ITEA 2

INFORMATION TECHNOLOGY FOR EUROPEAN ADVANCEMENT

FIONA

FIONA Deliverable D2.2.1
State of the Art on Service-Oriented Software Component Models

Final Version of Deliverable

.....

Document status:	Final version	
Dissemination level:	Public	
Version:	1.0	
Submission date:	31.3.2014	
Editor	Dennis Stampfer	Univ. of Applied Sciences Ulm (HSU)
Contributors:	Dennis Stampfer Alex Lotz Christian Schlegel	Univ. of Applied Sciences Ulm (HSU) Univ. of Applied Sciences Ulm (HSU) Univ. of Applied Sciences Ulm (HSU)

Supported by

Federal Ministry of Education and Research

Germany – Federal Ministry of Education and Research (BMBF)

Slovenia – Ministry of Economic Development and Technology (MGRT)

MŠMT

Czech Republic – Ministry of Education, Youth and Sports (MSMT)

Document History

Version / Date	Author	Remarks
V 1.0 / 31.3.2014	Dennis Stampfer	Initial version of deliverable.
--	Dennis Stampfer	Merged title page to template for public version.

Table of Contents

1. Introduction

FIONA aims to develop a modular, accessible framework to support the core functions of localisation and navigation in indoor and outdoor areas as well as facilitate the development of applications and services to be built upon them. Therefore, it will tailor a Service-Oriented Software Component Model to the needs of FIONA. This document presents FIONA deliverable D2.2.1 as the outcome of task T2.2 in WP2. It provides an overview of the state of the art technologies on Service-Oriented Software Component Models.

Several existing technologies are investigated and compared. First, the targets of FIONA with respect to Service-Oriented Software Component Models (SOSCM) and their relations are described. Then, the Unified Component Model (UCM) is addressed as a collection of requirements for next-generation state of the art approaches. For each technology, the specific scope and background as well as the most important details of the methodology are described in the context of FIONA goals and UCM in order to identify their progress towards UCM and FIONA goals.

1.1. FIONA Goals/Targets/Definitions

This section lists and defines aspects that the FIONA platform addresses. These aspects provide a set of terminology and guide the research and description for state of the art technologies.

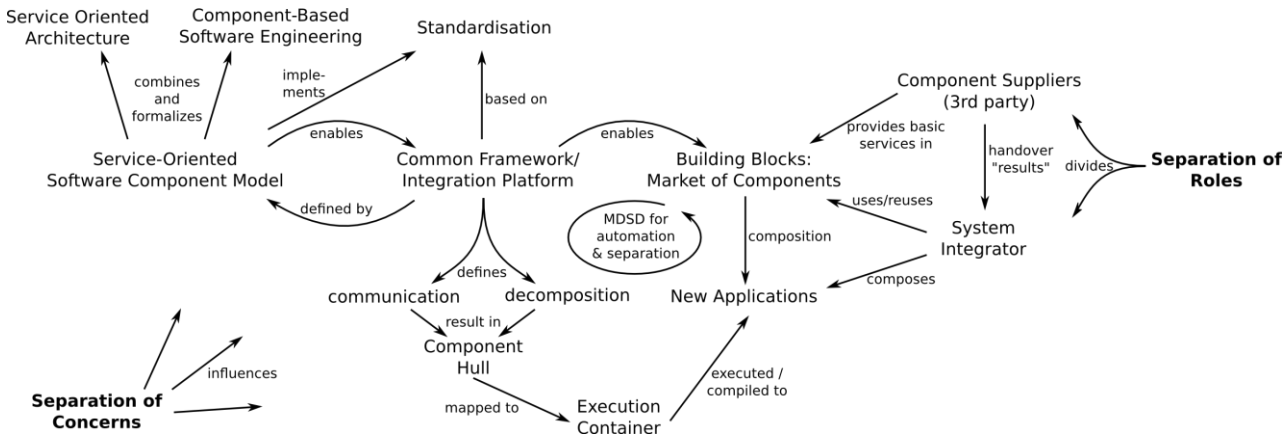

Figure 1 Terminology and relations

1.1.1. Software Integration Platform

A major requirement on the software integration platform is to provide the infrastructure for implementing the various methodologies and technologies ("FIONA basic services") in software components of FIONA and to support the composition of these software components to applications, both in-house and from third party suppliers.

Such a platform must provide a definition of software elements and their interaction and communication in a Service-Oriented Software Component Model which is the basis for an common framework for integration. Within the structure of the integration platform, decomposition of entities in components and communication among them has to be defined. These concepts result in a component hull (Figure 2) that maintains stable interfaces for the involved roles (e.g. developer and integrator).

Mappings towards the underlying infrastructure (e.g. operating system, communication middleware) are provided by the execution container. Programming towards a component hull prevents vendor lock-in on specific platforms.

