



RobMoSys

H2020—ICT—732410

ROBMO SYS

**COMPOSABLE MODELS AND SOFTWARE
FOR ROBOTICS SYSTEMS**

**DELIVERABLE D2.4:
FINAL (META-)MODELS, PROTOTYPICAL DSLs, TOOLS AND
IMPLEMENTATION**

Christian Schlegel (Technische Hochschule Ulm)

Dennis Stampfer (Technische 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: D2.4

Document title: Improved (meta-)models, prototypical DSLs, tools and implementation

Version: 1.0

Due date: December 31th, 2020

Delivery date: December 17th, 2020

Nature: Report (R)

Dissemination level: Public (PU)

Editor: Dennis Stampfer (THU), Christian Schlegel (THU)

Author(s): Alex Lotz (THU), Matthias Lutz (THU), Vineet Nagrath (THU),
Dennis Stampfer (THU), Christian Schlegel (THU),
Enea Scioni (KUL), Herman Bruyninckx (KUL),
Matteo Morelli (CEA), Ansgar Radermacher (CEA),
Luz Martinez Ramirez (TUM), Daniel Meyer-Delius (SIE)

Reviewer: Alfio Minissale (COMAU)

Executive Summary

This is an accumulative Deliverable for Tasks T2.2 to T2.7. The individual contributions of the respective Tasks are highlighted in the respective sections in this Deliverable document. This Deliverable is the *updated and final* version (M₄₈) of the Deliverables D2.2 and D2.3 for (meta-)models, prototypical DSLs, tools and implementations.

This Deliverable is about robotics (software) component (meta-)models for composition-oriented (software) engineering and their prototypical implementations (exploiting existing background of the partners as much as possible). It served as a software baseline for the other WPs and for Open Call 1 and Open Call 2.

The RobMoSys consortium uses a Wiki as main platform for sharing the information included in 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 the initial version remained stable, refinements and extensions as well as improvements are 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.

With the completion of the RobMoSys project, the euRobotics Topic Group with its community takes over the role of a trusted steward and host for a consolidated body-of-knowledge and trustee for these de-facto standards. This comprises the management of its further evolution, consolidation, and maintenance. The core model-driven tools (SmartMDSD, Papyrus4Robotics) are hosted as Eclipse projects for sustainability.

Summary of Updates to this Document

D2.4 - December 31th, 2020

This document is the updated and final version of the Deliverables D2.2 and D2.3. This document serves as an updated guide to the RobMoSys Wiki. Therefore, the following extensions of the RobMoSys Wiki contribute to this deliverable:

- Robotics Body-of-Knowledge
 - added cross-links between RobMoSys “**EU Digital Industrial Platform for Robotics**” and euRobotics Topic Group “**Stewardship Software Systems Engineering**”
- New Wiki pages
 - added “**Dependency Graphs**”
 - significantly extended **Tutorials for the SmartMDS Toolchain**
 - added “**Getting Started with Papyrus4Robotics**”
 - added “**Architectural Pattern for Component Coordination**”
 - added the “**Mobile Manipulation Stack**”
- Completion and improvement of various Wiki-Pages
 - completion of “**Task Definition Metamodel**” / “**Task Realization Metamodel**”
 - completion of “**Skill Definition Metamodel**” / “**Skill Realization Metamodel**”
 - improved readability of “**Technical User Stories**”
 - major updates of the “**Pilot Descriptions**”
- Update of the community corner:
 - results of 2nd ITP wave are *work in progress* as most of the 2nd wave ITPs were only completed by 30.11.2020

Overall, compared to the initial Delivery D2.2 and the updated Delivery D2.3, the final Deliverable D2.4 reflects the matured implementations of the composition structures within the RobMoSys consortium as well as the take-up of these structures within the *Integrated Technical Projects (ITPs)*.

D2.3 - March 31th, 2019

D2.3 is an update of the initial deliverable D2.2. D2.3 serves as an updated guide to the RobMoSys Wiki. Therefore, the following extensions of the RobMoSys Wiki contributed to D2.3:

- The “**Open Call 2 reading guide**”
- The “**Models directory**”
- Several new Wiki pages address Robotics Behavior Coordination models and views, as the following:
 - “**Support of Skills for Robotic Behavior**” provides an entry page and an overview of Robotics Behavior models implemented within the SmartMDS Toolchain
 - “**Skills for Robotic Behavior**”: this page conceptually describes the relations between the several new behavior-related models
 - “**Skill Definition Metamodel**” shows the metamodel for the new skill definition modeling language
- New “**community corner**” section within the RobMoSys Wiki that clusters new contributions from the *Integrated Technical Projects (ITPs)* of the first open call round in the dedicated Wiki pages:
 - “**Robotic Behavior in RobMoSys using Behavior Trees and the SmartMDS Toolchain (MOOD2be ITP)**”

- “Dealing with Metrics on Non-Functional Properties in RobMoSys (RoQME ITP)”
- “Using the YARP Framework and the R1 robot with RobMoSys (CARVE ITP)”
- “Benchmarking in the RobMoSys Ecosystem (Plug&Bench ITP)”
- “Safety Assessment of Robotics Systems Using Fault Injection in RobMoSys (eITUS ITP)”
- “Guaranteed Stability of Networked Control Systems (EG-IPC ITP)” (under review)
- New section in the RobMoSys Wiki with several subpages on “**Composition in an Ecosystem**” to illustrate composition by several examples that describe how RobMoSys tooling can be used to apply the specified concepts:
 - “Task-Level Composition for Robotic Behavior”
 - “Service-based composition of software components”
 - “Managing Cause-Effect Chains in Component Composition”
 - “Coordinating Activities and Life Cycle of Software Components”
- Extended Wiki pages that describe how the RobMoSys tooling supports in using the meta-model structures and modeling foundation guidelines
 - “Papyrus4Robotics”
 - “The SmartMDS Toolchain”
- Examples of Tier 2 domain structures have been extended and its support through RobMoSys tooling has been described
 - Wiki page on “Flexible Navigation Stack”

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.

Overall, compared to the initial Delivery D2.2, the updated Delivery D2.3 reflects the ongoing implementation efforts of the composition structures within the RobMoSys consortium as well as the take-up of these structures within the *Integrated Technical Projects (ITPs)* of the first open-call round. The core modeling structures related to the creation of individual components as well as first implementations of the system-level models have been realized in the two reference implementations, the “SmartMDS Toolchain” and “Papyrus for Robotics”. These tools have been used within the ITPs as a baseline and proved to be useful to extend the overall RobMoSys body of knowledge. This further allows to refine and to improve the overall RobMoSys composition structures as well as to gradually increase the awareness of the RobMoSys approach within the general robotics community.

D2.2 - June 30th, 2017

Initial version of this deliverable

Content

Executive Summary	3
Summary of Updates to this Document	4
Content	6
Introduction	7
Approach	8
Behavioral Modeling (Task T2.2)	10
Composition, Composability, Compositionality (Task T2.3)	10
Separation of Roles and Separation of Concerns (Task T2.4)	11
Non-functional Properties and QoS Management (Task T2.5)	11
Tooling	12
Introduction	12
Integration of Modeling Principles in a Meta-model (Task T2.6)	13
Tooling and Run-time Execution (Task T2.7)	14
Appendix	14

1 Introduction

RobMoSys is about managing the interfaces between different roles (robotics expert, domain expert, component supplier, system builder, installation, deployment and operation) and about separating 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 robot systems.

Deliverable D2.3 (update of initial version Deliverable D2.2) had the focus on the implementation of the RobMoSys composition structures considering the different roles and adhering to the generic meta-structures defined in the Deliverable D2.1.

The consolidation in form of this Deliverable D2.4 (final version) is based on experiences in implementing and using the RobMoSys composition structures within the core consortium of RobMoSys, within the RobMoSys ITPs, and within projects that are independent from RobMoSys and use the RobMoSys tools and thus apply the RobMoSys principles.

The core structures proved to be stable and required no modifications:

- The structural organization and arrangement of the RobMoSys Wiki and the descriptions of core concepts proved to be stable and required no modifications. The RobMoSys Wiki gives guidance with respect to the structure of the body-of-knowledge and allows for references to RobMoSys context from outside.
- The RobMoSys Wiki not only organizes the access to explanations and descriptions of RobMoSys core concepts, but also guides to the concrete (meta-)models, DSLs, tools and implementations. The links to that content are stable, but the final content referenced by those links is organized in repositories and undergoes adjustments, updates, and extensions.
- The core (meta-)models, DSLs, tools and implementations proved to be stable such that no modifications of core assets of RobMoSys had been necessary even that we now have all the insights from the ITPs. It is very important that concepts at a higher level of the abstraction pyramid are much more stable than those at a lower level (as changes at a particular level impact all subordinated levels).
- Thus, the RobMoSys Wiki content remained stable while at the same time, evolution and consolidations took place within the repositories. Of course, content of a repository is at any time conform to the RobMoSys Wiki parent structures under which it is attached.

A **Glossary** provides definitions for the most relevant terms in the context of RobMoSys. See

- Wiki Page on “**Glossary**”

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

- <https://robmosys.eu/wiki-sn-05/>

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

- <https://robmosys.eu/wiki-sn-05/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>

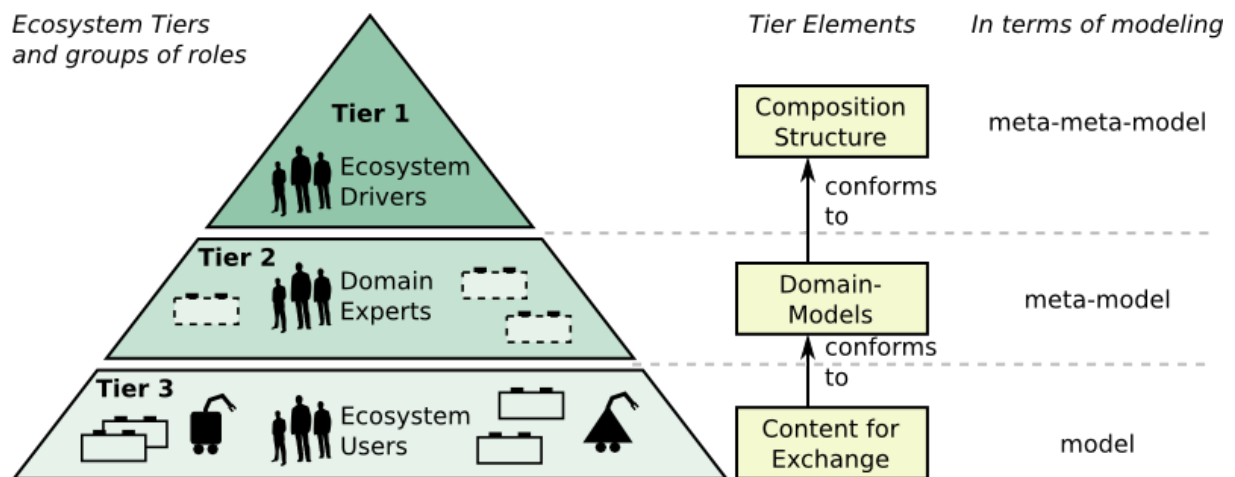


Figure 1: Tiers of an Ecosystem, their elements and the elements in terms of modeling.

Tier 1 distinguishes 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). These structures are refined for the robotics sub-domains (e.g. manipulation, object recognition) to provide guidance and structure for users of the ecosystem at Tier 3 (for example, building blocks suppliers and users).

While Deliverable D2.1 focusses on *generic* composition structures, the Deliverables D2.2, D2.3, and D2.4 address the **implementation of the RobMoSys composition structures** within concrete model-driven tooling.

2 Approach

The term “meta” in relation to a model refers to the abstraction between a model and its meta-model where the model conforms to its more abstract representation in a meta-model. Thereby, the meta-model by itself might be a model that conforms to yet another meta-model. Therefore, the meta-relation is not absolute but relative. In some cases, it makes sense to add further meta levels (such as in the term meta-meta-model in figure 1) in order to represent a hierarchy that is visible at once. However, the relative relation remains. Moreover, each individual meta-level by itself might be subdivided into further “sub” meta-levels such as is e.g. the case for Tier 1 which comprises three meta-levels (see figure 2). Again, because the meta relation is relative, there is no need to distinguish between a top-level (meta-)model and its “sub” (meta-)models as this distinction would be purely artificial. In this document the **RobMoSys composition structures** on the bottom level of composition Tier 1 will be referred to as **RobMoSys meta-models**.

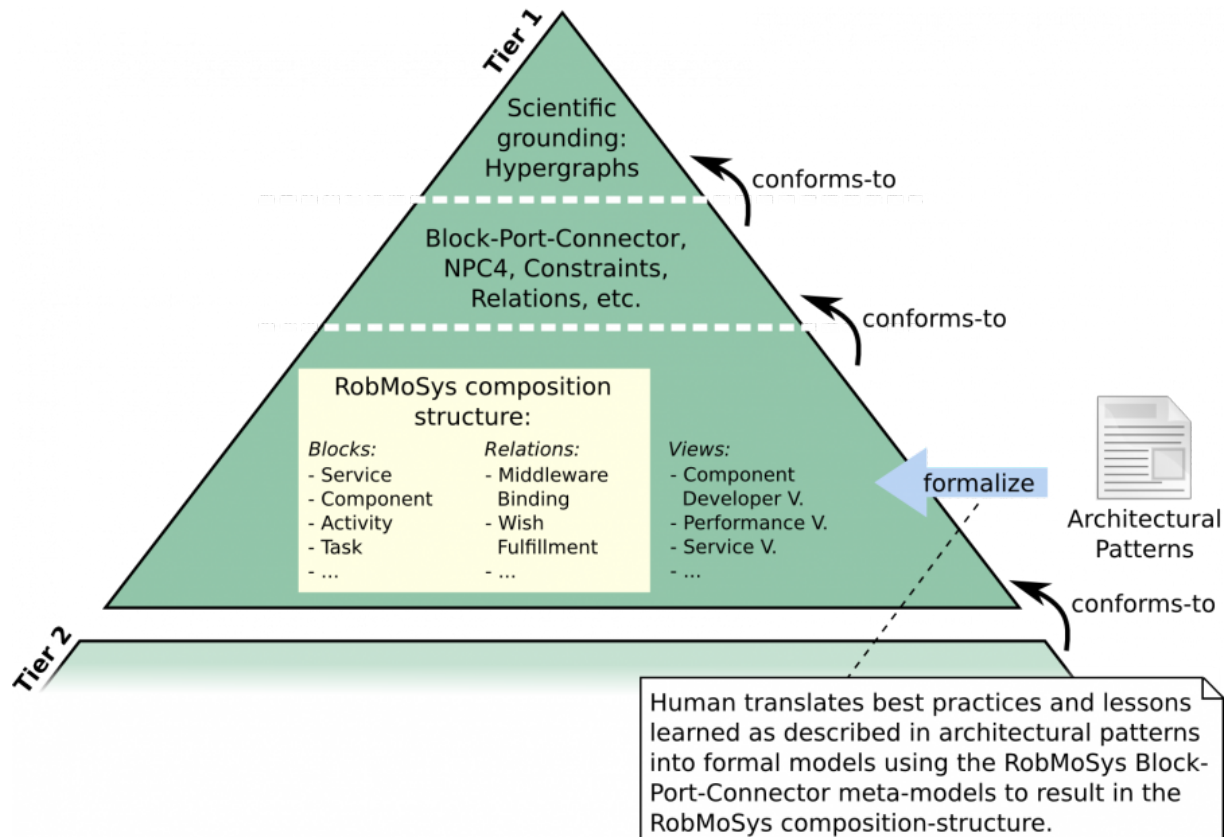


Figure 2: Details of the structure of Tier-1.

One of the benefits of the RobMoSys composition structures is to support role-specific views for the other two tiers, namely Tier 2 and Tier 3. It is important to notice that while the individual views focus on isolated aspects of an overall system, the views by themselves are not isolated but are interlinked over the RobMoSys composition structures. This is important for ensuring the overall system consistency, composition, composability and compositionality even if the individual roles independently contribute to the overall system.

The following list of Wiki pages provide further technical details with respect to the RobMoSys composition structures and views:

- Wiki pages on “**RobMoSys Composition Structures**”
- Wiki pages on “**Views**”

Another aspect of the RobMoSys composition structures (bottom layer of Tier 1) is that they serve as an intermediate abstraction level between the *generic* composition structures (middle and top layer of Tier 1, i.e. the blocks-ports-connectors, the entity-relation model and the hypergraph meta-level) and the models at Tier 2 and Tier 3. The bottom layer of Tier 1 (lowest abstraction level at Tier 1) is where the structural, behavioral and workflow knowledge is formalized. Model-driven tools are realized at Tier 1, but they cross all tiers to support creating and working on models of the respective tiers.

Two complementary reference implementations realize the composition structures in model-driven tools that are described in the respective Wiki pages:

- “**The SmartMDSD Toolchain**”
- “**Papyrus for Robotics**”

As of D2.3, the **SmartMDS Toolchain** implements composition structures related to the roles “Domain Expert” (i.e. “Service Designer”), “Component Supplier”, “System Builder”, “Performance Designer”, and “Behavior Developer”. A dedicated Wiki page on “**SmartMDS Toolchain Support for the RobMoSys Ecosystem Organization**” describes the tooling support for the different roles. The **Papyrus for Robotics** toolchain has a strong focus on composition structures for the “Safety Engineer”.

The following sections 2.1, 2.2, 2.3, and 2.4 accordingly address the Tasks T2.2, T2.3, T2.4 and T2.5 and individually refer to the according Wiki pages that describe the role-based composition structures in RobMoSys.

2.1 Behavioral Modeling (Task T2.2)

Task T2.2 refers to robotics behavior models that allow the modeling of situation-specific and dynamic behavior of the robot. This can be realized through Task coordination, different forms of process networks or finite state automata. As an initial baseline and outlined in Deliverable D2.2, this Task T2.2 contributed the SmartTCL language. That has been used by ITPs for defining robotic behaviors and for realizing pilots in RobMoSys. The following Wiki pages provide further technical details:

- “**Robotic Behavior Metamodel**”
- “**Gazebo/Tiago/SmartSoft Scenario**”
(this page provides examples of working behavior models)

Consolidated Implementations of the “Skill Definition”, “Skill Realization”, “Task Definition”, and “Task Realization” modeling languages are available since the **version 3.14 release of the SmartMDS Toolchain**. These are described in the dedicated wiki page on “**Skills for Robotic Behavior**”.

As one of the first early adopters, the “**Mood2Be**” ITP has created a new design tool based on the “Behavior Tree” approach, and integrated it with the Skill Models of the SmartMDS Toolchain. For details, see the wiki page on “**Robotic Behavior in RobMoSys using Behavior Trees and the SmartMDS Toolchain**”. The “Skill Definition” and “Skill Realization” modeling languages also played a major role in the “**MROS**” ITP.

2.2 Composition, Composability, Compositionality (Task T2.3)

Task T2.3 deals with challenges of - and around - software component (meta-)models. This includes the relationship between functional blocks and behavior models, their configurations and interplay within a component and the interaction between components on system level.

This Task T2.3 contributes to the initial baseline of this Deliverable with the RobMoSys component meta-model that also addresses the definition of services that again rely on a clear definition of communication patterns and communication objects. The following Wiki pages provide additional technical details with respect to the Task 2.3:

- “**Component Metamodel**”
- “**Service Metamodel**”
- “**Communication-Pattern Metamodel**”
- “**Communication-Object Metamodel**”
- “**System Component Architecture Metamodel**”

It is important to notice that a component meta-model in isolation is virtually useless as long as it ignores all the other (meta-)models around it. For instance, the component (meta-)models are used

(i.e. referenced) in system (meta-)models for composing the systems out of flexibly configurable building blocks. Therefore, the RobMoSys component meta-model allows the definition of structures with purposefully left open variation points that are used in later development phases (such as e.g. during system composition) to adjust the components to the application-specific system needs. This enables a systematic match-making (also referred to as management of constraints) between required application-specific system constraints and offered variability in components. This match-making ranges from syntactic matches, over matching intervals, up to matching constraints in the most generic form. As an initial baseline in D2.2 it is considered already a great step forward to support the involved developer roles in manually managing the constraints. Some of these match-makings are also automated using constraint solvers in respective implementations of the toolings.

The match-makings as described above appear on different levels such as:

- refining task-net and considering their resource constraints
- matching task-nets with services over skills
- selection of components with their services according to an architectural service design
- matching activity constraints of individual components with application-specific end-to-end requirements of system-level cause-effect-chains
- matching offered and required quality (e.g. accuracy) to minimize resources

Using these composition structures enables traceability of individual design choices and improves exchangeability and composability of individual building blocks because their properties (i.e. variability and constraints) are known and thus can be brought together between the original and exchanged parts.

As one of the first early adopters, the CARVE ITP was able to use the new RobMoSys tools to demonstrate composability and re-usability within the Yarp context. See dedicated Wiki page on **"Using the YARP Framework and the R1 robot with RobMoSys"**. Outside of RobMoSys, the BMWi PAiCE SeRoNet projects exploits the RobMoSys composition approach for establishing a brokerage platform for composable assets for service robots.

2.3 Separation of Roles and Separation of Concerns (Task T2.4)

Task T2.4 is about finding meaningful combinations of related concerns considering the needs of the involved developer roles. These needs and role-specific use-cases are collected in so-called "architectural patterns" which serve as input for the definition of the RobMoSys composition structures (see also Task 2.3).

The following Wiki pages provide additional technical details with respect to Task T2.4:

- **"Architectural Patterns"**
- **"Roles in the Ecosystem"**
- **"Separation of Levels and Separation of Concerns"**

These definitions already provided an initial baseline within the Deliverable D2.2. Moreover, as in the other Tasks above, these definitions have been iteratively refined.

Among other ITPs, the Plug&Bench ITP has directly adopted the idea of the role-specific views for developing representative benchmarks of components. See more details in the Wiki page on **"Benchmarking in the RobMoSys Ecosystem"**.

2.4 Non-functional Properties and QoS Management (Task T2.5)

In contrast to many other approaches in robotics, RobMoSys considers the management of

non-functional (i.e. QoS) aspects as a first class citizen from the very beginning in the overall robotics software development. This is reflected by the Task T2.5.

As an initial baseline for the Deliverable D2.2, this Task contributes a novel performance view that can be used to design and manage performance-related system aspects without violating the component-internal implementation constraints. Further technical details for the performance view can be found in the Wiki page:

- **“Cause-Effect-Chain and its Analysis Metamodels”**
- **“Architectural Pattern for Stepwise Management of Extra-Functional Properties”**

The RoQME ITP specifically extended the RobMoSys body-of-knowledge with respect to measuring different non-functional aspects of a robotic system at run-time (see the dedicated Wiki page on **“Dealing with Metrics on Non-Functional Properties in RobMoSys”**).

Cause effect-chains are just one instance of the generic concept of Dependency Graphs. Dependency graphs can model system-level requirements that span across different components. Examples are properties along data flows, such as, quality and aging of data, but also consistency aspects, triggering along computational chains and arrival time analysis. Further technical details on the concepts, models, and implementations can be found in the Wiki page:

- **“Dependency Graphs”**

3 Tooling

3.1 Introduction

While the RobMoSys composition structures by themselves are independent of any implementation technology, there are different implementation options that can be used. The RobMoSys consortium provides two complementary reference implementations of model-driven tooling using Eclipse modeling tools as the underlying technologies (see figure 3).

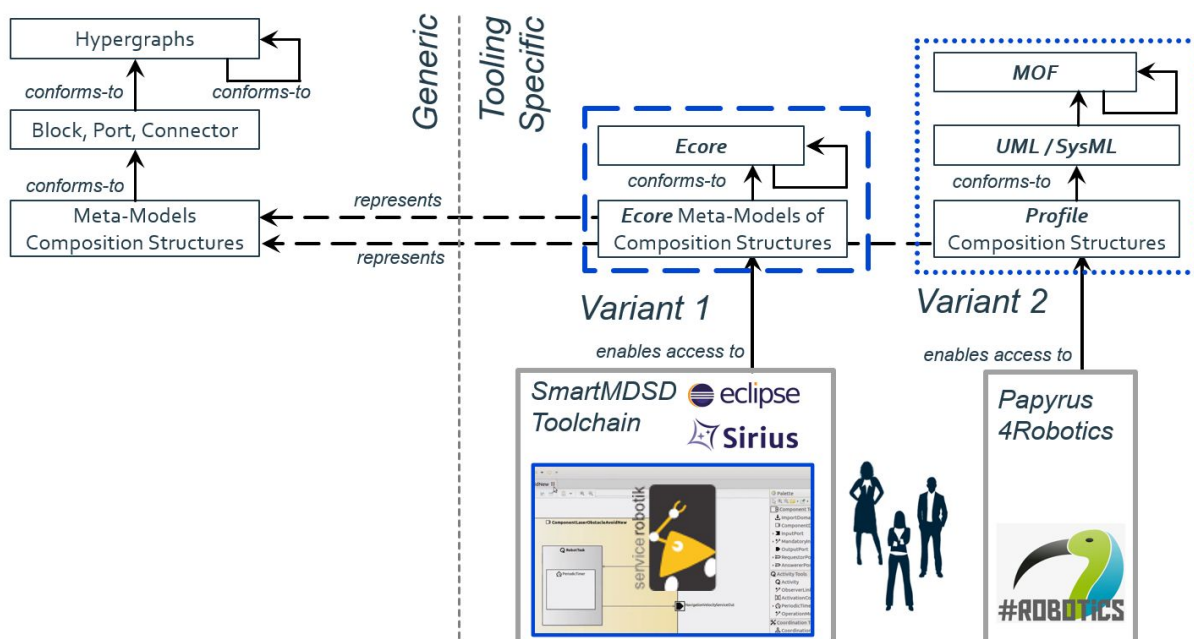


Figure 3: Realization alternatives

Moreover, various (graphical and textual) model editors as well as code generators are available that all conform to the RobMoSys meta-models. Model editors are implemented using Eclipse plugins such as Sirius, Papyrus and Xtext (see figure 4). The software baseline with code generators and the modeling tools in RobMoSys with Domain-Specific Languages (as part of this deliverable) provided other WPs and the open-call ITPs easy access to the RobMoSys concepts, composition structures and software infrastructure. The entry points for technical details for the current software baseline, integrated development environments, modeling tools, code generators, and generated software assets including statements on conformance to the RobMoSys composition structures are the following wiki pages:

- Wiki page on “The SmartSoft World”
- Wiki page on “Papyrus for Robotics”

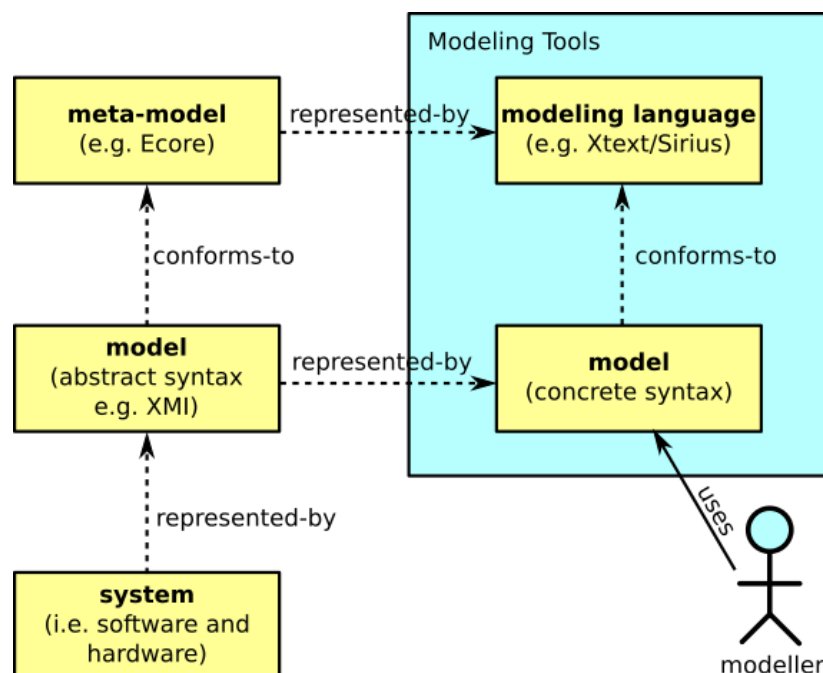


Figure 4: Modeling Tool Implementation Options

The RobMoSys composition structures as well as their realization in the conformant modeling tools have been iteratively refined in terms of coverage and conformance of the RobMoSys concepts. An initial software baseline has been used and improved by ITPs from the first open call round. These refined tools have been used by open call 2 which led to further consolidation and enrichment of the overall RobMoSys body-of-knowledge and tools.

3.2 Integration of Modeling Principles in a Meta-model (Task T2.6)

Task 2.6 is about enabling composition not only for the software aspects but in particular for the modeling tools and the meta-model realizations themselves. Therefore, as introduced above, RobMoSys separates the definition of the RobMoSys composition structures from their realizations using e.g. Eclipse Ecore. Moreover, even the Ecore-based realizations are independent of their actual implementations using e.g. Xtext/Sirius Eclipse plugins. This clear separation of technologies enables dedicated contributions from open calls and independent refinement of the MDSE tooling on different levels as shown in the following Wiki pages:

- “Realization Alternatives”
- “Modeling Principles”

- **“Modeling Twin”**

These technology-separation-structures proved to be useful and extensible within the ITPs of all rounds of open calls.

3.3 Tooling and Run-time Execution (Task T2.7)

Task T2.7 is about the realization of prototypical tooling that underpins the feasibility of modeling approaches from the preceding Tasks T2.2 to T2.6. The following Wiki page provide technical details for the roadmap and current status of the RobMoSys tooling:

- **“Roadmap of Tools and Software”**
- **“The SmartSoft World”**
- **“Papyrus for Robotics”**

The conformance of these tools and software baseline to the RobMoSys composition structures is described in the above wiki pages. In order to ensure that the tools themselves are usable considering the different roles on Tier 2 and Tier 3, some early system examples are developed using these tools. The following Wiki page provides details for the TIAGO navigation scenario:

- **“Gazebo/Tiago/SmartSoft Scenario”**

Meanwhile, the tools have been used to build and to operate several RobMoSys pilots such as:

- **“Intralogistics Industry 4.0 Robot Fleet Pilot”**
- **“Assistive Mobile Manipulation Pilot”**

The tools and the RobMoSys composition structures have been taken up and enriched within the ITPs, as can be seen by the **“Community Corner”** of the RobMoSys Wiki.

A dedicated Wiki page summarizes the models resulting from both, the RobMoSys Pilots and the ITPs, see:

- **“RobMoSys Model Directory”**




4 Appendix

A snapshot as of December 17th, 2020 of the RobMoSys Wiki is attached in the appendix for simple printing. The snapshot can be accessed online via <https://robmosys.eu/wiki-sn-05>. The live version of the wiki can be found at <http://www.robmosys.eu/wiki>.

RobMoSys Wiki

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 \[http://www.robmosys.eu\]](http://www.robmosys.eu).

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 

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 [RobMoSys Document Jumptag](#) to find referred pages. This is a live and evolving wiki, [stable Snapshots](#) are available.

Technical Material for the Second Open Call

We provide a [entry point and reading guide of the technical material for the RobMoSys Second Open Call](#) which is open from February 2019 till end of April 2019. For information about the open call, refer to <https://robmosys.eu/open-call-2/> [<https://robmosys.eu/open-call-2/>].



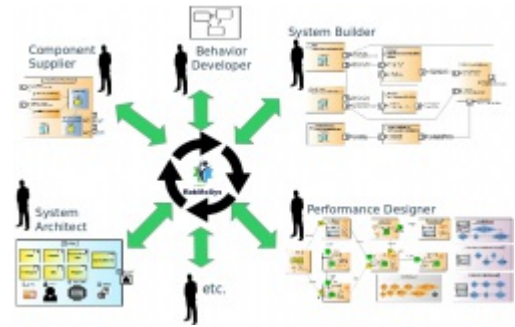
Glossary and FAQ

The [RobMoSys 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:

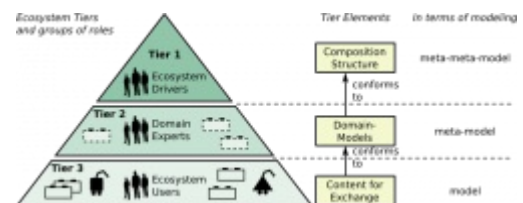
- Behavior Developer
- Component Supplier
- Function Developer
- Performance Designer
- Safety Engineer
- Service Designer
- System Architect
- System Builder



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.

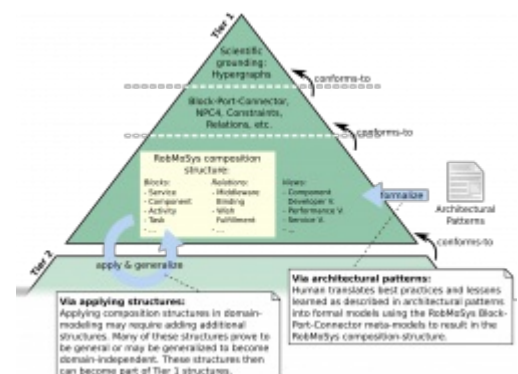


- Separation of Levels and Separation of Concerns
- Architectural Patterns
- Ecosystem Organization and Tiers
- User-Stories
- PC Analogy: Explaining RobMoSys by the example of the PC domain

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.

- 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.

- Flexible Navigation Stack
- Mobile Manipulation Stack
- Motion, Perception, Worldmodel Stack
- Active Object Recognition Stack
- See also the [RobMoSys Model Directory](#)



Tools and Software Baseline

RobMoSys provides a set of tools and a software baseline that conform to the RobMoSys approach. This set can serve as a starting-point for applying the RobMoSys methodology or to extend it.

Tooling

- [RobMoSys Tools, Assets and their Conformance](#)
- [Development Environments and Tools](#)
 - [The SmartMDSD Toolchain: An Integrated Development Environment \(IDE\) for robotics software](#)
 - [Papyrus for Robotics: A set of Papyrus-based DSLs and tools](#)
 - [Groot: an IDE to create, modify and monitor BehaviorTrees](#)
 - [BehaviorTree.CPP: a C++ framework to design, execute, monitor and log robotics behaviors, using Behavior Trees](#)
 - [RoQME Plugins for the SmartMDSD Toolchain: Tooling to enable modeling and monitoring of QoS in robotics systems](#)
 - [eITUS Safety View for Papyrus4Robotics](#)
- [Roadmap of Tools and Software](#)

Tutorials and Documentation

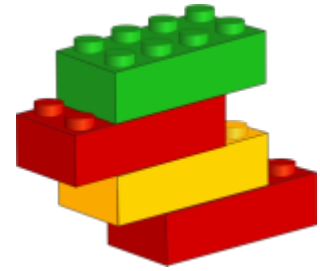
- [For the SmartMDSD Toolchain](#)
- [For Papyrus for Robotics](#)

Usable Domain Models, Components, and Systems

- Browse the [Model Directory](#) to see building blocks available for immediate composition with RobMoSys tooling.

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:
 - Skills for Robotic Behavior
 - Task-Level Composition for Robotic Behavior
 - Service-based composition of software components
 - Composition of algorithms
 - Dependency Graphs
 - Managing Cause-Effect Chains in Component Composition
 - Coordinating Activities and Life Cycle of Software Components

Pilot Skeletons: 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.



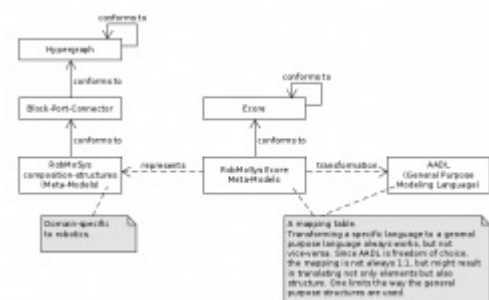
- Goods Transport in a Company:
 - Intralogistics Industry 4.0 Robot Fleet Pilot
- Mobile Manipulation for manufacturing applications on a product line:
 - Flexible Assembly Cell Pilot
 - Human Robot Collaboration for Assembly Pilot
- Mobile manipulation for assistive robotics in a domestic environment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

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.)
- Middleware (DDS)
- Industry 4.0 domain: OPC UA



Community Corner

In this section, we feature early adoptors of RobMoSys methodology, composition structures, or tooling; we present community-related information.



- Get in touch: [Discourse Forum \[https://discourse.robmosys.eu/\]](https://discourse.robmosys.eu/) and [Events \[https://robmosys.eu/events/\]](https://robmosys.eu/events/)
- [Integrated Technical Projects \(ITPs\) of Open Call I \[http://robmosys.eu/itp\]](http://robmosys.eu/itp)
- [Demonstrations and intermediate results \(2nd ITP wave\):](#)
 - [SafeCC4Robot \(SafeCC4Robot ITP\)](#)
 - [A toolchain for verifiable robot deliberation \(SCOPE ITP\)](#)
 - [Metacontrol for ROS2 systems \(MROS ITP\)](#)
 - [Advanced Robot Simulations for RobMoSys \(AROSYS ITP\)](#)
- [Demonstrations and intermediate results \(1st ITP wave\):](#)
 - [Guaranteed Stability of Networked Control Systems \(EG-IPC ITP\)](#)
 - [Safety Assessment of Robotics Systems Using Fault Injection in RobMoSys \(eITUS ITP\)](#)
 - [Robotic Behavior in RobMoSys using Behavior Trees and the SmartMDSD Toolchain \(MOOD2BE ITP\)](#)
 - [Dealing with Metrics on Non-Functional Properties in RobMoSys \(RoQME ITP\)](#)
 - [Using the YARP Framework and the R1 robot with RobMoSys \(CARVE ITP\)](#)
 - [Benchmarking in the RobMoSys Ecosystem \(Plug&Bench ITP\)](#)
- [Demonstrations and intermediate results \(2nd ITP wave\):](#)
 - [Component Composition from Real-time Function Blocks](#)
- [RobMoSys Tools, Assets and their Conformance](#)
- [EU Digital Industrial Platform for Robotics](#)

start · Last modified: 2019/05/28 13:00
<http://www.robmosys.eu/wiki/start>

Changelog

The RobMoSys consortium is continuously updating this wiki to provide early insights. This changelog will help you to track **major changes**.

November / December 2020

- added cross-links between RobMoSys “EU Digital Industrial Platform for Robotics” and euRobotics Topic Group “Stewardship Software Systems Engineering” including link to XITO [<https://www.xito.one/>]
- new wiki pages
 - added “Dependency Graphs”
- completion of Wiki pages
 - completion of task definition metamodel, task realization metamodel
 - completion of skill definition metamodel, skill realization metamodel
 - first details on the active object recognition stack
 - updated dangling links

August/September, 2019

- Major update of the RobMoSys Pilot descriptions

July 25th, 2019

- YARP-RobMoSys Mixed-Port Components
- CARVE Software for verified execution of Behavior Trees

June 27th, 2019

- Several minor updates
- Extended Tutorials for the SmartMDSD Toolchain
 - Cause-Effect-Chains and Activity Architecture [<https://wiki.servicerobotik-ulm.de/how-to-cause-effect-chains:start>]
 - Full list of Tutorials [<https://wiki.servicerobotik-ulm.de/tutorials:start>]. Some highlights are:
 - Developing Your First Software Component [<https://wiki.servicerobotik-ulm.de/tutorials:develop-your-first-component:start>] | video tutorial [https://youtu.be/BRI_HKMilNw]
 - Developing Your First System: Composing Software Components [<https://wiki.servicerobotik-ulm.de/tutorials:develop-your-first-system:start>] | video tutorial [https://www.youtube.com/watch?v=3LNhatjWb_c]
 - Accessing an OPC UA Device: Using the Plain OPC UA Port (DeviceClient) to create a Mixed-Port Component [<https://wiki.servicerobotik-ulm.de/tutorials:opcua-client:start>] | video tutorial [https://youtu.be/uPZ07_Gi3YE]
 - Composing a System with OPC UA Mixed-Port Components [<https://wiki.servicerobotik-ulm.de/tutorials:opcua-client-system:start-toolchain:>] | video tutorial [<https://youtu.be/udQiwRdzCVw>]

- Developing an OPC UA Server: Using the Plain OPC UA Port (ReadServer) [<https://wiki.servicerobotik-ulm.de/tutorials:opcua-server:start>] | video tutorial [<https://youtu.be/Ho7Fr2KefKQ>]
- Mixed-Port for ROS: Accessing ROS nodes from software components [<https://wiki.servicerobotik-ulm.de/tutorials:ros:mixed-port-component-ros-toolchain:>]
- For HowTo's see <https://wiki.servicerobotik-ulm.de/how-tos:start> [<https://wiki.servicerobotik-ulm.de/how-tos:start>]
- Information on [Getting Started With Papyrus4Robotics](#)
- ITP Tooling added
 - [RoQME Plugins for the SmartMDSD Toolchain](#)
- [RobMoSys Tools, Assets and their Conformance](#)
- [Community Corner](#)
 - [Guaranteed Stability of Networked Control Systems \(EG-IPC ITP\)](#)
- [EU Digital Industrial Platform for Robotics](#)
- [Improved readability of Technical User Stories](#)
- [Updated pilot descriptions](#)

January 30th, 2019

- Added a [reading guide for open call 2 technical material](#)
- [RobMoSys Model Directory](#)
- [Mobile Manipulation Stack](#)
- [Updating RobMoSys Tooling with ITP contributions](#)
- Several updates across the wiki to prepare for open call 2

January 25th, 2019

- [Community Corner:](#)
 - [Benchmarking in the RobMoSys Ecosystem](#)

January 18th, 2019

- Information on robot skill modeling:
 - [Skills for Robotic Behavior](#)
 - [Skill Definition Metamodel](#)
 - [Skill Realization Metamodel](#)
- Examples of how the SmartMDSD Toolchain supports skills:
 - [Support of Skills for Robotic Behavior](#)
- [Community Corner:](#)
 - [Safety Assessment of Robotics Systems Using Fault Injection in RobMoSys](#)
- [Refactored The SmartSoft World and The SmartMDSD Toolchain to match restructuring of upstream page.](#)

December 13, 2018

- [Community Corner: Robotic Behavior in RobMoSys using Behavior Trees and the SmartMDSD Toolchain \(MOOD2BE ITP\)](#)

November 30, 2018

- [Community Corner: Dealing with Metrics on Non-Functional Properties in RobMoSys \(RoQME ITP\)](#)

October 5, 2018

- More information about what “modeling” is in RobMoSys: [Modeling Principles - What is "Modeling"?](#)

July 5, 2018

- Started a [Community Corner](#) with a demonstration of [Robotic Behavior in RobMoSys using Behavior Trees and SmartSoft](#).

June 29, 2018

- Updated [Ecosystem Organization](#)
- Added description on how the [SmartMDSD Toolchain](#) supports the [RobMoSys Ecosystem Organization](#) in three composition tiers.
- Added a description of the relation between industry 4.0/OPC UA and RobMoSys: [OPC UA](#)
- Updated or added roles:
 - [Performance Designer](#)
 - [Component Supplier](#)
 - [Behavior Developer](#)
- Updated [Architectural Pattern for Task-Plot Coordination \(Robotic Behaviors\)](#)
- Added [Architectural Pattern for Component Coordination](#)
- Updated [Separation of Levels and Separation of Concerns](#)
- Taking snapshot of the wiki to <https://robmosys.eu/wiki-sn-02> [<https://robmosys.eu/wiki-sn-02>]

June 8, 2018

- Update of the [Flexible Navigation Stack](#)
 - The [Flexible Navigation Stack](#) is an example of domain models / [RobMoSys Composition Tier 2](#) contents.
 - The page now illustrates more details of the service definitions that are defined in this stack.
 - Description of support for the [Flexible Navigation Stack](#) in the [SmartSoft World](#) was added: [Support for the Flexible Navigation Stack](#)
- Update of the [Communication-Pattern Metamodel and view](#).
 - It now points to specific external documents for the definition of the communication patterns.
- Some areas of the wiki cited unpublished Work [Lotz2017] and [Stampfer2017]. These are two doctoral thesis that are very closely related to RobMoSys. They recently appeared online and the links and references have been updated:
 - [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. \[http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2\]](#)
 - [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. \[https://mediatum.ub.tum.de/?id=1362587\]](#)

May 30, 2018

- Added two new wiki pages to illustrate composition
 - [Managing Cause-Effect Chains in Component Composition](#)
 - [Coordinating Activities and Life Cycle of Software Components](#)
- And according description how this is supported by RobMoSys Tooling (here the [SmartMDSD](#)

Toolchain):

- Support for Managing Cause-Effect Chains in Component Composition
- Support for Coordinating Activities and Life Cycle of Software Components

May 2nd, 2018

- Information on Getting Started With Papyrus4Robotics available.

March 2nd, 2018

- Release of the SmartMDSD Toolchain v3 generation. Updated initial documentation, more to follow.

August 21st, 2017

- More Pilot descriptions added

August 1st, 2017

- Started a technical Frequently Asked Questions
- Composition in an Ecosystem now describes composition and provides examples where RobMoSys illustrates it.
- Pilot descriptions added
- Several minor corrections and adjustments.

June 23rd, 2017

- Major improvements of the RobMoSys composition structures
- Several corrections and refinements of multiple pages in the Wiki
- Wiki snapshot freeze [<http://robmosys.eu/wiki-sn-01/>]

June 13st, 2017

- Improvement of the main page/front page

June 6st, 2017

- Several small improvements of pages in the Modeling section
- Refined description of architectural pattern for software components

June 1st, 2017

- Added Service Metamodel
- Added Communication-Pattern Metamodel
- Added Communication-Object Metamodel
- Updated Component Metamodel

May 29th/31st, 2017

- Updated Glossary
- Added Roles and Views in the Ecosystem

- Added General Principles
- Added Modeling details
- Added other approaches in context of RobMoSys
- Added Tools Tools and Software Baseline

May 3rd, 2017

- Initial public release of
 - RobMoSys Glossary
 - Architectural Patterns
 - Separation of Levels and Separation of Concerns
 - Service-Based Composition Approach/Ecosystem Organization

changelog · Last modified: 2020/12/11 18:25
<http://www.robmosys.eu/wiki/changelog>

Snapshots

The RobMoSys consortium is continuously updating this wiki to provide new insights as soon as possible. During the project runtime, snapshots of the wiki are taken to freeze. The snapshots provide a stable reference to work with in project proposals. We encourage you to trace the updates in the live wiki.

Available snapshots:

December 17, 2020

- Snapshot due to deliverable D2.4
- <http://www.robmosys.eu/wiki-sn-05/> [<http://www.robmosys.eu/wiki-sn-05/>]

June 27, 2019

- Snapshot due to deliverable D2.6
- <http://www.robmosys.eu/wiki-sn-04/> [<http://www.robmosys.eu/wiki-sn-04/>]

January 31, 2019

- Snapshot due to opening of the second open call
- <http://www.robmosys.eu/wiki-sn-03/> [<http://www.robmosys.eu/wiki-sn-03/>]

June 29, 2018

- Snapshot due to deliverable D2.5
- <http://www.robmosys.eu/wiki-sn-02/> [<http://www.robmosys.eu/wiki-sn-02/>]

June 23, 2017

- Snapshot prior to opening the first open call.
- <http://www.robmosys.eu/wiki-sn-01/> [<http://www.robmosys.eu/wiki-sn-01/>]

RobMoSys Glossary

The glossary contains descriptions of used terms.

General Terms

Ecosystem

A collaboration model (cf. Bosch2010¹⁾, Iansiti2004²⁾), 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)

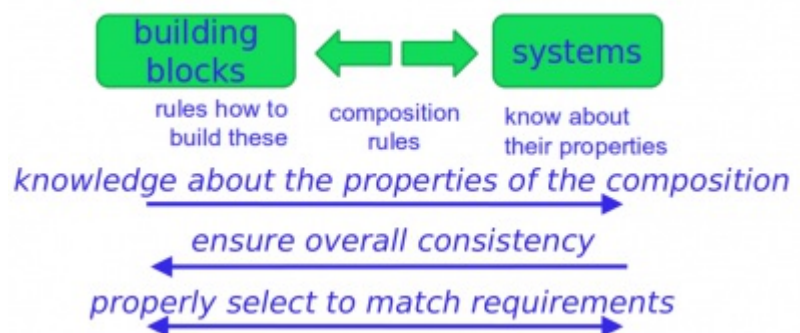
The activity that requires effort to combine components, requiring modification or additional action to make them work with others (see Petty2013³⁾).

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

See also: [System Composition in an Ecosystem](#)

Composability

- 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. Szyperski2002⁴).

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/) [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:

<http://robmosys.eu/wiki/glossary>

- 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 Sentilles2012⁶⁾) 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 modeling 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) [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](#) and [Skills for Robotic Behavior](#)

Service (Level)

See [Separation of Levels and Separation of Concerns](#) and [Service-based Composition](#)

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) reconfiguration
 - 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: 2019/05/20 10:46
<http://www.robmosys.eu/wiki/glossary>

RobMoSys Document Jumppage

The RobMoSys deliverables and open call documents name wiki pages. They can be accessed through this jumppage and will be kept updated according to the evolving wiki structure.

- Wiki page on [Getting Started With Papyrus4Robotics](#)
- Wiki page on [Papyrus for Robotics](#)
- Wiki page on [SmartMDSD Toolchain Support for the RobMoSys Ecosystem Organization](#)
- Wiki page on [The SmartMDSD Toolchain](#)
- Wiki page on [The SmartSoft World](#)
- Wiki page on [Roadmap of Tools and Software](#)
- Wiki page on [Gazebo/Tiago/SmartSoft Scenario](#)
- Wiki page on [Managing Cause-Effect Chains in Component Composition](#)
- Wiki page on [Dependency Graphs](#)
- Wiki page on [Coordinating Activities and Life Cycle of Software Components](#)
- Wiki page on [Task-Level Composition for Robotic Behavior](#)
- Wiki page on [Flexible Navigation Stack](#)
- Wiki Page on [Architectural Patterns](#)
- Wiki page on [Architectural Pattern for Component Coordination](#)
- Wiki page on [Architectural Pattern for Task-Plot Coordination \(Robotic Behaviors\)](#)
- Wiki page on [Architectural Pattern for Stepwise Management of Extra-Functional Properties](#)
- Wiki Page on [Ecosystem Organization](#)
- Wiki page on [Behavior Developer](#)
- Wiki page on [Component Supplier](#)
- Wiki page on [Performance Designer](#)
- Wiki Page on [Roles in the Ecosystem](#)
- Wiki Page on [PC Analogy: Explaining RobMoSys by the example of the PC domain](#)
- Wiki Page on [Separation of Levels and Separation of Concerns](#)
- Wiki Page on [General Principles](#)
- Wiki Page on [User-Stories](#)
- Wiki page on [RobMoSys Composition Structures](#)
- Wiki Page on [Scientific Grounding: Hypergraph and Entity-Relation model](#)
- Wiki page on [Robotic Behavior Metamodel](#)
- Wiki page on [Communication-Object Metamodel](#)
- Wiki page on [Communication-Pattern Metamodel](#)
- Wiki page on [Component Metamodel](#)
- Wiki page on [Cause-Effect-Chain and its Analysis Metamodels](#)
- Wiki page on [Service Metamodel](#)
- Wiki page on [System Component Architecture Metamodel](#)
- Wiki Page on [Block-Port-Connector](#)
- Wiki page on [Modeling Principles](#)
- Wiki page on [Modeling Twin](#)
- Wiki Page on [Preliminary Ecore implementation of ER and BPC meta-models](#)
- Wiki page on [Realization Alternatives](#)
- Wiki page on [Tier 1 in Detail](#)
- Wiki Page on [Tier 1 Structure](#)
- Wiki page on [Views](#)
- Wiki page on [OPC Unified Architecture \(OPC UA\)](#)

- Wiki Page on Your Role in the RobMoSys Ecosystem is part of the front-page

jumppage · Last modified: 2020/12/07 13:09
<http://www.robmosys.eu/wiki/jumppage>

RobMoSys Tools, Assets and their Conformance

List of RobMoSys Conformant Assets

The below list contains assets that are conformant to RobMoSys. Please click the links to find a “conformance sheet” that describes the individual level of conformance:

Tooling

Conformance sheets are being prepared for the following assets:

- [The SmartMDSD Toolchain: An Integrated Development Environment \(IDE\) for robotics software](#)
- [Papyrus for Robotics: A set of Papyrus-based DSLs and tools](#)

The conformance of the following is currently being checked:

- [Groot: an IDE to create, modify and monitor BehaviorTrees](#)
- [BehaviorTree.CPP: a C++ framework to design, execute, monitor and log robotics behaviors, using Behavior Trees](#)
- [RoQME Plugins for the SmartMDSD Toolchain: Tooling to enable modeling and monitoring of QoS in robotics systems](#)
- [eITUS Safety View for Papyrus4Robotics](#)
- [CARVE Software for verified execution of Behavior Trees](#)

Models and Components

Conformance sheets are being prepared for the following assets. At the moment, you will find RobMoSys conformant assets via the [RobMoSys Model Directory](#).

- [Composable Software Components created with the SmartMDSD Toolchain are conformant to RobMoSys](#)
- [YARP-RobMoSys Mixed-Port Components](#)

Methodology

Here we collect approaches and methodologies that are not part of the RobMoSys composition structures but conformant thereto.

- Methodologies developed by the EG-IPC ITP

What is Conformance?

Conformance assesses and describes the **degree to which a specific asset follows the RobMoSys methodology** and approach. An asset can be tools, models, components or methodologies. An asset that is conformant to RobMoSys can use the conformance label (logo) to make its conformance visible.

The purpose of conformance is to relate different assets in the RobMoSys ecosystem to the RobMoSys approach in order to create a “map” of the ecosystem. Conformance thus serves as a guide for users to choose the right assets for their task, role, or intended purpose of use.



The Conformance Label

Who can use the Conformance Label and how to apply?

The label can be used by any provider of an asset (e.g. authors, developers) that is listed on the RobMoSys wiki <https://robmosys.eu/wiki/conformance:start> [<https://robmosys.eu/wiki/conformance:start>]. To find out how to apply for the conformance label, see below. An asset with incubator label means: the conformance of this asset to RobMoSys is currently being investigated/assessed.

Providers of assets can request to be listed on the wiki page. This will trigger the RobMoSys consortium to investigate and assess the conformance via its internal wiki publishing process. Please get in touch with us and provide a facts sheet of your asset following the [conformance sheet template](#) and structure in the above “List of RobMoSys Conformant Assets”.

How to Use the Conformance Label?

Once you are listed on the “List of RobMoSys Conformant Assets” you are allowed to use the conformance label as follows:

1. Add the conformance logo to your asset website and link it to <https://robmosys.eu/wiki/conformance:start> [<https://robmosys.eu/wiki/conformance:start>]
2. As an alternative or addition, you can also use the sentence “this asset is RobMoSys conformant”. Make the words “RobMoSys conformant” a link to <https://robmosys.eu/wiki/conformance:start> [<https://robmosys.eu/wiki/conformance:start>]
3. Send us a screenshot of the website.

Find below 100 and 150 pixel versions of the label for your website:

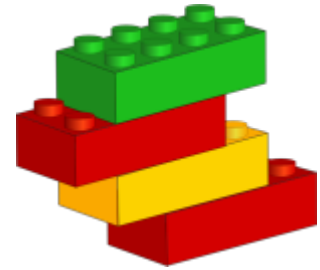


RobMoSys Conformance Explained

This section is work in progress: it will illustrate how to describe the conformance of assets.

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.

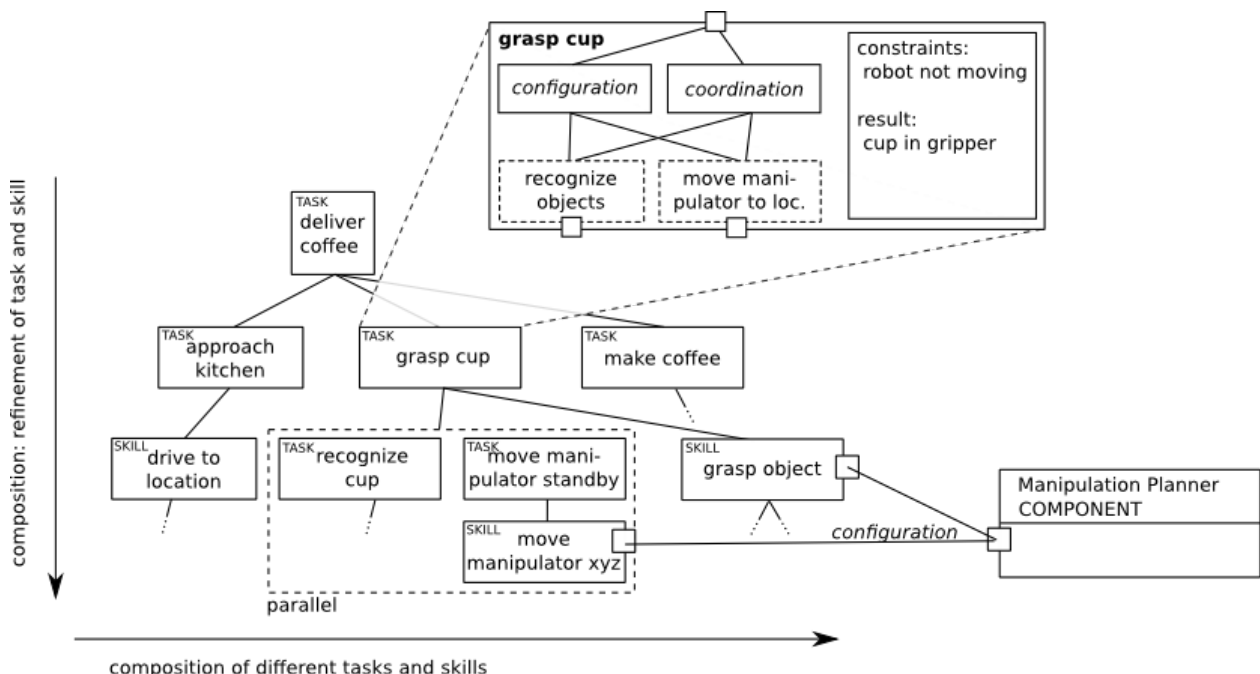


- [Introduction to Composition in an Ecosystem](#)
- We illustrate composition by:
 - [Skills for Robotic Behavior](#)
 - [Task-Level Composition for Robotic Behavior](#)
 - [Service-based composition of software components](#)
 - [Composition of algorithms](#)
 - [Dependency Graphs](#)
 - [Managing Cause-Effect Chains in Component Composition](#)
 - [Coordinating Activities and Life Cycle of Software Components](#)

composition:start · Last modified: 2020/12/07 10:11
<http://www.robmosys.eu/wiki/composition:start>

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 collision-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>]

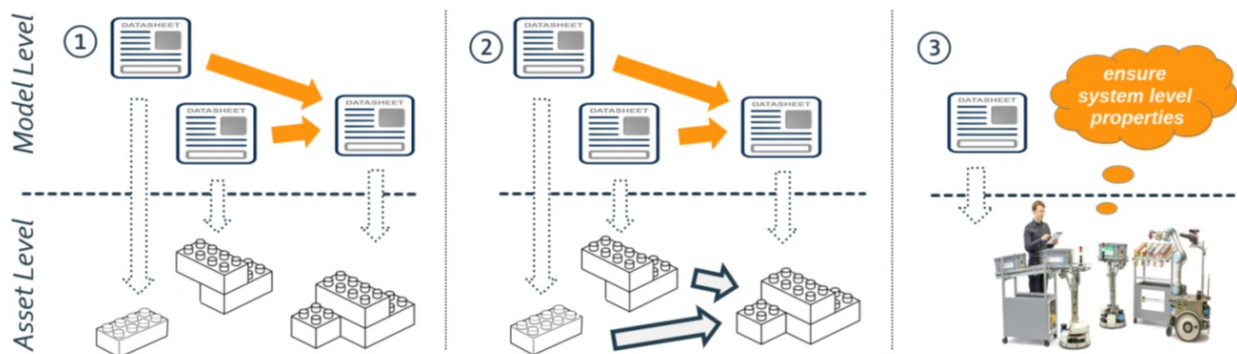
composition:task:start · Last modified: 2019/05/20 10:49
<http://www.robmosys.eu/wiki/composition:task:start>

Dependency Graphs

Dependency graphs can model system-level requirements that span across different components. Examples are properties along data flows, such as quality and aging of data, but also consistency aspects, triggering along computational chains and arrival time analysis.

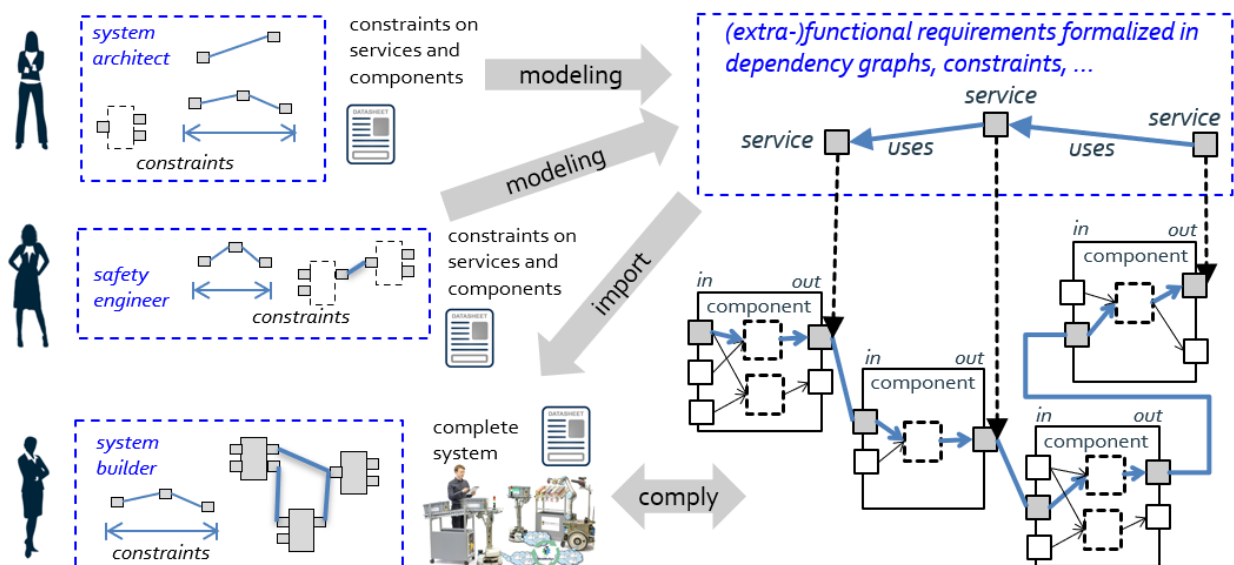
A dependency graph can also be used to express requirements on data privacy, for example, “there must not be any link between a service providing a raw camera image and a service connecting to the network outside the robot”. A dependency graph can also express that, for example, the blur of a camera image depends on the speed of the robot. In order to keep the blur below a given threshold, one can derive the related maximum allowed velocity. A dependency graph can guide the system builder in selecting and configuring components according to the input from a system architect, a safety engineer and others. Thereto, the foreseen variation points, explicated in the data sheet, are exploited.

Dependency Graphs and System Composition



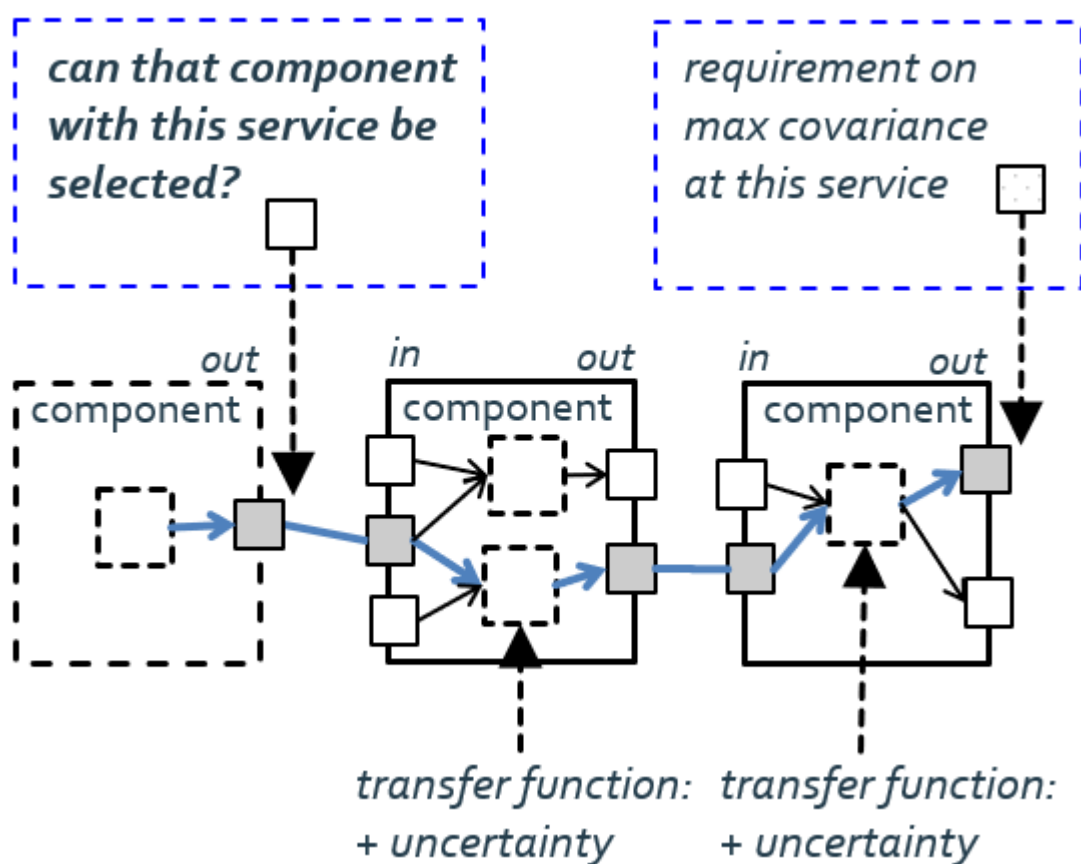
Dependency graphs can also be extracted from a virtual system composition (based on data sheets only, see above Figure). Before building the real system, one can already check whether the dependency graphs in the virtual system comply with the constraints of the dependency graphs representing the requirements. Tools can be used to try different configurations and to check for matches. In that way, dependency graphs also help to implement traceability from requirements to fulfillment by configurations of variation points. Dependency graphs also serve at runtime as sanity checks (before finally implementing a runtime decision) and for monitoring integrity.

Example: Dependency Graphs and Services



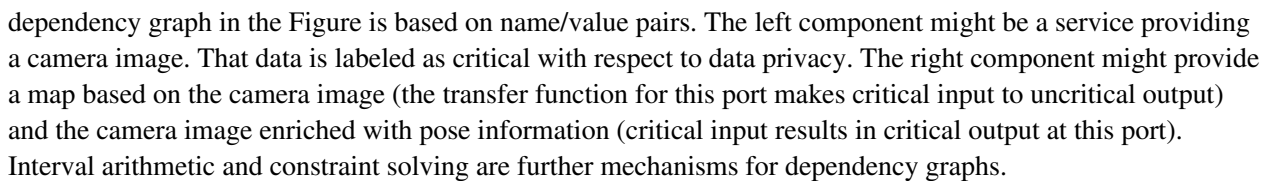
From a technical point of view, the meta-model of a dependency graph is again based on the entity-relation-model provided in layer 2 of Tier 1, which refines into different variants at layer 3 of Tier 1. The dependency graph shown in the above Figure is a simple one that expresses dependencies between services (entity service, relation uses). It gets checked by a mapping between the dependency graph and the graph resulting from the data sheet composition.

Example: Dependency Graphs and Error Propagation



propagation through a data flow across components. Transfer functions of a component specify what additional uncertainty comes on top. For example, the system builder can check whether the selected component is good enough to match the requirement at the end of the processing chain.

The



Dependency Graph Extensions for the SmartMDS Toolchain (SmartDG) [https://wiki.servicerobotik-ulm.de/tutorials:start#lesson_8dependency-graph_extensions_for_smartmdsd_toolchain_smartdg]

* Cause-Effect-Chains are another example of Dependency Graphs.

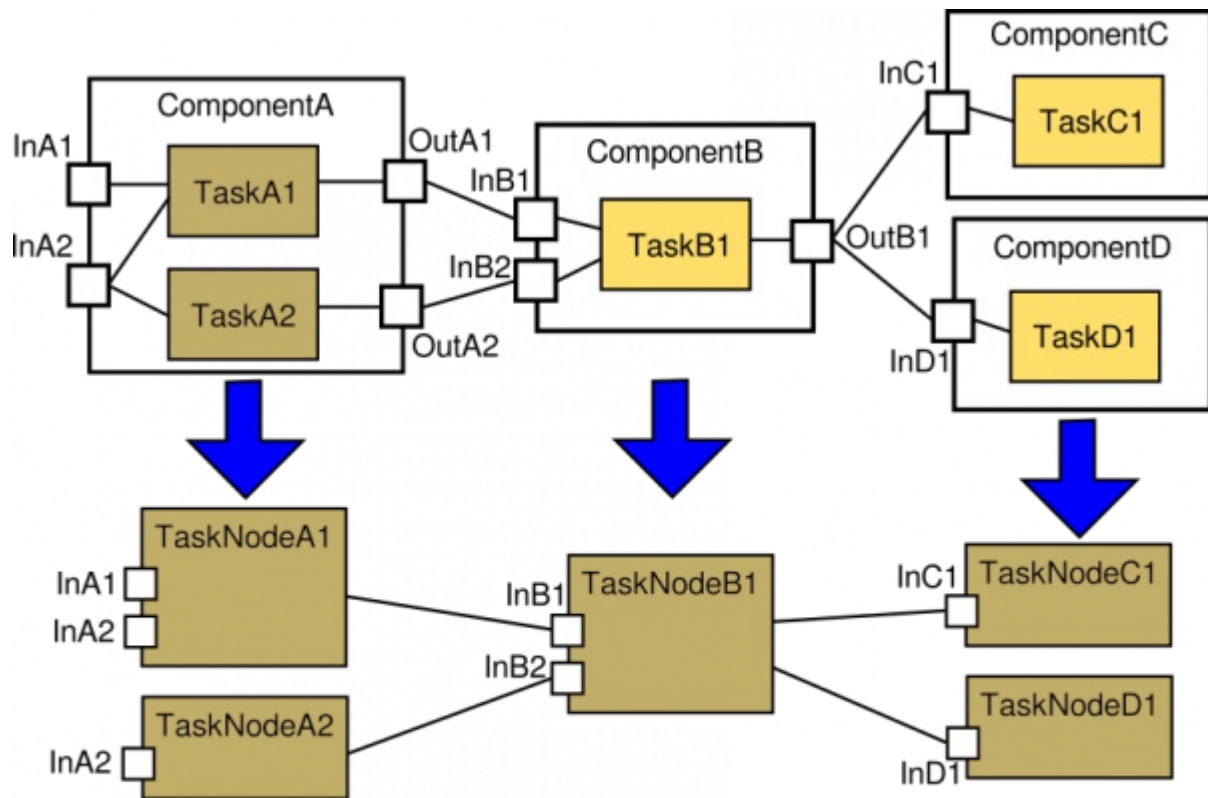
Managing Cause-Effect Chains in Component Composition

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 (SIMPAN). San Francisco, CA, USA, Dec. 2016, pp. 170–176. [LINK \[http://dx.doi.org/10.1109/SIMPAN.2016.7862392\]](http://dx.doi.org/10.1109/SIMPAN.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/) [<https://ptolemy.berkeley.edu/publications/papers/87/synchdataflow/>], and [Petri Net](https://en.wikipedia.org/wiki/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).



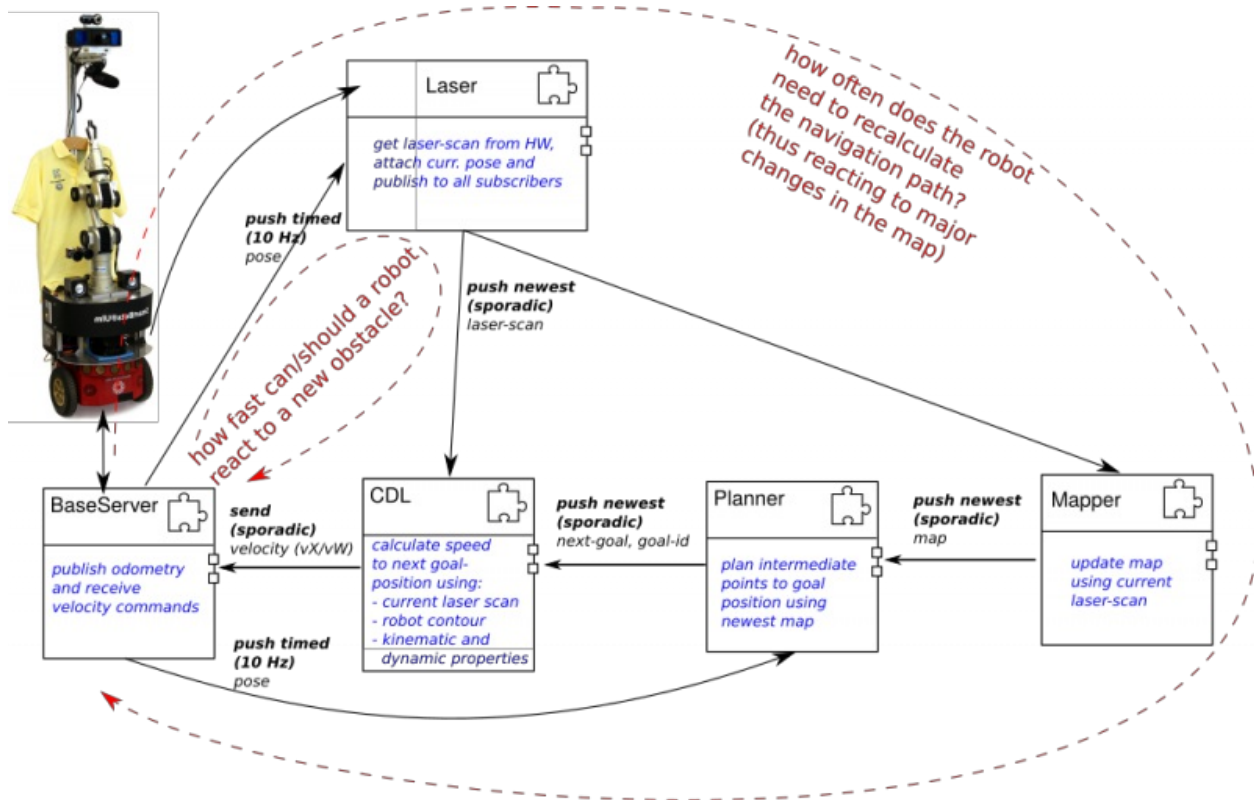
There is a relation between the System Component Architecture Metamodel (see also the System Builder Role and the System Configuration View) and the Cause-Effect Chain and its Analysis Metamodels (see also the Performance Designer Role and the Performance View). The figure above shows an illustration 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.



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

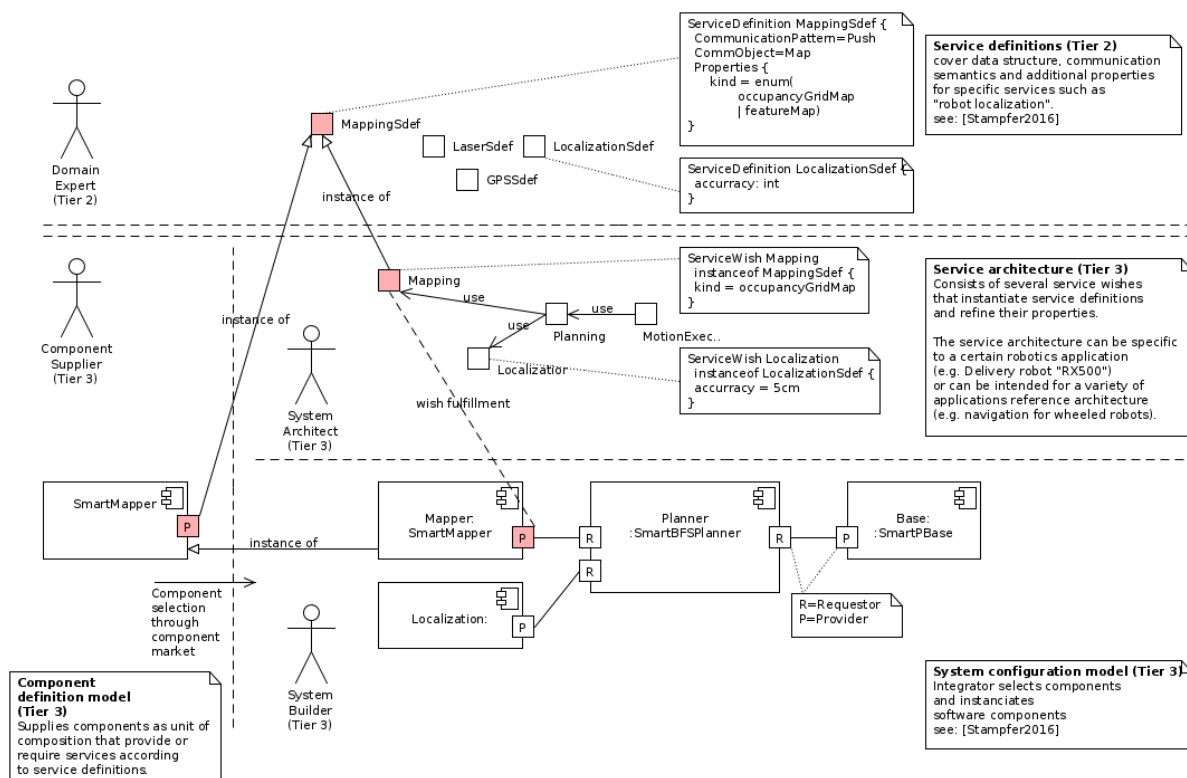
- The following page discusses the concrete models of this example using the [SmartMDSD Toolchain: Example Use-Case for Managing Cause-Effect Chains in Component Composition using the SmartMDSD Toolchain](#)

See also:

- [Architectural Pattern for Stepwise Management of Extra-Functional Properties](#)

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”.



