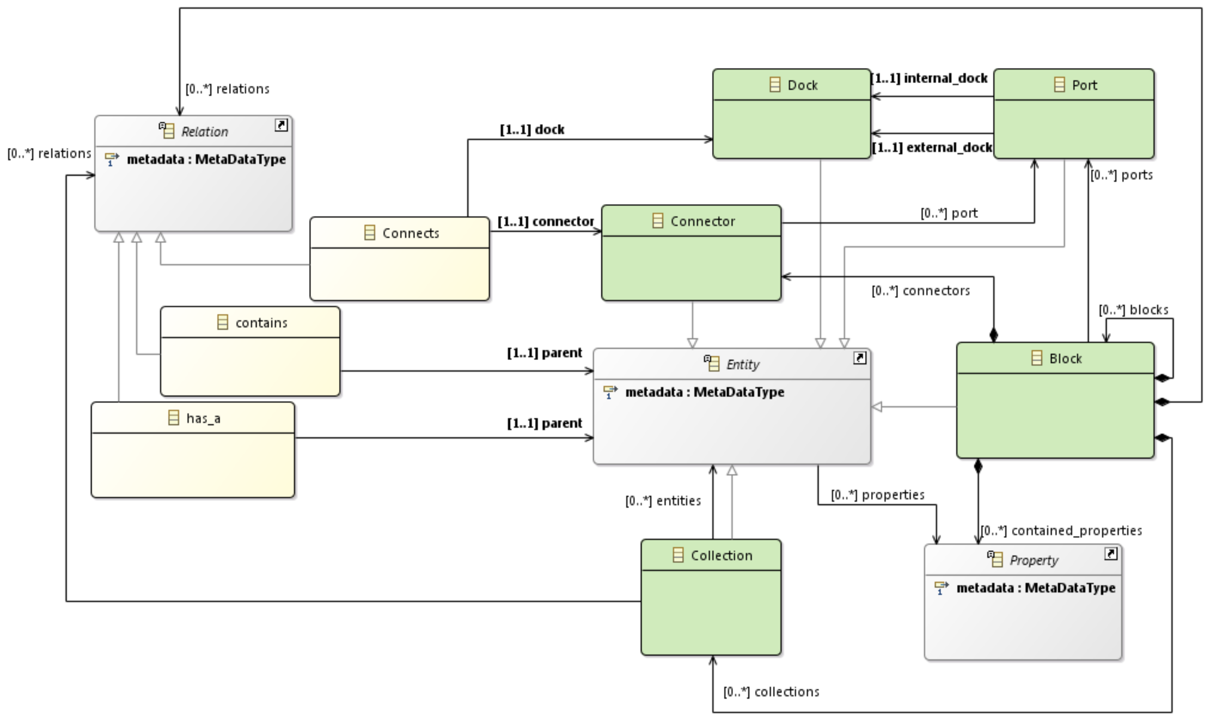
## Entity-Relation (ER) meta-model

The concepts provided by the ER meta-model comply with the definitions in Scientific Grounding



## Block-Port-Connector (BPC) meta-model

The following meta-model includes concepts that are defined in Block-Port-Connector



## Eclipse/Ecore implementation of ER and BPC meta-models

Eclipse/Ecore implementation of the above meta-models can be downloaded here

To access these meta-models you will need to:

1. Install Eclipse Neon Modeling [http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/neon3].

2. Import the plugins in your workspace.

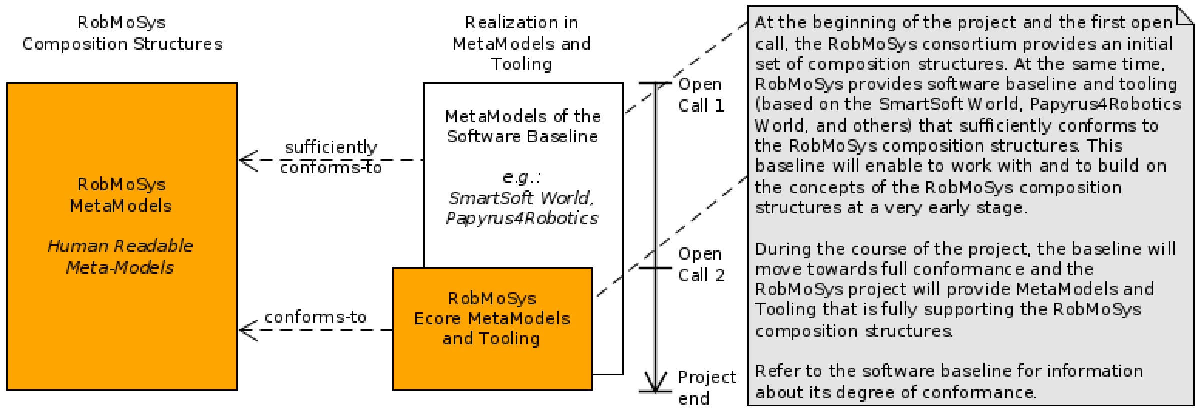
---

modeling:realization_alternatives:ecore_implem · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/modeling:realization_alternatives:ecore_implem

# Roadmap of MetaModeling

The RobMoSys project makes available a baseline of already existing metamodels. They sufficiently conform to the RobMoSys composition structures. For example, the SmartMARS metamodel form the The SmartSoft World and also metamodels in the Papyrus4Robotics World.

In the course of the project, RobMoSys is going to provide an Ecore implementation of the RobMoSys structures. RobMoSys Structures: Realization Alternatives describes this in more detail and also lists alternatives.


RobMoSys Composition Structures
Realization in MetaModels and Tooling
RobMoSys MetaModels
Human Readable Meta-Models
sufficiently conforms-to
MetaModels of the Software Baseline
e.g.: SmartSoft World, Papyrus4Robotics
conforms-to
RobMoSys Ecore MetaModels and Tooling
Open Call 1
Open Call 2
Project end

At the beginning of the project and the first open call, the RobMoSys consortium provides an initial set of composition structures. At the same time, RobMoSys provides software baseline and tooling (based on the SmartSoft World, Papyrus4Robotics World, and others) that sufficiently conforms to the RobMoSys composition structures. This baseline will enable to work with and to build on the concepts of the RobMoSys composition structures at a very early stage.

During the course of the project, the baseline will move towards full conformance and the RobMoSys project will provide MetaModels and Tooling that is fully supporting the RobMoSys composition structures.

Refer to the software baseline for information about its degree of conformance.

## See also

- The given description also holds true for the Roadmap of Tools and Software
- Conformance of SmartMARS Metamodel to RobMoSys composition structures

modeling:roadmap · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/modeling:roadmap

# Metamodels

The RobMoSys metamodels are the RobMoSys Composition Structures.

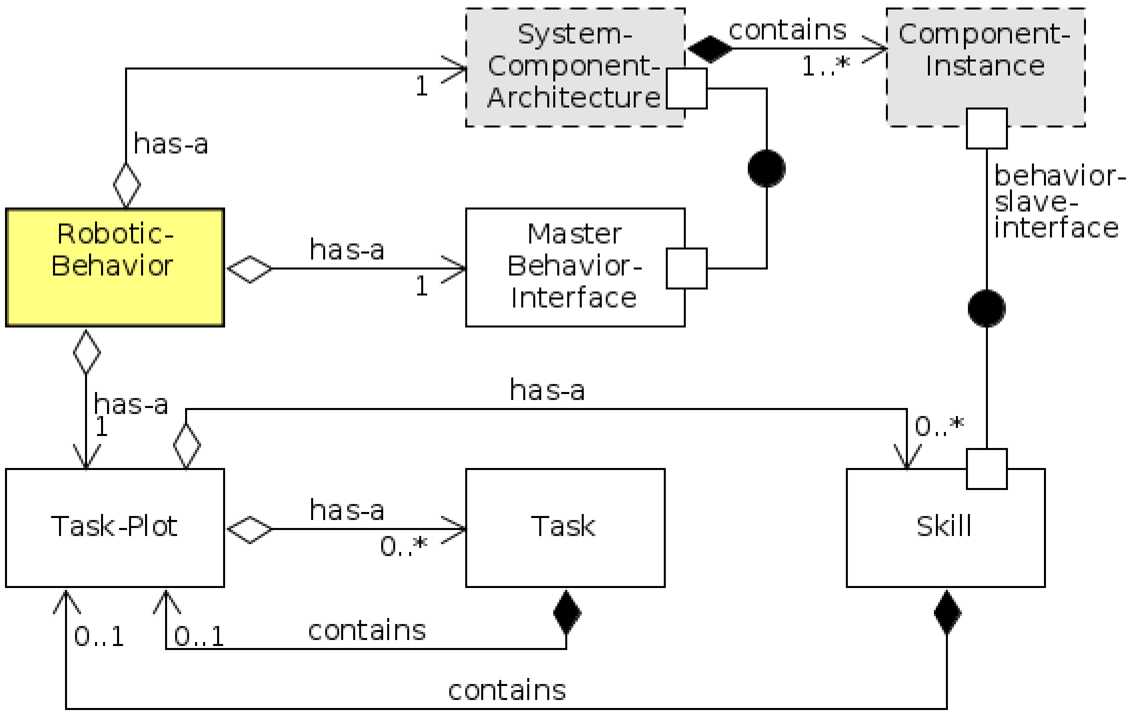
List of metamodels:

- Robotic Behavior Metamodel
- Communication-Object Metamodel
- Communication-Pattern Metamodel
- Component-Definition Metamodel
- Deployment Metamodel
- Cause-Effect-Chain and its Analysis Metamodels
- Platform Metamodel
- System Service Architecture and Service Fulfillment Metamodels
- Service-Definition Metamodel
- System Component Architecture Metamodel

# Robotic Behavior Metamodel

The Robotic Behavior Metamodel is one part of the RobMoSys Composition Structures that is responsible for specifying the overall run-time behavior of a robot acting in real-world environments.

The Robotic Behavior Metamodel defines structures for modeling task-plots of a robot (see figure below). Task-plots define sequences of tasks required to achieve certain goals at run-time. Each task itself can contain another task-plot. This introduces hierarchy into the task-plot modeling where high level tasks (such as e.g. making-coffee) are refined into lower level tasks (such as e.g. approach the kitchen, operate the coffee machine and bring the coffee back to the customer). At the lower end of the abstraction hierarchy, tasks eventually operate (i.e. to coordinate and configure) according software components that do the actual “work” of a task. In this sense, tasks are passive, they just delegate the work to components in the system and await the results (i.e. success or failure). The interaction between task-plots and components is over skills. In this sense, a skill abstracts the technical coordination interface of a component and makes it accessible for task-plots. A skill by itself might “inject” additional task-plots. This feature is particularly useful for modeling alternative behaviors in case of contingencies in the overall behavior. For example, a skill commanding a navigation component to approach a room might get the result that the navigation component failed to do so (e.g. due to a blocked hallway). In this situation, the according skill might inject an alternative strategy, namely to first go to another location and to try the current task later (or whatever other strategy might be appropriate here).



A service robot is a physical entity that needs to cope with the physical constraints of the real-world. For instance, actions of the robot, once performed, might be irreversible and always can fail. This also means that at each point in time, the control hierarchy on the robot must be clear. Simply speaking, a robot cannot decide in parallel to go to left and to right at the same time (for most of the robots, this is physically impossible). In consequence, there is typically only one entity on each robot that is responsible for executing the robotic behavior models namely the sequencer (see this page [http://www.servicerobotik-ulm.de/drupal/?q=node/86] for further details on sequencing).

For the interaction between the behavior model and the software components in a system, the robot behavior uses the “Master-Behavior-Interface”. Each component in the system by default implements the counter part “Slave-Behavior-Interface” (not displayed in the figure). Therefore, the robot-behavior depends on the system-component-architecture for the interaction with the according component-instances.

One existing realization of the robotic behavior meta-model is SmartTCL [http://www.servicerobotik-ulm.de/drupal/?q=node/84]. SmartTCL [http://www.servicerobotik-ulm.de/drupal/?q=node/84] conforms to the above presented meta-model and can be used as an initial software baseline already now.

See next:

- Deployment Metamodel

See also:

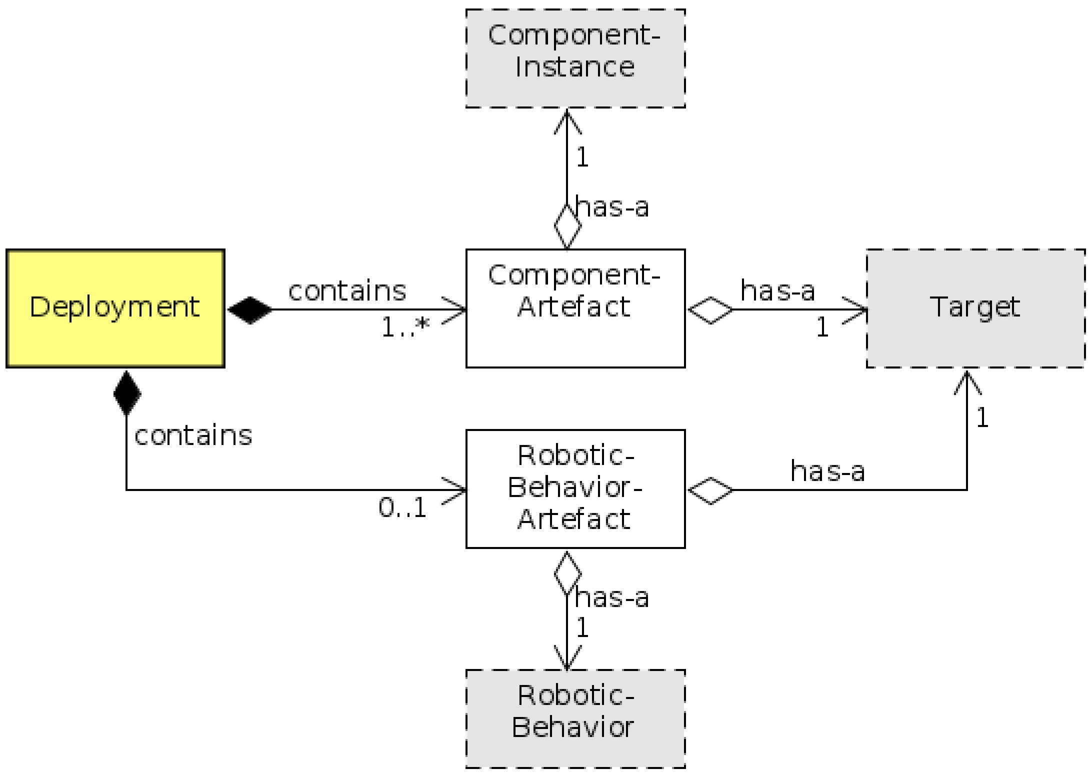
- System Component Architecture Metamodel
- Task Composition
- Architectural Pattern for Task-Plot Coordination (Robotic Behaviors)

---

modeling:metamodels:behavior · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/modeling:metamodels:behavior

# Deployment Metamodel

The Deployment Metamodel (see figure below) is part of the overall RobMoSys Composition Structures. This meta-model links (i.e. interfaces between) the three meta-models, namely System Component Architecture, Platform and Robotic Behavior.



The main concerns of this meta-model are to define artefacts and to assign them to selected targets. This meta-model is inspired by the UML deployment model. There are two artefact types namely component-artefacts and robotic-behavior-artefacts. Component-artefacts represent typically the precompiled binary form of component-instances (including generated ini-files and start scripts). The robotic-behavior-artefact is the physical representation of the robotic-behavior model (often this is an interpretable model).

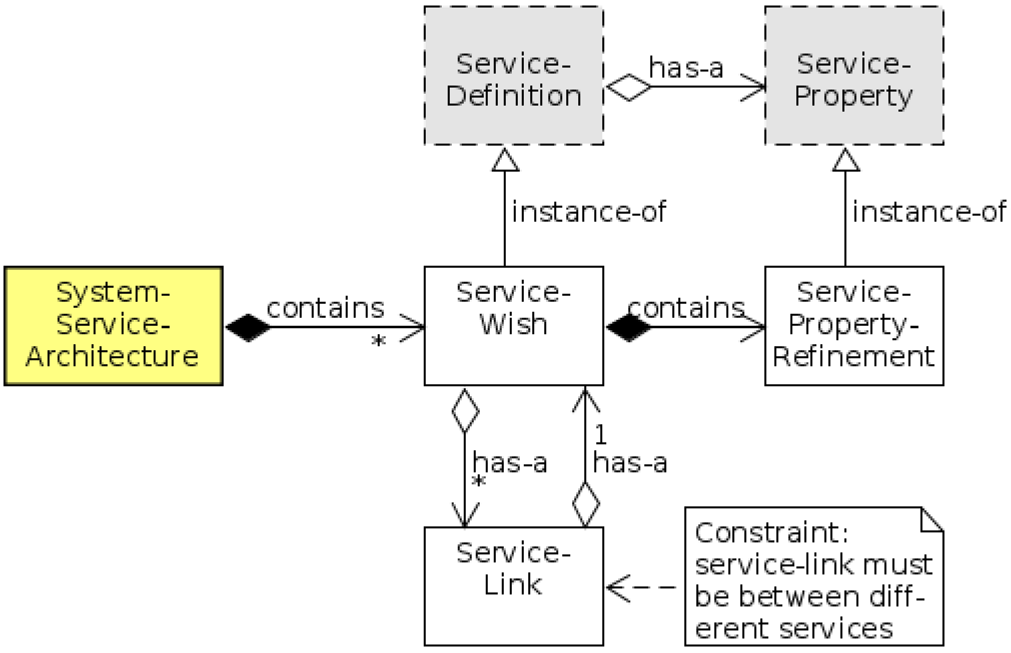
Depending on the used modeling tool, the deployment meta-model could also be connected with the actual deployment action that copies the component and robotic-behavior artefacts to the according target platforms. However, this is a matter of tooling and is independent of the deployment meta-model as such.

See also:

- Platform Metamodel
- System Component Architecture Metamodel
- Robotic Behavior Metamodel

# System Service Architecture and Service Fulfillment Metamodels

The System Service Architecture Metamodel is a particularly useful meta-model for System Architects. This meta-model allows the definition of service-based reference architectures for specific (sub-)domains on Tier 2. This meta-model depends on service-definitions and itself can be used to check “conformance” of a system-component-architecture to this service-based reference architecture. Checking this conformance is one of the main concerns of the service-fulfillment meta-model (see the following section below).


Service-Definition
has-a
Service-Property
instance-of
instance-of
System-Service-Architecture
contains
*
Service-Wish
contains
Service-Property-Refinement
has-a
*
1
has-a
Service-Link
Constraint: service-link must be between diff-erent services

The System Service Architecture Metamodel specifies service-wishes which are component-independent definitions of service-requirements for a set of systems. Moreover, links between service-wishes specify component-independent inter-service dependencies (i.e. a service-wish might depend on the existence of another service-wish).

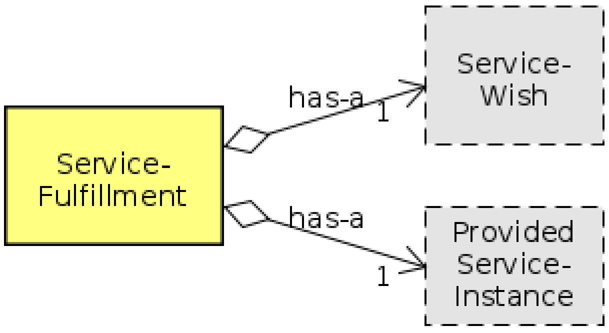
For example, a set of recurring services for a navigation stack (such as localization, mapping, path-planning, obstacle-avoidance, etc.) can be specified in advance independent of a concrete system and independent of concrete implementations in software components. In addition, it can be specified that a path-planning service typically depends on the existence of a localization service which itself depends on a mapping service, etc.

In addition, a service-wish can instantiate several service-properties which allow definition of specific Quality-of-Service (QoS) attributes. Examples for such attributes can be found here.

Please note, that the definition of service-based reference architectures seldom defines all services of one concrete system. Instead, a service-based reference architectures typically defines only the recurring services for (or from) a set of systems.

## Service Fulfillment Metamodel

The Service Fulfillment Metamodel maps the service-wishes from a system-service-architecture (see above) with the provided-service-instances from a system-component-architecture. This mapping of service-wishes to provided-service-instances is called service-fulfillment. This is a powerful meta-model that allows definition of domain-specific de-facto standard architecture and thus considerably increases reuse of recurring specifications and at the same time fosters competition on implementation level (conforming to modeled reference architectures).



While the Service Fulfillment Metamodel directly depends on the two meta-models “System Service Architecture” (see above) and “System Component Architecture”, the order of usage of these two models is not strict. For instance, an existing (i.e. fully specified) system-component-architecture can be used to check whether it conforms to a later (or independently) defined system-service-architecture. Or, a specified system-service-architecture can be used upfront to select conforming components (from a component repository) for a current (i.e. new) system-component-architecture under development. Of course, all the intermediate options are also possible with partial specifications of system-service-architectures and system-component-architectures with intermediate checking of conformance.

An interesting option for this meta-model is to use constraint solvers to automatically pre-select existing component-definitions from a component repository according to the specified system-service-architecture. This is a powerful mechanism that considerably improves efficiency in designing new systems.