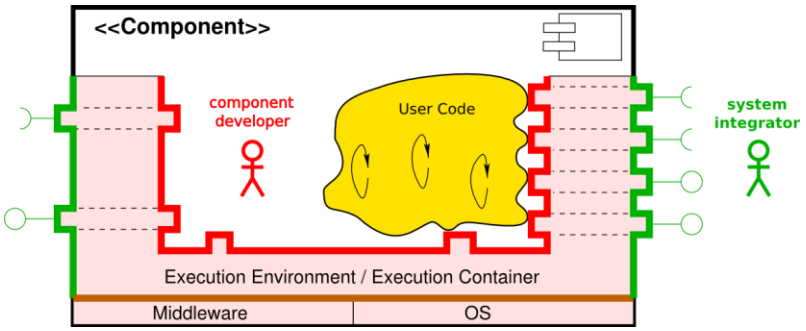

Figure 2: Component and its structure

1.1.1.1. Communication middleware

A communication middleware (e.g. OMG CORBA, OMG DDS, ACE) is a software layer between application and network stack of the operating system. They 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 *of* 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 tools. 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. Middleware independence has also been recognized by OMG in the Unified Component Model and has been included as requirement.

1.1.1.2. Component Based Software Engineering

Component Based Software Engineering (CBSE) is an approach for software engineering which has gained wide acceptance during the last decade.

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be developed independently and is subject to composition by third parties.” [1].

CBSE separates component development (functionality, technically driven) and system development (system integration, application driven). It shifts the system development process to reusing as many off-the-shelf components as possible and to develop only the small delta between reused components and application in mind. CBSE explicitly addresses reuse and it is therefore necessary to address composition when using principles of CBSE.

[2] [3] [4] provide a broader overview of robotics frameworks and CBSE in robotics.

1.1.1.3. Service-Oriented Architecture

Services “combine information and behaviour, hide the internal workings from outside intrusion and present a relatively simple interface to the rest of the program” [5]. A service based view defines a granularity of the entities in a component based approach. Services are self-contained and meaningful entities in a system. They provide a certain functionality using defined communication contracts. A Service-Oriented Architecture (SOA) has to ensure that services in the system do not get reduced to interfaces.

Service-Oriented Architectures (SOA) are "the policies, practices, frameworks that enable application functionality to be provided and consumed as sets of services published at a granularity relevant to the service consumer. Services can be invoked, published and discovered, and are abstracted away from the implementation using a single, standards-based form of interface" [5].

The term "Service" in the context of SOA is not to be mixed with "FIONA Basic Services". "FIONA Basic Services" are methodologies, technologies and implementations of e.g. localisation, perception, navigation and might be provided to the system by "SOA Services".

1.1.1.4. Service-Oriented Software Component Model

A Service-Oriented Software Component Model (SOSCM) combines and formalizes the principles of SOA and CBSE. This way, the architecture benefits from the composability and reuse of the system parts while at the same time ensuring interoperability of services with stable interfaces and proper abstraction.

A model represents and defines structure and elements of a system or methodology in a formal way. A component model is a standardisation, typically implemented in a common framework which is the basis for an integration platform. A common component model is mandatory for systematic reuse at component and service level and is an important step towards a market where component (third party) suppliers and integrators benefit.

A component model has to define how entities of the system are described as well as to specify how they interact. Component models should be described independently of implementations and therefore also independently of specific (meta-model) platforms. Becoming independent of platform and implementation, introduces a new level of abstraction that allows to integrate them to new platforms and create new areas in a market.

1.1.2. Separation of Roles and Separation of Concerns

1.1.2.1. Separation of Concerns

Separation of concerns is one of the most fundamental principles in software engineering. [6] describes separation of concerns with respect to the robotics domain. As robotic systems are particularly complex software systems, including lessons learned from this domain is relevant to FIONA.

Separation of concerns is a general strategy how to break problems into smaller ones, solve them separately and combining the results that solve the problem in the first place. By solving separate problems, one can focus on the individual problem at hand without requiring any knowledge about other parts or problems. Separation of concerns requires to identify the right granularity of decomposition of the original problem.

With respect to SOSCM, this means that every entity in the system (component, service), solves one particular problem or task. It can be used and seen independent of other entities which are not part of the same (sub)problem. By separating the concerns of computation, communication, configuration and coordination, it allows for example maintaining stable interfaces in the execution container, alternative implementations (mapping the execution container onto different OS and middlewares), etc. [6].

1.1.2.2. Separation of Roles

As there are many different roles involved in a development process, every role must be able to focus only on its specific task without requiring knowledge of other tasks. Important roles in the context of FIONA are component suppliers who provide basic services with components, such as localisation, obstacle detection or navigation. Another important role is the system integrator, who composes off-the-shelf components to new applications. However, separation of roles requires methods and tool support to handover artefacts and in particular knowledge between roles.

Separated roles need individual views on problems and models (parts of the overall system). For example, a component developer is responsible and therefore interested in the inner structure and implementation of a component (component as white-box) while the system integrator is interested in composing new applications by reusing building blocks (components as black-box).

Apart from few exceptions, it is still state-of-practice in robotics, that roles involved in the development of robots are too tightly coupled and every involved person needs to be an expert in every area. This ranges from algorithms over expertise of the application domain up to the final integration. Integrated scenarios are driven by technical achievements, rather than an integration methodology.