[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>]

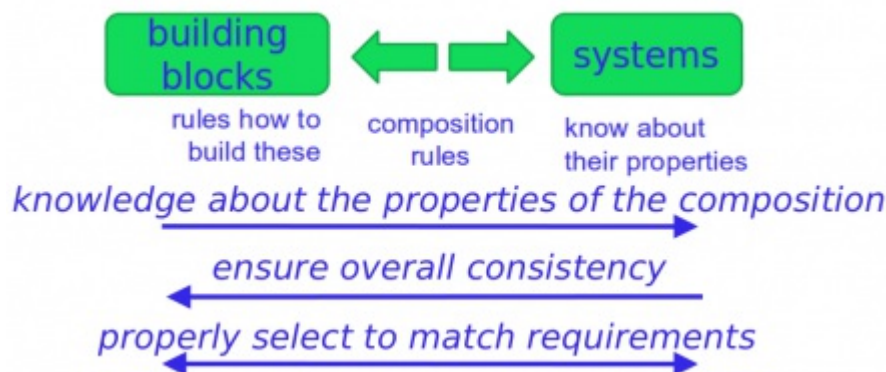
composition:service-based-composition:start · Last modified: 2019/05/20 10:49
<http://www.robmosys.eu/wiki/composition:service-based-composition:start>

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.



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.

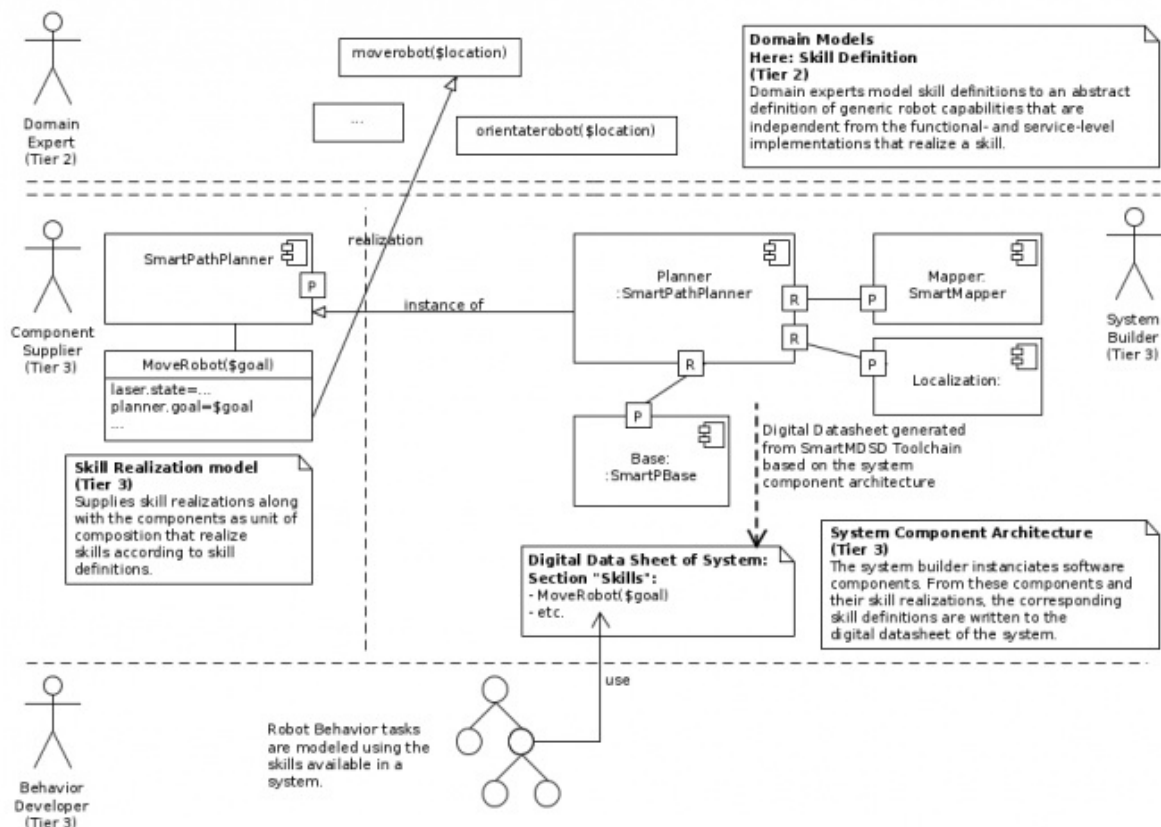
composition:introduction · Last modified: 2019/05/20 10:47
<http://www.robmosys.eu/wiki/composition:introduction>

Skills for Robotic Behavior

Robotics behavior coordination is the composition and coordination of functionalities (other terms: basic capabilities, skills, etc.) to realize a task a robot should perform. A robotic behavior realizing a logistics task would for example use the robots functionalities to move itself and to manipulate things to realize an order picking task. To do so, the robotic behavior needs (on the lower end) to deal with the coordination of the software components of the system that provide the functionality.

To allow for composition and to enable the separation of roles, robotics behavior blocks can be separated into different levels of abstraction: skill and task. While tasks describe how a robot does something in an abstract and independent manner, a skill provides access to functionalities realized by components for the usage within tasks. Skills therefore lift the level of abstraction from functional and service to a skill abstraction level, being usable from a task abstraction level for robotics behavior development.

The following sections describes the big picture how the separated roles will collaborate for robotics behavior development (see the following figure as well):



Skill Definition - Domain Experts (Tier 2)

The separation of the skill definition, modeling the interface and the skill realization enables the replacement and the composition of components (providing the same skill) and decouples the component from the Behavior Developer and the technology used to realize robotics behaviors on task level. The role of the Domain Expert

therefore is able to define skill definitions modeling the interface of skills on tier 2, not yet deciding how a skill is realized. As a result, the skill definition is any component that realizes it.

Skill Realization - Component Developers (Tier 3)

The skill definitions modeled by the Domain Experts are used by the role of the Component Developer to realize the coordination of its component, making the components functionality accessible for behavior coordination. The role of the Component Developer is decoupled from the role of the Behavior Developer and can realize the components functionality and the skill according to the skill definition only.

Behavior Development - Robotics Behavior Developer (Tier 3)

The role of the Robotics Behavior Developer is able to realize the behavior tasks (abstraction level) using the skill definitions as interface to the functionalities later provided by the components. In case the behavior is developed for a known system the list of available skills can be received from the system digital data sheet, provided by the role of the System Builder.

Skill Usage at Run-Time

To make use of existing skills (see system digital data sheet), a task level behavior coordination approach can use existing skills via a run-time skill interface. This enables the usage of different behavior coordination approaches without the need to realize the skills (accessing the components, lifting the level of abstraction) over and over again.

An example realization of the interface can be seen within the software component “ComponentSkillInterface” and is accessible via a json based protocol. The interface is kept lightweight and by purpose without RobMoSys communication patterns in order to enable the simple integration with existing infrastructure. It is realized using plain sockets or zeromq JSON communication.

The protocol is straight forward and can be sketched as follows. For further details see the software component [ComponentSkillInterface \[https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/ComponentSkillInterface\]](https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/ComponentSkillInterface).

Push Skill for Execution

```
{ "msg-type" : "push-skill" , "id" : <ID> , "skill" : { "name" : "<SKILLNAME>" , "skillDef
```

Skill execution result-msg:

```
{ "msg-type" : "skill-result" , "id" : <ID> , "result" : { "result" : "<SUCCESS|ERROR>" ,
```

Abort Skill

Abort current skill:

```
{ "msg-type" : "abort-current-running-skill" }
```

```
{ "msg-type" : "abort-current-running-skill-result" , "result" : "<SUCCESS|ERROR>" }
```

Abort skill:

```
{ "msg-type" : "abort-skill" , "id" : 2 }
```

```
{ "msg-type" : "abort-skill-result" , "id" : 2 , "result" : "SUCCESS" }
```

Optional Information Query

```
{ "msg-type" : "query" , "query" : { "type" : "<INFORMAION TO QUERY>" }}
```

RobMoSys Modeling Support

- Skill Realization Metamodel
- Skill Definition Metamodel
- Task Realization Metamodel
- Task Definition Metamodel
- Component-Definition Metamodel
- Behavior Developer
- Component Supplier
- Service Designer
- Architectural Pattern for Task-Plot Coordination (Robotic Behaviors)
- Separation of Levels and Separation of Concerns
- Ecosystem Organization

RobMoSys Tooling Support

The following page demonstrates how skills modeled using the [SmartMDSD Toolchain: Support of Skills for Robotic Behavior](#)

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>]

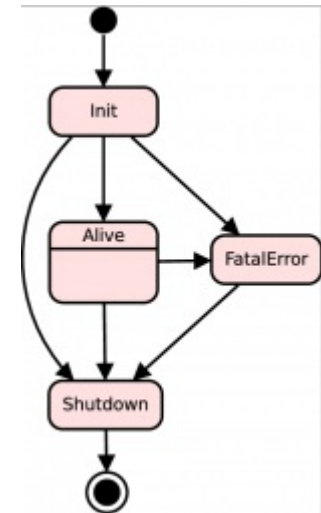
composition:skills:start · Last modified: 2020/12/04 13:43
<http://www.robmosys.eu/wiki/composition:skills:start>

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 lifecycle 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 **operation 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 be 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 and 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. 

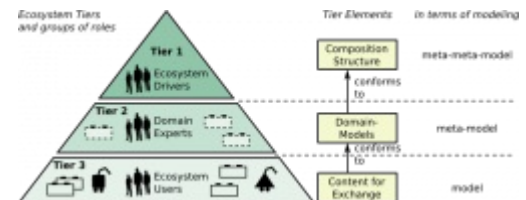
RobMoSys Tooling Support

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: 2019/05/20 10:49
<http://www.robmosys.eu/wiki/composition:component-activities:start>

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](#)

general_principles:start · Last modified: 2019/05/20 10:47
http://www.robmosys.eu/wiki/general_principles:start

Technical 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

Based on model driven tools, RobMoSys enables development of composable navigation components, with all their explicated properties, variation points, resource requirements etc. (the modeling twin / data sheet).

As system builder,

- I want to be able to compose robotics navigation components from commodity building blocks according to my needs (predictable properties, matching system requirements assured, free from interference) without having to develop such components from scratch every time, while ensuring my system requirements are matched and their properties are traceable.

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. A **data sheet** is a document that contains everything you need to know to be able to use a 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 **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).

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 a 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)

Consider the scenario in which a hardware device is broken and the identical device is not available anymore (deprecated, discontinued, only next version available). For example: when replacing laser-based localization with visual localization hardware, when replacing a 6 DOF manipulator with a 5 DOF manipulator, etc.

As a system builder,

- I want to check whether all my relevant system-level properties and constraints are matched when replacing an old device with a new one, while knowing how to configure the new HW.
- When removing a software component from a system, I want to know which constraints define the now empty spot in my design in order to fill it with the proper configuration to match again the system-level properties.

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

Composition of components

As a System builder,

- I want to predict selected properties of the composition of various software components given their individual properties, their configurations, their composition. (*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.

Quality of Service

As a system builder,

- 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:

1. 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
2. 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 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

As a system builder,

- When extending a system, I want to know that I do not interfere with the already setup components, allocated 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

Separation of roles 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.

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.

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. As a system builder,

- I would like to have support regarding 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.
- I want to have access to analysis tools 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

As a system builder,

- I want reusable and composable task blocks which express knowledge about how to execute tasks (action plot) and what are good ways to execute tasks (qualities).
- I want to manage 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.
- I model constraints for particular applications and environments.

As a system builder,

- I want to import safety 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.)

NOTE: 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: 2019/05/20 10:47
http://www.robmosys.eu/wiki/general_principles:user_stories

Ecosystem Organization

Composition Tiers

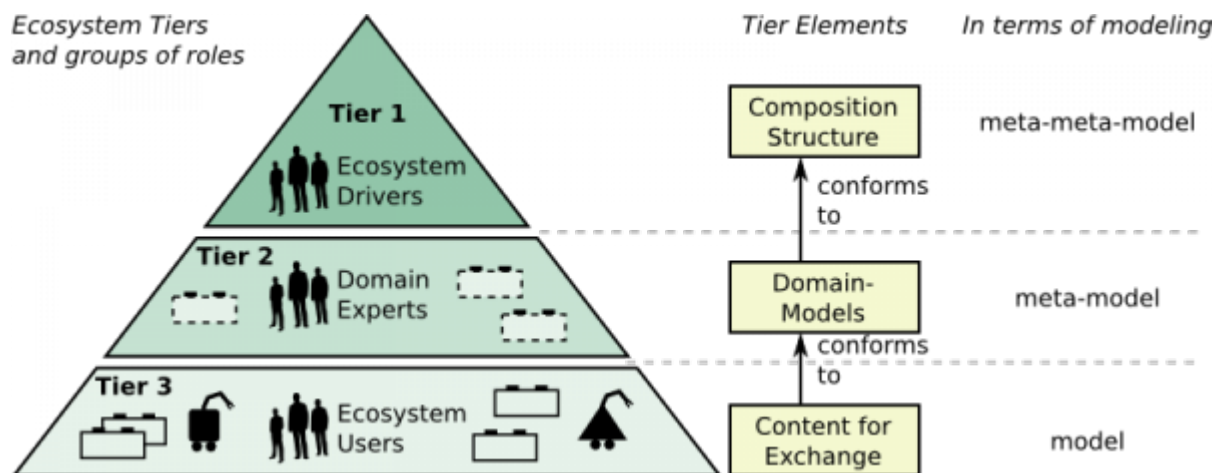
The general composition structure distinguishes three tiers.

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

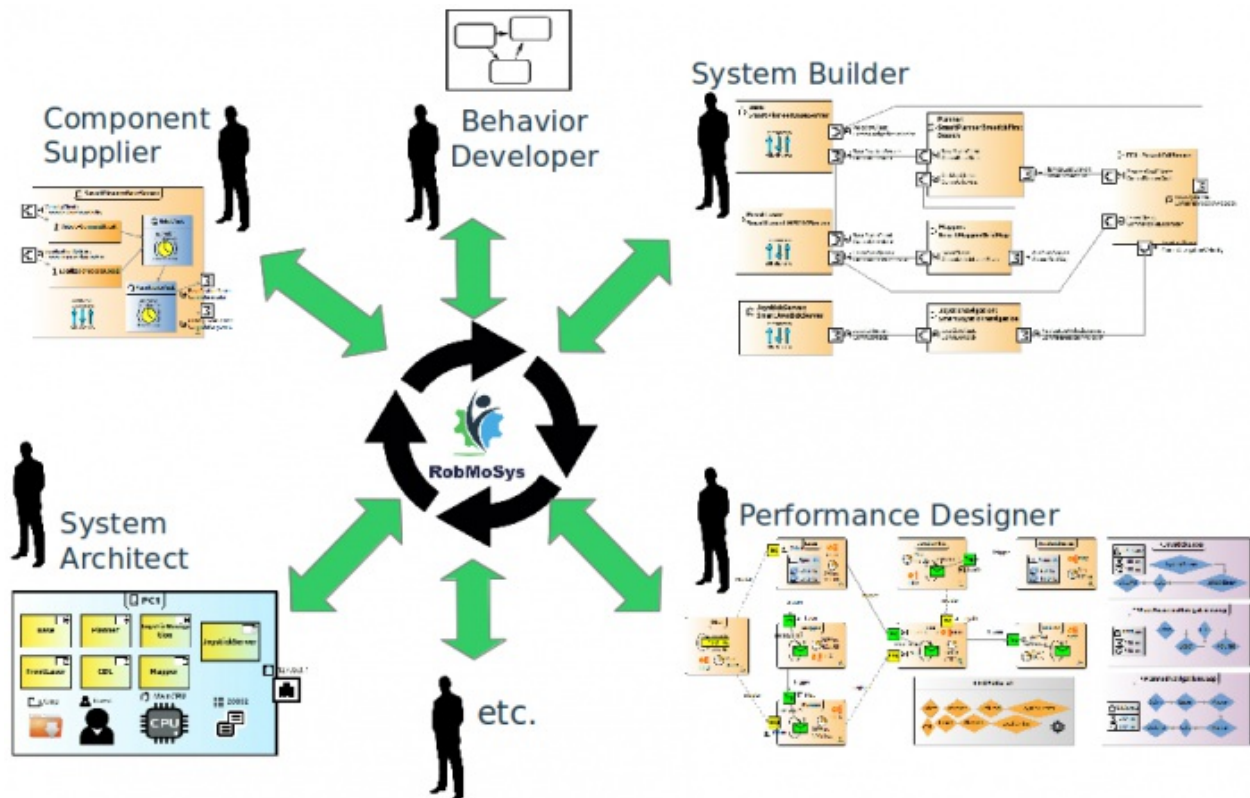
This document contains material from:

- 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]

general_principles:ecosystem:start · Last modified: 2019/05/20 10:49
http://www.robmosys.eu/wiki/general_principles:ecosystem:start

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](#)



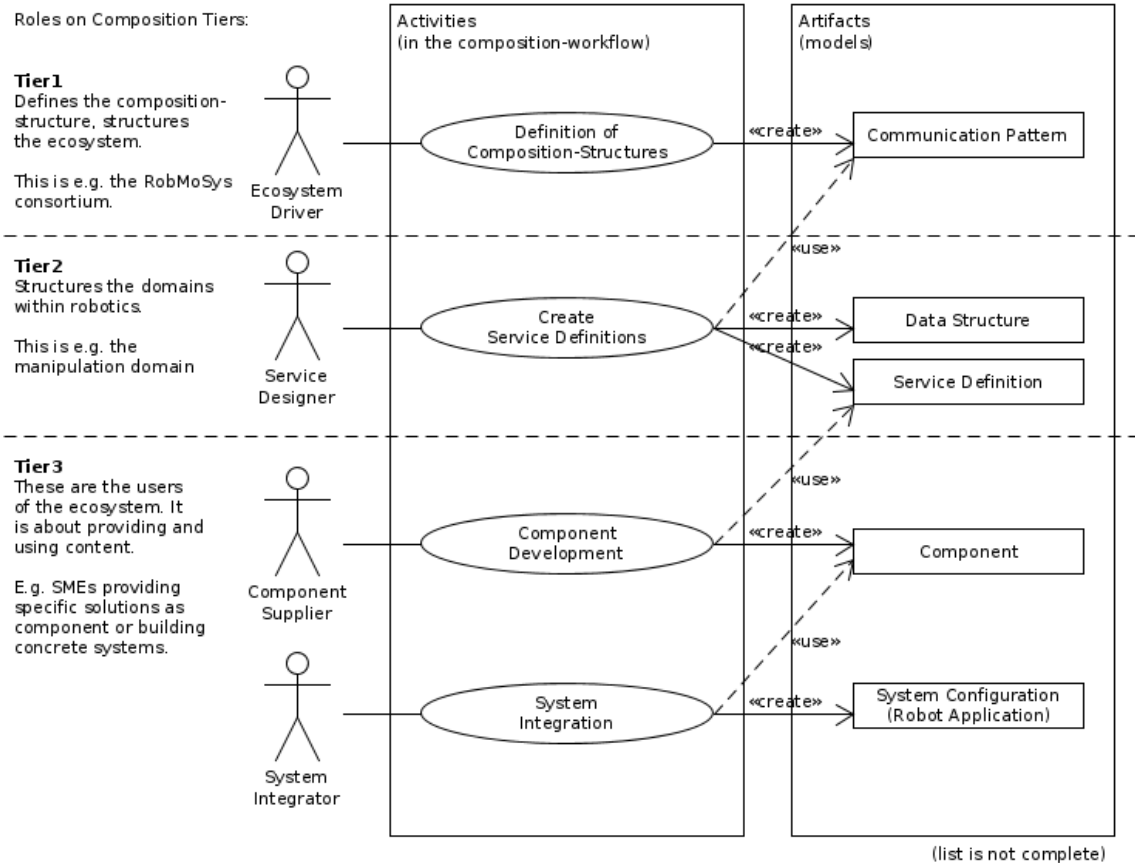
List of Roles

(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.



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](#)

general_principles:ecosystem:roles · Last modified: 2019/05/20 10:49
http://www.robmosys.eu/wiki/general_principles:ecosystem:roles

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

general_principles:ecosystem:roles:behavior_developer · Last modified: 2019/05/20 10:51
http://www.robmosys.eu/wiki/general_principles:ecosystem:roles:behavior_developer

Function Developer

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

Synonym:

- none

Related views and models:

-  to be defined

See also:

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

general_principles:ecosystem:roles:function_developer · Last modified: 2019/05/20 10:51
http://www.robmosys.eu/wiki/general_principles:ecosystem:roles:function_developer

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 by component 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 RobMoSys Glossary).

Related views and models:

- System Component Architecture Metamodel

See also:

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

general_principles:ecosystem:roles:system_builder · Last modified: 2019/05/20 10:51
http://www.robmosys.eu/wiki/general_principles:ecosystem:roles:system_builder

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

general_principles:ecosystem:roles:performance_designer · Last modified: 2019/05/20 10:51
http://www.robmosys.eu/wiki/general_principles:ecosystem:roles:performance_designer

System Architect

This role on Tier 3 designs a system architecture based on existing service definitions from service 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

general_principles:ecosystem:roles:system_architect · Last modified: 2019/05/20 10:51
http://www.robmosys.eu/wiki/general_principles:ecosystem:roles:system_architect

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

general_principles:ecosystem:roles:safety_engineer · Last modified: 2019/05/20 10:51
http://www.robmosys.eu/wiki/general_principles:ecosystem:roles:safety_engineer

Service Designer

These are the domain experts on Tier 2 that design individual service definitions for use by Tier 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>]

general_principles:ecosystem:roles:service_designer · Last modified: 2019/05/20 10:51
http://www.robmosys.eu/wiki/general_principles:ecosystem:roles:service_designer

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 by 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: 2019/05/20 10:51
http://www.robmosys.eu/wiki/general_principles:ecosystem:roles:component_supplier

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 like 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.

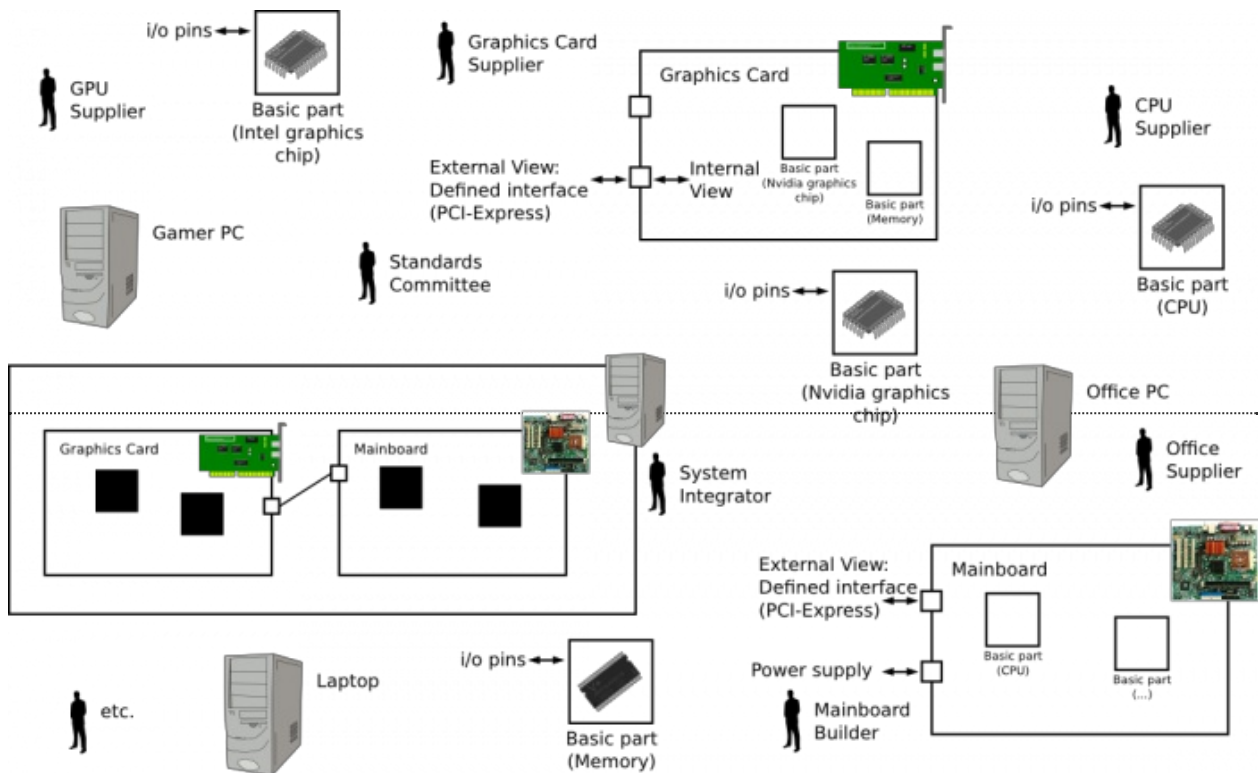
In robotics, there is neither a superordinate structure such as PCIe, no data-sheets for components, and no support for selecting components.

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

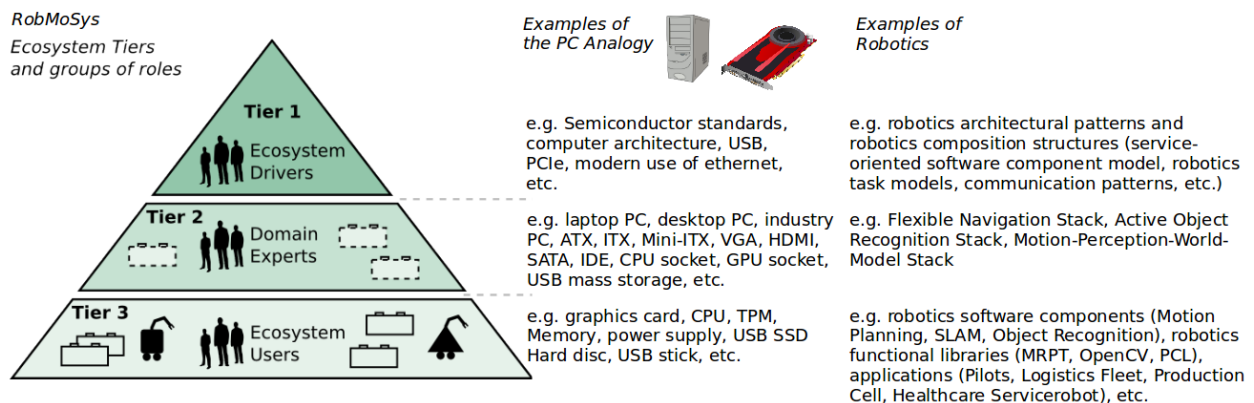
Exposing properties in a data sheet does not mean to expose intellectual property (IP). It is only about exposing the information that is relevant to use it (e.g. external view / interface), size of the device, power supply, etc. Information about the internals of the building block (circuit layout,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

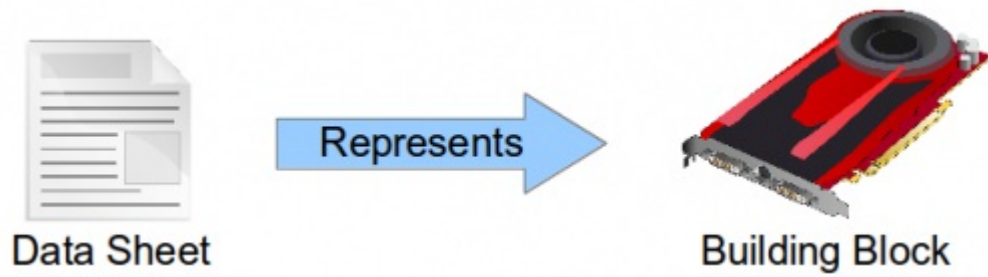


The RobMoSys composition Tiers illustrated with examples of the PC domain.

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 transferring 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 Domain to learn about the benefits.



general_principles:pc_analogy:start · Last modified: 2019/05/20 10:49
http://www.robmosys.eu/wiki/general_principles:pc_analogy:start

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: 2019/05/20 10:49
http://www.robmosys.eu/wiki/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 1

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.](#)

Example 2

Data privacy requirements can be considered as another example of an extra-functional property:

- the navigation system of a hospital robot requires an onboard camera
- the overall robot is composed out of several interconnected components forming different data flows
- the raw camera images make their way through different components
 - all services along a data flow that provide camera images (such as labeled images) are critical with respect to privacy
 - however, camera images can also be used to read door plates. The robot thus can report the room where it is. This does not contain any camera images anymore but just the name of the room. Thus, this service is not critical with respect to privacy
- a system-level privacy requirement might be given as follows:
 - the raw images of a camera must not leave the robot system
- thus, the system builder needs to do the following checks:
 - there must be no data flow from the camera via different components to services accessible from outside that still contain data that is critical with respect to privacy
 - there must be only such services reachable from outside the robot where all the data flows do not come from services considered as critical with respect to privacy
 - all the components that convert “critical data” (consume) into “uncritical data” (provide services) need to be trustworthy (as these connect different criticality regions). Of course, this holds true for any component having at least one (or all) ports dealing with critical data.

This example is described with more details in a dedicated wiki page: [Dependency Graphs for System Level Properties.](#)

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:stepwise_management_nfp · Last modified: 2020/12/07 10:45
http://www.robmosys.eu/wiki/general_principles:architectural_patterns:stepwise_management_nfp

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 the [Service 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.

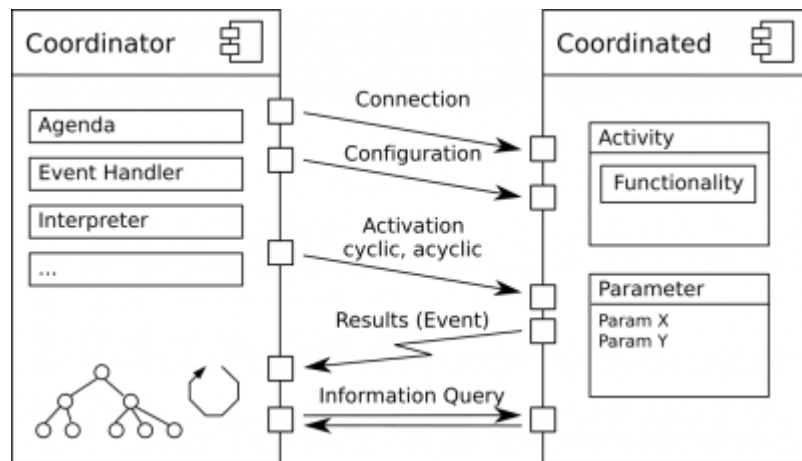
The coordination of the component needs to be possible without binding the behavior models (task level description) to a concrete component.

Solution

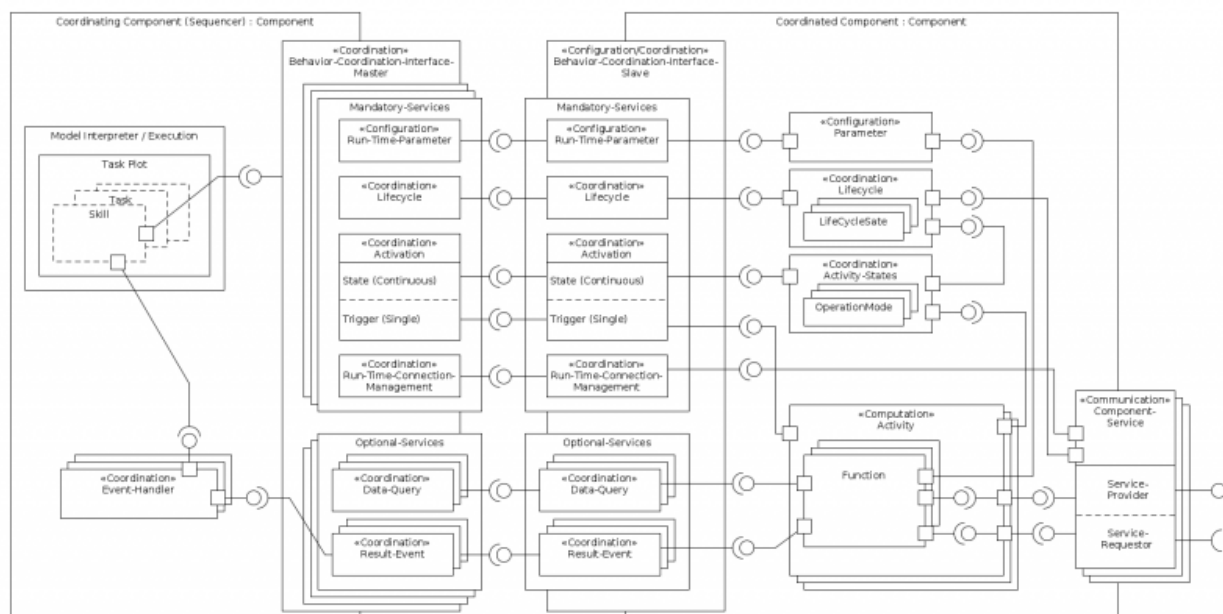
The solution is to define an uniform behavior coordination interface for robotics software components. The interface is two fold: the coordinating component part and the coordinated component part. The coordinating component part is typically realized/implemented by a sequencer component in case of a 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:



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>]

general_principles:architectural_patterns:component-coordination · Last modified: 2019/05/20 10:49
http://www.robmosys.eu/wiki/general_principles:architectural_patterns:component-coordination

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 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

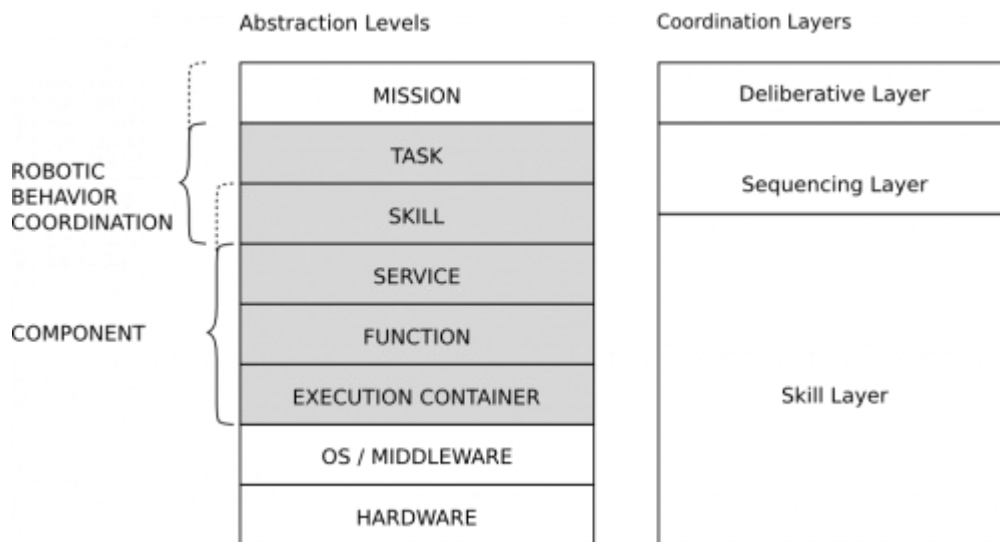
Service robots act in unstructured and open-ended environments that require flexibility and adaptability in execution for the robotic behavior. The basic robot functionality is realized by software components. Software components are typically general software building blocks that are independent of 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 [\[http://www.servicerobotik-ulm.de/drupal/?q=node/86\]](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 [\[http://www.servicerobotik-ulm.de/drupal/?q=node/86\]](http://www.servicerobotik-ulm.de/drupal/?q=node/86)

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

Discussion

Tasks describe via which steps (*what*: the ordering of steps) and in which manner (*how*: the kind of execution) to accomplish a particular job. This is done in an abstract manner by just referring to skills (using a domain-specific skill vocabulary). In this way, a task plot can be used with any robot that provides all the required skills. This allows the behavior developer to engineer task plots as a solution for a particular job to be done by a robot independently of the very concrete robot finally used in the deployment.

A *skill* provides access to functionalities realized by components. It is the bridge between generic descriptions of capabilities (independent of a particular implementation) and behavior interfaces (configuration: resources, parameters, wiring; coordination: modes, lifecycle, events) of a (set of) component(s) to achieve that capability (specific to the used components). A skill lifts the level of abstraction from a component-specific level to a generic level. Thus, different implementations for the same capabilities become replaceable as they are accessed in a uniform way.

Tasks and skills are domain-specific as they refer to domain-specific vocabularies. Nonetheless, a software component can be used in skill sets of different domains. Of course, one can also define a domain that holds generic tasks and skills that can then be used from within different domains.

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. [\[https://mediatum.ub.tum.de/?id=1362587\]](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: 2020/12/04 15:12
http://www.robmosys.eu/wiki/general_principles:architectural_patterns:robotic_behavior

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>]

general_principles:architectural_patterns:managing_transitive_system_states · Last modified: 2019/05/20 10:49
http://www.robmosys.eu/wiki/general_principles:architectural_patterns:managing_transitive_system_states

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>]

general_principles:architectural_patterns:service_definitions · Last modified: 2019/05/20 10:49
http://www.robmosys.eu/wiki/general_principles:architectural_patterns:service_definitions

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:

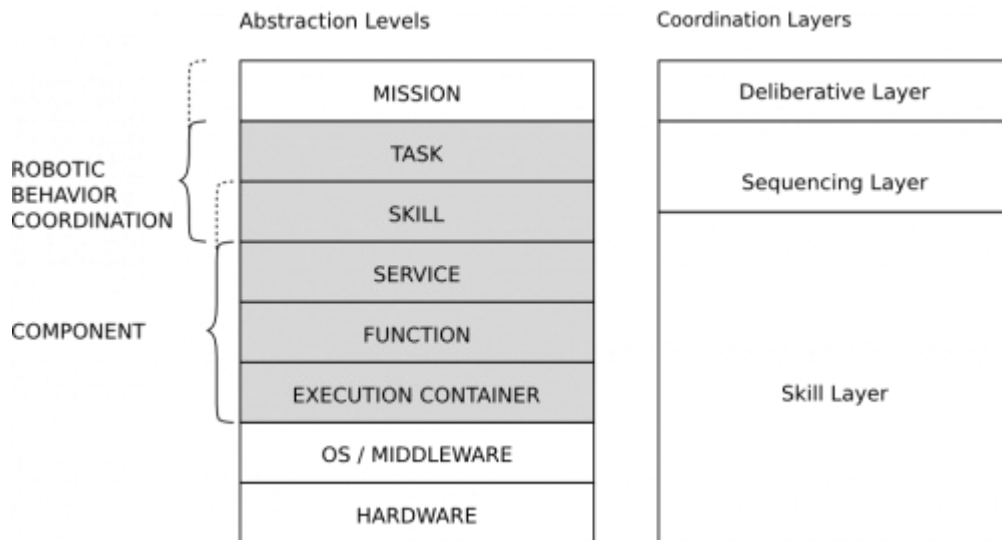
- Skill (level) requires a coordination interface for each component
- Service (level) specifies interaction points to other components (i.e. the communication concern)
- Function (level) realizes the component's internal functionality
- Execution Container (level) links functionality with the execution 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 skills¹⁾.

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](https://doi.org/10.1080/095281397147103)
[<http://dx.doi.org/10.1080/095281397147103>].

general_principles:architectural_patterns:components · Last modified: 2019/05/20 10:49
http://www.robmosys.eu/wiki/general_principles:architectural_patterns:components

Separation of Levels and Separation of Concerns

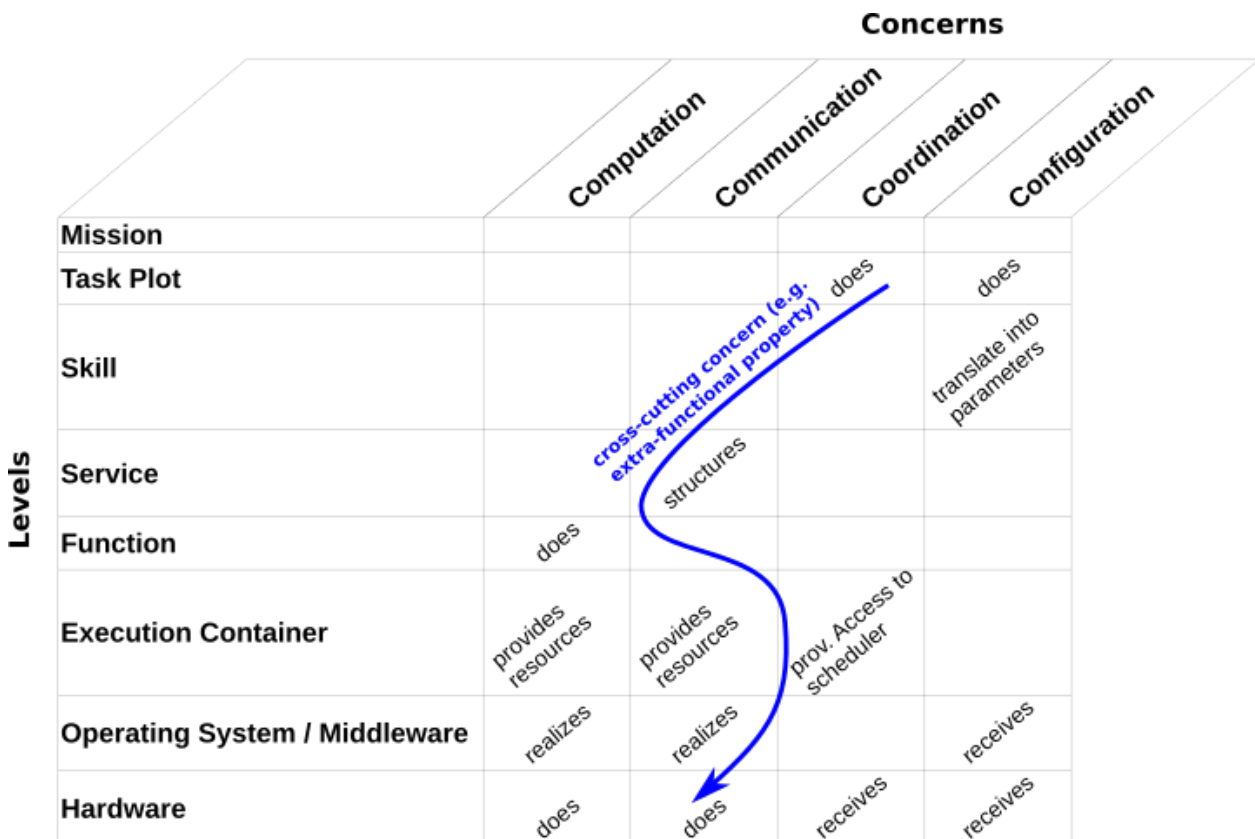
The figure below illustrates the separation of levels and the separation of concerns. Please also refer to the [RobMoSys Glossary](#) for descriptions of used terms. The levels indicate abstractions in a robotics system.

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.

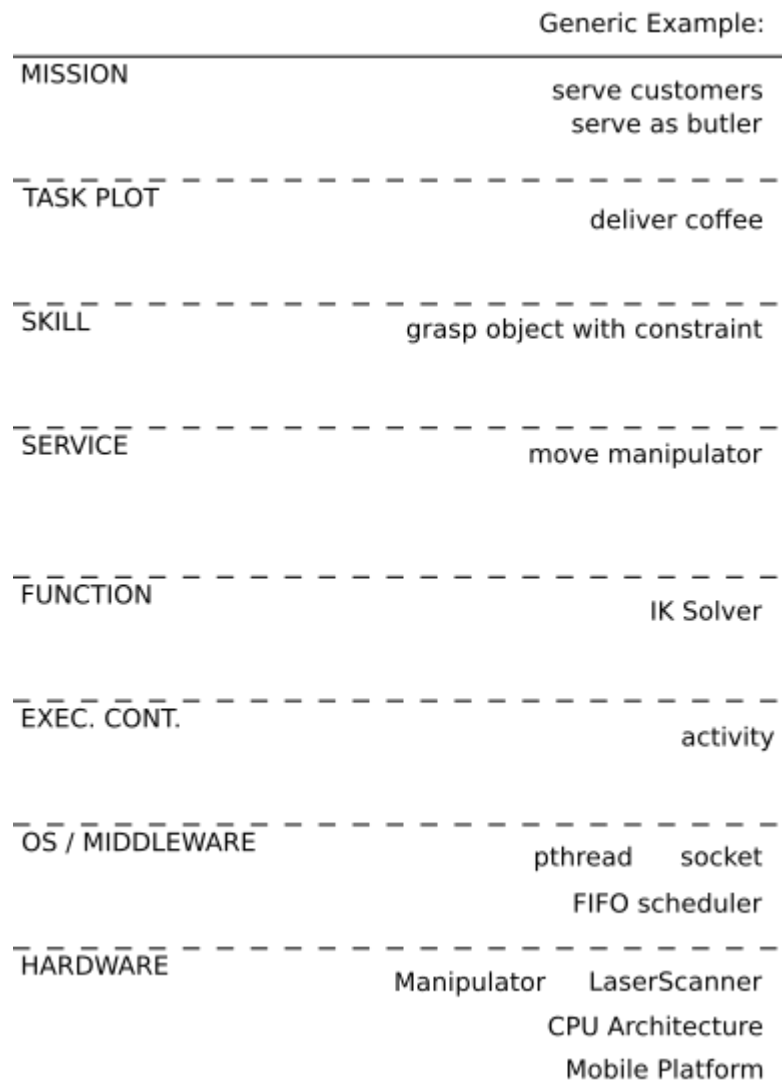
- By definition, a level can not be defined on its own, since its semantics is the relationships between the items at this “level” and those at the other levels. This exercise to get these relationships well-defined is a tough one, this is of high priority though, since “level”/“layer” is one of the 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)

A the skill level is an abstraction level that decouples task-level and service-level. The purpose of abstraction is to enable replacement and composition of components (components providing the same skill) and decoupling (e.g. separation of roles: component developer and behavior developer).

Skills provide access to the functionalities realized within components and make them accessible to the task level. Skills coordinate software components through RobMoSys Software Component Coordination interface. With skill definitions on Tier 2, skills enable the task modeling independent of the underlying software component architecture. Skill implementations are bundled with software components and are provided by the component supplier role.

A skill 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.

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](#)
- [Robotic Behavior in RobMoSys using Behavior Trees and SmartSoft](#)
- [Skills for Robotic Behavior](#)

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. pthread, 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.2010.5733835<https://doi.org/10.1109/IEEESTD.2010.5733835>

[<https://doi.org/10.1109/IEEESTD.2010.5733835>]

general_principles:separation_of_levels_and_separation_of_concerns · Last modified: 2019/05/20 10:47
http://www.robmosys.eu/wiki/general_principles:separation_of_levels_and_separation_of_concerns

Pilot Skeletons: 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.

pilots:start · Last modified: 2019/05/20 10:47
<http://www.robmosys.eu/wiki/pilots:start>

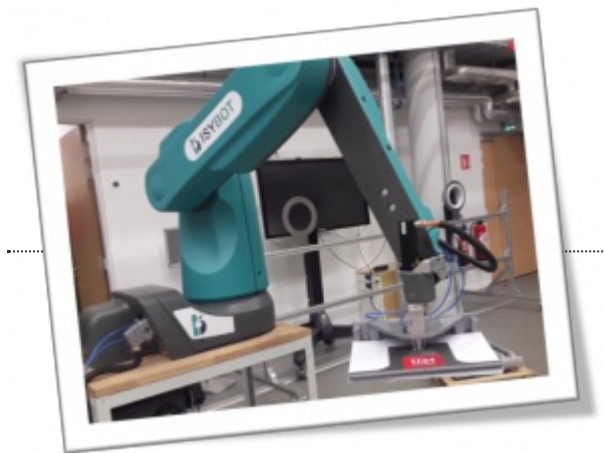
Human Robot Collaboration for Assembly Pilot

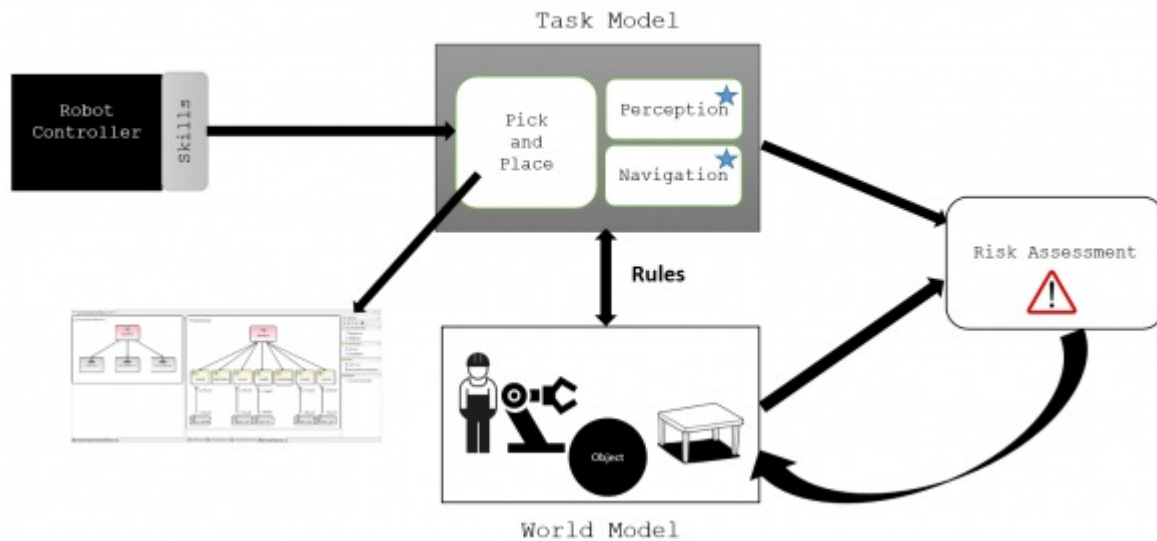
In the context of human-robot collaboration, the operator interacts with the robot with no fences and influences the task execution. Thus, taking into account the context and more generally the environment for task definition is both mandatory and challenging at the modeling level. Human-robot collaboration raises also important safety requirements related to the robot, the tool, the task and the environment. Therefore, safety and more particularly risk assessment is a major feature that this pilot aims to realize for reducing risk occurrence. 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.

The pilot demonstrates task and environment definition for a human-robot collaboration use case: Pick and Place through RobMoSys tools. The interaction between the robot and the operator is direct (with no fences) for carrying a heavy object from a given starting position to a target one.

Pilot Skeleton Resources and Software

The pilot is intended for the use with ISybot collaborative robot but can be extended to any collaborative robot with the same capabilities thanks to the generic task specification implemented in robmosys. This pilot focuses on task specification and safety functions.



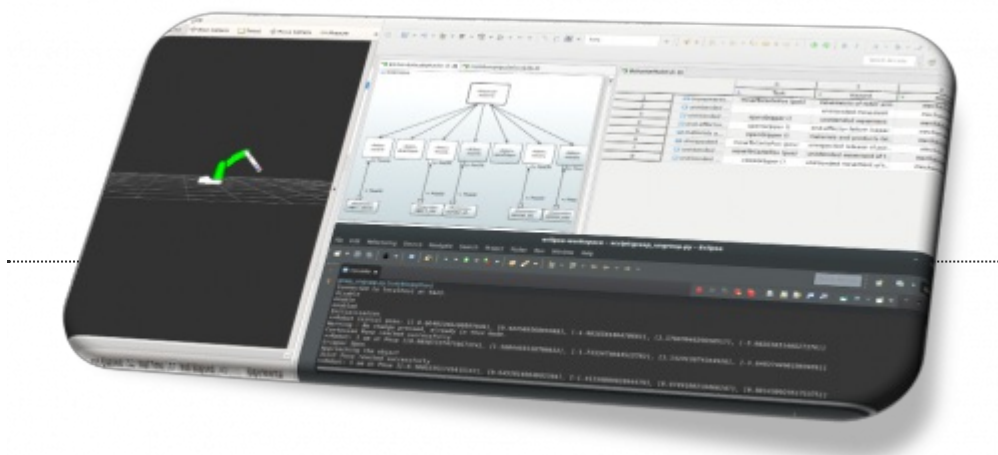


The gripper attached to the robot is used for grasping paper. We can replace this gripper with another one to manipulate other categories of objects. The operator interacts with the robot before task execution for teaching the task to the robot or during task execution for co-manipulation. This pilot will provide a basis that can be enriched with other functionalities in the second open call. A robot arm can be placed on a mobile base for example. A camera or other sensors can be added for object identification.

The development and deployment of the application can be done easily through a virtual machine [https://syncandshare.lrz.de/dl/fi8NYUJrHsMVuXyAy22MrKm7/CEAPilot_VirtualMachine.zip] that contains all the necessary tools, simulators, software, examples and documentation:

- Eclipse and Papyrus for Robotics
- Task Model for Pick and Place
- ROS Lunar
- ROS stack for Isybot
- A test script for visualizing the robot movements
- A specification document for describing the use case and the modeling steps
- A preliminary version of a risk assessment

In order to run the demo, download the virtual machine »here [https://syncandshare.lrz.de/dl/fi8NYUJrHsMVuXyAy22MrKm7/CEAPilot_VirtualMachine.zip]< and follow the steps described in the README file.



The examples were built to show several aspects of RobMoSys related to task specification, world models and safety functionality.

pilots:hr-collaboration · Last modified: 2019/09/12 09:49
<http://www.robmosys.eu/wiki/pilots:hr-collaboration>

Modular Education Robot Pilot

This Pilot aims at validating the RobMoSys methodology by applying it in an educational scenario using the e.DO Robot from Comau.. 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.



The pilot case will be based on the open architecture of e.DO platform (ROS compliant):

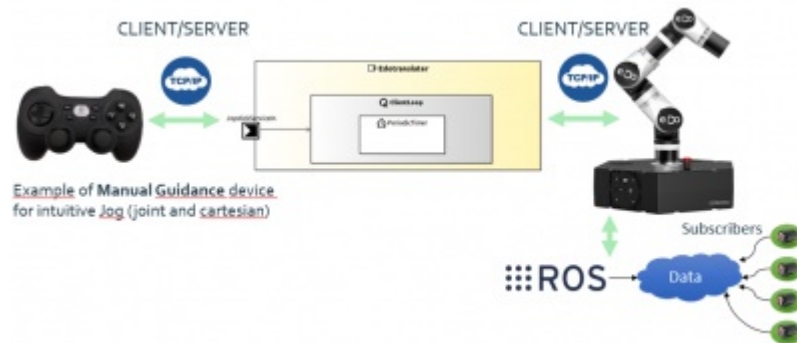
the main objectives of the Pilot are reported below and foresee the development of customized software functionalities on different levels pick and place application with integrated gripper:

- Basic coding (scratch programming) → task composition
- Emulation of industrial lines to speed up the integration.
- Implementation and test of advanced control algorithm
- Robotics domain and composition
- Motion
- Perception
- World model
- Task Composition
- Composition of algorithms

The benefits that will be demonstrates in the pilot case will be:

- Replacement of components
- Composition of components
- Reduction of development time
- Predictable safety
- Ease of use

A first PoC has been developed through a TCP/IP communication by using the SmartMDSD development environment. A standard components as a joystick has been integrated for programming the e.DO robot using the Smartsoft platform. The same approach can be used for different type of components (inertial sensor, visual device) for easy and intuitive programming and use of the robot. The virtual machine for this scenario is available.



Download

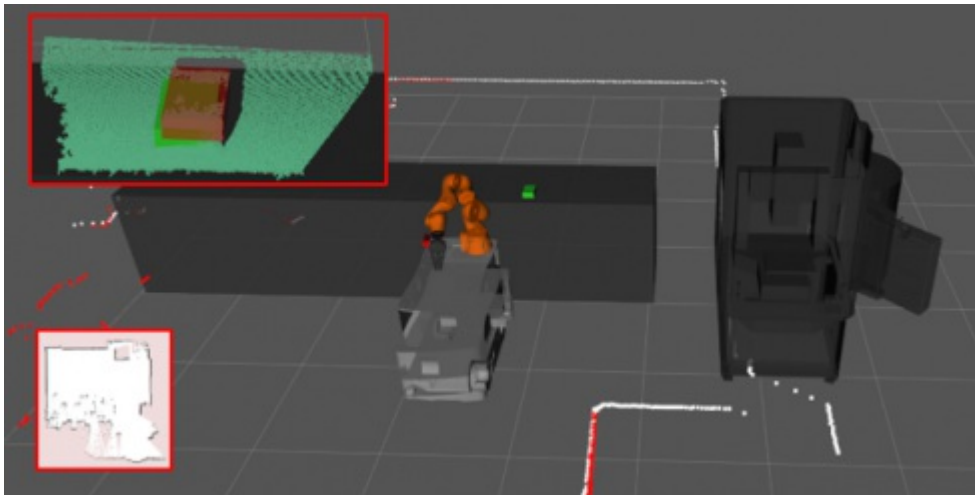
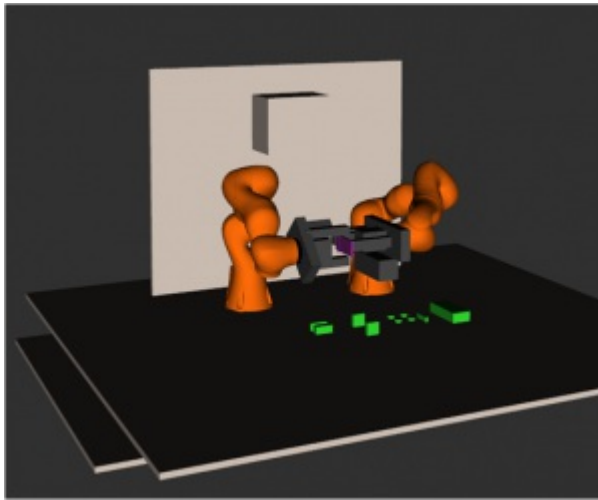
We provide the Pilot as Virtual Machine Download

[https://syncandshare.lrz.de/dl/fi49vXeU6WuyKma8HiGLZXoD/COMAU_Pilot_VM.zip].

pilots:education · Last modified: 2019/10/14 10:18
<http://www.robmosys.eu/wiki/pilots:education>

Flexible Assembly Cell Pilot

This pilot is about highly-flexible industrial production. Advanced automation devices, like autonomous robotic systems, are no longer based on simple I/O signal communication but provide a full-fledged, high-level application programming interface to access the device's features and functionality. This pilot showcases the development and programming of advanced production systems that can perform a large range of different tasks with high demands on performance and adaptability.



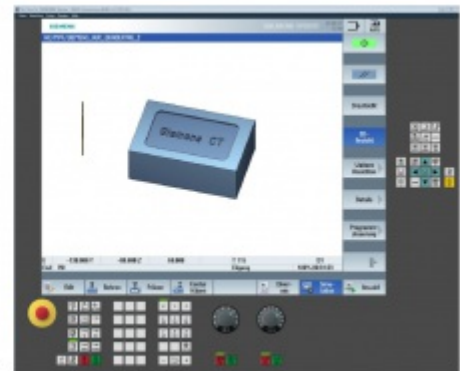
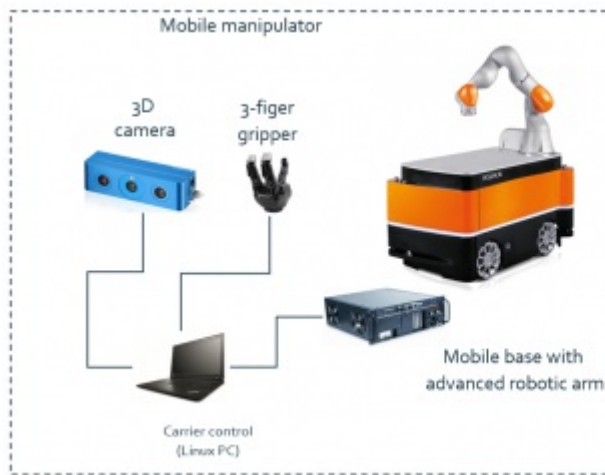
The Pilot's intended target system is an industrial production system with advanced automation equipment. A concrete example would be a mobile or a dual-arm manipulator system equipped with multiple 2D and 3D cameras.

This pilot focuses on:

- 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/software component the system builder should be able to replace a hardware/software component and check whether the system can still perform all the required tasks.

Pilot Skeleton Resources and Software

Base functionality such as object detection and manipulation, as well as navigation and an exemplary machine-tending application, are available for contributors to extend, modify or build upon. The (virtual) hardware setup consists of a mobile manipulator including a mobile base with an advanced robotic arm equipped with a 3D camera for perception and a gripper for object manipulation. A CNC milling machine is also part of the setup. The intended application of this mobile manipulator is a machine tending task where the mobile manipulator has to (1) fetch a work piece from their storage location (2) feed the machine with the work piece (3) retrieve the processed work piece from the machine (4) and bring it to a possibly different storage location. The initial location of the work piece is only roughly specified. In this setup, the 3D camera is used for accurately determining its pose. Furthermore, a navigation system is used to move the mobile base from one location to the other using laser and odometry data for localization. To communicate with the milling machine OPC UA is used basic interaction with the CNC control unit, for example, for running the milling program.



We provide a VirtualBox Image that allows to use the SmartMDS Toolchain together with our task framework to virtually execute these machine-tending tasks. The system consists largely of mixed-port components that provide access to the system functionality realized in ROS (legacy). This choice has been made to show how the RobMoSys approach can deal with industrial, off the shelf hardware, and related proprietary/legacy software. An executive control component executes the application by orchestrating the other components.

Pilot Examples and Intended Use

In order to run the demo following steps need to be executed:

- Download and extract the VirtualBox hard-drive image from:
<https://syncandshare.lrz.de/dl/fiQYffPGHg4ck4recY2Jbuva/robmosys-siemens.vdi.zip>
[<https://syncandshare.lrz.de/dl/fiQYffPGHg4ck4recY2Jbuva/robmosys-siemens.vdi.zip>]
- Set up a virtual machine using this image that contains a 64bit ubuntu system

- Boot and login with login:“robmosys” pass:“robmosys”
- Start a terminal (ctrl + alt + t)
- Execute: “roslaunch launch_kmr kmr_mock.launch” the rviz visualization should start up
- Run “smartsoft” in a second terminal
- Choose the Package “SystemCNCBehaviourTreeExecutor” and click “Deploy”
- Choose “Yes” in the dialog that pops up to start the system
- Click the return-key to start the scenario

pilots:flexible-assembly · Last modified: 2019/09/06 16:50
<http://www.robmosys.eu/wiki/pilots:flexible-assembly>

Intralogistics Industry 4.0 Robot Fleet Pilot

This pilot is about **goods transport** in a company, such as factory **intra-logistics**. It can be used to **showcase robotics navigation**, e.g. to show the performance of goods delivery and according non-functional requirements. It can be extended to object recognition and manipulation.

The pilot is intended for open call 2 contributors to showcase the ease of system integration via composition of software components to a complete robotics application.

The pilot is physically located at Ulm University of Applied Sciences; Germany and may be used on site or remotely. An excerpt is available in simulation for off-site use.

Pilot Facts Sheet for Open Call 2 [<https://robmosys.eu/open-call-2/>]

Videos of the pilot in action:

- <https://www.youtube.com/watch?v=qRSDxBOUVx0> [<https://www.youtube.com/watch?v=qRSDxBOUVx0>]
- https://www.youtube.com/watch?v=ML_BtZsiPHo [https://www.youtube.com/watch?v=ML_BtZsiPHo]





(a) Picking items



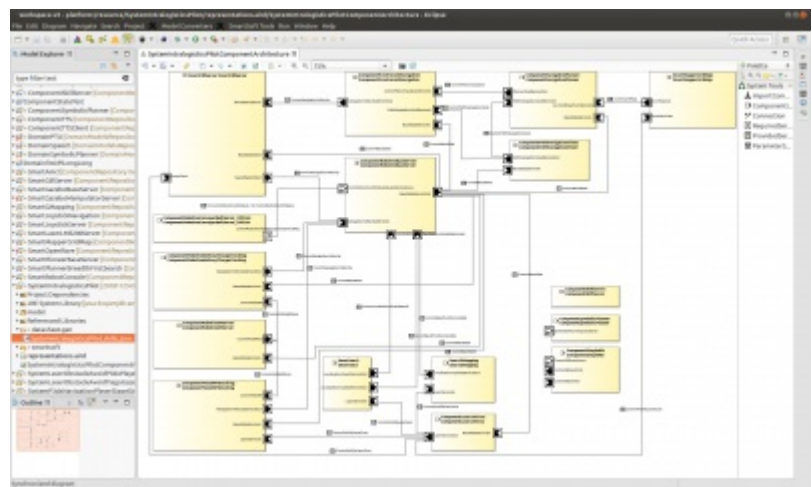
(b) Transport and autonomous delivery of boxes with goods



(c) Order picking into boxes

RobMoSys Pilot Skeleton and Tooling Support

The pilot is fully supported by the SmartMDSD Toolchain, an Integrated Software Development Environment (IDE) for system composition in an robotics software business ecosystem. This ensures full conformance to the RobMoSys methodology when using this pilot.



Pilot Skeleton Resources and Software

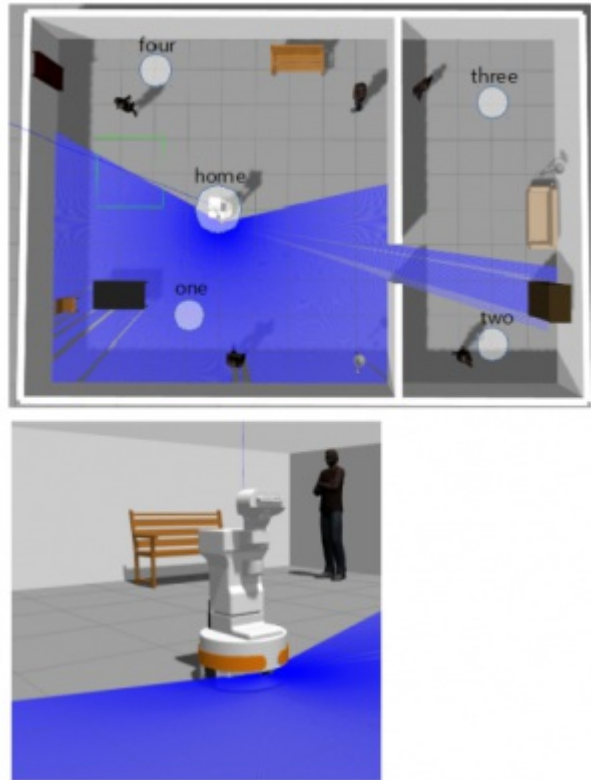
- You can use this pilot with the [SmartMDSD Toolchain](#)
- Software components for this pilot are available: see [RobMoSys Model Directory](#)
 - Robot platforms, mapping, planning, obstacle avoidance, etc. for immediate composition
 - Depending on your use-case, software component templates for manipulation and object recognition are available or you may want to use manipulation on task-level.
- [Documentation and Tutorials](#) available for use with the SmartMDSD Toolchain
- The pilot features the [flexible navigation stack](#).
- Components are coordinated using [Robotics behavior coordination in SmartSoft: SmartTCL](#)
- [Tutorial for running the Gazebo/Tiago/SmartSoft Scenario in simulation](#)

Pilot Examples and Intended Use

The pilot is intended for the use with FESTO Robotino but can be extended to any robot thanks to the Flexible Navigation Stack. It covers navigation via the Flexible Navigation Stack and mobile manipulation using the Mobile Manipulation Stack.

Potential Use-Cases for ITP demonstrations:

- Integrate your own robot (as e.g. shown by the CARVE ITP)
- Interact with the pilot on task-, service- or component-level.
- Work with a single robot or a fleet of robots
- Software components and system composition: e.g. composition of previously developed software components and/or exchange of software components to address new needs.
- Ecosystem collaboration including the different roles that participants can take
- Task level coordination and composition; skills; robotic behavior
- Managing of non-functional properties
- Dependency graphs for composed components to enable predictability for navigation



Usage of this Pilot

This pilot or skeletons of this pilot have been used in the following demonstrations:

- Robotic Behavior in RobMoSys using Behavior Trees and SmartSoft
- Dealing with Metrics on Non-Functional Properties in RobMoSys
- Using the YARP Framework and the R1 robot with RobMoSys

Assistive Mobile Manipulation Pilot

This pilot focuses specifically on robotic health-care adaptation. It showcases the development and programming of assistive mobile robots in unstructured and dynamic environments where the robot has to perform complex tasks combining several capabilities such as mobility, perception, navigation, manipulation and human-robot interaction. Furthermore, these capabilities have to be customised for an individual and for a specific apartment. The key to the success on these changeable environment is to encapsulate the different functionalities at design time, to combine them at deployment time and to leave them interact at run-time.



The system addressed in this Pilot is a TIAGo mobile base manipulator working in a standard apartment.

This pilot demonstrates:

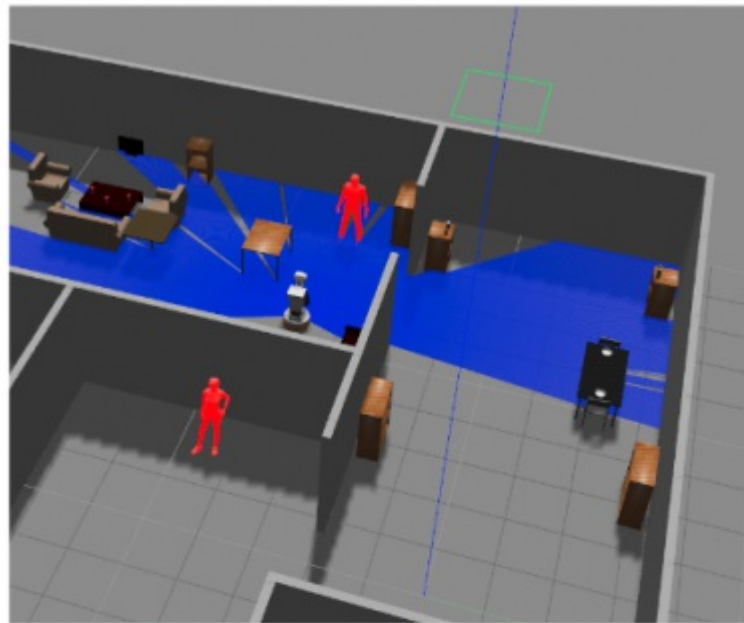
- Replacement of component(s): the System Builder wants to exchange components with the same functionality. The demonstrator prepared consist of an application that detects people using different combinations of software and hardware components. Like for example exchanging the component of the RGBD camera that uses deep learning to detect people with another component that is represented by the thermal camera that doesn't use the deep learning techniques but that is more expensive.
- Composition of components: the System Builder wants to create a new TIAGo robot checking 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.
- Task Coordination for Object Detection: the Behaviour Developer wants to prepare the coordination of the system components to look for standard objects in an apartment.

Pilot Skeleton Resources and Software

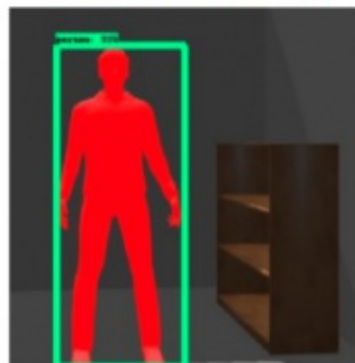
The setup consists of a TIAGo assistive mobile manipulator robot navigating in private apartment to assist the person living in the flat. Base functionality such as object and people detection, arm control, as well as navigation and an exemplary human-robot interaction interfaces, are available for contributors to extend, modify or build upon. The TIAGo robot consists of a wheeled mobile manipulator equipped with odometry, a laser scan, a depth camera, and a thermal sensor. The different capabilities can be tested in simulation and on the real robot. For people detection, two different algorithms, using two different sensors can be swapped or combined:

- RGB-D camera + Deep learning people detection
- Thermal camera + OpenCV people detection

The idea is to choose the right combination to fulfil the requirements.



Machine learning



Thermal camera



The intended application of this mobile manipulator is to assist people in a private apartment, so potential use-cases could be: creating assistive applications for TIAGo (welcome visitors to the apartment, finding objects in the apartment, delivering items to the elderly person, ...); extend the pilot skeleton with components related to specific fields like HRI, object/people recognition, manipulation. The interface with the pilot is possible on different levels (from component to task level).

The pilot resources can be downloaded and executed on any computer via docker container, which includes everything needed to run the TIAGo simulation and the SmartMDSD Toolchain. To speed up the development time, the components are bridges made as ROS interfaces to access the legacy code existing for the TIAGo robot. It also shows how RobMoSys works finely with legacy code that have been previously implemented, even if the RobMoSys benefits will be narrowed by the capacities and the design of the existing code. Nonetheless, some components may be replaced with native RobMoSys components.

The resources access will be granted specifically to the pilot developers, on demand. A ready-to-run container is available, the TIAGo docker (including Linux Ubuntu, SmartMDSD Toolchain, Eclipse IDE, Gazebo simulation, TIAGo ROS packages). All the instructions to build, run the dockers, run the demos and work on the pilot are reported in the readme files included in the resources repositories.

Software available:

- RobMoSys software components to use with SmartMDSD Toolchain
- TIAGo SmartMDSD repositories (navigation, SmartMDSD to ROS bridge ports, System TIAGo deployment)

- TIAGo ROS packages (manipulation, navigation and perception)
- Documentation and TIAGo tutorials [<http://wiki.ros.org/Robots/TIAGo/Tutorials>]

Extra 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.
- [Tutorial for running the Gazebo/Tiago/SmartSoft Scenario in simulation](#)

pilots:assistive-manipulation · Last modified: 2019/09/03 16:02
<http://www.robmosys.eu/wiki/pilots:assistive-manipulation>

RobMoSys Model Directory

A list of domain models, software components and systems for use with RobMoSys Tooling. Please see end of page for a legend.