See next:

- System Component Architecture Metamodel

See also:

- Service-Definition Metamodel

## Acknowledgement

This document contains material from:

- Stampfer2018 Dennis Stampfer, "Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics". Dissertation, Technische Universität München, München, Germany, 2018. [http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2]
- Lotz2018 Alex Lotz, "Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System", Dissertation, Technische Universität München, München 2018. [https://mediatum.ub.tum.de/?id=1362587]
- Lutz2017 Matthias Lutz, “Model-Driven Behavior Development for Service Robotic Systems: Bridging

the Gap between Software- and Behavior-Models,” 2017. (unpublished work)

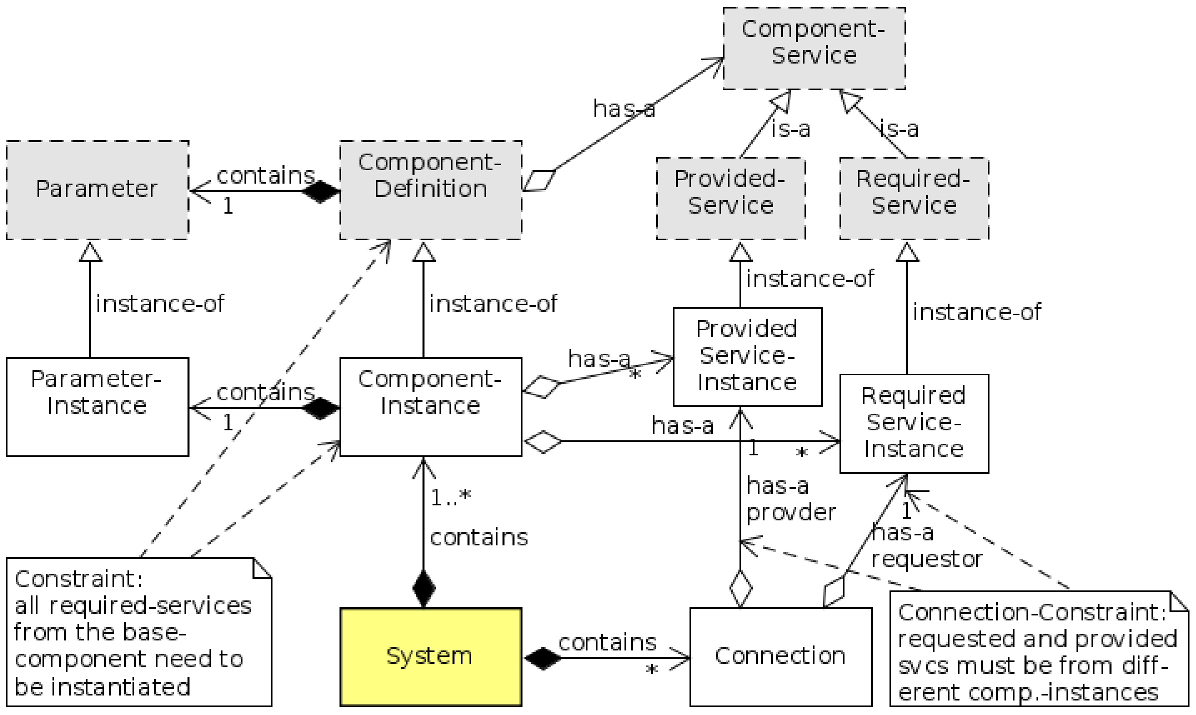
---

modeling:metamodels:service-architecture · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/modeling:metamodels:service-architecture

# System Component Architecture Metamodel

The System Component Architecture Metamodel depends on the Component-Definition Metamodel as part of the RobMoSys Composition Structures.

The System Component Architecture Metamodel (see figure below) is the platform-independent specification of a software system consisting of instantiated components. This means that selected component-definitions are instantiated and initially wired (i.e. connected). Please note, that at this point individual components can still be distributed over (i.e. deployed to) different target platforms (i.e. PCs) without affecting this model.



An instantiated component also instantiates its (internal) structures such as the definition of parameters and the component's provided/required services. By instantiating parameters, it is possible to define system-specific and application-related parameter values (i.e. parameter refinement) that differ from the default parameter values in the original component-definition. It is important to notice that a component-instance cannot instantiate any structures that have not been defined in the component-definition (base-model). Moreover, all the required services of a component-definition also need to be instantiated within the derived component-instance. This can be easily supported by modeling tools that can pre-generate component-instance models (using so called proposal-providers) out of selected component-definitions. This is an important functional constraint that allows checking that each required service also is connected to an according provided service of another component-instance in the system. Finally, a Connection defines initial wiring between provided and required services of different components. It is worth mentioning that this initial wiring can be dynamically changed at run-time (if needed) using the dynamic wiring pattern.

*At this point, it is also worth mentioning that at the moment a system is built from components as basic building blocks. In future versions of this meta-model the hierarchical definition for systems-of-systems (i.e. composite components) will be introduced. Composite components will be introduced as an extension to the current meta-*

*model that allows building systems out of sub-systems which again can be built out of yet other sub-systems and so forth.*

See next:

- Deployment Metamodel

See also:

- Component-Definition Metamodel

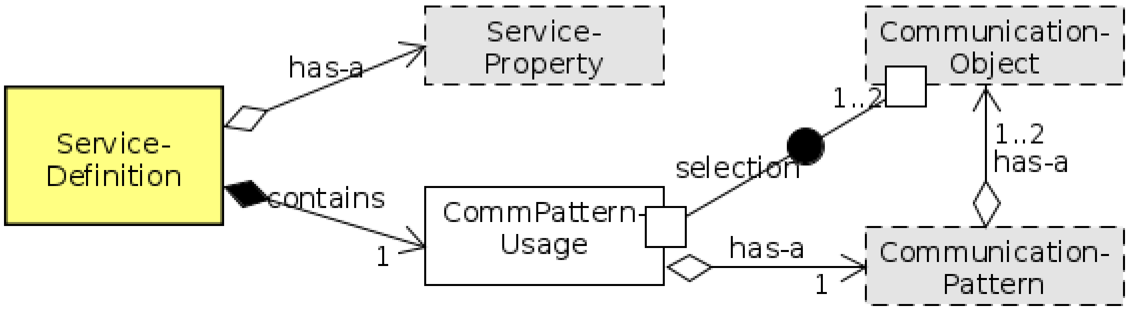
---

modeling:metamodels:system · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/modeling:metamodels:system

# Service-Definition Metamodel

The Service-Definition Metamodel is one of the core composition structures of RobMoSys.

A Service allows interaction (i.e. regular exchange of information) between software components. A Service consists of service-properties (defined in an external metamodel) and a communication-pattern-usage. The communication-pattern-usage selects a certain Communication Pattern with a pattern-specific selection of according number of communicated data-structures (i.e.Communication Objects).

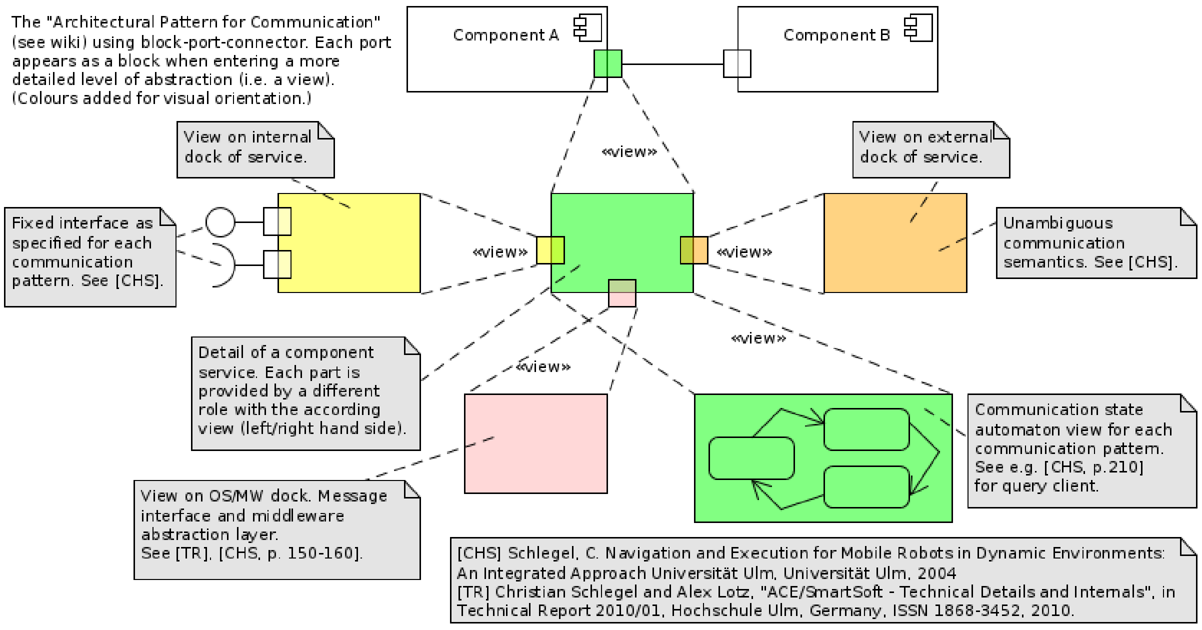


The service-definition is used as a base meta-model for component-definition and for service-architecture. The relation between these three service-related meta-models form a service triangle (see the example of a Service Triangle).

## Views of a Service

A service can be graphically represented as a port of a component (just like in UML). However, depending on the current role-specific view with an according level of abstraction, a service "port" can reveal additional details that are not visible (i.e. hidden/encapsulated) for another role. The more detailed view enrolls additional internal structures of the port and the port itself might appear as a block for that role (see figure below). This is a useful pattern to provide different levels of abstraction, each adequate for the according developer role (with certain responsibilities and concerns).

This pattern can be applied recursively, where the ports of the currently more detailed view can again contain additional internal structures (not visible for the current role). For instance, a the "external" port of a service (see orange block on the right in the figure below) provides sufficiently detailed and stable communication semantics between interacting components (defined through a selected Communication Pattern). Second, the "internal" port of a service provides a clear API towards implementation within a component (also defined as part of the Communication Pattern). Third, the "bottom" port of a service provides a generic middleware abstraction layer that allows using any general purpose communication middleware without affecting the communication semantics (see Communication Objects).


The "Architectural Pattern for Communication" (see wiki) using block-port-connector. Each port appears as a block when entering a more detailed level of abstraction (i.e. a view). (Colours added for visual orientation.)
Component A
Component B
«view»
View on internal dock of service.
View on external dock of service.
Fixed interface as specified for each communication pattern. See [CHS].
«view»
«view»
Unambiguous communication semantics. See [CHS].
«view»
Detail of a component service. Each part is provided by a different role with the according view (left/right hand side).
«view»
Communication state automaton view for each communication pattern. See e.g. [CHS, p.210] for query client.
View on OS/MW dock. Message interface and middleware abstraction layer. See [TR], [CHS, p. 150-160].
[CHS] Schlegel, C. Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach Universität Ulm, Universität Ulm, 2004
[TR] Christian Schlegel and Alex Lotz, "ACE/SmartSoft - Technical Details and Internals", in Technical Report 2010/01, Hochschule Ulm, Germany, ISSN 1868-3452, 2010.

References:

- Christian Schlegel. “Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach”. *Dissertation*. University of Ulm, 2004PDF [http://www.hs-ulm.de/users/cschlege/_downloads/phd-thesis-schlegel.pdf]
- Christian Schlegel and Alex Lotz, “ACE/SmartSoft - Technical Details and Internals”, in *Technical Report 2010/01*, Hochschule Ulm, Germany, ISSN 1868-3452, 2010.PDF [http://www.zafh-servicerobotik.de/dokumente/ZAFH-TR-01-2010-ISSN-1868-3452.pdf]

See next:

- Component-Definition Metamodel

See also:

- Communication-Pattern Metamodel
- Communication-Object Metamodel
- Service-Based Composition (Service Triangle)
- Service Design View

modeling:metamodels:service · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/modeling:metamodels:service

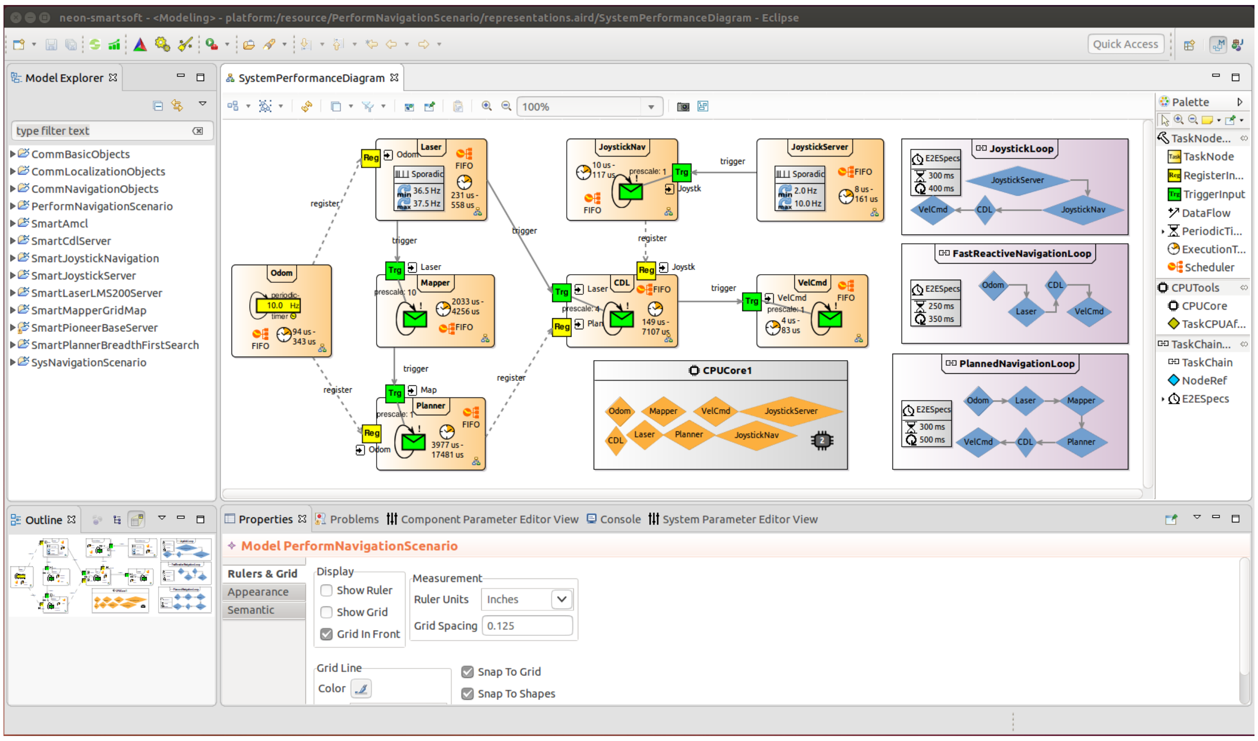
# Cause-Effect-Chain and its Analysis Metamodels

The Cause-Effect-Chain meta-model and the according Analysis Metamodel are two parts of the overall RobMoSys Composition Structures. See also Architectural Pattern for Stepwise Management of Extra-Functional Properties and Managing Cause-Effect Chains in Component Composition.

The main concern in these meta-models is to specify application-specific (often non-functional) system properties. This is considered as an important aspect in RobMoSys, which is however sparsely addressed in robotics research. One of the core publications that addresses this issue for a narrowed problem domain, namely for designing causal dependencies and overall end-to-end delays in a system, can be found here:

- Alex Lotz, Arne Hamann, Ralph Lange, Christian Heinzemann, Jan Staschulat, Vincent Kesel, Dennis Stampfer, Matthias Lutz, and Christian Schlegel. “Combining Robotics Component-Based Model-Driven Development with a Model-Based Performance Analysis.” In: IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR). San Francisco, CA, USA, Dec. 2016, pp. 170–176. LINK [http://dx.doi.org/10.1109/SIMPAR.2016.7862392]
- Alex Lotz, "Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System", Dissertation, Technische Universität München, München 2018. [https://mediatum.ub.tum.de/?id=1362587]

This publication also provides an initial version of a meta-model that is used as first version in RobMoSys for addressing the overall problem domain.



Model PerformNavigationScenario

Rulers & Grid | Appearance | Semantic

Display: Show Ruler [ ] Show Grid [ ] Grid In Front [✓]

Measurement: Ruler Units: Inches; Grid Spacing: 0.125

Grid Line: Color

Snap To Grid [✓] Snap To Shapes [✓]

An open-source reference implementation of according model-driven tooling (see above figure) is publicly available within the sourceforge git repository [https://sourceforge.net/p/smart-robotics/smartmdsd-v3/ci/master/tree/]. Further information thereto can be found here [http://www.servicerobotik-ulm.de/drupal/?q=node/83].

Later versions of the initial meta-model will be extended throughout the run-time of the RobMoSys project to address a broader problem domain.

## Acknowledgement

This document contains material from:

- Lotz2018 Alex Lotz, "Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System", Dissertation, Technische Universität München, München 2018. [https://mediatum.ub.tum.de/?id=1362587]
- Stampfer2018 Dennis Stampfer, "Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics". Dissertation, Technische Universität München, München, Germany, 2018. [http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2]
- Lutz2017 Matthias Lutz, “Model-Driven Behavior Development for Service Robotic Systems: Bridging the Gap between Software- and Behavior-Models,” 2017. (unpublished work)

---

modeling:metamodels:performance · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/modeling:metamodels:performance

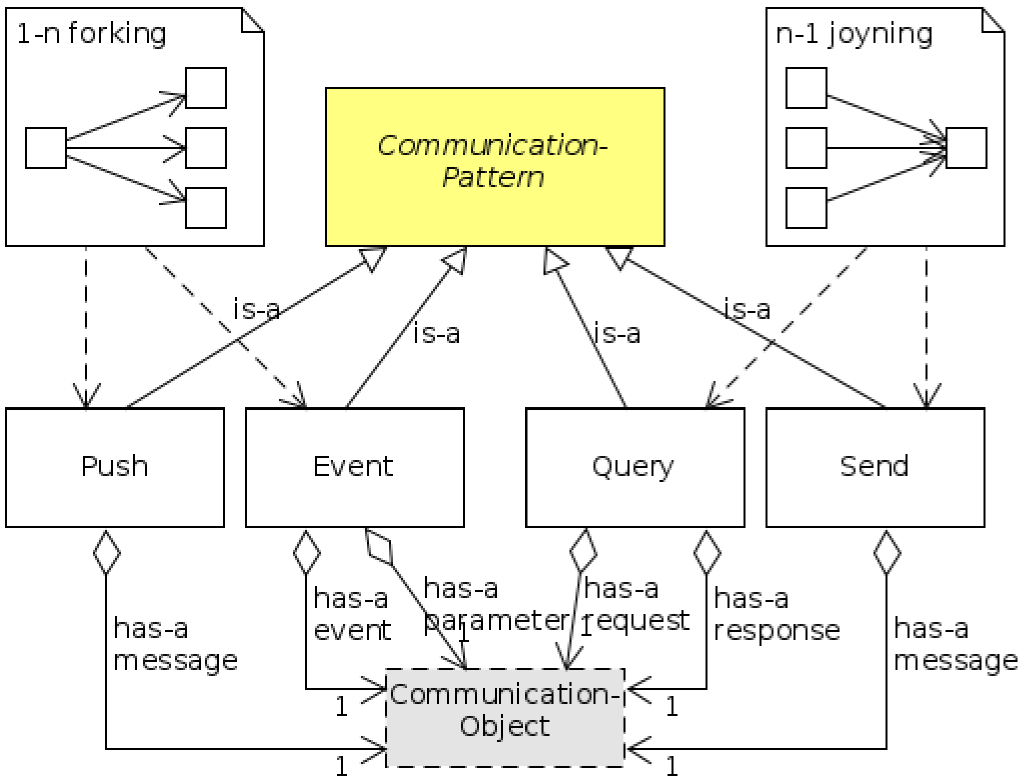
# Communication-Pattern Metamodel

The RobMoSys communication patterns define the semantics in which software components exchange data over Services, e.g. via one-way “send”, two-way “request-response”, and publish/subscribe mechanisms on a timely basis or based on availability of new data. RobMoSys defines communication patterns to enable composability of services and components.

The general concept of a communication pattern originates from [Schlegel2004] where it is described in the context of the SmartSoft Framework in 2004. Since that, the there described communication patterns have been extended by several activities and have proven to be of generic use (see e.g. [UCM]). RobMoSys adopts a set of existing communication patterns (see below) that have proven to be relevant. For their definition, the wiki provides specific pointers to existing external documentation.

It is important to have a fixed set of a few communication patterns that efficiently support composition through unambiguous communication semantics and clearly defined communication interfaces. In addition, the mapping to different communication middlewares becomes possible over a generic middleware abstraction layer that is part of each communication pattern.

The communication pattern metamodel is depicted below. The name of an individual pattern (middle row of elements in the figure, e.g. send, query, push) refers to its definition in an external document as described in the remainder of this page.