However, an expert cannot be an expert in every field and therefore cannot become a player in all markets of the different application domains. Lack of separation of roles is therefore a severe show-stopper towards a market [7]. Separation of roles reduces risks, efforts and costs as well as time-to-market and increases overall robustness of systems. A successful market and business ecosystem depends on separation of roles where building blocks can be handed over as black-box from one role to another, hiding complexity and still ensuring composability.

1.1.3. Model-Driven Software Development

While standardisation and systematic reuse can be achieved through CBSE (cf. [8]), Model-Driven Software Development (MDSD) raises development to the next level. It allows to separate domain knowledge and structure (software design) from implementation (templates in code generators) as well as automating the development process (code generation).

MDSD is not limited to code generation. It puts models into the focus of the development process (models not being documents/paperwork but being computational and the solution itself) and is a tool that:

- provides individual/focused views to separated problems/roles within the whole development process, e.g. component view, architectural view, configuration view, etc.
- separates technical problems from business logic / use cases and therefore
- decouples development in space (roles, teams) but also in time: speedup, automation, time-to-market.

In order to use MDSD for CBSE, the component model has to be specified (implemented) as meta-model for use in MDSD. It is necessary to develop tools for modeling (graphical, textual) the system,

code generation, configuration, checks/validation, compiling and deployment to support the different roles engaged in the overall development process.

Several integrated toolchains and projects exist for MDSD, one of the largest being the Eclipse Modelling Project [9]. It includes for example, Papyrus [10] that allows for graphical representation and modelling in UML as well as Xtext [11] for textual modelling. Tools and concepts behind MDSD target at being generic and extendable, thus being able to add domain specific extensions and being tailored to domain specific needs.

1.1.4. Building Blocks: Market of Components

Concepts and tools for the development of reusable components in a common framework and integration platform is a necessary prerequisite towards a market in which several roles and stakeholders can contribute and make use of. It creates opportunities for component and system developers/integrators, minimises effort and maximises efficiency.

Component developers can provide their components in the market where system developers/integrators can use them for composition of new applications. Within FIONA, it is desirable that the components providing "FIONA Basic Services" (implementations of e.g. localisation, obstacle detection) are introduced in the market. Exchange at the level of such a market must happen at the proper level of abstraction, as is provided by components and services, but not on the level of implementation / libraries. In the long term, new use-cases and business models will be enabled through the re-use of components as well as new components contributed from third party-suppliers.

1.2. Unified Component Model

In September 2013, the Object Management Group (OMG) released a Request for Proposal (RFP) on the „Unified Component Model“ (UCM) [12] [13] [14] [15]. Its goal is to create a new component model for Distributed, Real-Time and Embedded Systems.

Key statements:

- OMG defines a list of requirements towards a next-generation state-of-the-art and is requesting proposals for an Unified Component Model.
- Driven by a big standardisation player (OMG) who has established a variety of component standards (CORBA CCM, RTC, ...)
- Requires a formal component model, specified as meta-model
- Aims to be independent of middleware
- Aims to include multiple communication models (interaction patterns)
- Addresses system configuration and parameterisation
- Considers separation of concerns

Consequence to FIONA:

- The SOSCM used in FIONA (SmartSoft) has influenced and shaped the RFP. Its concepts have been confirmed by the RFP. FIONA concepts are therefore in line with state-of-the-art.
- Developments behind UCM are worth being followed within the scope of FIONA.

UCM is of interest to FIONA because OMG as a big standardisation player, with members covering a wide scope and is expected to combine state-of-the-art and lessons learned in component based

systems into UCM. UCM provides a list of requirements for the next-generation state-of-the-art in component models.

UCM was motivated by critics (complexity, long learning curve, lack of performance and high footprint) [12] of Lightweight CORBA Component Model (LwCCM) and is an evolution thereof. The CORBA component model (CCM) collected best practices and common use-cases of the CORBA middleware in a component model, mainly focusing on enterprise applications. LwCCM uses the CCM and tailors it to embedded systems. UCM follows the goals of LwCCM but plans to include trends and state-of-the-art in distributed, real-time and embedded systems.

The UCM RFP explicitly references SmartSoft as one of the component standards that need to be considered for UCM. SmartSoft can therefore be considered state-of-the-art. It is expected that the principles and methodologies of SmartSoft influence and contribute to the next-generation state of the art UCM.

At the time of preparing this document, the UCM is in the state of collecting proposals (deadline 19 May 2014). This version of the document focuses on the goals and requirements that UCM shall meet. Updated information on the UCM can be found in [13] and at RemedyIT [14], the driving force behind UCM within OMG.

1.2.1. Software Integration Platform

UCM is targeted to be a „simple, lightweight, middleware-agnostic, and flexible component model“ [12]. UCM aims to keep compatibility or at least portability to existing OMG approaches, e.g. LwCCM and RTC components.

UCM will be independent of a specific middleware standard and/or language (LwCCM depends on CCM and CORBA middleware), but be described in IDL and describe IDL mappings to languages. The RFP states that implementations should be replaceable by alternative implementations without requiring changes.