Tier 2 Domain Models

Name	Description	Purpose
CommBasicObjects [https://github.com/ServiceRobotics-Ulm/DomainModelsRepositories/tree/master/CommBasicObjects]	A collection of very basic service definitions and communication objects for use in almost every robotics system.	Universal
CommNavigationObjects [https://github.com/ServiceRobotics-Ulm/DomainModelsRepositories/tree/master/CommNavigationObjects]	A collection of domain models for wheeled robot navigation .	Navigation
CommRobotinoObjects [https://github.com/ServiceRobotics-Ulm/DomainModelsRepositories/tree/master/CommRobotinoObjects]	A collection of domain models for use with the FESTO Robotino robot.	Mobile-Base
CommLocalizationObjects [https://github.com/ServiceRobotics-Ulm/DomainModelsRepositories/tree/master/CommLocalizationObjects]	A collection of domain models for localization .	Localization
CommManipulationPlannerObjects [https://github.com/ServiceRobotics-Ulm/DomainModelsRepositories/tree/master/CommManipulationPlannerObjects]	A collection of domain models for (mobile) manipulation.	Mobile Manipulation
CommManipulatorObjects [https://github.com/ServiceRobotics-Ulm/DomainModelsRepositories/tree/master/CommManipulatorObjects]	A collection of domain models for manipulators.	Manipulation
CommObjectRecognitionObjects [https://github.com/ServiceRobotics-Ulm/DomainModelsRepositories/tree/master/CommObjectRecognitionObjects]	A collection of domain models for object recognition.	Object Recognition
CommTrackingObjects [https://github.com/ServiceRobotics-Ulm/DomainModelsRepositories/tree/master/CommTrackingObjects]	A collection of	Object

Name	Description	Purpose
Ulm/DomainModelsRepositories/tree/master/CommTrackingObjects]	domain models for tracking objects.	
DomainForklift [https://github.com/ServiceRobotics-Ulm/DomainModelsRepositories/tree/master/DomainForklift]	A collection of domain models for accessing forklift manipulators.	Actuator-Access
DomainPTU [https://github.com/ServiceRobotics-Ulm/DomainModelsRepositories/tree/master/DomainPTU]	A collection of domain models for accessing pan-tilt-devices.	Actuator-Access
DomainSpeech [https://github.com/ServiceRobotics-Ulm/DomainModelsRepositories/tree/master/DomainSpeech]	A collection of domain models for human-machine-interaction with speech.	Speech-To-Text
DomainSymbolicPlanner [https://github.com/ServiceRobotics-Ulm/DomainModelsRepositories/tree/master/DomainSymbolicPlanner]	A collection of domain models for symbolic planning.	Planner
DomainVision [https://github.com/ServiceRobotics-Ulm/DomainModelsRepositories/tree/master/DomainVision]	A collection of domain models related to vision.	Actuator-Access

Tier 3 Component Models

Name	Description
SmartCdServer [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartCdServer]	Implements the Curvature Distance Lookup (CDL) algorithm for fast lo obstacle avoidance. It considers the dynamics and kinematics of the robot as well as its polygonal shape.
ComponentLaserObstacleAvoid [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/ComponentLaserObstacleAvoid]	The SmartLaserObstacleAvoid component implements a simple reactive obstacle avoidance.
ComponentPlayerStageSimulator [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/ComponentPlayerStageSimulator]	The SmartPlayerStageSimulator simulates a robot in a 2D bitmapped environment using Player/Stage. It offers several services for controlling the robot, such as sending navigation commands, providing access to the robot's odometry and laser scans.
ComponentSymbolicPlanner [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/ComponentSymbolicPlanner]	Provides a symbolic planner service. Works with ff, metric-ff (suggested) and lama.

Name	Description
ComponentTTS [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/ComponentTTS]	SmartTTS is a component for text to speech (tts) synthesis. SmartTTS pipes speech output messages to stdin of an arbitrary executable. It is thus a simple wrapper for all tts applications that accept text via stdin, e.g. mbrola, festival or /bin/cat for debugging.
SmartAmcl [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartAmcl]	SmartAmcl implements the Adaptive Monte-Carlo Localization (Amcl) algorithm. Localization is based on a particle filter and a pre-captured grid map of the environment.
SmartCdlServer [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartCdlServer]	The SmartCdlServer is based on the Curvature Distance Lookup (CDL) algorithm for fast local obstacle avoidance.
SmartGMapping [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartGMapping]	SmartGMapping implements GMapping for simultaneous localization and mapping (SLAM).
SmartGazeboBaseServer [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartGazeboBaseServer]	The SmartGazeboBaseServer can be used to command a robot in a 3D environment using the Gazebo simulator.
SmartJoystickNavigation [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartJoystickNavigation]	The SmartJoystickNavigation component takes joystick input commands (CommJoystick) and translates them to v/omega navigation commands (CommNavigationVelocity).
SmartJoystickServer [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartJoystickServer]	The SmartJoystickServer provides access to input commands from a joystick via PushNewest communication pattern. The input commands are represented by x/y-value (as available) and an identifier for the button pressed.
SmartLaserLMS200Server [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartLaserLMS200Server]	The SmartLaserLMS200Server makes laser scans from SICK LMS 200 and PLS longer rangefinders available. Scans can be requested by push newest or query communication.
SmartMapperGridMap [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartMapperGridMap]	The SmartMapperGridMap provides mapping services based on occupancy grid maps. Laser scans are taken for building a current and a long-term map.
SmartPioneerBaseServer [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartPioneerBaseServer]	The SmartPioneerBaseServer makes P2OS-based robot platforms available. It handles all the communication with the hardware. It offers several services.

Name	Description
	for controlling the robot, such as sending navigation commands to the base and providing access to the robot's odometry.
SmartPlannerBreadthFirstSearch [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartPlannerBreadthFirstSearch]	The SmartPlannerBreadthFirstSearch provides path planning services based on grid maps. It uses a grid map from a map building component (e.g. SmartMapperGridMap) and sends an intermediate waypoint as well as the goalpoint to the motion execution (e.g. SmartCdlServer).
SmartRobotConsole [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartRobotConsole]	A simple coordination component using a console menu to configure software component for several simulation scenarios
Yarp_BaseStateService [https://github.com/CARVE-ROBMOSYS/Yarp-SmartSoft-Integration/tree/master/Bridges/BridgeV3_FromYarp_BaseStateService]	A mixed-port component that bridges between YARP and SmartSoft: This one is focussed on the BaseStateService for use with the R1 robot .
Yarp_GridMap2D Component [https://github.com/CARVE-ROBMOSYS/Yarp-SmartSoft-Integration/tree/master/Bridges/BridgeV3_FromYarp_GridMap2D]	A mixed-port component that bridges between YARP and SmartSoft: This one is focussed on the GridMap Service for use with the R1 robot .
Yarp_LaserService Component [https://github.com/CARVE-ROBMOSYS/Yarp-SmartSoft-Integration/tree/master/Bridges/BridgeV3_FromYarp_LaserService]	A mixed-port component that bridges between YARP and SmartSoft: This one is focussed on the Laser Service for use with the R1 robot .
Yarp_CommNavigationVelocity Component [https://github.com/CARVE-ROBMOSYS/Yarp-SmartSoft-Integration/tree/master/Bridges/BridgeV3_ToYarp_CommNavigationVelocity]	A mixed-port component that bridges between YARP and SmartSoft: This one is focussed on the NavigationVelocity Service for use with the R1 robot .
ComponentRobotinoBaseServer [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/ComponentRobotinoBaseServer]	The Robotino Base Server provides access to the robotino robot. It handles the communication with the hardware or the simulator. It offers several services for controlling the robot, such as sending navigation commands to the base and providing access to the robot's odometry. Position updates can be sent to the component to overcome odometry failures.
ComponentLaserLMS1xx [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/ComponentLaserLMS1xx]	The ComponentLaserLMS1xx makes laser scans from SICK LMS 1xx sensors available.
SmartKinectV2Server [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartKinectV2Server]	The SmartKinectV2Server captures

Name	Description
Ulm/ComponentRepository/tree/master/SmartKinectV2Server	RGB, depth and range images from the Microsoft Kinect. Undistorted images can be requested by push or query communication.
SmartObjectRecognition [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartObjectRecognition]	A component for object recognition
ComponentOpenRave [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/ComponentOpenRave]	This component is based on the OpenRAVE framework. It allows to plan a trajectory to a given point with the specified manipulator.
ComponentSkillInterface [https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/ComponentSkillInterface]	The ComponentSkillInterface provides an interface to call and execute skills as explained in Skills for Robotic Behavior.
SmartRobotinoLaserServer [http://wiki.openrobotino.org/index.php?title=Smartsoft]	The SmartRobotinoLaserServer provides laser scans from the robotino SIM simulator or other scanners operated by robotino daemons. Scans can be requested by push or query communication.
SmartRobotinoIRServer [http://wiki.openrobotino.org/index.php?title=Smartsoft]	The SmartRobotinoIRServer provides an interface to request an IR scan from the robotino platforms (and simulator).
SmartRobotinoRPCBridge [http://wiki.openrobotino.org/index.php?title=Smartsoft]	The SmartRobotinoRPCBridge component is the interface component between the robotino rpcd and the SmartSoft components. It is used to transfer data between the two system parts, e.g. the map as it is recorded by the SLAM component.
SmartRobotinoImageServer [http://wiki.openrobotino.org/index.php?title=Smartsoft]	The SmartRobotinoImageServer provides images captured by a robotino daemon or the simulator.
SmartFestoFleetCom [http://wiki.openrobotino.org/index.php?title=Smartsoft]	The SmartFestoFleetCom is the interface component to the FESTO Fleet-Manager (MPS)
SmartFestoGripperServer [http://wiki.openrobotino.org/index.php?title=Smartsoft]	The SmartFestoGripperServer provides access to the FESTO gripper for Robotino3.
SmartFestoMPSDocking [http://wiki.openrobotino.org/index.php?title=Smartsoft]	The SmartFestoMPSDocking performs the docking/undocking of a Robotino equipped with a conveyor belt, to a FESTO MPS station.
SmartMarkerTracker [http://wiki.openrobotino.org/index.php?title=Smartsoft]	The SmartMarkerTracker is capable of detecting visual markers and docking to MPS stations equipped with visual markers.

Name	Description
SmartNavigationPlanner [http://wiki.openrobotino.org/index.php?title=Smartsoft]	The SmartNavigationPlanner uses ompl to perform planning for navigation. The component is used in the context of corridor based fleet navigation.
SmartPurePursuitNavigation [http://wiki.openrobotino.org/index.php?title=Smartsoft]	The SmartPurePursuitNavigation realizes a pure-pursuit navigation, calculating velocities to follow a trajectory.
SmartRobotinoBatteryChargerDocking [http://wiki.openrobotino.org/index.php?title=Smartsoft]	The SmartRobotinoBatteryChargerDock performs the docking/undocking of Robotino3 to a battery charger station.
SmartRobotinoConveyerBeltServer [http://wiki.openrobotino.org/index.php?title=Smartsoft]	The SmartRobotinoConveyerBeltServer provides access the robotino3 conveyer belt, used to transport KL (small load carrier).
SmartRobotinoMasterRPCBridge [http://wiki.openrobotino.org/index.php?title=Smartsoft]	The component is the interface component between the the robotino rpcd and the SmartSoft components is used to transfer data between the system parts, e.g. the map as it is recorded by the SLAM component. This component is among the set of fleet coordination components.
SmartPathNavigationServer [http://wiki.openrobotino.org/index.php?title=Smartsoft]	The SmartPathNavigationServer is a coordinating component for corridor based local coordinated navigation with multiple robots in a fleet.
SmartPathNavigationClient [http://wiki.openrobotino.org/index.php?title=Smartsoft]	The SmartPathNavigationClient is client component for corridor based fleet navigation.
Sequencer	The Sequencer/SmartLispServer component is responsible for interpreting and executing the behavior models which are written in SmartT (Smart Task Coordination language). To execute the tasks, Sequencer orchestrates all the other components.
SmartSimpleKB [https://sourceforge.net/p/smartsoft-ace/code/HEAD/tree/trunk/src/components/SmartSimpleKB/]	The SmartSimpleKB component represents the knowledge base of the robot system. System wide runtime information is saved in the knowledge base. It contains the current state of robot as well as application related information like the location of shelves, properties of objects and grasping strategies.

Name	Description
SmartBoxDetection [http://zafh-intralogistik.de/arbeitspakete/arbeitspaket-2/]	The SmartBoxDetection component recognizes individual packaging boxes for partial commissioning processes. The required boxes are searched and recognized with the help of box dimensions and the defined storage area in the shelf. Subsequently, position and orientation of the recognized boxes are provided through the service.
SmartRackDetection [http://zafh-intralogistik.de/arbeitspakete/arbeitspaket-2/]	The SmartRackDetection component recognizes the position and orientation of a shelf in which objects are arranged in an organized manner (A-Frame). Based on the pose of the shelf, the component calculates the position of the individual objects and provides this information through the service.
SmartRealSensePersonTracker [http://zafh-intralogistik.de/arbeitspakete/arbeitspaket-2/]	The SmartRealSensePersonTracker component runs a person recognition algorithm on a RGBD data stream. The component can be configured to recognize one specific person in the image and calculate its position relative to the robot. This position can be used by the robot to follow this person.
SmartKinectV2Server [http://zafh-intralogistik.de/arbeitspakete/arbeitspaket-2/]	The SmartKinectV2Server component abstracts the accesses to color and depth data that was recorded by a Kinect (Xbox one) camera. For recognizing objects the component provides RGBD streams as well as individual color or depth images.
SmartRealSenseV2Server [http://zafh-intralogistik.de/arbeitspakete/arbeitspaket-2/]	The SmartRealSenseV2Server component abstracts the accesses to color and depth data that was recorded by a RealSense (D435) camera. For recognizing objects the component provides RGBD streams as well as individual color or depth images.
SmartURServer	SmartURServer component abstracts the accesses to a UR5 robot arm by encapsulating the communication with the hardware and making it easily accessible through services.
SmartSchunkGripperServer	The SmartSchunkGripperServer component abstracts the accesses to gripper from Schunk by encapsulating the communication with the hardware.

Name	Description and making it easily accessible through services.
-------------	--

Tier 3 Systems

Name	Description	Pu
SystemTiagoNavigation [https://github.com/ServiceRobotics-Ulm/SystemRepository/tree/master/SystemTiagoNavigation]	A pilot skeleton that covers the navigation aspect of the Intralogistics Industry 4.0 Robot Fleet Pilot and Assistive Mobile Manipulation Pilot. This system covers the TIAGo Robot in simulation/Gazebo.	Na
SystemP3dxNavigationRealWorld [https://github.com/ServiceRobotics-Ulm/SystemRepository/tree/master/SystemP3dxNavigationRealWorld]	A pilot skeleton that covers the navigation aspect of the Intralogistics Industry 4.0 Robot Fleet Pilot and Assistive Mobile Manipulation Pilot. This system covers the Pioneer P3dx Robot in a real world setting.	Na
SystemP3dxNavigationPlayerStageSimulator [https://github.com/ServiceRobotics-Ulm/SystemRepository/tree/master/SystemP3dxNavigationPlayerStageSimulator]	-A pilot skeleton that covers the navigation aspect of the Intralogistics Industry 4.0 Robot Fleet Pilot and Assistive Mobile Manipulation Pilot. This system covers the Pioneer P3dx Robot in simulation with Player/Stage	Na
SystemLaserObstacleAvoidP3dxPlayerStageSimulator [https://github.com/ServiceRobotics-Ulm/SystemRepository/tree/master/SystemLaserObstacleAvoidP3dxPlayerStageSimulator]	A system used in the tutorials. It uses a primitive obstacle avoidance component. This system is for use with the Pioneer P3DX robot.	Ex

Name	Description	Ex
SystemLaserObstacleAvoidTiagoGazeboSimulator [https://github.com/ServiceRobotics-Ulm/SystemRepository/tree/master/SystemLaserObstacleAvoidTiagoGazeboSimulator]	A system used in the tutorials. It uses a primitive obstacle avoidance component. This system is for use with the TIAGo robot in the gazebo simulator.	
SystemTTS [https://github.com/ServiceRobotics-Ulm/SystemRepository/tree/master/SystemTTS]	An example that illustrates the use of text to speech components.	Sp To

Explanation/Legend

Status Descriptions

Ready	This model can be used for immediate composition with other models
InProgress	This model is currently being worked on. This can be ongoing work such as implementation or migration from earlier tooling versions (as e.g. with the SmartMDSD Toolchain v2)
Planned	This model is scheduled and it will soon be worked on and going to be available.

Vendor Index

Short	Vendor Name	Website
HSU	Ulm University of Applied Sciences, Service Robotics Research Center	http://www.servicerobotik-ulm.de [http://www.servicerobotik-ulm.de]
CARVE	RobMoSys Integrated Technical Project “CARVE”	https://robmosys.eu/carve/ [https://robmosys.eu/carve/]
REC	Robotics Equipment Corporation GmbH	http://servicerobotics.eu [http://servicerobotics.eu]
ZAFH Intralogistik	ZAFH Intralogistik - Kollaborative Systeme zur Flexibilisierung der Intralogistik	http://zafh-intralogistik.de [http://zafh-intralogistik.de]

Tooling

SmartMDSD Toolchain v3	Models tagged with “SmartMDSD Toolchain v3” are fully RobMoSys conformant and available for immediate composition.
SmartMDSD Toolchain v2 [https://wiki.servicerobotik-ulm.de/how-tos:v2-or-v3]	Models tagged with “SmartMDSD Toolchain v2” are sufficiently conformant to RobMoSys and available for immediate composition with the SmartMDSD Toolchain v2. These components are under migration to full RobMoSys conformance and use in v3.

This is an old revision of the document!

Community Corner

In this section, we feature early adoptors of RobMoSys methodology, composition structures, or tooling; we present community-related information.



- Get in touch: [Discourse Forum \[https://discourse.robmosys.eu/\]](https://discourse.robmosys.eu/) and [Events \[https://robmosys.eu/events/\]](https://robmosys.eu/events/)
- [Integrated Technical Projects \(ITPs\) of Open Call I \[http://robmosys.eu/itp\]](http://robmosys.eu/itp)
- Demonstrations and intermediate results:
 - [Guaranteed Stability of Networked Control Systems \(EG-IPC ITP\)](#)
 - [Safety Assessment of Robotics Systems Using Fault Injection in RobMoSys \(eITUS ITP\)](#)
 - [Robotic Behavior in RobMoSys using Behavior Trees and the SmartMDSD Toolchain \(MOOD2BE ITP\)](#)
 - [Dealing with Metrics on Non-Functional Properties in RobMoSys \(RoQME ITP\)](#)
 - [Using the YARP Framework and the R1 robot with RobMoSys \(CARVE ITP\)](#)
 - [Benchmarking in the RobMoSys Ecosystem \(Plug&Bench ITP\)](#)
- [RobMoSys Tools, Assets and their Conformance](#)
- [EU Digital Industrial Platform for Robotics](#)

community:start · Last modified: 2020/03/02 09:51
<http://www.robmosys.eu/wiki/community:start>

Safety Assessment of Robotics Systems Using Fault Injection in RobMoSys

This demonstrator explains how to use the safety related functionalities developed as part of the eITUS Project via a safety analysis use case scenario. A video is also presented, which goes through a demonstration of showing how to perform safety analysis in the context of RobMoSys by using and extending the Papyrus4Robotics toolchain and Gazebo.

eITUS Demonstrator in the context of RobMoSys User Stories

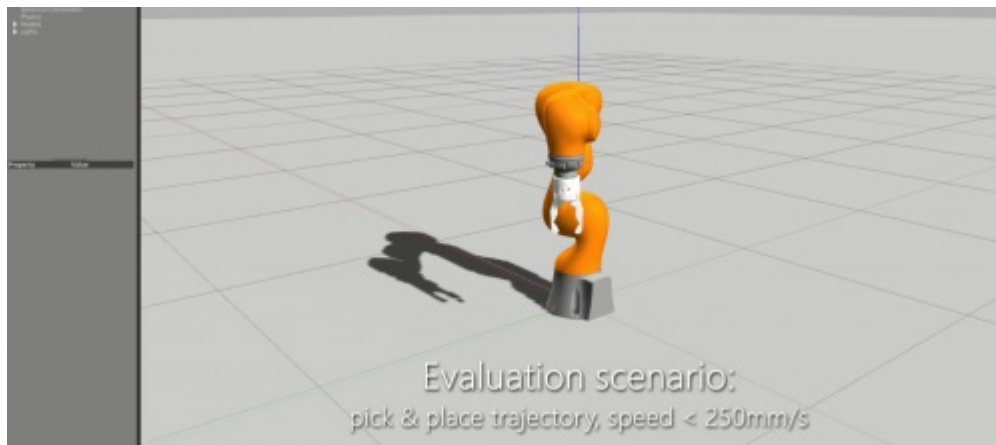


Figure 1. eITUS Evaluation Scenario

eITUS [<https://robmosys.eu/e-itus/>] stands for Experimental Infrastructure Towards Ubiquitously Safe Robotics Systems using RobMoSys. Nowadays, safety is becoming a crucial property of robotic systems. ISO 12100, ISO 13849 and IEC 62061 are some of the most accepted safety standards in robotics, covering aspects such as functional safety. Functional safety is the aspect of safety that aims to avoid unacceptable risks. The system should be designed to properly handle likely human errors, hardware failures and operational/environmental stress.

The safety analysis and validation steps are fundamental aspects to perform the safety assessment. Some of the commonly used risk assessment methods are Preliminary Hazard Analysis, Hazard Operability Analysis, Failure Modes and Effects Analysis and Fault Tree Analysis. Furthermore, fault injection simulations complete these analyses by finding unexpected hazards (fault forecasting) and verifying the implemented safety mechanisms. Figure 2 illustrates how safety analysis is related to RobMoSys' views.

Safety Analysis with RobMoSys

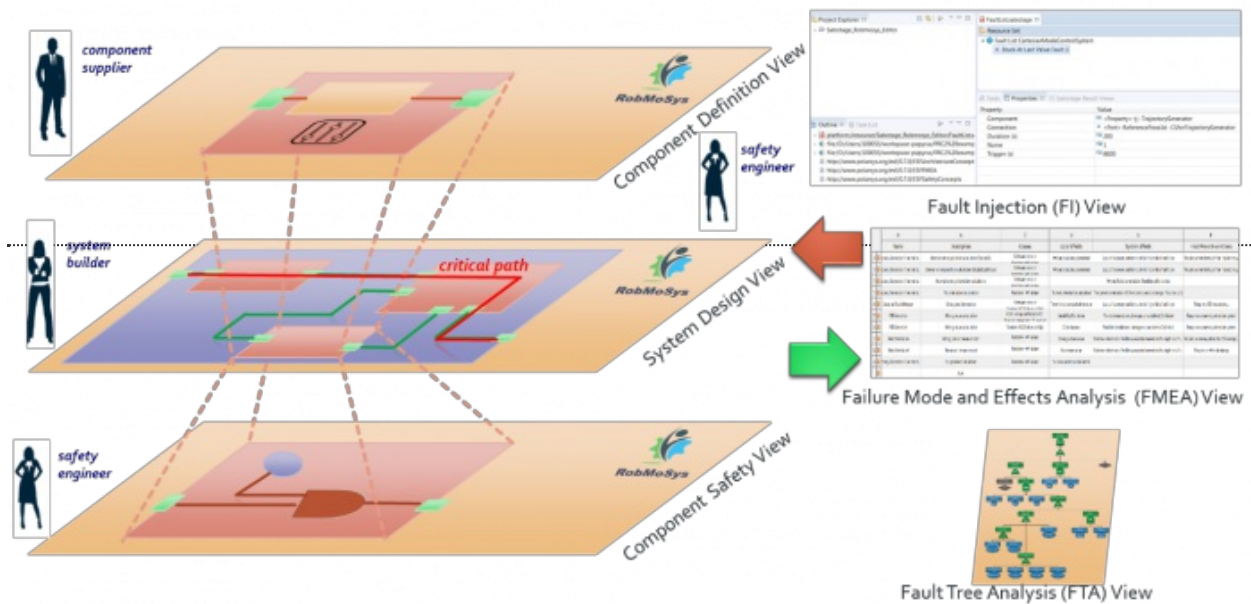


Figure 2. Safety Analysis with RobMoSys

The RobMoSys project defines structures which enable the management of the interfaces between different robotics-related domains, levels of abstraction and roles. eITUS will broaden the ecosystem by considering safety aspects such as the development of a safety view and the introduction of a new role called safety engineer (cf. Figure 3).

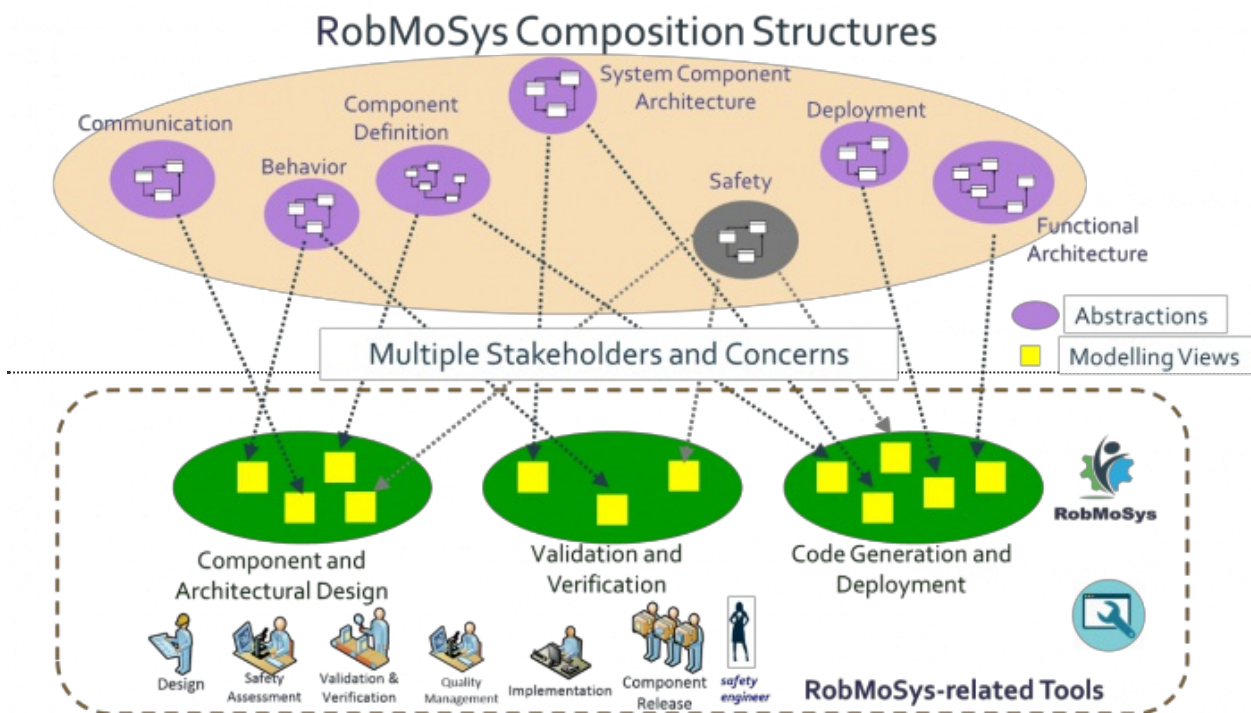


Figure 3. eITUS in terms of RobMoSys Composition Structures

The eITUS approach is explained by using a Cartesian Mode Control System as a use case scenario, whose

model in Papyrus is depicted in Figure 4.

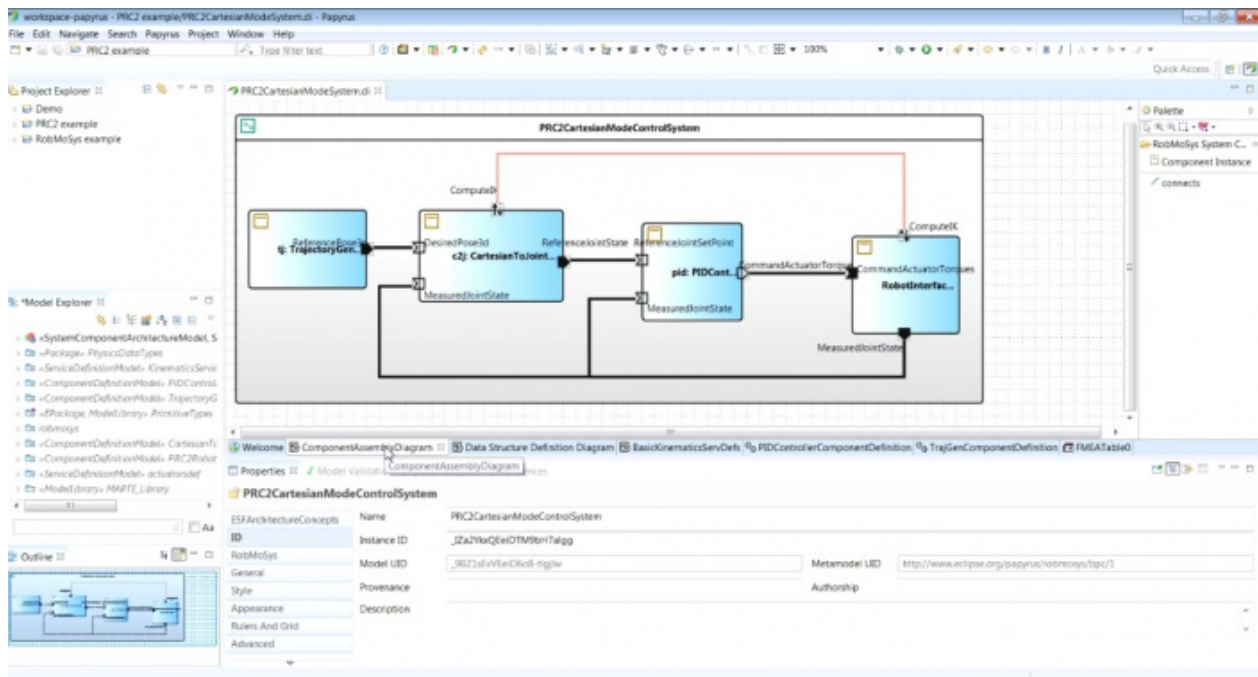


Figure 4. Cartesian Mode Control System modelled in Papyrus4Robotics

Afterwards, the safety engineers created and completes its associated FMEA.

	A	B	C	D	E
	Name	Description	Causes	Local Effects	System Effects
0	jectory Generator: Kinematics...	Internal wrong position calculation (Stuckat)	Software error or Random HW Failure	Wrong trajectory generated	Loss of runaway position control. High risk of collision
1	jectory Generator: Kinematics...	Internal wrong position calculation (Stuckat/setValue)	Software error or Random HW Failure	Wrong trajectory generated	Loss of runaway position control. High risk of collision
2	jectory Generator: Kinematics...	Internal wrong orientation calculation	Software error or Random HW Failure	Wrong trajectory generated	Wrong Robot orientation. Possible collision risk
3	jectory Generator: Kinematics...	No orientation calculation	Random HW Failure	No new orientation calculated	The current orientation of the robot does not change. Possible collision
4	CartesianToJointMapper	Wrong transformation	Software error or Random HW Failure	There is not an equivalent value	Loss of runaway position control. High risk of collision
5	PIDController	Wrong value calculation	Poor Tuning (Software error)	Instability/Oscillation	Possible breakdown, damages or accidents (Collision)
6	PIDController	Wrong value calculation	Random HW Failure (BitFlip)	Disturbances	Possible breakdown, damages or accidents (Collision)
7	RobotInterface4	Wrong sensor measurement	Random HW Failure	Wrong sensor value	Unknown robot state. Possible unwanted movement with a high risk of collision
8	RobotInterface4	No sensor measurement	Random HW Failure	No sensor value	Unknown robot state. Possible unwanted movement with a high risk of collision
9	jectory Generator: Kinematics...	No position calculation	Random HW Failure	No new position calculated	Loss of runaway position control. High risk of collision
		N/A			

Figure 5. Safety Analysis: FMEA view

Once the component failure modes are determined, fault injection simulations can be executed. The eITUS framework sets up, configures, executes and analyses the simulation results. Model-based design combined with a simulation-based fault injection technique and a virtual robot poses as a promising solution for an early safety assessment of robotics systems. The added value of including robots and environment models is that the maximum time before the robot dynamics are unsafely affected can be identified. In other words, it allows quantitatively estimating the relationship of an individual failure to the degree of misbehaviour on robot level.

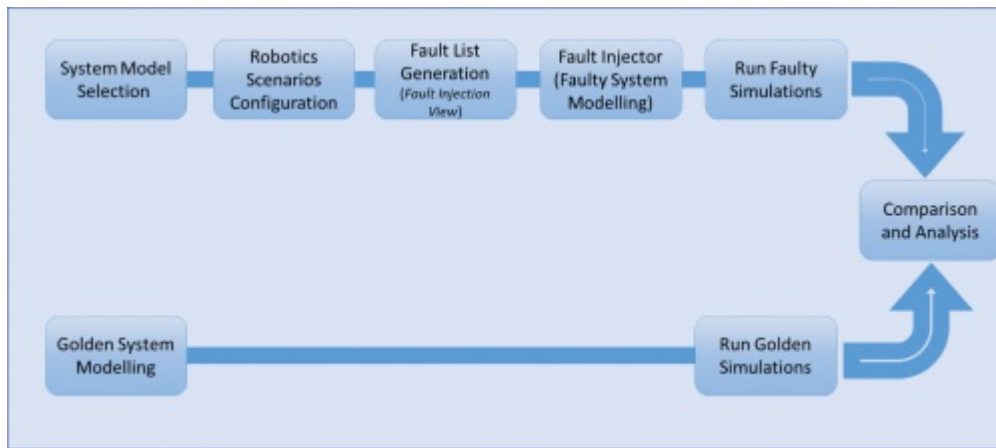


Figure 6. Workflow of Fault Injection Simulations

Before starting the fault injection experiments, the Golden system model, which represents a model without any faults in place, and its corresponding simulations must exist.

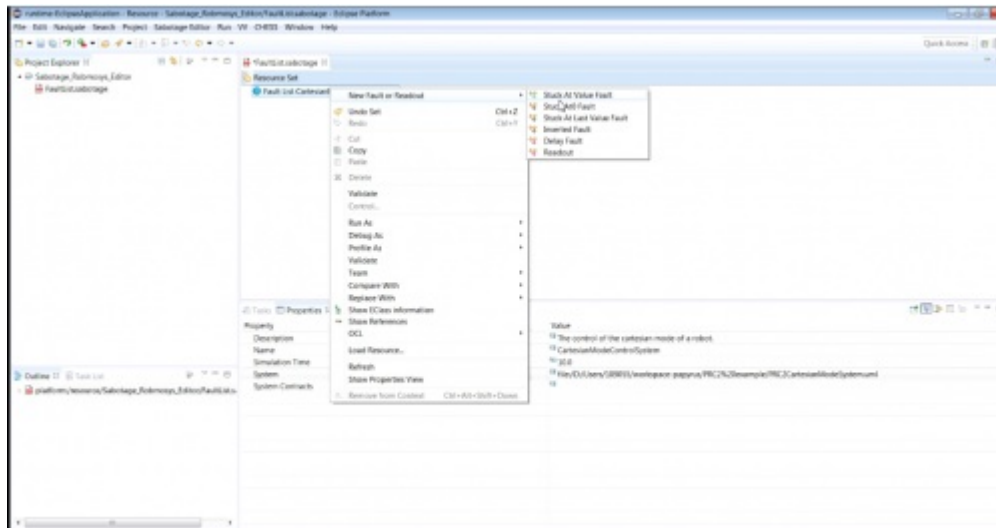


Figure 7. Fault Injection View: Creation of the Fault List

Once the Golden results have been executed, the safety engineer starts by selecting the system model and the robotics scenario, which includes the operational situation and the robot.

After that, it is important to define the fault injection policy which is referred to as the fault list. This configuration process includes the definition of fault locations (where to inject the fault?), fault injection times (When to trigger the fault?), fault durations (For how long the fault present in the system is?) and the fault model (How does the component fail?).

The original system model is modified though the fault injector script according to the fault list. Out of these faulty models the deployed code is generated, and the simulations are run.

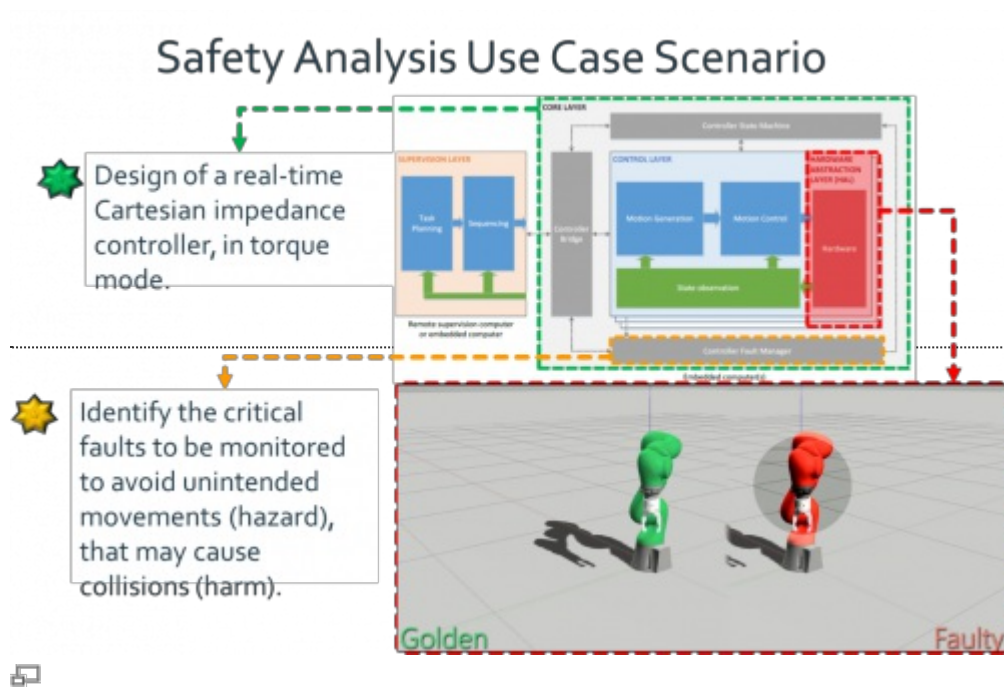
Finally, the obtained simulation traces are compared with respect to the Golden ones. This allows determining if a sufficient level of safety has been reached.

The following video shows:

- A use case scenario based on Papyrus4Robotics and extended with safety concepts (e.g. failure mode) and safety analysis (FMEA and Fault Injection Views).
- A safety analysis for a real time cartesian impedance controller.

- A real time cartesian impedance controller designed by with RobMoSys Golden and faulty components in the Gazebo simulator.

It is important to highlight how this is an ongoing work and further improvements are planned to be released by the end of the project.



From a technical perspective, the benefits of the eITUS methodology and tools will lead to:

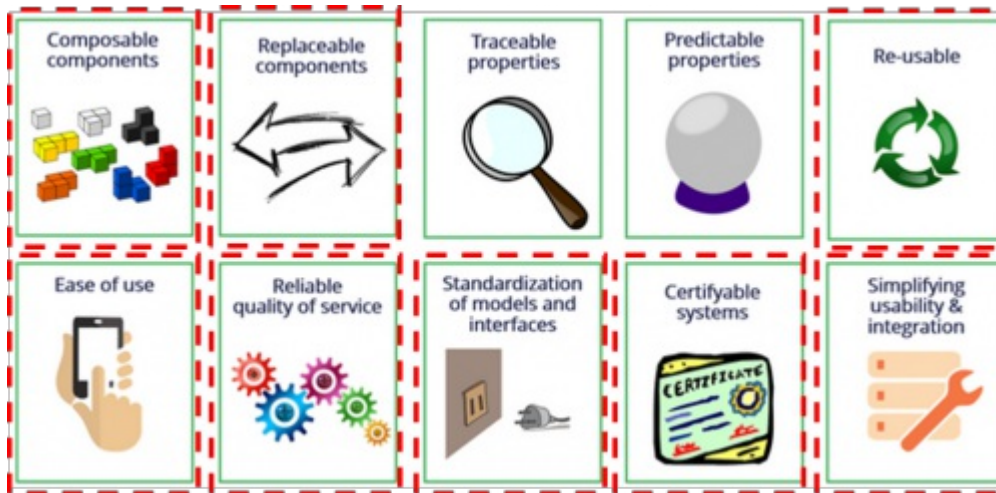


Figure 8. Benefits of the eITUS methodology from a technical perspective

- **Composable Components:**
 - The already defined software components can be used to compose a certain application such as the real-time Cartesian Impedance Controller. The same applies to safety artefacts.
- **Replaceable Components:**
 - eITUS uses a robot arm for the use case, however, replacing it with a different robot would be possible.
- **Re-Usable:**
 - eITUS supports the modeling of reusable domain- and application-specific safety analyses.
- **Ease of Use:**
 - eITUS provides an easy way to model safety related aspects and integrate them in the development Flow.
 - eITUS supports the separation of roles and views by defining a safety engineer responsible for the FMEA view completion.
- **Reliable Quality of Service:**
 - eITUS addresses Safety aspects considered an integral part of quality
- **Standardisation of models and interfaces:**
 - eITUS standardises safety nomenclature such as the definition of FMEAs or failure modes.
- **Certifyable Systems:**
 - eITUS helps on developing and delivering safety analyses in a formal way, by creating FMEA and Fault Injection tests. Safety artefacts such as FMEA are totally required to proceed to the certification of safe robotics system.
- **Simplifying Usability and Integration:**
 - eITUS integrates safety analysis views with fault injection simulations in a simple and transparent way.

The main incentives from commercial point-of-view are described in Figure 9.

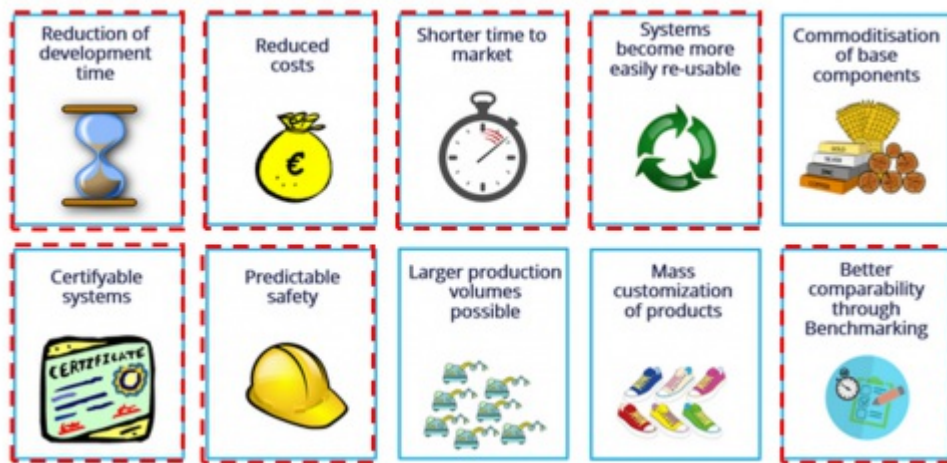


Figure 9. eITUS main incentives from commercial point-of-view

Acknowledgements

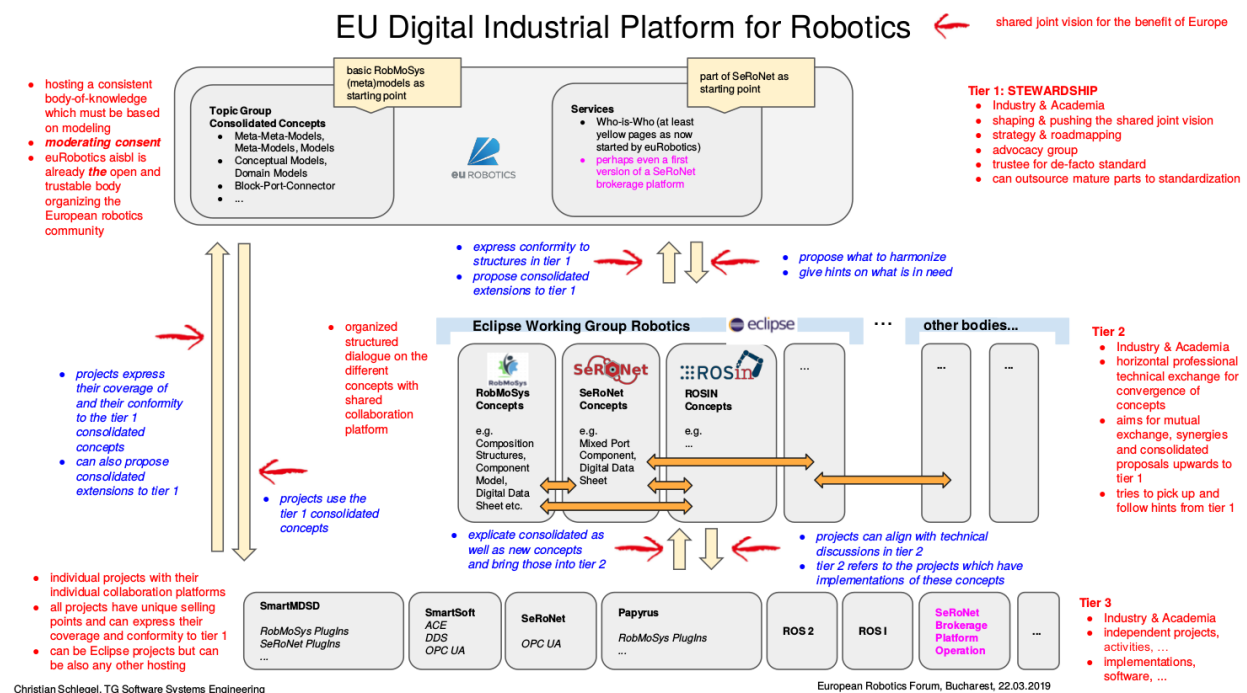
This demonstration has been performed by the eITUS consortium as a RobMoSys (robmosys.eu) Integrated Technical Project (ITP). This project is a joint effort between AKEO Plus, Tecnalia Research and Innovation and CEA, which is a RobMoSys core partner.

community:safety-analysis:start · Last modified: 2019/05/20 10:49
<http://www.robmosys.eu/wiki/community:safety-analysis:start>

EU Digital Industrial Platform for Robotics

RobMoSys is part of activities towards an “EU Digital Industrial Platform for Robotics” as a shared joint vision for the benefit of Europe in robotics. Below is a slide from the European Robotics Forum, Bucharest, 22.03.2019 presented in the context of the Topic Group “Software Systems Engineering”. It describes a proposal of how to organize bodies and projects in three tiers to advance the european ecosystem towards an EU Digital Industrial Platform for Robotics. This proposal is shared by RobMoSys.

Full set of slides



See also:

- Stewardship of the euRobotics Topic Group "Software Software Engineering, Systems Integration and Systems Engineering" [<https://sparc-robotics-portal.eu/web/software-engineering/stewardship-software-engineering-systems-integration-and-systems-engineering>]
- XITO [<https://www.xito.one/>]: The marketplace and application building platform for robotics based on RobMoSys technology
- Roadmap of Tools and Software
- Roadmap of MetaModeling

Dealing with Metrics on Non-Functional Properties in RobMoSys (RoQME ITP)

The purpose of this demonstration is to realize a proof-of-concept of the RoQME approach in a real-world example. It is a test-bed for current developments and provides valuable foundation for further developments.

Introduction

The demonstration focuses on the run-time aspects of RoQME as you will see RoQME in action. The design-time part of RoQME and how RoQME will be presented to the user through tooling will be covered in an update to this demonstration (see roadmap).

The video below illustrates a typical intralogistics scenario. In this scenario the robot receives boxes from given stations and delivers them to different stations. The setting is equipped with a sensor detecting when a person enters or exits the robot working area. The video illustrates how the RoQME approach can provide metrics on non-functional properties such as safety or performance.

Disclaimer: This demonstration is a proof-of-concept of the technical feasibility. Please look at the roadmap to see how it is planned to advance this demonstration and to make it generally accessible via tooling.

Goal of RoQME in the context of RobMoSys

The RoQME project aims to provide a comprehensive set of model-based tools enabling:

1. The specification of global robot Quality of Service (QoS) metrics defined on Non-Functional Properties (NFP). These metrics are defined in terms of the (internal and external) contextual information available to the robot; and
2. The generation (from the previous specification) of an actual RobMoSys-compliant component providing information to other components (either continuously or on demand) about the runtime evolution of the metrics previously specified.

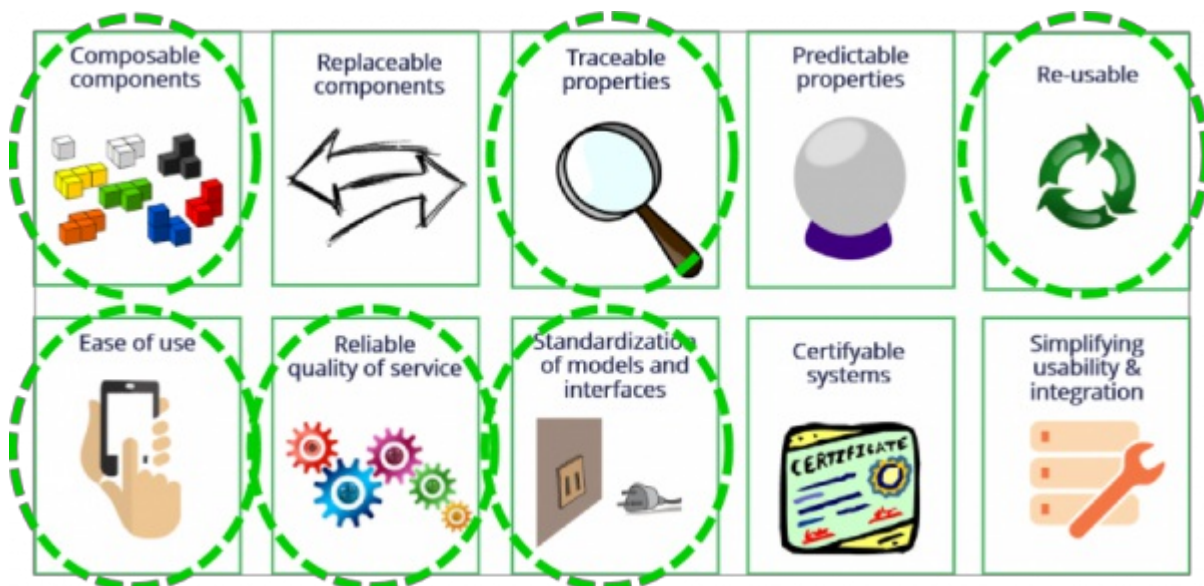
It is worth noting that RoQME enables modeling all kinds of system-level (rather than component-level) NFP, such as, safety, performance, reliability, resource consumption, user comfort, etc. It is also worth mentioning that RoQME does not prescribe how the computed metrics are used by other components, probably in a distributed manner, as the intrinsic cross-cutting nature of NFP may scatter and have impact both in the robot behaviour and architecture.

What RoQME provides

- A new **QoS Engineer role** (see section Technical Details), associated with a **RoQME view**, specifically dedicated to the modeling of QoS metrics defined on NFPs. The RoQME View builds on:
 - The **RoQME meta-model**, which provides QoS Engineers with the modeling concepts needed to specify:
 - What are the relevant NFPs in a particular robotic application;
 - Which contextual information is available to the robot; and
 - How the different contexts impact the selected NFPs. These impacts will then be used to compute the QoS metrics defined on the different NFP.
 - A **RoQME-to-RobMoSys mapping meta-model** enabling the seamless integration of the RoQME models into the RobMoSys Ecosystem.
 - **RoQME plugins for the SmartMDSD Toolchain** providing:
 - A textual **model editor** enabling the creation and validation of QoS models by Domain Experts and QoS Engineers.
 - A **model-to-code transformation** enabling the generation of a metrics provider component that conforms to the RobMoSys structures. This tool is intended to be used by QoS Engineers.
 - A **QoS Metrics Service Definition** enabling the use of the RoQME QoS Metrics Provider by any component in a RobMoSys conformant architecture, willing to use the computed metrics.

This demonstration is in the Context of RobMoSys User-Stories

In the context of the RobMoSys technical user stories [<https://robmosys.eu/user-stories/>], this demonstration shows:



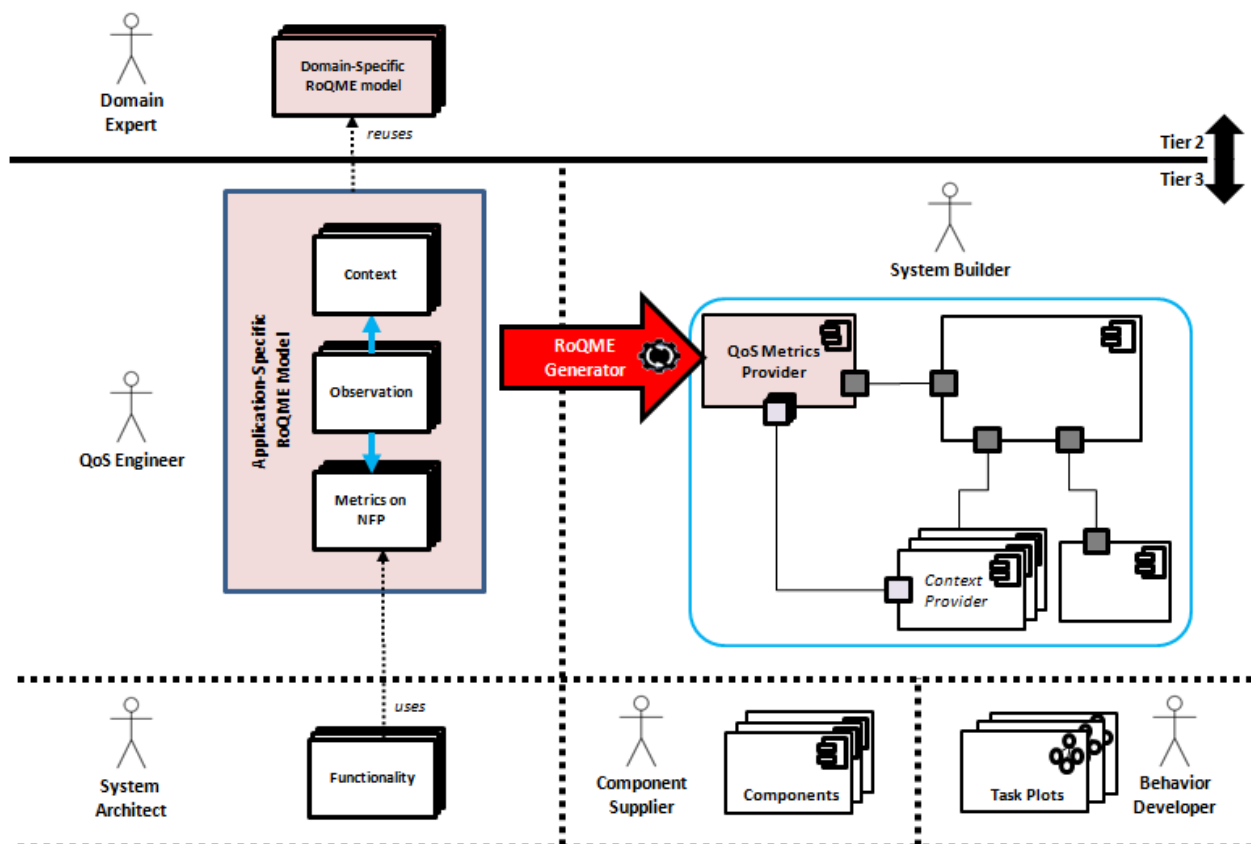
- **Composable components**
 - The RoQME QoS Metric provider is a composable software component generated from a RoQME model, i.e., it can be seamlessly composed into a RobMoSys-compliant system
 - The scenario was composed out of previously developed software components (e.g., components for hardware access, mapping, path planning, collision avoidance, machine-to-machine

communication, box pickup/drop-off) and the newly generated RoQME QoS Metric provider component

- **Traceable properties**
 - RoQME supports traceable NFP, i.e., it is possible to trace which contexts are being responsible for the evolution of the metrics defined on the selected NFP. In other words, it is possible to trace back what causes the improvement or degradation of each NFP.
- **Ease of use**
 - RoQME provides an easy way to model QoS metrics defined on NFP
 - RoQME supports separation of roles and views (e.g. QoS Engineer focusing on NFP vs. System Architect focusing on functional aspects)
- **Re-usable models**
 - RoQME supports the modeling of reusable domain- and application-specific metrics on NFP
- **Reliable QoS**
 - RoQME enables the use of QoS metrics on NFP (e.g., for adaptation, benchmarking, etc.)
- **Standardization**
 - RoQME domain-models defined at tier-2 can be (re-)used in a variety of concrete tier-3 applications (e.g., standard metrics on safety defined in the context of mobile robotics can be reused/refined as standard metrics in the context of intralogistics, and these, in turn, reused/refined in different particular intralogistics applications).

Technical Details

The following figure outlines the role, the models and the software artifacts contributed by RoQME and how they relate to those of RobMoSys. These contributions and the links to RobMoSys are briefly described next. Further details can be found at [Vicente2018a].



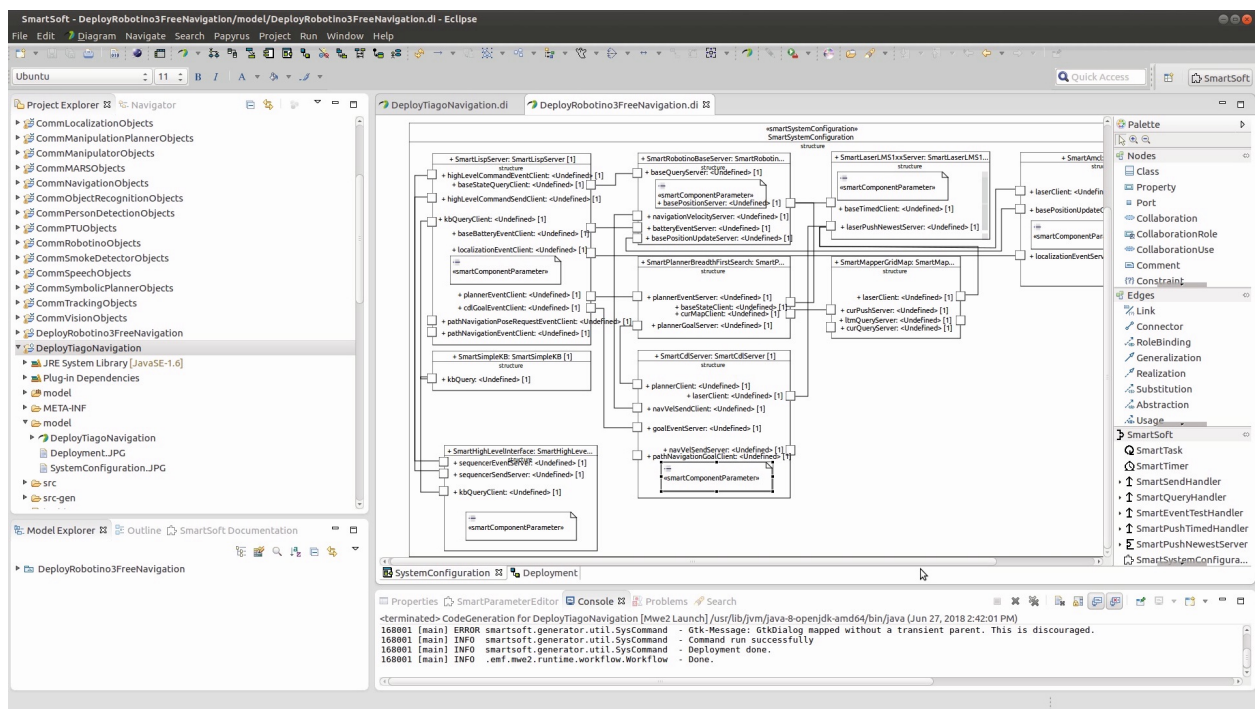
Domain Experts (Tier-2) identify and agree the specification of relevant domain-specific non-functional properties. The resulting **Domain-Specific RoQME Models** are made available in a RobMoSys repository for being reused by other roles at Tier-3.

QoS Engineers create **Application-Specific RoQME Models** that gather the non-functional aspects to be measured in the robotics system under development while, in parallel, the System Architect describes its functionality. The QoS Engineer may create application-specific RoQME models from scratch or by reusing and extending some of the domain-specific RoQME models already available in the RobMoSys repository. The models developed by the QoS Engineer specify (1) the non-functional properties considered relevant for the application being developed; (2) the contextual information available to the robot; and (3) a number of observations describing how (2) impacts (1). These observations will enable the estimation of the QoS metrics defined on the selected non-functional properties.

The QoS Engineer uses the **RoQME Generator** to create a **QoS Metrics Provider** component from the previous model. The resulting component is made available to the System Builder, who integrates it into the RobMoSys application architecture by (1) connecting it to the corresponding context providers, and (2) to any other component willing to use the computed metrics (if any).

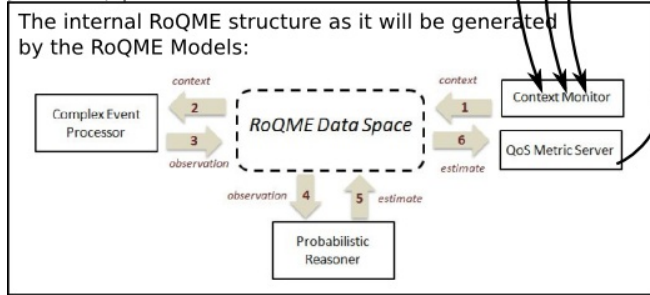
System Component Architecture

The figure included below shows a screenshot of the system component architecture diagram, as modeled in the SmartMDSD Toolchain. The system was composed from previously developed software components.



Metrics Component

The following picture shows the QoS Metrics Provider component used in the demonstration. The hull of this component was developed using the SmartMDSD Toolchain, while its internals were manually implemented using the support software libraries developed in RoQME for context monitoring, complex event processing, and metrics estimation. Further developments in RoQME will realize the full generation of this component based on the RoQME models (see roadmap). As illustrated in the picture, the component receives contexts from other components in the system (context providers) and processes them according to the RoQME approach (lower left). It then provides the estimated QoS metrics to other components for further processing or adaptation (e.g., by the Sequencer).



The RoQME Model for the Experiment

The RoQME model detailed next specifies: (1) the non-functional properties identified as relevant for the experiment and for which the RoQME runtime infrastructure will provide estimations (metrics) in the range [0, 1]; (2) the context data provided by the robot; and (3) the observations which, based on the specified context patterns, will more or less impact (reinforcing or undermining) the previous non-functional properties. This model uses a preliminary syntax, which is subject to improvement in the final textual model editor that is currently being developed in Rome.

Non-Functional Properties:

- **[SAFETY]:** degree to which the robot is successful in producing the following desired results:
 - The robot does not bump
 - The velocity of the robot is bounded (lower than V) when there is a person in the scenario area.

It is worth mentioning that the reference value for SAFETY is 1, that is, a priori, the system is considered safe and, only if there is evidence against this belief, the value of the safety metric is reduced accordingly. The reference value of a property is its a-priori probability of being optimal, and is the value the property tends to when there is no evidence making it improve (reinforcing evidence) or worsen (undermining evidence). Note that most properties use a reference value equal to 0.5.

- **[PERFORMANCE]:** degree to which the robot is successful in producing the following desired results:
 - The robot completes the tasks successfully (i.e., tasks are not aborted and the robot does not enter into an ERROR state).
 - The robot completes each task within a given time-slot (MAX_JOB_TIME)

Contexts:

Contexts:

- Bump : Event
- Velocity: number (unit: m/s)
- Robot_State : Enum {

```

    IDLE_NOT_CHARGING,
    IDLE_CHARGING,
    BUSY_DRIVING_WITH_LOAD,
    BUSY_DRIVING_EMPTY,
    ERROR
  }
  • Job_State : Enum {
    NOT_STARTED,
    STARTED,
    FINISHED,
    ABORTED
  }

```

Derived context (from Job_State):

- time_jobDone: Number > 0 (unit: s)

This derived context is calculated as the time passed since a job starts (Job_State::STARTED) until it finishes (Job_State::FINISHED).

- Person_State: Enum {
 IN,
 OUT
 }

Observations:

- O1. Bump **undermines** [SAFETY] VERY_HIGH
- O2. Velocity > V && Person_State::PERSON_IN **undermines** [SAFETY] VERY_HIGH
- O3. Job_State::JOB_FINISHED && time_jobDone < MAX_JOB_TIME \Rightarrow **reinforces** [PERFORMANCE] HIGH
- O4. Robot_State::ERROR **undermines** [PERFORMANCE]
- O5. Job_State::ABORTED **undermines** [PERFORMANCE]

Description of the Story

The story starts with the robot in an IDLE state and placed in its initial position. There is an operator ready to handover boxes to the robot.

1. The robot is ordered a new task (Job_State::STARTED) so it moves to the handover position (Robot_State::BUSY_DRIVING_EMPTY). When it is ready, the operator places a box on top of it. The robot moves to the delivery position (Robot_State::BUSY_DRIVING_WITH_LOAD) and puts the box on the belt. The task is completed successfully (Job_State::FINISHED) and within the required time-slot \Rightarrow Observation O3 is fired and **PERFORMANCE** improves.
2. A visitor enters the room and meets the operator (Person_State::IN). Simultaneously, the robot is ordered a new task (Job_State::STARTED) so it starts moving again to the handover position (Robot_State::BUSY_DRIVING_EMPTY). On its way, the robot moves faster than allowed when a person is in the robot working area \Rightarrow Observation O2 is fired and **SAFETY** is significantly worsen.
3. The visitor, not being aware of the robot, bumps into it (Bump) when turning around for leaving the room \Rightarrow Observation O1 is fired and **SAFETY** is significantly worsen.
4. Immediately after the bumping, the robot enters into the ERROR state (Robot_State::ERROR) \Rightarrow Observation O4 is fired and **PERFORMANCE** is worsen.
5. After the error, the task is aborted (Job_State::ABORTED) \Rightarrow Observation O5 is fired and **PERFORMANCE** is worsen.

6. The robot is ordered a new task (Job_State::STARTED) which is successfully completed (Job_State::FINISHED) \Rightarrow Observation O3 is fired and **PERFORMANCE** improves.

Visualization

RoQME provides a visualization tool that displays at run-time: (1) the contextual information obtained from the robot; (2) the observations fired when the context patterns identified in the RoQME model are detected; and (3) the estimation of the metrics defined on the non-functional properties, as they vary according to the previous observations. The following screenshots show the graphics produced by the visualization tool during the experiment for the SAFETY (top) and the PERFORMANCE (bottom) non-functional properties, together with the observations with impact on their estimations. Labels (numbered circles) referring to the different story steps described in the previous section have been manually added to make the experiment timeline clearer.



Simulation and benchmarking

RoQME also provides a module for recording the data generated while running real-world experiments (i.e., experiments that involve real robots). It records: a) the context data generated by the robot/s; b) the identified observations, and c) the QoS metric estimations. The recorded data can then be used for simulating the experiment as many times as needed without the need of using the real robot/s again. Besides, this information can also be used, for instance, for benchmarking purposes.

Current State and Roadmap

The core functionality of collecting contexts and processing metrics on non-functional properties is already operational in RoQME although, at the moment, the QoS metrics provider component is implemented manually.

The following items are currently being developed and are expected to be available and usable in the form of RoQME plugins for the SmartMDSD Toolchain. Its expected release date is March 2019. It will cover:

- Modeling support for the Domain Expert to model QoS Domain Models (Tier 2)
- Modeling support for the QoS Engineer to model QoS Models (Tier 3)
- A number of code generators for automatically deriving all the RoQME software artifacts, in particular, the hull and the internals of the QoS Metrics Provider Component. In case the complex event processor or the reasoner (or both), need to be executed out of the QoS Metrics Provider Component, i.e., out of the robot (in different computers), they will be also generated as independent artifacts.

All the RoQME partners are pledged to making the knowledge generated in the course of the Project as widely and freely available as possible for subsequent research and development. For this reason, the RoQME partners are fully committed to open-access and open-source policies.

Discussion

To discuss this demonstration, see: <https://discourse.robmosys.eu/t/demonstration-dealing-with-metrics-on-non-functional-properties-in-robmosys-roqme-ityp> [<https://discourse.robmosys.eu/t/demonstration-dealing-with-metrics-on-non-functional-properties-in-robmosys-roqme-ityp>]

Acknowledgements

This technical demonstration has been realized by the RoQME project team and the Ulm University of Applied Sciences. The methods and tools behind the estimation of the non-functional properties are being developed by the RoQME ITP (University of Extremadura, University of Málaga and Biometric Vox). The SmartMDSD Toolchain and the intralogistics use-case/pilot application are being developed by the Ulm University of Applied Sciences.

RoQME is an Integrated Technical Project (ITP) of EU H2020 RobMoSys (robmosys.eu). Ulm University of Applied Sciences is a RobMoSys core partner.

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

See also

For further information and to learn more about the concepts used here, see:

- [The SmartSoft World and the The SmartMDSD Toolchain](#), the tool that was used to develop the software components and to compose the application.
- This scenario uses software components from the [Intralogistics Industry 4.0 Robot Fleet Pilot](#)
- [RoQME Integrated Technical Project](#) [<https://robmosys.eu/roqme/>].

You can follow the RoQME Project updates at:

- [The ResearchGate RoQME Project](#) [<https://www.researchgate.net/project/RoQME-Dealing-with-non-functional-properties-through-global-Robot-Quality-of-Service-Metrics>]
- [The Linked RoQME Group](#) [<https://www.linkedin.com/groups/12096769/>]

- The RoQME Twitter account[https://twitter.com/roqme_itp]
- The RoQME section in the RobMoSys website[<https://robmosys.eu/roqme/>]
- The Community Corner section in the RobMoSys wiki[<https://robmosys.eu/wiki/community:start>]

Related Publications

[Espin2018] Espín, J. M., Font, R., Ingles-Romero, J. F. & Vicente-Chicote, C. Towards the Application of Global Quality-of-Service Metrics in Biometric Systems. IberSPEECH 2018, 21-23 November 2018, Barcelona (Spain). Download. [Download \[https://www.researchgate.net/publication/328890945\]](https://www.researchgate.net/publication/328890945)

[Ingles2018a] Inglés-Romero, J. F., Spín, J. M., Jiménez, R., Font, R. & Vicente-Chicote, C. Towards the Use of Quality of Service Metrics in Reinforcement Learning: A Robotics Example. 5th International Workshop on Model-driven Robot Software Engineering (MORSE 2018), in conjunction with MODELS 2018, 15 October 2018, Copenhagen (Denmark). Download. [Download \[https://www.researchgate.net/publication/327243001\]](https://www.researchgate.net/publication/327243001)

[Ingles2018] Inglés-Romero, J. F. How well does my robot work? RoQME: A project aimed at measuring quality of service in robotics. April 2018. Available at Medium.com. [Download \[https://medium.com/biometric-vox/how-well-does-my-robot-work-ca98ecc1ab79\]](https://medium.com/biometric-vox/how-well-does-my-robot-work-ca98ecc1ab79)

[Lutz2019] Lutz, M., Inglés-Romero, J. F., Stampfer, D., Lotz, A., Vicente-Chicote, C., & Schlegel, C. (2019). Managing Variability as a Means to Promote Composability: A Robotics Perspective. In A. Rosado da Cruz, & M. Ferreira da Cruz (Eds.), New Perspectives on Information Systems Modeling and Design (pp. 274-295). Hershey, PA: IGI Global. DOI: 10.4018/978-1-5225-7271-8.ch012 [Download \[http://doi.org/10.4018/978-1-5225-7271-8.ch012\]](http://doi.org/10.4018/978-1-5225-7271-8.ch012)

[Vicente2018a] Vicente-Chicote, C., Inglés-Romero, J. F., Martínez, J., Stampfer, D., Lotz, A., Lutz, M. & Schlegel, C. A Component-Based and Model-Driven Approach to Deal with Non-Functional Properties through Global QoS Metrics. 5th International Workshop on Interplay of Model-Driven and Component-Based Software Engineering (ModComp 2018), in conjunction with MODELS 2018, 14 October 2018, Copenhagen (Denmark). . [Download \[https://www.researchgate.net/publication/328102310\]](https://www.researchgate.net/publication/328102310)

[Vicente2018b] Vicente-Chicote, C., Berrocal, J., García-Alonso, J. M., Hernández, J., Bandera, A., Martínez, J., Romero-Garcés, A., Font, R. & Inglés-Romero, J. F., (2018). RoQME: Dealing with Non-Functional Properties through Global Robot QoS Metrics. SISTEDES Conference, 17-19 September 2018, Sevilla (Spain). [Download \[https://www.researchgate.net/publication/327239527\]](https://www.researchgate.net/publication/327239527)

community:roqme-intralog-scenario:start · Last modified: 2019/05/20 10:49
<http://www.robmosys.eu/wiki/community:roqme-intralog-scenario:start>



Page moved: <https://robmosys.eu/wiki/community:behavior-tree-demo:start>
[<https://robmosys.eu/wiki/community:behavior-tree-demo:start>]

community:behavior-tree-demo · Last modified: 2019/05/20 10:47
<http://www.robmosys.eu/wiki/community:behavior-tree-demo>

Robotic Behavior in RobMoSys using Behavior Trees and the SmartMDSD Toolchain (MOOD2be ITP)

This demonstration shows task-level composition (robotic behavior) in RobMoSys using behavior trees.

Introduction

MOOD2Be provides two pieces of softwares:

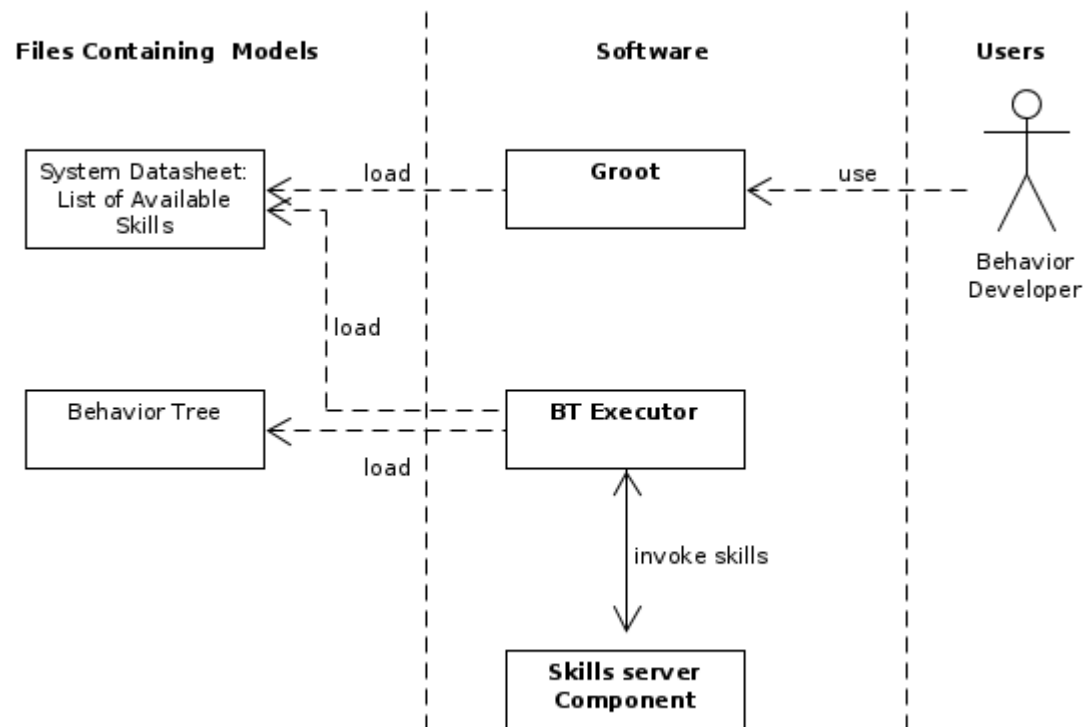
- **BehaviorTree.CPP**: a C++ framework and toolset to create, execute and debug Behavior Trees.
- **Groot**: a graphical IDE to create, edit, monitor and analyze Behavior Trees.

BehaviorTree.CPP allows the creation of a Executor where the actual behavior trees are executed, whilst Groot is a graphical tool to help the Behavior Design to be more productive. The demonstration focuses on the integration of the **SmartMDSD Toolchain**, an Integrated Development Environment (IDE) for robotics software development conform-to RobMoSys, with Groot, a graphical user interface to edit behavior trees. The development of the behavior tree is based on the skills provided by a system that was modeled using the SmartMDSD Toolchain.

Demonstration: building task-level coordinators using Behavior Trees

In the following video we can see the workflow from the perspective of the **Behavior Developer**:

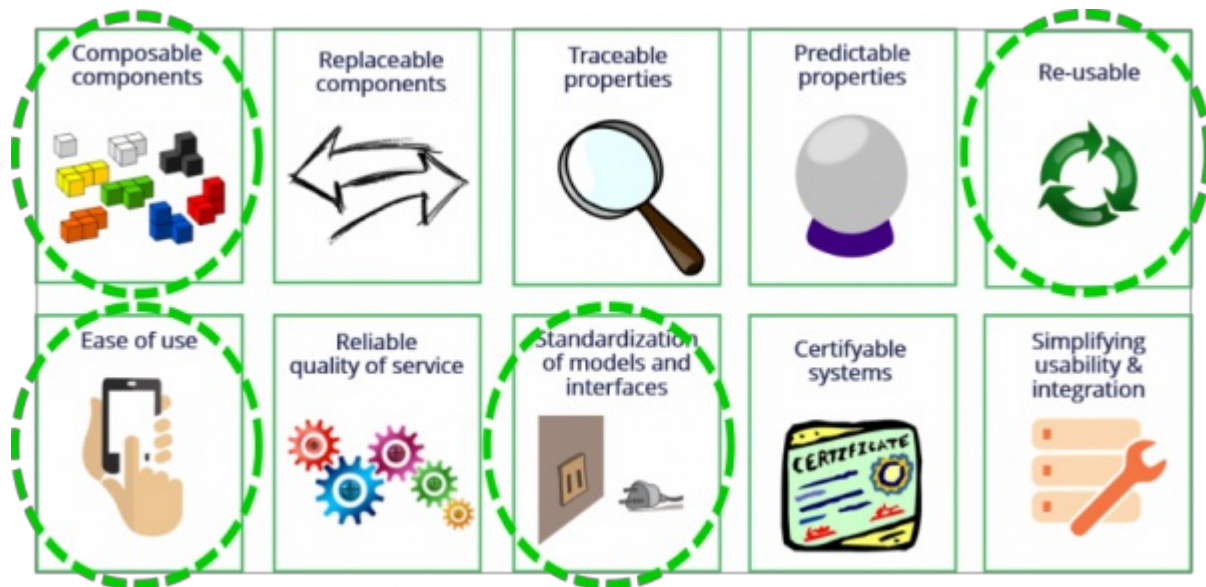
1. A list of available **Skills** is provided to the Behavior Developer in the form of a file containing the model of the skills. These Skills are part of the **Digital Datasheet** of a system.
2. The Behavior Developer loads this file into Groot and use the generated “palette” of Actions to design a **behavior tree**; the outcome of this process is also a file, containing the model of the tree.
3. A specific application (an “**Executor**” created using BehaviorTree.CPP) loads both these files to create and execute the behavior tree.
4. The Executor communicates through a the rest of the system using a Skill Server, which provides access to the components skills.