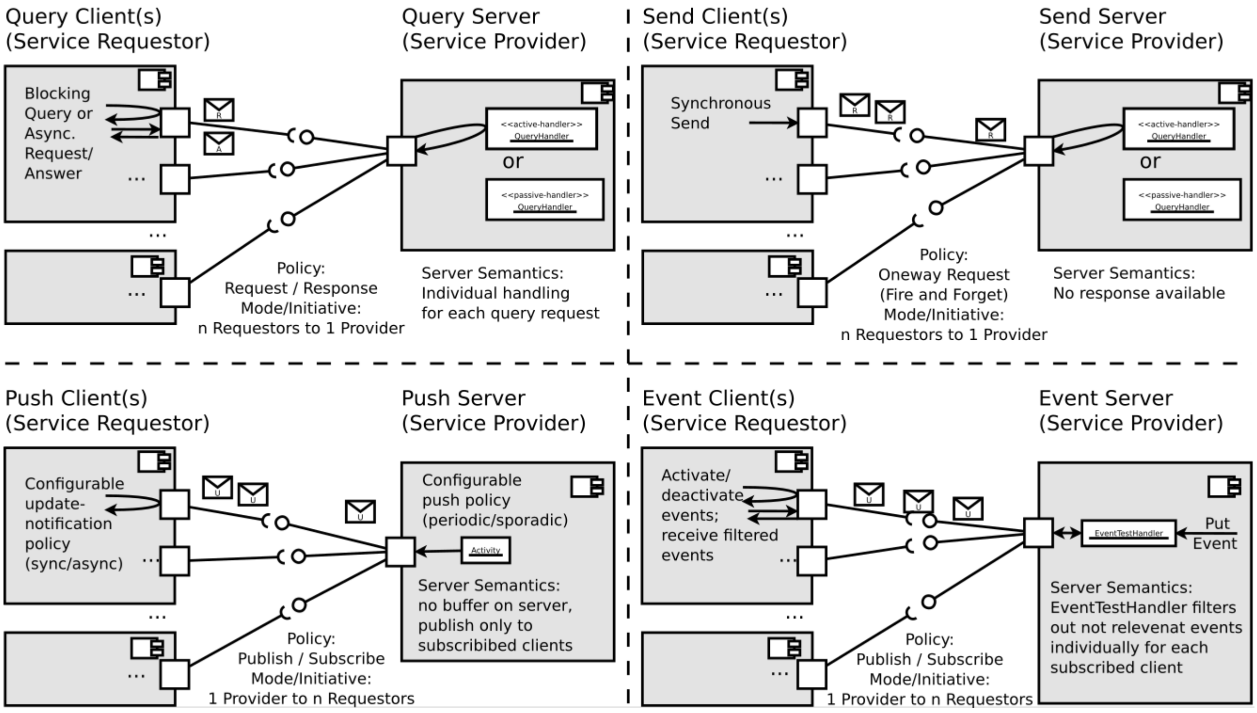


## Component Communication Patterns

The four communication patterns (see table below) define the basic set of recurring communication semantics that proved to be sufficient for all robotics use-cases related to inter-component communication at the service level (for service level, see Separation of Levels and Separation of Concerns).

| Pattern Name | Interaction Model | Description | Definition |
|---|---|---|---|
| **Send** | Client/Server | One way communication | [Schlegel2004, pp. 85-88] |
| **Query** | Client/Server | Two way request/response | [Schlegel2004, pp. 88-96] |
| **Push** | Publisher/Subscriber | 1-n distribution | [Schlegel2004, pp. 96-99] |
| **Event** | Publisher/Subscriber | 1-n asynchronous condition notification | [Schlegel2004, pp. 103-112] |

The figure below provides a schematic overview of the communication semantics.



Query Client(s) (Service Requestor)
Query Server (Service Provider)
Blocking Query or Async. Request/ Answer
<<active-handler>> QueryHandler
or
<<passive-handler>> QueryHandler
Policy: Request / Response Mode/Initiative: n Requestors to 1 Provider
Server Semantics: Individual handling for each query request
Send Client(s) (Service Requestor)
Send Server (Service Provider)
Synchronous Send
Policy: Oneway Request (Fire and Forget) Mode/Initiative: n Requestors to 1 Provider
Server Semantics: No response available
Push Client(s) (Service Requestor)
Push Server (Service Provider)
Configurable update-notification policy (sync/async)
Configurable push policy (periodic/sporadic)
Activity
Server Semantics: no buffer on server, publish only to subscribibed clients
Policy: Publish / Subscribe Mode/Initiative: 1 Provider to n Requestors
Event Client(s) (Service Requestor)
Event Server (Service Provider)
Activate/ deactivate events; receive filtered events
EventTestHandler
Put Event
Server Semantics: EventTestHandler filters out not relevenat events individually for each subscribed client
Policy: Publish / Subscribe Mode/Initiative: 1 Provider to n Requestors

## Coordination and Configuration Patterns

The four coordination and configuration patterns (see table below) provide recurring semantics that proved to be sufficient for robotics use-cases related to behavior coordination (coordination of software components at the lower / skill level of the robotic behavior by the sequencing layer; for layers in robotic behavior coordination see Architectural Pattern for Task-Plot Coordination (Robotic Behaviors)).

| Pattern Name | Interaction Model | Description | Definition |
|---|---|---|---|
| **Parameter** | Master/Slave | Run-time configuration of components, see [Stampfer2016] | see [Lutz2014], [Lutz2017] |
| **State** | Master/Slave | Lifecycle management and mode (de-)activation | see [Schlegel2011] |

| Pattern Name | Interaction Model | Description | Definition |
|---|---|---|---|
| Dynamic Wiring | Master/Slave | Run-time connection re-wiring | see [Schlegel2004, p. 11] |
| Monitoring | Master/Slave | Run-time monitoring and introspection of components [Stampfer2016] | see [Lotz2011] |

Each component in a system should by default implement the slave part of each of the four patterns. In addition, there is typically one specific component per system that implements the master part of the patterns and that is responsible to centrally coordinate the robot behavior (the sequencer, see Architectural Pattern for Task-Plot Coordination (Robotic Behaviors) and for Component Coordination for further details).

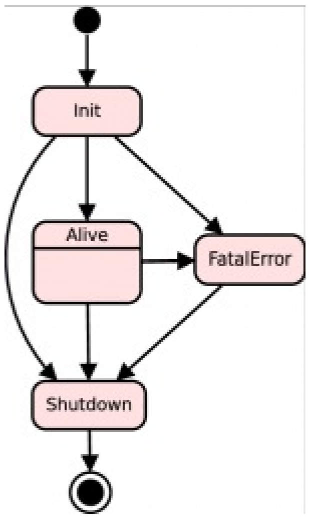
## Parameter

The Parameter pattern allows run-time configuration of components. The following links provide further details:

- Parameter Definition [http://www.servicerobotik-ulm.de/toolchain-manual/html/ch02s02s03.html]
- Parameter Usage in a Component [http://www.servicerobotik-ulm.de/toolchain-manual/html/ch02s03s02.html#UsingToolchain_ComponentDevelopmentView_CompModeling_CompParameter

## State

The state management of a component is one of the central patterns for run-time coordination of components. On the one hand, state management is about the generic lifecycle state-automaton (see figure on the right) that each component implements by default and that allows coordinated handling of the component's start-up and shutdown procedures as well as the component's fatal-error mode. In addition, component's individual run-time modes can be specified as explained in the following reference:



- Christian Schlegel, Alex Lotz and Andreas Steck, “SmartSoft - The State Management of a Component”, in *Technical Report 2011/01*, Hochschule Ulm, Germany, ISSN 1868-3452, 2011. PDF [http://www.zafh-servicerobotik.de/dokumente/ZAFH-TR-01-2011-ISSN-1868-3452.pdf]
- Coordinating Activities and Life Cycle of Software Components
- Coordinating Activities and Life Cycle of Software Components [https://robmosys.eu/wiki/composition:component-activities:start]

## Dynamic Wiring

Dynamic Wiring is used to increase run-time robustness and flexibility by dynamically changing the wiring between components. Additional details can be found here:

- Christian Schlegel. “Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach”. *Dissertation*. University of Ulm, 2004PDF [http://www.hs-ulm.de/users/cschlege/_downloads/phd-thesis-schlegel.pdf]

## Monitoring

Run-time Monitoring and Introspection of components is an important aspect in robotics that requires dedicated interaction mechanisms. The following reference provides details of a concrete realization:

- Alex Lotz, Andreas Steck, and Christian Schlegel. “Runtime Monitoring of Robotics Software Components: Increasing Robustness of Service Robotic Systems”, in *International Conference on Advanced Robotics (ICAR '11)*, Tallinn, Estonia, June 2011.IEEE-Link

[http://ieeexplore.ieee.org/document/5174736/?tp=&arnumber=5174736]

## RobMoSys Tooling Support

### Tooling Support by the SmartSoft World

- The SmartSoft World is fully conformant to the RobMoSys communication patterns. The mapping of communication patterns in the SmartSoft World is described in
  - Christian Schlegel and Alex Lotz, “ACE/SmartSoft - Technical Details and Internals”, in *Technical Report 2010/01*, Hochschule Ulm, Germany, ISSN 1868-3452, 2010.PDF [http://www.zafh-servicerobotik.de/dokumente/ZAFH-TR-01-2010-ISSN-1868-3452.pdf]
- The SmartMDSD Toolchain allows to use RobMoSys compliant communication patterns and also is an example of how to realize their metamodel with Ecore.
- The SmartTCL [http://www.servicerobotik-ulm.de/drupal/?q=node/84] language conforms to the RobMoSys composition structures and can be used for Robot Behavior Coordination [http://www.servicerobotik-ulm.de/drupal/?q=node/86].
- See also Conformance of SmartSoft to RobMoSys composition structures

## See Also

- Architectural Pattern for Task-Plot Coordination (Robotic Behaviors)
- Architectural Pattern for Component Coordination
- Communication Pattern View
- Service-Definition Metamodel
- Component Metamodel

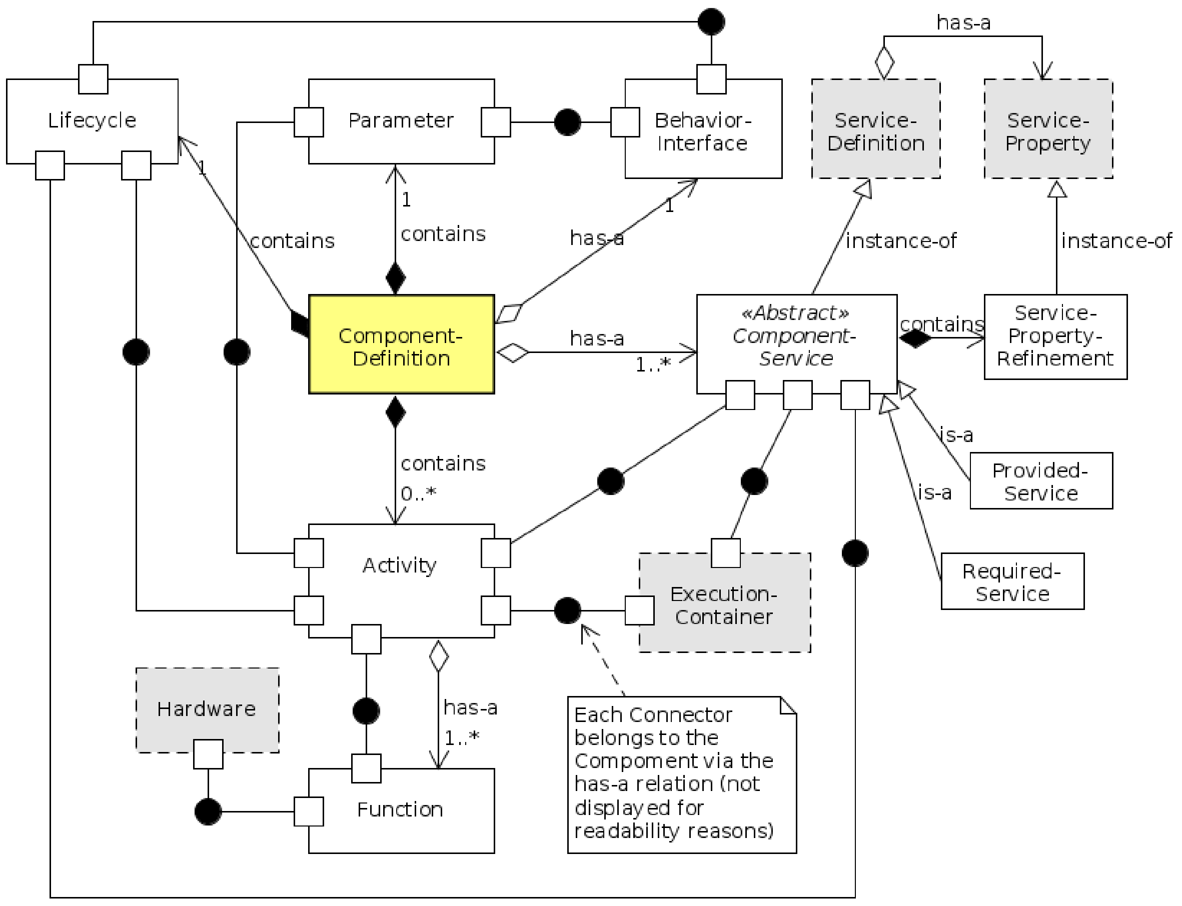
## References

- [Schlegel2004] Christian Schlegel. “Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach”. *Dissertation*. University of Ulm, 2004PDF [http://www.hs-ulm.de/users/cschlege/_downloads/phd-thesis-schlegel.pdf]
- [Stampfer2016] D. Stampfer, A. Lotz, M. Lutz und C. Schlegel, „The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software,“ in Journal of Software Engineering for Robotics (JOSER), 2016, pp. 3-19. Link [http://joser.unibg.it/index.php/joser/article/view/91]
- [UCM] Object Management Group (OMG). Unified Component Model for Distributed, Real-Time and Embedded Systems RFP (UCM). Document number: mars/2013-09-10. Sept. 2013. LINK [http://www.omg.org/cgi-bin/doc?mars/2013-09-10].
- [Lutz2017] Matthias Lutz, “Model-Driven Behavior Development for Service Robotic Systems: Bridging the Gap between Software- and Behavior-Models,” 2017. (unpublished work)

# Component-Definition Metamodel

A Component-Definition Metamodel is one of the core composition structures of RobMoSys.

The Component Metamodel (shown in the figure below) combines two complementary concerns namely **structure** and **interaction**. Individual blocks define the main entities of a component (including the component root element itself, highlighted with the yellow background color). For specifying **structure**, the blocks are connected using either the **contains** or the **has-a** relation (as defined in block-port-connector). For specifying **interaction**, the blocks are additionally connected using dedicated **ports**, **connectors** and **connections** (as also defined in block-port-connector). Moreover, two blocks (highlighted with the gray background color and dashed border-line) represent model elements that are defined in a separate metamodel (described in the next pages).



A component **contains** one **Parameter** structure, one **Lifecycle** state automaton and **has-a Behavior Interface**. The **Parameter** structure can be a Metamodel (or a DSL) by itself and the **Behavior Interface** allows run-time coordination and configuration from a higher robotics behavior coordination layer (see JOSER2016[1] for further details on both elements). The **Lifecycle** state automaton coordinates the different operational modes of a component. Some generic modes are for example *Init*, *Shutdown* and *Fatal-Error* (see TR2011[2] for more details).

The next core element of a Component is the **Activity** which is an abstract representation of a thread. A Component can define several Activities (depending on the component-internal functional needs). An Activity is independent of a certain thread realization and can be later mapped to a certain implementation by the selection of an according target platform. Moreover, an Activity provides a wrapper for the **Functions**. This is important for gaining control over execution characteristics of a component. This also considerably increases the flexibility (i.e. adjustability) of the component with respect to adapting the component to the different needs of various (at this point even unforeseen) systems.

A **Function** represents a functional block that can be designed using any preferred engineering methodology. From the component's internal point of view, a **Function** needs to be integrated into an**Activity** in order not to prematurely define any computational models that are not really relevant from the local functional point of view but might considerably restrict the compositionality of this component in different systems (see SIMPAR2016[3] for an example). In some cases, a**Function** might need to interact with specific hardware devices (such as e.g. sensors or actuators).

The last element of a Component is a **Service**. A Component can have several*required* and/or *provided* **Services**. A **Service** is the only allowed interaction point of a component to interact with other (not yet known) components. The definition of a service is described in a separate metamodel. Moreover, a**Function** interacts with the component's services over the surrounding **Activity** only. Again, this is important to gain control over execution characteristics as argued above.

See next:

- System Service Architecture Metamodel
- System Component Architecture Metamodel

See also:

- Service-Definition Metamodel
- Communication-Pattern Metamodel
- Component Development View

## References

1)

Dennis Stampfer, Alex Lotz, Matthias Lutz, Christian Schlegel. “The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software”. In *Journal of Software Engineering for Robotics (JOSER 2016)*, Link [http://joser.unibg.it/index.php/joser/article/view/91]

2)

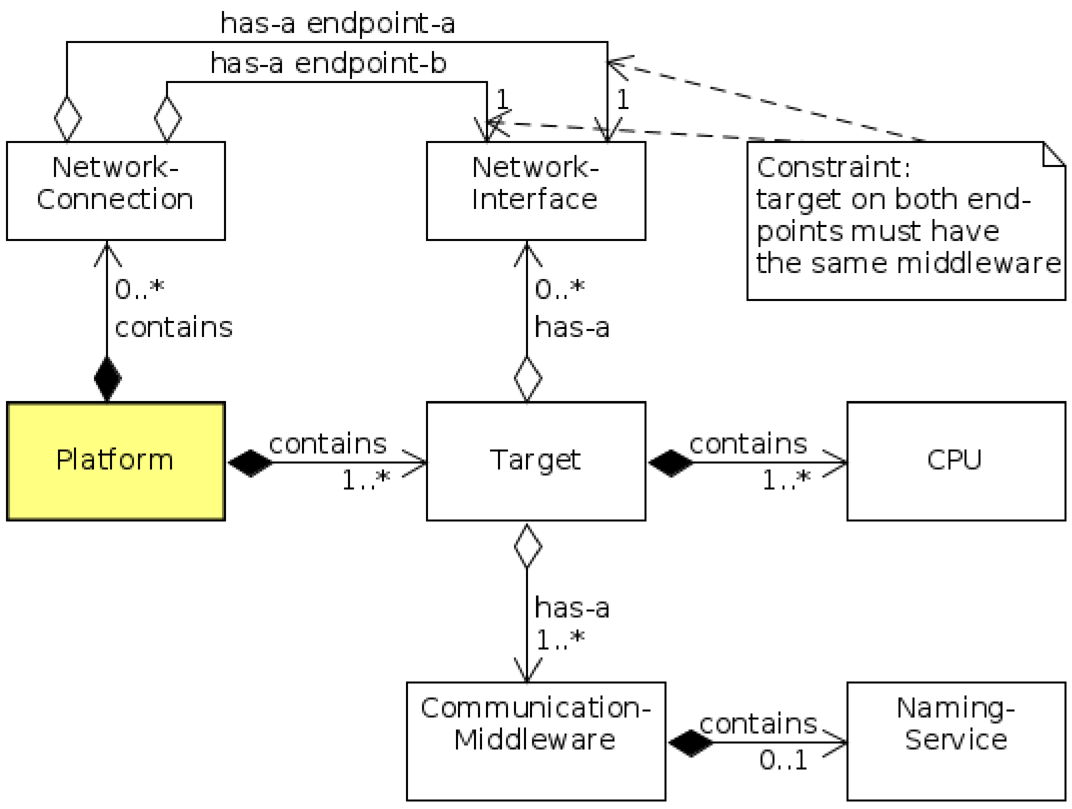
Christian Schlegel, Alex Lotz and Andreas Steck, “SmartSoft - The State Management of a Component”, in *Technical Report 2011/01*, Hochschule Ulm, Germany, ISSN 1868-3452, 2011.PDF [http://www.zafh-servicerobotik.de/dokumente/ZAFH-TR-01-2011-ISSN-1868-3452.pdf]

3)

Alex Lotz, Arne Hamann, Ralph Lange, Christian Heinzemann, Jan Staschulat, Vincent Kesel, Dennis Stampfer, Matthias Lutz, and Christian Schlegel. “Combining Robotics Component-Based Model-Driven Development with a Model-Based Performance Analysis”. In *IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2016)*. San Francisco, CA, USA, 2016.DOI [https://doi.org/10.1109/SIMPAR.2016.7862392]

# Platform Metamodel

The Platform Metamodel (see figure below) is one part of the overall RobMoSys Composition Structures. It defines the target platforms on the robot where the software components are later deployed to. The here described metamodel has no direct relation to the Digital Platform as described in the glossary. Please note that the current version of the Platform Metamodel is reduced to the most basic elements that are sufficient for deploying and executing software components. However, further versions of this metamodel might be extended to reveal additional details.



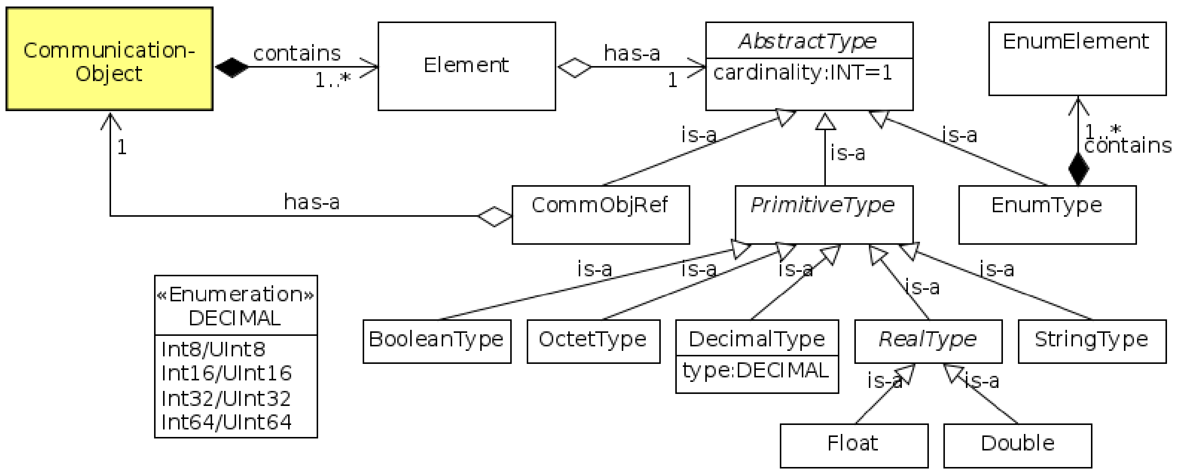
The two core elements of the platform meta-model are the targets and the network-connections. A target is basically a PC on the robot. Each target (i.e. a PC) has several CPUs and can have several network-interfaces. In addition, a target can use a specific communication-middleware (optionally with a middleware-specific naming-service). A network-connections links two network-interfaces and requires (as a constraint) that both connected targets use the same communication-interface (otherwise the components from the two targets would not be able to communicate).