UCM will include multiple communication models (interaction patterns) which shall also support extension and include at least request/reply, publish/subscribe as well as asynchronous handling of both method invocation and invocation handling on client and service side. It requires the communication model to be separated from components. The RFP refers to software connectors, so it can be assumed that proposals might use the principle behind software connectors.

It considers security and specifically asks policies (authentication, audit, authorization, message protection) and plans to take into account resource-constrained environments (resource awareness).

System parameterization and configuration seems to be a focus, since the RFP asks for initial configuration of component, container and communication elements as well as deployment and runtime configuration. This is not only limited to parameters and configurations, but also by providing a component lifecycle model and its management.

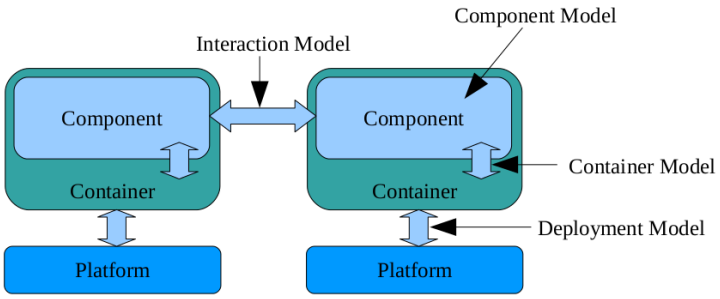

Figure 3: Unified Component Model

1.2.2. Separation of Roles and Separation of Concerns

Separation of roles and concerns is not explicitly mentioned but evident in requirements description and pre-final drafts. Separation is addressed by asking for separation of functional specifications from non-functional specifications, meaning component from component container. This both refers to component hull / execution container as well as inner and outer view on components for different roles.

The draft presentation and talks mention different roles and UCMs impact on them. However, it does not describe or explicitly target their separation.

1.2.3. Model-Driven Software Development

As a component model, UCM does not directly ask or refer to MDSD. However, it requires models to be MOF-compliant. Since this is a basis towards MDSD, it is expected that the implementations or show-cases of proposals will make use of MDSD.

1.2.4. Building Blocks: Market of Components

UCM defines the basis that is required for a market of components, however does not explicitly address it as target. It supports composition of components, but does not address composite components (systems of systems). It tries to keep all elements extensible. This might be a benefit for custom development, but might limit the exchangeability of components and therefore might be contradictory to market needs.

2. Generic Software Component Models

There are several generic or multipurpose frameworks that aim to support CBSE in general (e.g. JavaBeans, CORBA, DDS, etc.). We use CORBA and CORBA related activities as a representative example.

Key statements:

- General purpose.
- Freedom *of* choice instead of freedom *from* choice: too many ways to do one thing, no guidance with respect to component design.
- Even though (Lw)CCM and CORBA are standardized by OMG, OMG is looking for a new standard: Unified Component Model.

Consequence to FIONA:

- Following freedom *from* choice instead of freedom *of* choice will provide guidance for developers and integrators, e.g.:
  - Separation of concerns
  - Communication semantics (communication patterns)
  - Abstraction layers

Generic Software Component models usually base on the concept of freedom *of* choice, providing freedom and alternatives with respect to defining and implementing component interfaces and functionality. However, it has severe drawbacks as they are designed to support as many different styles of programming and as many use-cases as possible. As a result, there is a overwhelming number of ways on how to implement even a simple two-way communication using one of the provided generic purpose tools. Unfortunately, the various options result in completely different behaviours at the system level architecture.

In contrast (e.g. SmartSoft) follows the idea of freedom *from* choice instead of freedom *of* choice [16]. In this term, a developer can expect assistance by strictly reducing the number of offered alternatives such that he can rely on system level conformance of his contributions as long as he sticks to the imposed restrictions.

2.1. (Lightweight) CORBA Component Model

CORBA (Common Object Request Broker Architecture) is one of the most used middleware standards. CORBA and its object model was designed to support as many different programming styles and use-cases as possible. This resulted in too many ways to do one thing and the various options led to different system behaviours as well as [17] tightly coupled and ad-hoc implementations. The CORBA Component Model (CCM) [18] [19] was introduced to overcome these issues. It extends the CORBA object model and integrates commonly used CORBA patterns in a standard environment. Its main contribution is the standardization of component development using CORBA as middleware infrastructure on which CCM depends.

The CORBA Lightweight Component Model (LwCCM) [18] [20] is a profile based on CCM and defines a subset that tailors and simplifies CCM to support constraints of embedded domain, e.g. limited processing, small codebase, distributed, cross language and cross platform. It combines best practices and common use cases to avoid doing one thing in multiple ways. Even though LwCCM relies on

CORBA as underlying middleware, LwCCM is designed to hide CORBA middleware mechanisms from the developer.