This process doesn't require any code-generation nor compilation, since both the nodes and the trees are create programmatically at run-time. This workflow is shown in the following video.

Demonstration is in the Context of RobMoSys User-Stories

In the context of the RobMoSys technical user stories [<https://robmosys.eu/user-stories/>], this demonstration shows:



- **Reusability:** Actions, Conditions and Skills, which corresponds to Nodes of a behavior tree, are highly reusable piece of software that are application independent. Furthermore, SubTrees can also be reusable as parts of more complex Trees.
- **Easy to use:** Groot is more than a simpleGUI. It is an IDE for behavior trees, which supports editing, logs analysis and real-time debugging. Both software and documentation are meant to provide a fast learning curve and high productivity.
- **Composable components:** trees are hierarchical and composable. Simple behaviors can be composed into complex ones.
- **Standardization of models and interfaces:** the skill interfaces are defined as part of RobMoSys Tier 2 Domain Models. These are general descriptions of skills and can be mapped to a specific software components (Tier 3) providing these skills (such as the robotino base robot provides the “move-to” skill).

Technical Details

Based on an existing robotics application developed with the SmartMDSD Toolchain and the Skills that are provided by the application’s components, the Behavior Developer selects and composes actions and conditions to develop the behavior tree.

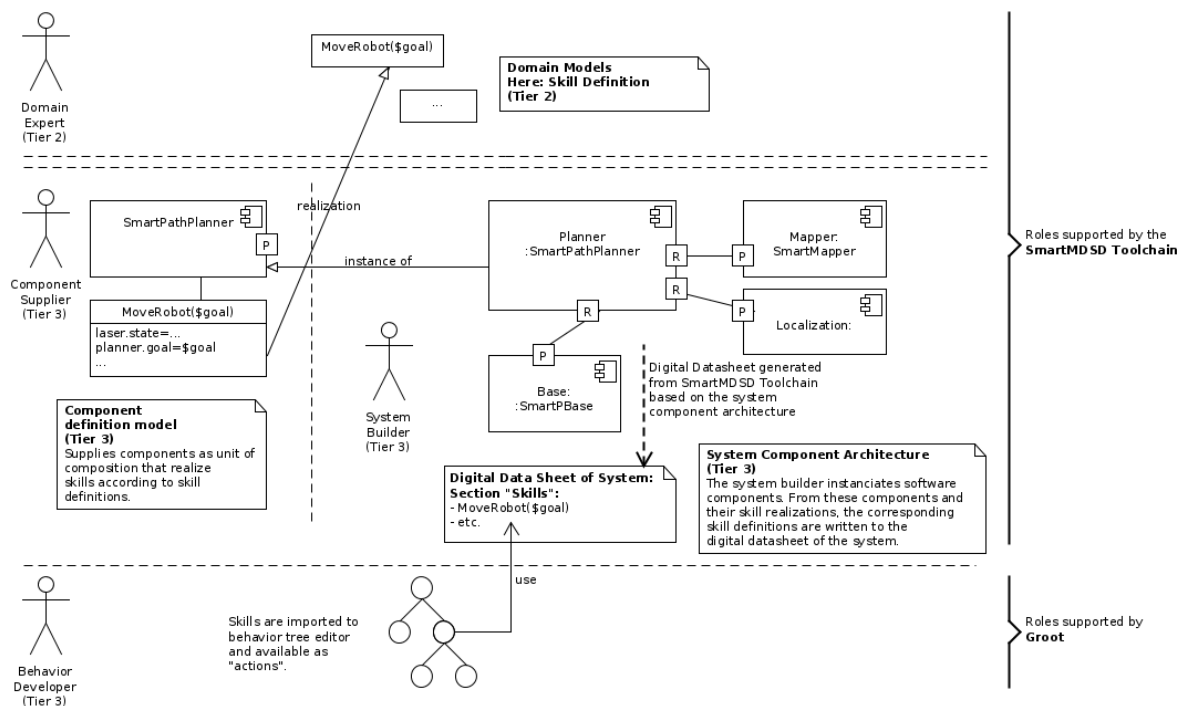
The behavior tree is used to do task-level coordination and its actions execute RobMoSys skills. These skills (e.g. MoveBaseToGoal) use the RobMoSys software component coordination interface to directly coordinate the components e.g. setting component configurations.

In the next diagram we can see how the different RobMoSys roles interact:

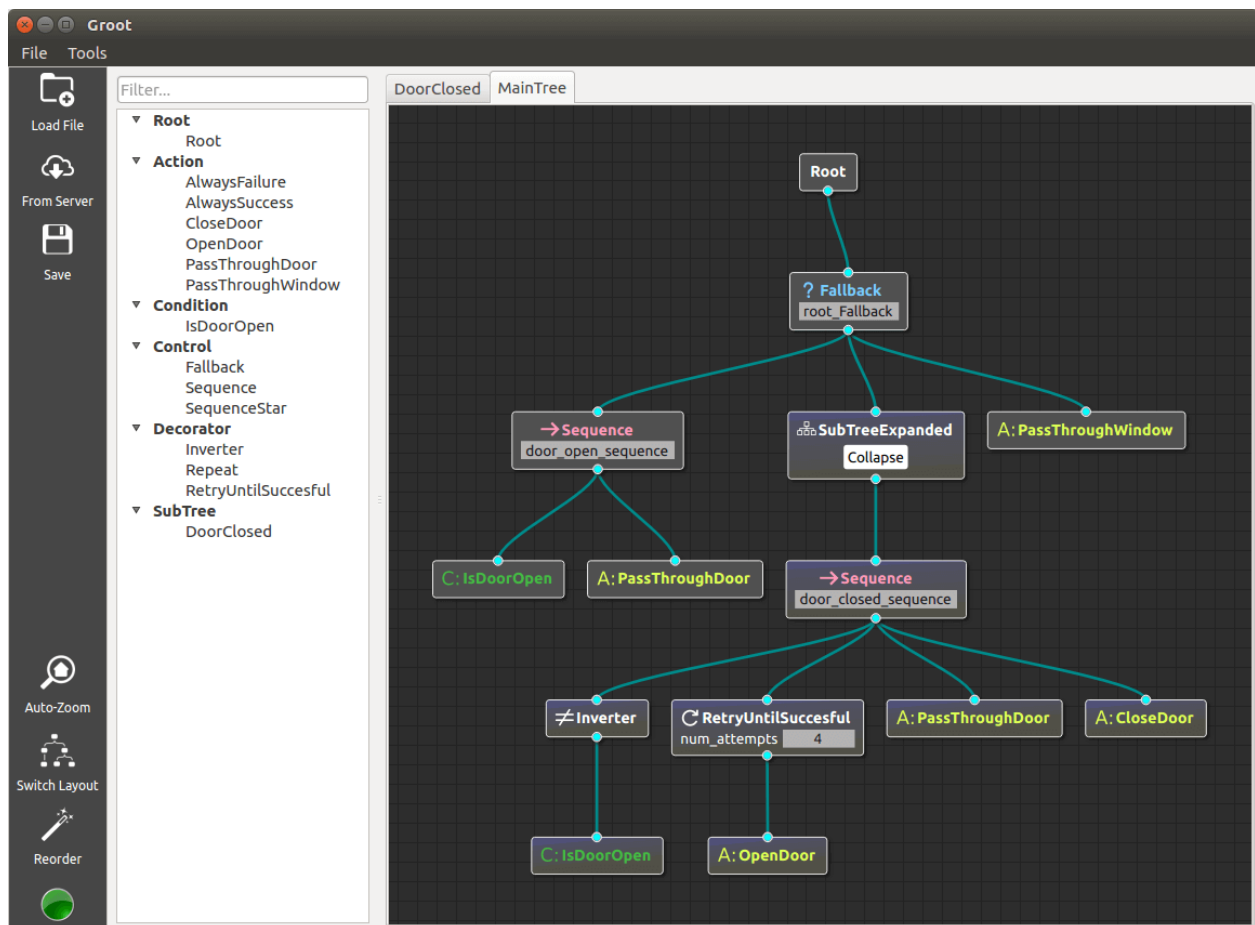
1. The Domain Expert defines a set of domain specific **Skill Definitions**.
2. These definitions are implemented as **Skill Realizations** by the Component Developer.
3. The System Builder composes multiple components; as a result, a particular software system provides a **set of Skills in the Digital Datasheet**.
4. The Behavior Designer uses this skills to create a Behavior Tree; the model of the tree can be directly executed.

The picture below illustrates this workflow. Note how the different roles are supported by different tools in the RobMoSys ecosystem: Domain Expert, Component Supplier and System Builder are supported by the

SmartMDSD Toolchain. The Behavior Developer is supported by Groot/BehaviorTree.CPP.

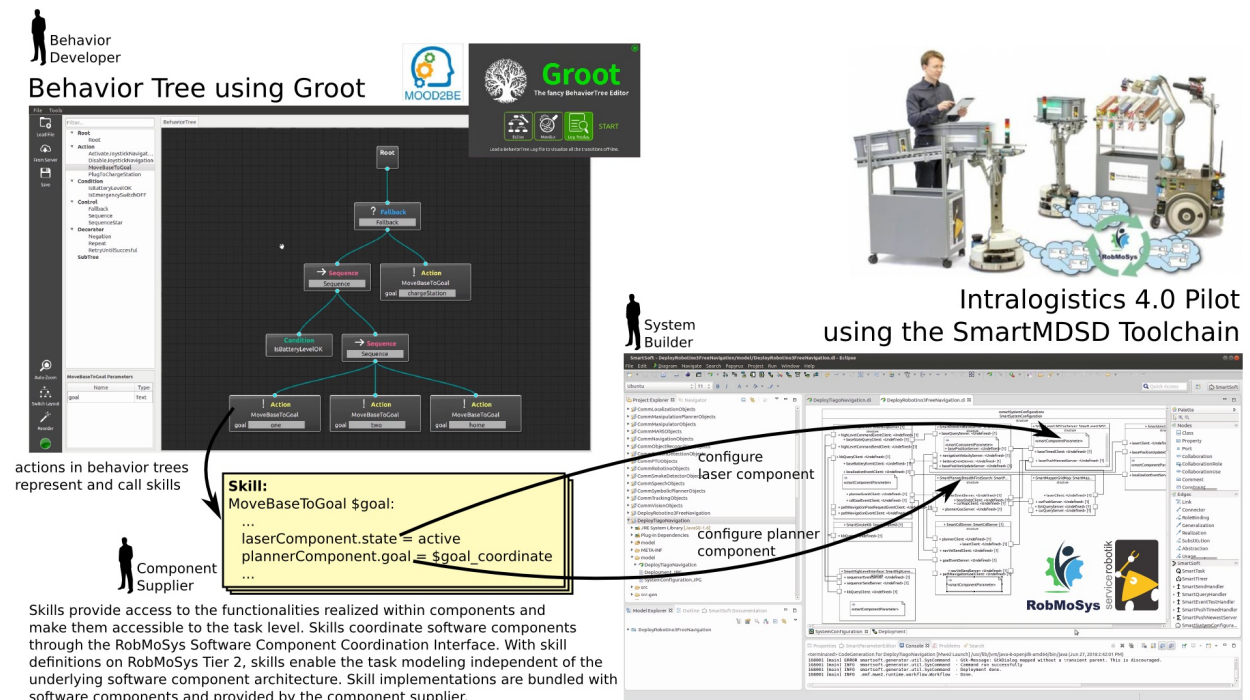


The below screenshot shows Groot in action while editing a behavior tree:



Previous demonstration (June 2018)

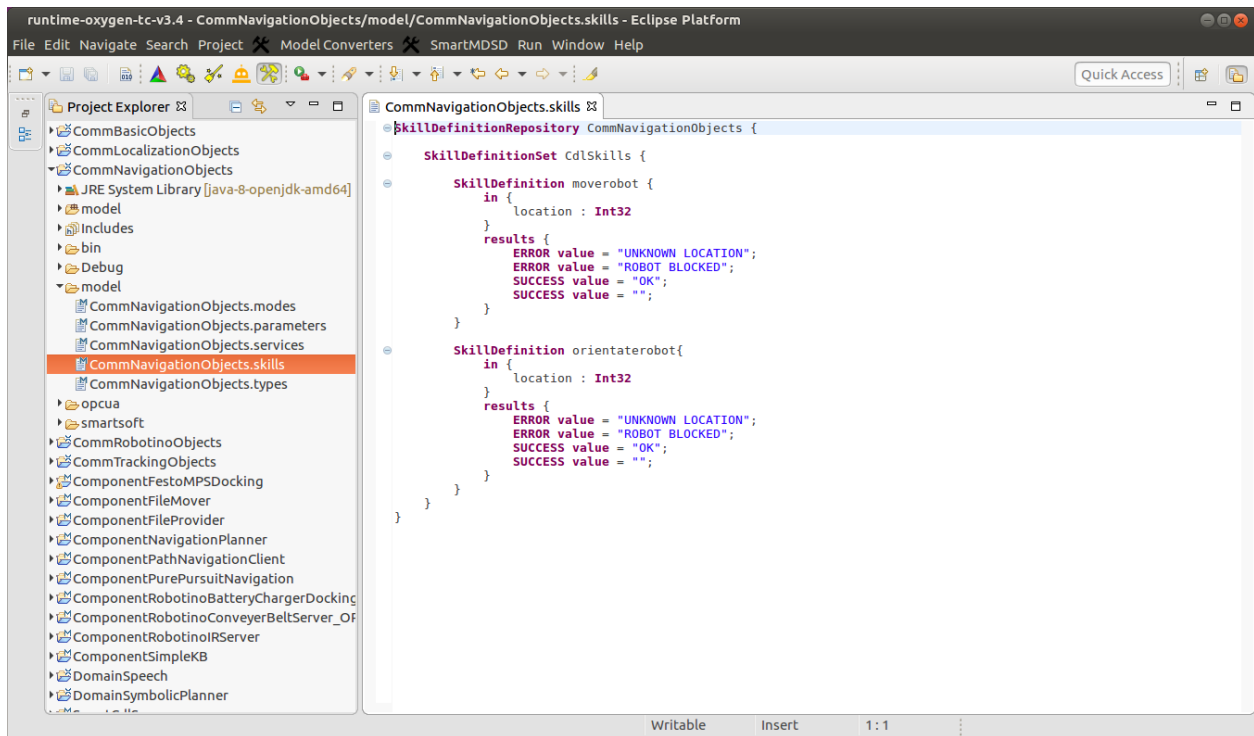
A previous version of this demonstration as of June 27 2018 focused on the technical feasibility of behavior trees communicating with components from the SmartMDSD Toolchain. At that time, it was manual effort by the system builder to “connect” the behavior engine with software components. This manual effort is now replaced by a full model-driven workflow and interaction between component supplier, system builder and behavior developer roles.



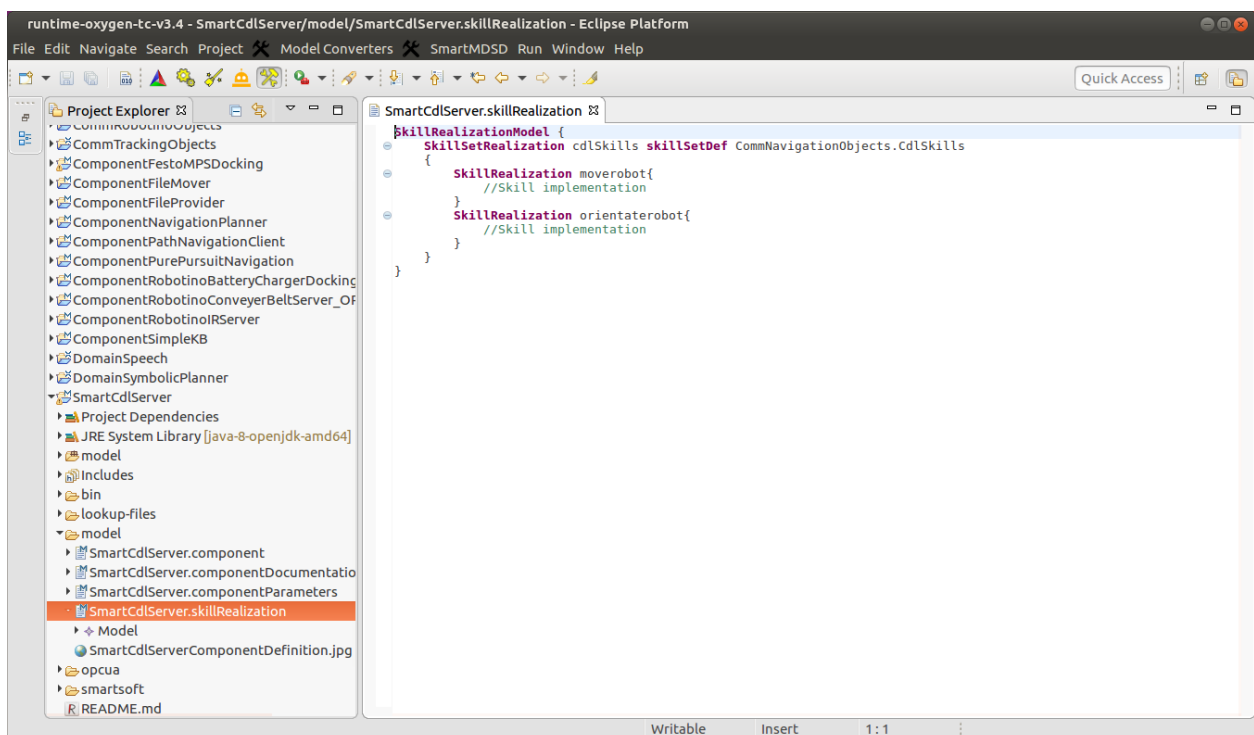
In the below video, the execution of the behavior tree is first shown in simulation using the “Gazebo/Tiago/SmartSoft Scenario” as provided by RobMoSys. The behavior tree is then executed on a FESTO Robotino Robot as part of the RobMoSys “Intralogistics Industry 4.0” Pilot. Finally, the video demonstrates the visualization of an automatically generated log file, that allow the user to analyze the execution of the behavior tree offline.

Skill Modeling by the SmartMDSD Toolchain

Skills are modeled by the **Domain Expert** role on Tier 2. The below screenshot shows the **skill definition** model in a Domain Model Project.



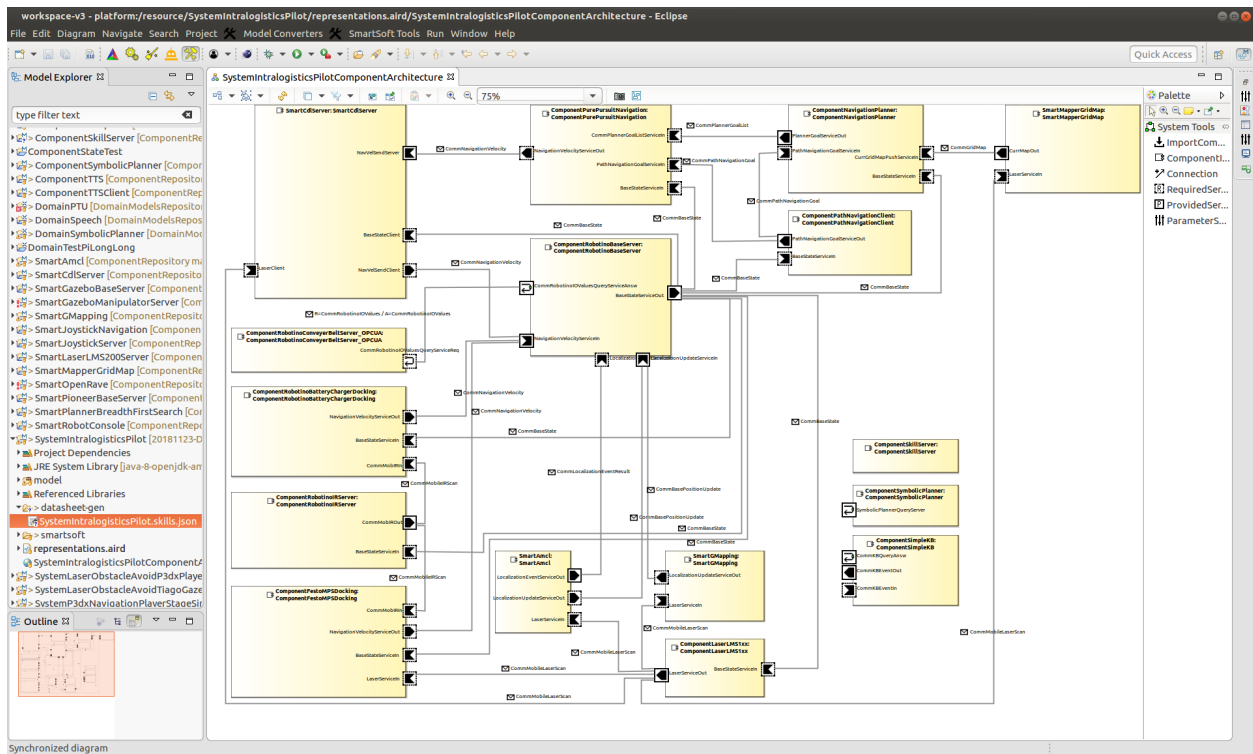
Skills are realized by software components. The below screenshot shows the skill realization as it is created by the **Component Supplier** role in a component project (Tier 3).



System Component Architecture

The figure included below shows a screenshot of the system component architecture diagram of the demonstration. It was modeled using the SmartMDS Toolchain. The system was composed by the system builder from previously developed software components.

The modeled system comes with a digital data sheet (see project explorer on lower left). It lists the skills which come with the components of the system. This datasheet can be imported to Groot for use by the Behavior Developer.



Current State and Roadmap

Both **BehaviorTree.CPP** and **Groot** are currently “feature complete”. These softwares include extensive documentation and unit tests; they have a good level of maturity and can be already used in real-world applications. In terms of community and dissemination, the code is available on Github and has an increasing number of users, which started contributing with bug reports, bug fixes and feature requests.

The additional goals for the rest of the MOOD2Be project (running till Feb. 2019) are:

- Further improve tutorials and documentation, particularly in the context of the **SmartMDS Toolchain**.
- Increase the reliability of the software.
- Further promote the tool in the robotic community.
- Include new features proposed by the users, for instance the ability to include XML files into each other (similarly to header files in C++).

Support of the behavior developer by the SmartMDS Toolchain is ongoing work and available as a prototype which is expected for release soon:

- Modeling of skill definitions for Domain Models / Tier 2: Domain Expert
- Realization of skills in software components / Tier 3: Component Supplier / Behavior Developer
- Digital data sheet of a system containing a “available skills” section: System Builder and Behavior Developer

Discussion

To discuss this demonstration, see <https://discourse.robmosys.eu/t/demonstration-of-behavior-trees-mood2be-smartmdsd-toolchain> [<https://discourse.robmosys.eu/t/demonstration-of-behavior-trees-mood2be-smartmdsd-toolchain>]

Acknowledgements

This technical demonstration has been realized by the MOOD2Be project and the Ulm University of Applied Sciences. The behavior tree engine and the GUI are being developed by the MOOD2be ITP. The SmartMDSD Toolchain and the intralogistics use-case/pilot application are being developed by the Ulm University of Applied Sciences.

MOOD2be is an Integrated Technical Project (ITP) of EU H2020 RobMoSys (robmosys.eu). Ulm University of Applied Sciences is a RobMoSys core partner.

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

See also

Fur further information and to learn more about the concepts used here, see:

- [BehaviorTree.CPP](#)
- [Groot](#)
- [The SmartSoft World and the](#)
- [The SmartMDSD Toolchain](#), the tool that was used to develop the software components and to compose the application.
- [This scenario uses software components from the Intralogistics Industry 4.0 Robot Fleet Pilot](#)
- [Architectural Pattern for Task-Plot Coordination \(Robotic Behaviors\)](#)
- [Task-Level Composition for Robotic Behavior](#)
- [Behavior Developer role](#)
- [Gazebo/TIAGo/SmartSoft Scenario](#)
- [MOOD2be Integrated Technical Project](#) [<https://robmosys.eu/mood2be>]
- [Skills for Robotic Behavior](#)
- [Support of Skills for Robotic Behavior](#)

community:behavior-tree-demo:start · Last modified: 2019/05/20 10:49
<http://www.robmosys.eu/wiki/community:behavior-tree-demo:start>

Using the YARP Framework and the R1 robot with RobMoSys

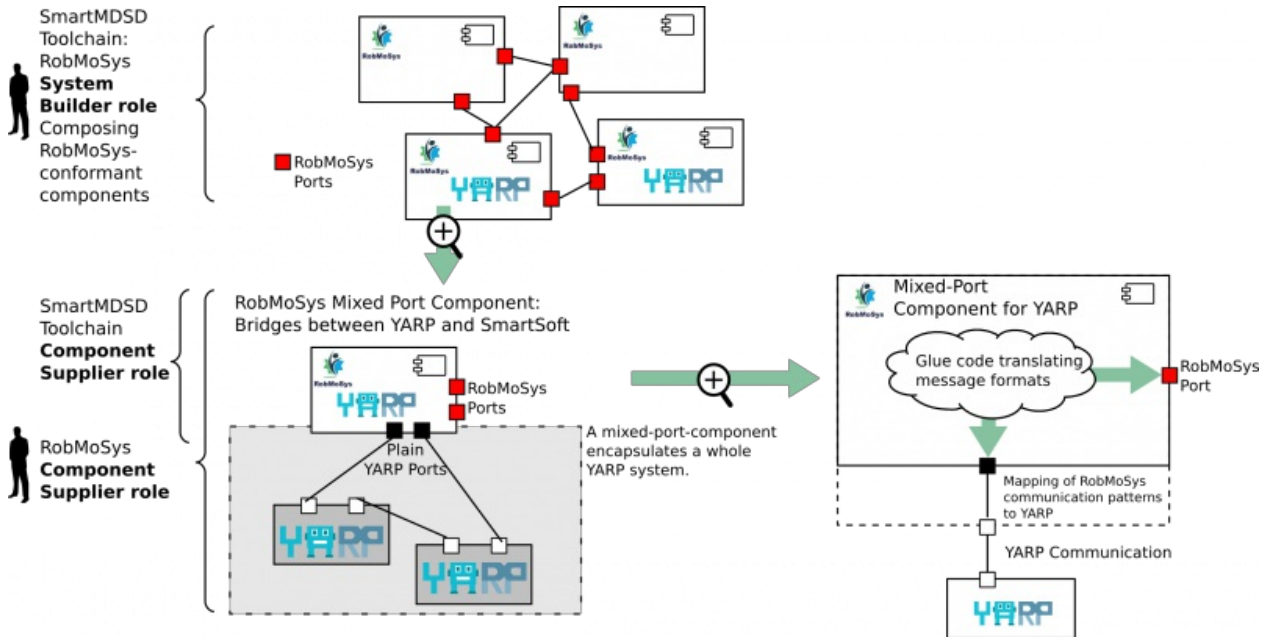
This demonstration shows an example and proof-of-concept how to use software building blocks based on the YARP Framework [<http://www.yarp.it>] via RobMoSys tooling (here with the [SmartMDS Toolchain](#)).



Introduction

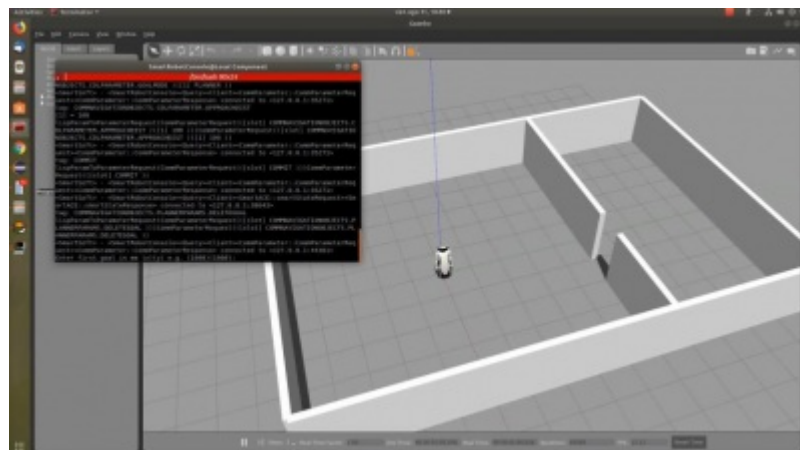
This demonstration is a proof-of-concept, early development result, and test-bed in the effort to develop a general model-driven methodology to enable the use of any robotics Framework with the RobMoSys approach by utilizing a so-called “Mixed-Port Component”. These components have “one leg” in RobMoSys and another leg in the specific world. More precisely, these components bridge RobMoSys with other frameworks. See the

below illustration for YARP:



The specific technical demonstration featured here shows:

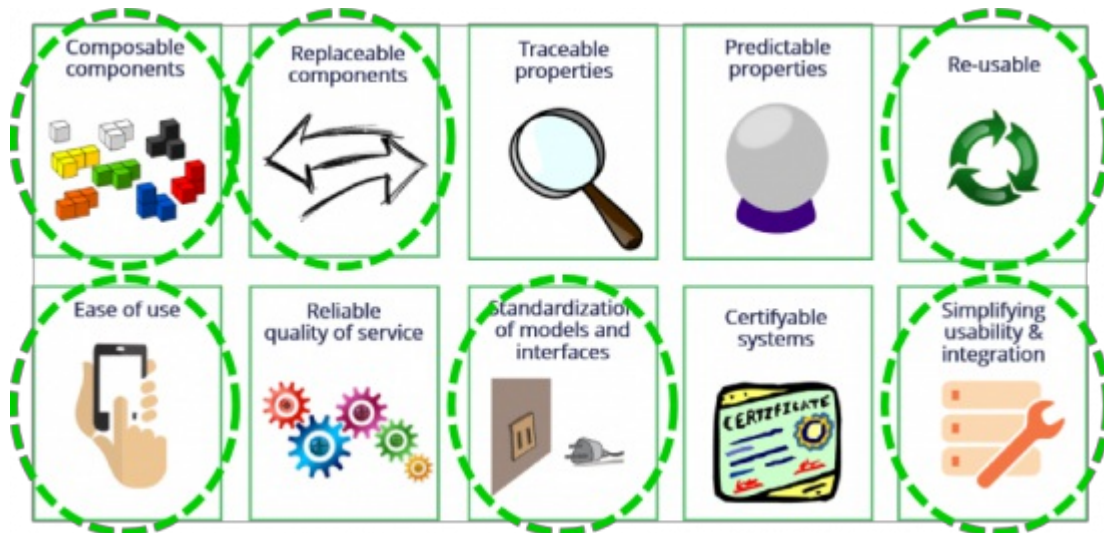
- The R1 robot running with RobMoSys components and YARP components in the Gazebo simulator.
- The example realizes the Flexible Navigation Stack [https://robmosys.eu/wiki/domain_models:navigation-stack:start] using software components from the Intralogistics Industry 4.0 Robot Fleet Pilot with the R1 robot by IIT.
- The scenario is realized using the SmartMDSD Toolchain, a RobMoSys-conformant tooling: The SmartMDSD Toolchain is used to compose software components that a) encapsulate a YARP system via specific mixed-port components and b) connect to native SmartSoft components modeled and generated via the RobMoSys tool SmartMDSD Toolchain. Prior to this step, these mixed-port components have been developed themselves using the RobMoSys approach via the SmartMDSD Toolchain.



The use of YARP and RobMoSys development artifacts in one system illustrates how the structures of RobMoSys can connect two worlds that previously were divided. The immediate benefit of both communities is that they can collaborate and share development efforts more easily. This is ongoing work and further improvements and more native support are planned.

In the Context of RobMoSys User-Stories

In context of the RobMoSys technical user stories [<https://robmosys.eu/user-stories/>], the demonstration shows:



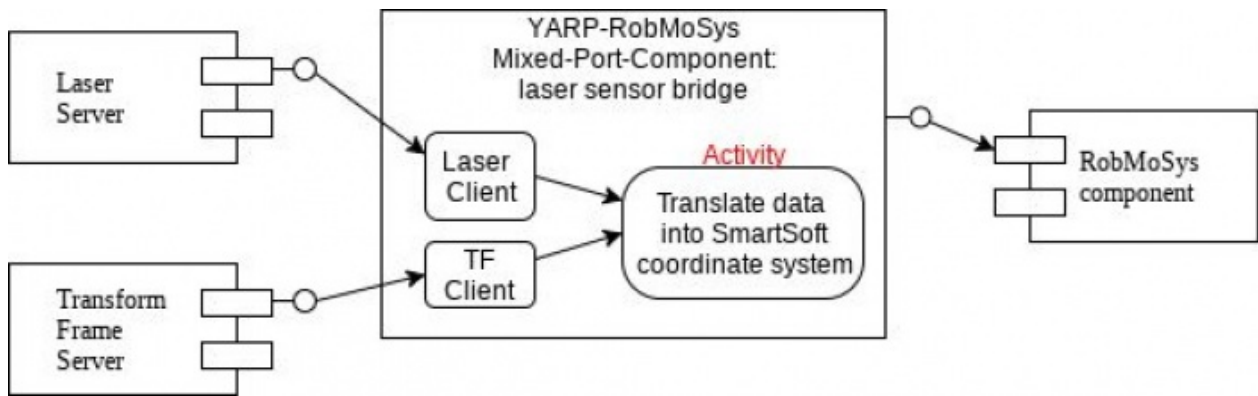
- **Composable components:** use the provided software components and compose the application from scratch.
- **Replaceable components:** Taking the [Gazebo/TIAGo/SmartSoft Scenario](#) as a start and modify it to replace the TIAGo robot with the R1 robot
- **Re-usable:** Use this system as a basis for navigation and build on-top any application that uses a mobile base. This demo shows how to re-use the R1 robot in Gazebo.
- **Standardization of models and interfaces:** it's support via RobMoSys and its benefit is demonstrated by the use of RobMoSys Tier 2 Domain Models: The interfaces (service definitions) of the [Flexible Navigation Stack](#) [https://robmosys.eu/wiki/domain_models:navigation-stack:start] have been modeled and are re-used here. It is an example of how general domain models can be mapped to a variety of concrete applications (R1 robot as described here, [Gazebo/TIAGo/SmartSoft Scenario](#), [Intralogistics Industry 4.0 Robot Fleet Pilot](#), etc.)
- **Ease-of-use and Simplifying usability & integration:** Bridging between the SmartSoft Framework (it is conform to RobMoSys) and the YARP Framework thanks to RobMoSys composition structures. This support is at the moment very basic and will improve in the near future (see section “Current State and Roadmap” below)

Disclaimer: This demonstration is a proof-of-concept of the technical feasibility. Please look at the roadmap to see how it is planned to advance this demonstration and to make it generally accessible via tooling.

Technical Details

To realize the demonstration, several mixed-port components have been developed using the RobMoSys way of component development via the SmartMDSD Toolchain. The mixed-port components are hybrid components (a RobMoSys component that has „one leg“ in the SmartSoft World and „another leg“ in the YARP world) and realize the communication between YARP and components of the SmartSoft Framework.

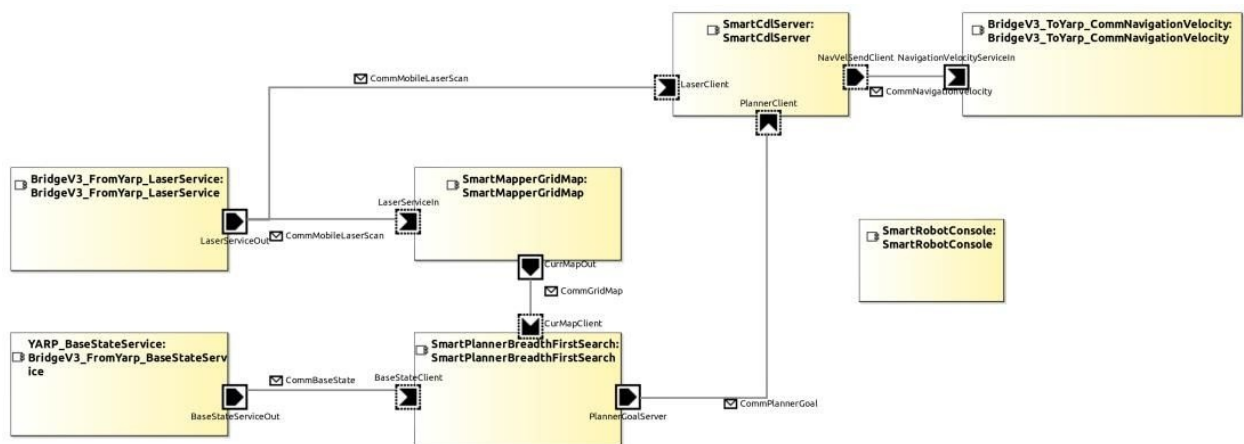
Beneath each mixed-port component lies a whole system of YARP components that become accessible through the RobMoSys services of the mixed-port component. The way these services are designed follows the principle of the [Flexible Navigation Stack](#) [https://robmosys.eu/wiki/domain_models:navigation-stack:start], an example of RobMoSys Tier 2 domain models [https://robmosys.eu/wiki/general_principles:ecosystem:start].



For the moment, the mixed-port components are manually implemented. In general, glue logic can convert between RobMoSys and other frameworks. The effort to do so and the reuse of such a mixed-port component heavily depends on the structures of the target framework. In case of YARP, it was possible to map the RobMoSys communication patterns to YARP. At the moment, the RobMoSys Send, Query and Push patterns have been mapped to YARP already. In some cases this required to extend YARP (i.e. to support asynchronous Query). See the roadmap on how RobMoSys and CARVE plan to extend the native support.

Components in the System

Below figure shows a screenshot of the system component architecture diagram as modeled in the SmartMDSD Toolchain (see [GitHub Repository \[https://github.com/CARVE-ROBMOSYS\]](https://github.com/CARVE-ROBMOSYS) for SmartMDSD Toolchain project).



The scenario features the following software components:

- **SmartMapperGridMap** This component receives a current laser-scan and accumulates the information from this scan into a locally maintained grid-map.
- **SmartPlannerBreadthFirstSearch** This component takes a current grid-map and the current destination location as input and calculates a path (consisting of intermediate way-points) to reach that destination

SmartCdlServer: This component implements an obstacle-avoidance algorithm, such as e.g. the Curvature Distance Lookup (CDL) 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.

- **SmartRobotConsole** A very simple if-then-else sequencer component to coordinate all components via the software component's coordination interface
- **BridgeV3_FromYarp_BaseStateService** accessing R1 odometry from YARP via RobMoSys domain

models

- **BridgeV3_FromYarp_LaserService** accessing YARP laser ranger components via RobMoSys domain models
- **BridgeV3_ToYarp_CommNavigationVelocity** accessing the R1 robot base via RobMoSys domain models of the Flexible Navigation Stack [https://robmosys.eu/wiki/domain_models:navigation-stack:start] in the Gazebo simulator

Further Information

Fur further information, see also the following RobMoSys wiki resources to learn more about the RobMoSys concepts used here:

- Component Supplier Role [https://robmosys.eu/wiki/general_principles:ecosystem:roles:component_supplier]
- Component-Definition Metamodel [<https://robmosys.eu/wiki/modeling:metamodels:component>] (software component (meta-)model)
- Architectural Pattern for Communication [https://robmosys.eu/wiki/general_principles:architectural_patterns:communication] and Communication patterns [<https://robmosys.eu/wiki/modeling:metamodels:commpattern>]
- Flexible Navigation Stack [https://robmosys.eu/wiki/domain_models:navigation-stack:start]
- Gazebo/TIAGo/SmartSoft Scenario
- The SmartSoft World and the The SmartMDSD Toolchain
- This scenario uses software components in context of the Intralogistics Industry 4.0 Robot Fleet Pilot

Current State and Roadmap

As of September 2018, you can:

- Use the R1 robot driven by the YARP framework from RobMoSys via the provided mixed-port components in the SmartMDSD Toolchain
- Build your own bridges to the YARP framework by following the example components provided

Please note that this demonstration is work in progress. The following future work is is under preparation:

- The general concept of “Mixed-Port-Components” to bridge between RobMoSys and any world.

Documentation how to reproduce the here described example in order to use the R1 robot in Gazebo via RobMoSys methodology.

- Integrating RobMoSys and YARP Framework more natively according to the RobMoSys composition structures. A focus will be set on YARP conforming to the RobMoSys communication patterns. This will heavily reduce the manual effort in mixed-port components. Mixed-Port-Concept comes with adequate model-driven tooling support to make the benefits accessible by users.
- For this purpose, the RobMoSys Communication Patterns Send, Query, and Push patterns have been mapped to YARP already. More mappings are to follow.

Videos

This list summarizes the demos which show the work done in the ITP CARVE:

Video demonstrating YARP-SmartSoft integration: https://youtu.be/hyz7RK1_XsU
[https://youtu.be/hyz7RK1_XsU]

Video demonstrating static analysis of a correct BT with NuSMV: <https://youtu.be/N0Utz-C2HwU>
[<https://youtu.be/N0Utz-C2HwU>]

Video demonstrating static analysis of an incorrect BT with NuSMV: https://youtu.be/v_fSNNppIE8
[https://youtu.be/v_fSNNppIE8]

Videos demonstrating scenario 1 execution and runtime monitors for scenario 1 and scenario 2 in simulation:
<https://youtu.be/QII2lXeIXeM> [<https://youtu.be/QII2lXeIXeM>], <https://youtu.be/iKbhblOxxrw>
[<https://youtu.be/iKbhblOxxrw>]

Videos demonstrating scenario 1-3 on the real robot:

- Scenario 1: <https://youtu.be/b7TeRX1uzoc> [<https://youtu.be/b7TeRX1uzoc>]
- Scenario 2: <https://youtu.be/q25uvU443Tg> [<https://youtu.be/q25uvU443Tg>]
- Scenario 3: <https://youtu.be/-wE457T0718> [<https://youtu.be/-wE457T0718>]

Runtime monitors detect when a skill becomes irresponsive (simulation): <https://youtu.be/QXL4qzp6Qsk>
[<https://youtu.be/QXL4qzp6Qsk>]

Runtime monitors detect unexpected behavior in the environment (real robot): https://youtu.be/DzF2GC_Ib3U
[https://youtu.be/DzF2GC_Ib3U]

Discussion

To discuss this demonstration, join the discussion at Discourse [<https://discourse.robmosys.eu/t/first-yarp-smartsoft-integration-carve/131>]

Acknowledgements

This technical demonstration has been realized by the CARVE project. The methodology to generalize the approach and integrate YARP access with the RobMoSys structures is a joint effort of the CARVE project and the Ulm University of Applied Sciences. The general methodology behind “Mixed Port Components” is driven by Ulm University of Applied Sciences.

CARVE is an Integrated Technical Project (ITP) of EU H2020 RobMoSys (robmosys.eu). Ulm University of Applied Sciences is a RobMoSys core partner.

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

Picture of the R1 robot by D.Farina-A.Abrusci © 2016 IIT.

See Also

- CARVE Integrated Technical Project [<https://robmosys.eu/carve>]
- The CARVE Github repository that contains the code presented here[<https://github.com/CARVE-ROBMOSYS/Yarp-SmartSoft-Integration>]
- YARP Framework [<http://www.yarp.it>]
- R1 Robot [<https://www.youtube.com/watch?v=aninEl1GVns>]
- Source code for the R1 robot[<https://github.com/robotology/cer>]
- Robot model for gazebo [<https://github.com/robotology/cer-sim>]
- The SmartMDS Toolchain, the tool that was used to develop the software components and to compose the application

Benchmarking in the RobMoSys Ecosystem

What is benchmarking?

Benchmarking is increasingly important to autonomous robotics. To go out of the laboratories and become real products, robots need **benchmarks**: standardised, objective ways to characterise, measure and compare their performance in a modular and composable way.

Users -from system integrators to consumers- need objective evaluation of components to **choose** products that meet their needs; industry needs composable benchmarks to **predict** the performance of component-based solutions at design time; research needs benchmarks to **compare** novel approaches with established references.

What is Plug&Bench ?



Plug&Bench logo

Plug&Bench expands the RobMoSys Ecosystem with new elements for the definition of standardized and easy-to use **performance benchmarks**. Plug&Bench's models let experimenters define, implement and run benchmark by building on a formalised framework. This eases their work, avoids *ad hoc* solutions and opens the way to a modular and composable evaluation of systems.

How to benchmark an autonomous robot?

Execution of a proper benchmark is a scientific experiment: a **benchmarking experiment**. The *reproducibility* and *repeatability* of the benchmark make different executions (possibly by different people, at different times, on different systems) comparable. Shared metamodels and models are important to achieve this result.

The methodological foundations of Plug&Bench are described in [this document](#)

[https://www.dropbox.com/s/kirlrvb90pxgs8e/foundations_benchmark.pdf?dl=0]. They inherit from a successful line of European projects about robot benchmarking comprising *RAWSEEDS* (FP6), *RoCKIn* (FP7), *RockEU2* (H2020) and *SciRoc* (H2020).

What does Plug&Bench add to RobMoSys?

The **Plug&Bench Benchmark Metamodel** (Figure 1) is a new element in the set of [RobMoSys Composition Structures](#). It defines the *Benchmarking Component*, i.e. an extension of the [Component-Definition Metamodel](#) providing all the elements needed to describe and execute a benchmark in a standardised manner.

You can download the Plug&Bench Benchmark Metamodel from [this repository](#)

[https://www.dropbox.com/sh/ql42y28qdvhzxdj/AACVLxpjP58DNjTrMxliguK_a?dl=0]. This note

[https://www.dropbox.com/sh/ql42y28qdvhzxdj/AACVLxpjP58DNjTrMxliguK_a?dl=0] explains its contents, while the accompanying document

[https://www.dropbox.com/sh/ql42y28qdvhzxdj/AAD3QuXvIbmYfqHdfjf_de7fa/documentation_metamodel.pdf?dl=0] describes the metamodel.

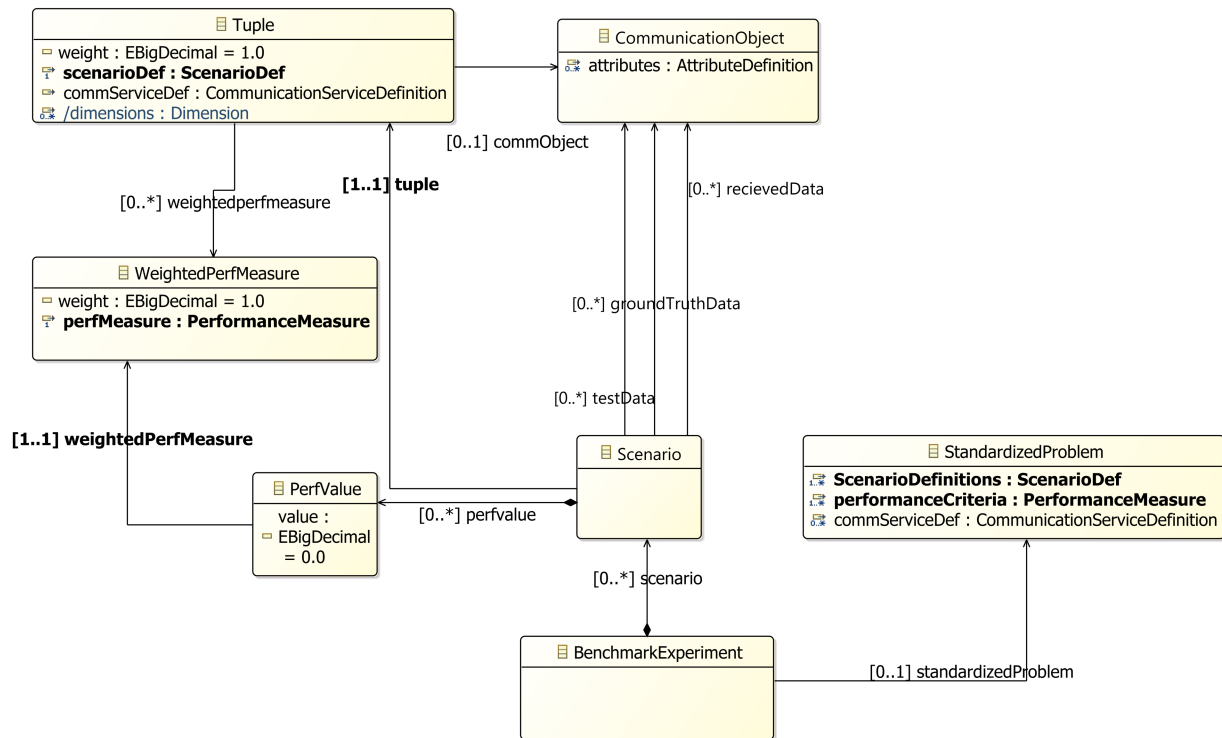


Figure 1: ecore class diagram of the Plug&Bench benchmark metamodel

Plug&Bench also adds a new element to the set of RobMoSys Roles in the Ecosystem: the **Benchmark Developer**. A description of all the connections between Plug&Bench and the RobMoSys Ecosystem is provided by a specific document [https://www.dropbox.com/s/ymgqld896xocj6p/benchmark_in_robmosys.pdf?dl=0].

Plug&Bench provides a **Benchmark Engineering Tool** [<https://github.com/Mohamedgalil/plug-and-bench>] supporting the Benchmark Developer in building *benchmark models* based on the Plug&Bench Benchmark Metamodel.

A Virtual Machine with all prerequisites already installed for the integration with SmartMDSD is also available on <https://owncloud.fraunhofer.de/index.php/s/N9PbIHWzYKMBcCj> [<https://owncloud.fraunhofer.de/index.php/s/N9PbIHWzYKMBcCj>].

Are there any example benchmark models?

Yes. By using the **Benchmark Engineering Tool** [<https://github.com/Mohamedgalil/plug-and-bench>], three benchmark models have been defined:

- *Screw-hole Localizer* benchmark
- *Trajectory Planner* benchmark
- *Trajectory Follower* benchmark (see Figure 2)

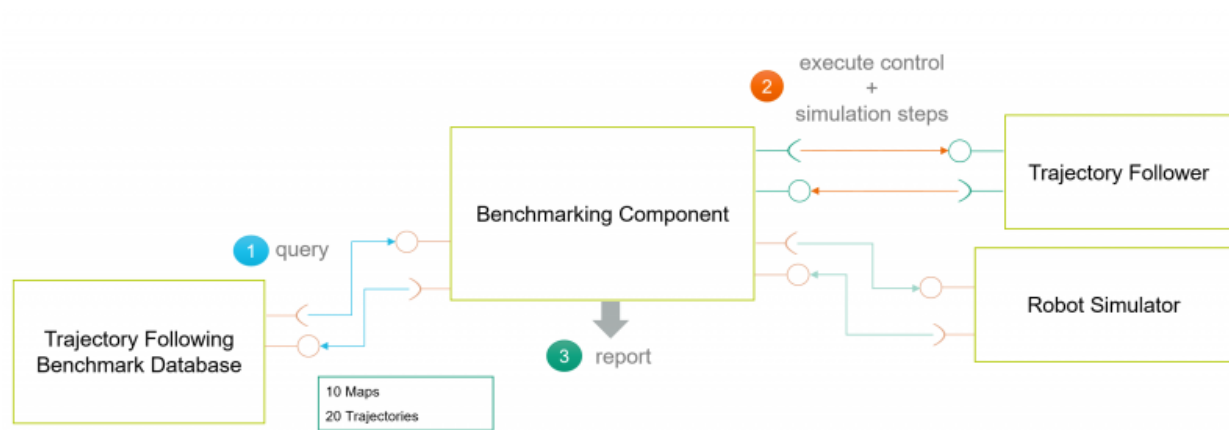


Figure 2: structure of the Trajectory Follower benchmark

Plug&Bench benchmark models can be downloaded from [this repository](https://www.dropbox.com/sh/4gx7vb0eimg517j/AAAM0qVQfyFOVPoycUaODxqla?dl=0)
[\[https://www.dropbox.com/sh/4gx7vb0eimg517j/AAAM0qVQfyFOVPoycUaODxqla?dl=0\]](https://www.dropbox.com/sh/4gx7vb0eimg517j/AAAM0qVQfyFOVPoycUaODxqla?dl=0).

Acknowledgements

Plug&Bench is an Integrated Technical Project (ITP) of EU H2020 project RobMoSys [robmosys.eu](http://www.robmosys.eu)
[\[http://www.robmosys.eu/\]](http://www.robmosys.eu). The Consortium of Plug&Bench comprises Politecnico di Milano
<https://www.polimi.it/en/> [\[https://www.polimi.it/en/\]](https://www.polimi.it/en/) and Fraunhofer IPA <https://www.ipa.fraunhofer.de/en.html>
[\[https://www.polimi.it/en/\]](https://www.polimi.it/en/).

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

community:benchmarking:start · Last modified: 2019/05/20 10:49
<http://www.robmosys.eu/wiki/community:benchmarking:start>

Guaranteed Stability of Networked Control Systems

This is a contribution of the EG-IPC (Extension of Intrinsically Passive Control model and integration in the RobMoSys ecosystem“) integrated technical project (ITP) of RobMoSys.

Overview

At its core, the Energy-Guarded Intrinsically Passive Control (EG-IPC) ITP has developed a set of components that allows a user to create networked control systems that are guaranteed to be stable. The primary use case is teleoperation of robots with force feedback, but as will become apparent later, the energy-based approach can be applied to many other different situations. In fact, any setup where you want two systems in different places to act as if they were physically connected is a candidate for the EG-IPC approach.

At the core of the approach is an energy modeling and control viewpoint: all systems and controllers are analysed and controlled in terms of their energy production and consumption. By assuring that the energy exerted by the system as a whole does not exceed (a constant multiple of) the amount of energy inserted, the system can be analytically and practically stable and safe to interact with. We have made these techniques more accessible and interoperable by embedding them in the RobMoSys approach, creating meta-models and models to design and analyse energy-based control systems.

To achieve these results, we started by developing a metamodel (“design language”) of the bond-graph notation, which is a natural and versatile modeling language to describe multi-domain physical systems [1]. We import the standard interfaces defined in the bond-graph metamodel required to generate components suitable for the EG-IPC approach. Building from the formalization of the bond-graph entities, we developed a more pragmatic metamodel for describing, designing and analysing EG-IPC systems. Finally, we put these metamodels to the test by implementing their elements in the SmartMDS Toolchain¹⁾ and building a concrete teleoperation use case with them.

This page is outlined as follows: we begin by presenting the motivation for this ITP, followed by the objectives and its role in the RobMoSys ecosystem. We continue by diving into the developed metamodels. Finally, we provide more details on the applications with a haptic teleoperation use case.

Why bother?

A long-standing and well known problem in teleoperation concerns the stability of haptic force-feedback systems [2]. When naïvely coupling the position commands of a master device and the force readings of a slave device, the smallest delay in the communication channel will create uncontrollable oscillations. This brings critical problems to distributed control systems like the bilateral teleoperation setup represented in Figure 1.

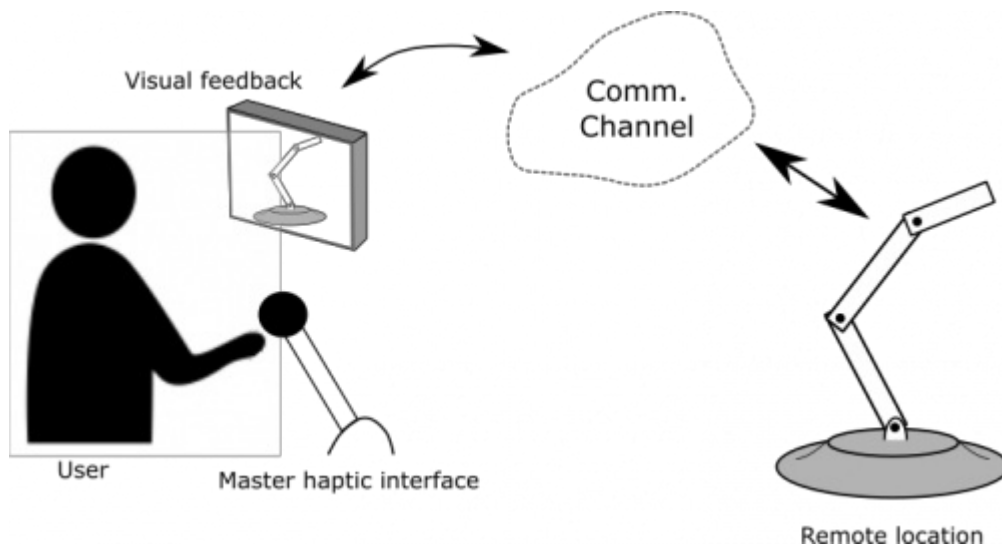


Fig.1 Robot teleoperation with haptic feedback.

A common mitigation technique is the approach of passivity: when all components in a distributed system have the property of being passive - that is, that more energy cannot be extracted from a system than the one already stored or inputted into it. Thus, the system as a whole is stable. The problem with the aforementioned teleoperation setup is that undesired extra energy can be created by the quantization and delays [3], [4], leading to active behaviour that breaks passivity and hence compromising stability.

The EG-IPC ITP envisions all-time stable, loop control composable components using ‘energy guards’ to preserve passivity of the distributed system. We built upon the existing approach to guarantee passivity of components by wrapping them in a Passivity Layer (PL). Broadly speaking, this layer acts as a wrapper that monitors the energy exchange among the components inside, and limits the output when the energy balance is violated.

This energy-based approach is a somewhat novel paradigm, so it can be difficult to get started with. However, there are two main benefits that are not easily attained with other approaches:

- Energy exchange is intuitive, also for non-experts. Systems built from an energy-based viewpoint are mathematically stable, but at the same time can be reasoned about without advanced mathematics. Moreover, many other system failures or bugs can be detected using energy analysis: if a part of the system is not functional, it will typically use much more or less energy than expected. The same goes for unexpected or dangerous behaviour.

- Energy exchange is universal. Many different physical systems (mechanical, electrical, hydraulic, chemical) can be connected with (virtual) energetic bonds, as long as there is an energy-based controller available. The domain experts can create these controllers, and system integrators and architects can use a common design language to apply these components in their products.

Energy-based control in more detail

Energy-based control is a model-driven loop control strategy for multi-domain physical systems, such as robotic applications. This method relies on the exchange of energy among components to perform a specific controlled action. An energy-based controller is modeled as a dynamic system that exchanges energy with the plant [5]. This allows a robot to interact with the environment by shaping the energetic behaviour to take a desired form. Such an idea leads to formulate Intrinsically Passive Controllers (IPC), which are control components that exchange power while preserving passivity. The Figure 2 is a representation of an IPC controlling a robotic arm; the IPC is modeled as a set spring-damper elements that exchanges energy with the robot.

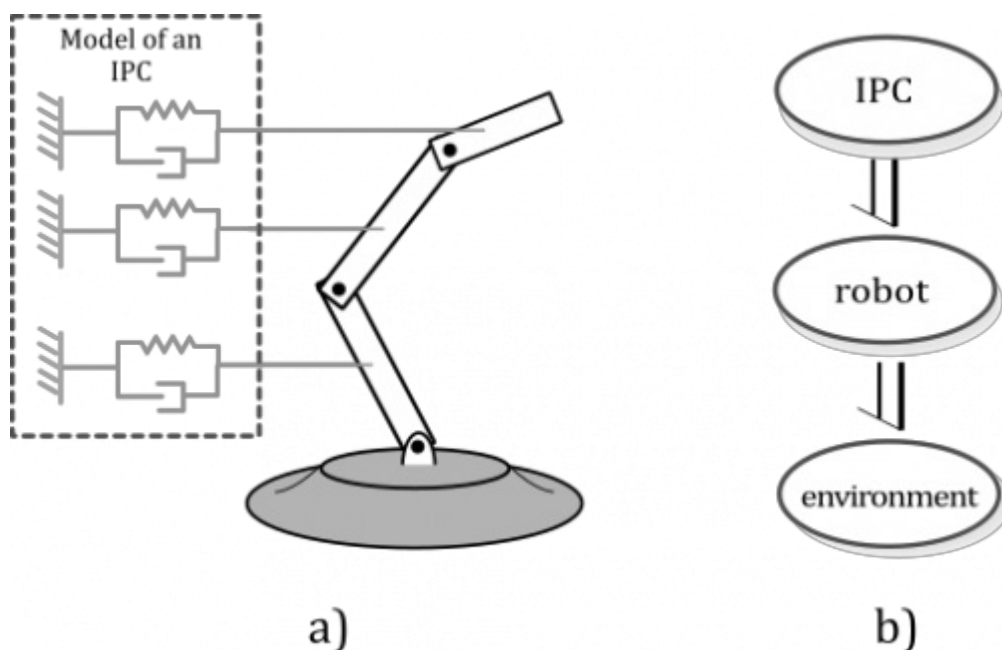


Fig.2 Representation of a robotic application under the energy-based control approach

What is an ‘Energy-Guarded IPC’?

An Energy-Guard (EG) is an arrangement of functional blocks known as Passivity Layers (PLs) that guarantee stability of energy-based components when dealing with computational and communication delays. Figure 3 illustrates an energy-based software component (red oval) inside an arrangement of PLs (blue oval). A PL is placed on each energy-exchanging port of the ‘guarded’ component to guarantee passivity on every interaction. This arrangement is known as Energy-Guarded Component (EG-Comp). EG-IPC stands for Energy-Guarded Intrinsically Passive Controller.

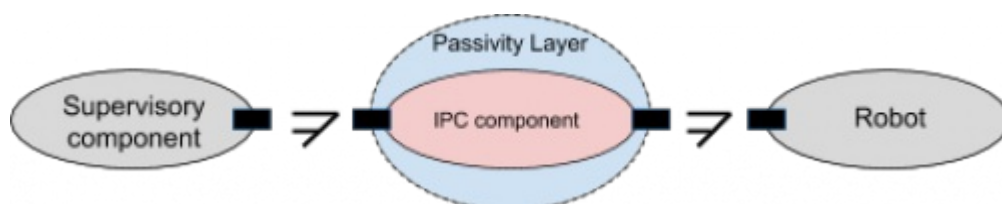


Fig.3 Basic architecture of an Energy-Guarded IPC component

Impact of EG-IPC

When passivity is violated, the stability of the system is not guaranteed, and often indeed violated in practice. The EG-IPC blocks will guarantee a basic safety level due to the inherent passive structure of the block. As such, the potential user group consist of all system architects and system builders of complex robotic applications being, according to RobMoSys, the main part of the robotic applications that will be developed in the coming years. The diagram in Figure 4 illustrates the role of the EG-IPC in the RobMoSys ecosystem.

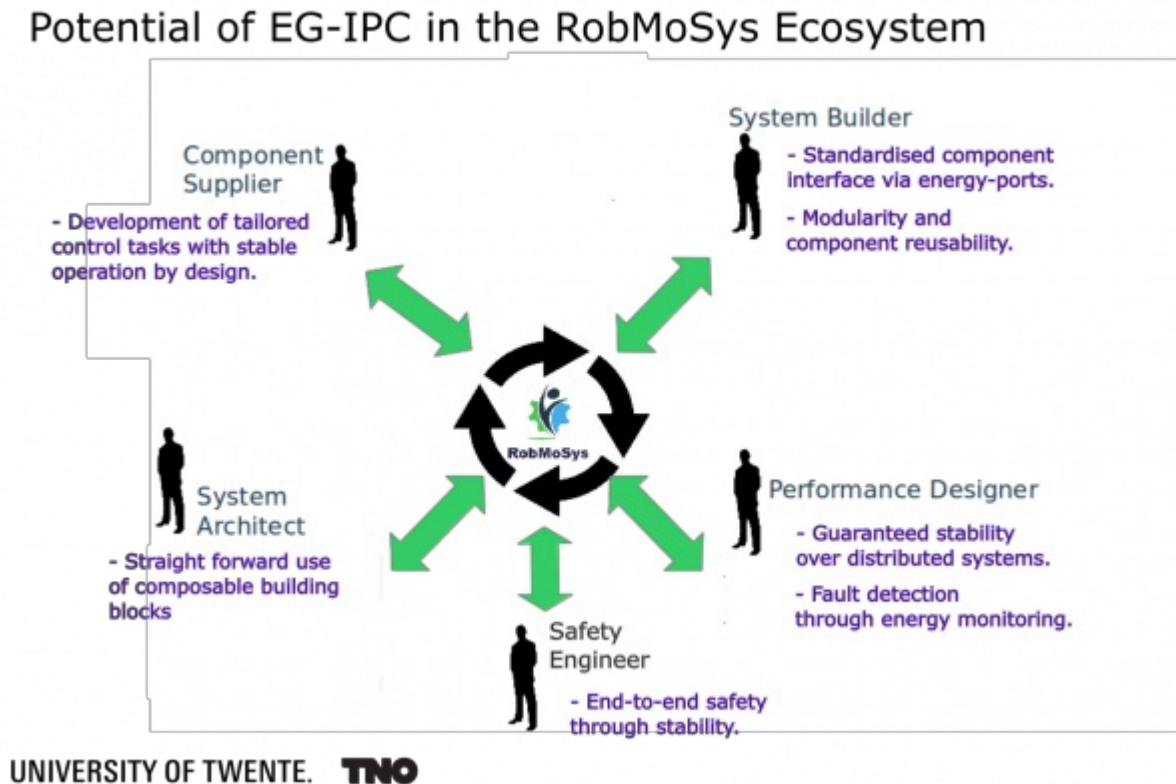


Fig. 4: EG-IPC in the RobMoSys ecosystem

From a technical perspective, the benefits of the EG-IPC component will lead to:

- **Composable components:** Due to the usage of a standardized interface known as power-port, any energy-based component is guaranteed to exchange power, contributing to the composability of the system. The EG-IPC follows this architectural pattern.
- **Predictable properties:** The monitoring of energy provides an insight about the system behaviour. Abnormalities such as active behaviour and other faults can be predicted.
- **Replaceable components:** As the EG-Component architecture is composable, the 'guarded' component can always be replaced - i.e. by a communication channel or another controller. This benefit is only possible when the components have the standard power-ports interface.
- **Re-usable:** Given the composability of the EG-Component architecture, any EG-IPC component can be re-used in other applications.

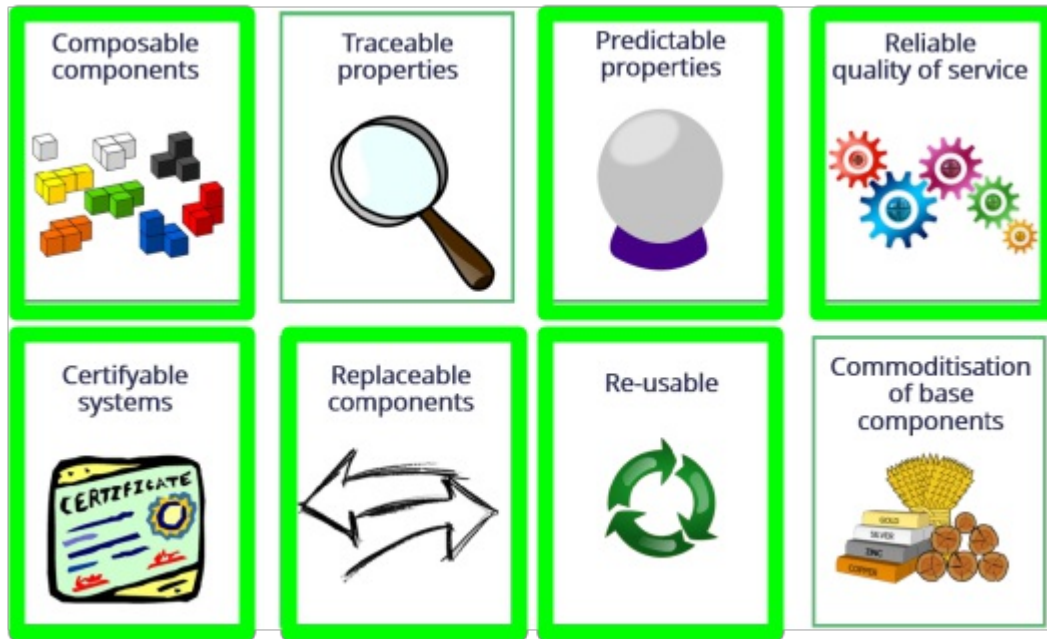


Fig. 5: Expected technical user stories for EG-IPC

Composition of the EG-IPC

Following the model-driven approach of RobMoSys, we developed metamodels to define the structure of the EG-IPC blocks. This entails formalisation of the generic IPC structure whereby adding energy guards on the interfaces of the components where needed and additional interfaces to communicate and synchronise the energetic state.

Required metamodels

The EG-IPC project presents a formalized metamodel for the bond-graph language that captures the features of the energy-based modeling and control, which are critical to describe the power exchange between components under the energy-guarded component architecture. The bond-graph entities are represented in Figure 6. The formal definitions can be found in [6].

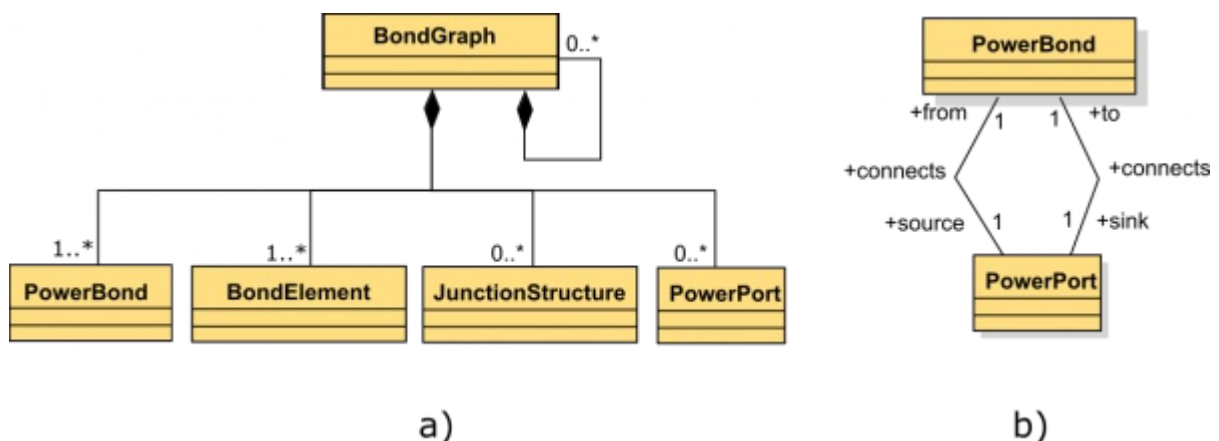


Fig.6: Structure of the bond-graph metamodel.

As an IPC is a controller which behaves as a physical system, its metamodel conforms to bond-graph language and feedback control as shown in Figure 7. By definition of the passivity property, an IPC has a constraint that indicates that the energy extracted from the system cannot be greater than the energy introduced to the system.

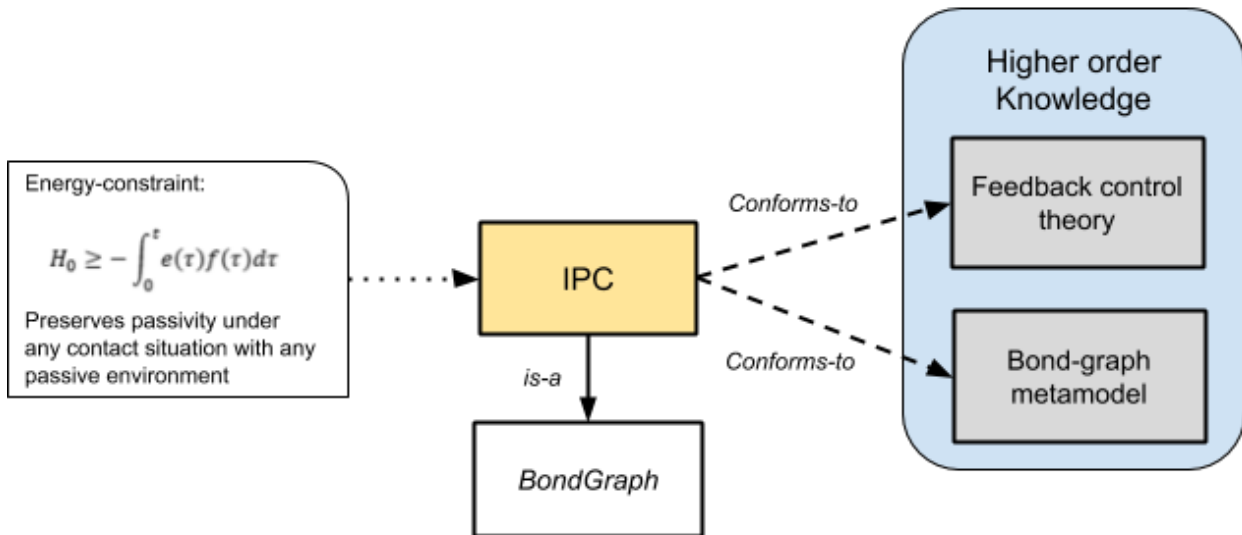


Fig.7.- IPC metamodel structure.

The EG-IPC metamodel, shown in Fig.8, makes use of the bond-graph and IPC metamodels to simplify the definition of its entities. The Passivity Layer component monitors the energy interacting with the system, guaranteeing the passivity and stability by bounding the extracted energy.

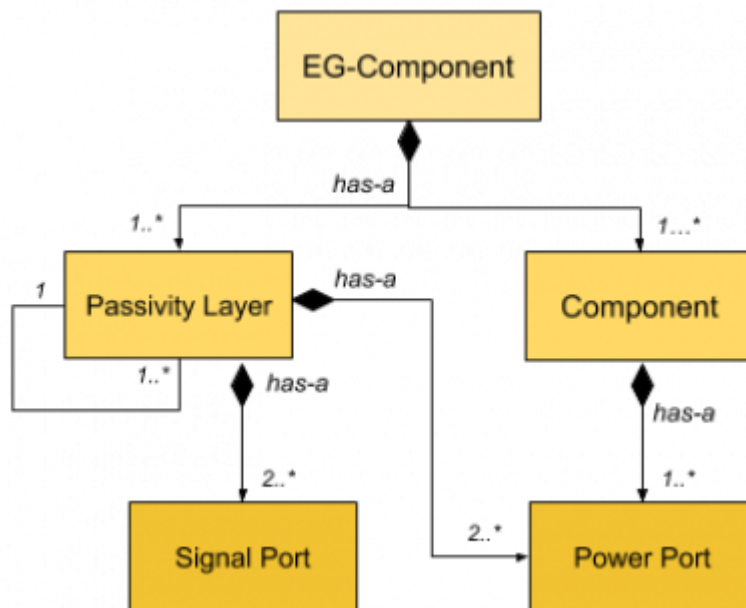


Fig.8: Structure of the EG-IPC metamodel.

The generated metamodels are aligned to the RobMoSys modeling approach as they will be able to generate models suitable for:

- Human-software documentation: In form of datasheet of the (EG-)IPC components.
- Software tooling and standardization: By using the power port and power bond entities to connect components and using power as a standard interaction unit.
- (Future) verification and validation of models.

Component architecture

An EG-Component⁽²⁾ is any component inside a PL structure. If such component is an IPC, the EG-Component architecture will guarantee its stability when dealing with computational and communication delays. The basic

operation tasks are:

- Passivity preservation
- Energy monitoring
- Fault detection

An example of an ‘EG-Component’ is shown in Figure 9. An IPC component is wrapped by two Passivity Layer. The ‘guarded’ IPC component interacts with the Passivity Layers via power ports. The example in Figure 9 follows the sign convention of the arrow representing the power bond. As described by the bond-graph metamodel, the power is exchanged in both directions of the bond.