See also:

- Deployment Metamodel

# Communication-Object Metamodel

Communication Objects define data structures that are communicated through services between components. The definition of communication objects requires primitive data types such as Int, Double, String, etc. and complex data types (i.e. composed data types). The figure below shows a simple metamodel of communication objects. A fully fledged communication objects modeling language that conforms to this metamodel is the SmartSoft communication object DSL [http://www.servicerobotik-ulm.de/toolchain-manual/html/ch02s02s02.html].



Typically, communication middlewares such as e.g. CORBA or DDS provide an *Interface Definition Language (IDL)* that allows specification of communication structures. RobMoSys requires a middleware-independent language. The SmartSoft communication object DSL [http://www.servicerobotik-ulm.de/toolchain-manual/html/ch02s02s02.html] provides a fully fledged Xtext-based language that is compliant to the metamodel in the figure above and that can be used already now for the definition of services.

At some point the communication object needs to be serialized (i.e. marshalled) into a middleware-specific representation. The following references provide details for how this can be achieved for a CORBA-based and a message-based middlewares:

- Christian Schlegel. “Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach”. *Dissertation*. University of Ulm, 2004PDF [http://www.hs-ulm.de/users/cschlege/_downloads/phd-thesis-schlegel.pdf]
- Christian Schlegel and Alex Lotz, “ACE/SmartSoft - Technical Details and Internals”, in *Technical Report 2010/01*, Hochschule Ulm, Germany, ISSN 1868-3452, 2010.PDF [http://www.zafh-servicerobotik.de/dokumente/ZAFH-TR-01-2010-ISSN-1868-3452.pdf]
- Dennis Stampfer, Alex Lotz, Matthias Lutz, and Christian Schlegel. “The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software”. In *Journal of Software Engineering for Robotics*, Special Issue on Domain-Specific Languages and Models for Robotic Systems, Vol 7, No 1 (2016). Link [http://joser.unibg.it/index.php/joser/article/view/91]

See next:

- Communication-Pattern Metamodel
- Service-Definition Metamodel

# Scientific Grounding

The highest abstraction level that is considered in RobMoSys is related to Hierarchical Hypergraphs and Entity-Relation models. The Entity-Relationship[1] model was one of the first approaches for formal "data base" models of knowledge.[2] It has gained renewed interest because of the rising popularity of the "Semantic Web"[3].

One of the main challenges is to represent context, more in particular, to deal with the combinatorial explosion in the number of relationships needed to represent – and interconnect – all relevant pieces of information and knowledge in multi-domain ICT and engineering systems. Such interconnected knowledge forms graph networks of links and properties. This fact poses difficulties to Lisp, Prolog, or other "programming languages" for Artificial Intelligence (AI), since they only have representations for relationship trees as first-class citizens. The same holds for the frame languages [https://en.wikipedia.org/wiki/Frame_language] [4] in AI, which considered "multiple inheritance" as a key feature. This last feature, together with "data encapsulation", are two major aspects of strict object oriented languages and models, that make "open world" [5][6] knowledge representations difficult; the SOLID [https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)] principles of object orientation better support knowledge representation, especially via its "D" feature, that is, the *Dependency inversion principle*, which states that one should "depend upon abstractions, not on concretions". However, none of these approaches offers infinitely composable knowledge representations, because they only partially support the essential features outlined in the sections below.

## Hierarchical Hypergraph

The modern, higher-order, version of the Entity-Relationship model is that of a hierarchical (property) hypergraph[7][8]:

- hierarchical : every node and every edge can be a full graph in itself. In other words, any Relation can be considered an Entity in itself, and can hence be used as an argument in another, higher-order Relationship.
- hypergraph: every edge can connect more than two nodes; that is, it is an n-ary "hyperedge"
- property meta data: every node and every edge in the graph has a property data structure attached to it; two (mandatory) parts of those properties are the following meta data:
  - unique node/edge identifier : other relationships in the graph can refer to this node or edge.
  - meta model identifier : each node or edge has a type, indicated by the unique identifier of the graph that models that type.

Often used synonyms for the term "Entity" are: object, concept, atom, primitive. "Relationships" are also called: rules, axioms, constraints, links, expressions. Often used extra meta data is the so-called provenance of a model: who made it? when? what version is it? Etcetera. State of the art formal meta models to represent such provenance are W3C provenance[9], and Dublin Core[10].

## Entity-Relation Model

Each "thing" to be modelled will have a number of data structures that represent its properties. That can be done via (possibly nested) key-value pairs, with each key having, a type, a unique identifier (with which Relationships can refer to it), and a role to play in the "thing" properties. While efficient implementations of those properties can be realised with the rich data structure primitives in computer programming langauges, the meaning of such properties, as just described above, is a hierarchical hypergraph.

A Relationship between Entities is a named directed graph, representing the Role that each Entity plays as an Argument in the Relationship:

- the top node carries the meta data of the Relationship, of which the two major ones are: (i) its unique "identifier " (with which other Relationships can refer to it), and (ii) the context (all the externally defined Entities and Relationships whose names are being used in the model of this Relationship). Other meta data in the top node are: type and provenance. In addition to the identifier (which in principle should only be computer-readable), models often carry human-readbale names and description strings, possibly in various languages. However, these are not used in linking Entities together into Relationships.
- from the top node, there are Role edges towards each of the Entity nodes that figure as Arguments in the relationship. Each Role edge also has similar meta data properties as the top node, but the most distinguishing one represents the purpose ("role") of a particular argument in the Relationship. This is formally represented by a specific Relationship in itself.

Each "value" in an Argument has a domain (or "universe of discourse"): the type and the set(s) of possible values that the "key" can have. In other words, that domain brings its own property data structure to each argument. Remark the recurring pattern of "identifiers", "types" and "contexts", in the nodes and edges of a hierarchical hypergraph. And also remark that the graph is directed : pointing from the Relationship to the Entities, and down to the latters' proeprties.

## Natural modelling levels of abstraction

"Abstraction" is a key concept in modelling, but it is hard to define axiomatically. Below, three core "meta meta" forms of modelling are described[11]:

- **mereology** – parts: there is already quite some (mature) formalisation available in the state of the art, to structure "Entities'; for example, the Wikipedia article [https://en.wikipedia.org/wiki/Mereology] in the subject has a good overview and pointers to the literature. The key Relationship is has-a (also called, "has-part" or similarly equivalent names), and is-equal.
- **topology** – structure of interconnections between parts: this kind of structural model is a key property of any system, and also here the state of the art insights and formalizations are sufficiently mature to have unambiguous and consistent semantics of formal models, to the extent that it is realistic to develop "standards" and "tools".

Concretization (or specialization) can be considered as the opposite of abstraction. In this sense, raising the level of abstraction means to get more general purpose while lowering the level of abstraction means to get more specific with respect to e.g. a certain domain. It is only natural that the general purpose (i.e. higher) abstraction levels tend to leave open some semantic variability. For instance, UML (as one representative for general-purpose modeling languages) purposefully defines *"semantic variation points"*. These "semantic variation points" can be fixed by e.g. deriving domain-specific models (in terms of UML by defining UML profiles). In this sense, RobMoSys as well defines several levels of abstractions, with "Hierarchical Hypergraphs" and "Entity-Relation" levels on top, over "Block-Port-Connector" and "RobMoSys composition structures" and down to concrete realizations (sometimes "reference implementations"). Going gown this abstraction hierarchy also means getting more domain-specific and narrowing semantic variability.

---

# Formalization

This section provides formal specifications for the Hierarchical Hypergraphs and for an Entity-Relation model.

## Hierarchical Hypergraph

- “a hypergraph H is a pair H = (X,E) where X is a set of elements called nodes or vertices, and E is a set of non-empty subsets of X called hyperedges or edges” Wiki:Hypergraph [https://en.wikipedia.org/wiki/Hypergraph]
- hyperedge: each vertex in the graph can connect more than two nodes
- hierarchy: each node or edge in the graph can be a full graph in itself

## Entity-Relation Model

Entity-Relation is a specialization of a Hypergraph. Therefore, Entity-Relation **conforms-to** a Hypergraph.

- **entity**
  - the “things”
  - entity instantiates a node of its meta-model
    - uniquely referencing an element of its meta-model
  - entity has a unique identifier
    - uniquely referencing this primitive
- **relation**
  - n-ary link between primitives
  - relation instantiates a hyper-edge of its meta-model
    - uniquely referencing an element of its meta-model
  - relation has a unique identifier
    - uniquely referencing this relation
- **property**
  - attribute of a primitive or a relation

## Basic set of standard relations for linking different levels of abstraction

We do not introduce a RobMoSys specific definition for these relations. Instead, we just use their “common sense definition”. The following explanations are just typical “common sense descriptions”:

- **is-a**
  - this is inheritance
  - an element of a model derives from an element of a metamodel
- **instance-of**
  - this is often just a synonym for “is-a”
  - one talks of an instance when it is the final element in an inheritance hierarchy. What is considered a final element depends on what parts of the inheritance hierarchy you see.
- **conforms-to**
  - a meta-model is a model that defines the language for expressing a model. A model represents an abstracted representation of an artefact. A model conforms to a meta-model. One model can have multiple models to which it conforms.
- **constraints**
  - this is a particular relation
  - it can be applied to primitives, relations and properties

See next:

- Block-Port-Connector

# References

1)

P. P.-S. Chen. The entity-relationship model—Toward a unified view of data. ACM Transactions on Database Systems, 1(1):9–36, 1976.

2)

At more or less the same time, similar developments took place around knowledge representations via “programming languages”, such as Lisp or Prolog.

3)

T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. Scientific American, 284(5):34–43, 2001.

4)

M. L. Minsky. A framework for representing knowledge. In P. H. Winston and B. Horn, editors, The psychology of computer vision. 1975.

5)

R. Reiter. On closed world data bases. In Logic and Data Bases, pages 55–76. 1978.

6)

S. J. Russell and P. Norvig. Artificial Intelligence: A Modern Approach. Prentice Hall, 3rd edition, 2009.

7)

G. Engels and A. Schürr. Encapsulated hierarchical graphs, graph types, and meta types. Electronic Notes in Theoretical Computer Science, 2:101–109, 1995.

8)

M. Levene and A. Poulovassilis. An object-oriented data model formalised through hypergraphs. Data & Knowledge Engineering, 6:205–224, 1991.

9)

W3C. An overview of the prov family of documents.https://www.w3.org/TR/prov-overview/ [https://www.w3.org/TR/prov-overview/], 2013.

10)

Dublin Core Metadata Initiative. Dublin core metadata element set.http://dublincore.org/documents/dces/ [http://dublincore.org/documents/dces/].

11)

P. Borst, H. Akkermans, and J. Top. “Engineering ontologies”.*International Journal on Human-Computer Studies*, 46:365–406, 1997.

modeling:hypergraph-er · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/modeling:hypergraph-er

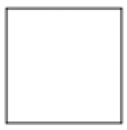
# Block-Port-Connector

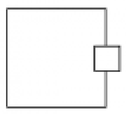
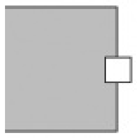
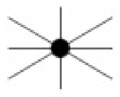
The Block-Port-Connector model is a specialization of the more abstract Hypergraph and Entity-Relation model.

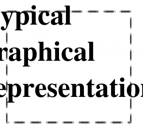
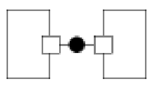
The following generic **relations** have been introduced already:**is-a**, **instance-of**, **conforms-to** and **constraints**. There are two additional (i.e. more specific) relations that need to be introduced:

| **Relation** | **Explanation** | **Typical graphical representation** | **Typical textual representation** |
|---|---|---|---|
| **contains** | # can be applied to entities and can be applied to relations<br>* this realizes hierarchical composition (nested composition); in a hierarchical composition elements are enclosed by another element<br>* contains is topology<br>* the contained elements are not accessible/visible (in contrast to elements in a collection)<br>* the contained elements can or cannot exist without the parent (depending on the context) | an arrow with a diamond (filled with black color for ownership or white color for no ownership) | contains(A,a,b,c)<br>contains(B,m,n) |
| **has-a** | # can be applied to entities and can be applied to relations<br>* this realizes aggregation<br>* has-a is mereology<br>* in aggregation, elements remain at the same level<br>* elements linked with has-a remain aceesible/visible<br>* the contained elements can or cannot exist without the parent (depending on the context) | an arrow with a diamond (filled with black color for ownership or white color for no ownership) | has-a(A,a,b) |

The generic **entity** is refined as follows:

| **Entity/Relation** | **Model and Description** | **Typical graphical representation** | **Typical textual representation** |
|---|---|---|---|
| **block** | **Model:**<br>* is-a **entity**<br>* possibly has-a property (or many)<br>* possibly has-a port (or many)<br>* possibly contains property (or many) |  | block(block-A) |

| Entity/Relation | Model and Description | Typical graphical representation | Typical textual representation |
|---|---|---|---|
| | * possibly contains block (or many)<br>* possibly contains collection (or many)<br>* possibly contains connector (or many)<br>* possibly contains relation (or many) | | |
| | **Description:**<br>the only interaction points of a block are ports | | |
| **port** | **Model:**<br>* is-a **entity**<br>* has-a internal dock<br>* has-a external dock |  | port(Port-A) |
| | **Description:**<br>it is the only interaction point over which a block can interact with other blocks;<br>when attached to a block, the internal dock becomes a private to the block (contains) and the external dock becomes public (has-a) | | |
| | **Comment:**<br>In textual representation, access to docks can be represented e.g. like internal-dock(Port-A), external-dock(Port-A) | | |
| **dock** | **Model:**<br>* is-a **entity** |  | dock(Dock-A) |
| | **Description:**<br>A dock is used to semantically differentiate between the “internal” and “external” sides of a port with respect to the port's parent block. | | |
| | **Comment:**<br>In a graphical representation, the internal dock and the external dock can be highlighted, for example by different colors (be careful, not to start an irrelevant activity in introducing such graphical notions into existing tools which cannot handle that). | | |
| **connector** | **Model:**<br>* is-a **entity**<br>* connects ports (n-ary relation) |  | connector(connector-A) |
| | **Description:**<br>can connect ports as long as no block boundaries are crossed | | |
| | **Comment:**<br>In graphical representation, the connector itself is represented by a dot. With the connects-relation, star-shaped lines (connects-relations) originate from the dot in the center. | | |
| **collection** | **Model:**<br>* is-a **entity**<br>* possibly has-a entity (or many)<br>* possibly has-a relation (or many) | | collection(collection-C,k,l,m,n) |

| Entity/Relation | Model and Description | Typical graphical representation | Typical textual representation |
|---|---|---|---|
| | **Description:**<br>A collection can group any combination of entities and / or relations. The enclosement formed by a collection is just a virtual one where the elements are openly accessible (in contrast to nesting).<br>A collection can pick any elements out of blocks ignoring block boundaries ⇒ this is particularly useful to specify modeling views |  | |
| | **Comment:**<br>In the graphical representation, the dotted box can enclose entities and / or relations where you can cross the dotted line to "enter" the collection | | |
| **connects** | **Model:**<br>is-a **relation** |  | connects(connector-A,external-dock(Port-A)) |
| | **Description:**<br>links a dock of a port to a connector (binary relation) | | |

There is a specific relation between the RobMoSys composition structures and the modeling views as is discussed on the next page. The important point at this level here is to provide a base-level that allows specification of both kinds. The specific part for specifying modeling views is the **collection** definition while all the other specifications are used to define the RobMoSys composition structures.

Please note that while **blocks** and **ports** are semantically different, depending on the current role-specific view with according level of abstraction, ports can contain additional structures and thus might appear as blocks on that detailed abstraction level (see service-definition metamodel).

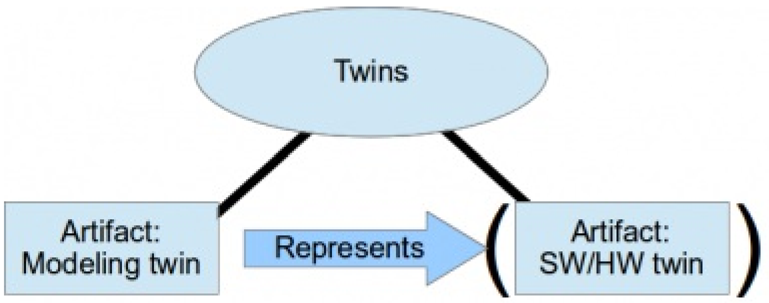
## See next

- RobMoSys composition structures

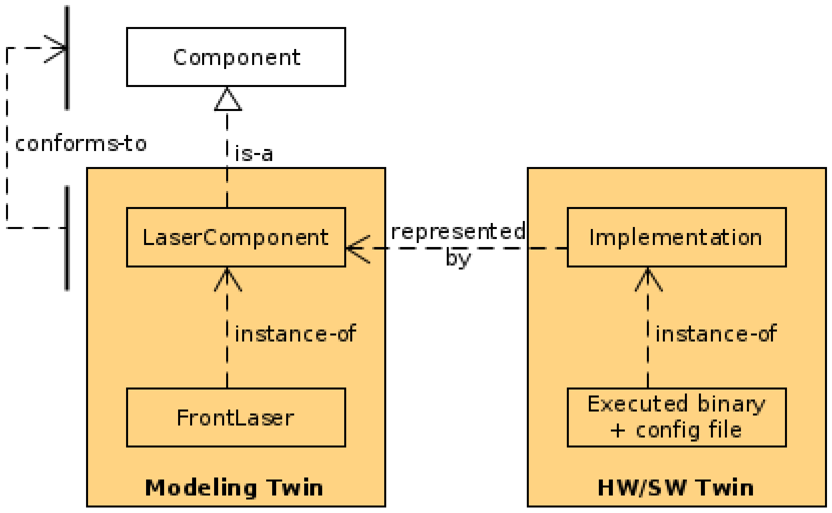
modeling:principles:block-port-connector · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/modeling:principles:block-port-connector

# Modeling Twin

All entities in the market and all entities that are shared in the ecosystem come as twins. Twins consist of a model (modeling twin) that represents the Software or Hardware artifact (SW/HW twin). Think of the modeling twin as a bridge between traditional software artifacts and the modeling world. The modeling twin is similar to data sheets in the PC Analogy.



The modeling twin is always supplied and handed over between roles in the ecosystem. The SW/HW twin might be supplied later or might not exist at all. It might not exist, for example, when the artifact is purely intended for modeling. Entities in the market will never be just HW/SW artifacts without a modeling twin as then the artifact cannot be used. One can continue building a system independently with only the modeling twin, then supplying the HW/SW twin later.



The modeling twin is a representative and abstraction of the artifact it represents. It explicates necessary properties to work with it. Supplying a modeling twin does not equal to exposing all details: IP can still be protected as the modeling twin only have to expose the information that is relevant to use it: internal structures can remain hidden.

The modeling twin is is similar to the "digital twin"[1] in IoT and industry 4.0. It, however, is beyond bridging the physical world to the digital world: it focuses on having a representative of physical entities or software entities for modeling purposes.

## See also

- PC Analogy

1)

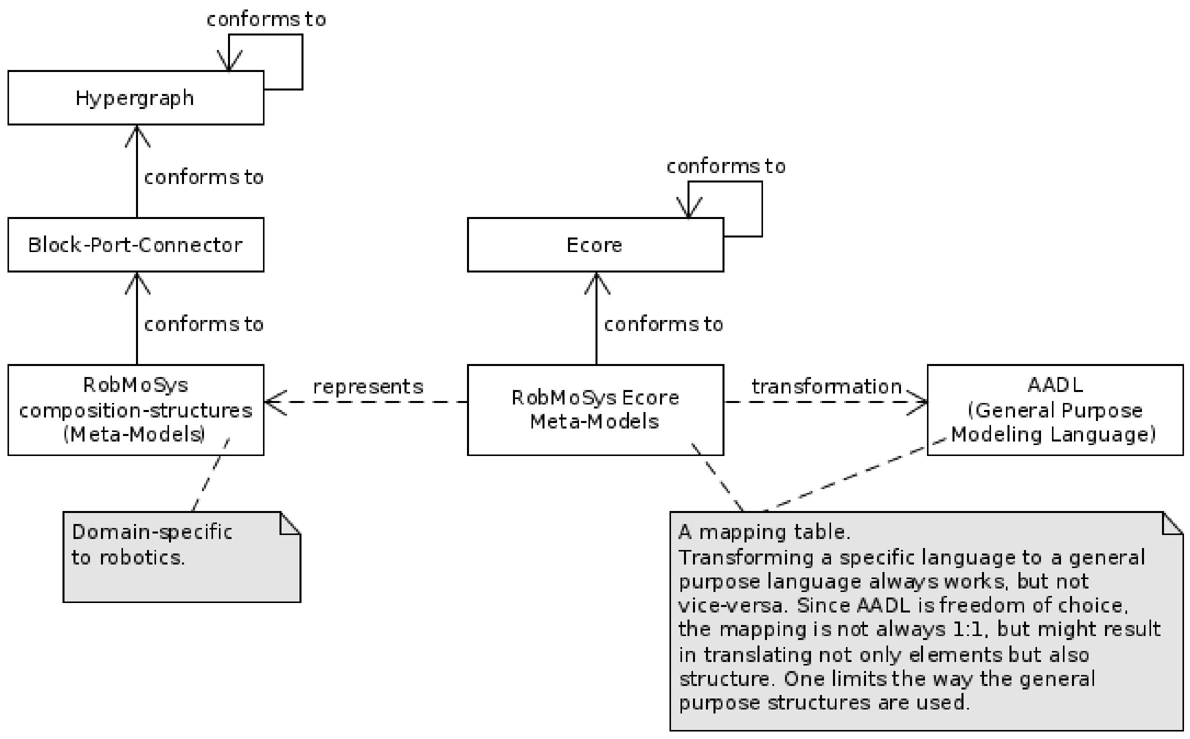
Dr. Michael Grieves and John Vickers. “Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems (Excerpt)”, Excerpted based on: Trans-Disciplinary Perspectives on System Complexity. Online [http://research.fit.edu/camid/documents/doc_mgr/1221/Origin%20and%20Types%20of%20the%20Digital%20Twin.pdf

# RobMoSys Structures: Realization Alternatives

This page describes alternatives for realizing the RobMoSys Composition Structures. This list of alternatives shows examples and is not meant to be complete.