2.1.1. Software Integration Platform

The implementation of CORBA Components is described using the Component Implementation Definition Language (CIDL). Components communicate through ports which expose their interfaces to clients. Communication mechanisms include synchronous and asynchronous method invocation, events (loosely coupled asynchronous communication based on the observer pattern), attributes (configuration) and lifecycle control.

CCM defines techniques and structures to implement CORBA servers that can host CORBA components (execution container). Together with the IDL description of the interfaces, component skeletons are generated and compiled with the component implementation. These component programs (such as JAR, dll, shared library, etc.) are executed in component servers.

Component Servers have no prior knowledge how to configure or instantiate the components, but can configure them through a configuration interface. The configuration interface can be extended by components to allow for component specific configuration. The component server can control the components lifecycle and activate or deactivate components to preserve limited resources.

Quality of Service (QoS) is available within Real-Time CORBA [21].

2.1.2. Model-Driven Software Development

CCM models components using the Component Implementation Definition Language (CIDL). Similar to most middlewares, CORBA uses code generation for skeletons/stubs towards the communication as well as implementation skeletons. CORBA and CCM does not directly address MDSD but can be used as target implementation for MDSD.

There are activities that make use of MDSD in relation to CCM. For example, [22] focuses on component composition distinguishing between white-box and black-box view. Components and their relations can be modelled using UML Profiles. MDE tools and concepts are used to transform and generate code against CCM.

The Open CORBA Component Model Platform (OpenCCM) [23] is an open source implementation of CCM. Its toolchains support modelling components, implementations, and assemblies with UML.

3. Robotics Software Component Models

(Service) robotics is a domain with very high technical requirements from a very broad field with big complexity, therefore system integration is very challenging. Many examples of autonomous robots show impressive achievements in this domain. However, most of these are technical demonstrations lacking a conceptual approach for software development. Every player in the robotics community develops systems from scratch over and over again with very limited possibilities for reuse, system composition and separation of roles.

Key statements:

- Not yet a widespread use of systematic software engineering in robotics (e.g. ROS stands out as the largest integration platform in robotics while ignoring industrial needs).
- The need for component models and meta models has been recognized by the robotics domain and is provided by some projects (e.g. BRICS, SmartSoft and PROTEUS).
- Concepts of SmartSoft have contributed to Unified Component Model and shapes activities within Europe (e.g. Topic Group at EU Robotics AISBL, Research Agenda) and are in line with them.
- Robotics is a domain with very high complexity, thus needing separation of concerns, separation of roles, system integration and system composition.

Consequence to FIONA:

- Robotics shows that integration and composition is important as well as for FIONA.
- FIONAs approach to use and extend a Service-Oriented Software Component Model from robotics (SmartSoft) has been confirmed by recent initiatives such as (Topic Group at EU Robotics AISBL, EU Research Agenda/Multi Annual Roadmap).

It has become obvious that (IEEE ICRA SDIR workshop series, EU FP7 BRICS project, workshops at European Robotics Forum ERF, Journal of Software Engineering for Robotics) software development for robotics is a research field (recognized in EU Strategic Research Agenda and Multi Annual Roadmap [24]) and only a design abstraction supporting separation of concerns and separation of roles can adequately address the software complexity.

Within the research activities in Europe, several groups are joining forces in software development within the topic group "Software Engineering, Systems Integration, Systems Engineering" in euRobotics AISBL (topic group coordinators Prof. Schlegel / Hochschule Ulm, Dr. Lafrenz / Technische Universität München) to push these topics towards the next level in accordance to the EU strategic research agenda [24].

An overview on CBSE in robotics and on design principles to enable the development of reusable and maintainable software building blocks is given in [2] [3] [25]. Up to now, many fundamental requirements on CBSE and MDSD are not fulfilled by currently wide-spread robotics software frameworks. The Robot Operating System (ROS) is the most widely used integration platform for robotics applications. However, the most advanced concepts for software development for robotics are the RTC Specification and SmartSoft as is detailed below.

FIONA can build and learn from insights, approaches, methods and tools of the robotics domain in order to come up with a tailored integration platform.

3.1. SmartSoft

The term SmartSoft originally stands for the component based robotics framework that was published in 1999 [26] [27]. Comparable with other robotic frameworks or middlewares, SmartSoft provides communication mechanisms to exchange information between components and provides a component container. However, there are clear and explicit communication semantics (communication patterns) and a clear component model. Both is realized as a meta-model and independent on middleware or implementation. In addition, SmartSoft provides implementations and tools the Eclipse based SmartMDSD Toolchain [27] to provide tool support in an integrated environment.

3.1.1. Integration Platform

The distinguishing factor of SmartSoft is that the communication semantics are implemented in a fixed set of communication patterns which are aligned with the mindset of SOA.

Comm. Pattern	Description		Config. Pattern	Description
Send	one-way communication		**Parameter**	component configuration
Query	two-way request-response		**State/Lifecycle**	activate/deactivate comp. services
PushNewest	1-to-n publish-subscribe		**DynamicWiring**	dynamic component wiring
PushTimed	1-to-n publish-subscribe		**Event**	asynchronous notification
			Monitoring	introspection of components
			(internally based on communication patterns)	

Table 1: SmartSoft communication patterns (left) and coordination/coordination patterns (right).