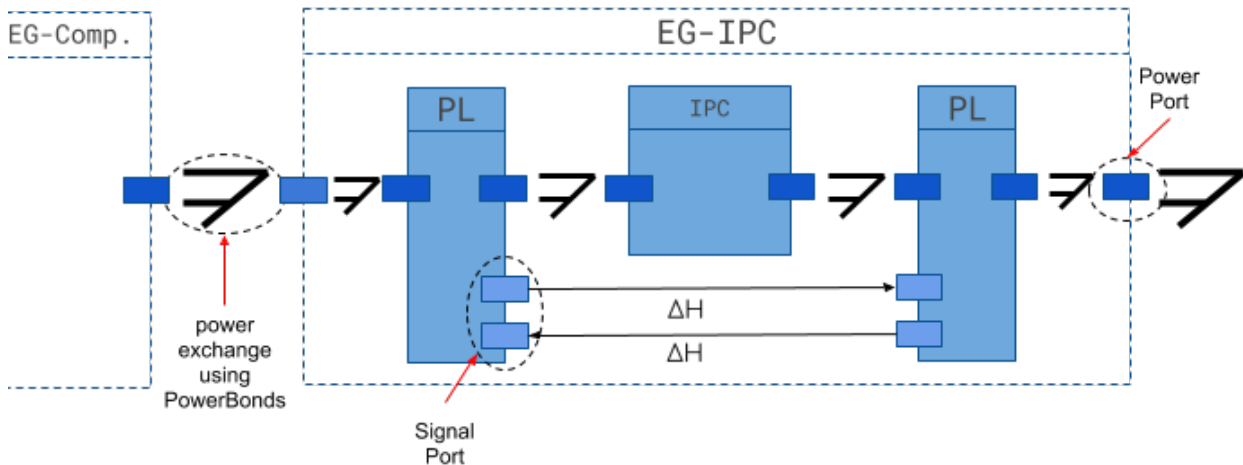


Fig.9.- Basic architecture of an energy-guarded component.

Use Cases

Haptic Teleoperation: Manipulation Use Case

A setup was created to test the functionalities of EG-IPC in a teleoperation environment that involved the operator receiving haptic feedback over network communication. This setup is displayed in Fig 10. Similar to the example in Fig 1, it involves a Franka Emika Panda robotic arm that serves as a haptic device (right robot in left picture), a video feedback system and a remote Franka Emika Panda robot that should perform a remote task. Between the two robots, an ethernet network with a possibility to increase communication delays was installed. The main task objective of this setup was to remotely open and close a door.



Fig. 10 - The haptic teleoperation use case setup

A block scheme of the teleoperation architecture that is implemented can be seen in Fig 11. The two robotic arms are connected through an impedance controller and communication channel. Surrounding these

components are energy guards to ensure EG-IPC. Explanation of used symbols can be found in Table 1. The architecture is implemented using ROS. Each block indicates a ROS node, communicating the indicated variables over ROS topics. Although the adherence to the metamodel was not enforced by tooling, the implementation still conforms to the different components and their interfaces (Fig 12).

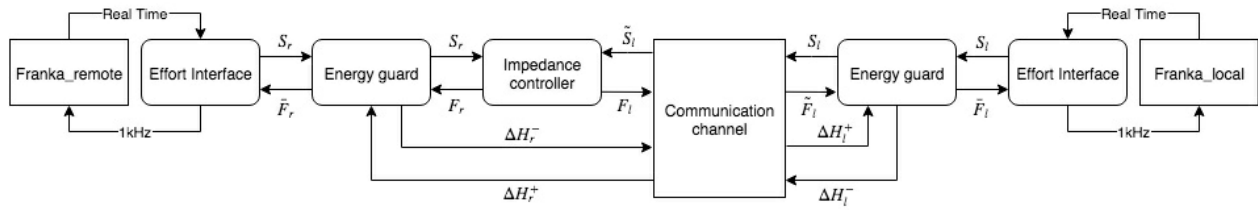


Fig. 11 - Block scheme of the haptic teleoperation use case setup

Symbol	Definition
\subscript_l	Subscript for local robot (operator side)
\subscript_r	Subscript for remote robot (remote task side)
S	Cartesian robot state, includes:
X	Current end effector pose (position + orientation as quaternion)
V	Current end effector twist (linear + angular velocity)
F_{cmd}	Last applied end effector wrench (forces + torques)
F	Desired end effector wrench determined by the impedance controller
\bar{F}	Desired end effector wrench saturated by the energy guard
$\tilde{}$	Information affected (delayed) by the communication channel
ΔH	Energy packet, + indicates addition, - indicates subtraction. Energy subtracted at the one side turns into (delayed) addition on the other side and vice versa.

Tab. 1 - Definition of the symbols used in Fig 11

Franka Emika Panda robotic arms are used, which are controlled in real time through their control cabinets, the Franka Controller Interface (FCI). The local robot (Franka_local) is held by the operator and services as haptic device. The remote robot (Franka_remote) is the teleoperated robot that is located at the remote site. The effort interfaces are written for the Franka robots specifically. They communicate with the FCI's in a monitored 1kHz loop. If this loop is broken, the robot will apply its safety brakes. In this loop, the effort interface writes torque commands to the actuators and receives information on the robot's state in real time.

The impedance controller is the heart of the teleoperation controller, functioning as a virtual stiffness. It multiplies the difference between the robot poses by a tunable stiffness, which gives the desired impedance control. These control actions put in the effort to match the end effector poses of the robots.

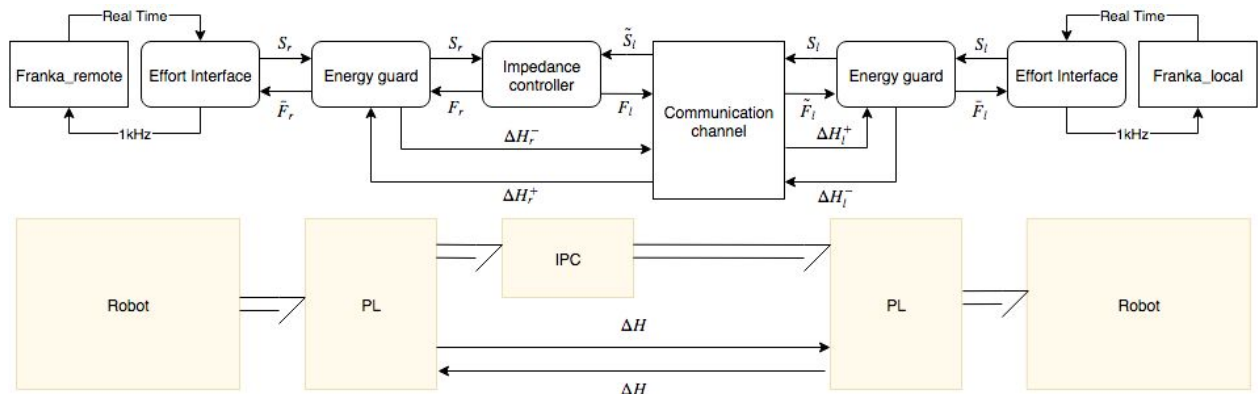


Fig. 12 - Correspondence between implementation and metamodel

Because passivity (and therefore stability) of the impedance controller combined with the communication channel cannot be ensured by default, energy guards surround this subset of components to guarantee stable

teleoperation control. It monitors these amount of available energy in virtual energy tanks using the EG-IPC conventions. Since these components are well-defined and separated by generic interfaces, they are reusable and composable: the impedance controller is not aware what type of system it controls, and only performs calculations on positions and velocities. The passivity layer does not need specifics of the robot model to still estimate the (kinetic) energy that is spent and needed, or characteristics of the communication channel to modify its behaviour. Still, there is a role for the system architect to pick the right composition and tuning of parameters for the system to work optimally.

Integration into RobMoSys Tooling: Navigation Use Case

In order to test our adherence to the RobMoSys ecosystem, we tried to recreate our metamodels using the SmartMDSD Toolchain, which is an IDE for robotics software conformant to the RobMoSys approach. In order to make use of existing modelling and simulation components provided by RobMoSys, we worked on a use case relating to navigation: platooning of small wheeled robotic platforms like the P3Dx or the Tiago.

Platooning

In Fig. 13 the simplest platooning use case is shown: two robots are controlled via virtual spring-dampers (impedance control).

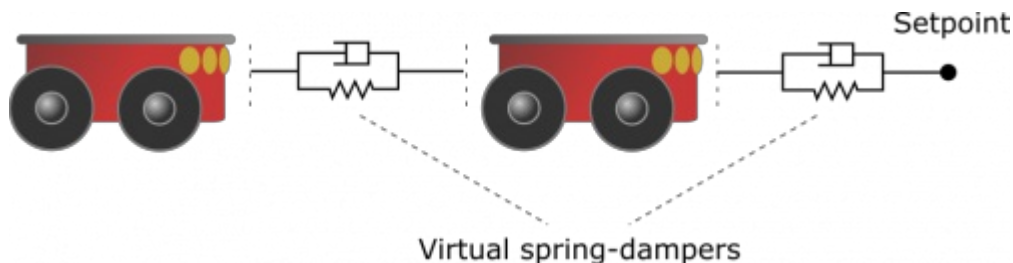


Fig. 13 - A simple platooning application of IPC. Robotic platforms are following each other with virtual spring-damper dynamics.

The leader robot is following a setpoint (provided by the user or a navigation algorithm). The follower follows a setpoint at a fixed distance behind the leader robot. Since it is reasonable to assume that both robots perform their control locally, and are connected by a wireless link that will not be perfect, energy guards are in place here. With the right tuning, the following will be smooth, non-oscillatory and stable. When the leader robot halts or the connection is lost, the follower robot will slow down in its approach to the latest known position. When the follower gets stuck, the leader will slow down too, while still being ‘dragged’ towards the setpoint. By monitoring the virtual forces on the setpoint, this situation can be recognized. Of course the setpoint can also be a robot that is directly remote-controlled.

In order for this to work, the only prerequisites are that the robots can be force-controlled and that the distance between two robots is known. To be force-controlled, the robot controller must be able to translate a force into an acceleration of the robot, but it is free to keep the acceleration virtual and give e.g. velocity commands. In addition, the controller must have a way to estimate the energy spent. Knowing the distance between two robots (and the rate of change of the distance) can be derived from individual positions, or from direct distance measurements.

Once this system is in place, it could be used to quickly compose a system of many such robots forming one platoon:

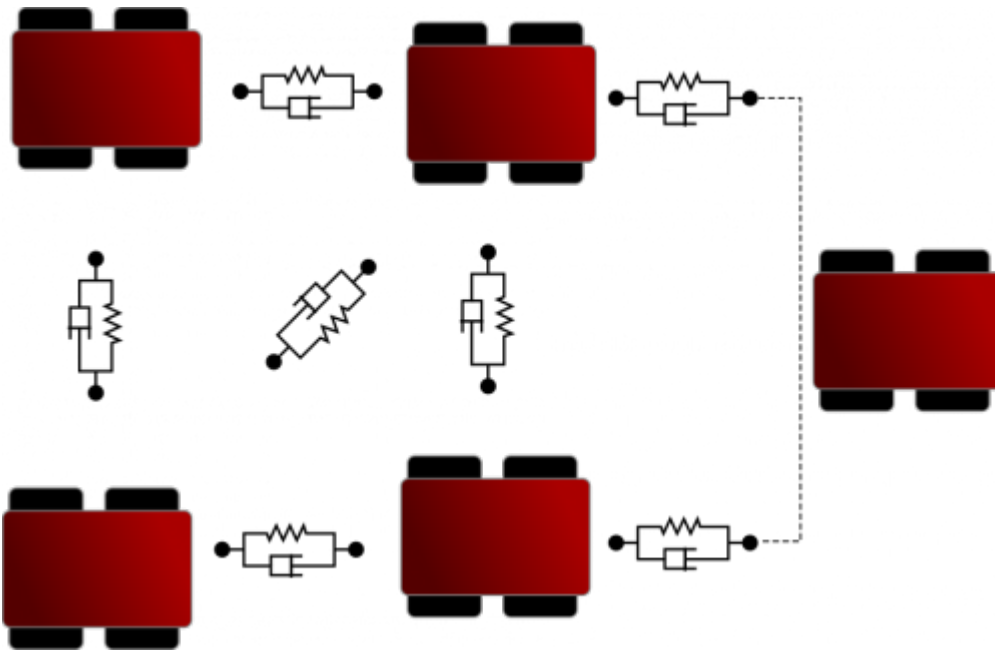


Fig. 14: Extendable platooning.

Implementation with the SmartMDSD Toolchain

A start was made with implementing our metamodels in the SmartMDSD Toolchain, in the form of DomainModels. Several data types and services are defined to create a software analogue of power ports. A generic power-based impedance controller was implemented, and some components for conversion of forces to rotational and linear velocities. These components were then composed into the simple leader-follower use case (Fig 15).

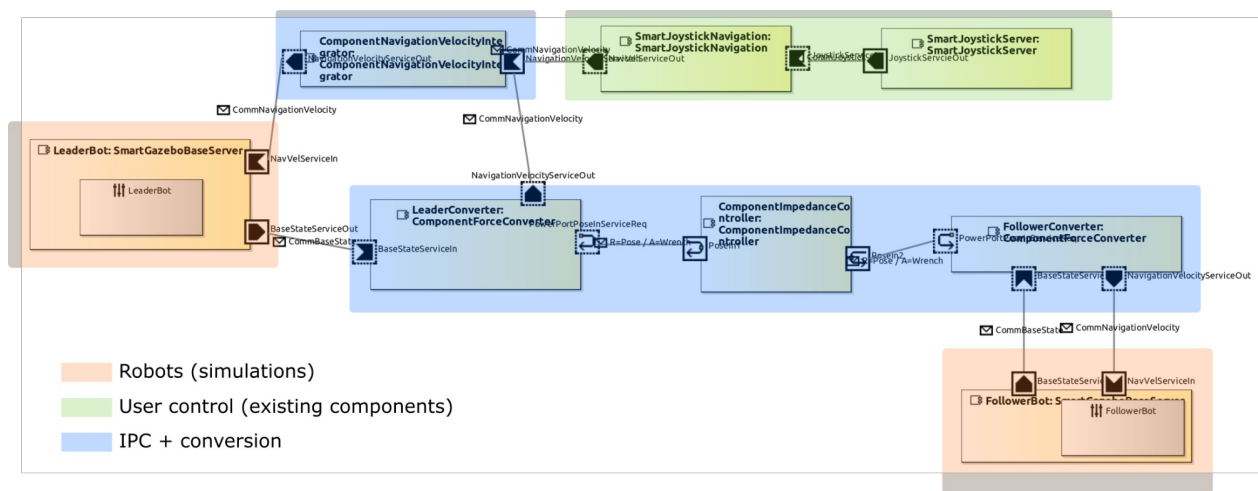


Fig. 15. Screenshot of the SmartMDSD Toolchain: system architecture of the simple leader-follower navigation use case. Existing components for robot simulation and joystick control were connected with new components for IPC.

This exercise had the following benefits:

- By being very formal about the interfaces between components, interdependencies (some unintentional) were quickly identified. For example, although for strict functionality only the position of the devices is needed, the system will function more efficiently (less filtering and processing) when also the velocity is included.
- The translation of power ports (as a modeling construct) into implementable software interfaces gave us more insight into the desired communication patterns (e.g. request-response, guaranteed delivery).
- Some of our desired modelling possibilities (wrapping (sets of) components into other components, two-

way causality of ports) were not possible in the SmartMDSD Toolchain, challenging both us and the toolchain developers to be flexible.

All in all, by being formal and explicit in an implementation-focused way, many hidden problems and inconsistencies became apparent early in the process.

A note on linking to high-order (sequence) control

The energy-based paradigm implies the use of power as interaction currency between components [7]. This approach becomes incompatible with high-level sequence controllers when the control signal is only velocity or position. Other robotic applications, such as navigation, could be benefited by the properties of passivity if an energy consistent interface is provided. To make this interconnection, we propose using a Passive Power Adapter (PPA) to connect a generic trajectory planner to any energy-based component.

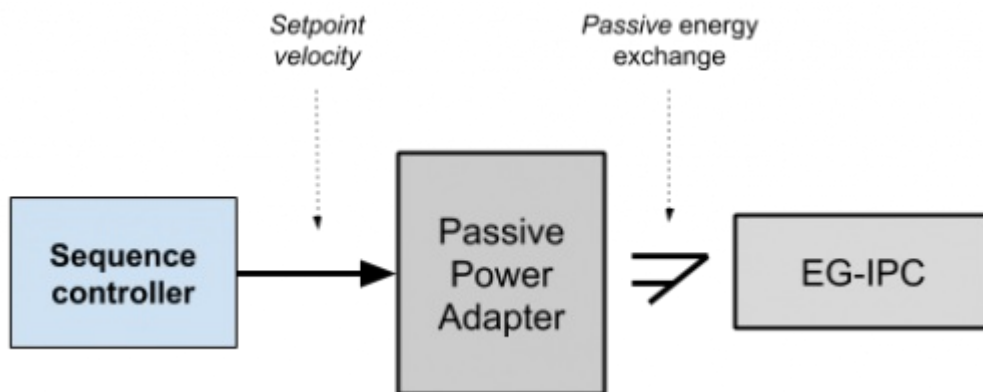


Fig.15. PPA connecting a velocity setpoint generator to energy based components..

The PPA, shown in Fig.15, contains a modulated source of flow, a power estimator and a Passivity Layer. The novel PPA would provide an interface to an EG-IPC without compromising passivity as the PPA's power port abides the estimation of power.

Bibliography

- [1] P. C. Breedveld, "Multibond graph elements in physical systems theory," J. Frankl. Inst., vol. 319, no. 1, pp. 1–36, Jan. 1985.
- [2] P. F. Hokayem and M. W. Spong, "Bilateral teleoperation: An historical survey," Automatica, vol. 42, no. 12, pp. 2035–2057, Dec. 2006.
- [3] J. E. (James E. Colgate, "The control of dynamically interacting systems," Thesis, Massachusetts Institute of Technology, 1988.
- [4] R. B. Gillespie and M. R. Cutkosky, "STABLE USER-SPECIFIC HAPTIC RENDERING OF THE VIRTUAL WALL," 1999.
- [5] R. Ortega and M. W. Spong, "Adaptive motion control of rigid robots: a tutorial," in Proceedings of the 27th IEEE Conference on Decision and Control, 1988, pp. 1575–1584 vol.2.
- [6] R. Cobos, J. de Oliveira, D. Dresscher, and J. Broenink, "A Bond-Graph metamodel," JOT, p. 18, 2019.
- [7] P. J. Gawthrop and G. P. Bevan, "Bond-graph modeling," IEEE Control Syst., vol. 27, no. 2, pp. 24–45, Apr. 2007.

1)

More details about the SmartMDSD Toolchain: <https://wiki.servicerobotik-ulm.de> [<https://wiki.servicerobotik-ulm.de>]

2)

A scientific paper with more details about the EG-IPC and EG-Component architecture approach is currently

in development

community:intrinsically-passive-control:start · Last modified: 2019/05/20 10:49
<http://www.robmosys.eu/wiki/community:intrinsically-passive-control:start>

Page-Status: Incubator

This page is under consolidation and not yet part of the RobMoSys Body of Knowledge. [Read more.](#)

Introduction and Goal of COCORF in the context of RobMoSys

Hard real-time control software is often designed as monolithic components in order to circumvent the lack of temporal determinism of today's general purpose robot middlewares. COCORF will overcome this by combining the RobMoSys composition approach with the proven microblx real-time function block framework [1, 2]. This will enable building modular and predicable control systems by composing reusable blocks based on the function block model of computation. Microblx introspection facilities will be used to generate RobMoSys compliant digital function block data-sheets containing formalized constraints and QoS properties. Interoperability to existing robotics middlewares as well as separation of criticality domains will be realized by means of mixed-port function blocks. To validate the technology and to support early adopters, the approach will be illustrated by developing a generic control reference architecture and accompanying demonstration for the Assistive Mobile Manipulation Pilot. In this manner, COCORF lays the foundation for an extended RobMoSys Ecosystem of reusable, real-time function blocks.

The project foresees four milestones:

- **M1:** RobMoSys compliant composition DSL and digital data-sheet for function blocks
- **M2:** Mixed-port function blocks to ROS are available
- **M3:** Reference architecture completed
- **M4:** End of project

Contribution to RobMoSys User Stories

COCORF contributes to the following RobMoSys user stories [8, 9]:

- [Composition of components](https://robmosys.eu/wiki/general_principles:user_stories#composition_of_components)
[https://robmosys.eu/wiki/general_principles:user_stories#composition_of_components]:

By using the COCORF composition extensions, system builders are supported in the construction of reusable, hard real-time safe components.

- [Management of Non-Functional Properties](https://robmosys.eu/wiki/general_principles:user_stories#management_of_non-functional_properties)
[https://robmosys.eu/wiki/general_principles:user_stories#management_of_non-functional_properties]:

microblx supports management of non-functional properties by providing relevant information. For example, the current execution timing properties (i.e. min/max) of schedules are made available on a `tstats` port (also see demo below).

- [Description of building blocks via model-based data sheets](https://robmosys.eu/wiki/general_principles:user_stories#description_of_building_blocks_via_model-based_data_sheets)
[https://robmosys.eu/wiki/general_principles:user_stories#description_of_building_blocks_via_model-based_data_sheets]

(WIP). COCORF is developing tooling to allow automatic data-sheet generation from function blocks. This provides Component builders with accurate and up-to-date information on the blocks they are composing.

Technical Details

Technical Description

Composition DSL and digital data-sheet

microblx uses a Lua based internal DSL to describe systems in so-called `.usc` (microblx system composition) files. Until COCORF, these descriptions contained a flat list of blocks, configurations and connections. This works fine for small systems, but falls short for larger systems where reusability and modularity are of more concern.

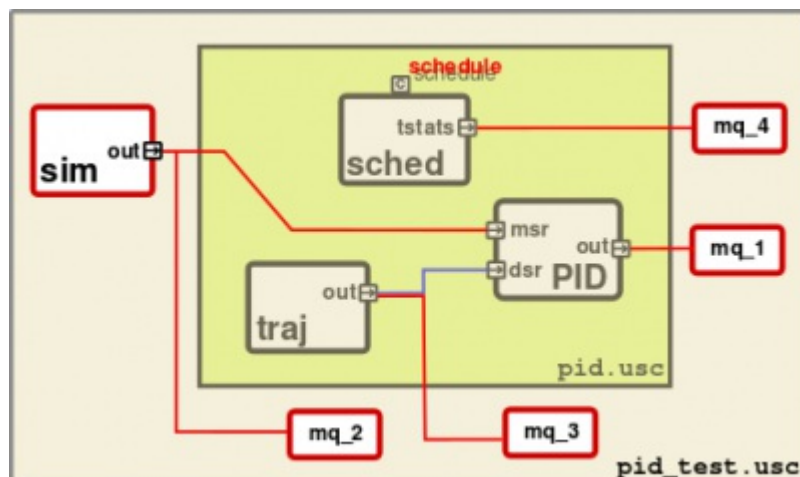
The technical requirements for the composition extensions as well as the high-level approach are elaborated in a technical requirements document [5]. The main requirements are:

- The DSL must allow to include one or more subsystems
- An included subsystem must not require any knowledge of the supersystem (or that it is being included at all).
- It must be possible to override configurations of the subsystem from the super-system.
- It must be possible to connect to ports of an included subsystem
- It should be possible to remove elements of the subsystem from the supersystem (blocks, connections).
- The use of (third party) schedule computation tools via a form of plugin mechanism shall be supported. Nevertheless, it must be possible to specify schedules manually.

The implementation went largely as planned, however the following insight required deviating from the original plan. The original idea of extending `ubx_launch(1)` with a `-a` option to specify an activity model was abandoned in favor of the more generic approach of a “model mixin”. The latter permits specifying one or more additional models on the command line, of which each will be *merged* into the primary one. This not only addresses the requirement of supporting *late* binding to the platform (i.e. typically to different activities), but provides an additional, orthogonal composition “operator”. Although not explored in depth, this promises to be useful for further use-cases (e.g. instrumentation), where functionality must be added while pollution of a reusable composition avoided.

Demo: PID controller

As a self-contained example (to be shown at the ERF), a software only demo was prepared to illustrate i) how composition enables reuse and how ii) late binding to platform specific aspects is achieved using the model mixin feature. The code for this example can be found here [6].



The green composition (`pid.usc`) defines a reusable controller consisting of a trajectory generator (block `traj`, a 10d ramp in the example), a PID controller and a schedule (encapsulated by a `passive_rig` block). One connection is made within the `pid` composition: from the trajectory generator `out` to the `PID_des` port.

This composition can be now be reused in different use-cases and respective environment. This is achieved by *including* the basic composition in a parent composition, which *overlays* it to adapt it to its specific environment. For this demo, the `pid.usc` composition is to be reused in a simulation/testing environment (`pid_test.usc`). To that end a simulated sensor signal (`sim`) block is created and connected to the `PIDmsr` input. The schedule configuration is extended to include triggering of the new `sim` block. Additional connections are created to export the various internal signals from the blocks using real-time message queues (`mq_X` blocks) for debugging purposes. The connection from `sched.tstats` exports timing statistics such as worst case execution times out of the real-time context.

Alternatively, `pid.usc` could be reused on a real system by including it in a parent composition similar to `pid_test.usc`. Instead of creating and connecting a simulator block, such a composition would instead create a robot driver or sensor block and create the appropriate connections.

The example can be launched using the following command (executed from the directory `microblx/examples/usc/pid`):

```
$ ubx_launch -c pid_test.usc,ptrig_nrt.usc
```

This launches `pid_test.usc` after merging the platform specific, `minimalpthread_nrt.usc` composition, which instantiates a platform specific pthread trigger with non-real-time priorities (hence the `_nrt`).

For running with real-time priorities, the `ptrig_rt.usc` file can be specified instead of `ptrig_nrt.usc`. Note that this requires to be run with elevated privileges:

```
$ sudo ubx_launch -c pid_test.usc,ptrig_rt.usc
```

The exported data can be viewed using the `ubx-mq(1)` command:

```
$ ubx-mq list
243b40de92698defa93a145ace0616d2 1 trig_1-tstats
e8cd7da078a86726031ad64f35f5a6c0 10 ramp_des-out
e8cd7da078a86726031ad64f35f5a6c0 10 ramp_msr-out
e8cd7da078a86726031ad64f35f5a6c0 10 controller_pid-out
```

and

```
$ ubx-mq read controller_pid-out  
{10202389818.28,10202389818.28,10202389818.28,10202389818.28,10202389818.28,10202389818.  
{10202434989.98,10202434989.98,10202434989.98,10202434989.98,10202434989.98,10202434989.  
...  
}
```

Digital data-sheet generation

Digital data sheets [10] summarize important information about function blocks. This information, rendered in a human readable form supports system builders to effectively decide whether a block is suitable for a specific scenario and ii) how it can be used. For instance, a digital data-sheet should contain general information about the block such as its Software license, but also specific technical information such as the block API.

For microblx, digital datasheets can be generated by leveraging the existing introspection functionalities. To that end, the `ubx-modinfo` tool was extended to generate data sheets in various formats including JSON, reStructuredText and plain text. For example, the following shows the plain text version of the PID controller data sheet:

```
$ ubx-moinfo show pid
```

```

module pid
  license: BSD-3-Clause

  types:

  blocks:
    pid [state: preinit, steps: 0] (type: cblock, prototype: false, attrs: )
  ports:
    msr [in: double #conn: 0] // measured input signal
    des [in: double #conn: 0] // desired input signal
    out [out: double #conn: 0] // controller output
  configs:
    Kp [double] nil // P-gain (def: 0)
    Ki [double] nil // I-gain (def: 0)
    Kd [double] nil // D-gain (def: 0)
    data_len [long] nil // length of signal array (def: 1)

```

Furthermore, the tool is already used to generate documentation in the reStructuredText format which in turn is used by the sphinx tool. The following command generates a reSt document for the PID block:

```

ubx-modinfo dump pid -f rest
Module pid
-----

Block pid
^^^^^^^^^^

| **Type**:      cblock
| **Attributes**:
| **Meta-data**: { doc='',  realtime=true,}
| **License**:   BSD-3-Clause

Configs
^^^^^^

.. csv-table::
   :header: "name", "type", "doc"

   Kp, ``double``, "P-gain (def: 0)"
   Ki, ``double``, "I-gain (def: 0)"
   Kd, ``double``, "D-gain (def: 0)"
   data_len, ``long``, "length of signal array (def: 1)"

Ports
^^^^

.. csv-table::
   :header: "name", "out type", "out len", "in type", "in len", "doc"

   msr, , , ``double``, 1, "measured input signal"
   des, , , ``double``, 1, "desired input signal"
   out, ``double``, 1, , , "controller output"

```

The result can be seen [here \[https://microblx.readthedocs.io/en/latest/block_index.html#module-trig\]](https://microblx.readthedocs.io/en/latest/block_index.html#module-trig).

Mixed-port function blocks to ROS

Started in March 2020.

Reference architecture

Work planned to start in July 2020.

Link/Entry point to RobMoSys

COCORF has several connections to RobMoSys. Firstly, the developments in workpackage 1 (composable DSL and digital data-sheet) are *applications* of the RobMoSys approach in a concrete, industrial software framework. Similarly, workpackage 2 (mixed port function blocks) transfers the RobMoSys mixed port connector approach from component based development to hard real-time function blocks.

Current State and Roadmap

The composition extensions have been realized and a first version has been announced on discourse [7]. Block digital data-sheet support has been completed and is already used to generate stub function block data-sheets[11]. The code changes for both features have been merged to the microblx development branch and are scheduled for release with the upcoming v0.8 release. At the time of writing, a third release candidate `v0.8-rc3` has been released.

Workpackage 2 has the goal of implementing mixed port function blocks to connect to the ROS framework. Work has started in March 2020 as planned. First investigations of existing code has been carried out and a development of a prototype has started.

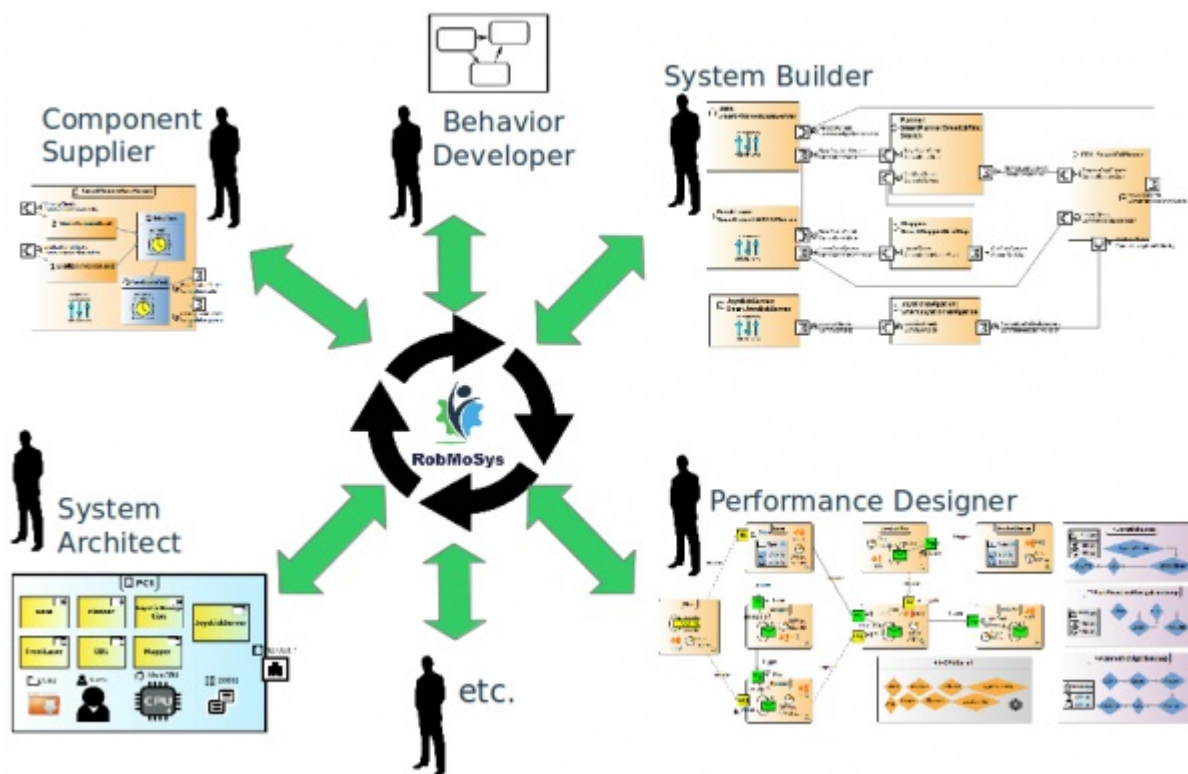
References

- [1] Markus Klotzbücher, Herman Bruyninckx, microblx: a reflective, real-time safe, embedded function blockframework, Proceedings of the Real-Time Linux Workshop 2013
- [2] Markus Klotzbuecher et al., microblx repository, 2020, <https://github.com/kmarkus/microblx>
- [3] RobMoSys, Roles in the Ecosystem, 2019, [Roles in the Ecosystem \[https://robmosys.eu/wiki-sn-03/general_principles:ecosystem:roles\]](https://robmosys.eu/wiki-sn-03/general_principles:ecosystem:roles)
- [4] RobMoSys, Mixed-port Component, 2019, [Mixed-port component \[https://robmosys.eu/wiki/open-call-2?s%5B%5D=mixed&s%5B%5D=port&s%5B%5D=component#mixed-port_component\]](https://robmosys.eu/wiki/open-call-2?s%5B%5D=mixed&s%5B%5D=port&s%5B%5D=component#mixed-port_component)
- [5] [Technical requirements for RobMoSys compliant composition for microblx launch DSL] (<https://github.com/kmarkus/microblx/blob/cocorf/docs/dev/001-blockdiagram-composition.md>)
- [6] [PID demo](<https://github.com/kmarkus/microblx/tree/cocorf/examples/usc/pid>)
- [7] [COCORF composition extension announcement](<https://discourse.robmosys.eu/t/cocorf-draft-concept-for-microblx-composition-extensions/259>)
- [8] <https://robmosys.eu/user-stories/>
- [9] https://robmosys.eu/wiki/general_principles:user_stories
- [10] <https://wiki.servicerobotik-ulm.de/how-tos:documentation-datasheet:start>
- [11] https://microblx.readthedocs.io/en/latest/block_index.html

community:cocorf:start · Last modified: 2020/04/08 10:55
<http://www.robmosys.eu/wiki/community:cocorf: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.

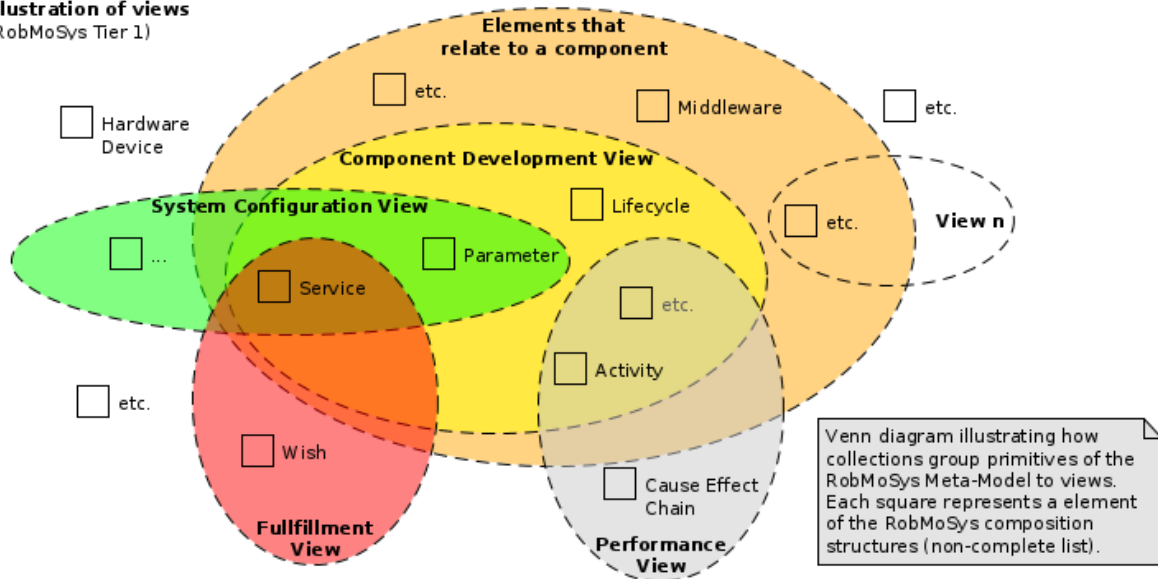


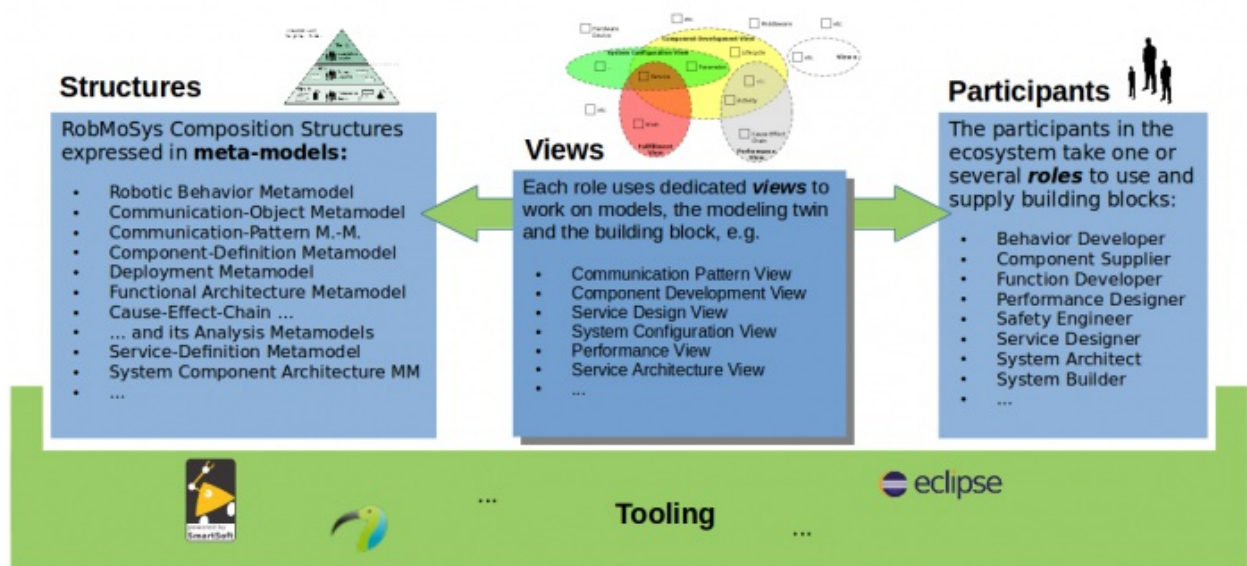
Each role that participates in the ecosystem uses a dedicated view to focus on its responsibility and expertise.

The concept of “views” groups basic primitives of the RobMoSys composition structures. A view is related to a role and establishes the link between primitives in the RobMoSys composition structures 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.

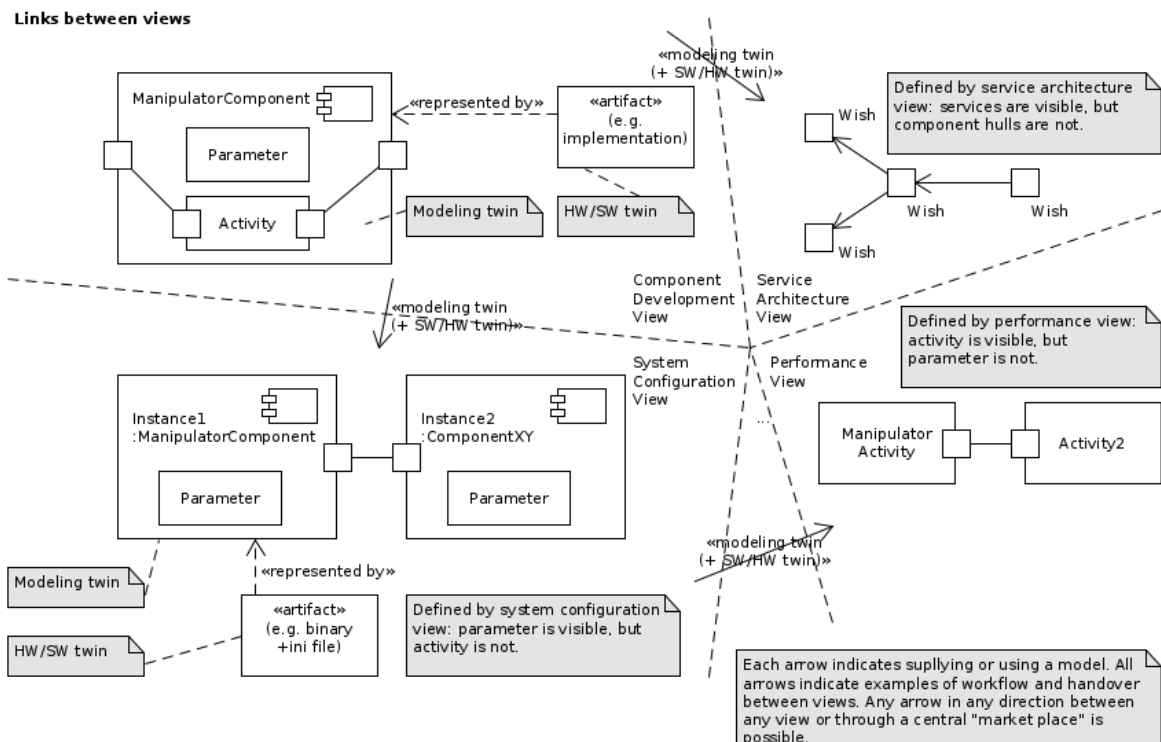
Illustration of views
(RobMoSys Tier 1)





Links Between Views: Example 1

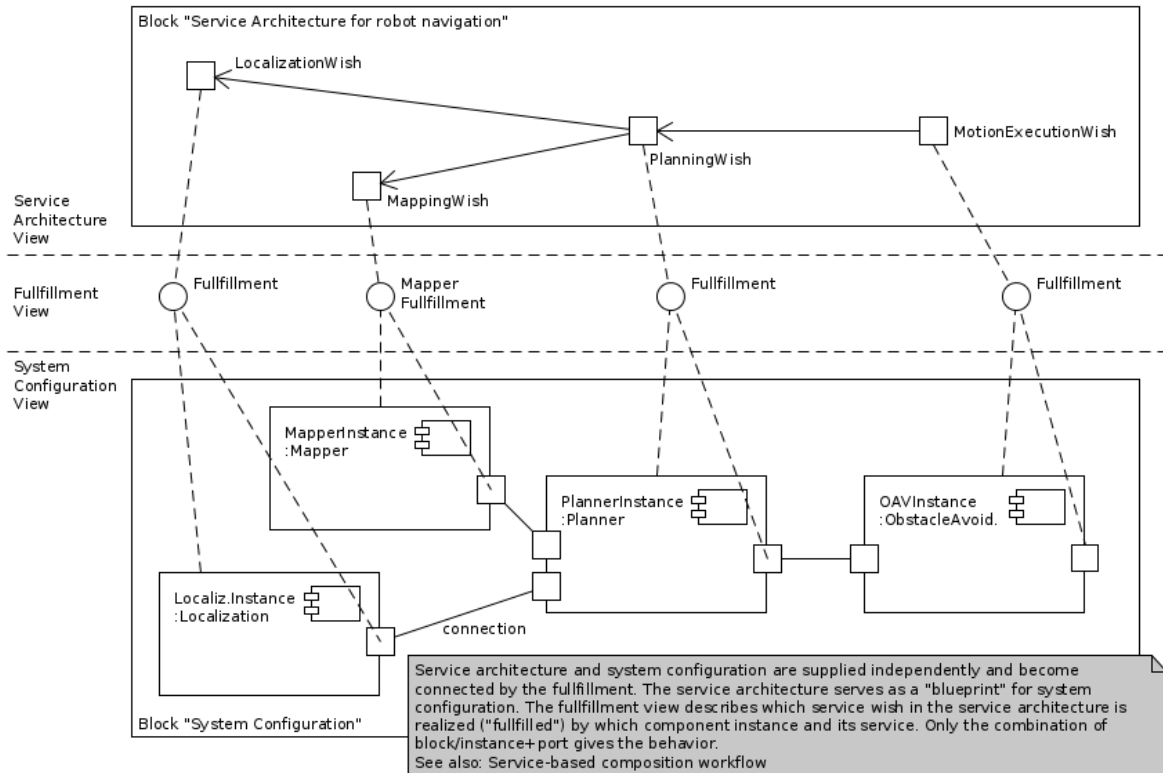
The figure below illustrates the link between several views. The Modeling Twin is handed over between one view to the next. There is no strict order in the sense of a strictly order value chain. Instead, the interactions form a network of collaborating roles consisting of various bilateral interactions between suppliers and consumers.



Links Between Views: 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



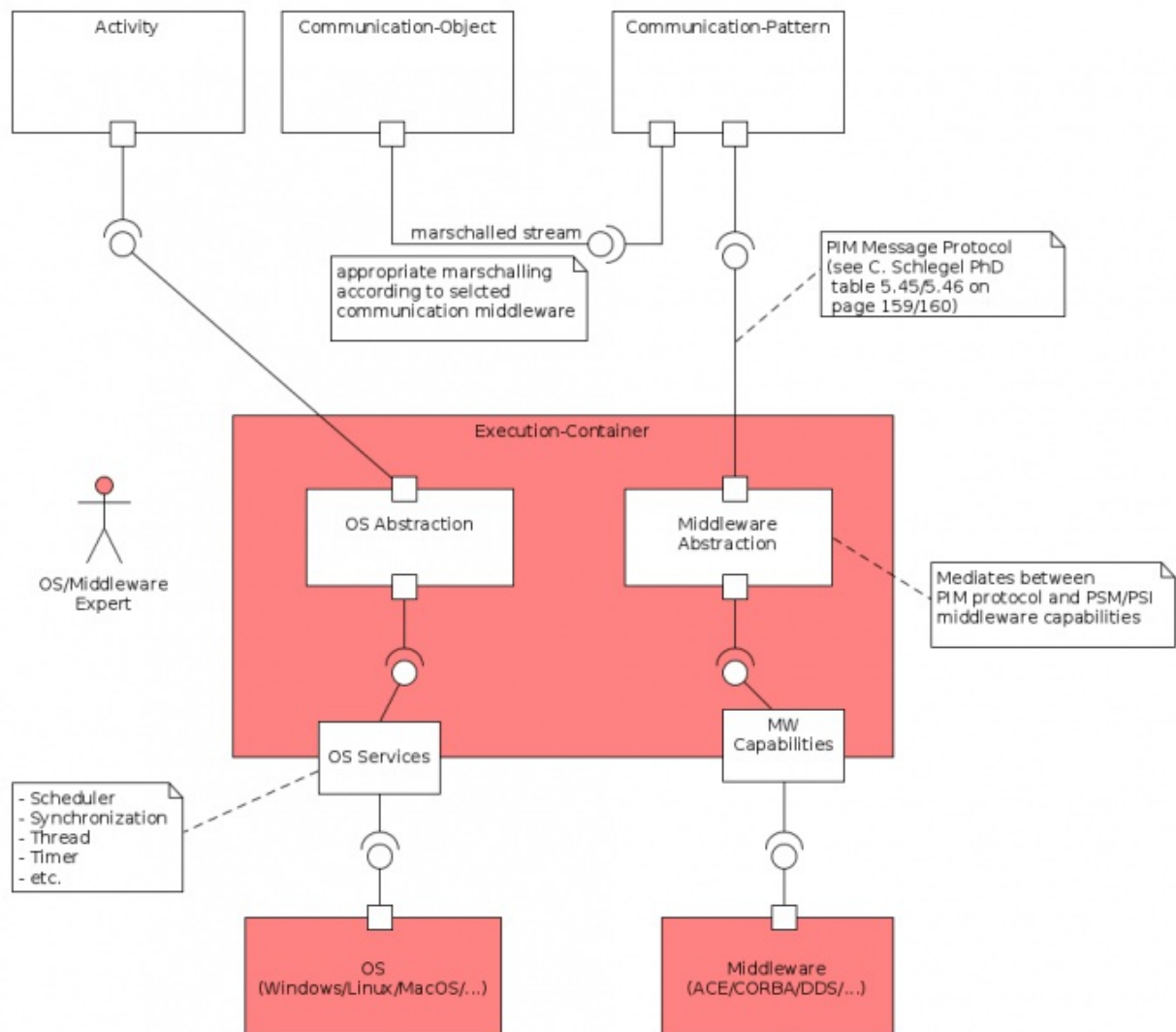
See also

- [Views in the RobMoSys Glossary](#)
- [Views in RobMoSys Composition Structures](#)
- [Views in the PC domain analogy](#)
- [Roles in the Ecosystem](#)

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.



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

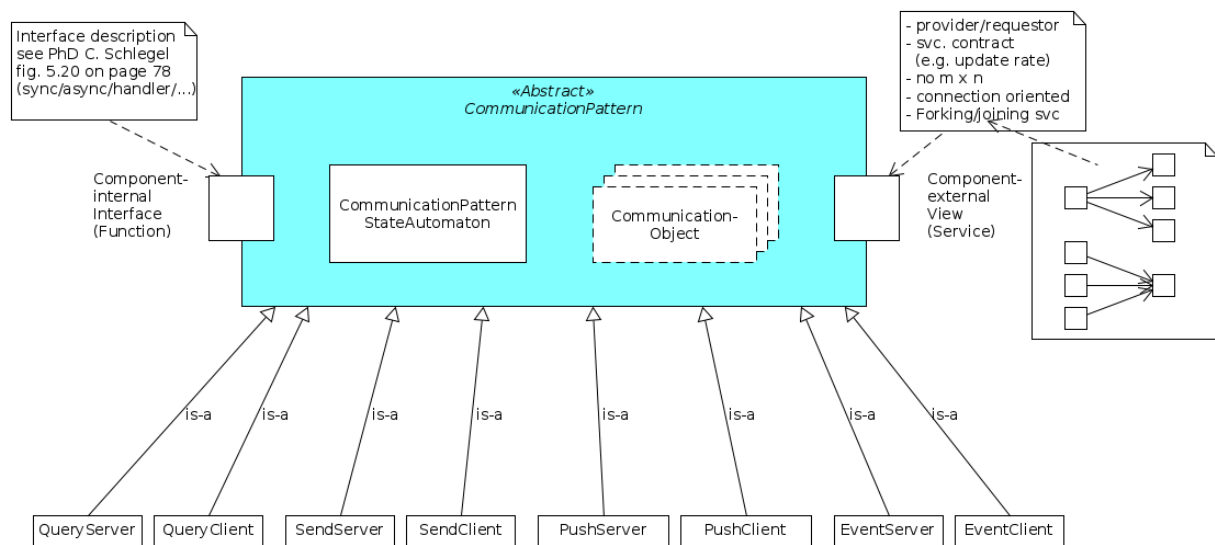
complied with).

modeling:views:execution_container · Last modified: 2019/05/20 10:49
http://www.robmosys.eu/wiki/modeling:views:execution_container

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.



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 existing API 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>]

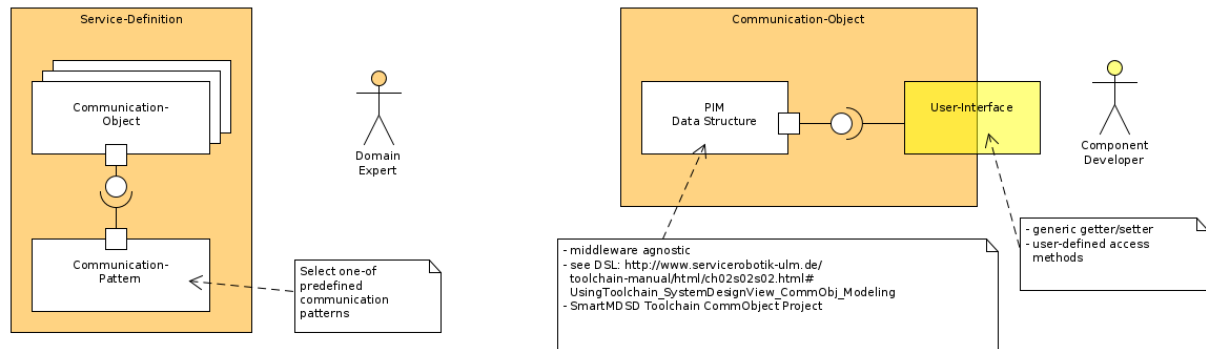
See also

- Communication-Pattern Metamodel

Service Design View

The service design view clusters elements of the Service Metamodel that are relevant to the Service Designer.

Service Design View

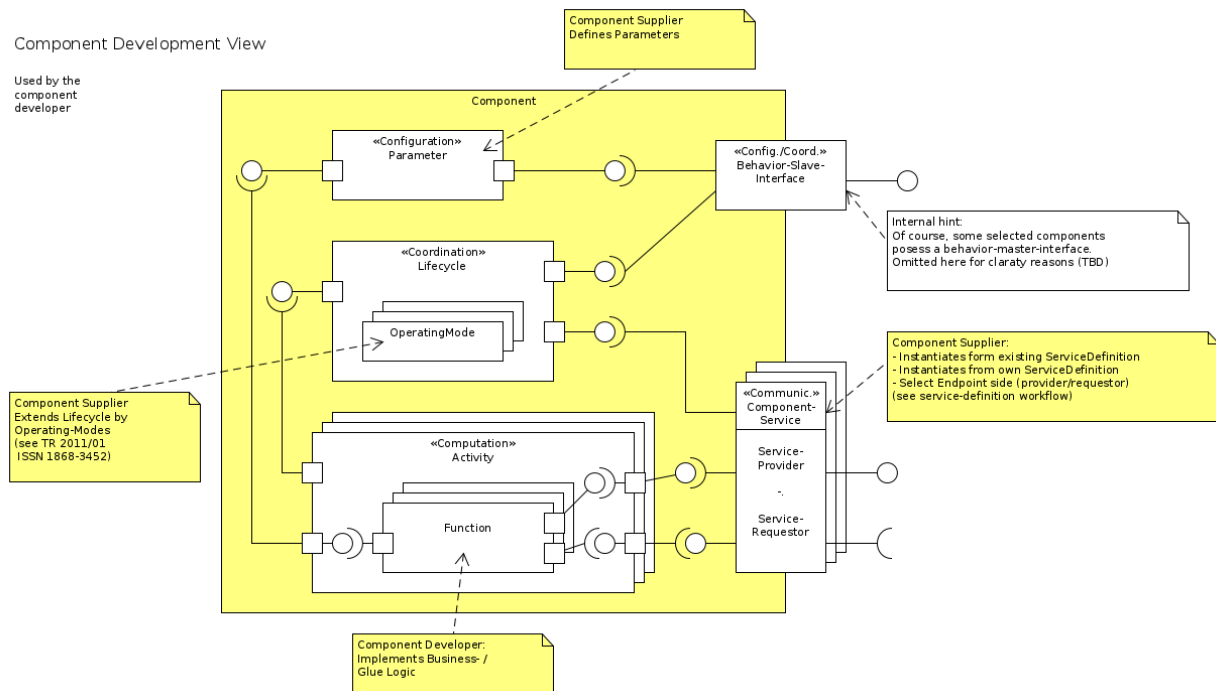


A service definition (shown on the left in the figure) comprises of a selection of a communication pattern and a selection of a communication object. A communication object is a data structure that is communicated between a service provider and a service requestor. The exact direction of communication 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.

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.



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 proper 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

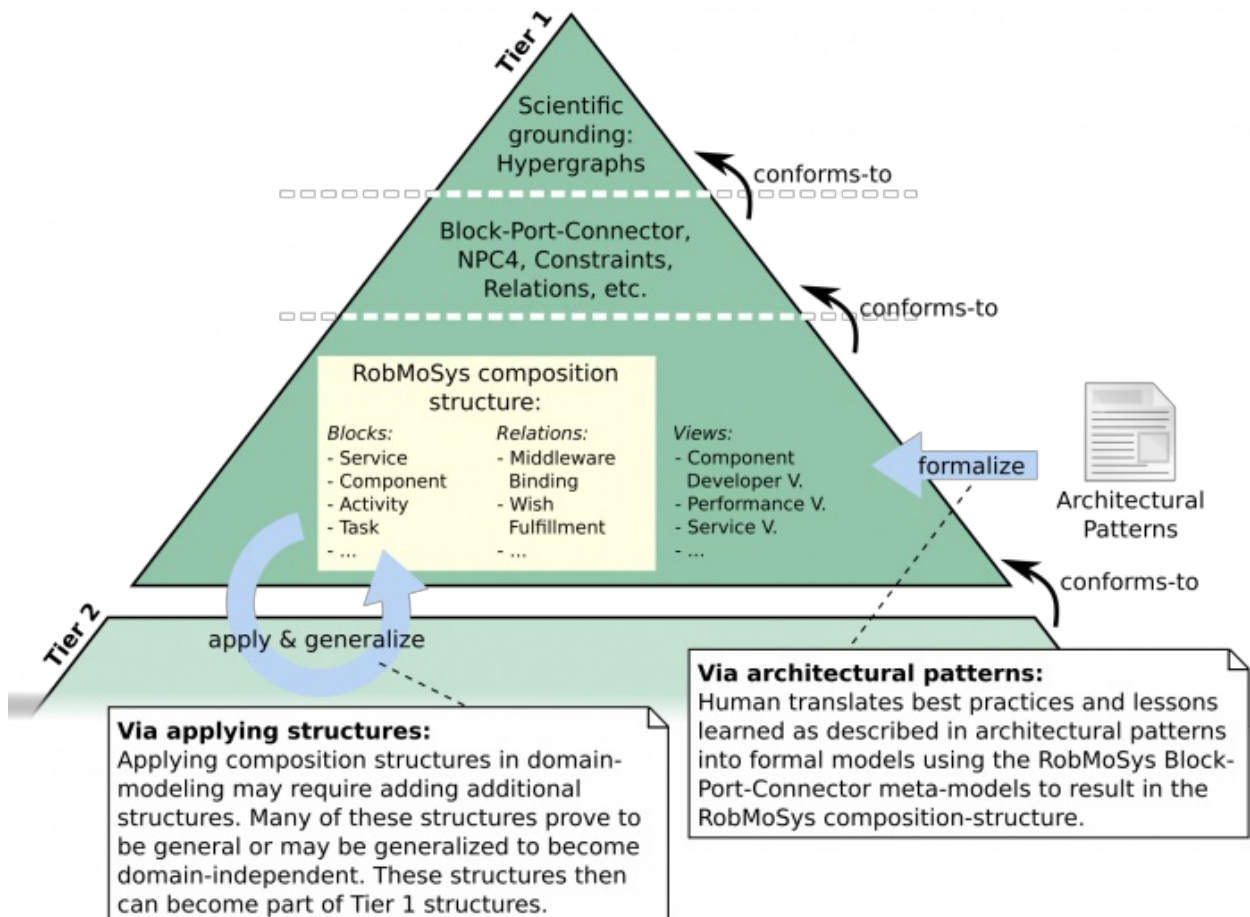
more precisely of a thread). An activity wraps a functional block which by itself is passive and only gets active by the execution environment of its parent Activity. This is an effective decoupling of the design and implementation of functional parts within a component and the execution of the functions. 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).

modeling:views:component_development · Last modified: 2019/05/20 10:49
http://www.robmosys.eu/wiki/modeling:views:component_development

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.



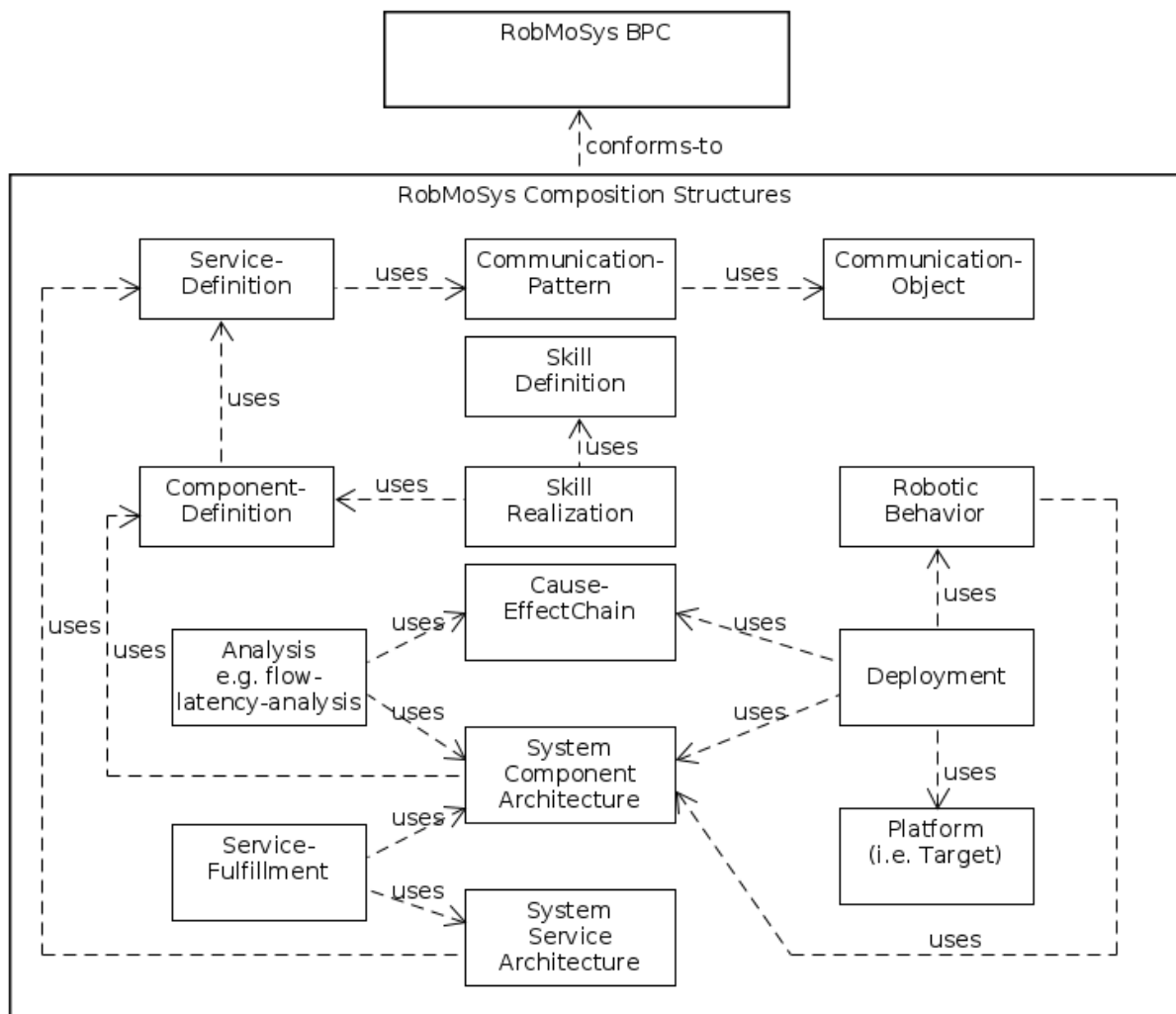
The figure (above) shows an exemplary list of possible composition structures (highlighted with the yellow background color), which can be clustered into (a) specializations of **blocks** and (b) specializations of **relations**. One of the central structures defined by RobMoSys is a consistent and rich 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 [Frequently Asked Questions](#).

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](#)
- [Skill Definition Metamodel](#)
- [Skill Realization Metamodel](#)
- [System Component Architecture Metamodel](#)
- [Task Definition Metamodel](#)
- [Task Realization Metamodel](#)

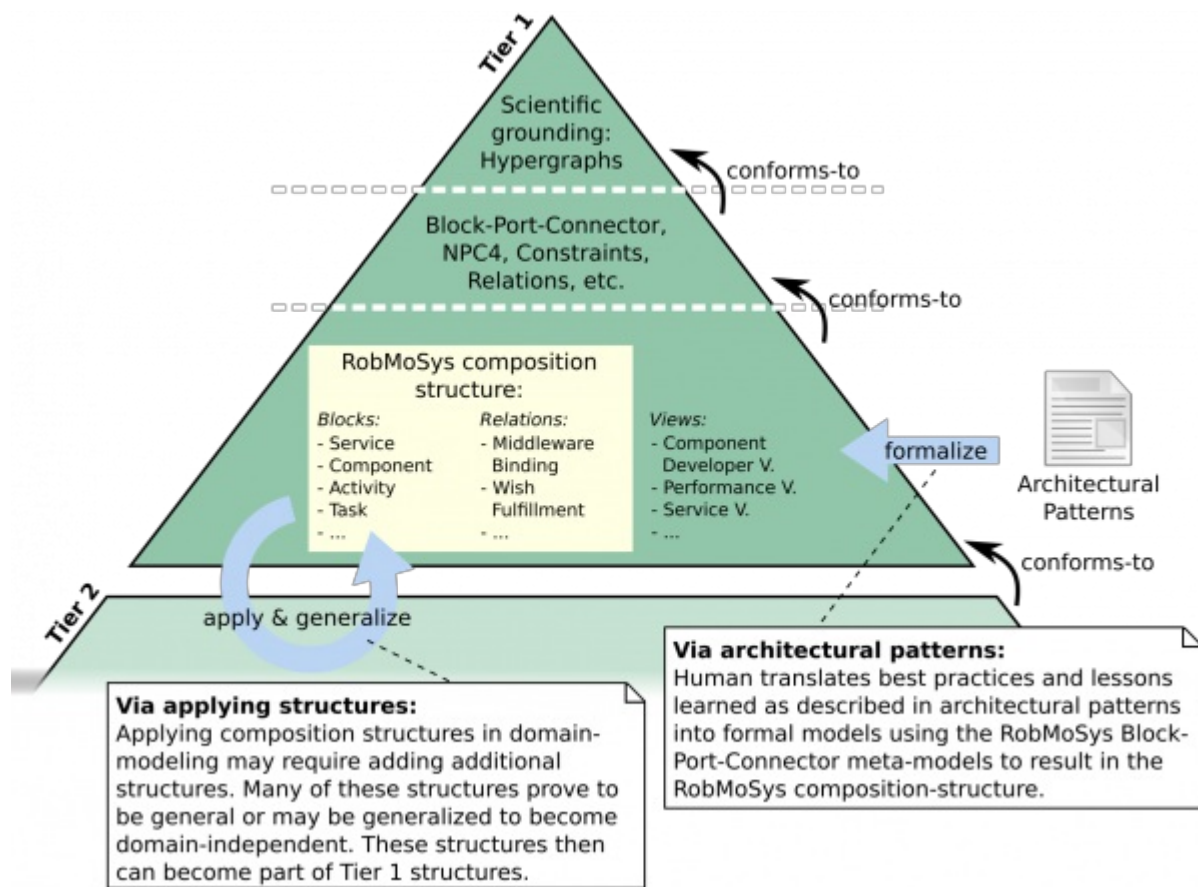
See also:

- [RobMoSys Views](#)

modeling:composition-structures:start · Last modified: 2019/05/20 10:49
<http://www.robmosys.eu/wiki/modeling:composition-structures:start>

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.



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.

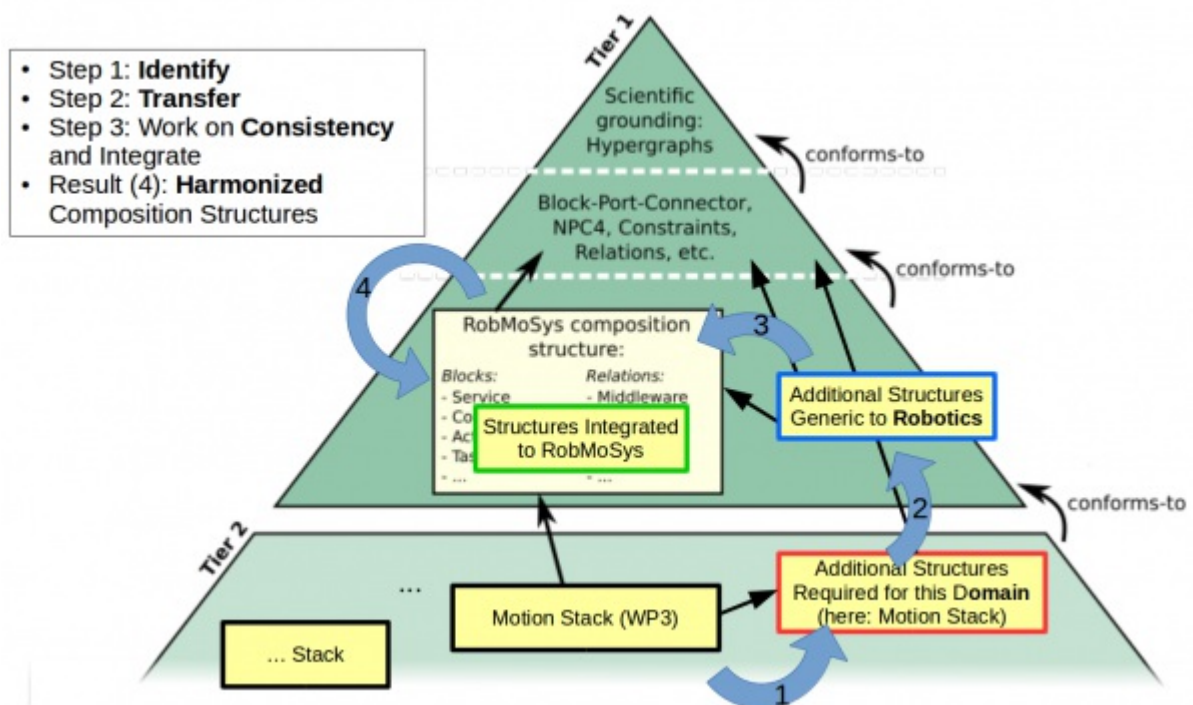
The [RobMoSys Views](#) page provides further details.

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](#).

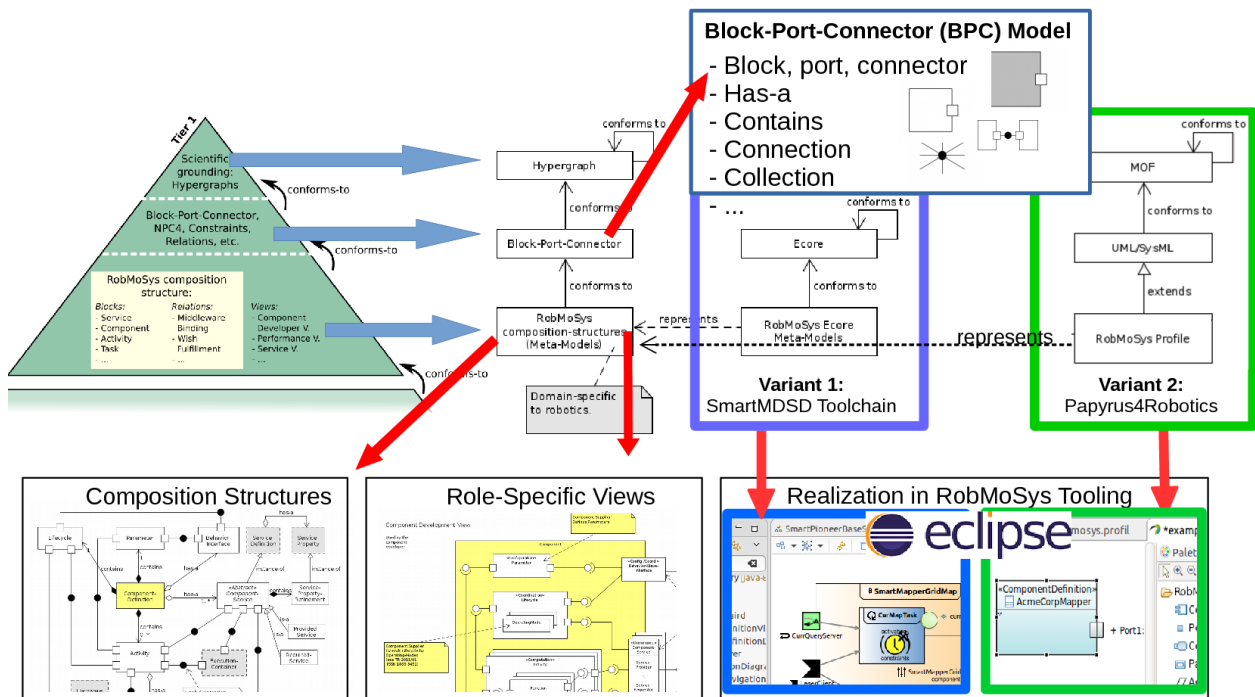
The second approach is to evolve the composition structures over time by generalizing existing domain-specific structures. In some cases, the composition structures of Tier 1 may not be sufficient or not complete for modeling in a particular robotics domain. This situation requires additional structures 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.



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).



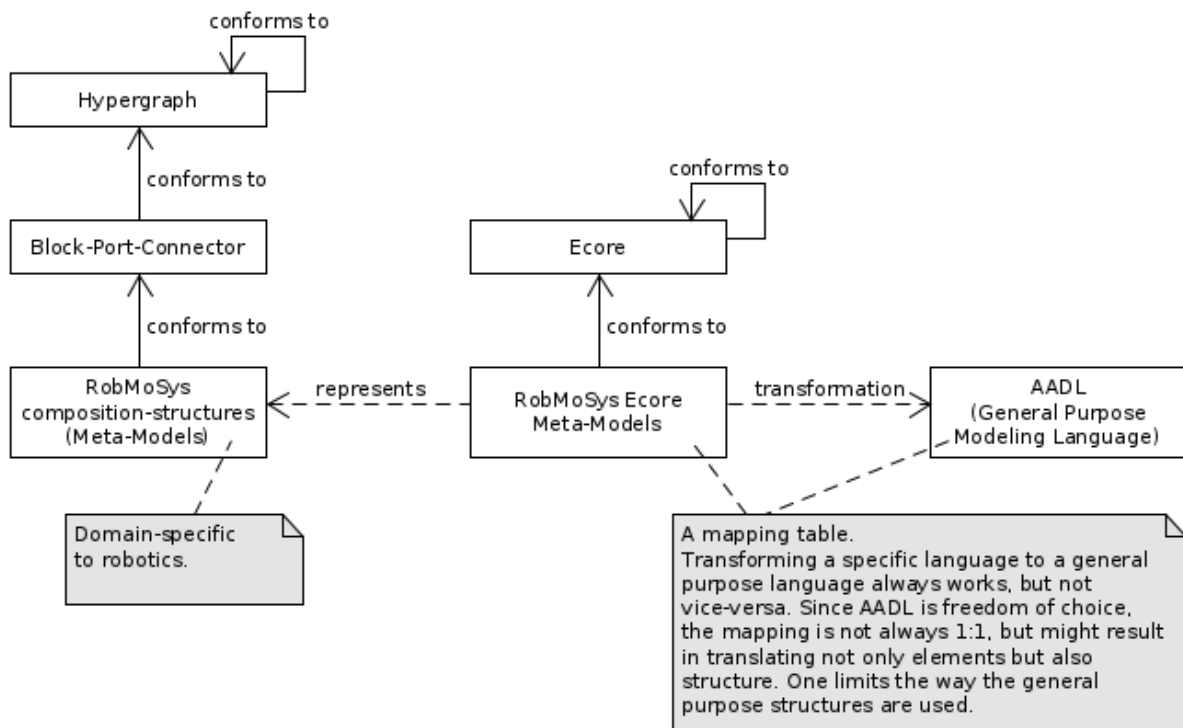
modeling:tier1 · Last modified: 2019/05/20 10:47
<http://www.robmosys.eu/wiki/modeling:tier1>

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.