## Example 1: Using Ecore

A meta-model is an abstract representation of a model. A meta-model in itself can be considered as a model that may or may not have an even more abstract representation (i.e. a meta-meta-model). There are no theoretical limits for going up the abstraction hierarchy. However, from a practical point of view, at a certain abstraction level it simply does not make much sense to go further up the hierarchy. Instead, there often is a meta-level that is abstract enough to define its own language. Example languages for such a level are: Eclipse Ecore and Essential MOF (EMOF). Nevertheless, it might make sense to go higher up the abstraction hierarchy above Ecore in order to define meta-levels that ease interfacing between the different realization technologies. Such a higher meta-level is for instance the Hypergraph notation. The relation between e.g. the Ecore based meta-models and the more abstract meta-levels is depicted in the figure below.

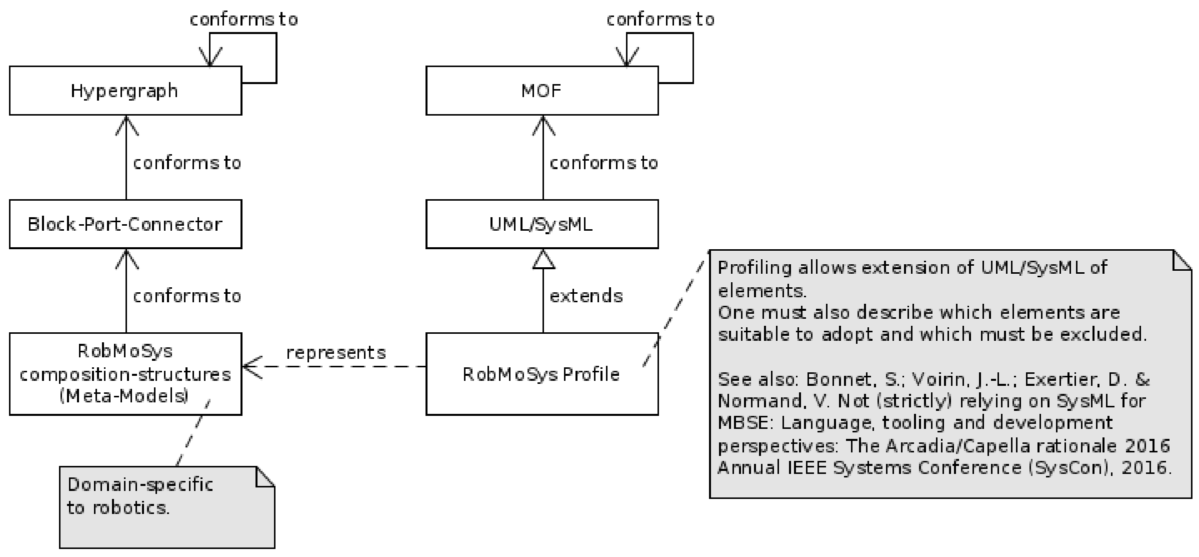

conforms to
Hypergraph
conforms to
Block-Port-Connector
conforms to
RobMoSys composition-structures (Meta-Models)
Domain-specific to robotics.
represents
conforms to
Ecore
conforms to
RobMoSys Ecore Meta-Models
transformation
AADL (General Purpose Modeling Language)
A mapping table.
Transforming a specific language to a general purpose language always works, but not vice-versa. Since AADL is freedom of choice, the mapping is not always 1:1, but might result in translating not only elements but also structure. One limits the way the general purpose structures are used.

The left side of the figure shows a meta-level hierarchy starting with a Hypergraph on top, over Blocks-Ports-Connectors and down to RobMoSys composition structures. This hierarchy allows formal definition of meta-levels for the required structures independent of a particular realization technology. In the middle of the figure, a specific realization technology (in this case Ecore) is used to implement the RobMoSys meta-models. This is only an example and many other technologies can be used instead in a similar fashion. Moreover, other existing modeling languages (such as AADL) can be easily interlinked with the RobMoSys structures by defining model-to-model transformations. This is a powerful extension mechanism that allows usage of matured and powerful tools in robotics.

In the course of the project, RobMoSys is going to provide an Ecore implementation of the RobMoSys structures.

A preliminary implementation of Ecore meta-models for the two topmost abstraction levels within Tier 1 (on the left in the figure above), namely the Entity-Relation and Block-Port-Connector meta-models, is available at Preliminary Ecore implementation of ER and BPC meta-models

## Example 2: Using UML/SysML Profiling

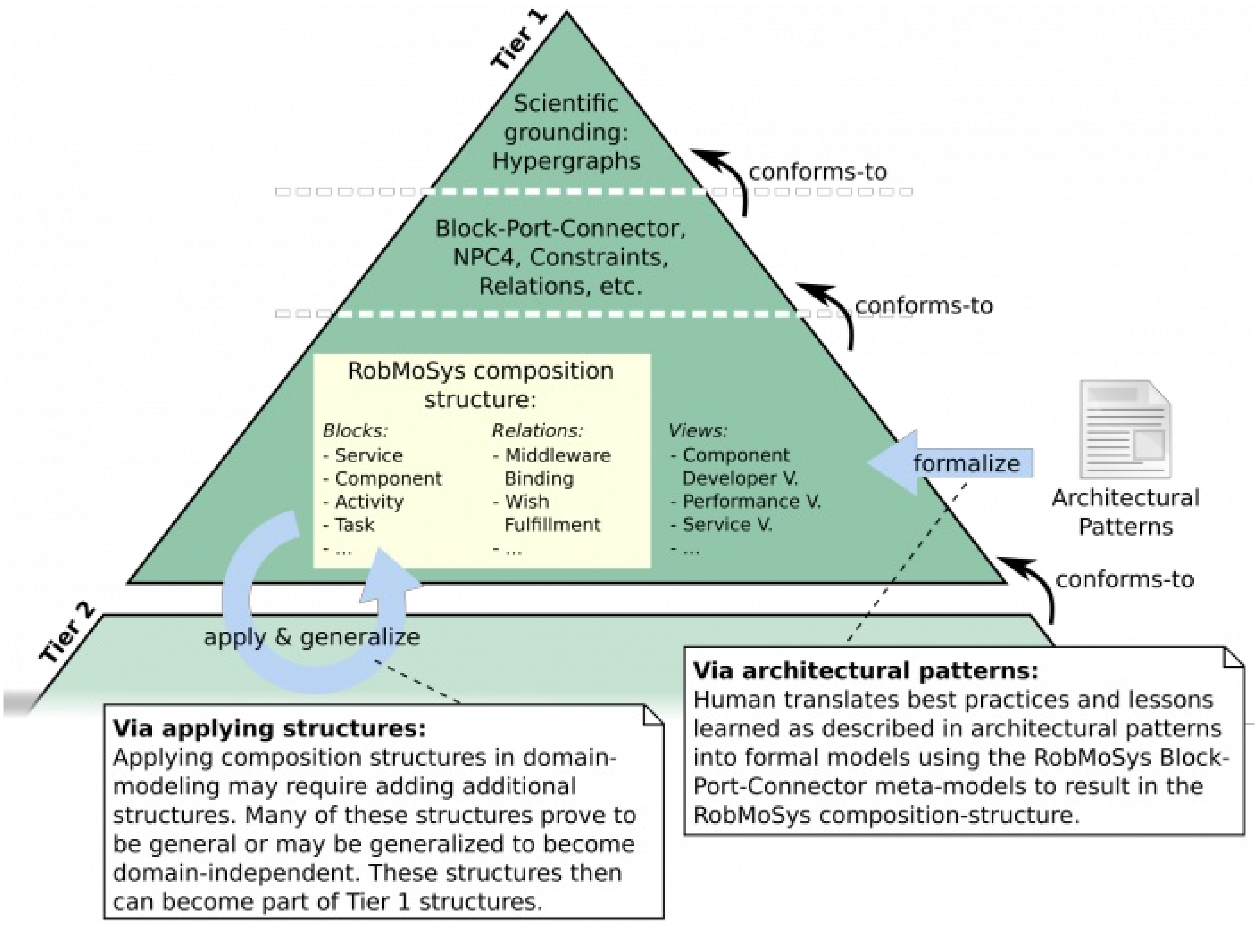

conforms to
Hypergraph
conforms to
Block-Port-Connector
conforms to
RobMoSys composition-structures (Meta-Models)
Domain-specific to robotics.
conforms to
MOF
conforms to
UML/SysML
extends
represents
RobMoSys Profile
Profiling allows extension of UML/SysML of elements.
One must also describe which elements are suitable to adopt and which must be excluded.
See also: Bonnet, S.; Voirin, J.-L.; Exertier, D. & Normand, V. Not (strictly) relying on SysML for MBSE: Language, tooling and development perspectives: The Arcadia/Capella rationale 2016 Annual IEEE Systems Conference (SysCon), 2016.

The figure above shows another example of using a different realization technology, in this case the UML/SysML and MOF as base structures. The RobMoSys structures on the left are unaffected by this different technology choice. It is worth mentioning that while the UML standard also specifies the graphical notation, the extension mechanism through profiling might be a bit more challenging when it comes to restricting the already defined modeling structures. These pros and cons need to be traded off when choosing a modeling technology.

modeling:realization_alternatives · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/modeling:realization_alternatives

# RobMoSys Composition Structures

The RobMoSys composition structures is a bottom abstraction layer on Tier 1 (see figure below). This layer defines all the robotics meta-structures that are required to consistently model robotic systems throughout several development phases and thereby supporting different developer roles. The meta-structures follow a general composition-oriented approach where systems can be constructed out of reusable building blocks with explicated properties. In other words, RobMoSys enables the composition of robotics applications with managed, assured and maintained system-level properties via model-driven techniques. This enables communication of design intent, analysis of system design before it is being built and understanding of design change impacts. Therefore, the RobMoSys composition structures adhere to the general block-port-connector meta-structures and can be considered as a further specialization thereof.

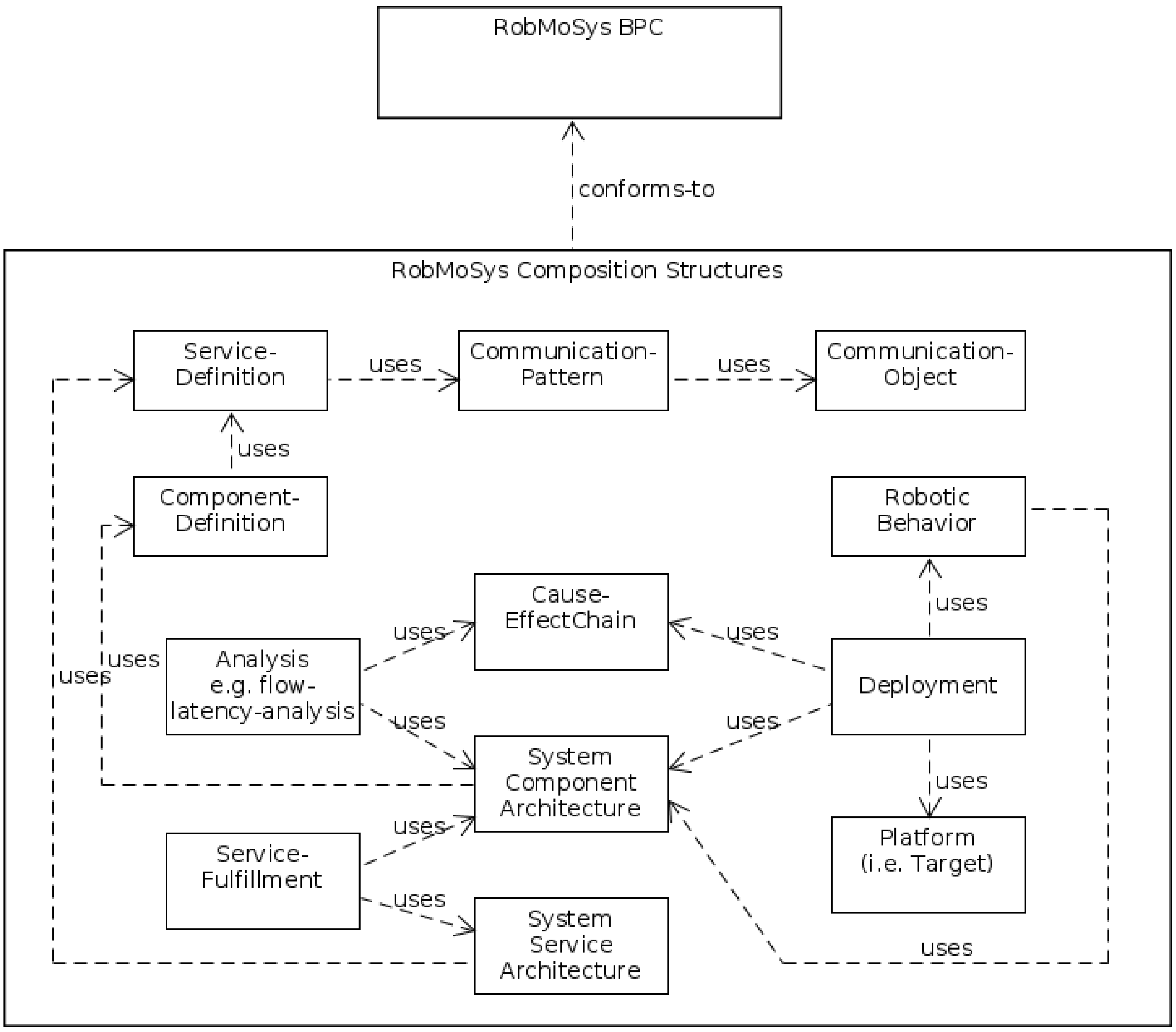

Tier 1
Scientific grounding: Hypergraphs
conforms-to
Block-Port-Connector, NPC4, Constraints, Relations, etc.
conforms-to
RobMoSys composition structure:
Blocks:
- Service
- Component
- Activity
- Task
- ...
Relations:
- Middleware Binding
- Wish Fulfillment
- ...
Views:
- Component Developer V.
- Performance V.
- Service V.
- ...
formalize
Architectural Patterns
conforms-to
Tier 2
apply & generalize
Via architectural patterns:
Human translates best practices and lessons learned as described in architectural patterns into formal models using the RobMoSys Block-Port-Connector meta-models to result in the RobMoSys composition-structure.
Via applying structures:
Applying composition structures in domain-modeling may require adding additional structures. Many of these structures prove to be general or may be generalized to become domain-independent. These structures then can become part of Tier 1 structures.

The figure (above) shows an exemplary list of possible composition structures (highlighted with the yellow background color), which can be clustered into (a) specializations of **blocks** and (b) specializations of **relations**. One of the central structures defined by RobMoSys is a consistent and rich enough **component model** that considers the interaction with related structures around the component model (such as e.g. the definition of communication services and the binding to different middlewares). These structures are described below in separate pages. An interesting point is that RobMoSys by purpose does not aim at one huge meta-model that covers all robotics aspects at once. Instead, RobMoSys foresees the definition of modeling views that cluster related modeling concerns in dedicated views, while at the same time connecting several views in order to be able to define model-driven tooling that supports the design of consistent overall models and to communicate the design intents to successive developer roles and successive development phases. In this sense, composition does not only apply to designing robotics software but is also applied to designing the modeling tools, thus making them easily extensible and composable.

Are you new to model-driven engineering? Find introduction literature in the FAQ.

## Overview of RobMoSys composition structures

The figure below provided an overview of the RobMoSys composition structures (i.e. the **RobMoSys Metamodels**). Each block in the figure represents a separate Metamodel that is individually described in a separate page (see below). There are high-level relations between the metamodels that are depicted with the **uses** keyword.



The next pages individually describe the RobMoSys metamodels in a human-readable notation using the general definitions of block-port-connector. There is a straightforward way to transform this representations using a dedicated modeling technology as described here.

Each metamodel (presented next) addresses two main modeling needs namely **structure** and **interaction**. **Structure** defines the structural relations (such as**has-a** and **contains**) between the individual model elements. Structure can often be directly translated into a modeling technology such as Ecore. **Interactions** define the important interaction relations (using **port**, **connector** and **connects**) between specific model elements. In contrast to structure, interactions are often transformed into software APIs (e.g. through code generation) and must not always be visible on model level.

## List of Metamodels

- Robotic Behavior Metamodel
- Communication-Object Metamodel
- Communication-Pattern Metamodel
- Component-Definition Metamodel
- Deployment Metamodel
- Cause-Effect-Chain and its Analysis Metamodels
- Platform Metamodel
- System Service Architecture and Service Fulfillment Metamodels
- Service-Definition Metamodel
- System Component Architecture Metamodel

See also:

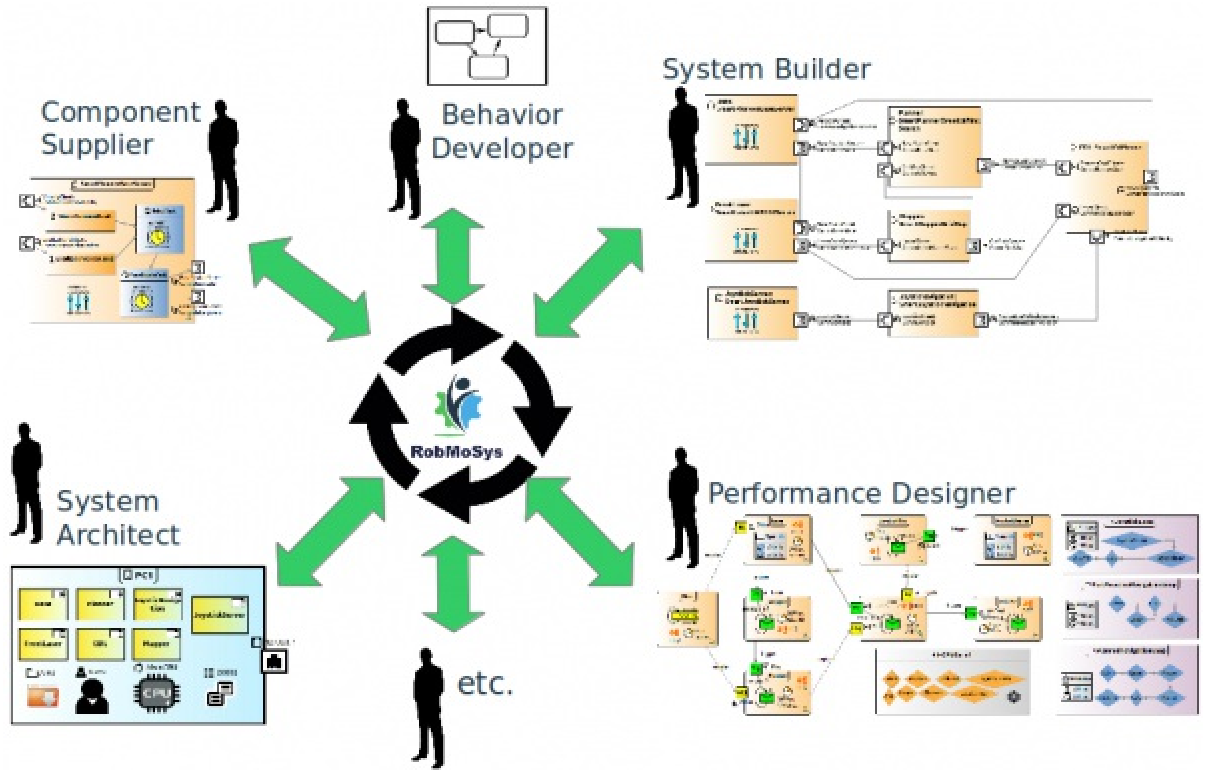
- RobMoSys Views

---

modeling:composition-structures:start · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/modeling:composition-structures:start

# RobMoSys Views

Roles in the Ecosystem come with specific views. The benefit of a view to a role is to present only what is relevant for the role's responsibility, thereby hiding the complexity that is not relevant for that role, but is still relevant for the whole system in the end. The system in the end consists of many concrete models based on the RobMoSys Composition Structures. These elements are contributed by roles that work through views and interact such that the contributed elements are composable to form a system. As a result, the individual role can focus on its responsibility and expertise alone, while working decoupled from other roles. This is enabled by the RobMoSys Composition Structures.