Similar to other robotics frameworks and middlewares, SmartSoft as well provides both the publish-subscribe and the request-response communication semantics. However, by explicating typical requirements from various robotics use-cases, SmartSoft refines the two generic communication mechanisms into further communication patterns with distinctive communication policies. This provides stable communication policies between components and thus enables reuse of components with stable and self describing services. The communication patterns strictly separate the unambiguous communication semantics between components and the component's internal communication interface, thus separating the sphere of influence between component developers and system integrators. This directly supports a black-box and white-box view for components which is the basis for system integration.

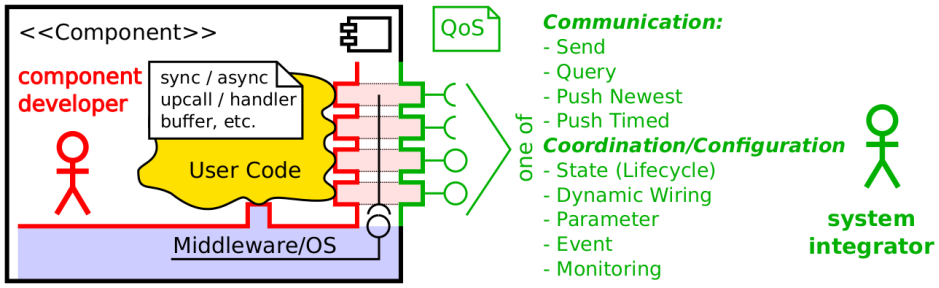

Figure 4: Mastering the link between component inside / outside view by communication patterns

SmartSoft is capable to configure and parameterize components (variability management). Thereby a component developer can define initial default values a component can use for stand-alone execution,

then a system integrator can further refine these values in order to tailor the component to the actual target application and target system. Finally, the parameters can even be configured at run-time (using the configuration patterns in Table 1 on the right) in order to appropriately react to changing run-time conditions.

In SmartSoft, resource awareness was considered an important aspect from the very beginning. For instance, each component consists of a lifecycle state automaton [28] which allows to control and adjust the amount of resources a component is allowed to consume during its execution.

3.1.2. Separation of Roles and Separation of Concerns

Beyond the direct support of component developers and system integrators, SmartSoft also provides an abstraction layer for the underlying communication middleware. This allows to implement the SmartSoft communication patterns on top of any middleware independent of the actual technology. At the moment two main implementations of SmartSoft are available, one based is on CORBA and another more lightweight one is based on pure message passing (ACE). This abstraction allows to build on top of established middleware concepts and at the same time provides stability for the internal implementations inside of components. Component implementations do not depend on the middleware and thus it is able to seamlessly migrate components from CORBA implementation to ACE implementation of SmartSoft. Its flexibility has also been shown by running SmartSoft on iOS (mapping of execution container within FIONA-Project) [29].

The SmartSoft approach and the SmartSoft MDSD toolchain supports and explicitly targets separation of roles as needed in a robotics business ecosystem [30] [7] (Figure 5).

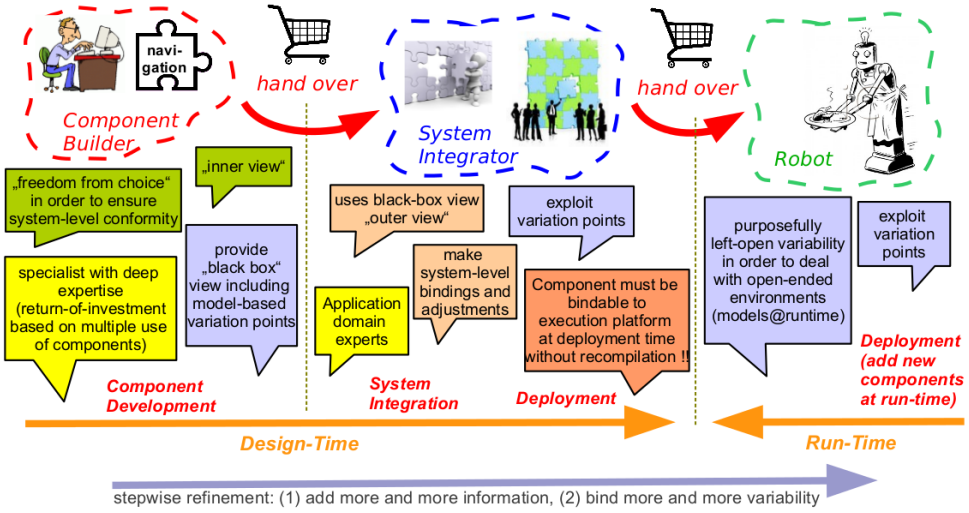

Figure 5: MDSD to manage the hand-over from one role to another role as key ingredient towards a robotics business ecosystem.