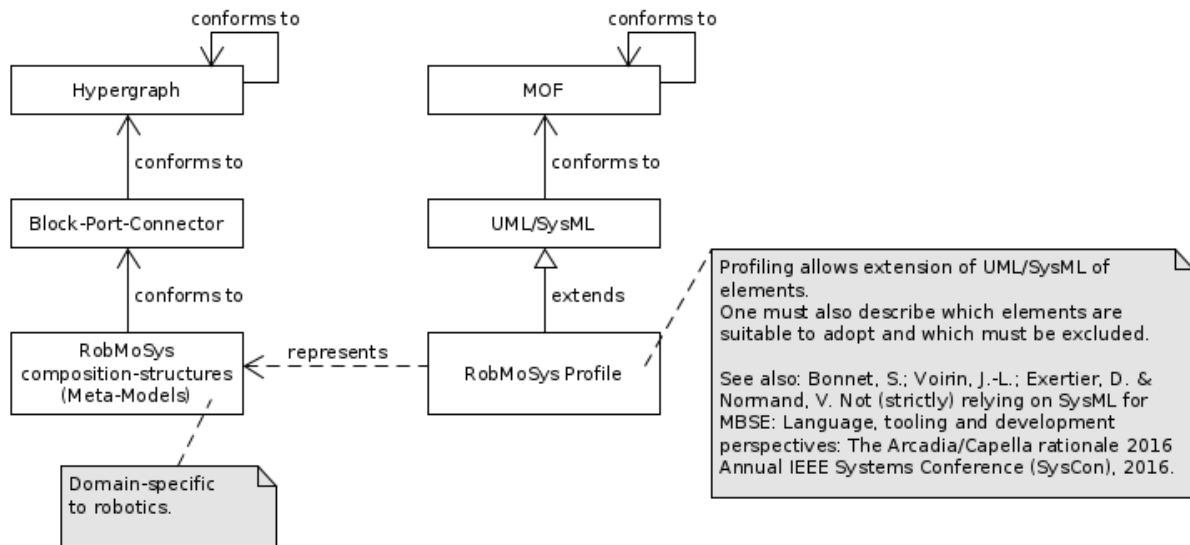
The left side of the figure shows a meta-level hierarchy starting with a Hypergraph on top, over Blocks-Ports-Connectors and down to RobMoSys composition structures. This hierarchy allows formal definition of meta-levels for the required structures independent of a particular realization technology. 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



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.


Block-Port-Connector


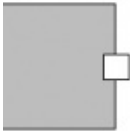
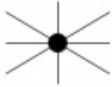
The Block-Port-Connector model is a specialization of the more abstract Hypergraph and Entity-Relation model.

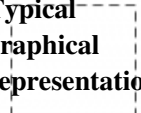
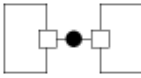
The following generic **relations** have been introduced already: **is-a**, **instance-of**, **conforms-to** and **constraints**. There are two additional (i.e. more specific) relations that need to be introduced:

Relation	Explanation	Typical graphical representation	Typical textual representation
contains	<p># can be applied to entities and can be applied to relations</p> <ul style="list-style-type: none"> * this realizes hierarchical composition (nested composition); in a hierarchical composition elements are enclosed by another element * contains is topology * the contained elements are not accessible/visible (in contrast to elements in a collection) * 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) contains(B,m,n)
has-a	<p># can be applied to entities and can be applied to relations</p> <ul style="list-style-type: none"> * this realizes aggregation * has-a is mereology * in aggregation, elements remain at the same level * elements linked with has-a remain accessible/visible * 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	<p>Model:</p> <ul style="list-style-type: none"> * is-a entity * possibly has-a property (or many) * possibly has-a port (or many) * possibly contains property (or many) 		block(block-A)

Entity/Relation	Model and Description	Typical graphical representation	Typical textual representation
	<ul style="list-style-type: none"> * possibly contains block (or many) * possibly contains collection (or many) * possibly contains connector (or many) * possibly contains relation (or many) 		
	Description: the only interaction points of a block are ports		
port	Model: <ul style="list-style-type: none"> * is-a entity * has-a internal dock * has-a external dock 		port(Port-A)
	Description: it is the only interaction point over which a block can interact with other blocks; when attached to a block, the internal dock becomes a private to the block (contains) and the external dock becomes public (has-a)		
	Comment: In textual representation, access to docks can be represented e.g. like internal-dock(Port-A), external-dock(Port-A)		
dock	Model: <ul style="list-style-type: none"> * is-a entity 		dock(Dock-A)
	Description: 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: 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: <ul style="list-style-type: none"> * is-a entity * connects ports (n-ary relation) 		connector(connector-A)
	Description: can connect ports as long as no block boundaries are crossed		
	Comment: 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: <ul style="list-style-type: none"> * is-a entity * possibly has-a entity (or many) * possibly has-a relation (or many) 		collection(collection-C,k,l,m,n)

Entity/Relation	Model and Description Description:	Typical graphical representation	Typical textual representation
	<p>A collection can group any combination of entities and / or relations. The enclosure formed by a collection is just a virtual one where the elements are openly accessible (in contrast to nesting).</p> <p>A collection can pick any elements out of blocks ignoring block boundaries \Rightarrow this is particularly useful to specify modeling views</p> <p>Comment:</p> <p>In the graphical representation, the dotted box can enclose entities and / or relations where you can cross the dotted line to “enter” the collection</p>		
connects	<p>Model:</p> <p>is-a relation</p> <p>Description:</p> <p>links a dock of a port to a connector (binary relation)</p>		connects(connector-A, external-dock(Port-A))

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.

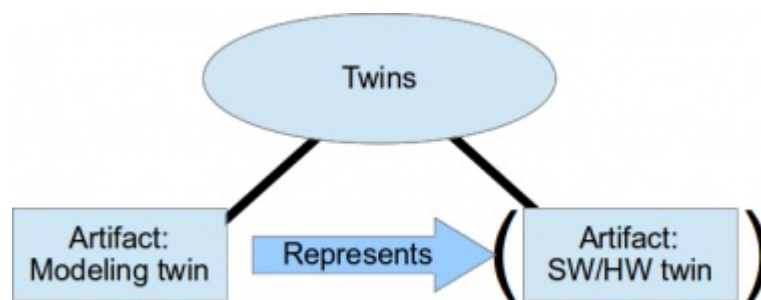
Please note that while **blocks** and **ports** are semantically different, depending on the current role-specific view with according level of abstraction, ports can contain additional structures and thus might appear as blocks on that detailed abstraction level (see service-definition metamodel).

See next

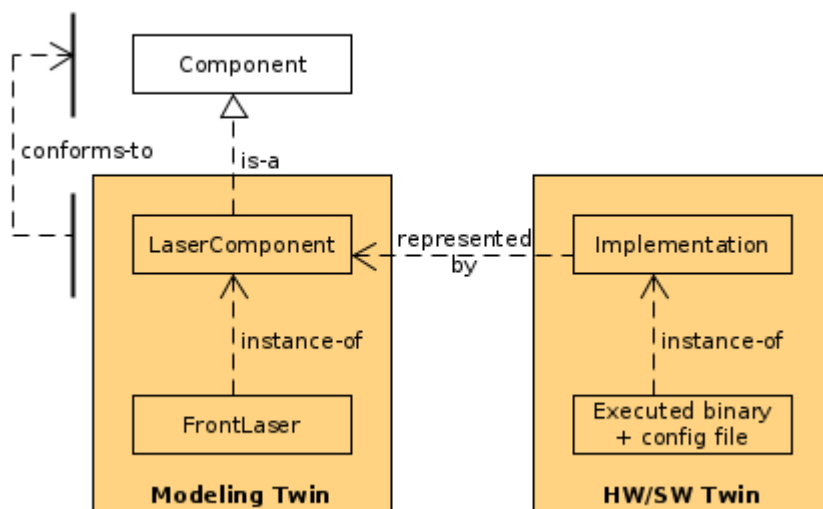
- RobMoSys composition structures

Modeling 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 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]

modeling:principles:modeling-twin · Last modified: 2019/05/20 10:49
<http://www.robmosys.eu/wiki/modeling:principles:modeling-twin>

Scientific Grounding

The highest abstraction level that is considered in RobMoSys is related to Hierarchical Hypergraphs and Entity-Relation models. The Entity-Relationship¹⁾ 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_\(object-oriented_design\)](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) hypergraph⁷⁾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 provenance⁹⁾, and Dublin Core¹⁰⁾.

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 languages, 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-readable 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 latter’s properties.

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 described¹¹⁾:

- **mereology** – parts: there is already quite some (mature) formalisation available in the state of the art, to structure “Entities”; for example, the [Wikipedia article \[https://en.wikipedia.org/wiki/Mereology\]](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 down 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) [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. Poulouvasilis. 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: 2019/05/20 10:47
<http://www.robmosys.eu/wiki/modeling:hypergraph-er>

Basic Modeling Principles

What is "Modeling"?

“Modelling” is at the core of the RobMoSys ambition and approach, and the project has identified these five complementary levels in model formality:

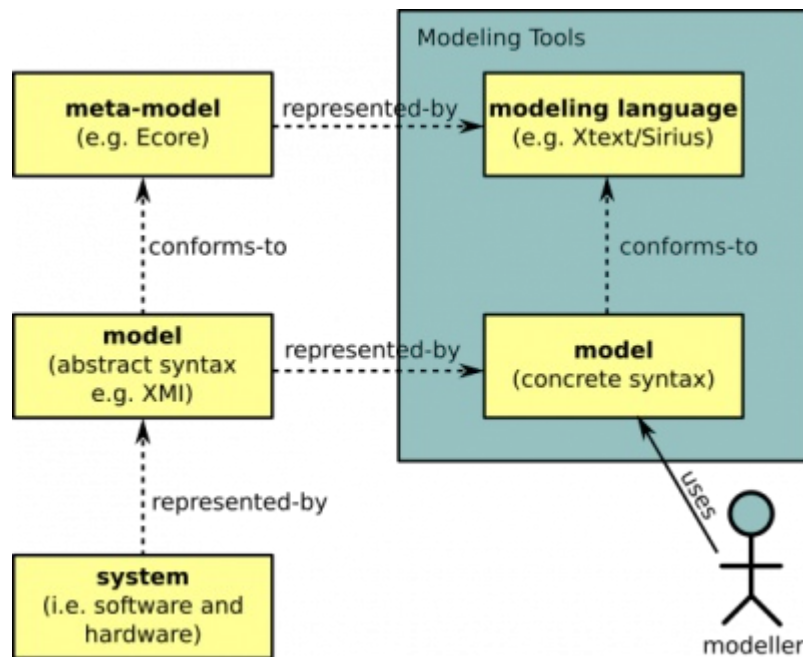
- 1. Models for human discussions: such models must provide guidance to discussions between human developers, and hence result in more and more harmonized interpretations of the relevant entities and relations.
- 2. Models for human software documentation: only when the mentioned “level 1.” harmonization has matured in a community, one can expect software to be developed whose meaning and behaviour are well understood by all developers in that community.
- 3. Models for software tools and standards: the harmonization has reached a level of maturity where the explanation in a standards document, and the availability of a reference implementation that conforms to the standard document, suffice to let everyone use software artefacts based on the standard with unambiguous interpretations. Hence, support from software tools becomes realistic, or even mandatory as the major pragmatic way of software development in a broader community.
- 4. Models for verification and validation: the standardisation has been formalized so far that the meaning and behaviour of software artefacts can be checked automatically.
- 5. Models for run-time dialogues between machines: the formalization and its automatic checking has become (1) efficient enough to be used at runtime by the robots themselves, and (2) rich enough so that the machines can set up added-value cooperations themselves via dialogues, or can configure, adapt and explain their own behaviour, with only minor human interaction.

The project's pragmatic focus is on levels 2. and 3., but it does keep 4. and 5. in mind in every discussion and decision about how and what to put in a model. Note that levels can overlap to some extent.

The aim of RobMoSys is to cover the modeling levels with functional tooling. This include adequate code-grounding (where applicable) to actually use the models in real-systems (e.g. code-generation towards frameworks, executable components, analysis tools, etc.).

Meta-Models, Modeling Languages, and Models

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.



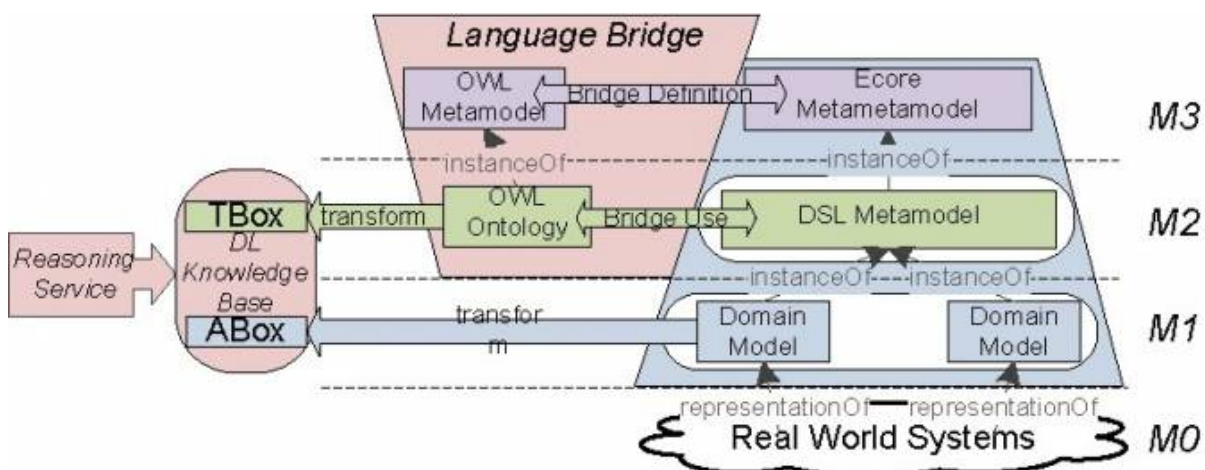
A modeller (i.e. a modeling-tool user who creates models) always works with a concrete syntax. This syntax can be textual, graphical, tabular or any combination thereof. The concrete syntax (sometimes also called notation) is defined by (i.e. it conforms to) the **modeling language**. The concrete syntax 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 Frequently Asked Questions.

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 \[http://twouse.blogspot.de/2010/08/owl-ecore-language-bridges.html\]](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: 2019/05/20 10:47
<http://www.robmosys.eu/wiki/modeling:principles>

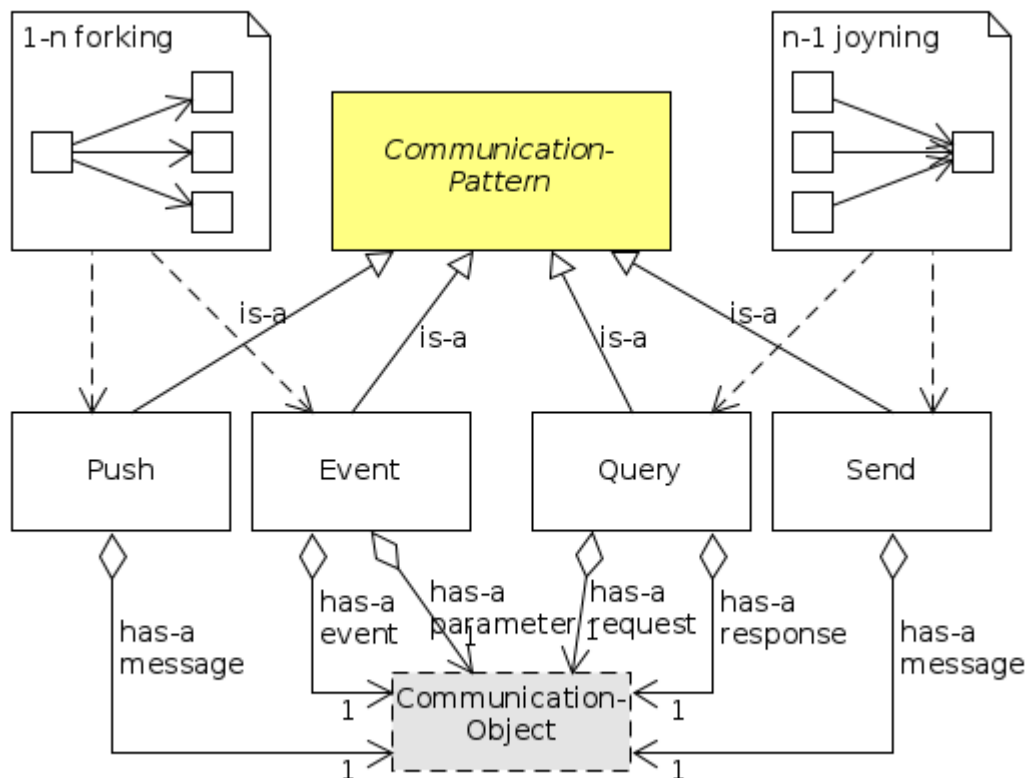
Communication-Pattern Metamodel

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.

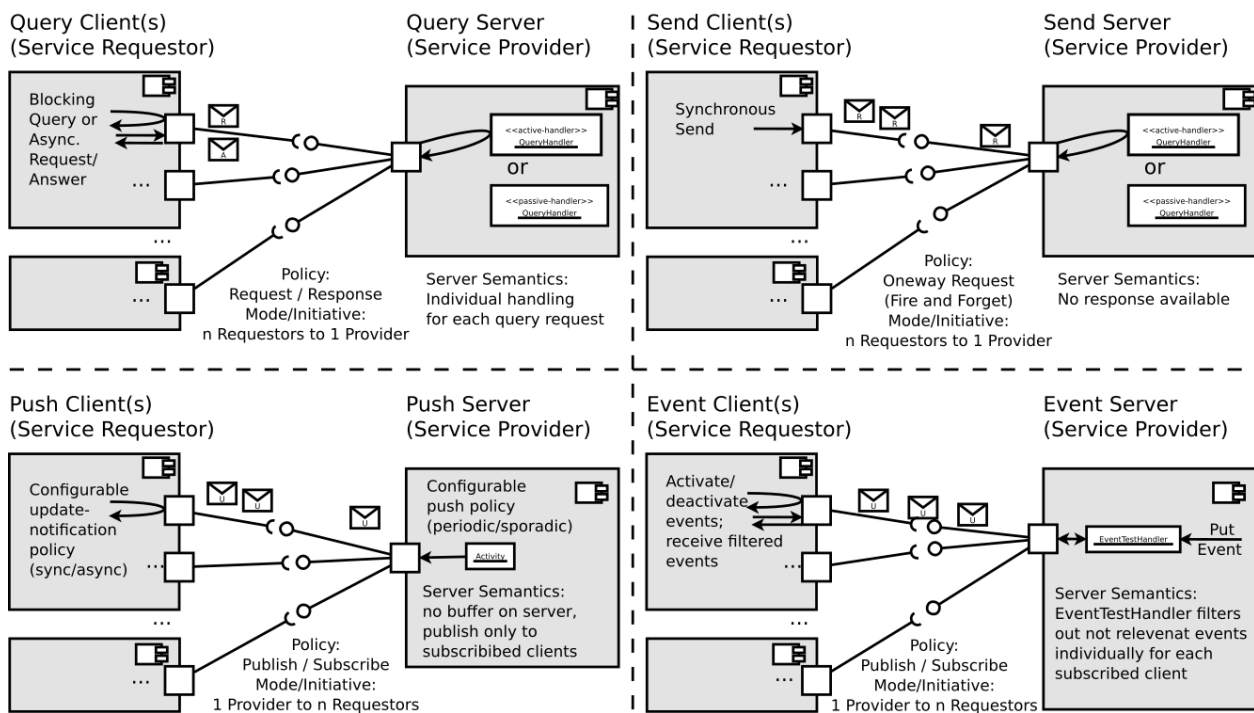


Component Communication Patterns

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.



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 [Schlegel2014], [Lutz2017]
State	Master/Slave	Lifecycle management and mode (de-)activation	see [Schlegel2011]

Dynamic Wiring	Master/Slave	Run-time connection re-wiring	see [Schlegel2004, p. 112]
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

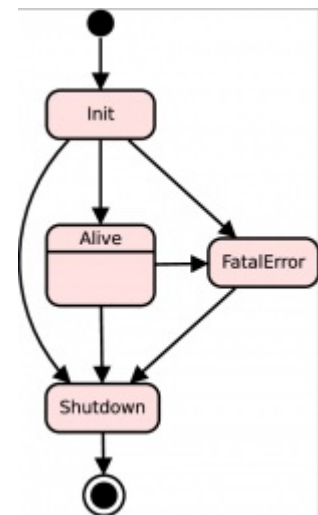
The Parameter pattern allows run-time configuration of components. The following links provide further details:

- 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, 2004. Please see References below for a link.

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

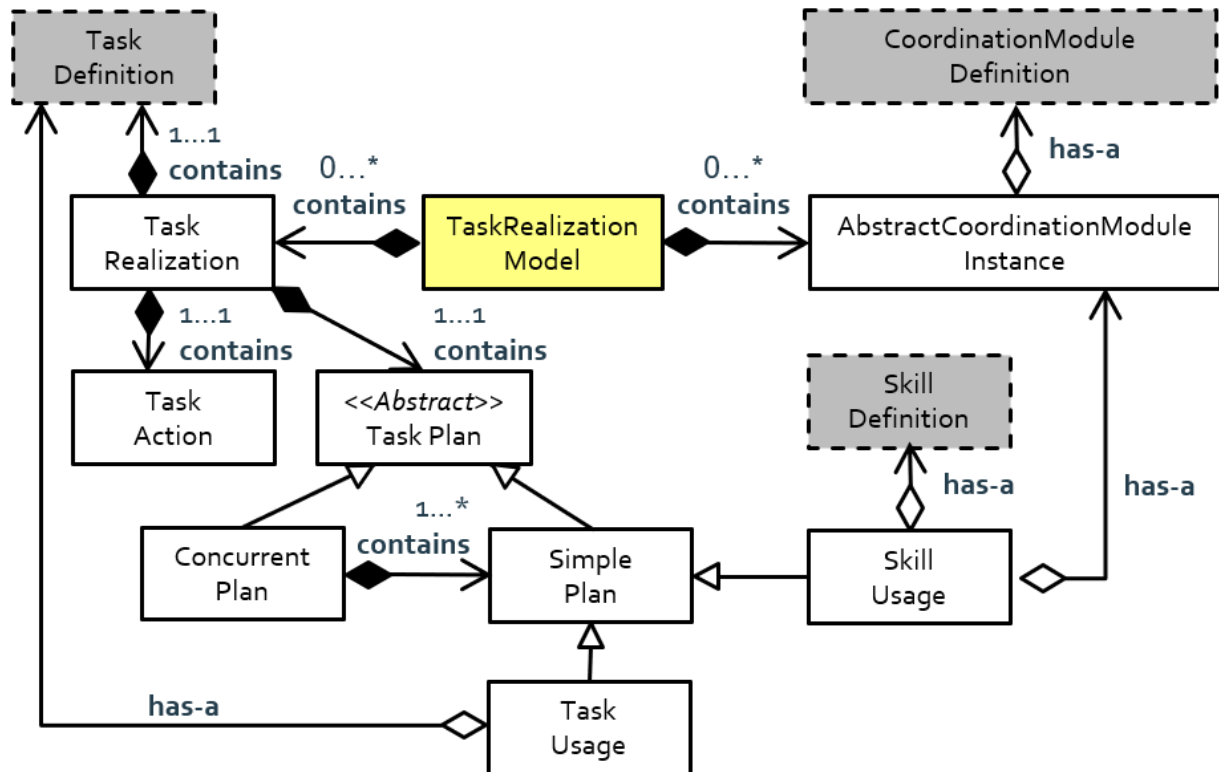
References

- [Schlegel2004] Christian Schlegel. “Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach”. *Dissertation*. University of Ulm, 2004.PDF [<http://www.servicerobotik-ulm.de/drupal/sites/default/files/phd-thesis-schlegel.pdf>]
- [Schlegel2014] C. Schlegel, D. Stampfer. “The SmartMDSD Toolchain: Supporting dynamic reconfiguration by managing variability in robotics software development”. Tutorial on Managing Software Variability in Robot Control Systems. Robotics Science and Systems Conference (RSS 2014), Berkeley, CA, July 13th 2014. Tutorial Video [<https://youtu.be/2U4KxSgwtqY>]
- [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 [https://www.researchgate.net/publication/305723618_The_SmartMDSD_Toolchain_An_Integrated_MDSD_Workflow]
- [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)

modeling:metamodels:commpattern · Last modified: 2020/12/06 22:51
<http://www.robmosys.eu/wiki/modeling:metamodels:commpattern>

Task Realization Metamodel

The Behavior Developer operates at Composition Tier 3 and can use the task vocabulary, the input-arguments, the output-arguments, and the return values as specified with their semantics by the domain experts in task definitions at Composition Tier 2.



See also

- [Architectural Pattern for Task-Plot Coordination \(Robotic Behaviors\)](#)
- [Behavior Developer](#)
- [Robotic Behavior Metamodel](#)
- [Task Definition Metamodel](#)
- [Skill Definition Metamodel](#)
- [Skill Realization Metamodel](#)

Acknowledgement

This document contains material from:

- Schlegel2021 Christian Schlegel, Alex Lotz, Matthias Lutz, Dennis Stampfer. “Composition, Separation of Roles and Model-Driven Approaches as Enabler of a Robotics Software Ecosystem”. Chapter 3 in book Software Engineering for Robotics, Ana Cavalcanti, Jon Timmis, Brijesh Dongol, Rob Hierons, Jim Woodcock (editors), Springer Nature Switzerland, (to appear as Open Access)
- Schlegel2020 Christian Schlegel, Dennis Stampfer, Alex Lotz, Matthias Lutz, “Robot Programming”.

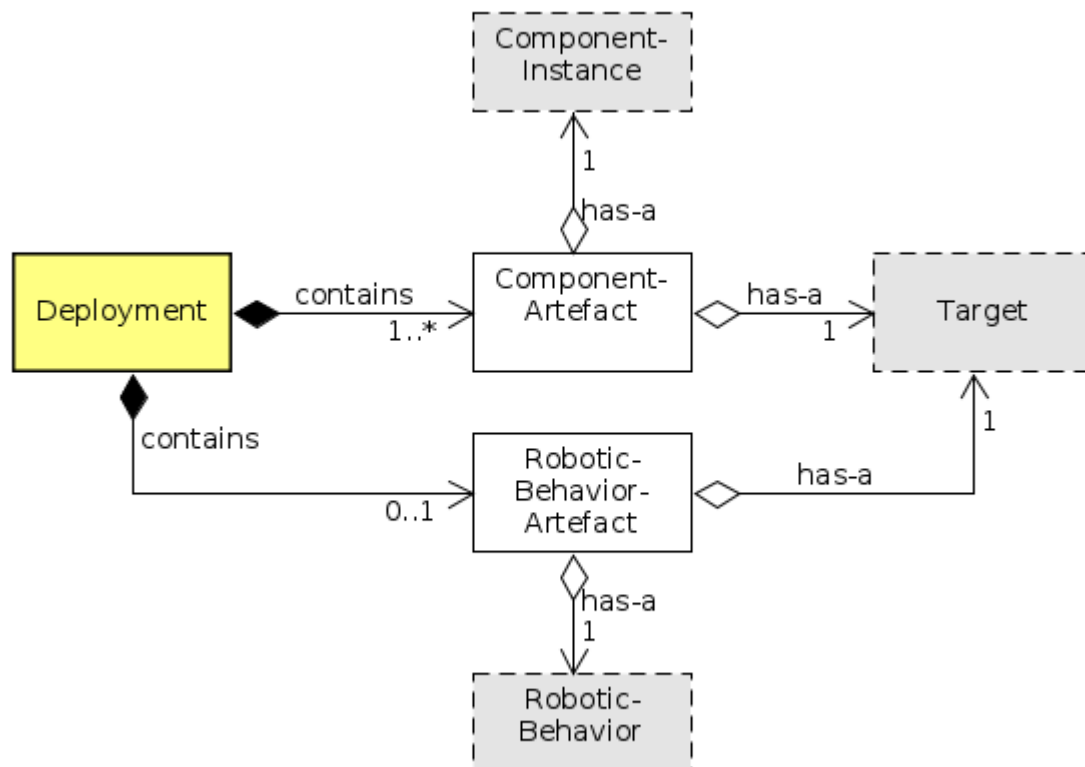
Chapter 8 in book Mechatronics and Robotics: New Trends and Challenges, Marina Indri and Roberto Oboe (editors), CRC Press.

- Lutz2017 Matthias Lutz, “Model-Driven Behavior Development for Service Robotic Systems: Bridging the Gap between Software- and Behavior-Models,” 2017. (unpublished work)

modeling:metamodels:task-realization · Last modified: 2020/12/04 17:31
<http://www.robmosys.eu/wiki/modeling:metamodels:task-realization>

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.

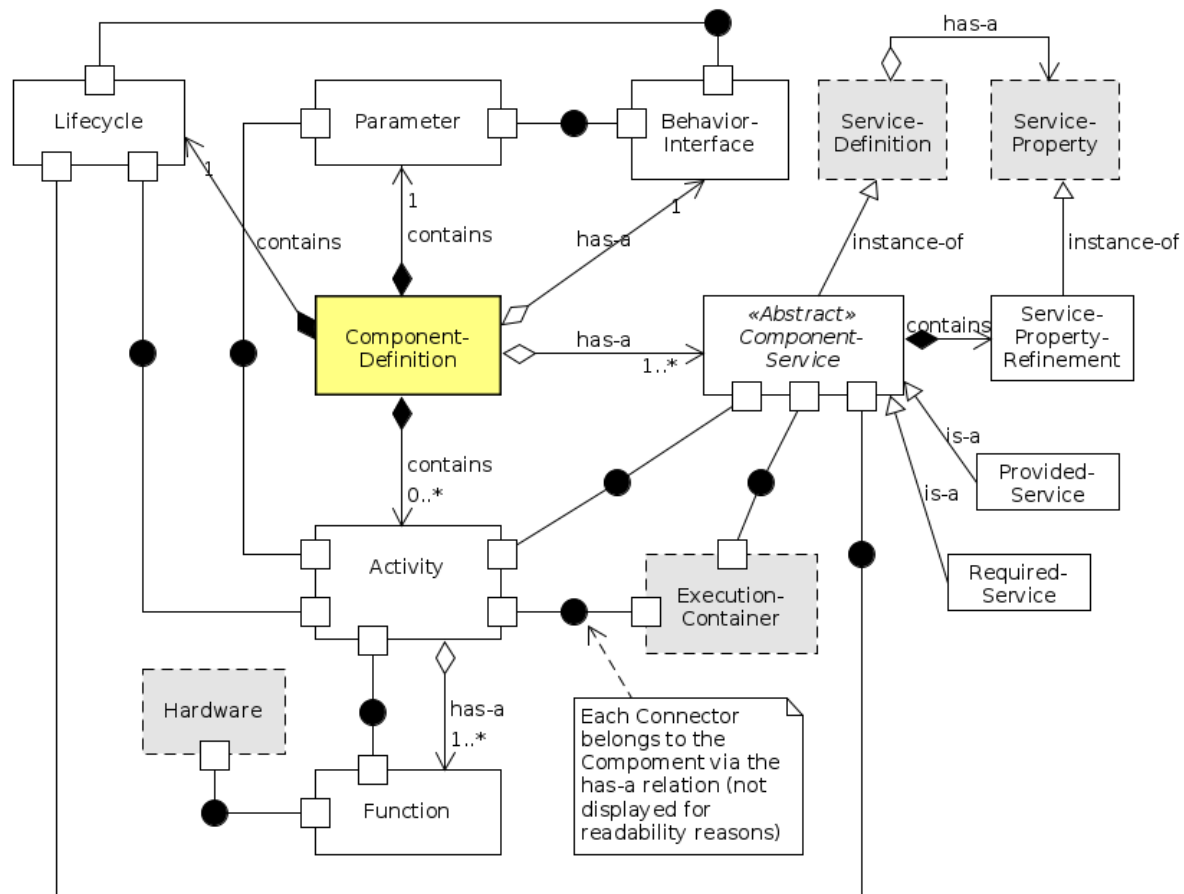
See also:

- [Platform Metamodel](#)
- [System Component Architecture Metamodel](#)
- [Robotic Behavior Metamodel](#)

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 JOSER2016¹) 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 TR2011² 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 SIMPAR2016³⁾ 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 (JOSE 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 (SIMPAN 2016)*. San Francisco, CA, USA, 2016.DOI [<https://doi.org/10.1109/SIMPAN.2016.7862392>]

modeling:metamodels:component · Last modified: 2019/05/20 10:49
<http://www.robmosys.eu/wiki/modeling:metamodels:component>

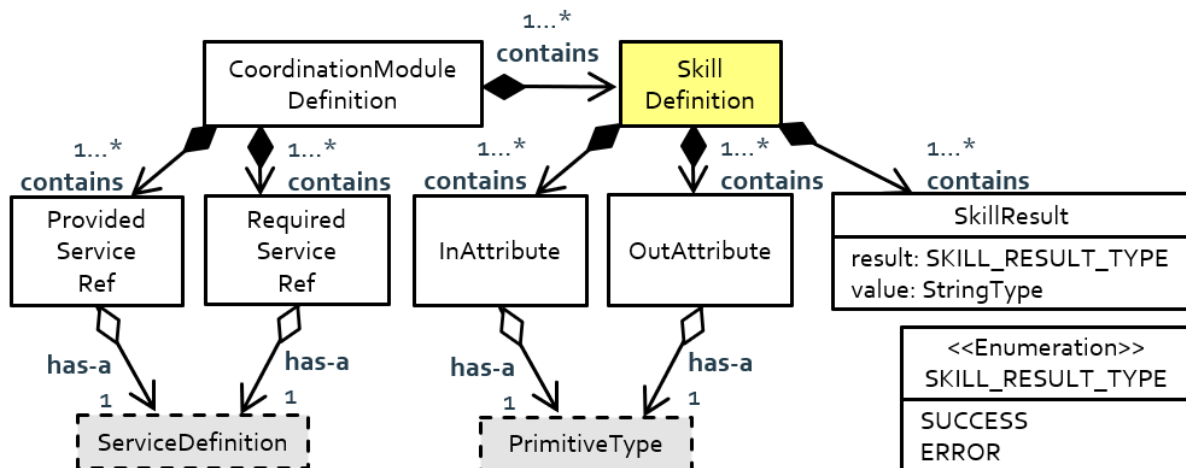
Skill Definition Metamodel

A **skill** provides access to functionalities realized by components for robotics behavior development. Skills lift the level of abstraction from functional and service to a skill abstraction level, being usable from a task abstraction level for robotics behavior development.

Skill definitions define the interface of skills on Composition Tier 2. Skills are realized at Composition Tier 3 next to the component providing the required functionality, following a skill definition. Separating skill definitions and skill realizations enables the replacement and the composition of components (providing the same skill) and decouples the component from the behavior developer and the technology used to realize robotics behavior development. Skill definitions and skills are grouped in sets to model their semantic cohesion.

The domain-specific skill definitions at Composition Tier 2 ensure that different components with all their specifics present their capabilities in the same way. In consequence, skills belonging to the same domain represent the same capability when they use the same name and their input / output arguments as well as their result values follow the same domain-specific vocabulary and meaning.

A skill definition as the interface of a skill contains input and output attributes of a primitive type (int, bool , etc.) and return a result (SkillResult). The sum of all contained SkillResults models the possible results a skill could evaluate to. Each SkillResult maps to either *SUCCESS* or *ERROR* and contains an additional value to provide further result details (e.g. *ERROR* and “*Path Blocked*” for a *moveto* Skill).



See also

- [Separation of Levels and Separation of Concerns](#)
- [Skill Realization Metamodel](#)
- [Component-Definition Metamodel](#)
- [Behavior Developer](#)
- [Component Supplier](#)

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>]

modeling:metamodels:skill-definition · Last modified: 2020/12/04 17:13
<http://www.robmosys.eu/wiki/modeling:metamodels:skill-definition>

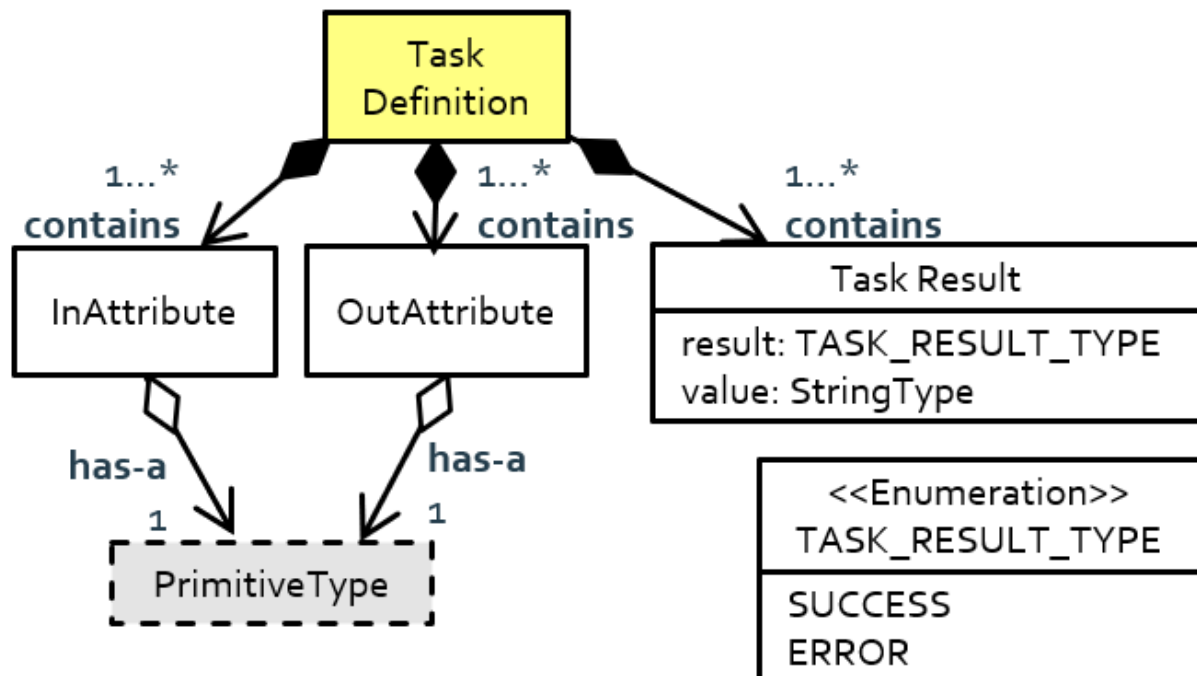
Task Definition Metamodel

Tasks describe via which steps (*what*: the ordering of steps) and in which manner (*how*: the kind of execution) to accomplish a particular job. For this, a vocabulary with a semantics is needed not only for the task names but also for the in-attributes, the out-attributes, and the task results. The task definition meta-model allows domain experts at Composition Tier 2 to specify that vocabulary and its meaning for a particular domain. Tasks are realized at Composition Tier 3.

Separating task definitions and task realizations makes sure that the behavior developer uses only that vocabulary with the given semantics as has been agreed by the domain experts. This not only ensures composability of task blocks but also ensures that the robot finally shows the extended behavior.

The domain-specific task definitions at Composition Tier 2 ensure that different tasks talk about activities in the same way. In consequence, tasks belonging to the same domain talk about the same activity when they use the same name and their input / output arguments as well as their result values follow the same domain-specific vocabulary and meaning.

A task definition as the interface specification of a task contains input and output attributes of a primitive type (int, bool, etc.) and returns a result (TaskResult). The set of all contained SkillResults models the possible results a task could evaluate to. Each TaskResult maps to either *SUCCESS* or *ERROR* and contains an additional value to provide further result details (e.g. *ERROR* and “No Path” for a *transportation* task).



See also

- Architectural Pattern for Task-Plot Coordination (Robotic Behaviors)
- Robotic Behavior Metamodel
- Task Realization Metamodel

- Skill Definition Metamodel
- Skill Realization Metamodel

Acknowledgement

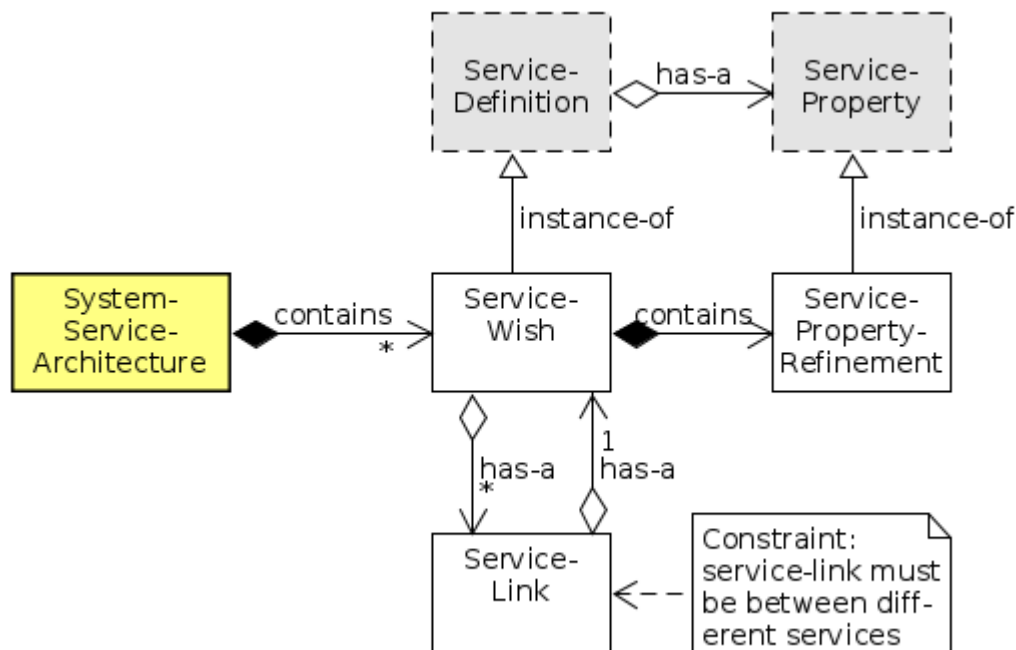
This document contains material from:

- Schlegel2021 Christian Schlegel, Alex Lotz, Matthias Lutz, Dennis Stampfer. “Composition, Separation of Roles and Model-Driven Approaches as Enabler of a Robotics Software Ecosystem”. Chapter 3 in book Software Engineering for Robotics, Ana Cavalcanti, Jon Timmis, Brijesh Dongol, Rob Hierons, Jim Woodcock (editors), Springer Nature Switzerland, (to appear as Open Access)
- Schlegel2020 Christian Schlegel, Dennis Stampfer, Alex Lotz, Matthias Lutz, “Robot Programming”. Chapter 8 in book Mechatronics and Robotics: New Trends and Challenges, Marina Indri and Roberto Oboe (editors), CRC Press.
- Lutz2017 Matthias Lutz, “Model-Driven Behavior Development for Service Robotic Systems: Bridging the Gap between Software- and Behavior-Models,” 2017. (unpublished work)

modeling:metamodels:task-definition · Last modified: 2020/12/04 17:17
<http://www.robmosys.eu/wiki/modeling:metamodels:task-definition>

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 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).



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).

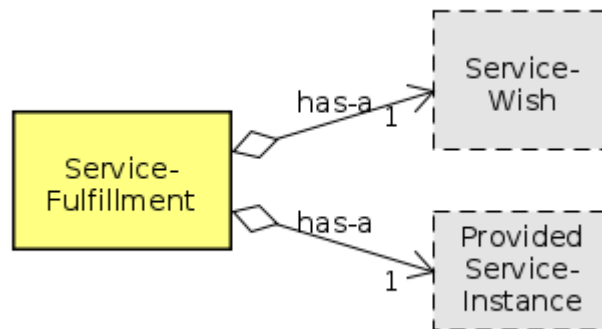
For example, a set of recurring services for a navigation stack (such as localization, mapping, path-planning, obstacle-avoidance, etc.) can be specified in advance independent of a concrete system and independent of concrete implementations in software components. In addition, it can be specified that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

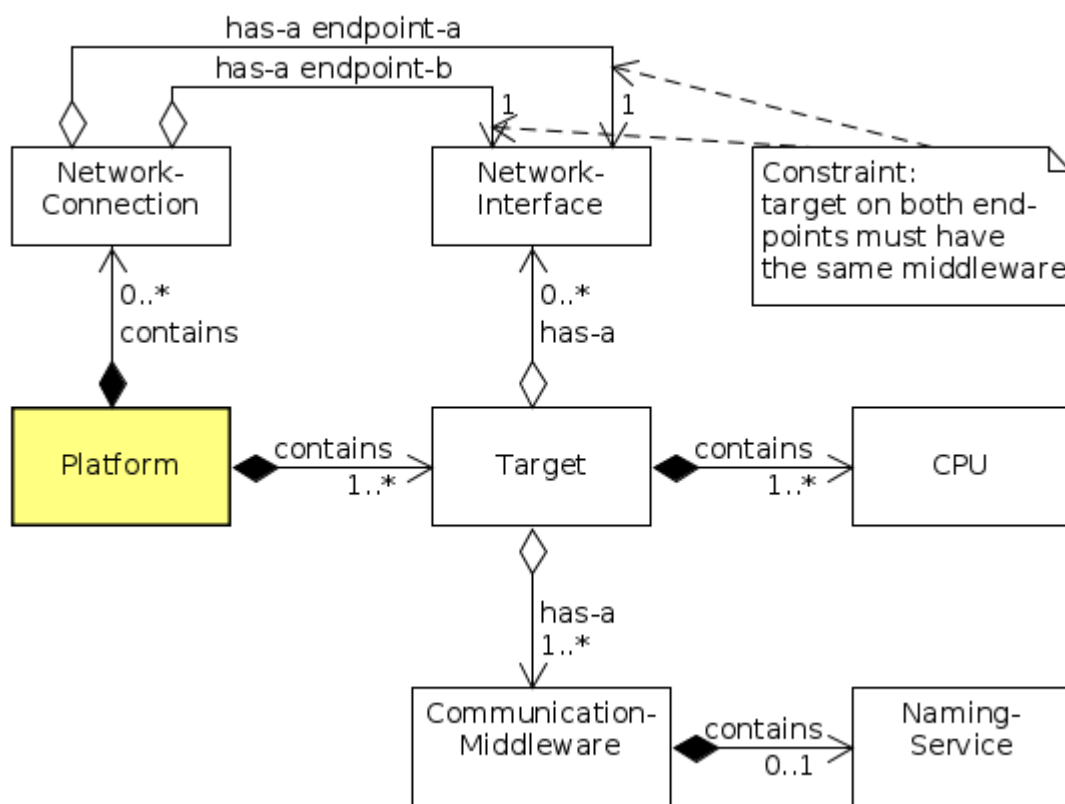
- Stampfer2018 Dennis Stampfer, "Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics". Dissertation, Technische Universität München, München, Germany, 2018. [<http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2>]
- 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: 2019/05/20 10:49
<http://www.robmosys.eu/wiki/modeling:metamodels:service-architecture>

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](#)

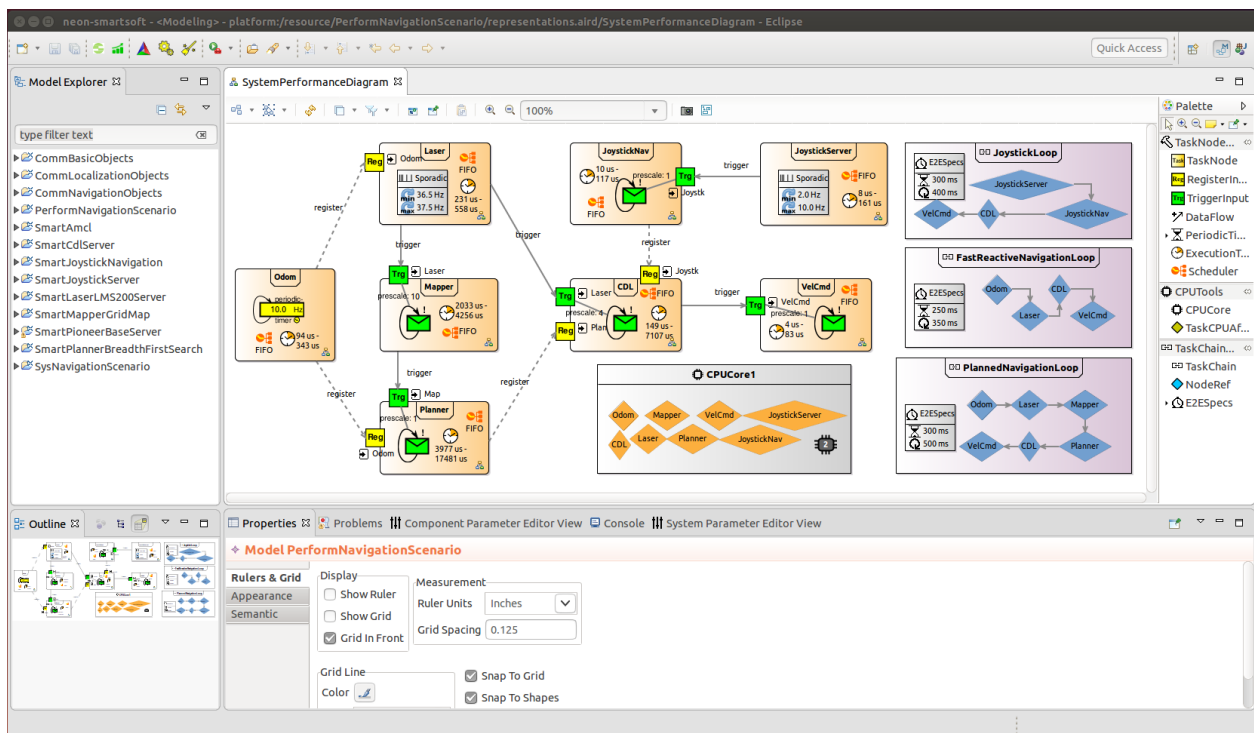
Cause-Effect-Chain and its Analysis Metamodels

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.



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:

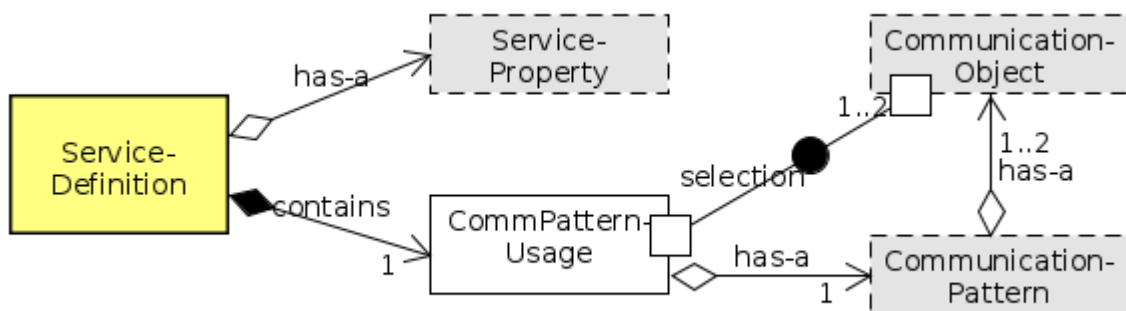
- Lotz2018 Alex Lotz, "Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System", Dissertation, Technische Universität München, München 2018. [<https://mediatum.ub.tum.de/?id=1362587>]
- Stampfer2018 Dennis Stampfer, "Contributions to System 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: 2019/05/20 10:49
<http://www.robmosys.eu/wiki/modeling:metamodels:performance>

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).

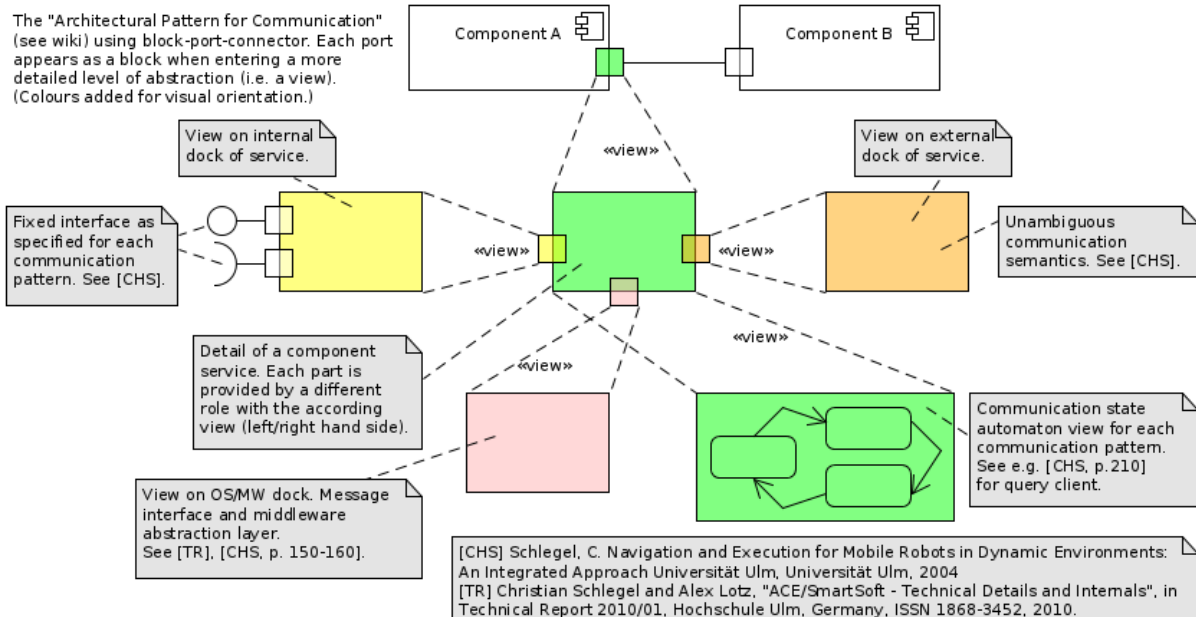


The service-definition is used as a base meta-model for component-definition and for service-architecture. The relation between these three service-related meta-models form a service triangle (see the example of a Service Triangle).

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).



References:

- Christian Schlegel. "Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach". *Dissertation*. University of Ulm, 2004.PDF [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:

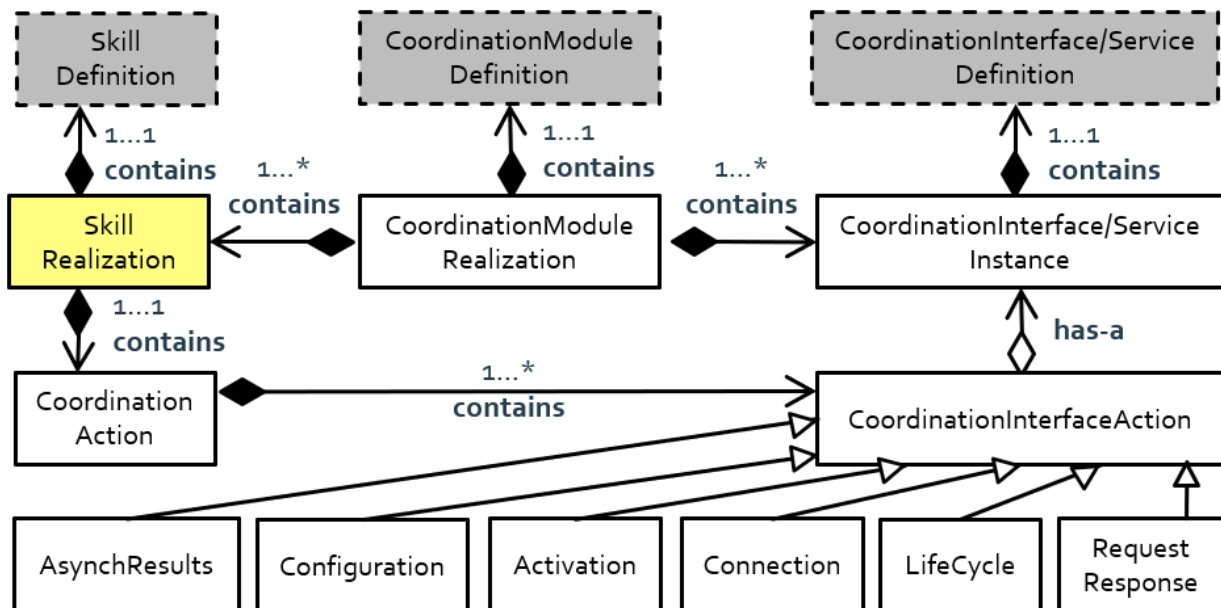
- [Component-Definition Metamodel](#)

See also:

- [Communication-Pattern Metamodel](#)
- [Communication-Object Metamodel](#)
- [Service-Based Composition \(Service Triangle\)](#)
- [Service Design View](#)

Skill Realization Metamodel

The component developer makes the functionality of a component accessible in form of skills. Thereto, the skill realization model of a component adheres to domain-specific skill definitions. This way, different implementations of skills in different (sets of) components all present their functionality and their coordination in the same domain-specific way. This is important in order to decouple the role of the component developer from the role of the behavior developer.



See also

- [Skill Definition Metamodel](#)
- [Component-Definition Metamodel](#)
- [Behavior Developer](#)
- [Component Supplier](#)
- [Task Definition Metamodel](#)
- [Task Realization Metamodel](#)

Acknowledgement

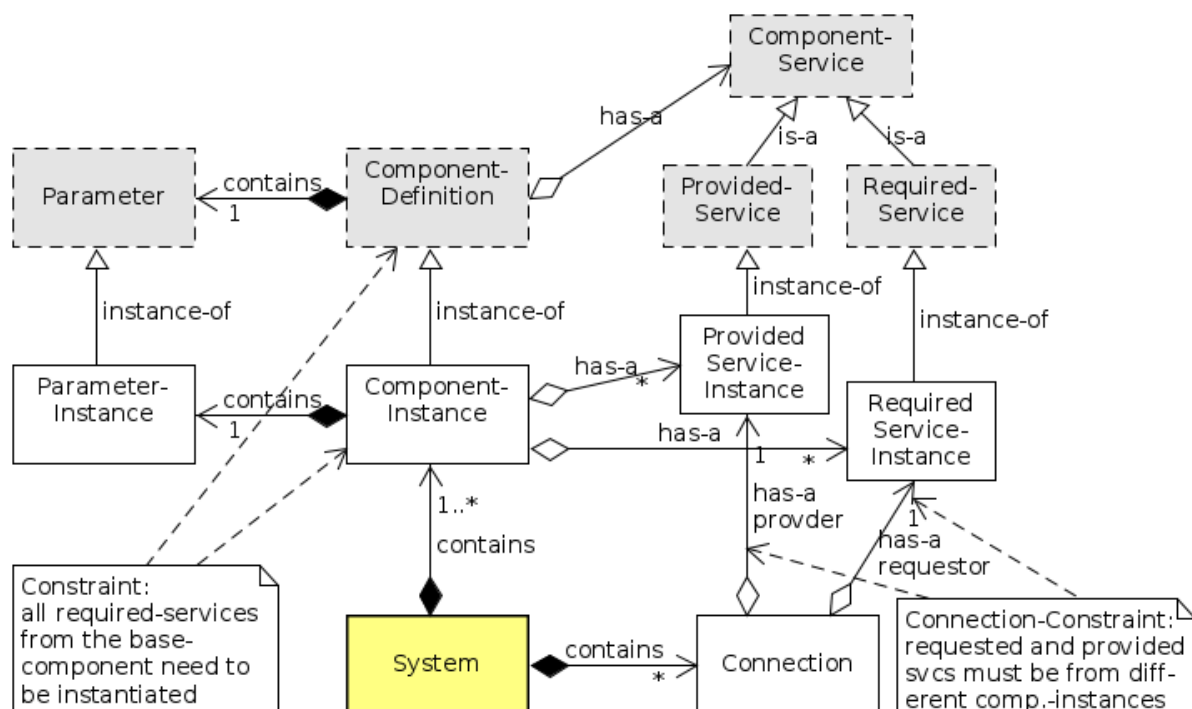
This document contains material from:

- 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->

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](#)

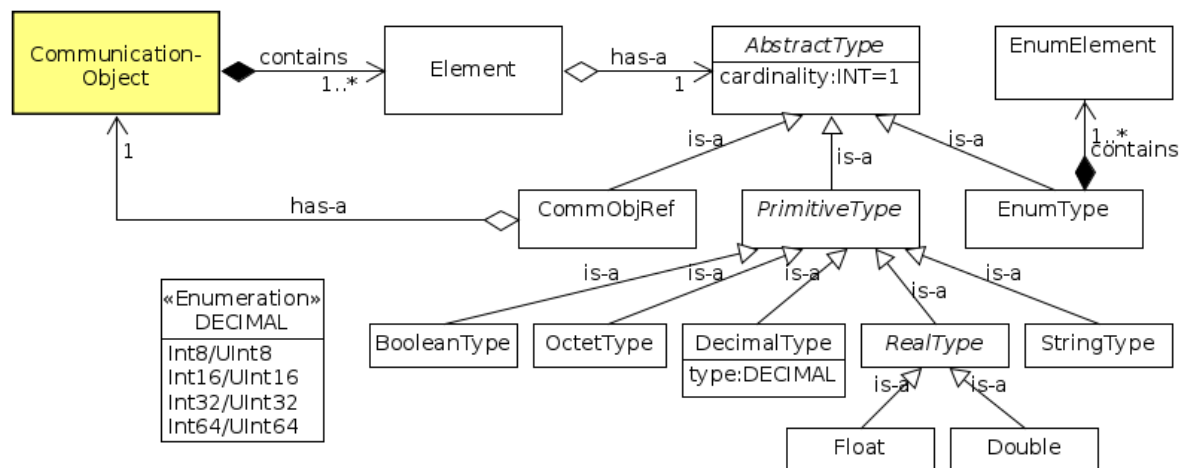
See also:

- [Component-Definition Metamodel](#)

modeling:metamodels:system · Last modified: 2019/05/20 10:49
<http://www.robmosys.eu/wiki/modeling:metamodels:system>

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 SmartMDS Toolchain: An Integrated MDS 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>]

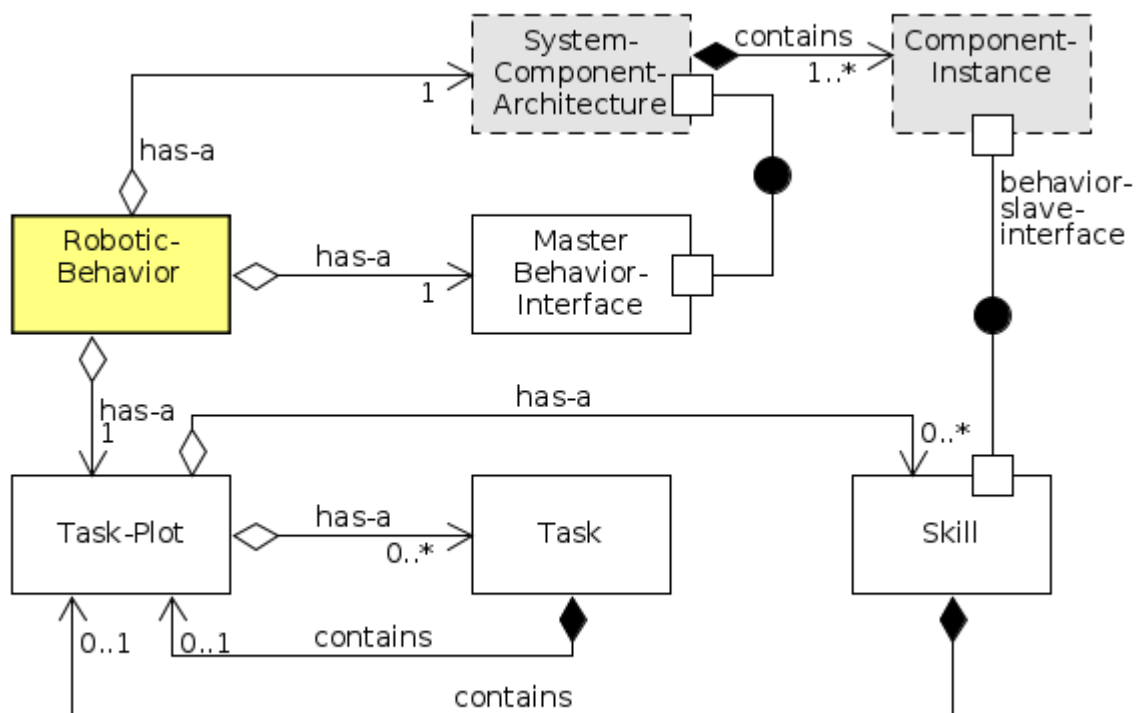
See next:

- [Communication-Pattern Metamodel](#)
- [Service-Definition Metamodel](#)

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 \[http://www.servicerobotik-ulm.de/drupal/?q=node/86\]](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 \(Robotic Behaviors\)](#)
- [Task Definition Metamodel](#)
- [Task Realization Metamodel](#)
- [Skill Definition Metamodel](#)
- [Skill Realization Metamodel](#)

modeling:metamodels:behavior · Last modified: 2020/12/04 14:31
<http://www.robmosys.eu/wiki/modeling:metamodels:behavior>

Metamodels

The RobMoSys metamodels are the [RobMoSys Composition Structures](#).

List of metamodels:

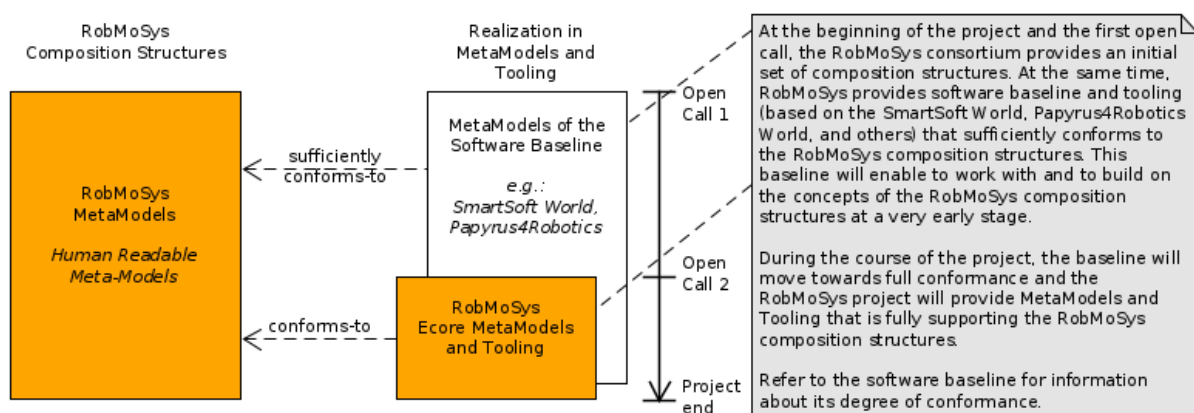
- [Robotic Behavior Metamodel](#)
- [Communication-Object Metamodel](#)
- [Communication-Pattern Metamodel](#)
- [Component-Definition Metamodel](#)
- [Deployment Metamodel](#)
- [Cause-Effect-Chain and its Analysis Metamodels](#)
- [Platform Metamodel](#)
- [System Service Architecture and Service Fulfillment Metamodels](#)
- [Service-Definition Metamodel](#)
- [Skill Definition Metamodel](#)
- [Skill Realization Metamodel](#)
- [System Component Architecture Metamodel](#)
- [Task Definition Metamodel](#)
- [Task Realization Metamodel](#)

modeling:metamodels:start · Last modified: 2019/05/20 10:49
<http://www.robmosys.eu/wiki/modeling:metamodels:start>

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 from 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.



Sustainability after the Project Lifetime

- Stewardship of the euRobotics Topic Group "Software Engineering, Systems Integration and Systems Engineering" [<https://sparc-robotics-portal.eu/web/software-engineering/stewardship-software-engineering-systems-integration-and-systems-engineering/>]
- SmartMDSD as Open Source Eclipse Project [<https://projects.eclipse.org/projects/modeling.smartmdsd>]
- Papyrus4Robotics as Open Source Eclipse Project [<https://www.eclipse.org/papyrus/components/robotics/>]
- XITO [<https://www.xito.one/>]: The marketplace and application building platform for robotics based on RobMoSys technology

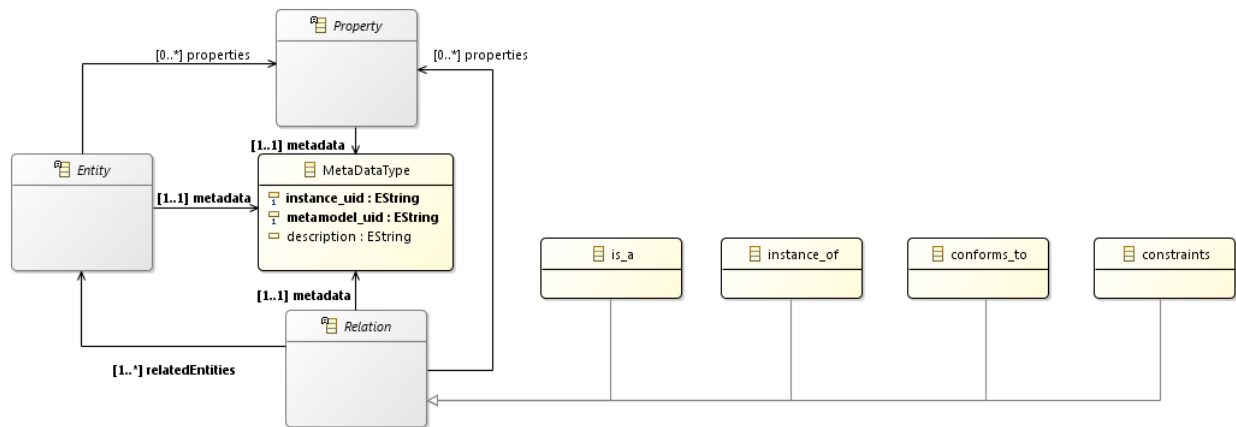
See also

- The given description also holds true for the [Roadmap of Tools and Software](#)
- [Conformance of SmartMARS Metamodel to RobMoSys composition structures](#)
- [EU Digital Industrial Platform for Robotics](#)

Preliminary Ecore implementation of ER and BPC meta-models

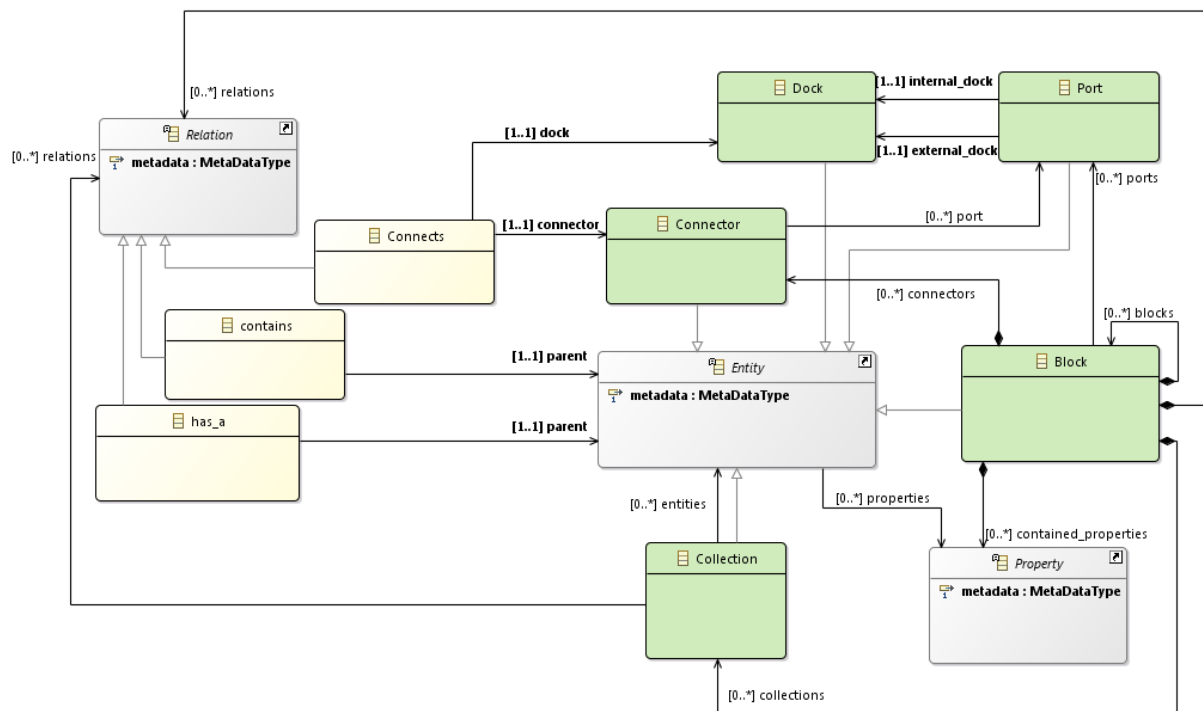
Entity-Relation (ER) meta-model

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:

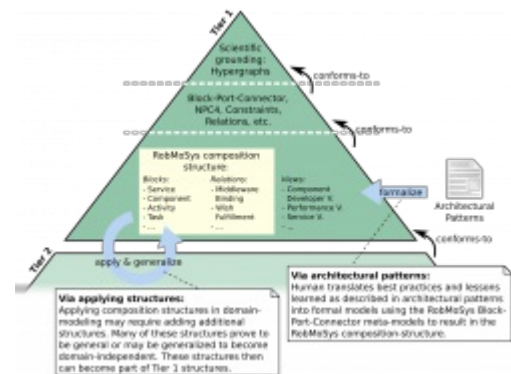
1. Install Eclipse Neon Modeling [<http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/neon3>].
2. Import the plugins in your workspace.

modeling:realization_alternatives:ecore_implement · Last modified: 2019/05/20 10:49
http://www.robmosys.eu/wiki/modeling:realization_alternatives:ecore_implement

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.

- [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](#)

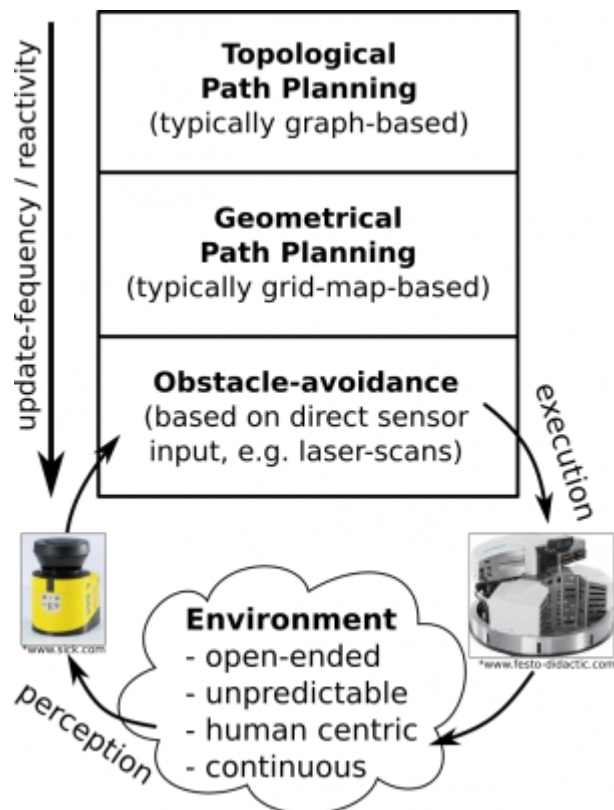


modeling:start · Last modified: 2019/05/20 10:47
<http://www.robmosys.eu/wiki/modeling:start>

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).



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 planning 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

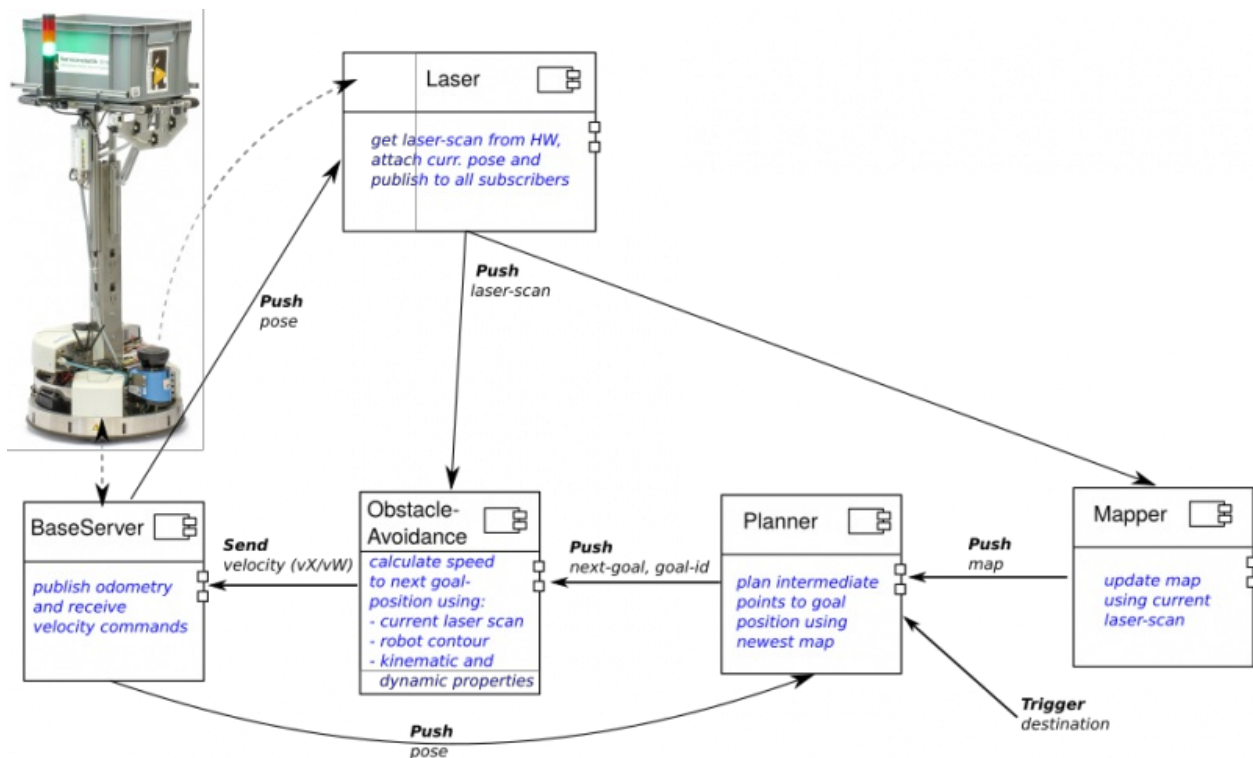
In some cases, even the intermediate level is not sufficient. For instance, if a robot needs to navigate in an entire building consisting of several floors, maybe connected over elevators, then building a single huge grid map becomes complicated, too inefficient and too resource consuming. In these cases, 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 navigation¹⁾ 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 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.