Each role that participates in the ecosystem uses a dedicated view to focus on its responsibility and expertise.

The concept of “views” groups basic primitives of the RobMoSys composition strucures. A view is related to a role and establishes the link between primitives in the RobMoSys composition strucures and the RobMoSys roles.

A role has a specific view on the system at an adequate abstraction level using relevant elements only. A view is not only in the sense of a perspective where one only sees a part of the system but does not see the rest, even if it is there. Instead, a view shows an excerpt of the whole system that can be viewed independently of the other parts. These other parts might even not exist at the time of having the view on the system, because it is composed to other parts to form the complete system later.

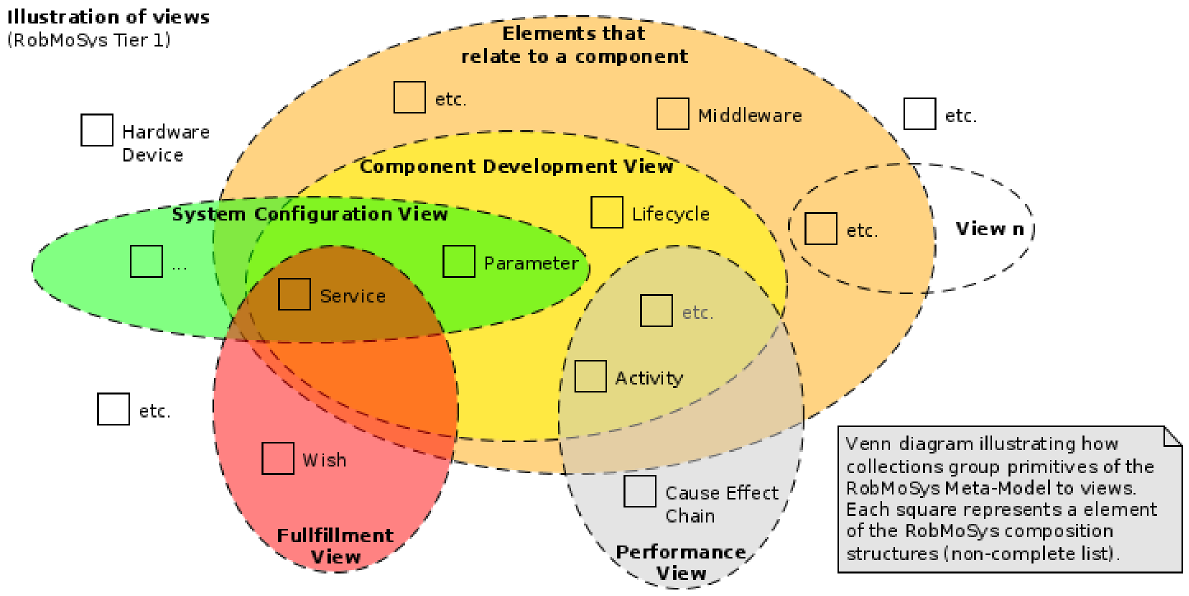


Illustration of views
(RobMoSys Tier 1)
Elements that relate to a component
etc.
Middleware
etc.
Hardware Device
Component Development View
System Configuration View
Lifecycle
etc.
View n
...
Parameter
Service
etc.
Activity
etc.
Wish
Cause Effect Chain
Fullfillment View
Performance View
Venn diagram illustrating how collections group primitives of the RobMoSys Meta-Model to views. Each square represents a element of the RobMoSys composition structures (non-complete list).

Example: Consider a closed book. The view of a front cover is a certain perspective on the book. Even though only the front cover is visible, the whole book is lying there. The book also consists of its pages and the back cover which are not visible, even if they are there. It, however, makes perfect sense to only look at the back cover of a book, its content pages or even the individual chapters separately (an excerpt of the book) as both the front page and the back page can be designed differently (separation of roles) and then be put together.

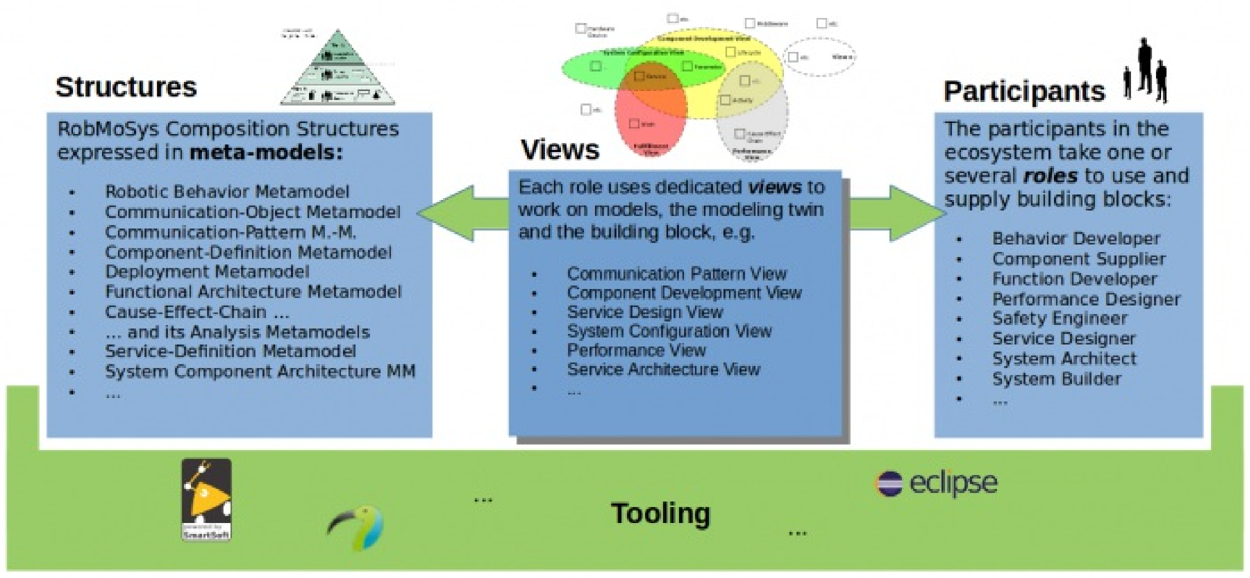
## List of Views

(alphabetical order)

- Communication Pattern View
- Component Development View
- Execution Container View
- Service Design View

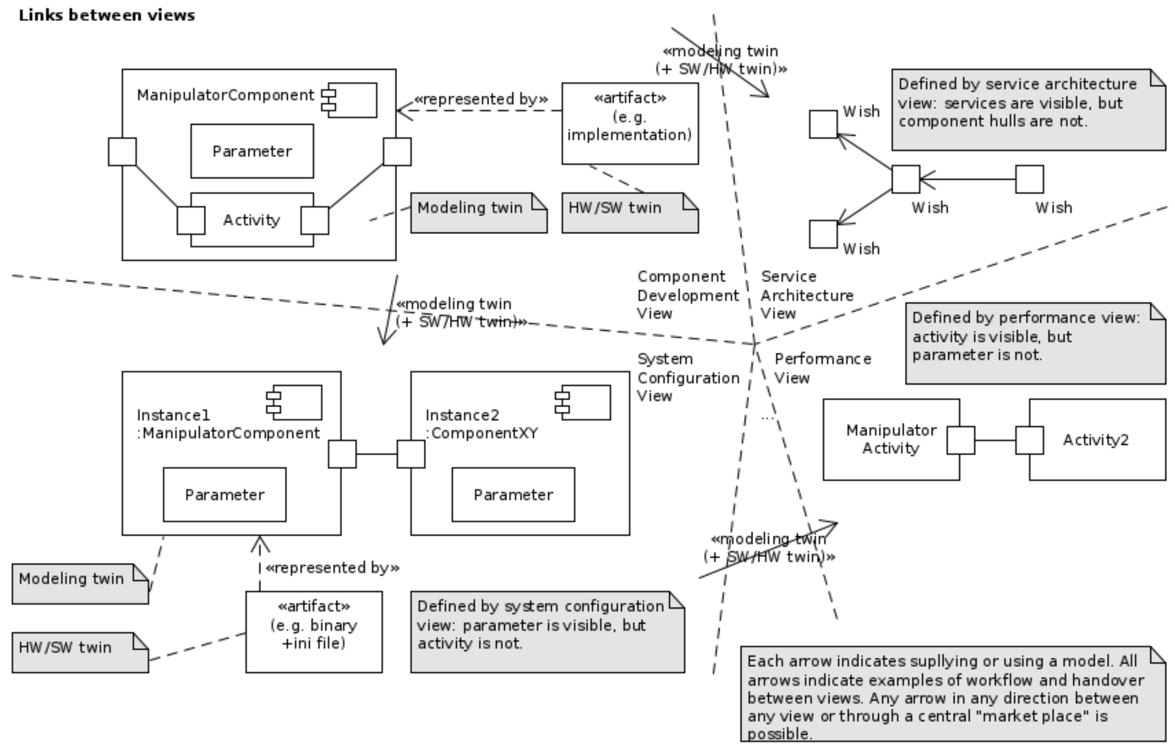
- Deployment View
- Service Architecture View
- Service Fulfillment View
- …

## Views in relation to composition structures and roles


Structures
RobMoSys Composition Structures expressed in meta-models:
Robotic Behavior Metamodel
Communication-Object Metamodel
Communication-Pattern M.-M.
Component-Definition Metamodel
Deployment Metamodel
Functional Architecture Metamodel
Cause-Effect-Chain ...
... and its Analysis Metamodels
Service-Definition Metamodel
System Component Architecture MM
...
Views
Each role uses dedicated views to work on models, the modeling twin and the building block, e.g.
Communication Pattern View
Component Development View
Service Design View
System Configuration View
Performance View
Service Architecture View
...
Participants
The participants in the ecosystem take one or several roles to use and supply building blocks:
Behavior Developer
Component Supplier
Function Developer
Performance Designer
Safety Engineer
Service Designer
System Architect
System Builder
...
Tooling
eclipse
...
...

## Links Between Views: Example 1

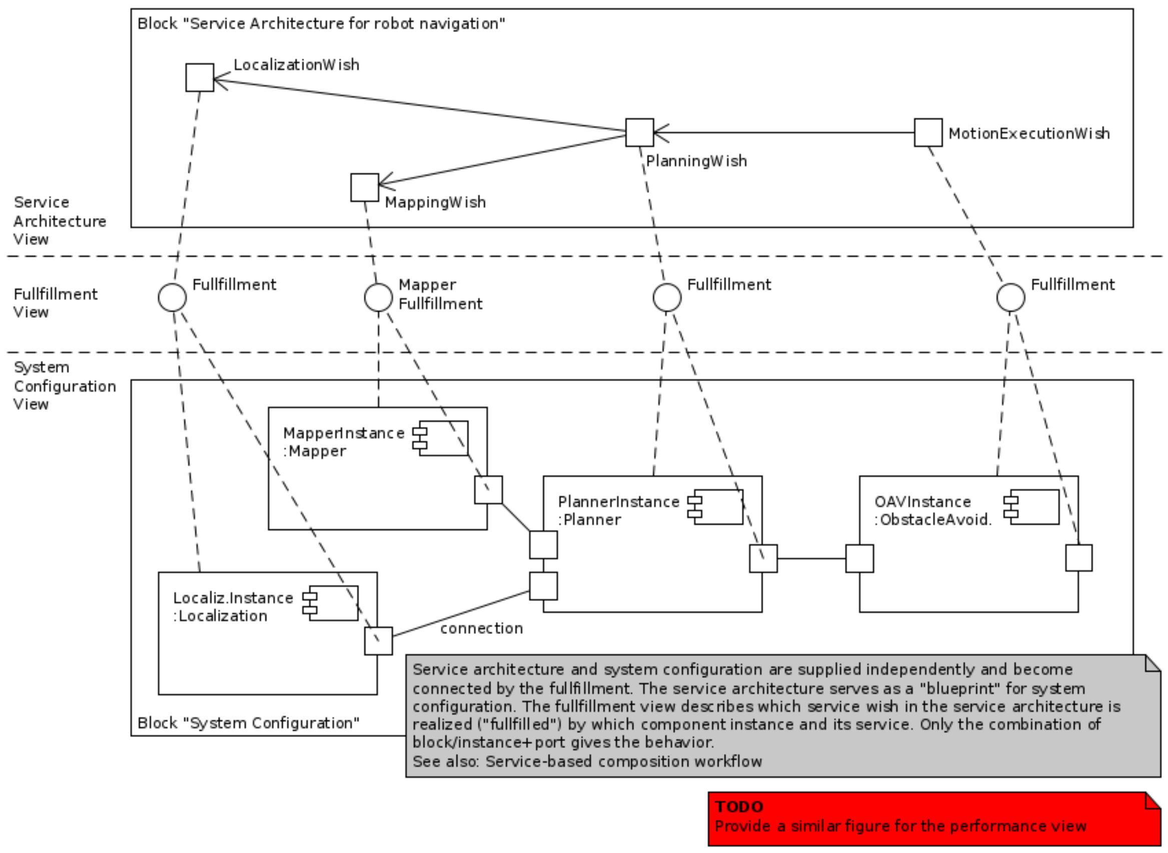
The figure below illustrates the link between several views. The Modeling Twin is handed over between one view to the next. There is no strict order in the sense of a strictly order value chain. Instead, the interactions form a network of collaborating roles consisting of various bilateral interactions between suppliers and consumers.


Links between views
ManipulatorComponent
Parameter
Activity
«represented by»
«artifact» (e.g. implementation)
Modeling twin
HW/SW twin
«modeling twin (+ SW/HW twin)»
Defined by service architecture view: services are visible, but component hulls are not.
Wish
Wish
Wish
Wish
Component Development View
Service Architecture View
«modeling twin (+ SW/HW twin)»
Defined by performance view: activity is visible, but parameter is not.
System Configuration View
Performance View
Instance1 :ManipulatorComponent
Parameter
Instance2 :ComponentXY
Parameter
Manipulator Activity
Activity2
...
Modeling twin
«represented by»
«artifact» (e.g. binary +ini file)
HW/SW twin
Defined by system configuration view: parameter is visible, but activity is not.
«modeling twin (+ SW/HW twin)»
Each arrow indicates supllying or using a model. All arrows indicate examples of workflow and handover between views. Any arrow in any direction between any view or through a central "market place" is possible.

## Links Between Views: Example 2

The figure below illustrates an example where two views are connected by a third view. The service architecture can serve as a blueprint for system configuration.

**Two views are linked by a third view**



Service architecture and system configuration are supplied independently and become connected by the fullfillment. The service architecture serves as a "blueprint" for system configuration. The fullfillment view describes which service wish in the service architecture is realized ("fullfilled") by which component instance and its service. Only the combination of block/instance+port gives the behavior.
See also: Service-based composition workflow

**TODO**
Provide a similar figure for the performance view

## See also

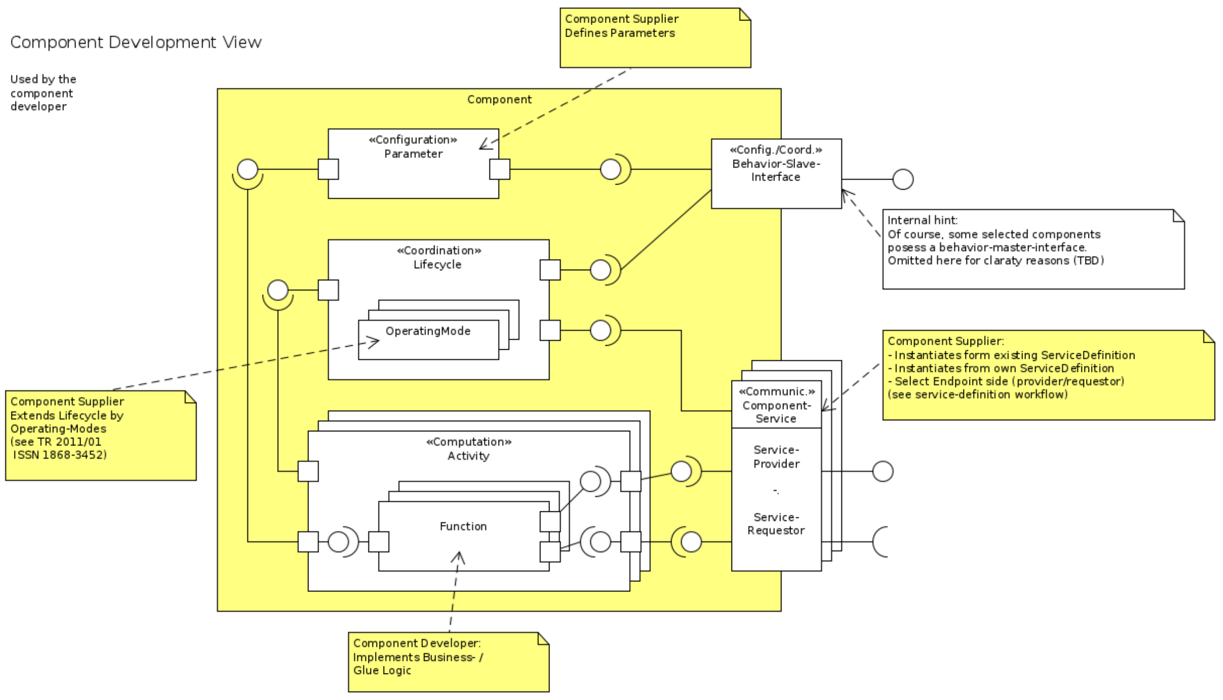
- Views in the RobMoSys Glossary
- Views in RobMoSys Composition Structures
- Views in the PC domain analogy
- Roles in the Ecosystem

modeling:views:start · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/modeling:views:start

# Component Development View

The component developer view clusters elements of the Component Metamodel that are relevant to the Component Supplier.

The component development view (shown in the figure below) needs to be rich enough and provide sufficient structures such that this model can serve as a consistent baseline for all the successive development steps (such as e.g. system composition/configuration) that rely on proper component models. At the same time, the component development view should avoid definition of too many low level details that are more related to internal knowledge that is not required for supporting composition with respect to the surrounding models. In this way, the component development view always is a trade-off between providing enough structures where needed and leaving enough design freedom for the internal realization.


Component Development View
Used by the component developer
Component Supplier Defines Parameters
Component
«Configuration» Parameter
«Config./Coord.» Behavior-Slave-Interface
Internal hint: Of course, some selected components posess a behavior-master-interface. Omitted here for claraty reasons (TBD)
«Coordination» Lifecycle
OperatingMode
Component Supplier: - Instantiates form existing ServiceDefinition - Instantiates from own ServiceDefinition - Select Endpoint side (provider/requestor) (see service-definition workflow)
Component Supplier Extends Lifecycle by Operating-Modes (see TR 2011/01 ISSN 1868-3452)
«Communic.» Component-Service
Service-Provider
-.
Service-Requestor
«Computation» Activity
Function
Component Developer: Implements Business- / Glue Logic

The only interaction point of a component with other components is through services. Therefore, a component can specify several provided and/or required services. A special kind of service is the behavior-interface which is used by the behavior coordination layer to coordinate this component at run-time (i.e. to set propper configurations, to activate/deactivate certain component modes, etc.). Therefore, the behavior-interface interacts with the component's internal parameter structure and the component's lifecycle state automaton which also defines the component-specific run-time modes.

The component's services interact within a component with Activities and the component's Lifecycle. The component's Lifecycle affects the lifelines of services and the activation/deactivation of Activities.

Regarding a component's Services, as long as the component is initializing (during start-up) or as long as a component is in a fatal-error mode, then the provided services might be physically available but not ready to properly offer a service (i.e. not able to answer query requests).

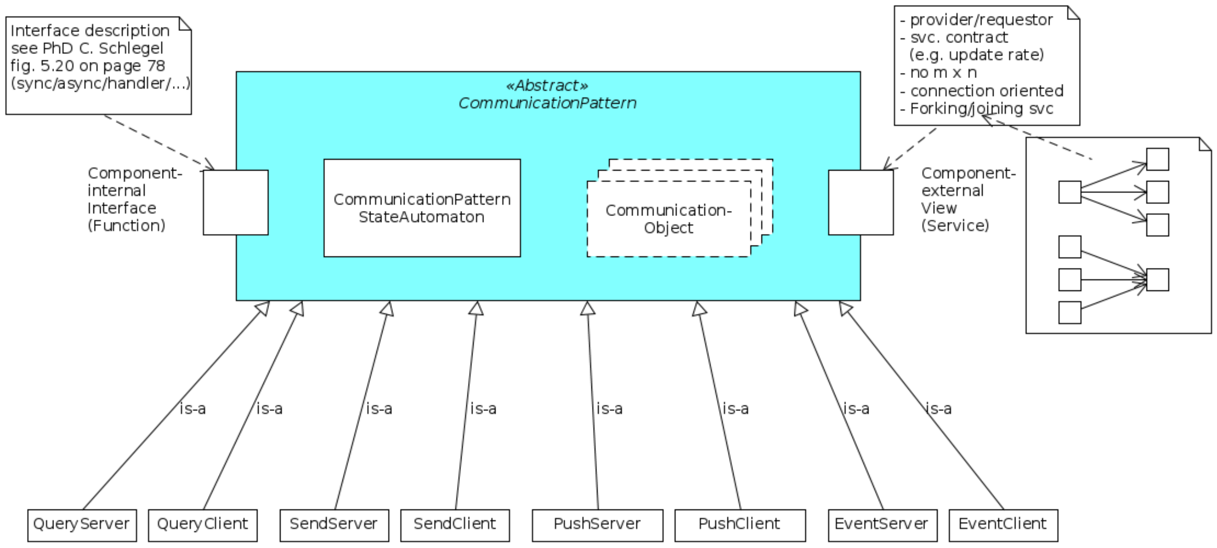
The next component-internal structural element is an Activity, which is an abstract representation of a task (or more precisely of a thread). An activity wraps a functional block which by itself is passive and only gets active by the execution environment of its parent Activity. This is an effective decoupling of the design and implementation of functional parts within a component and the execution of the functions. This even allows configuration of the execution characteristics for individual functions even after the component has been fully implemented and shipped to e.g. a system builder and without affecting the component's internal implementation.