3.1.3. Model-Driven Software Development

Principles behind SmartSoft have been explicated in a MDE component model. This component model is implemented in the SmartMDSD Toolchain using the Eclipse Modelling Tools. The toolchain supports the overall development process thereby providing different views according to the different developer roles such as the component developer and the system integrator. In contrast to related approached such as the BRICS Component Model, all DSLs and Models in the SmartMDSD toolchain are integrated into a holistic development workflow. The benefit is that the toolchain not only supports

the different roles to focus on their individual tasks but in particular automates the knowledge handover between these roles by model transformations.

3.1.4. Building Blocks: Market of Components

One of the principles behind SmartSoft is that robotic systems have to be composed from components, rather than programmed. The Service-Oriented Software Component Model therefore allows for strict separation of concerns and separation of roles. Model-driven software development (MDSD) is seen as an enabler to drive robotics towards a business ecosystem for robotics software [7]. The concepts of SmartSoft are implemented in an integrated model-driven toolchain to support the complete development process from component developer over system integration up to runtime aspects.

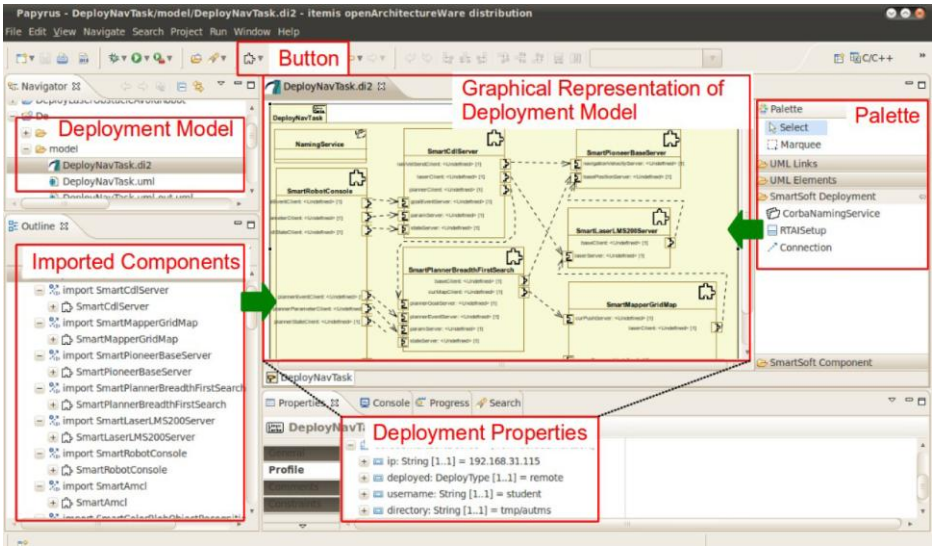

Figure 6: SmartSoft MDSD Toolchain reusing components

SmartSoft is very advanced in providing re-usable off-the-shelf components. It provides components in a repository [31] ready for reuse and composition of new robotic applications through the SmartSoft MDSD toolchain. New applications have been composed from such components for a number of complex robot applications, for example the Robot Butler scenario and other applications on several robot platforms as collected in [32]. System composition has also been successfully applied in a number of projects such as the research project "ZAFH Servicerobotik" where external partners reused and collaborated through components. A logistics showcase re-using SmartSoft components has been shown using the "Robotino 3" with FESTO [33]. SmartSoft and provided components have been exploited by the student project RoboCup. A new team of students reuses existing components in new configurations and composes them to the different challenges in preparation for the competition to the Robocup@Home German Open Challenge. The student teams do not have prior robotics knowledge and the complete team changes every year without overlap.

3.2. BRICS

BRICS (best practice in robotics) [34] was a European project funded under FP7 and finished in February 2013. The main objective of BRICS was to collect best practices in the robot development process from the robotics community. It aimed at exploiting model-driven engineering (MDE) as

enabling approach to reducing the development effort of engineering robotic systems by making best practice robotics solutions more easily reusable. Outcomes of BRICS are:

- BRICS Component Model (BCM), a component model for robotics.
- BRICS Integrated Development Environment" (BRIDE), a model-driven engineering tool for component modelling.
- BRICS Open Code Repository (BROCRE) , a set of tools to browse and install BRICS software.

BRICS contributed to development of ROS as it automated and sped up development. However, at their own opinion it contributed "only in a small part of the overall development process" [35].

3.2.1. Integration Platform

BRICS used ROS as a starting point because it was one of the most popular frameworks at that time. One focus in BRICS was to fill the gaps of ROS in order to overcome its limitations such as the insufficient support for system integration. Thereby BRICS aimed to harmonize all the demands from the robotics community (independent of the individual experience) which as a logical conclusion led to a very generic component model, BCM [36].

The BCM is a valid abstraction for ROS and Orocos. However, it is very generic and is missing important features such as a decent definition of communication semantics or configuration capabilities. Among others, these limitations are the reason why BCM is not able to support a black-box/white-box view for components and thus is insufficient with respect to system integration.

A set of Eclipse based tools was published under the "BRICS Integrated Development Environment" (BRIDE) [37] and allows for creating ROS packages, nodes, coordinators and launch files for deployment. BRIDE uses the BRICS Component Model and provides mappings to ROS and Orocos.