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 **sensor** 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., odometry) 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 location²⁾ 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 component 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 odometry (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 **ObstacleAvoidance** 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 (see [Robotic 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>].

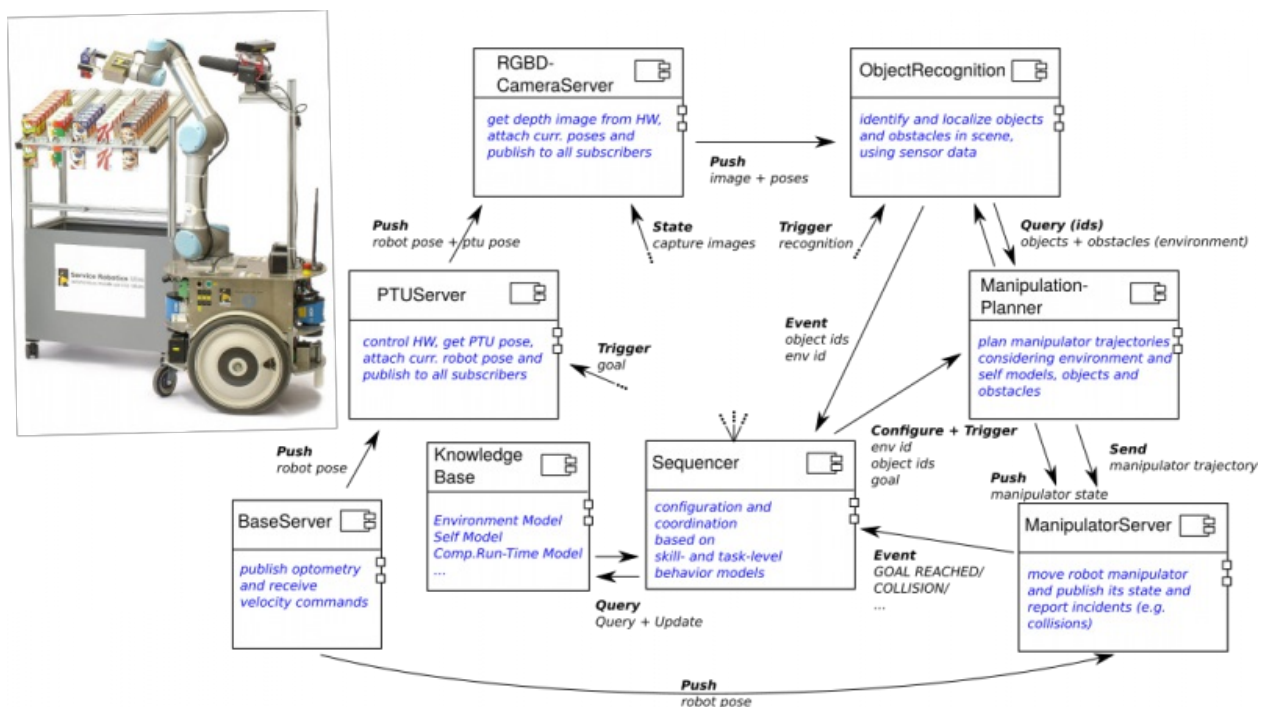
domain_models:navigation-stack:start · Last modified: 2019/05/20 10:49
http://www.robmosys.eu/wiki/domain_models:navigation-stack:start

Mobile Manipulation Stack

The mobile manipulation stack is a set of components that realize specific services to provide a flexibly applicable mobile manipulation capability for a service robot. The services defined ([service definitions](#)) for the mobile manipulation stack are a typical example of the [Composition Tier 2](#) contents of the RobMoSys Ecosystem. This mobile manipulation stack can be used with various robot platforms and different kinds of sensors. Moreover, it is able to deal with unstructured environments of variable scale. The focus hereinafter is to emphasize the general design choices and architectural decisions of the mobile manipulation components.

The mobile manipulation stack components and services

The figure below illustrates the interaction of the mobile manipulation components over generic mobile manipulation services. While those services are always stable, there are several alternatives for each of the 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 mobile manipulation stack flexibly usable in different applications with different requirements with respect to envisioned environments and the used manipulators and robot platforms.



The mobile manipulation stack consists of the following hardware-related components, namely **BaseServer**, **PTU Server**, **RGBD Camera Server** and **Manipulator Server**. These components abstract away the used hardware. While the components themselves are specific to a particular platform (e.g. Robotino3, UR, etc.), they implement the platform-independent [services](#).

The **Manipulation Planner** and the **Object Recognition** component implement different capabilities of the stack, namely (1) recognition of objects and environments and (2) collision free manipulation planning. Again,

similar to the two hardware-related components above, the components internally implement a specific algorithm and are exchangeable due to the algorithm-independent service definitions that they individually implement.

Overall, the above component realize different capabilities providing skills to be used for coordination on above abstraction levels. The **Sequencer** component is the coordinating component executing the skill- and task- level behavior models. The sequencer uses the **KnowledgeBase** component to memorize necessary information, states and models, e.g. environment model, self model etc. Further information regarding coordination for mobile manipulation and object recognition, as well as distributed knowledge representation can be found in ¹⁾.

RobMoSys Modeling Support

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

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

RobMoSys Tooling Support

- The concrete models of the mobile manipulation stack are modeled and realized using the SmartMDSD Toolchain.
- see RobMoSys Model Directory

¹⁾

Matthias Lutz, Dennis Stampfer, Alex Lotz, Christian Schlegel. Service Robot Control Architectures for Flexible and Robust Real-World Task Execution: Best Practices and Patterns. Workshop Roboter-Kontrollarchitekturen, Informatik 2014, Springer LNI der GI, ISBN 978-3-88579-626-8, Stuttgart, September 2014. Link [<https://dl.gi.de/handle/20.500.12116/2738>]

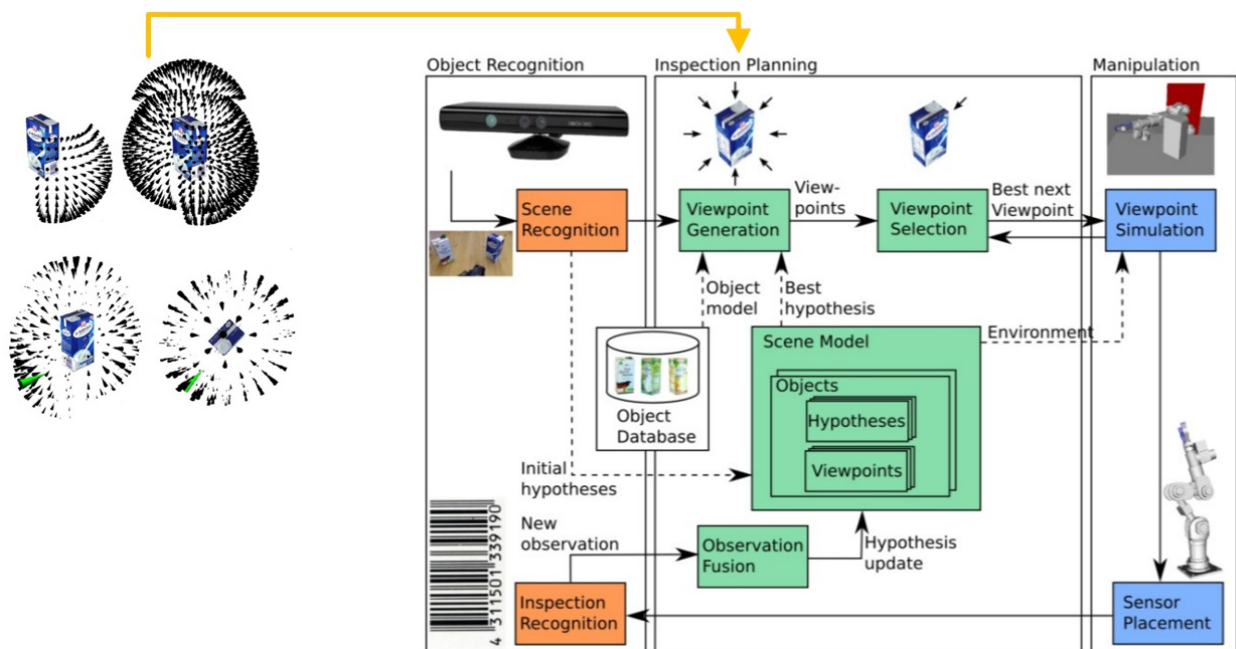
domain_models:mobile-manipulation-stack:start · Last modified: 2019/05/20 10:49
http://www.robmosys.eu/wiki/domain_models:mobile-manipulation-stack:start

Active Object Recognition Stack

The active object recognition stack is a set of components that realize specific services to provide a flexibly applicable object recognition capability for a service robot. It reuses many parts of the mobile manipulation stack. It combines object recognition and manipulation such that out of possible viewpoints of a manipulator-mounted camera the viewpoint with the best information gain for the given task can be determined. It then places the camera accordingly.

The active object recognition stack components and services

This is an overview on the mechanisms of the active object recognition stack.



Explanations to be added:

- Tier 2 services for this stack beyond mobile manipulation services
- Tier 3 Inspection Planning Component

Additional Material

- see the mobile manipulation stack for the baseline reused in this stack
- see D. Stampfer, M. Lutz and C. Schlegel, "Information driven sensor placement for robust active object recognition based on multiple views," 2012 IEEE International Conference on Technologies for Practical Robot Applications (TePRA), Woburn, MA, 2012, pp. 133-138, doi: [10.1109/TePRA.2012.6215667](https://doi.org/10.1109/TePRA.2012.6215667) [<https://ieeexplore.ieee.org/document/6215667>] for the baseline implementation of the required services and components

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.

- [Flexible Navigation Stack](#)
- [Mobile Manipulation Stack](#)
- [Motion, Perception, Worldmodel Stack](#)
- [Active Object Recognition Stack](#)
- See also the [RobMoSys Model Directory](#)



domain_models:start · Last modified: 2020/12/09 20:35
http://www.robmosys.eu/wiki/domain_models:start

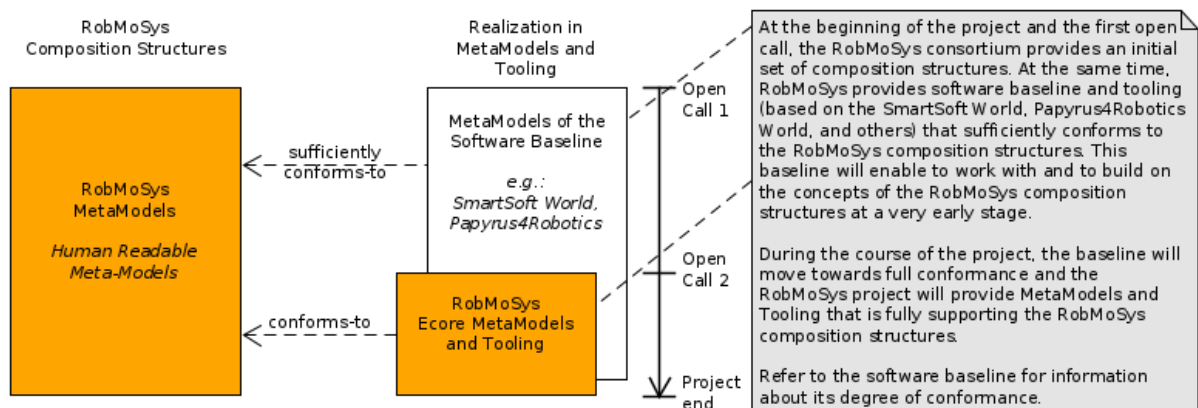


This page has moved to [RobMoSys Model Directory](#)

baseline:components:smartsoft · Last modified: 2019/05/20 10:49
<http://www.robmosys.eu/wiki/baseline:components:smartsoft>

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.



Sustainability after the Project Lifetime

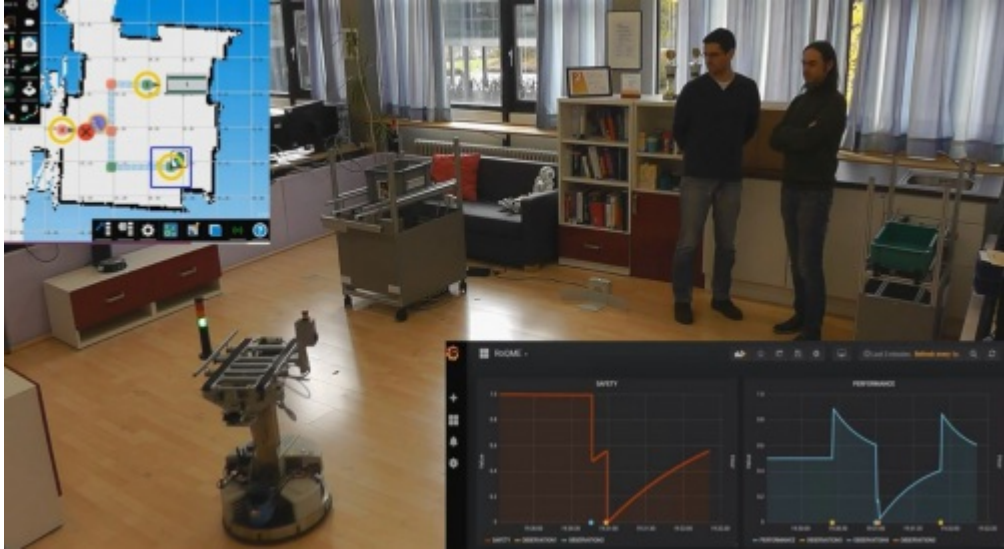
- Stewardship of the euRobotics Topic Group "Software Engineering, Systems Integration and Systems Engineering" [<https://sparc-robotics-portal.eu/web/software-engineering/stewardship-software-engineering-systems-integration-and-systems-engineering/>]
- SmartMDSD as Open Source Eclipse Project [<https://projects.eclipse.org/projects/modeling.smartmdsd>]
- Papyrus4Robotics as Open Source Eclipse Project [<https://www.eclipse.org/papyrus/components/robotics/>]
- XITO [<https://www.xito.one/>]: The marketplace and application building platform for robotics based on RobMoSys technology

See also

- EU Digital Industrial Platform for Robotics
- Roadmap of MetaModeling
- Conformance of SmartSoft to RobMoSys composition structures

RoQME Plugins for the SmartMDSD Toolchain

The RoQME Toolchain enables the modeling (at design-time) of QoS metrics defined on system-level non-functional properties (e.g., safety, performance, reliability, etc.) and their estimation (at runtime) based on the contextual information then available.

Authors	Cristina Vicente-Chicote, Universidad de Extremadura (Spain) Pablo García-Ojeda, Universidad de Extremadura (Spain) Daniel García-Pérez, Universidad de Extremadura (Spain) Jesús Martínez, Universidad de Málaga (Spain) Adrián Romero-Garcés, Universidad de Málaga (Spain) Juan F. Inglés-Romero, Biometric Vox, S.L. (Spain)
Website	https://robmosys.eu/roqme/ [https://robmosys.eu/roqme/]
License	GNU General Public License, version 3.0 or later (GPLv3+)
Screenshot	 <p>Screenshot taken from the video entitled “Dealing with Metrics on Non-Functional Properties in RobMoSys”, available at: https://www.youtube.com/watch?v=fb1uLT5CNjg [https://www.youtube.com/watch?v=fb1uLT5CNjg]</p>

Description

The RoQME Toolchain aims at allowing RobMoSys Domain Experts, QoS Engineers and System Builders to deal, both at design-time and at runtime, with system-level non-functional properties such as safety, performance, reliability, accuracy, etc. At design-time, Domain Experts and QoS Engineers can respectively model domain- and application-specific QoS metrics defined on non-functional properties relevant for their particular domain or application. It is worth noting that domain-specific RoQME models are designed for reuse, i.e., they can be imported and reused in as many application-specific RoQME models as needed.

The RoQME Toolchain also supports the generation of QoS Metrics Providers (component models conforming to the RobMoSys structures) out of the application-specific RoQME models defined by QoS Engineers. The resulting component models can be seamlessly integrated into any RobMoSys architecture, and

connected by the System Builder to any other component willing to use the QoS metrics computed at runtime. Finally, the RoQME Toolchain also generates an application-specific visualization tool that displays the evolution of both the context data and the QoS metrics computed out of them, according to the specifications gathered in the source RoQME model.

Documentation and Download

The RoQME Tool Chain is now publically available in GitHub at: <https://github.com/roqme/robmosys-roqme-itp> [<https://github.com/roqme/robmosys-roqme-itp>].

RoQME is released as a free and open source Eclipse plug-in for the SmartMDSD Tool Chain. Instructions on how to install and use RoQME can be found in the /docs folder. Additionally, two videos illustrating how to install the required software and how to create and compile a RoQME project are also available in the RoQME YouTube channel at: <https://youtu.be/iurFbIG7XtM> [<https://youtu.be/iurFbIG7XtM>] and <https://youtu.be/gFZyDYMVBtk> [<https://youtu.be/gFZyDYMVBtk>], respectively.

Additional material related with the technical demonstration realized by the RoQME team and the Ulm University of Applied Sciences, illustrating the methods and tools provided by RoQME in the context of the RobMoSys intralogistics use-case/pilot scenario can be found at: <https://robmosys.eu/wiki/community:roqme-intralogs-scenario:start> [<https://robmosys.eu/wiki/community:roqme-intralogs-scenario:start>].

Features

- Support for Domain Experts (Tier 2):
 - A textual **model editor** enabling the creation and validation of domain-specific RoQME models, i.e., models gathering QoS metrics that can be reused in the context of particular application domains (e.g., intralogistics, healthcare, household, etc.) or for particular types of robots (e.g., wheeled, legged, UGV/UMV/UAV, etc.).
- Support for QoS Experts (Tier 3):
 - A textual **model editor** enabling the creation and validation of application-specific RoQME models. These models may eventually import the RoQME models defined by Domain Experts at Tier 2.
- Support for System Builders (Tier 3):
 - A **model transformation** enabling the generation of QoS Metrics Providers (component models conforming to the RobMoSys structures) out of the RoQME models defined by QoS Engineers. The resulting component models can be seamlessly integrated into a RobMoSys architecture and connected to any component willing to use the computed metrics.
- Run-Time support:
 - Tool enabling the visualization of the QoS metrics computed at runtime.

Relation to other RobMoSys Tools

The RoQME Toolchain, delivered as an Eclipse plug-in, is intended to be used with SmartMDSD.

Further Resources

- RoQME Integrated Technical Project [<https://robmosys.eu/roqme>]
- Plugins to the SmartMDSD Toolchain at GitHub [<https://github.com/roqme/robmosys-roqme-itp>]
- **Publications:**

- M. Lutz, J.F. Inglés-Romero, D. Stampfer, A. Lotz, C. Vicente-Chicote, C. Schlegel. "Managing Variability as a Means to Promote Composability: A Robotics Perspective" [<https://www.researchgate.net/publication/328792784>], in New Perspectives on Information Systems Modeling and Design. IGI-Global, November 2018, ch. 12, pp. 274-295. DOI: 10.4018/978-1-5225-7271-8.ch012 [<http://dx.doi.org/10.4018/978-1-5225-7271-8.ch012>]
- J. M. Espín López, R. Font, J. F. Inglés-Romero, C. Vicente-Chicote. Towards the Application of Global Quality-of-Service Metrics in Biometric Systems [<https://www.researchgate.net/publication/328890945>]. Proc. IberSPEECH 2018. Barcelona (Spain), 21-23 November 2018. DOI: 10.21437/IberSPEECH.2018 [<http://dx.doi.org/10.21437/IberSPEECH.2018>]
- J.F. Inglés-Romero, J.M. Espín, R. Jiménez, R. Font, C. Vicente-Chicote. Towards the Use of Quality of Service Metrics in Reinforcement Learning [https://www.researchgate.net/publication/327243001_Towards_the_Use_of_Quality-of-Service_Metrics_in_Reinforcement_Learning_A_Robotics_Example]: A Robotics Example. Proc. 5th International Workshop on Model-driven Robot Software Engineering (MORSE'18), in conjunction with MODELS 2018. Copenhagen (Denmark), 15 October 2018.
- C. Vicente-Chicote, J.F. Inglés-Romero, J. Martínez, D. Stampfer, A. Lotz, M. Lutz, C. Schlegel. A Component-Based and Model-Driven Approach to Deal with Non-Functional Properties through Global QoS Metrics [https://www.researchgate.net/publication/328102310_A_Component-Based_and_Model-Driven_Approach_to_Deal_with_Non-Functional_Properties_through_Global_QoS_Metrics]. Proc. 5th International Workshop on Interplay of Model-Driven and Component-Based Software Engineering (ModComp'18), in conjunction with MODELS 2018. Copenhagen (Denmark), 14 October 2018.
- C. Vicente-Chicote, J. Berrocal, J. García-Alonso, J. Hernández, A. J. Bandera, J. Martínez, A. Romero-Garcés, R. Font and J.F. Inglés-Romero. RoQME: Dealing with Non-Functional Properties through Global Robot QoS Metrics [<https://www.researchgate.net/publication/327239527>]. XXIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'18), Sevilla (España), 17-19 September, 2018.
- **RoQME in action:** QoS metrics in an intralogistics scenario (video)
 - <https://www.youtube.com/watch?v=fb1uLT5CNjg&feature=youtu.be> [<https://www.youtube.com/watch?v=fb1uLT5CNjg&feature=youtu.be>]
- **Social Networks:**
 - ResearchGate Project: <https://www.researchgate.net/project/RoQME-QoS-Metrics-on-NFP> [<https://www.researchgate.net/project/RoQME-QoS-Metrics-on-NFP>]
 - LinkedIn Group: <https://www.linkedin.com/groups/12096769/> [<https://www.linkedin.com/groups/12096769/>]
 - Twitter Account: https://twitter.com/RoQME_ITP [https://twitter.com/RoQME_ITP]

baseline:environment_tools:roqme-plugins · Last modified: 2019/05/29 09:08
http://www.robmosys.eu/wiki/baseline:environment_tools:roqme-plugins

eITUS Safety View for Papyrus4Robotics

The eITUS Safety View is part of the Papyrus4Robotics toolchain and organised as follows:

- Failure Modes and Effects Analysis View (FMEA)
- Safety Requirements Table View
- Fault Injection View (specification of the fault injection experiments and a separate view to show simulation results/traces).

Authors	The eITUS consortium and coaching team. Tecnalia, Akeo, CEA																																																																																																																											
Website	Part of the Papyrus4Robotics Toolchain [https://hudson.eclipse.org/papyrus/job/papyrus-robotics-2018-12/lastSuccessfulBuild/artifact/releng/org.eclipse.papyrus.robotics.p2/target/repository/plugins/]																																																																																																																											
License	https://projects.eclipse.org/license/epl-2.0 [https://projects.eclipse.org/license/epl-2.0]																																																																																																																											
Screenshot	<div><div><div>AkeoModel_Faulty2.di</div><div>AkeoModel_Golden2.di</div><div>AkeoModel_Faulty1.di</div></div><table><thead><tr><th></th><th></th><th>A</th><th>B</th><th>C</th><th>D</th></tr><tr><th></th><th></th><th>Name</th><th>ESFCore::AbstractSElement...</th><th>Causes</th><th>Local Effects</th></tr></thead><tbody><tr><td>0</td><td>No sensor/encoder measu...</td><td>No sensor/encoder measurement in joint 1 x</td><td>N/A</td><td>Random Hardware failure</td><td>No sensor value</td></tr><tr><td>1</td><td>No sensor/encoder measu...</td><td>No sensor/encoder measurement in joint 1 y</td><td>N/A</td><td>Random Hardware failure</td><td>No Sensor Value</td></tr><tr><td>2</td><td>No sensor/encoder measu...</td><td>No sensor/encoder measurement in joint 1 z</td><td>N/A</td><td>Random Hardware failure</td><td>No Sensor Value</td></tr><tr><td>3</td><td>Internal wrong position ca...</td><td>Internal wrong position calculation (stuckat0)</td><td>N/A</td><td>Software error or Random Hardware fail...</td><td>Unwanted trajectory generator</td></tr><tr><td>4</td><td>Internal wrong position ca...</td><td>Internal wrong position calculation. Too High value.</td><td>N/A</td><td>Software error or Random Hardware fail...</td><td>Unwanted trajectory generator</td></tr><tr><td>5</td><td>Internal wrong position ca...</td><td>Internal wrong position calculation. Too Low value.</td><td>N/A</td><td>Software error or Random Hardware fail...</td><td>Unwanted trajectory generator</td></tr><tr><td>6</td><td>Internal wrong position ca...</td><td>Internal wrong position calculation. Out of range value.</td><td>N/A</td><td>Software error or Random Hardware fail...</td><td>Unwanted trajectory generator</td></tr><tr><td>7</td><td>Internal wrong position ca...</td><td>Internal wrong position calculation. Early value.</td><td>N/A</td><td>Software error or Random Hardware fail...</td><td>Unwanted trajectory generator</td></tr><tr><td>8</td><td>Internal wrong position ca...</td><td>Internal wrong position calculation. Late value.</td><td>N/A</td><td>Software error or Random Hardware fail...</td><td>Unwanted trajectory generator</td></tr><tr><td>9</td><td>Wrong constants value set</td><td>Wrong constants value set</td><td>N/A</td><td>Software error</td><td>Instability torque value</td></tr><tr><td>10</td><td>Wrong output value calcul...</td><td>Wrong output value calculated</td><td>N/A</td><td>Memory failure (bit flip)</td><td>Instability or Oscillation</td></tr><tr><td>11</td><td>Wrong output value calcul...</td><td>Wrong output value calculated</td><td>N/A</td><td>External interactions (Dynamic Perturbati...</td><td>Disturbances</td></tr><tr><td>12</td><td>Motion control omission</td><td>Motion control omission</td><td>N/A</td><td>Software error or Random Hardware fail...</td><td>Unwanted motor speed</td></tr><tr><td>13</td><td>Motion control comission</td><td>Motion control comission</td><td>N/A</td><td>Software error or Random Hardware fail...</td><td>Unwanted motor speed</td></tr><tr><td>14</td><td>Early Motion control</td><td>Early Motion control</td><td>N/A</td><td>Software error or Random Hardware fail...</td><td>Unwanted motor speed</td></tr><tr><td>15</td><td>Late Motion Control</td><td>Late Motion Control</td><td>N/A</td><td>Software error or Random Hardware fail...</td><td>Unwanted motor speed</td></tr><tr><td>16</td><td>Too High Value motion co...</td><td>Too High Value motion control</td><td>N/A</td><td>Software error or Random Hardware fail...</td><td>Unwanted motor speed</td></tr><tr><td>17</td><td>Too Low Value motion con...</td><td>Too Low Value motion control</td><td>N/A</td><td>Software error or Random Hardware fail...</td><td>Unwanted motor speed</td></tr></tbody></table></div>						A	B	C	D			Name	ESFCore::AbstractSElement...	Causes	Local Effects	0	No sensor/encoder measu...	No sensor/encoder measurement in joint 1 x	N/A	Random Hardware failure	No sensor value	1	No sensor/encoder measu...	No sensor/encoder measurement in joint 1 y	N/A	Random Hardware failure	No Sensor Value	2	No sensor/encoder measu...	No sensor/encoder measurement in joint 1 z	N/A	Random Hardware failure	No Sensor Value	3	Internal wrong position ca...	Internal wrong position calculation (stuckat0)	N/A	Software error or Random Hardware fail...	Unwanted trajectory generator	4	Internal wrong position ca...	Internal wrong position calculation. Too High value.	N/A	Software error or Random Hardware fail...	Unwanted trajectory generator	5	Internal wrong position ca...	Internal wrong position calculation. Too Low value.	N/A	Software error or Random Hardware fail...	Unwanted trajectory generator	6	Internal wrong position ca...	Internal wrong position calculation. Out of range value.	N/A	Software error or Random Hardware fail...	Unwanted trajectory generator	7	Internal wrong position ca...	Internal wrong position calculation. Early value.	N/A	Software error or Random Hardware fail...	Unwanted trajectory generator	8	Internal wrong position ca...	Internal wrong position calculation. Late value.	N/A	Software error or Random Hardware fail...	Unwanted trajectory generator	9	Wrong constants value set	Wrong constants value set	N/A	Software error	Instability torque value	10	Wrong output value calcul...	Wrong output value calculated	N/A	Memory failure (bit flip)	Instability or Oscillation	11	Wrong output value calcul...	Wrong output value calculated	N/A	External interactions (Dynamic Perturbati...	Disturbances	12	Motion control omission	Motion control omission	N/A	Software error or Random Hardware fail...	Unwanted motor speed	13	Motion control comission	Motion control comission	N/A	Software error or Random Hardware fail...	Unwanted motor speed	14	Early Motion control	Early Motion control	N/A	Software error or Random Hardware fail...	Unwanted motor speed	15	Late Motion Control	Late Motion Control	N/A	Software error or Random Hardware fail...	Unwanted motor speed	16	Too High Value motion co...	Too High Value motion control	N/A	Software error or Random Hardware fail...	Unwanted motor speed	17	Too Low Value motion con...	Too Low Value motion control	N/A	Software error or Random Hardware fail...	Unwanted motor speed
		A	B	C	D																																																																																																																							
		Name	ESFCore::AbstractSElement...	Causes	Local Effects																																																																																																																							
0	No sensor/encoder measu...	No sensor/encoder measurement in joint 1 x	N/A	Random Hardware failure	No sensor value																																																																																																																							
1	No sensor/encoder measu...	No sensor/encoder measurement in joint 1 y	N/A	Random Hardware failure	No Sensor Value																																																																																																																							
2	No sensor/encoder measu...	No sensor/encoder measurement in joint 1 z	N/A	Random Hardware failure	No Sensor Value																																																																																																																							
3	Internal wrong position ca...	Internal wrong position calculation (stuckat0)	N/A	Software error or Random Hardware fail...	Unwanted trajectory generator																																																																																																																							
4	Internal wrong position ca...	Internal wrong position calculation. Too High value.	N/A	Software error or Random Hardware fail...	Unwanted trajectory generator																																																																																																																							
5	Internal wrong position ca...	Internal wrong position calculation. Too Low value.	N/A	Software error or Random Hardware fail...	Unwanted trajectory generator																																																																																																																							
6	Internal wrong position ca...	Internal wrong position calculation. Out of range value.	N/A	Software error or Random Hardware fail...	Unwanted trajectory generator																																																																																																																							
7	Internal wrong position ca...	Internal wrong position calculation. Early value.	N/A	Software error or Random Hardware fail...	Unwanted trajectory generator																																																																																																																							
8	Internal wrong position ca...	Internal wrong position calculation. Late value.	N/A	Software error or Random Hardware fail...	Unwanted trajectory generator																																																																																																																							
9	Wrong constants value set	Wrong constants value set	N/A	Software error	Instability torque value																																																																																																																							
10	Wrong output value calcul...	Wrong output value calculated	N/A	Memory failure (bit flip)	Instability or Oscillation																																																																																																																							
11	Wrong output value calcul...	Wrong output value calculated	N/A	External interactions (Dynamic Perturbati...	Disturbances																																																																																																																							
12	Motion control omission	Motion control omission	N/A	Software error or Random Hardware fail...	Unwanted motor speed																																																																																																																							
13	Motion control comission	Motion control comission	N/A	Software error or Random Hardware fail...	Unwanted motor speed																																																																																																																							
14	Early Motion control	Early Motion control	N/A	Software error or Random Hardware fail...	Unwanted motor speed																																																																																																																							
15	Late Motion Control	Late Motion Control	N/A	Software error or Random Hardware fail...	Unwanted motor speed																																																																																																																							
16	Too High Value motion co...	Too High Value motion control	N/A	Software error or Random Hardware fail...	Unwanted motor speed																																																																																																																							
17	Too Low Value motion con...	Too Low Value motion control	N/A	Software error or Random Hardware fail...	Unwanted motor speed																																																																																																																							

Description

The fault injection view can be used together with the FMEA, FTA and safety requirements view to complement or verify those analyses. Regarding the fault injection view, the eITUS framework sets up, configures, executes and analyses the simulation results. Model-based design combined with a simulation-based fault injection technique and a virtual robot poses as a promising solution for an early safety assessment of robotics systems.

The safety engineer extends the nominal or fault free behaviour of the controller by introducing saboteurs in inputs/output ports of the design. This configuration process includes the definition of fault locations (*Where to inject the fault?*), fault injection times (*When to trigger the fault?*), fault durations (*For how long the fault is present in the system?*) and the fault model (*How does the component fail?*). The original system model is modified though the fault injector script according to the fault list.

The fault list is used to produce a faulty model only in terms of reproducible and prearranged fault models. All this allows to:

- exhaustively explore all possible behaviours of a system architecture with respect to some safety property of interest (e.g. the pre-defined safety requirement “The velocity of the Robot arm must not be

greater than 0,25 m/s.”)

- simulate the behaviour of system architectures early in the development process to explore potential hazards.

The screenshot displays a software interface with a table of safety requirements and a linked failure mode. The table has columns for ID, Name, Criticality, and Failure Modes (FMEA). The failure mode 'No sensor/encoder measurement in axis x' is linked to the requirements.

	B	C	D	E
	Text	Name	Criticality	Failure Modes (FMEA)
0	SR1	the robot arm must not be greater than 0.	SR1	270
1	SR1	the robot arm must not be greater than 0.	SR1	0
2	SR1	the robot arm must not be greater than 0.	SR1	48

Failure Modes (FMEA):

- No sensor/encoder measurement in axis x
- Trajectory Generator: no position calculation
- Internal wrong position calculation (stuck at 0)

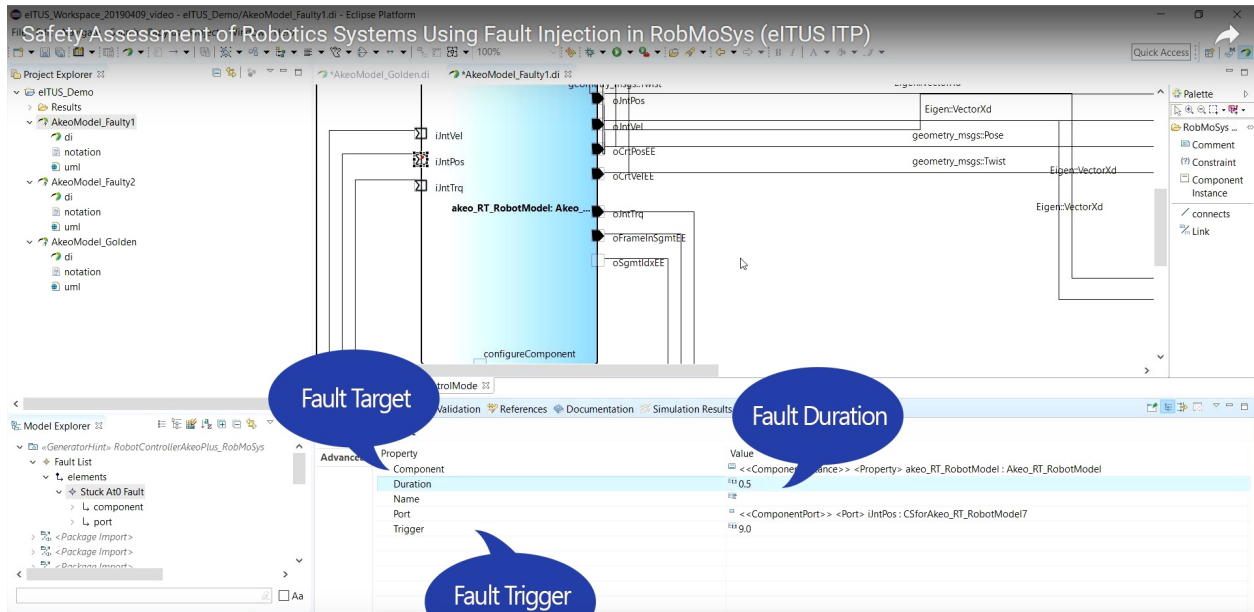
Specification of the safety requirements and link to the addressed failure mode

The screenshot displays a software interface showing the creation of a fault list. A context menu is open over a component, with options like 'New Fault Injection experiment', 'Load Fault Injection experiment', 'Save Fault Injection experiment', 'New Fault', and 'New Observation'. The 'New Fault' option is selected.

Fault Injection View: Creation of the Fault List

The screenshot displays a software interface showing the Fault Model. A blue speech bubble labeled 'Fault Model' points to a component. A context menu is open over the component, with options like 'Choose fault type', 'StuckAt0Fault', 'StuckAtLastValueFault', 'StuckAtValueFault', 'InvertedFault', 'TooHighFault', 'TooLowFault', 'RampUpFault', 'RampDownFault', 'BitFlipFault', 'OscillationFault', 'RandomFault', and 'DelayFault'.

Fault Injection View: List of Fault Models



Fault Injection View: configuration of a fault

Features

The safety view can be used to:

- Create modular FMEAs, link them to the system model and define traceable safety requirements.
- Specify the fault-injection experiments
- Generate a Faulty version of the controller and run it in simulation.
- Visualize and analyse the resulting simulation traces to refine and validate the safety of the system.

Relation to other RobMoSys Tools

Related to the Fault Tree Analysis View (FTA) available in the Papyrus4Robotics toolchain.

Further Resources

- eITUS Integrated Technical Project [<https://robmosys.eu/e-itus/>]
- Showcase: <https://robmosys.eu/wiki/community:safety-analysis:start>
[<https://robmosys.eu/wiki/community:safety-analysis:start>]

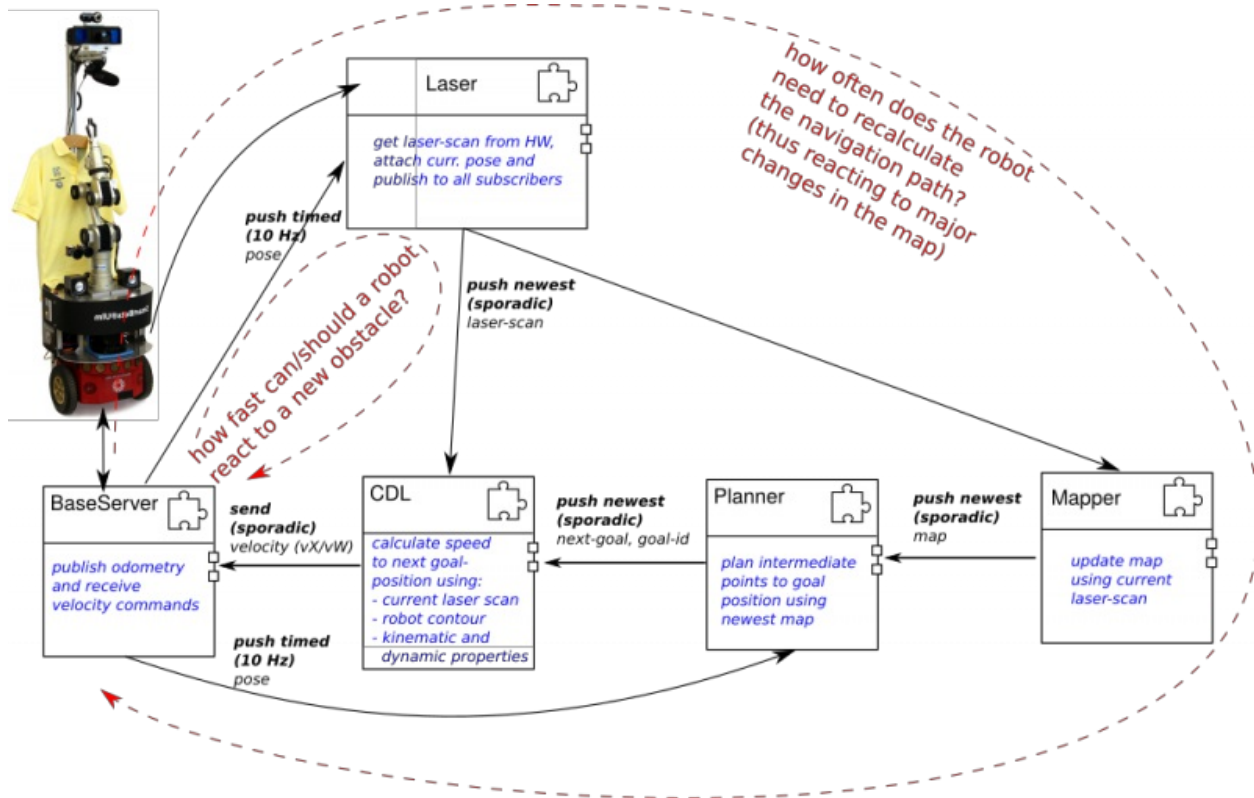
baseline:environment_tools:eitus-safety · Last modified: 2019/07/12 11:58
http://www.robmosys.eu/wiki/baseline:environment_tools:eitus-safety

Support for Managing Cause-Effect Chains in Component Composition

This page uses the SmartMDS 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.

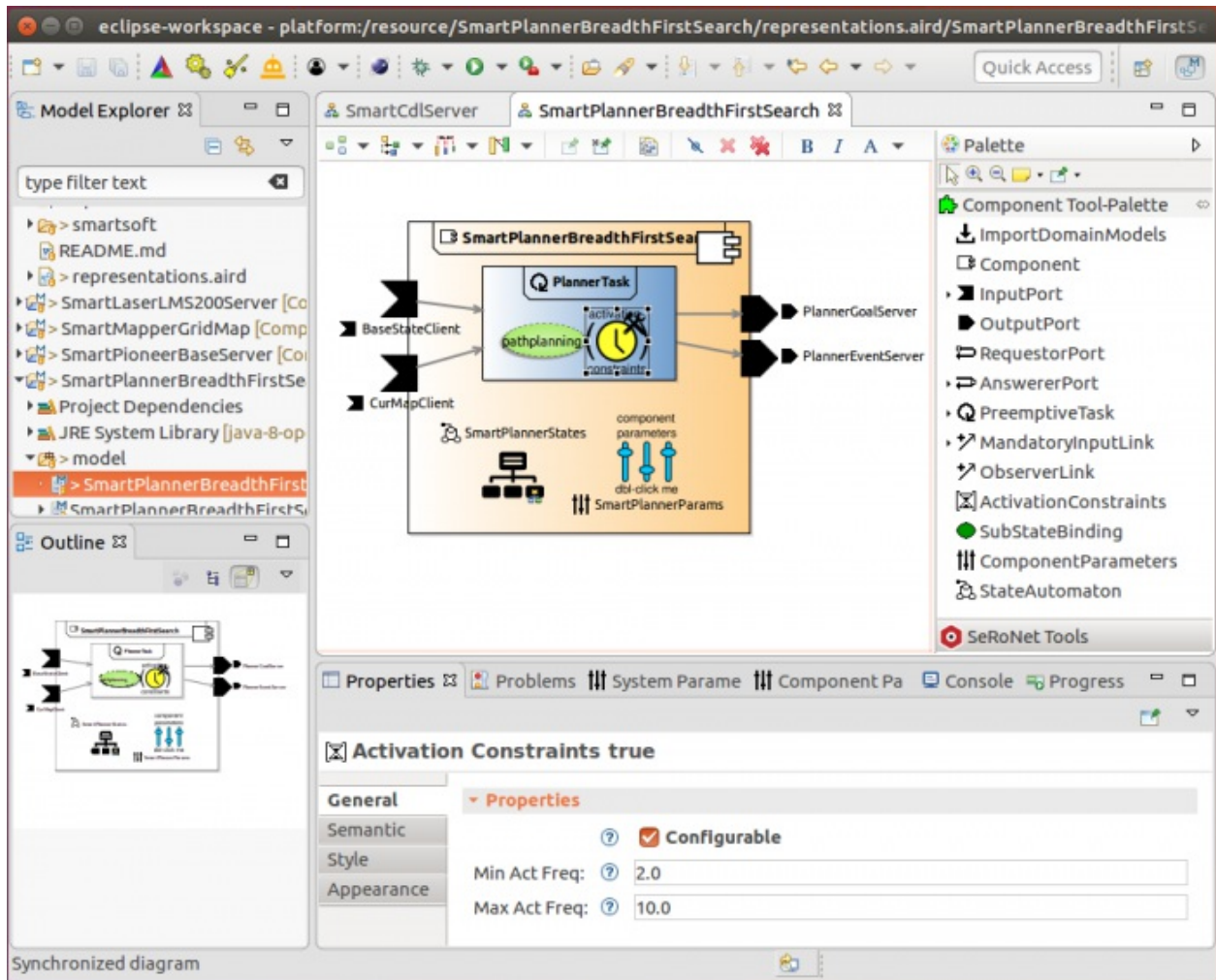


The example system in the figure above consists of five navigation components, from which two are related to hardware devices (i.e., the Pioneer Base and the SICK Laser) and the other three components respectively implementing collision-avoidance (i.e., the CDL component), mapping and path-planning. 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).



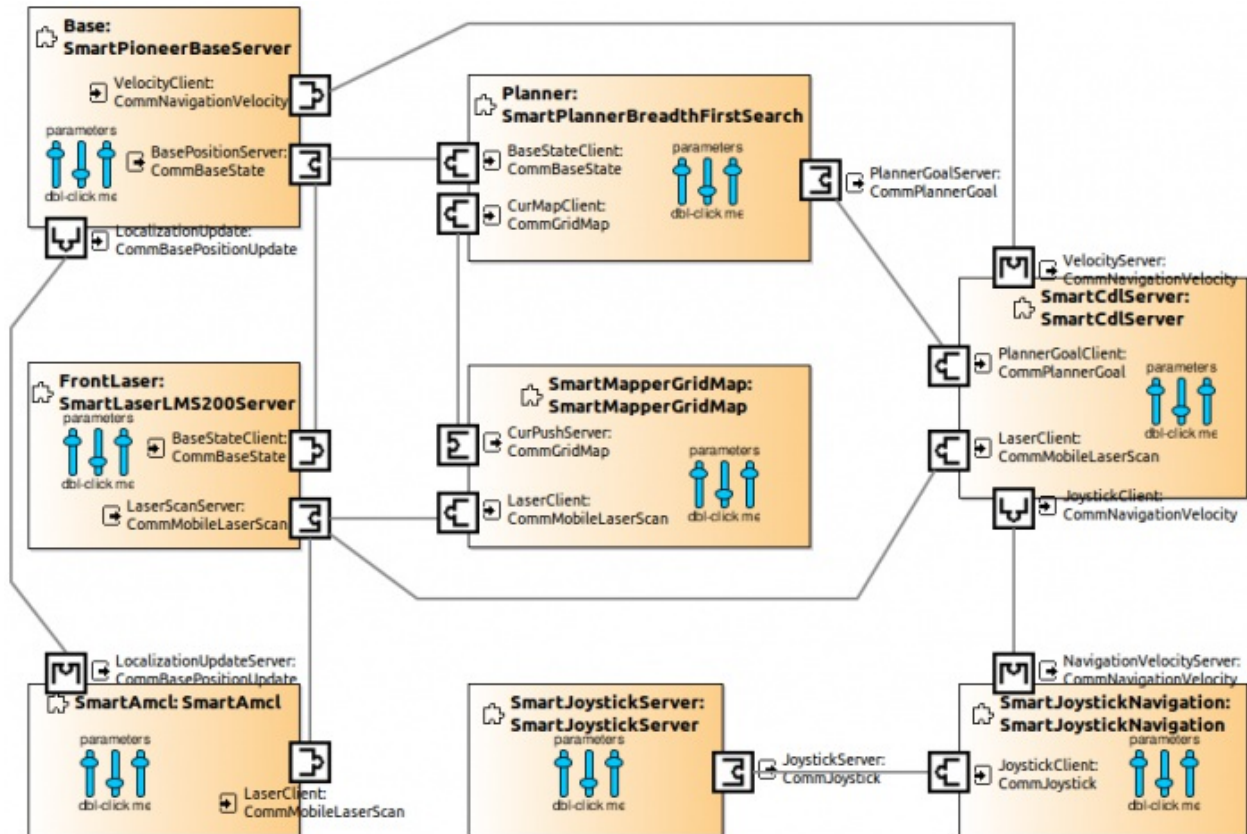
The figure above shows the model of the “SmartPlannerBreadthFirstSearch” component as a representative example for demonstrating the role of the Component Supplier. The responsibility of this role is to define and to implement a component so that it can be (re-)used in different systems. Among other 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

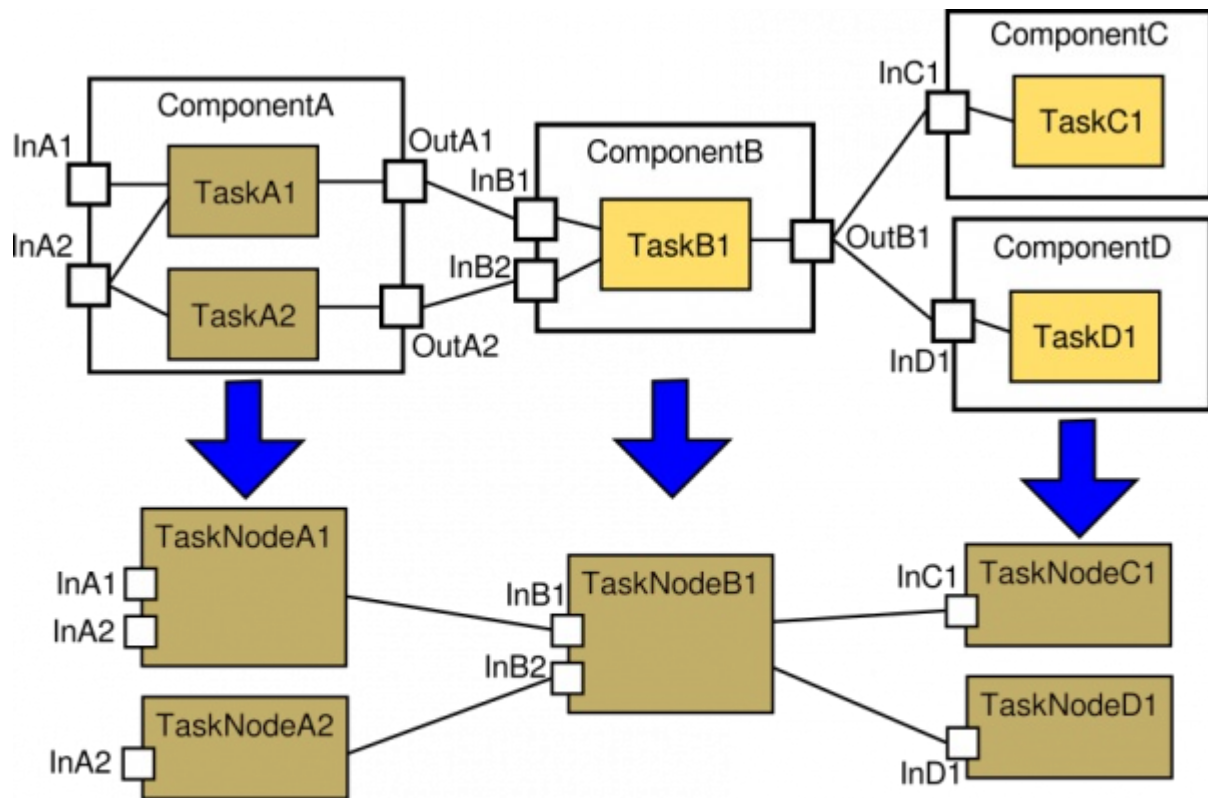
these components.



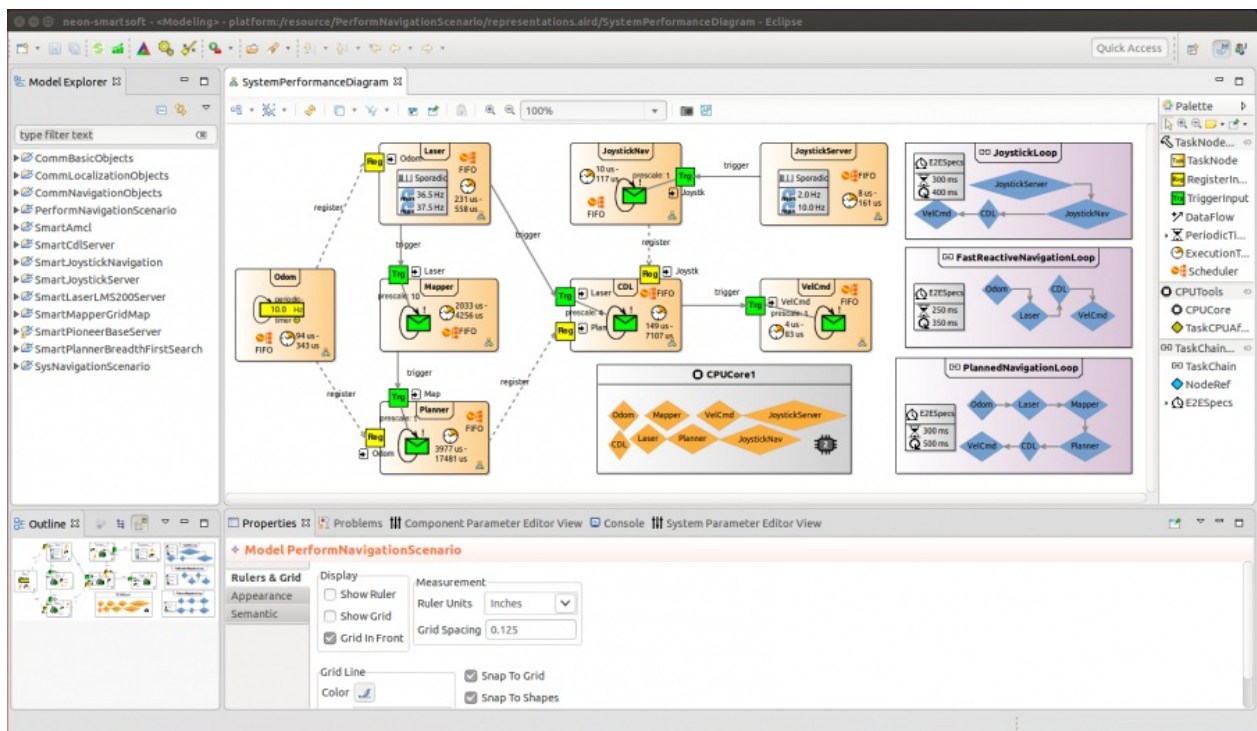
- System Configuration View
- System Builder Role
- System Component Architecture Metamodel

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.



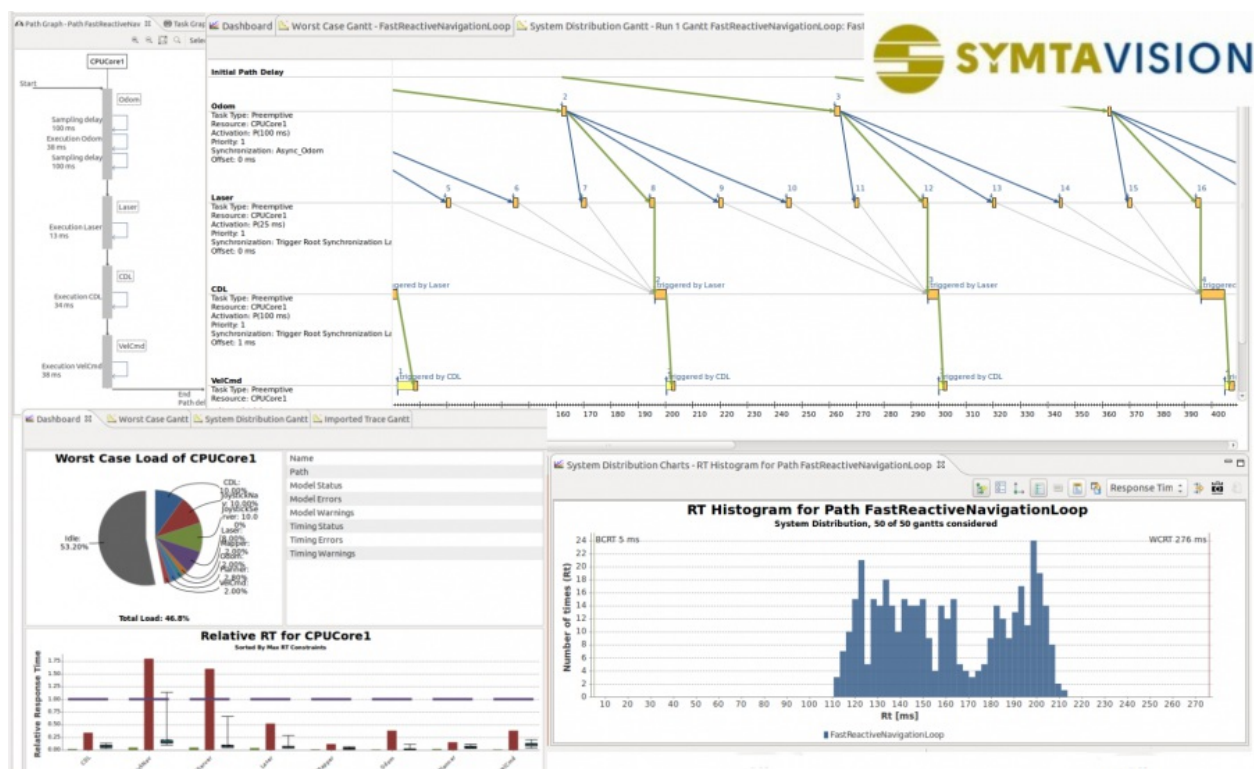
The transformation of a system-configuration model of the navigation scenario into an activity-net results in the model shown in the above figure. In this model individual **activity nodes** (orange blocks in the figure) can be refined by selecting reasonable activation semantics (i.e., selecting a DataTrigger 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.



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 Syntavision). 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](#)
- [Tutorial on Cause-Effect-Chains and Activity Architecture \[https://wiki.servicerobotik-ulm.de/how-tos:cause-effect-chains:start\]](https://wiki.servicerobotik-ulm.de/how-tos:cause-effect-chains:start)

Video

Tooling Support and Roadmap

[Tutorial on Cause-Effect-Chains and Activity Architecture \[https://wiki.servicerobotik-ulm.de/how-tos:cause-effect-chains:start\]](https://wiki.servicerobotik-ulm.de/how-tos:cause-effect-chains:start)

The above described example is developed with the SmartMDSD Toolchain V3 (Technology Preview) whose implementation (together with the presented example) can be found on [SourceForge \[https://sourceforge.net/p/smart-robotics/smartmdsd-v3/ci/master/tree/\]](https://sourceforge.net/p/smart-robotics/smartmdsd-v3/ci/master/tree/). Please note that the features from the Technology Preview are currently under migration into the newest stable SmartMDSD Toolchain release (see [SmartMDSD Toolchain Vendor Website \[https://wiki.servicerobotik-ulm.de/smartmdsd-toolchain:start-toolchain:\]](https://wiki.servicerobotik-ulm.de/smartmdsd-toolchain:start-toolchain:) and in the [RobMoSys Wiki](#)).

The following features from the Technology Preview have already been migrated into the [latest stable SmartMDSD Toolchain release \[https://github.com/ServiceRobotics-Ulm/SmartMDSD-Toolchain/releases/latest\]](https://github.com/ServiceRobotics-Ulm/SmartMDSD-Toolchain/releases/latest):

Feature	Migration Status in stable release
Textual Grammar	migrated
Graphical Notation	under migration
Deployment Integration	migrated
Model Generation for SymTA/S	under migration

If you are interested in understanding all the details of the cause-effect chain approach or to exactly reproduce the presented example, then please use the technology preview. If you are interested in the development of new systems using stable tooling, then please use our [latest stable SmartMDSD Toolchain release \[https://github.com/ServiceRobotics-Ulm/SmartMDSD-Toolchain/releases/latest\]](https://github.com/ServiceRobotics-Ulm/SmartMDSD-Toolchain/releases/latest), where you can already now develop the cause-effect models textually (which serves as a basis also for the graphical notations that will be http://robmosys.eu/wiki/baseline:environment_tools:smartsoft:smartmdsd-toolchain:cause-effect-chain:

added in one of the upcoming releases). Moreover, we are currently exploring different options for open-source model analysis tools that we can use in the same way as SymTA/S. If you have questions, ask them at Discourse Forum [<https://discourse.robmosys.eu/>].

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>]

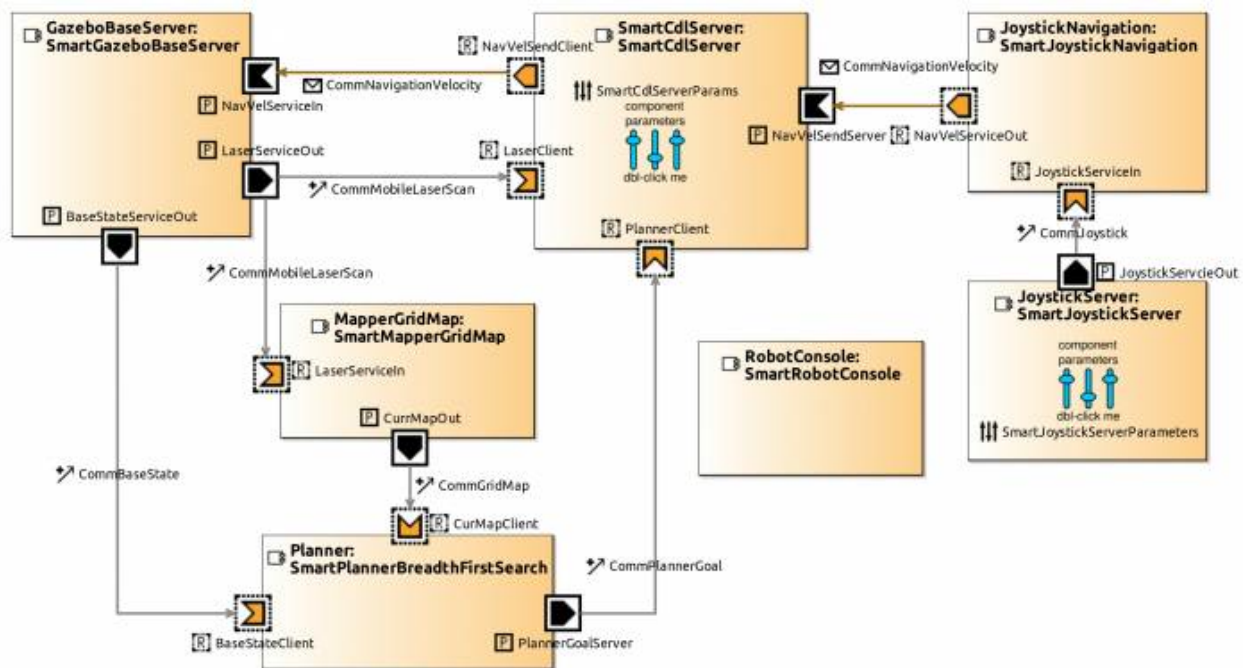
baseline:environment_tools:smartsoft:smartmdsd-toolchain:cause-effect-chain:start · Last modified: 2019/05/20 10:52
http://www.robmosys.eu/wiki/baseline:environment_tools:smartsoft:smartmdsd-toolchain:cause-effect-chain:start

Support for the Flexible Navigation Stack

This page describes how the SmartMDS Toolchain and the SmartSoft World supports the Flexible 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 SmartMDS Toolchain displaying the flexible navigation stack:



Available Software Components in the SmartSoft World

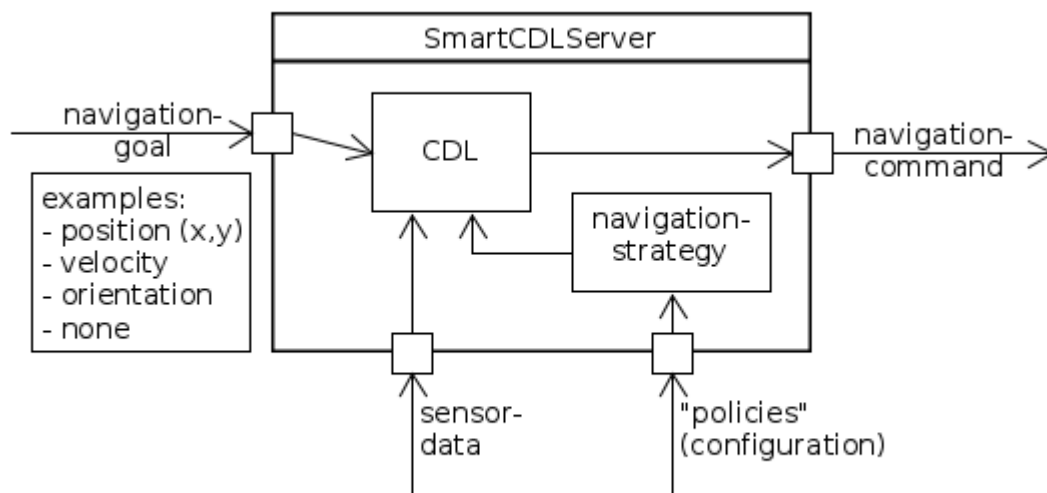
The five ready-to-use navigation components of the navigation stack can be downloaded from the SmartSoft Github component repository [<https://github.com/ServiceRobotics-Ulm/ComponentRepository>]. The following list of references provides documentation for the five navigation components:

- 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 is a 2D grid map component

[Ulm/ComponentRepository/tree/master/SmartMapperGridMap](https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartMapperGridMap)]: this component calculates up to date occupancy grid maps

- [SmartAmcl \[https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartAmcl\]](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) [<http://wiki.ros.org/amcl>] algorithm.

The [SmartCdLServer \[https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartCdLServer\]](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 \[https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartPioneerBaseServer\]](https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartPioneerBaseServer)

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

- [SmartLaserLMS200Server \[https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartLaserLMS200Server\]](https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartLaserLMS200Server): provides laser-scans.

The Flexible Navigation Stack with FESTO Robotino3

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

- **SmartRobotinoBaseServer:** see the [Robotino3 Wiki \[http://wiki.openrobotino.org/index.php?title=Smartsoft\]](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) [<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) [<https://doi.org/10.1109/ETFA.2016.7733602>].

baseline:environment_tools:smartsoft:smartmdsd-toolchain:navigation-stack:start · Last modified: 2020/12/04 15:05
http://www.robmosys.eu/wiki/baseline:environment_tools:smartsoft:smartmdsd-toolchain:navigation-stack:start

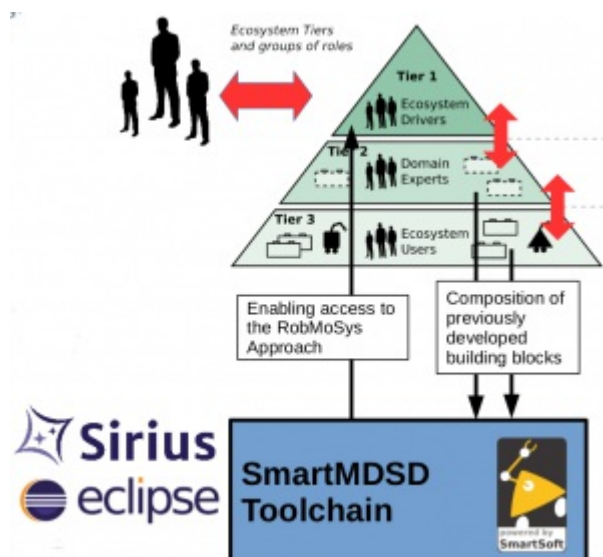
SmartMDSD Toolchain Support for the RobMoSys Ecosystem Organization

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.

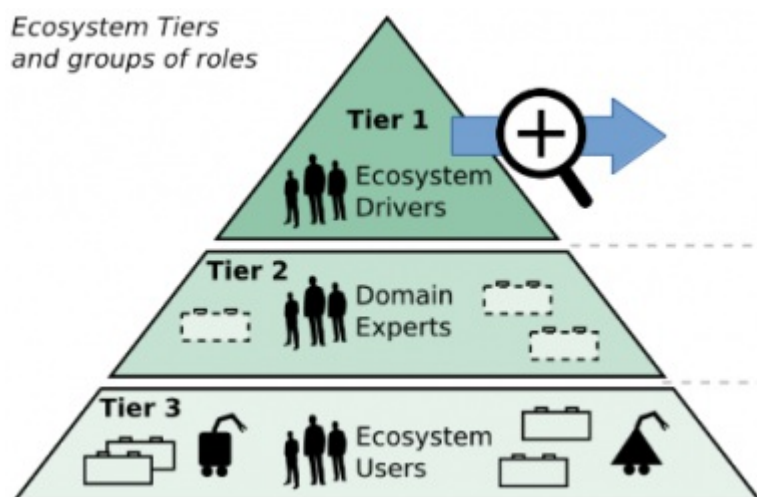
Therefore, SmartMDSD Toolchain provides textual and graphical model editors, and implements a fully fledged code-generator that generates C++ code for SmartSoft Software Components. Moreover, dedicated model editors of the SmartMDSD Toolchain support the different developer roles in their individual 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.

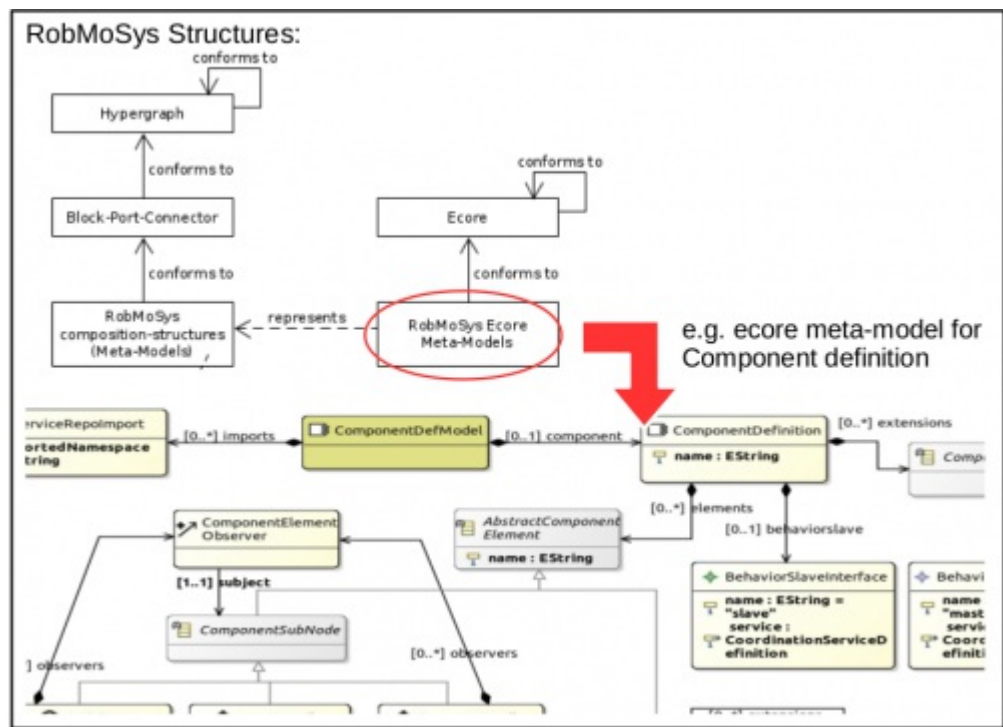


Support for Composition Tier 1

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

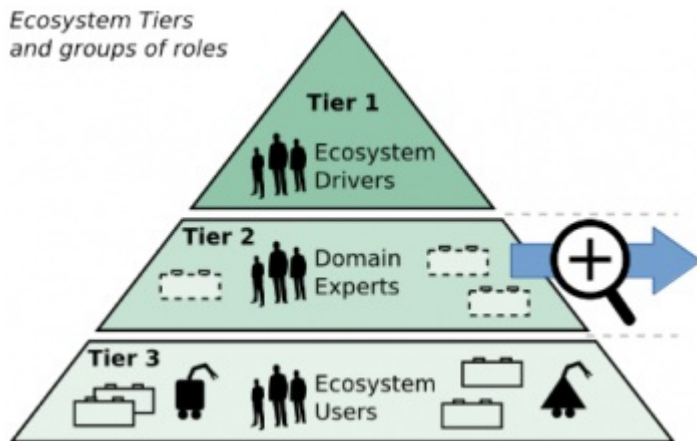


The figure on the right illustrates by the example of the component meta-model how the RobMoSys composition structures are realized based on Eclipse Ecore. This and many other meta-models are implemented within the SmartMDSD Toolchain and are used to provide dedicated model editors for specific roles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



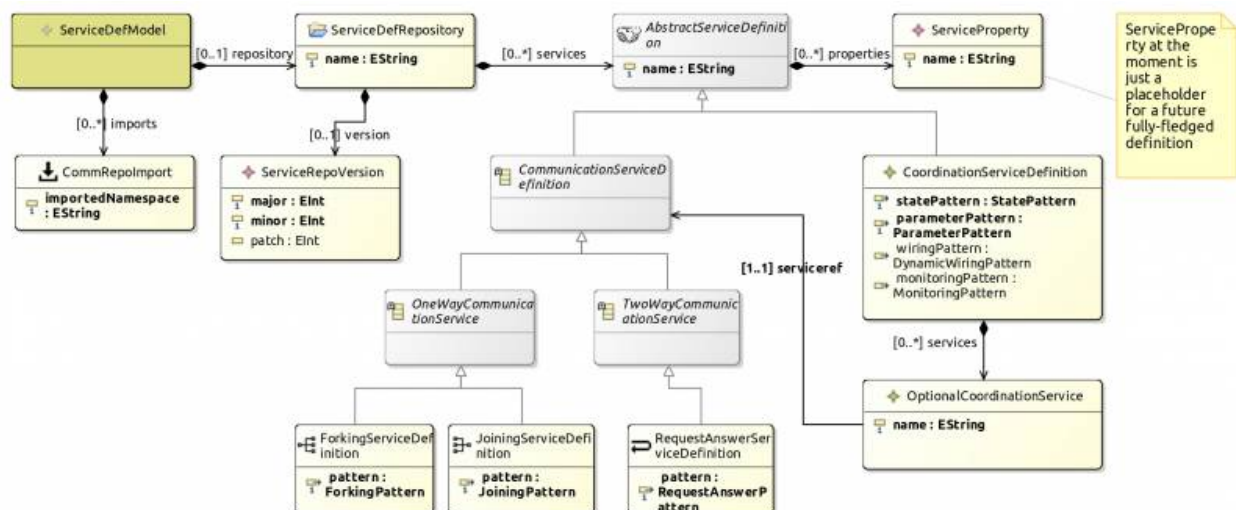
Support for Composition Tier 2

Ecosystem Tiers and groups of roles

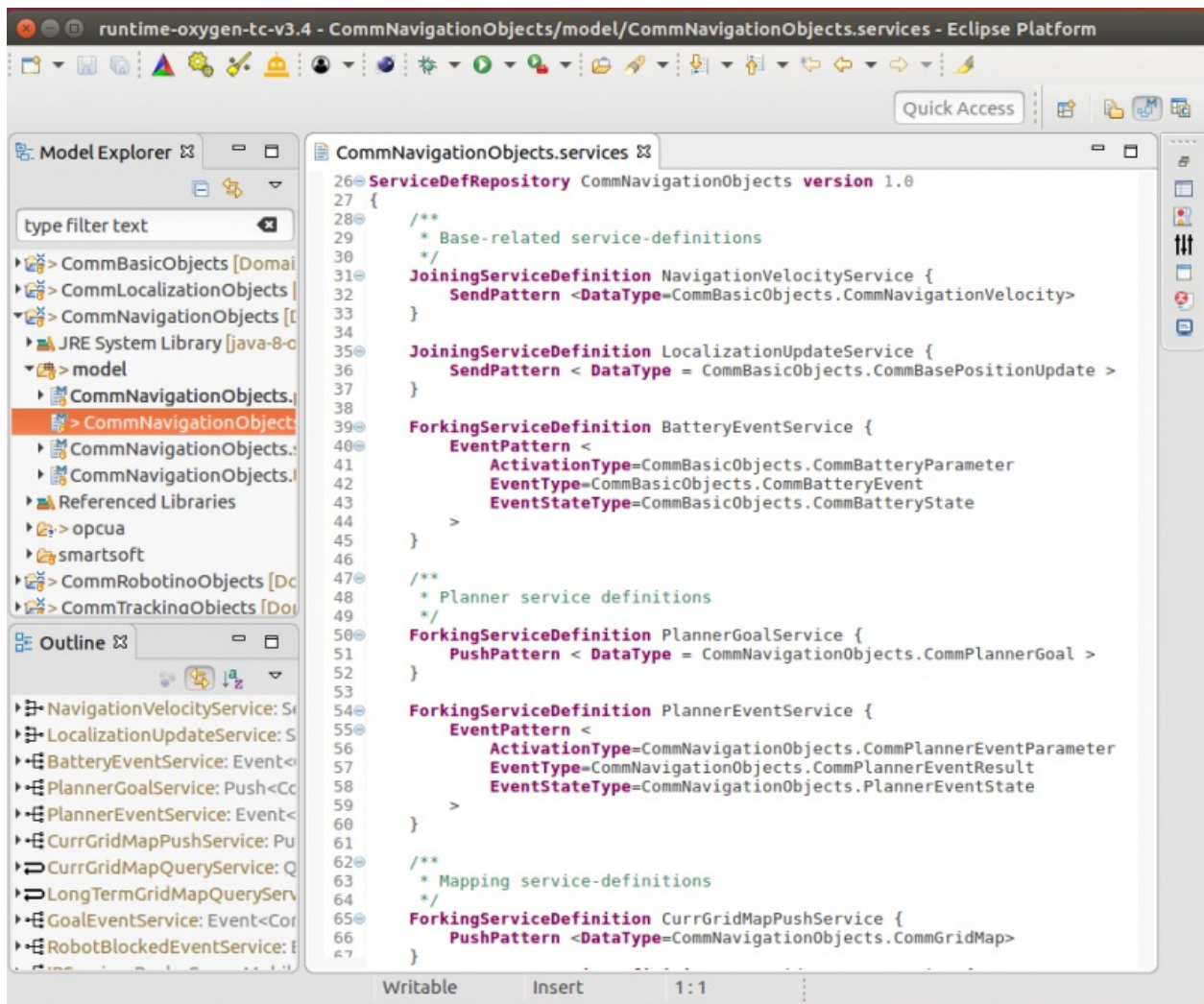


The SmartMDSD Toolchain supports in modeling **domain structures** (i.e., domain models) that conform to the RobMoSys composition structures defined at Tier 1 (see above). On the one hand, this means that the Toolchain internally implements the related Service-Definition Metamodel (see the Ecore diagram 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.