As mentioned above, it is important that a structural model provides enough details that are required to communicate the structural knowledge of a component to other developer roles as well as to provide a sound foundation for the later development steps. In this respect, it is equally important to mention which parts have been omitted on purpose in order not to intermix the responsibilities and concerns that become relevant in later development steps. The most important parts that have been omitted on purpose are: (1) the mapping of services to a particular communication middleware (which is the responsibility of another developer role) (2) the mapping of Activities to a particular execution container such as Windows/Linux threads, or QNX/RTAI real-time threads (again a responsibility of another developer role) and (3) the definition of the services by themselves (which might be the responsibility of domain experts).

# Communication Pattern View

The communication pattern view clusters elements of the communication pattern metamodel that defines a fixed and stable set of recurring communication semantics.

This set of recurring communication semantics is defined for the robotics domain independent of an underlying communication middleware which can be flexibly selected in another development phase.


Interface description see PhD C. Schlegel fig. 5.20 on page 78 (sync/async/handler/...)
«Abstract» CommunicationPattern
- provider/requestor
- svc. contract (e.g. update rate)
- no m x n
- connection oriented
- Forking/joining svc
Component-internal Interface (Function)
CommunicationPattern StateAutomaton
Communication-Object
Component-external View (Service)
is-a
QueryServer
QueryClient
SendServer
SendClient
PushServer
PushClient
EventServer
EventClient

The communication patterns consist of an internal and external view of the component interface. The external view is defined by the behavior of the communication pattern itself. References therefore are provided in Communication-Pattern Metamodel.

While the API of the internal component view can be implemented manually such that the behavior of the communication patterns is ensured, this implementation requires a lot of knowledge about the internal behavior of the communication patterns and the middleware abstraction level. Hence, RobMoSys uses the existing C++ open-source reference specification of the API derived from the SmartSoft framework [https://github.com/Servicerobotics-Ulm/SmartSoftComponentDeveloperAPIcpp]. Using the existingAPI specification increases independence of the component's internal business logic from the different framework implementations, each based on a specific middleware solution. Besides, the existing API is well time-tested over the past 10 years, which saves a lot of efforts of redefining this API.

## RobMoSys Tooling Support

- In the SmartSoft World, the component internal interface is defined here [https://github.com/Servicerobotics-Ulm/SmartSoftComponentDeveloperAPIcpp]

## See also

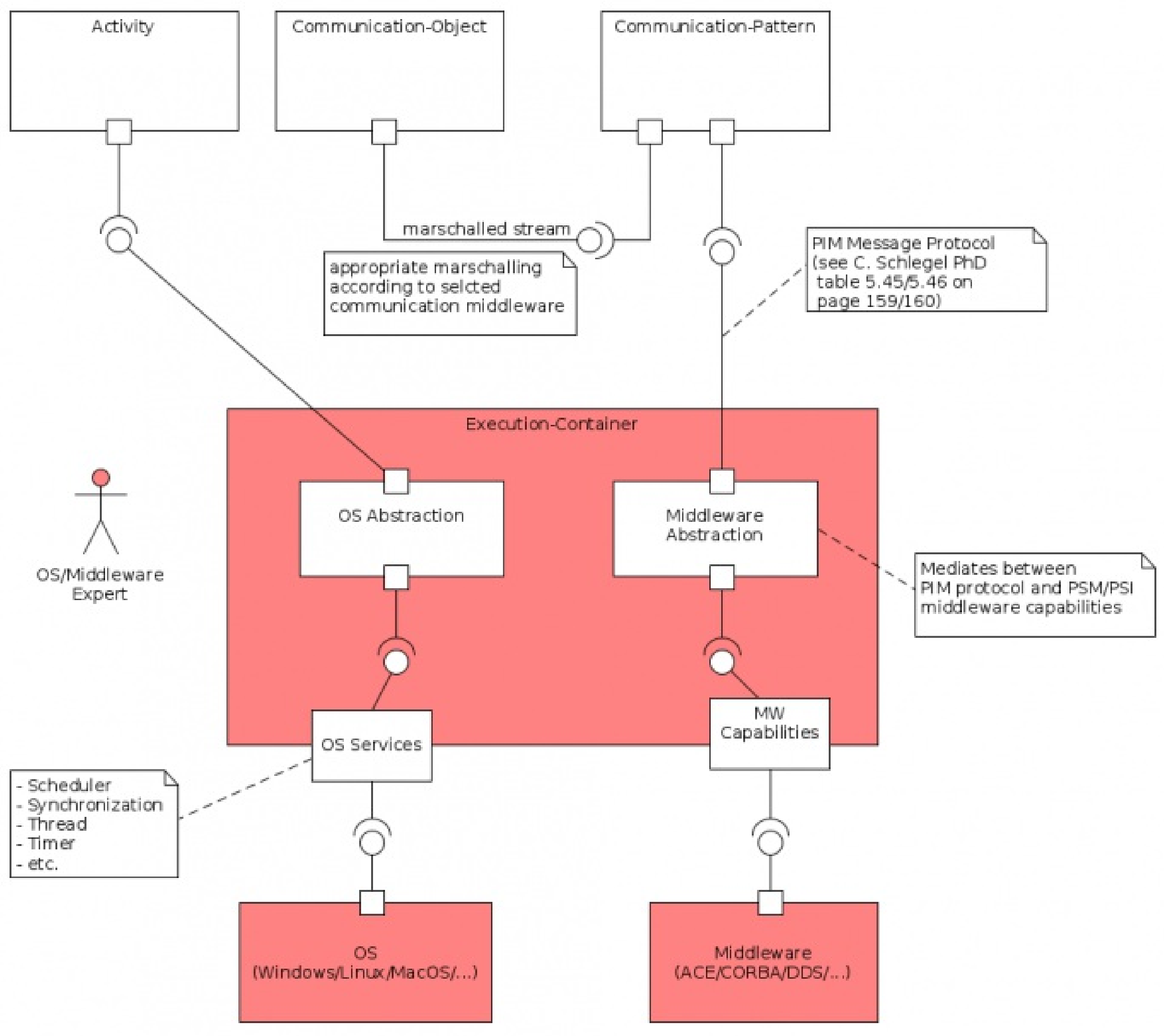
- Communication-Pattern Metamodel

modeling:views:communication_pattern · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/modeling:views:communication_pattern

# Execution Container View

The Execution Container View shows the mapping from platform independent models (such as components and services) into concrete platforms (i.e. Operating Systems and Communication Middlewares).

A component (see Component Metamodel) is at first independent of an actual execution environment. The actual mapping towards a communication middleware and an operating system (OS) is done in a later development step (such as e.g. the deployment step). For example, during the deployment phase of component to a specific platform, an accordingly used operating system and communication middleware become known which can then be mapped to the so far independent component.


Activity
Communication-Object
Communication-Pattern
marschalled stream
appropriate marschalling according to selcted communication middleware
PIM Message Protocol (see C. Schlegel PhD table 5.45/5.46 on page 159/160)
Execution-Container
OS/Middleware Expert
OS Abstraction
Middleware Abstraction
Mediates between PIM protocol and PSM/PSI middleware capabilities
OS Services
MW Capabilities
- Scheduler
- Synchronization
- Thread
- Timer
- etc.
OS (Windows/Linux/MacOS/...)
Middleware (ACE/CORBA/DDS/...)

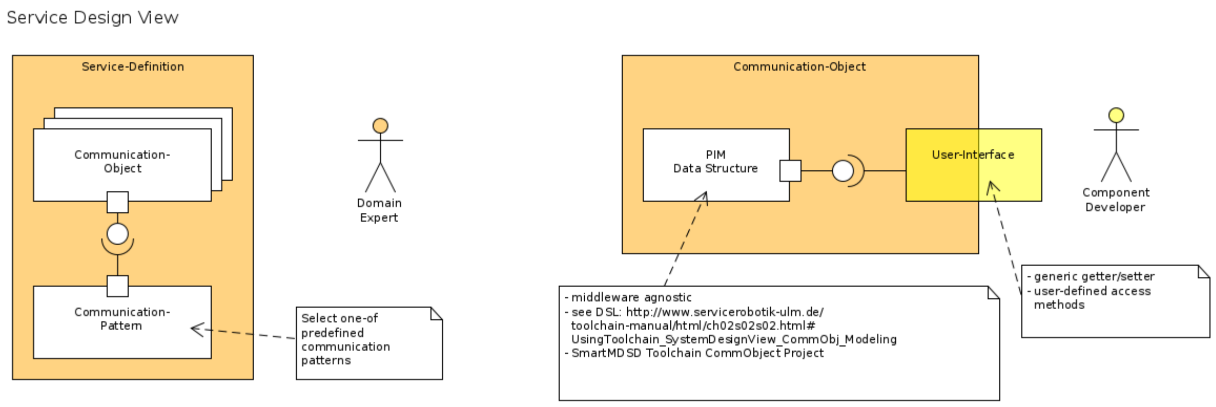
At this point an Activity becomes a certain implementation of a thread (such as e.g. a Windows thread or an RTAI real-time thread). Also, the actual marshaling (i.e. the serialization technique for the communicated data structures) and the used communication environment are selected. This should not affect the possible functional constraints of a component and different communication middlewares should be usable (as long as there are no specific constraints such as e.g. a specific real-time requirements for communication, which then should be complied with).

---

modeling:views:execution_container · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/modeling:views:execution_container

# Service Design View

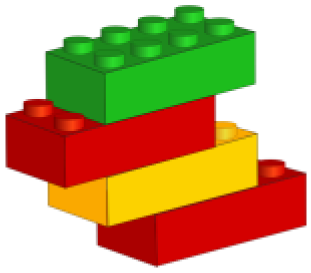
The service design view clusters elements of the Service Metamodel that are relevant to theService Designer.


Service Design View
Service-Definition
Communication-Object
Communication-Pattern
Domain Expert
Select one-of predefined communication patterns
Communication-Object
PIM Data Structure
User-Interface
Component Developer
- middleware agnostic
- see DSL: http://www.servicerobotik-ulm.de/toolchain-manual/html/ch02s02s02.html#UsingToolchain_SystemDesignView_CommObj_Modeling
- SmartMDSD Toolchain CommObject Project
- generic getter/setter
- user-defined access methods

A service definition (shown on the left in the figure) comprises of a selection of a communication pattern and a selection of a communication object. A communication object is a data structure that is communicated between a service provider and a service requestor. The exact direction of communication is defined by the communication pattern (see also Communication Pattern View). The communicated data structure is independent of the underlying communication middleware that is linked in another development phase as explained in the preceding section above.

modeling:views:service_design · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/modeling:views:service_design

# Composition in an Ecosystem



RobMoSys adopts a composition-oriented approach to system integration that manages, maintains and assures system-level properties, while preserving modularity and independence of existing robotics platforms and code bases, yet can build on top of them.

- Introduction to Composition in an Ecosystem
- We illustrate composition by:
    - Task-Level Composition for Robotic Behavior
    - Service-based composition of software components
    - Composition of algorithms
    - Managing Cause-Effect Chains in Component Composition
    - Coordinating Activities and Life Cycle of Software Components

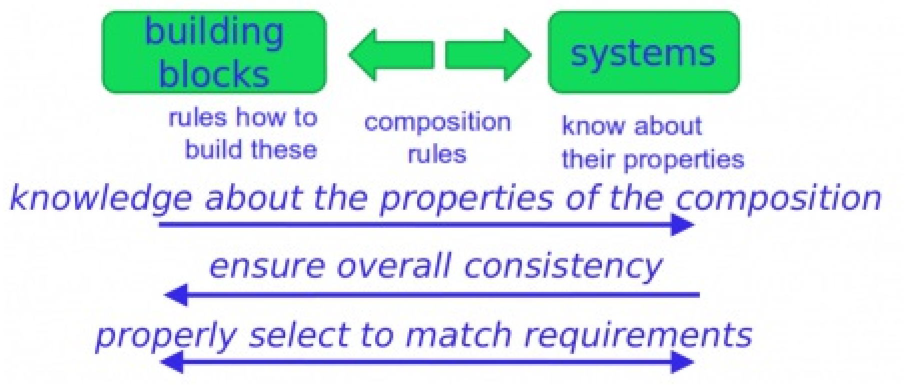
composition:start · Last modified: 2018/06/29 17:55
http://www.robmosys.eu/wiki-sn-02/composition:start

# Introduction to System Composition in an Ecosystem

RobMoSys adopts a composition-oriented approach to system integration that manages, maintains and assures system-level properties, while preserving modularity and independence of existing robotics platforms and code bases, yet can build on top of them. System Composition is the action or activity of putting together a service robotics system from existing building blocks (e.g. software components) in a meaningful way, flexibly combining and re-combining them depending on the application's needs.

- Composition is about the management of the interfaces between different roles (participants in an ecosystem) in an efficient and systematic way.
- Composition is about guiding the roles via superordinate composition-structures.
- Composition is about explicating and managing properties.
- Composition is about the right levels of abstraction.
- Composition is about access restriction and views for roles.

We operationalize architectural patterns and composition such that properties of system-of-systems become known in order to build trust in the system under development.


building blocks
systems
rules how to build these
composition rules
know about their properties
knowledge about the properties of the composition
ensure overall consistency
properly select to match requirements

System composition puts a focus on the new whole that is created from existing parts rather than on making parts work together only by glueing them together: the whole still consists of its parts, they do still exist as entities and are thus still exchangeable. This is in contrast to integration.

Software components, for example, that are subject to composition shall be taken as-is (and only configured on model level within predefined configuration boundaries). Software components thus have to be built with this intention right from the beginning. The context in which they will later be composed is unknown, which puts special requirements on their composability and the overall workflow.

Composition is about guiding the roles via superordinate composition-structures. It is about adhering to a composition structure, thus gaining immediate access to all other parts that also adhere to this (same) structure. In contrast, integration is about building adapters between (all) parts or even modifying the parts themselves.

## System Integration

A distinction between integration and composition can be drawn by the effort (see [1]): the ability to readily combine and recombine composable components distinguishes them from integrated components, which are modified with high effort to make them work with others, essentially by writing adapters. The integrated part amalgamates with the whole (i.e. the whole becomes one part, individual parts blend together, as red and green water will mix), thus making it hard to remove or exchange individual parts from the whole. If they are removed, it requires new adapters/adjustments.

**Acknowledgement**

This document contains material from:

- Lotz2018 Alex Lotz, "Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System", Dissertation, Technische Universität München, München 2018. [https://mediatum.ub.tum.de/?id=1362587]
- Lutz2017 Matthias Lutz, "Model-Driven Behavior Development for Service Robotic Systems: Bridging the Gap between Software- and Behavior-Models," 2017. (unpublished work)
- Stampfer2018 Dennis Stampfer, "Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics". Dissertation, Technische Universität München, München, Germany, 2018. [http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2]

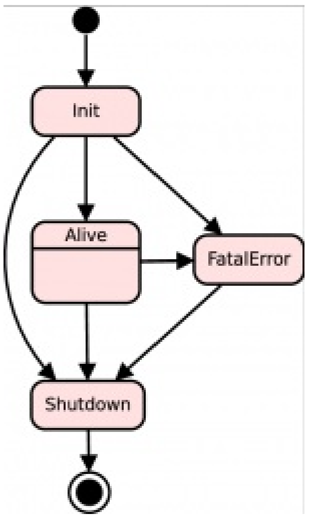
1)

Mikel D. Petty and Eric W. Weisel. "A Composability Lexicon", in Proc. Spring 2003 Simulation Interoperability Workshop, March 2003, Orlando, USA.

# Coordinating Activities and Life Cycle of Software Components

The coordination of software components at run-time and the run-time management of the component's internal resources are fundamentally important for designing robust and efficient systems. Therefore, RobMoSys specifies a generic **component lifecycle** that can be extended by component-specific**operation modes** (see the technical report below for further technical details).

The **component lifecycle** (see figure on the right) is a generic state automaton that every component has by default and that manages the initialization, shutdown and operation of a component in a uniform way. This lifecycle does not require a detailed metamodel as it is the same for every component and thus is an implicit part of the Component-Definition Metamodel (see the “Lifecycle” element in the component metamodel). The lifecycle is defined here:



- Christian Schlegel, Alex Lotz and Andreas Steck, “SmartSoft - The State Management of a Component”, in *Technical Report 2011/01*, Hochschule Ulm, Germany, ISSN 1868-3452, 2011. PDF [http://www.zafh-servicerobotik.de/dokumente/ZAFH-TR-01-2011-ISSN-1868-3452.pdf]

Moreover, every component can specify individual **operation modes** (see Component Development View) which can be dynamically (de-)activated at run-time to manage the component's internal activities and thus the component's functional resource consumption. There is an interesting relation between the component's **operation modes**, **services** and **functions**. The component's **operation modes** interface between the component's internal **functions** (implemented within relevant **activities**) and the component's **services**. Each **operating mode** activates related **activities** and thus **functions**. As **activities** are responsible for generating data for related **services**, activating a certain **operating mode** indirectly activates respective **services**. Deactivating a certain **operation mode** means that one or several related **activities** are deactivated (i.e., each deactivated activity stops before its next execution cycle until this activity gets activated again). This is a uniform mechanism to dynamically manage the component's resources at run-time in a consistent way without violating the component's internal implementation.

Overall, the management of the component's **lifecycle** and the management of the component's**operation modes** is an important part of the component's**coordination interface** (see Coordination and Configuration Patterns). Several robotic frameworks such as SmartSoft and RT-Middleware support this component lifecycle directly and other frameworks such as ROS are currently working on the implementation of a similar component lifecyle under the term Managed nodes [http://design.ros2.org/articles/node_lifecycle.html].

## Example Use-Case

The Gazebo/TIAGo/SmartSoft Scenario consists of several components each implementing at least the generic **component lifecycle** as described above. This already allows coordinated startup (i.e., initialization) and shutdown (i.e., destruction) of these components. During regular operation, each component at least has two regular **opertion modes**:

- **Neutral**: all the component's internal activities are in a standby state
- **Active**: all the component's internal activities are activated and operational

At runtime, only one of these modes can be active at a time and switching between them is possible at any time. The **Neutral** mode is reserved for the inactive (i.e., passive) state of a component. This means that a component might by fully started and ready to deliver a service but is within a standby mode and does not consume its specific resources. Switching into the **Active** mode means that the component wakes up an continuously delivers its service(s). These two modes are the default **operation modes** of a component which covers the majority of all use-cases.

In some cases, it is reasonable to have a more detailed definition of the **Active** mode (i.e., if a component can have several partial activation of its internal functionalities). For example, the component “SmartMapperGridMap” from the Gazebo/TIAGo/SmartSoft Scenario provides two main functionalities, namely to build long-term maps and update local grid maps. For coordinating these two functionalities, this component provides (besides of the default “Neutral” mode) the following three **operation modes** (instead of the generic “Active” mode):

- **BuildCurrMap**: for updating only the current (local) map
- **BuildLtmMap**: for building the long-term map
- **BuildBothMaps**: for building and updating both maps (highest resource demands)

These modes enable the robot to dynamically coordinate the amount of resources a component consumes depending on the current situation and the task a robot is performing. Switching into the **Neutral** mode is always possible for each component in situations where this component is not used in a system. In this way, it is not necessary to completely kill a component (if currently not needed) and start it again (if needed again) which is typically more time-consuming than just switching between respective component's **operation modes**.

Concrete models for these component examples are presented and discussed in the Example for Coordinating Activities and Life Cycle of Software Components using the SmartMDSD Toolchain.

## RobMoSys Modeling Support

- Component Development View
- Component-Definition Metamodel

The operation mode in the component-definition metamodel is modeled via the lifecycle metamodel which is yet to be described. Fix Me!

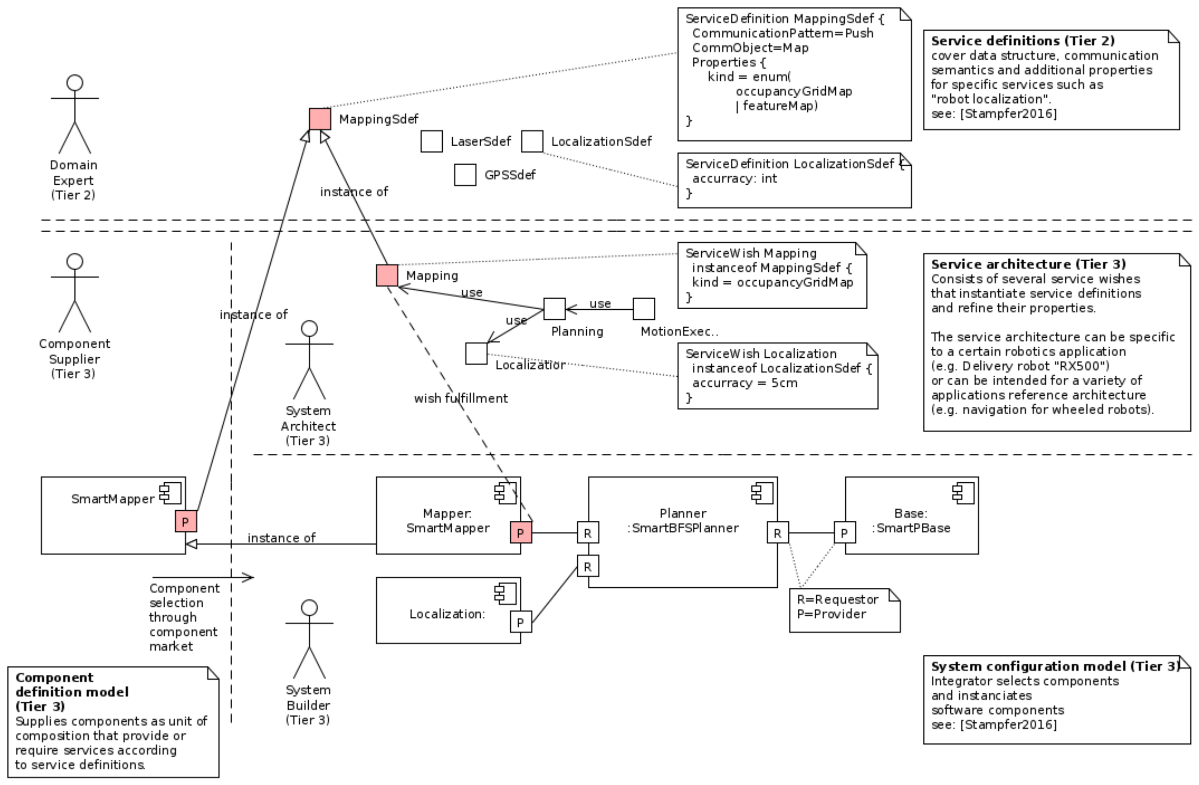
## RobMoSys Tooling Support

The following page demonstrates how concrete **operating modes** are modeled in existing navigation components using the SmartMDSD Toolchain: Example for Coordinating Activities and Life Cycle of Software Components using the SmartMDSD Toolchain

composition:component-activities:start · Last modified: 2018/06/29 17:54
http://www.robmosys.eu/wiki-sn-02/composition:component-activities:start

# Service-based Composition

The service-based composition approach is an example to illustrate the use of the composition tiers. Below is the illustration that corresponds to the role descriptions. The service-based composition approach uses service-definitions as central architectural element for composition of software components. We call the links between service definition, service wish, and service with fulfillment the "service triangle".


ServiceDefinition MappingSdef { CommunicationPattern=Push CommObject=Map Properties { kind = enum( occupancyGridMap | featureMap) }
Service definitions (Tier 2) cover data structure, communication semantics and additional properties for specific services such as "robot localization". see: [Stampfer2016]
Domain Expert (Tier 2)
MappingSdef
LaserSdef
LocalizationSdef
GPSSdef
instance of
ServiceDefinition LocalizationSdef { accurracy: int }
Component Supplier (Tier 3)
Mapping
use
use
use
Planning
MotionExec..
Localization
instance of
wish fulfillment
System Architect (Tier 3)
ServiceWish Mapping instanceof MappingSdef { kind = occupancyGridMap }
ServiceWish Localization instanceof LocalizationSdef { accurracy = 5cm }
Service architecture (Tier 3) Consists of several service wishes that instantiate service definitions and refine their properties.
The service architecture can be specific to a certain robotics application (e.g. Delivery robot "RX500") or can be intended for a variety of applications reference architecture (e.g. navigation for wheeled robots).
SmartMapper
P
instance of
Mapper: SmartMapper
P
R
Planner :SmartBFSPlanner
R
R
P
Base: :SmartPBase
Localization:
P
R=Requestor P=Provider
Component selection through component market
System Builder (Tier 3)
Component definition model (Tier 3) Supplies components as unit of composition that provide or require services according to service definitions.
System configuration model (Tier 3) Integrator selects components and instanciates software components see: [Stampfer2016]

[Stampfer2016] Dennis Stampfer, Alex Lotz, Matthias Lutz and Christian Schlegel. "The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software". Special Issue on Domain-Specific Languages and Models in Robotics, Journal of Software Engineering for Robotics (JOSER), 7(1), 3-19 ISSN: 2035-3928, July 2016.

## RobMoSys Modeling Support

- Composition Structures
- Component Definition Metamodel
- Service Definition Metamodel

## RobMoSys Tooling Support

- Support for Service-based Composition by the SmartMDSD Toolchain

## See also

- Architectural Pattern for Service Definitions

## Acknowledgement

This document contains material from:

- Stampfer2018 Dennis Stampfer, "Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics". Dissertation, Technische Universität München, München, Germany, 2018. [http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2]

---

composition:service-based-composition:start · Last modified: 2018/06/29 17:54
http://www.robmosys.eu/wiki-sn-02/composition:service-based-composition:start

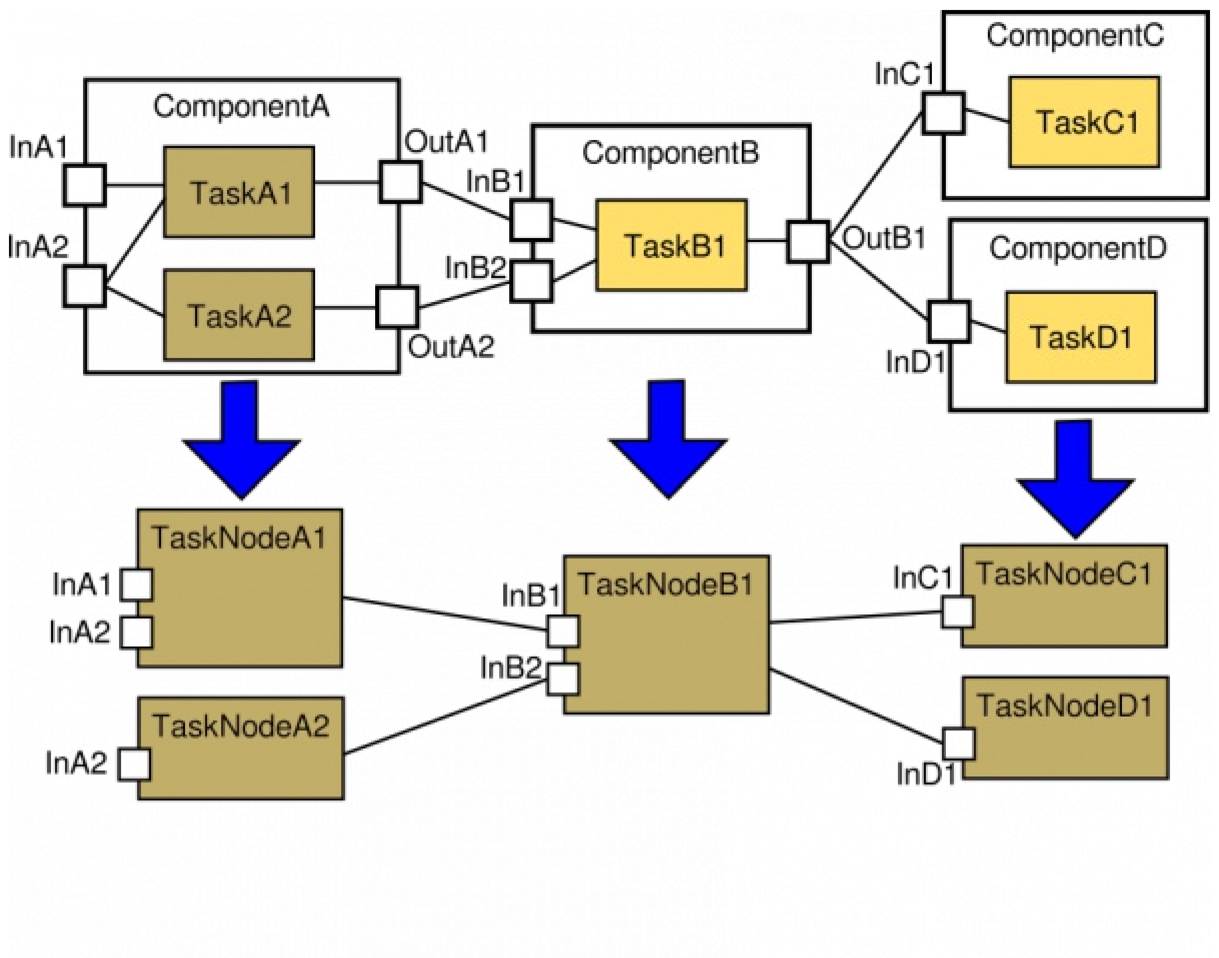
# Managing Cause-Effect Chains in Component Composition

Composition can be found everywhere in a system and consider different aspects of that system. There is a general distinction between **vertical composition** (as e.g. demonstrated by the Service-based Composition) and **horizontal composition**. This wiki page describes an example of **horizontal composition** using "Cause-Effect Chains".

While **vertical composition** addresses the combination of parts at **different levels** of abstraction (see Separation of Levels and Separation of Concerns), **horizontal composition** focuses on the combination of parts at **the same level** of abstraction. One example for the latter kind of composition is the definition of the so called **Cause-Effect Chains** for the purpose of refining specific system-level, performance-related, and non-functional properties. The following reference provides further details of this topic:

- Alex Lotz, Arne Hamann, Ralph Lange, Christian Heinzemann, Jan Staschulat, Vincent Kesel, Dennis Stampfer, Matthias Lutz, and Christian Schlegel. "Combining Robotics Component-Based Model-Driven Development with a Model-Based Performance Analysis." In: IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR). San Francisco, CA, USA, Dec. 2016, pp. 170–176. LINK [http://dx.doi.org/10.1109/SIMPAR.2016.7862392]

In brief, the management of "Cause-Effect Chains" addresses the problem of combining different **models of computation** such as e.g. Synchronous Data-Flow (SDF) [https://ptolemy.berkeley.edu/publications/papers/87/synchdataflow/], and Petri Net [https://en.wikipedia.org/wiki/Petri_net]. That is, individual components typically specify parts of the overall, system-level **models of computation** by the definition of **activities** (i.e., the threads of that component). As the component should be used in different systems and different systems often require different **models of computation**, this component needs to be configured differently for each individual system so that a required **model of computation** is realized. Therefore, the **activities** of individual components are configured in a system so that the interaction of **activities** from different components are either directly linked (i.e., in a trigger relation) or loosely coupled (i.e., registers semantics). The constraint of a direct link is then mapped onto a related scheduling strategy (which depends on the capabilities of the used operating system).

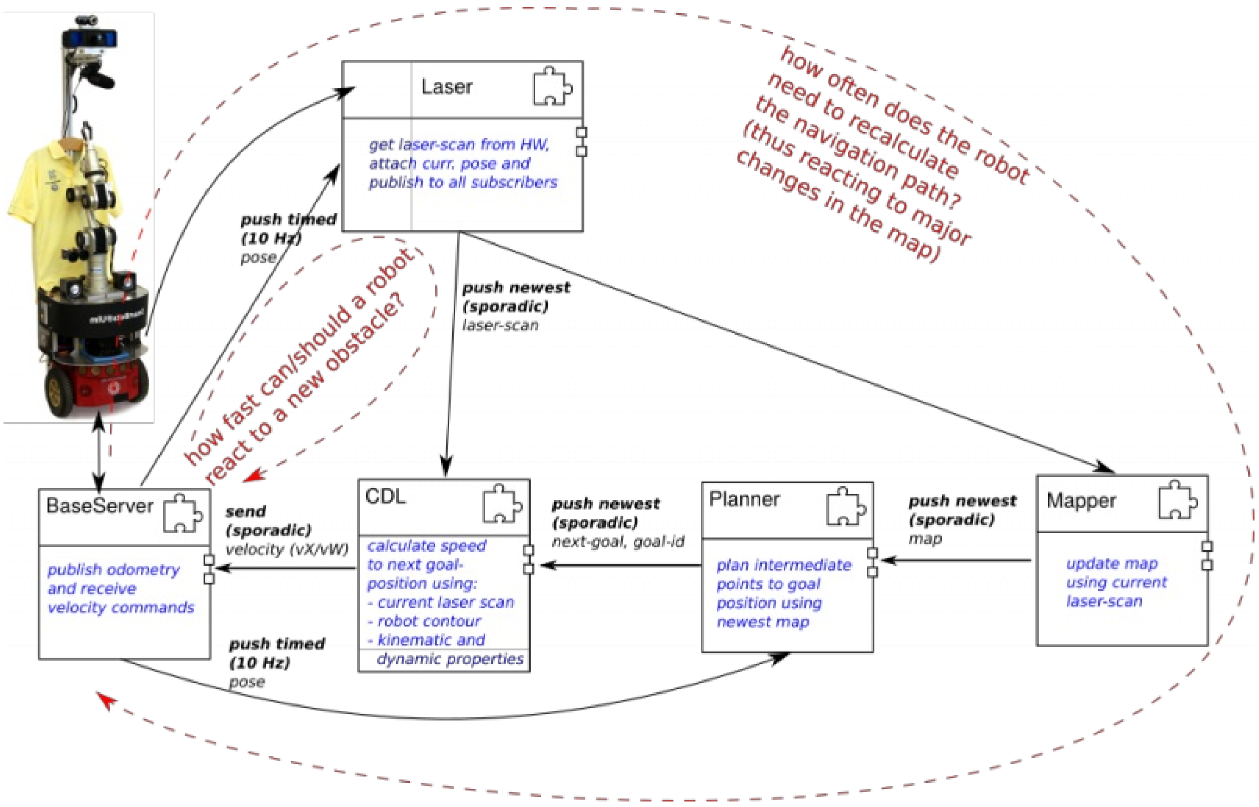


There is a relation between the System Component Architecture Metamodel (see also the System Builder Role and the System Configuration View) and the Cause-Effect Chain and its Analysis Metamodels (see also the Performance Designer Role and the Performance View). The figure above shows an illustration of models that demonstrate this relation. In particular, the Cause-Effect Chain metamodel (an example model is sketched in the lower half of the figure) removes component-boundaries by purpose to hide model-details that are not relevant for that modeling view. This results in a directed graph consisting of **activity nodes** (the orange blocks in the lower half of the figure) and abstract communication links. Consequently, an existing System Component Architecture model can be transformed into a Cause-Effect Chain model which again is enriched by further details related to refining the links between the **activity nodes** (i.e., specifying whether the links are loosely coupled or directly linked).

Moreover, the Component Definition meta-model enables the modeling of **components** with **activities** so that a component can be fully implemented and supplied to different system builders. The selected level of details of a Component Definition meta-model leaves the relevant aspects related to the specification of **models of computation** open for later configuration in different systems. As a result, existing components can be flexibly instantiated in different systems (conforming to the System Component Architecture Metamodel) and the configuration of components can be adjusted (conforming to the Cause-Effect-Chain and its Analysis Metamodels) without violating the component's internal implementation so that overall system-level requirements such as end-to-end delay demands, and CPU load requirements are satisfied for the current system under development. This management of Cause-Effect Chains is one of the leading examples for horizontal composition, providing a general mechanism that can be applied for other aspects of a system in a similar way.

## Example Use-Case for Managing Cause-Effect Chains

The figure below shows an example system derived from the Gazebo/TIAGo/SmartSoft Scenario consisting of navigation components altogether providing collision-avoidance and path-planning navigation functionality. This example is used in the following to discuss different aspects related to managing cause-effect chains which are again related to managing performance-related system aspects.



Laser
get laser-scan from HW, attach curr. pose and publish to all subscribers
how often does the robot need to recalculate the navigation path? (thus reacting to major changes in the map)
push timed (10 Hz) pose
push newest (sporadic) laser-scan
how fast can/should a robot react to a new obstacle?
BaseServer
publish odometry and receive velocity commands
send (sporadic) velocity (vX/vW)
CDL
calculate speed to next goal-position using: - current laser scan - robot contour - kinematic and dynamic properties
push newest (sporadic) next-goal, goal-id
Planner
plan intermediate points to goal position using newest map
push newest (sporadic) map
Mapper
update map using current laser-scan
push timed (10 Hz) pose

The example system in the figure above consists of five navigation components, from which two are related to hardware devices (i.e., the Pioneer Base and the SICK Laser) and the other three components respectively implementing collision-avoidance (i.e., the CDL component), mapping and path-planning. As an example, two performance-related design questions are introduced in the following with the focus on discussing the architectural choices and the relevant modeling options:

1. How fast can a robot react to sudden obstacles taking the current components into account?
2. How often does the robot need to recalculate the path to its current destination (thus reacting to major map changes)?

## RobMoSys Modeling Support

- Cause-Effect-Chain and its Analysis Metamodels

## RobMoSys Tooling Support

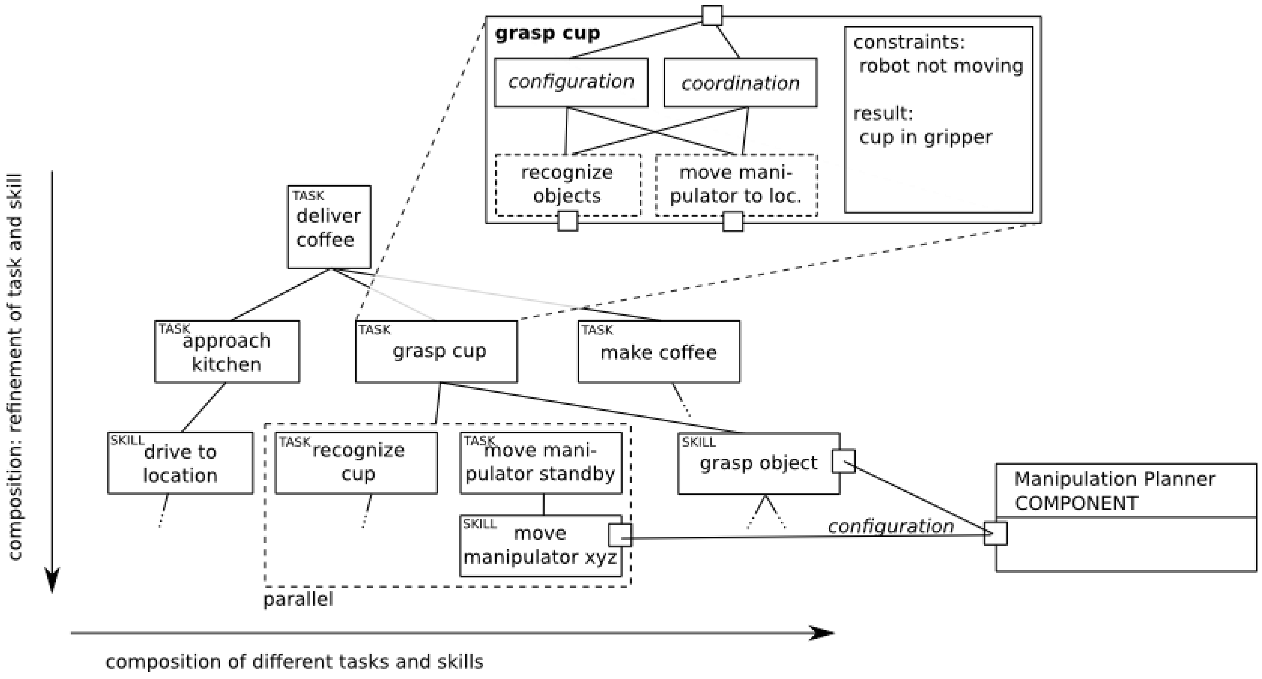
- The following page discusses the concrete models of this example using the SmartMDSD Toolchain: Example Use-Case for Managing Cause-Effect Chains in Component Composition using the SmartMDSD Toolchain

See also:

- Architectural Pattern for Stepwise Management of Extra-Functional Properties

# Task-Level Composition for Robotic Behavior

- Below is an example of how tasks can be composed for Robotic Behavior
- It shows how tasks and skills can be composed flexibly
- Several tasks can be composed to be executed in sequence or in parallel (horizontal composition)
- A task can be refined with other tasks (vertical composition): Abstract tasks are refined to more concrete tasks.
- Refinement of tasks may be static or dynamic
    - Static: The tasks and eventually the order is known. E.g. making coffee always involves approaching the machine, putting a cup into the machine, pressing the button, etc.
    - Dynamic: The tasks and the order are not known in advance (i.e. to be solved by symbolic planning): E.g. it is not known what is the best way to clean up the table after customers left (what order, what to stack into each other, what to carry at once/first/next/last, etc.)
- Skills will finally translate to configurations of one or more components (lower right). E.g. moving the manipulator requires to configure the component for collision-free manipulation-planning in a certain environment and the manipulator component to move along these collission-free trajectories.
- Grasp cup relies on the existence of a task "recognize-object" which is later bound to "recognize-cup".
- There are constraints that have to be maintained during the execution of a task, for example: the robot is not moving while manipulating.
- There are results of a task that effect execution of other tasks, even after the current task was finished. For example, grasping a cup means that the cup still is in the gripper after the execution is done.



## See also

- Architectural Pattern for Task-Plot Coordination (Robotic Behaviors)
- Architectural Pattern for Component Coordination
- Robotic Behavior Metamodel

## Acknowledgement

This document contains material from:

- Lotz2018 Alex Lotz, "Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System", Dissertation, Technische Universität München, München 2018. [https://mediatum.ub.tum.de/?id=1362587]
- Lutz2017 Matthias Lutz, “Model-Driven Behavior Development for Service Robotic Systems: Bridging the Gap between Software- and Behavior-Models,” 2017. (unpublished work)
- Stampfer2018 Dennis Stampfer, "Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics". Dissertation, Technische Universität München, München, Germany, 2018. [http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2]

composition:task:start · Last modified: 2018/06/29 17:54
http://www.robmosys.eu/wiki-sn-02/composition:task:start