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



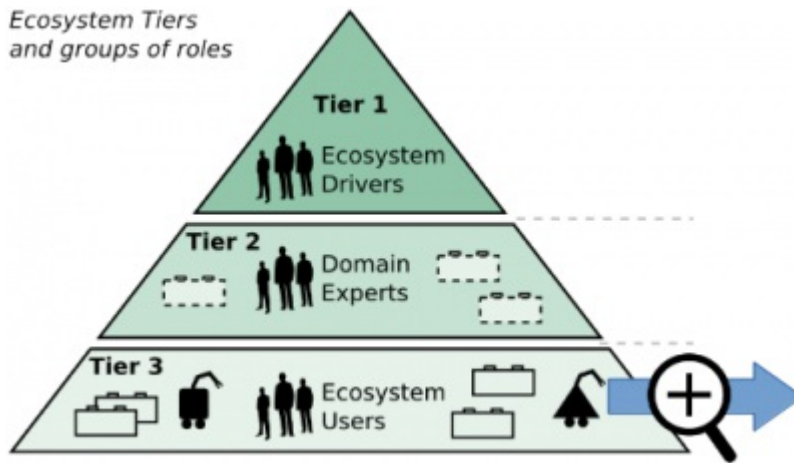
- DomainModelsRepositories [<https://github.com/ServiceRobotics-Ulm/DomainModelsRepositories>]

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**.

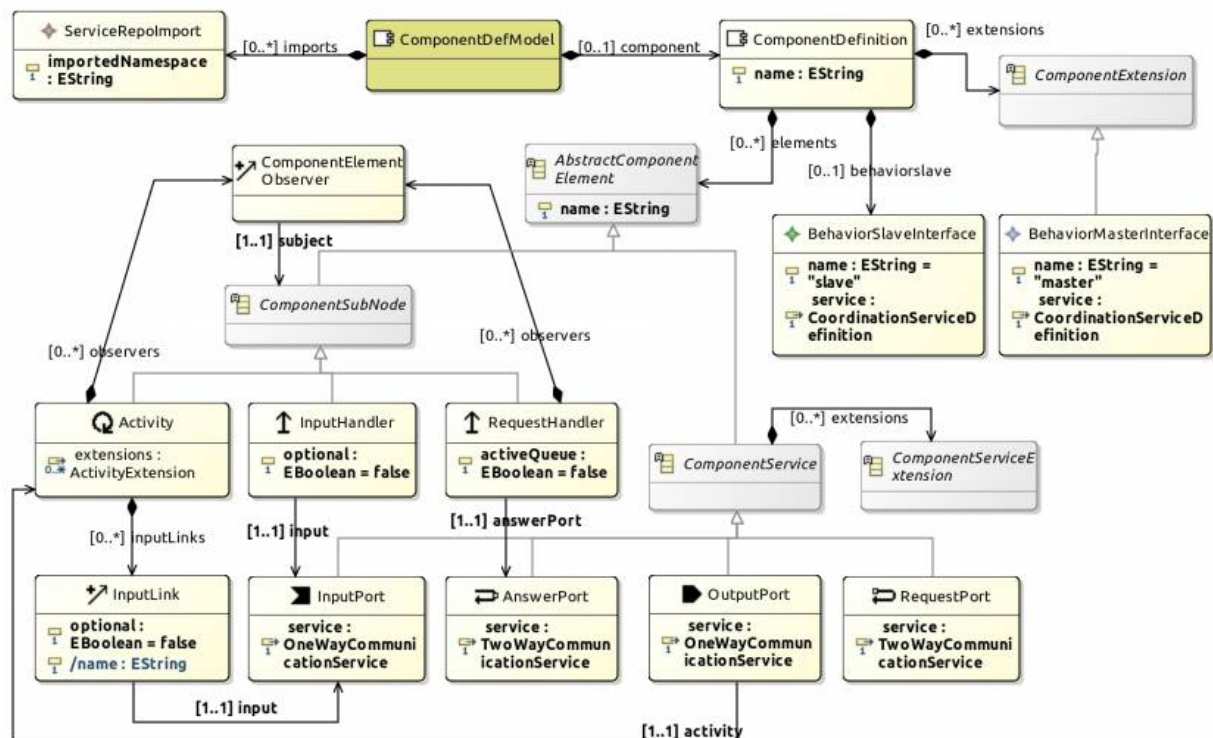
Ecosystem Tiers and groups of roles



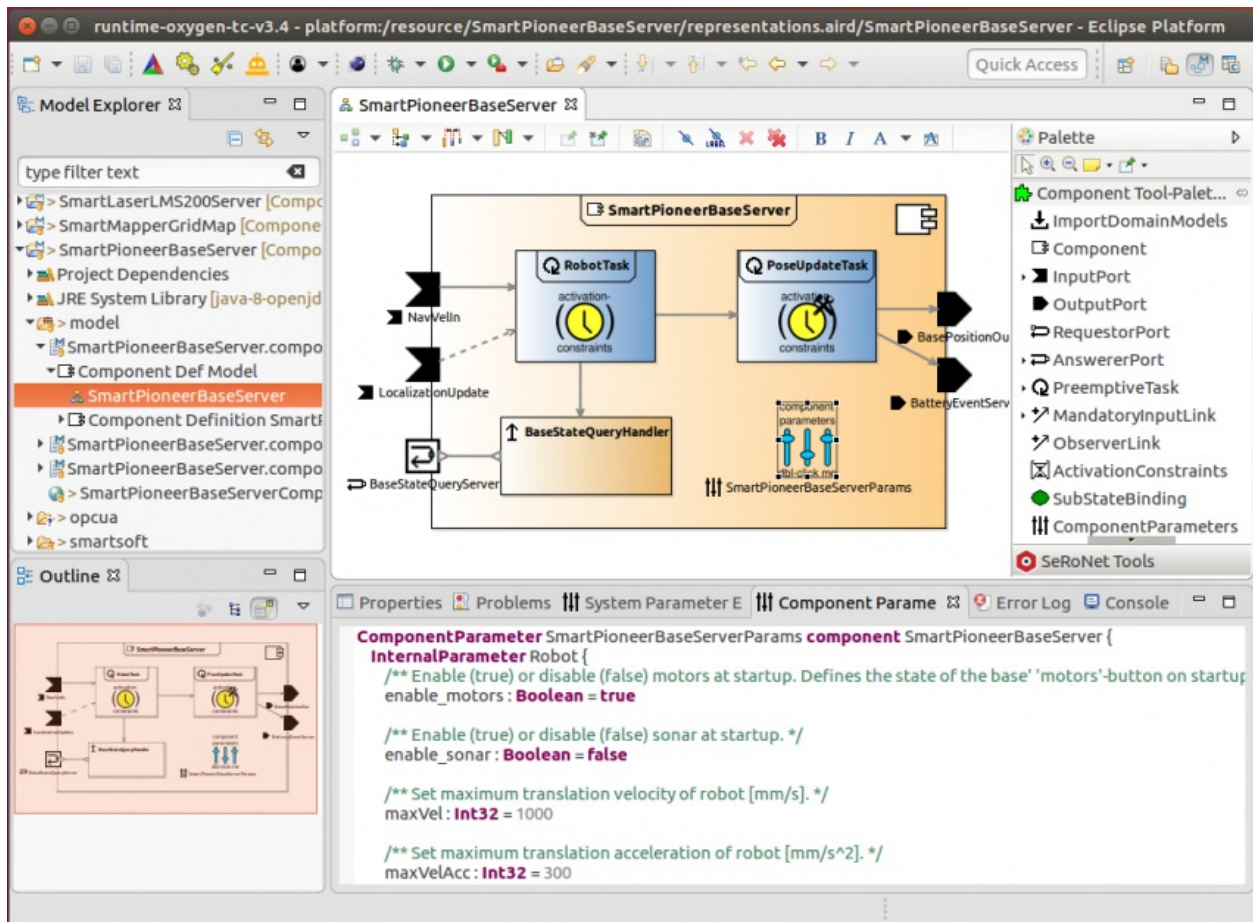
Similar to Tier 2, the SmartMDS Toolchain implements several RobMoSys composition structures (i.e., Ecore-based meta-models) for Tier 3 related to component development, system composition, and deployment. On the one hand, at this Tier 3, Component Suppliers can develop individual components that realize 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 SmartMDS Toolchain as well. A most recent version of the meta-model realizations can be found in the SmartMDS Toolchain sources (which are open-source using the BSD3 License).



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



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: 2019/05/20 10:52
http://www.robmosys.eu/wiki/baseline:environment_tools:smartsoft:smartmdsd-toolchain:ecosystem-tiers:start

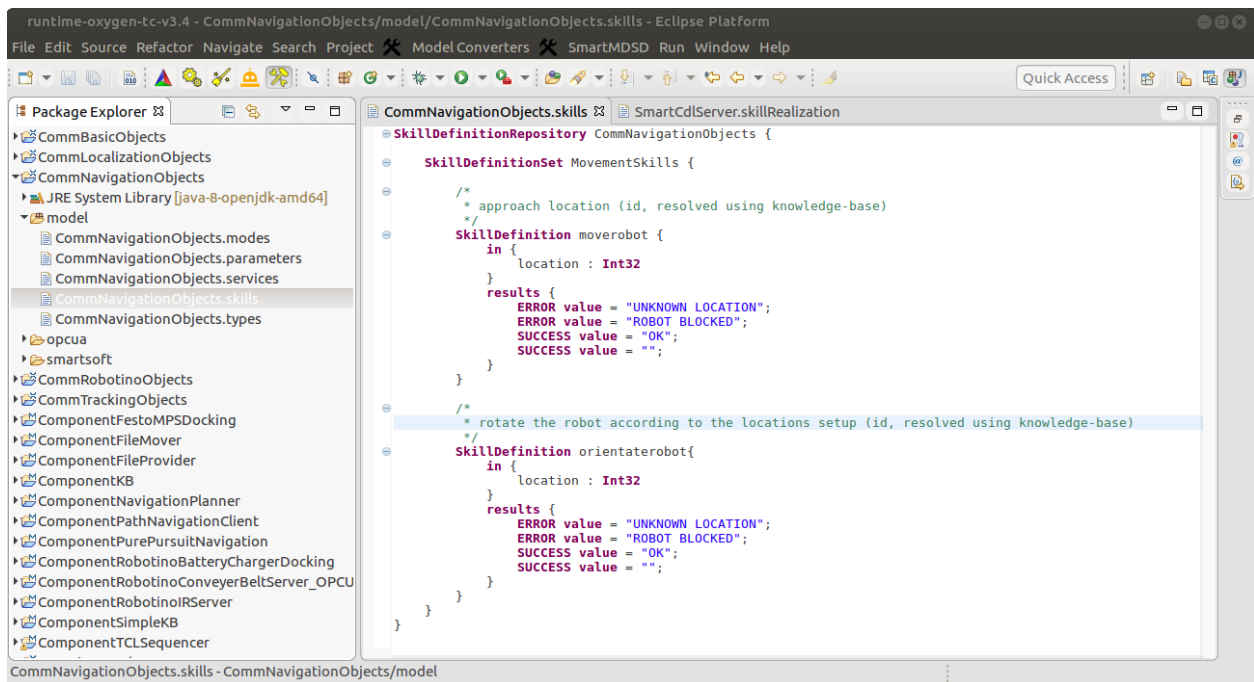
Support of Skills for Robotic Behavior

This page describes how the SmartSoft World and the SmartMDSD Toolchain support Skills for Robotic Behavior.

This description is based on the SmartMDSD Toolchain v3.7.

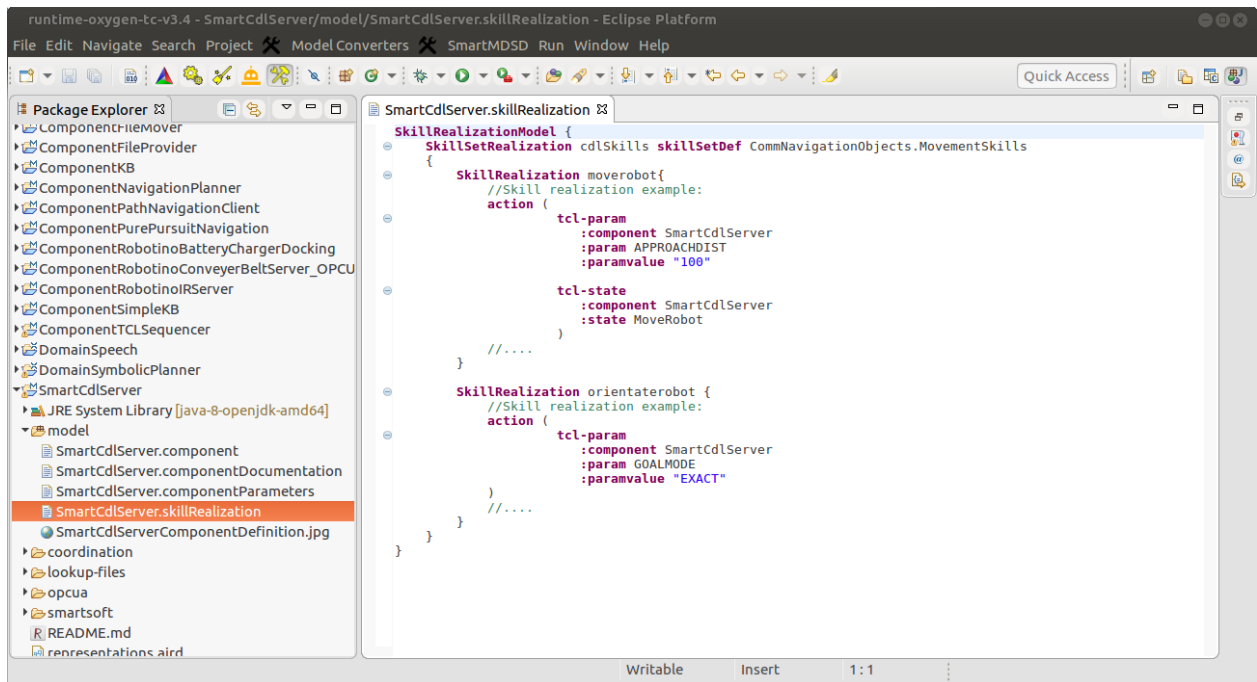
Definition of Skills - Tier 2

The SmartMDSD Toolchain supports the developer in modelling skill definitions with an xtext DSL and the accompanying tooling. Skill definitions are modeled in Domain Model repositories projects, along with the other tier 2 models (see example in figure below). The component developer is able to use those definitions to realize the skills.



Realization of Skills - Tier 3 (Component Developer)

Using the modeled skill definition, the SmartMDSD Toolchain supports the role of the component developer to realize skills using an xtext DSL and the accompanying tooling. This skill realizations interact with the components and coordinate them using the components coordination interface (parameter, events, activation etc.), see example in figure below.



Examples

Robot Navigation Examples: moverobot

Example skills for robot navigation, i.e. moverobot and orientaterobot, can be found at:

- Skill Definition: Domain Model repository [<https://github.com/ServiceRobotics-Ulm/DomainModelsRepositories/tree/master/CommNavigationObjects/model>]
- Skill Realization: CDL Component [<https://github.com/ServiceRobotics-Ulm/ComponentRepository/tree/master/SmartCdServer/model>]

Intralogistics i4.0 Robot Fleet Pilot

To see the skill support of the SmartMDSD Toolchain in action, please refer to [Robotic Behavior in RobMoSys using Behavior Trees and the SmartMDSD Toolchain \(MOOD2be ITP\)](#).

See also

- [Skills for Robotic Behavior](#)
- [Robotic Behavior in RobMoSys using Behavior Trees and the SmartMDSD Toolchain \(MOOD2be ITP\)](#)
- [The SmartMDSD Toolchain can be used together with Groot, a GUI to develop behavior trees.](#)

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. \[http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2\]](#)

baseline:environment_tools:smartsoft:smartmdsd-toolchain:skills:start · Last modified: 2019/05/20 10:52
http://www.robmosys.eu/wiki/baseline:environment_tools:smartsoft:smartmdsd-toolchain:skills:start

Support for Coordinating Activities and Life Cycle of Software Components

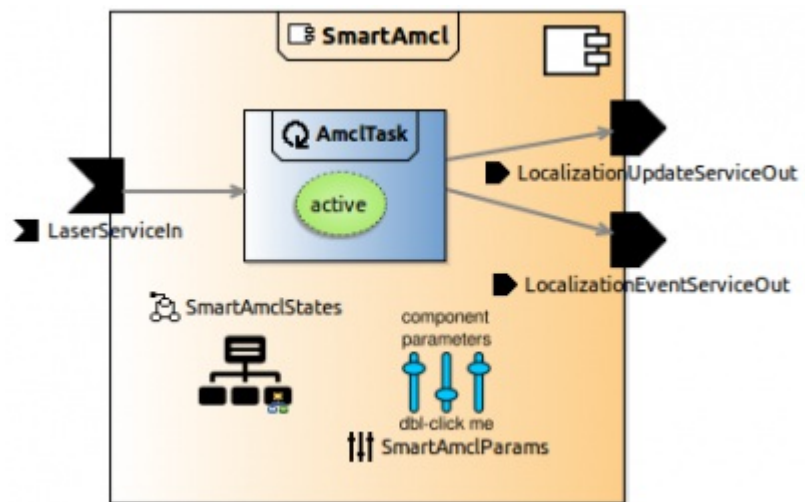
This page describes how the SmartMDS Toolchain supports Coordinating Activities and Life Cycle of Software Components.

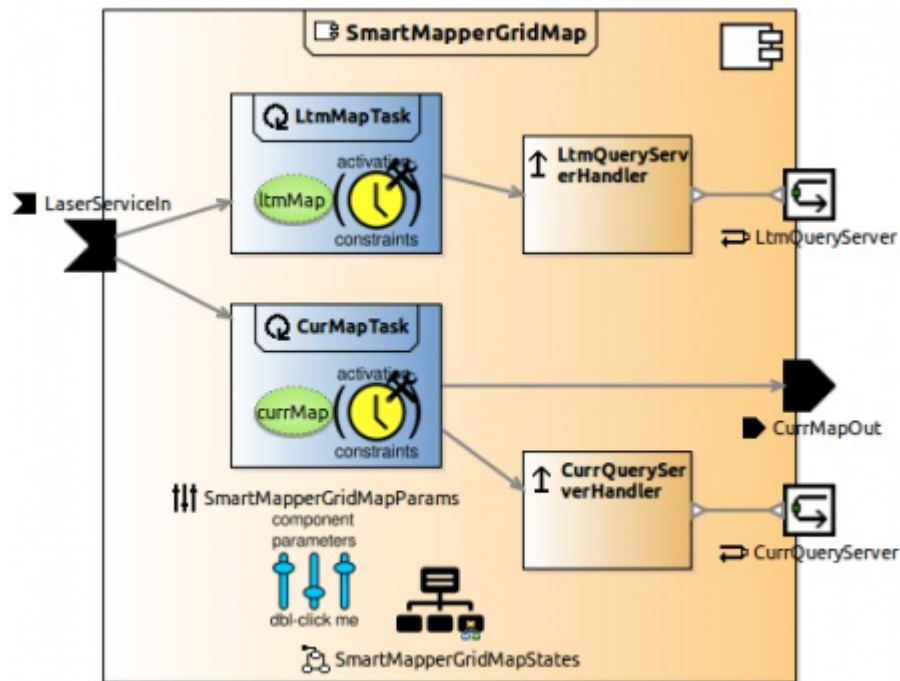
Example Use-Cases for Component Operation Modes

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

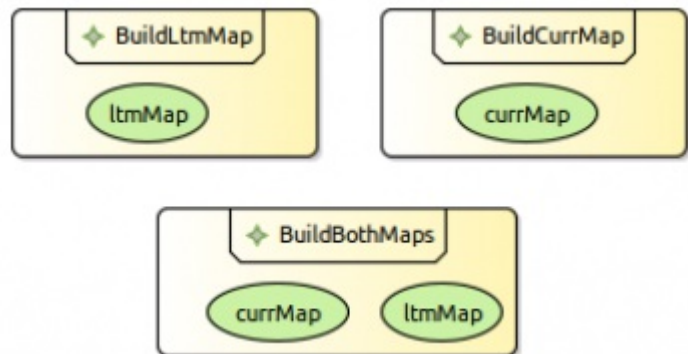
implicitly available for each component by default. Additionally, the component's lifecycle provides two default **operation modes** called “active” and “neutral” (as part of the “Alive” submachine within the component's lifecycle). That is, if the “active” **operation mode** is activated, then the referenced **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](#)

baseline:environment_tools:smartsoft:smartmdsd-toolchain:component-activities:start · Last modified: 2019/05/20 10:52

http://www.robmosys.eu/wiki/baseline:environment_tools:smartsoft:smartmdsd-toolchain:component-activities:start

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.



Authors	Service Robotics Research Center at the Ulm University of Applied Sciences
Website	https://wiki.servicerobotik-ulm.de/smartmdsd-toolchain:start [https://wiki.servicerobotik-ulm.de/smartmdsd-toolchain:start]
License	BSD3

Getting Started and Download

- [SmartMDSD Toolchain Website](https://wiki.servicerobotik-ulm.de/smartmdsd-toolchain:start) [<https://wiki.servicerobotik-ulm.de/smartmdsd-toolchain:start>]
- [SmartMDSD Toolchain VirtualBox virtual machine image](https://wiki.servicerobotik-ulm.de/virtual-machine) [<https://wiki.servicerobotik-ulm.de/virtual-machine>] (development environment preinstalled)
- [Getting Started Guide](https://wiki.servicerobotik-ulm.de/getting-started-guide) [<https://wiki.servicerobotik-ulm.de/getting-started-guide>]
- For documentation and tutorials, please refer to the [SmartMDSD Toolchain Website](https://wiki.servicerobotik-ulm.de/smartmdsd-toolchain:start) [<https://wiki.servicerobotik-ulm.de/smartmdsd-toolchain:start>]

Tutorials and HowTo's

- [Running the Gazebo/Tiago/SmartSoft Scenario](#)
- Full list of Tutorials [<https://wiki.servicerobotik-ulm.de/tutorials:start>]. Some highlights are:
 - [Developing Your First Software Component](https://wiki.servicerobotik-ulm.de/tutorials:develop-your-first-component:start) [<https://wiki.servicerobotik-ulm.de/tutorials:develop-your-first-component:start>] | video tutorial [https://youtu.be/BRI_HKMilNw]
 - [Developing Your First System: Composing Software Components](https://wiki.servicerobotik-ulm.de/tutorials:develop-your-first-system:start) [<https://wiki.servicerobotik-ulm.de/tutorials:develop-your-first-system:start>] | video tutorial [https://www.youtube.com/watch?v=3LNhatjWb_c]
 - [Accessing an OPC UA Device: Using the Plain OPC UA Port \(DeviceClient\) to create a Mixed-Port Component](https://wiki.servicerobotik-ulm.de/tutorials:opcua-client:start) [<https://wiki.servicerobotik-ulm.de/tutorials:opcua-client:start>] | video tutorial [https://youtu.be/uPZ07_Gi3YE]
 - [Composing a System with OPC UA Mixed-Port Components](https://wiki.servicerobotik-ulm.de/tutorials:opcua-client-system:start) [<https://wiki.servicerobotik-ulm.de/tutorials:opcua-client-system:start>] | video tutorial [<https://youtu.be/udQiwRdzCVw>]
 - [Developing an OPC UA Server: Using the Plain OPC UA Port \(ReadServer\)](#)

- [\[https://wiki.servicerobotik-ulm.de/tutorials:opcua-server:start\]](https://wiki.servicerobotik-ulm.de/tutorials:opcua-server:start) | video tutorial [\[https://youtu.be/Ho7Fr2KefKQ\]](https://youtu.be/Ho7Fr2KefKQ)
- Mixed-Port for ROS: Accessing ROS nodes from software components [\[https://wiki.servicerobotik-ulm.de/tutorials:ros:mixed-port-component-ros\]](https://wiki.servicerobotik-ulm.de/tutorials:ros:mixed-port-component-ros) | video tutorial [\[https://youtu.be/N1oMMIBIx5k\]](https://youtu.be/N1oMMIBIx5k)
- Checking System Level Properties: Dependency-Graph Extensions [\[https://wiki.servicerobotik-ulm.de/tutorials:start#lesson_8dependency-graph_extensions_for_smartmdsd_toolchain_smartdg\]](https://wiki.servicerobotik-ulm.de/tutorials:start#lesson_8dependency-graph_extensions_for_smartmdsd_toolchain_smartdg) | video tutorial [\[https://youtu.be/tXEsvgSH9bU\]](https://youtu.be/tXEsvgSH9bU)
- For HowTo's see <https://wiki.servicerobotik-ulm.de/how-tos:start> [\[https://wiki.servicerobotik-ulm.de/how-tos:start\]](https://wiki.servicerobotik-ulm.de/how-tos:start)

RobMoSys Support and Use Cases

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

- Support of Skills for Robotic Behavior
- 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 and Models

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

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

- Domain Models Repositories [\[https://github.com/Servicerobotics-Ulm/DomainModelsRepositories\]](https://github.com/Servicerobotics-Ulm/DomainModelsRepositories): These are examples of RobMoSys Composition Tier 2
- Component Repository [\[https://github.com/Servicerobotics-Ulm/ComponentRepository\]](https://github.com/Servicerobotics-Ulm/ComponentRepository): These are examples of previously developed building blocks for Tier 3
- System Repository [\[https://github.com/Servicerobotics-Ulm/SystemRepository\]](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

For use with the SmartMDSD Toolchain v2, see: [List of available components \[http://www.servicerobotik-ulm.de/drupal/doxygen/components_commrep/group__componentGroup.html\]](http://www.servicerobotik-ulm.de/drupal/doxygen/components_commrep/group__componentGroup.html).

baseline:environment_tools:smartsoft:smartmdsd-toolchain:start · Last modified: 2020/12/07 12:07
http://www.robmosys.eu/wiki/baseline:environment_tools:smartsoft:smartmdsd-toolchain:start

Support for Service-based Composition

This page uses the SmartMDSD Toolchain to illustrate the support for Service-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.

baseline:environment_tools:smartsoft:smartmdsd-toolchain:service-composition:start · Last modified: 2019/05/20

10:52

http://www.robmosys.eu/wiki/baseline:environment_tools:smartsoft:smartmdsd-toolchain:service-composition:start

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 sourcesoftware 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) [<http://www.servicerobotik-ulm.de/drupal/?q=node/7>]

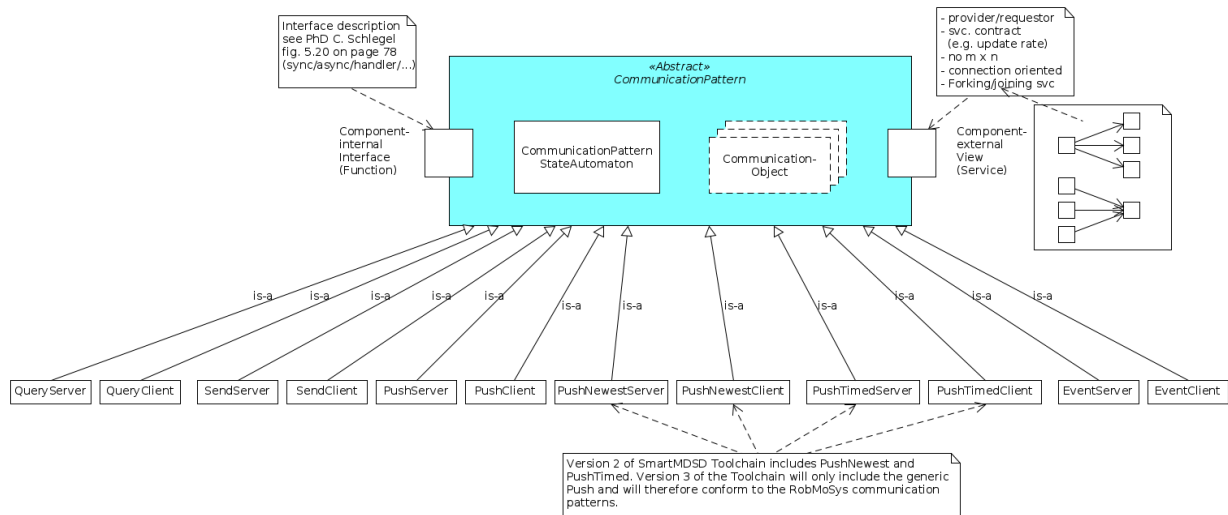
The SmartMDSD Toolchain (version 2.x) and the SmartSoft framework (version 2.x) are very matured (TRL 6) are – among others – used by FESTO Robotino. They will be supported for a while but are not fully conform to RobMoSys.

The SmartMDSD Toolchain (version 3.x) is fully conform to RobMoSys and meanwhile achieved the same maturity level as the version 2.x series (in use within other projects such as BMWi PAiCE SeRoNet and used by FESTO Robotino).

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.



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

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: - Communication Object (data structure) - Communication Patterns (comm. semantics) 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 based on needs.

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) [<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) [<http://www.servicerobotik-ulm.de/drupal/?q=node/91>] in simulation

Selected Publications

- 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. [Link](http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2) [<http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2>]
- 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. [Link](https://mediatum.ub.tum.de/?id=1362587) [<https://mediatum.ub.tum.de/?id=1362587>]
- Dennis Stampfer, Alex Lotz, Matthias Lutz, and Christian Schlegel. “The SmartMDS Toolchain: An Integrated MDS 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) [<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 (SIMPAP). San Francisco, CA, USA, Dec. 2016, pp. 170–176. [LINK \[http://dx.doi.org/10.1109/SIMPAP.2016.7862392\]](http://dx.doi.org/10.1109/SIMPAP.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 \[https://www.gi.de/service/publikationen/lni/gi-edition-proceedings-2014/gi-edition-lecture-notes-in-informatics-lni-p-232.html\]](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.

baseline:environment_tools:smartsoft:start · Last modified: 2020/12/05 09:13
http://www.robmosys.eu/wiki/baseline:environment_tools:smartsoft:start

Page-Status: Incubator

This page is under consolidation and not yet part of the RobMoSys Body of Knowledge. [Read more.](#)

The DConv data converter tool

DConv is a computer program for the automatic conversion of objects of different data type but same semantics. DConv can generate code or perform the actual conversion as a runtime service. The underlying mechanism relies on semantic meta data which has to be attached to existing type definitions, like C data structs or ROS topics.

Motivation

Software design and development of robotics applications consists of integrating functionalities of heterogeneous nature, often shipped as a set of libraries or software components. During the integration process, data type conversion is one of the unavoidable pieces of “glue-code” that must be provided.

Ensuring a correct conversion is a tedious and error-prone process, also due to informal documentation of the code providing incomplete descriptions of all choices affecting the actual meaning of the data: physical units, ordering of variables in a composite data structure, selection of any of the 24 different Euler angle representations, reference frames for coordinate values, just to name a few. Kinematics and dynamics of rigid bodies, which is ubiquitous in robotics applications, are particularly affected by the variability of mathematical formalisms, for example for the representation of orientation.

“Standardized” types expressed with an IDL (Interface Description Language) would not solve the problem, as IDLs only address the layout of the digital type, and cannot represent the *meaning* of the type by means of semantic tags or constraints. Furthermore, traditional consistency checking based on the equivalence of the data type prevents more meaningful analysis based on the semantic of the data, that is, what it *represents*. For example, a pose can be represented by an homogeneous transformation matrix, or by a pair with position vector and quaternion.

The tool

At the moment, the tool consist of a simple Domain Specific Language (DSL) to manually augment existing digital type definitions with semantic annotations. The schema for such annotations is now focused on the *geometry* domain, and essentially accounts for the position and orientation of rigid bodies.

There is also a runtime, the tool itself, which parses the semantic annotations and can generate C code performing the conversion between two instances of the annotated type. The DConv tool can be invoked programmatically to enable the integration with existing code generation tools.

More limited and experimental support for automatic runtime conversion is also available. (**TODO** add more explanation, which languages are supported)

DSL

The following are some excerpts from a sample annotation document, which we call “dproto model”.

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

These are essentially some definitions of abstract data types that can be referenced from the rest of the model. `Scalar`, `Vector`, etc. are keywords of the language. An `algebraic` entry allows to give a name to a general, abstract layout, such as three named scalars, or a sequence of three.

This is an actual dproto definition:

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

The *semantic* is thus a simple tag which can be interpreted in the context of the Geometry domain; `ddr` stands for Digital Data Representation, and refers to the actual digital type being tagged by the model. It can be defined elsewhere in the document, as in the following:

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

The `algebraic` attribute establishes which is the abstract layout of the type, in this case an unordered set of the three named fields. The 'dr' attribute (Digital Representation) then determines the mapping between the actual type and the internal abstract type (the `algebraic`); in this case, a mapping between the fields `x`, `y` and `z` (which appear in the ROS message) and the fields of the 'position_named'.

Another dproto model with same semantic and algebraic type, but different mapping (e.g. `a=X`, `b=Y`, `c=Z`) would be recognized as compatible and convertible.

An `alias` is an explicitly stated equivalence relation between two algebraic types, like in the following:

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

By defining the mapping between the fields (0,1 and 2 represents indices in the `Vector3` that defines `position3`), the `alias` essentially enables some automatic direct conversions (see next section).

Conversions

The subject of the conversion is not the data prototype dproto, but an instance of it, called data block (dblx). A data block is the combination of the data instance itself with the metadata i.e., its dproto. In fact, it is not necessary to pack the whole dproto definition with the data, as long as the implementation allows to retrieve that information from a unique identifier.

A conversion from a dblx (the source) and another (the target) consists in computing the values of the target dblx from the values of the source, such that they have the same information content. The conversion is possible only if the dblx are semantically equivalent, which is verifiable from their dproto definitions.

The following table summarizes the types of possible conversions performed by the `dconv` tool. Conversions can be grouped coarsely in two sets: direct and indirect. Direct conversions can be performed in a single step, as

the mapping between two data types can be inferred right away from the corresponding meta data. Indirect conversions, on the other hand, require finding a *path* across multiple dproto and compose the individual conversion functions.

#	Conversion type	Requirements on the two dproto (see note *)
1	direct plain	same coord and algebraic
2	direct, via alias	same coord, different algebraic but alias known
3	direct, via conversion	different coord but known conversion relation
4	direct composite	both dproto are composites, members convertible
5	indirect	an ordered sequence of direct conversions through other dprotos must be available
6	indirect composite	Like #4, but at least one member requires an indirect conversion
		* <i>the requirement which is common to all cases is, of course, having compatible semantic</i>

The `conversion` keyword used in the table refers to another relation that can be explicitly mentioned in the DSL document (e.g. the model), like in the following example:

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

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

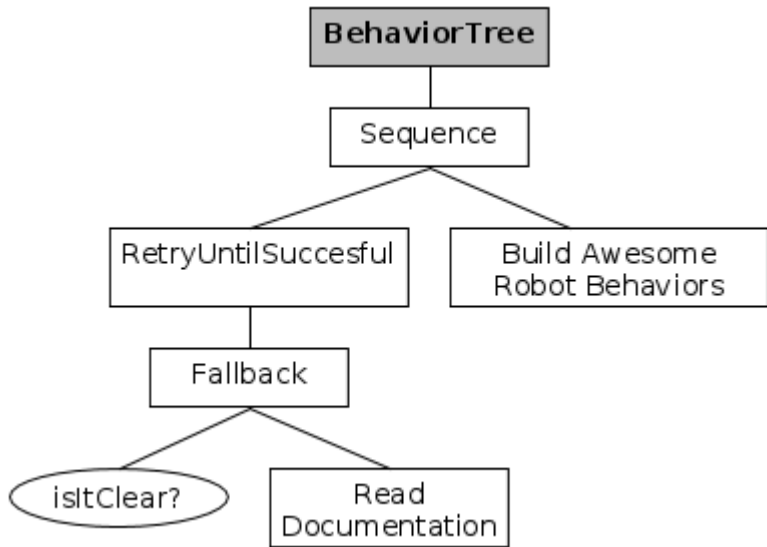
conversion QUAT -> ROT = quat2rot
```

The `conversion` essentially declares that there exist a function suitable for the conversion of the two types. Of course the tool must know what to do with the provided id (`quat2rot` in this case), such as generating suitable code for the invocation of the function.

The tool would therefore be shipped along with a set of pre-implemented conversion functions of various kind, for specific concrete types, like a traditional API. However, the fundamental difference is that such a library would not be meant for the final user, but purely as a backend for the tool. The conversion functions themselves would exhibit exactly the same issues of regular libraries (e.g. support for only one specific concrete digital data type for any semantic type), but that would not matter as the tool would expose more versatile conversion capabilities.

BehaviorTree.CPP

BehaviorTree.CPP is a C++ framework to design, execute, monitor and log robotics behaviors, using Behavior Trees.

Authors	Davide Faconti, Eurecat
Website	https://behaviortree.github.io/BehaviorTree.CPP/ [https://behaviortree.github.io/BehaviorTree.CPP/]
License	MIT
Screenshot	

Description

Hierarchical Finite State Machines are often used to design the behaviors of a robot. The purpose of this abstraction is to have a better Separation of Concern (Computation vs Coordination) and Separation of Roles (Component Developer vs Behavior Developer). This library provides an alternative to HFSM based on BehaviorTrees. Unlike most of the other implementations which use scripting languages such as Lua or Python, this library is implemented in C++. Nevertheless, behaviors can be modified and loaded at run-time without the need for recompiling the user's application. The framework provides multiple tools to help the user design, compose and debug robots behaviors.

Features

BehaviorTree.CPP provides multiple tools to help the user design, compose and debug robots behaviors to the Behavior Developer. State transitions can be recorded on file or be published in real-time to allow tools such as Groot to visualize them in a human friendly way.

Relation to other RobMoSys Tools

The BT.CPP library is completely Middleware independent. In the context of RobMosys, it was demonstrated how a **generic execution engine** can load **aspecific plugin** which is meant to interact with the SmartMDSD Toolchain.

This plugins contains the interface with software components or the more abstract concept of “skills”. In other words, reading the file manifest containing the available skills, the engine can register the corresponding Actions programmatically at run-time, freeing up the user from both manual code writing and the need for code generation.

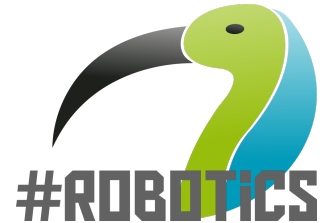
Further Resources

- MOOD2Be Integrated Technical Project [<https://robmosys.eu/mood2be>]
- Tutorials: https://behaviortree.github.io/BehaviorTree.CPP/BT_basics/
[https://behaviortree.github.io/BehaviorTree.CPP/BT_basics/]
- <https://github.com/BehaviorTree/BehaviorTree.CPP> [<https://github.com/BehaviorTree/BehaviorTree.CPP>]

baseline:environment_tools:behaviortree.cpp · Last modified: 2019/05/20 10:49
http://www.robmosys.eu/wiki/baseline:environment_tools:behaviortree.cpp

Papyrus for Robotics

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 successfully 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 domain specific modeling languages (DSMLs) and tools are being collected under the *Papyrus for Robotics* umbrella.

Authors	Commissariat à l'Énergie Atomique (CEA-List) [https://www.eclipse.org/membership/showMember.php?member_id=937]
Website	https://www.eclipse.org/papyrus/components/robotics/ [https://www.eclipse.org/papyrus/components/robotics/]
License	Eclipse Public License 2.0[https://projects.eclipse.org/license/epl-2.0]

Getting Started and Download

- Papyrus for Robotics Website [<https://www.eclipse.org/papyrus/components/robotics/>] — It contains a download link for a RCP (Rich Client Platform, i.e. a ready-to-use Eclipse Installation) with Papyrus for Robotics installed, as well as an update-site for use in existing Eclipse installations (either 2018-09 or 2018-12). Use the “Check for updates” action to automatically get updates in the future.
- Getting Started Guide [https://wiki.eclipse.org/Papyrus/customizations/robotics#Getting_started]

Capabilities

Papyrus for Robotics uses UML/SysML as underlying realization technology. The platform uses the UML profile mechanism to enable the implementation of DSMLs that assist RobMoSys' ecosystem users in designing robotics systems.

Papyrus for Robotics features a **modeling front-end** which conforms to RobMoSys' foundational principles of separation of roles and concerns. It provides custom architecture view points, diagram notations and property views, including (but not limited to) those for the definition of software components, services, the representation of the system's architecture, and coordination and configuration policies.

Below's a list of additional capabilities — already available or for which the integration in *Papyrus for Robotics* is currently on-going.

- **Safety Analysis** (available). *Papyrus for Robotics* features a safety module to perform dysfunctional analysis on system architecture's components, including (but not limited to) Failure Mode and Effects Analysis (FMEA) and Fault Tree Analysis (FTA). Model-based safety analysis is enabled by a dedicated DSML, modeling views and a set of analysis and report generation modules.

- **Performance Analysis** (integration on-going). *Papyrus for Robotics* supports the Architectural Pattern for Stepwise Management of Extra-Functional Properties, with a focus on the timing properties of software component architectures. Concretely, it enables the analysis of end-to-end response times and resource utilization of component compositions by leveraging concepts in Compositional Performance Analysis (CPA), such as paths and cause-effect chains.
- **Code-Generation** (integration on-going). *Papyrus for Robotics* includes generators that transform models of software component architectures, platform descriptions and deployment specifications into code. The eITUS Integrated Technical Project (ITP) [<https://robmosys.eu/e-itus/>] uses the *Papyrus for Robotics* code-generation capabilities to target Orocos-RTT [<http://www.orocos.org/rtt>] and Gazebo to enable early safety assessment of robotics systems using fault injection simulations.

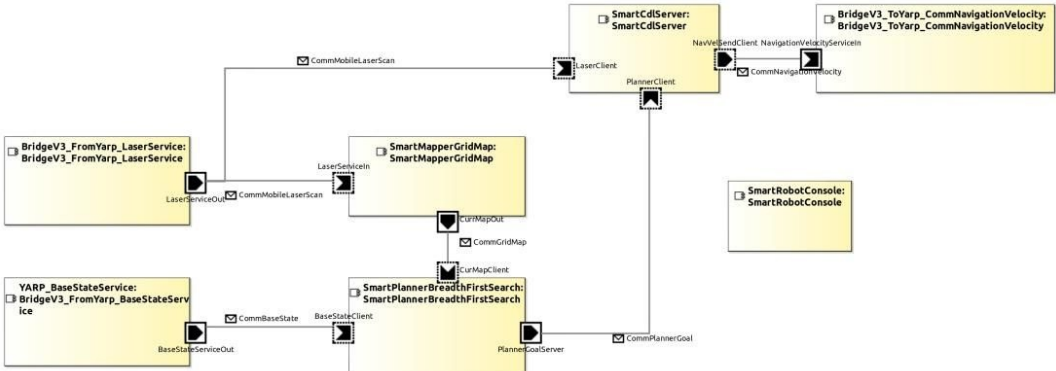
Further Resources

- RobMoSys Use Cases
 - Safety Assessment of Robotics Systems Using Fault Injection in RobMoSys (eITUS ITP)| Video [<https://www.youtube.com/watch?v=YrwJ-GTVACw>]
 - Papyrus for Robotics Overview and Safety Analysis Example[http://www.servicerobotik-ulm.de/models2018/assets/slides/RobMoSys_MODELS_Papyrus4Robotics.pdf] @ The RobMoSys Tutorial (MODELS 2018 conference in Copenhagen)
 - Component Algorithm Development using Fast Transition between System Definition and Simulation Models (Simulink) [<https://robmosys.eu/download/robmosys-tooling-and-concrete-usage-scenarios-papyrus4robotics-matteo-morelli-cea/?wpdmdl=17132>]
- More videos on Papyrus Companions[https://www.youtube.com/channel/UCxyPoBIZc_rKLS7_K2dtwYA]

baseline:environment_tools:papyrus4robotics · Last modified: 2019/06/13 13:27
http://www.robmosys.eu/wiki/baseline:environment_tools:papyrus4robotics

YARP-RobMoSys Mixed-Port Components

A set of software components for interoperability between YARP and SmartSoft as developed by the [CARVE ITP](https://robmosys.eu/carve/) [<https://robmosys.eu/carve/>].

Authors	Alberto Cardellino, Michele Colledanchise, Lorenzo Natale (CARVE ITP)
Website	https://github.com/CARVE-ROBMOSYS/Yarp-SmartSoft-Integration [https://github.com/CARVE-ROBMOSYS/Yarp-SmartSoft-Integration]
License	BSD-2Clause
Screenshot	

Description

This package contains software components for execution of Behavior Trees in SmartSoft, as well as components and messages for YARP-SmartSoft integration. In particular:

- **Bridges:** SmartSoft components aimed to work as specialized bridges between pure YARP executable and pure SmartSoft components. They translate a message from a framework to the other one, by following proper communication patterns on each side.
- **BehaviorTrees:** This folder contains a SmartSoft message defining the tick semantic and two SmartSoft components. The first component runs the behaviour trees and send the tick messages using the query pattern. The second component, the TickManager receives the tick requests from the engine and dispatch them to the actual skills. This components acts also as a bridge in case the requested skill is implemented in a different framework.
- **Integration:** This folder contains the software library based on YARP which reproduce the RobMoSys communication pattern and some examples.
- **Components and Systems:** Contains the CARVE scenarios components and description as a RobMoSys System Architecture. This MDSD project contains an instantiation of all the SmartSoft components required to run the demo along with the component connections. In order to run the demo, the robot or the simulator is required along with some additional YARP modules.

Conformance to RobMoSys

- This software package contains software tools that allow YARP integration within Smartsoft.
- It has been developed with the RobMoSys-conformant SmartMDSD Toolchain.

Relation to other RobMoSys assets

- Components are composable with SmartMDSD Toolchain components

Further Resources

- Code and installation instructions are available on the GitHub page: <https://github.com/CARVE-ROBMOSYS/Yarp-SmartSoft-Integration> [<https://github.com/CARVE-ROBMOSYS/Yarp-SmartSoft-Integration>]
- Demo in the community corner: [Using the YARP Framework and the R1 robot with RobMoSys](#)

baseline:environment_tools:yarp-mixed-port:start · Last modified: 2019/07/24 13:44
http://www.robmosys.eu/wiki/baseline:environment_tools:yarp-mixed-port:start

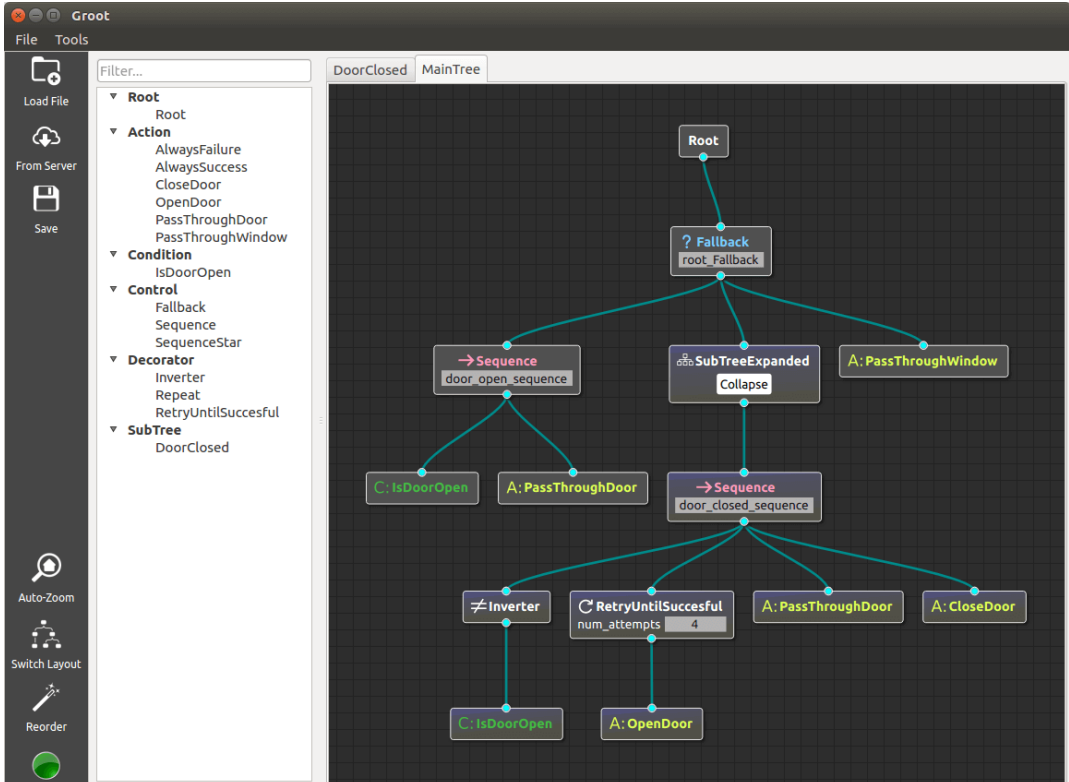
Getting Started With Papyrus4Robotics

This page is a placeholder. Please refer to [Papyrus/Customizations/Robotics](https://wiki.eclipse.org/Papyrus/customizations/robotics)
[<https://wiki.eclipse.org/Papyrus/customizations/robotics>].

baseline:environment_tools:getting_started_with_papyrus4robotics · Last modified: 2019/05/20 10:49
http://www.robmosys.eu/wiki/baseline:environment_tools:getting_started_with_papyrus4robotics

Groot

Groot, an IDE to create, modify and monitor BehaviorTrees.

Authors	Davide Faconti, Eurecat
Website	https://github.com/BehaviorTree/Groot [https://github.com/BehaviorTree/Groot]
License	MIT
Screenshot	

Description

Groot is an optional graphical application that can be used together with BehaviorTree.CPP (BT.CPP) to improve the efficiency and productivity of the Behavior Developer.

In software programming, it is possible to write code using a plain text editor, but a modern IDE allows the software developer to be much more productive. Similarly, any behavior tree is ultimately represented in a XML format, that can be edited by hand, but a tool like Groot provides many additional features and make the development process easier, more reliable and, ultimately, more enjoyable.

Features

Groot can be used to either:

- Create or Edit behavior trees that are executed by the BT.CPP engine.
- Monitor a BT.CPP engine in real-time, showing the current state of the robot.
- Visualize off-line logs recorded using the BT.CPP engine.

Relation to other RobMoSys Tools

Groot can load the palette of available Actions looking at the set of available skills in the SmartMDSD Toolchain.

Further Resources

Further Resources

- MOOD2Be Integrated Technical Project [<https://robmosys.eu/mood2be>]
- https://behaviortree.github.io/BehaviorTree.CPP/BT_basics/
[https://behaviortree.github.io/BehaviorTree.CPP/BT_basics/]

baseline:environment_tools:groot · Last modified: 2019/05/20 10:49
http://www.robmosys.eu/wiki/baseline:environment_tools:groot

Page-Status: Incubator

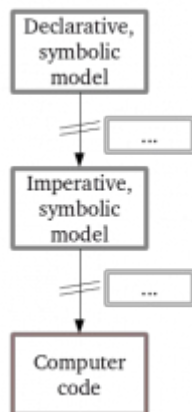
This page is under consolidation and not yet part of the RobMoSys Body of Knowledge. [Read more.](#)

The Kin-Gen code generator

Kin-Gen is a code generator for kinematics solvers for articulated robots, such as industrial manipulators or legged machines.

It is a command line tool(chain), taking as input a **robot model** and a user query; the first is a description of the connectivity and geometry of the robot, the second is a list of desired solvers (algorithms) to be implemented.

Motivation



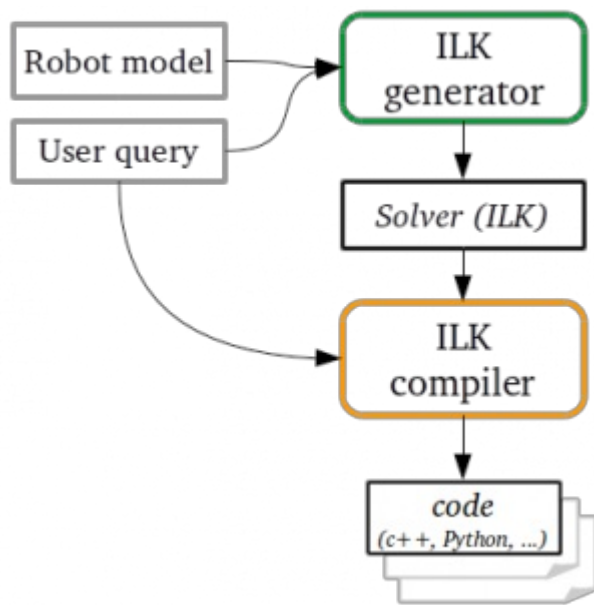
Implementing a solver involves numerous choices pertaining to technologies and mathematical formalism: the programming language and the layout of the digital data structures, but also the representation of orientation or the coordinate frames, among the others. We shall call them *grounding choices*, as they allow to ground an abstract specification (a *declarative symbolic model*, in the figure) into concrete code.

Traditional APIs provide an implementation for only one set of choices, or a few, as it would be unmaintainable to cover all possibilities. In addition, some of the choices are often *implicit* or undocumented, especially those related to the semantics of mathematical/geometrical objects (vectors, jacobians, coordinates, etc). All these issues make the integration of diverse APIs into an application more difficult.

A code generator tool relying on mechanisms for **model composition** (e.g. a model for the robot connectivity, a model for the geometric primitives, a model for the physical quantity kinds and units of measure), could be very configurable and allow to generate code matching exactly the desired semantics. The generated code itself would still be very poorly composable like traditional APIs, but it would fit the constraints imposed by the target environment (OS, robotics middleware, system developer's taste).

Current toolchain

The figure in this section illustrates the current structure of the toolchain:



Besides the inputs (robot model and query), the figure shows the two components of the toolchain, the Generator and the Compiler, along with an intermediate artefact called ILK-Solver. ILK stands for Intermediate Language for Kinematics.

The ILK-solver is a key element of the toolchain. It is an *imperative model* of a solver, meaning it encodes the ordered sequence of operations required to compute something, but with no reference to grounding choices whatsoever. Each operation or entity referenced by the ILK-solver model is a **semantic primitive** (in the context of kinematics solvers), like “relative pose” or “relative velocity”; for example, an homogeneous transformation matrix is *not* a pose, but a possible representation of it, and as such it should not be referenced by the model whenever the concept of pose is needed; doing otherwise would introduce further,

unnecessary constraints limiting the spectrum of possibilities for the code generator.

As depicted in the figure, the Generator creates the ILK-model given the robot description and the query, which tells which solvers are of interest (e.g. forward position kinematics); the query contains a *declarative model* of a solver, i.e. some symbolic references which can be interpreted as a kinematics solver for the given robot (e.g. “forward kinematics: position of the origin of the end-effector frame with respect to the origin of the base frame”). The Generator exploits robotics-specific knowledge to turn the declarative model into the imperative model.

The explicit and concrete representation of the imperative solver, which we called ILK-solver or ILK-model, allows to decouple the robotics-specific knowledge about kinematics solvers (in the Generator) from the logic to ground the ILK-solver into actual computer code (in the Compiler). The compiler interprets the solver model and its configuration options, and generates corresponding code in a target programming language. The Compiler does not know anything about the robot model, and can interpret ILK-models in isolation.

This approach is borrowed from the domain of traditional compilers, which turn code into machine instructions. What we are aiming to do with the ILK-Compiler is the same thing but at a higher level of abstraction, where the “programming language” (the ILK) is really a DSL for kinematics (and eventually dynamics) solvers.

Features and limitations

The toolchain is undergoing active development, thus this list is going to change.

At the moment, the tool supports robots with tree topology, with prismatic and revolute joints. The query allows to request for inverse kinematics and forward kinematics solvers; forward solvers can compute relative positions, velocities, and geometric Jacobians.

At the moment, configurability of the compiler is limited to the target programming languages; C++ (with Eigen), Python (Numpy) and Julia are currently supported. The generated API, in terms of number of functions and outputs for each function, is completely customizable via the query.

Configurability in terms of digital data types and mathematical representations will be added by integrating another, complementary tool [TODO link to DConv page].

Research

Current research related to the Kin-Gen tool is focused on the identification of the semantic primitives required to express kinematics/dynamics solvers as generically as possible, i.e. as “ungrounded” as possible. Every common operation related to the motion of multi-body systems, like the differentiation of a velocity vector, has to be exhaustively and explicitly modeled, by: separating the coordinate-free semantics from the coordinate-vector-specific operations, identifying every involved entity (e.g. a body velocity is always relative to another body), identifying the policies and choices which might erroneously be considered as constraints, but are in fact customary (e.g. the body velocity computed via a Jacobian need not use base frame coordinates).

baseline:environment_tools:kin-gen · Last modified: 2019/06/25 15:03
http://www.robmosys.eu/wiki/baseline:environment_tools:kin-gen

CARVE Software for verified execution of Behavior Trees

A verified interpreter for Behavior Trees. This package contains a compiler that generates the interpreter that execute a given Behavior Tree. The interpreted can be executed in any C/C++ code or as SmartSoft component.

Authors	Alberto Tacchella, Alberto Cardellino, Armando Tacchella, Lorenzo Natale, Michele Colledanchise, Stefano Soffia (CARVE ITP [https://robmosys.eu/carve/])
Website	https://github.com/CARVE-ROBMOSYS/BTCompiler [https://github.com/CARVE-ROBMOSYS/BTCompiler]
License	BSD-2Clause
Screenshot	-

Description

The goal of this package is to provide an execution engine for Behavior Trees which can be integrated into the SmartSoft framework and which provably conforms to the operational semantics for BTs (as defined in the deliverable D3.2 of the ITP CARVE, Syntax and Semantic of Behavior Trees for Robotics). In order to achieve this goal we translated the abstract operational semantics into a concrete algorithm written in Gallina, the language of the Coq proof assistant. This algorithm is then extracted into actual executable code (written in the OCaml programming language) using Coq's program extraction mechanism. Finally, the generated OCaml code is compiled into a shared library which can be called from the SmartSoft runtime.

The input to the compiler is a BT described using the Groot (<https://github.com/BehaviorTree/Groot> [<https://github.com/BehaviorTree/Groot>]) syntax as xml.

Conformance to RobMoSys

This work is conformant to the RobMoSys Task-Skill metamodel. The BT interpreter is an engine for a Task, described using a Behavior Tree syntax. Leaves within a BT represent Skills.

Relation to other RobMoSys assets

Behavior Trees can be designed using the Groot editor, because the syntax used to generate the interpreter is compatible. The BehaviorTree C++ library is an alternative execution engine for behavior Trees.

Further Resources

- Code and installation instructions are available on the GitHub page: <https://github.com/CARVE-ROBMOSYS/BTCompiler> [<https://github.com/CARVE-ROBMOSYS/BTCompiler>]

- Code for executing the engine within SmartSoft is available here: <https://github.com/CARVE-ROBMOSYS/Yarp-SmartSoft-Integration/tree/master/BehaviorTree> [<https://github.com/CARVE-ROBMOSYS/Yarp-SmartSoft-Integration/tree/master/BehaviorTree>]
- Check out CARVE official website for information on the project and how the BT engine has been used with the R1 humanoid robot to solve navigation and grasping task.
- [Robotic Behavior Metamodel](#)
- [Skill Definition Metamodel](#)

baseline:environment_tools:carve-interpreter · Last modified: 2019/07/25 14:03
http://www.robmosys.eu/wiki/baseline:environment_tools:carve-interpreter

IDEs & Toolchains

Placeholder page. Please refer to [Tools and Software Baseline](#).

baseline:environment_tools:start · Last modified: 2019/05/20 10:49
http://www.robmosys.eu/wiki/baseline:environment_tools:start

Gazebo/TIA Go/SmartSoft Scenario

Please note: this scenario is currently under maintenance and will not work out of the box when using from the virtual machine. Please contact us if you are interested.

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.

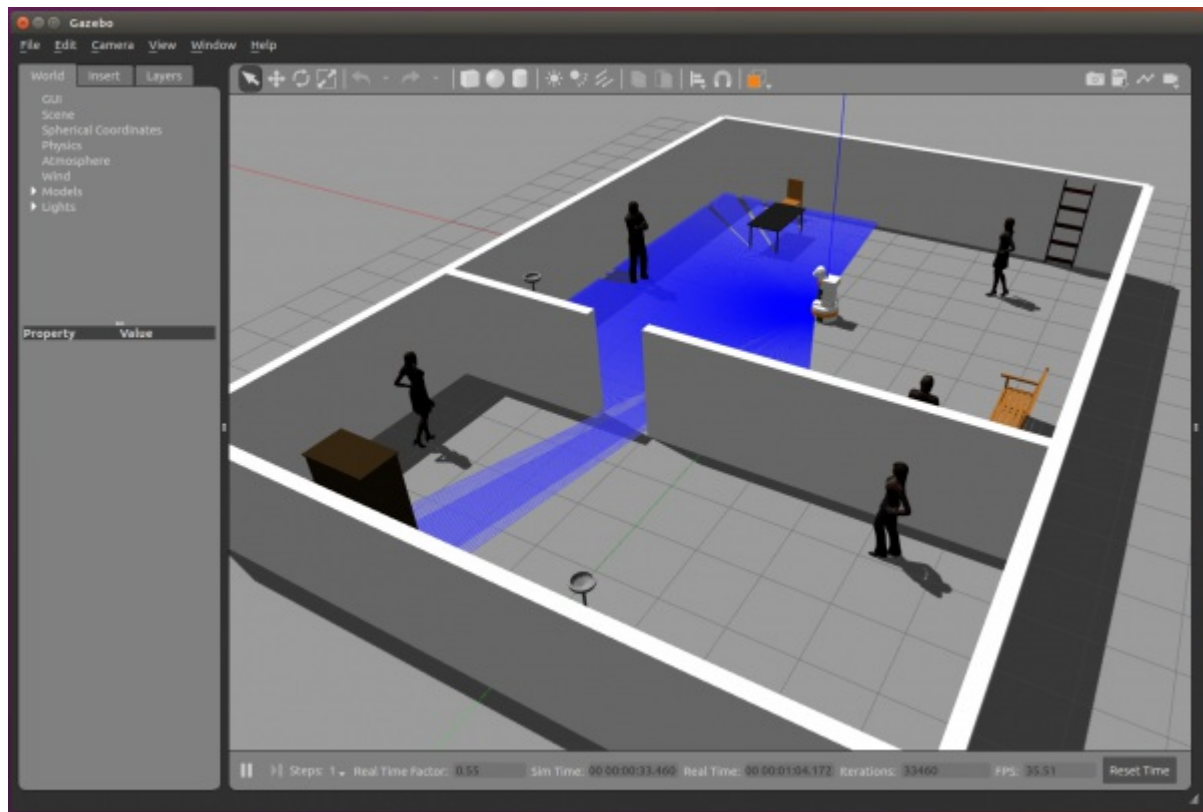
The TIAGo robot platform in simulation can be used with the SmartMDSD Toolchain as available software for the open calls where we emphasize: “do not re-invent in open call projects but build on existing technologies and tools”.

The scenario includes:

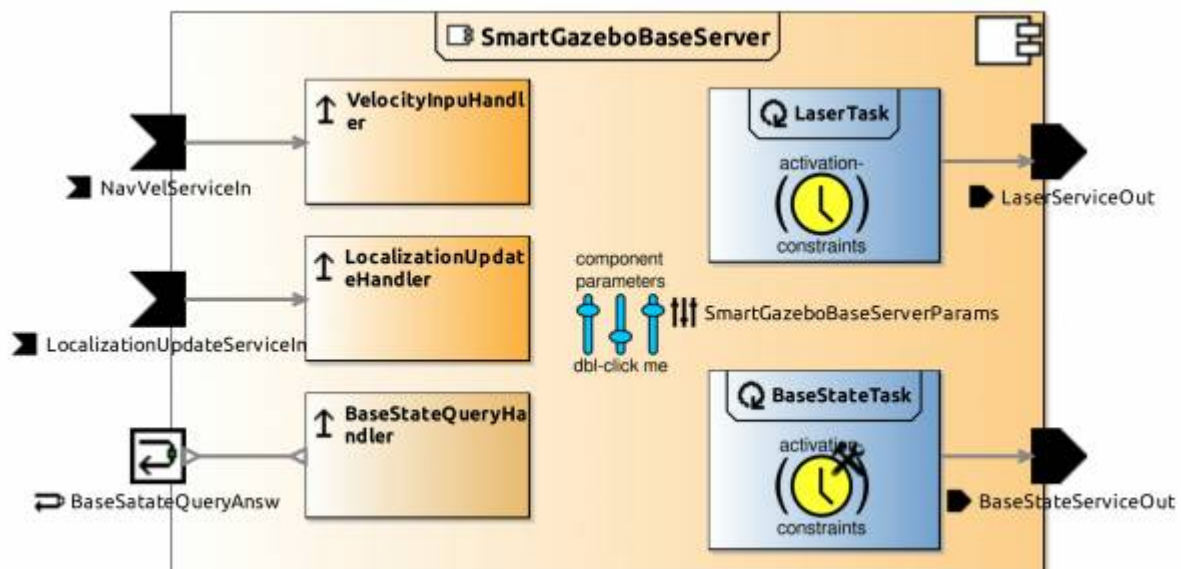
- Navigation Stack: obstacle avoidance (CDL), recording maps with Gmapping, localization, path planning
- SmartTCL for behavior coordination to move TIAGo around in the gazebo simulator

Available Baseline: Gazebo/TIAGo with the SmartMDSD Toolchain v3

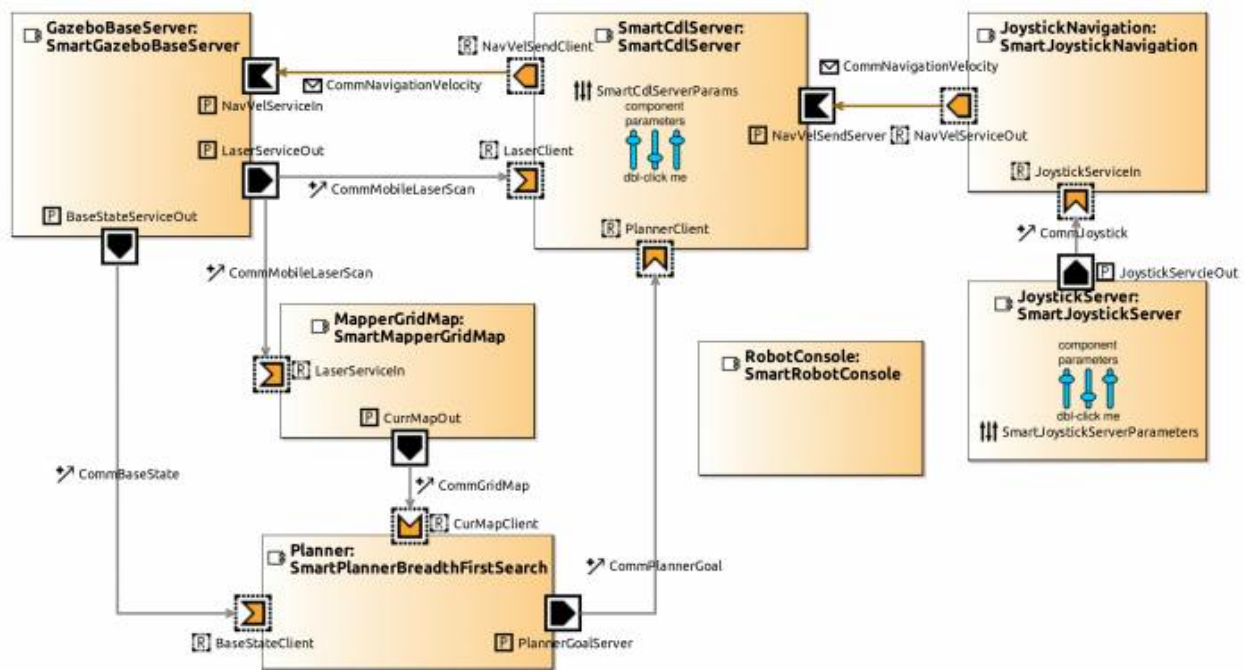
The models and components to run the Pal-Robotics TIAGo using SmartSoft/SmartMDSD Toolchain within Gazebo are available in the SmartMDSD Toolchain v3 Virtual Machine as described here. If you are interested in trying out the scenario with the SmartMDSD Toolchain v2, please refer to <http://www.servicerobotik-ulm.de/drupal/?q=node/91>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



Open the SmartMDS Toolchain in the virtual machine and take a look at the components. The main software component that interacts with the [Gazebo Simulation](http://gazebo.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 (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.

To run the full scenario, launch the SmartMDS Toolchain, right-click on the Eclipse project SystemTiagoNavigation, select SmartSoft Build Tools, click Deploy. The scenario control menu will appear. Choose menu-start with your keyboard and hit enter to start the system.

Wait until the Gazebo simulation starts, loads the Tiago models and all the navigation components 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 neighbour room on the right.

To stop the scenario, choose menu-stop from the scenario control menu.

See also

- This scenario is featured in Robotic Behavior in RobMoSys using Behavior Trees and SmartSoft
- A variant of this scenario is featured in Using the YARP Framework and the R1 robot with RobMoSys

baseline:scenarios:tiago_smartsoft · Last modified: 2019/10/22 13:59
http://www.robmosys.eu/wiki/baseline:scenarios:tiago_smartsoft

Tools and Software Baseline

RobMoSys provides a set of tools and a software baseline that conform to the RobMoSys approach. This set can serve as a starting-point for applying the RobMoSys methodology or to extend it.

Tooling

- [RobMoSys Tools, Assets and their Conformance](#)
- [Development Environments and Tools](#)
 - [The SmartMDSD Toolchain: An Integrated Development Environment \(IDE\) for robotics software](#)
 - [Papyrus for Robotics: A set of Papyrus-based DSLs and tools](#)
 - [Groot: an IDE to create, modify and monitor BehaviorTrees](#)
 - [BehaviorTree.CPP: a C++ framework to design, execute, monitor and log robotics behaviors, using Behavior Trees](#)
 - [RoQME Plugins for the SmartMDSD Toolchain: Tooling to enable modeling and monitoring of QoS in robotics systems](#)
 - [eITUS Safety View for Papyrus4Robotics](#)
- [Roadmap of Tools and Software](#)

Tutorials and Documentation

- [For the SmartMDSD Toolchain](#)
- [For Papyrus for Robotics](#)

Usable Domain Models, Components, and Systems

- Browse the [Model Directory](#) to see building blocks available for immediate composition with RobMoSys tooling.

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.

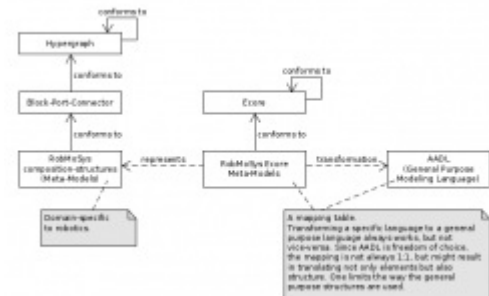
Within the context of UML the term “*semantic variation point*” has been coined to express the purposeful semantic ambiguity for certain UML elements. Because UML is a general purpose modeling language, this semantic ambiguity makes sense and can be narrowed within the derived domain-specific models 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: 2019/05/20 10:47
http://www.robmosys.eu/wiki/other_approaches:modeling_languages

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.

- 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



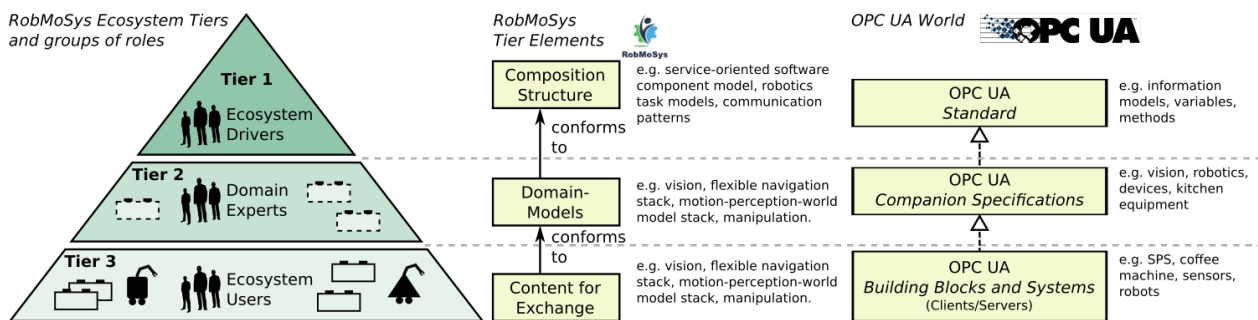
other_approaches:start · Last modified: 2019/05/20 10:47
http://www.robmosys.eu/wiki/other_approaches:start

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.



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> [<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> [<https://opcfoundation.org/markets-collaboration/vdma-robotics>]
- [BMW/PAiCE Project “Service Robot Network”](https://www.seronet-projekt.de): <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: 2019/05/20 10:47
http://www.robmosys.eu/wiki/other_approaches:opc-ua

Page-Status: Incubator

This page is under consolidation and not yet part of the RobMoSys Body of Knowledge. [Read more.](#)

wiki-process:incubator-label-include · Last modified: 2020/01/23 09:44
<http://www.robmosys.eu/wiki/wiki-process:incubator-label-include>

Wiki Publishing Process

Information on how the RobMoSys wiki publishes information will be found here. This page is under internal preparation.

The label **Incubator** on a page marks contributions as “drafts” or “proposals” that are not yet entirely consolidated for RobMoSys but that are intended to be included soon in the “RobMoSys Body of Knowledge”

wiki-process:start · Last modified: 2019/06/25 15:00
<http://www.robmosys.eu/wiki/wiki-process:start>