3.2.2. Separation of Roles and Separation of Concerns

In the context of BRICS, separation of concerns is considered important which results in the so called "5Cs" (5 concerns) [36], namely "Coordination", "Configuration", "Composition", "Communication" and as an orthogonal concern the "Composition". Each of these concerns address an important aspect of a system.

3.2.3. Model-Driven Software Development

MDSD (MDE) was in focus in BRICS from the beginning. For instance, BCM is a component model implemented using Eclipse Modeling Tools. An approach based on Software Product Lines (SPLs) to handle similar system variants is presented in [38]. A DSL, basically to abstract concepts defined in the ROS launch files is presented in [39]. All these concepts have in common that they are tightly connected to concepts in ROS and are a clear advancement in the software development of ROS. However, the BCM is not complete and misses for example communication semantics and therefore modeling and development for ROS suffers from BCM limitations.

3.2.4. Building Blocks: Market of Components

BRICS developed an additional tool called BROCRE (BRICS Open Code Repository) which allows to manage code bases from various sources. Basically, BROCRE provides a GUI for several ROS tools for package management (including Debian package management APT and versioning systems such

as SVN, Git, Mercurial, etc.). Thus, similar as in ROS, the exchange takes place on the level of libraries and BROCRE is thus limited with respect to system integration.

3.3. ROS

The Robot Operating System (ROS) [40] is a set of software libraries and tools that help to build robot applications [41]. This distinction in libraries (e.g. OpenCV, PCL, navigation libraries, sensor-drivers, ...) versus ROS itself (system structure, communication, ...) is important. The huge codebase of libraries from ROS (including, for example, navigation, perception, simulation and visualization) might be reused in other applications or frameworks (e.g. within components of SmartSoft). ROS stands out as one of the largest integration platforms with implementations and mappings to several languages and platforms. The popularity of ROS led to a huge variety of new algorithms and solutions of technical challenges in robotics. ROS is a typical representative of the current situation in robotics software.

ROS supports rapid prototyping by providing a set of separated tools for Ubuntu Linux which allow to easily compile custom libraries in a development environment. The development environment is mainly based on CMake and bash scripts. In addition, ROS provides a self-made communication middleware with so called Topics as the main communication mechanism. Topics implement an m:n publish-subscribe communication semantics (which can be compared to the blackboard pattern). Most of the tools in ROS are based on Topics. In addition, ROS provides further special purpose communication mechanisms, such as Services (similar to a synchronous remote procedure call), Actionlib (asynchronous request-response) and Transformation Frames (for transforming coordinate frames).

ROS includes a large set of distributed single development tools, but does not provide an integrated toolchain. ROS is missing an explicated component (meta-) model, which is a key requirement towards system composition. It misses separation at several levels, for example internal and external views of a component. Furthermore, ROS is missing any tool support for system integrators (such as deployment modelling, system composition and initial configuration in launch files, etc.).

3.3.1. Integration Platform

The main focus of ROS is on the unification of the building process for various libraries mostly provided by academic institutions. Therefore, integration takes place on the level of libraries. Even so it is possible to decompose code artefacts into distributed Nodes, a clear encapsulation in the sense of reusable building blocks as well as the black-box/white-box view are not possible with the current structures in ROS. For instance, in order to use Nodes implemented by other institutions, it is necessary to investigate their source code in order to really understand the used communication characteristics.

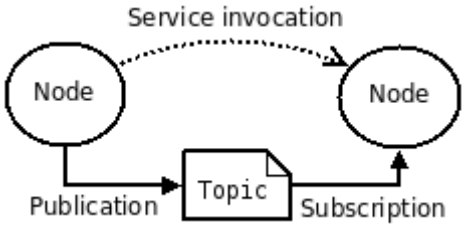

Figure 7: ROS basic concept [42]

Even for ROS, stability (in terms of changes of interfaces and mechanisms), semantics (explicit specification of behaviour and structures in a way independent from implementation) and robustness (in terms of quality control) are still major issues.

ROS provides two mechanisms for communication: (i) a publish/subscribe interaction called topic and (ii) a synchronous request/response interaction called service. In addition, users are allowed to introduce and use a variety of further communication mechanisms provided as add-on libraries (e.g. actionlib) and different styles of using communication mechanisms are encoded as part of the user code. As consequence, node-builders not only bind their user code to different styles and flavours of non-interoperable communication mechanisms but they even use them across components thus violating the concept of stable and interoperable component interfaces. The apparent effect for ROS users of a missing clear separation of user code and framework code are frequently changing APIs. As no clear definition of the different communication mechanisms is available, an abstract component model which is independent of code fragments is not available. Components with individual communication mechanisms become non-separable and cannot be reused separately. Providing components with unprecisely specified interfaces makes it difficult for the application builder to come up with systems with explicated properties like resource requirements.

Although ROS sees its nodes as components, ROS lacks a pivotal property of a component based approach. CBSE requires identified stable structures which provide an execution container and guide the component developer such that he ends up with system level conformance for composability. Instead, ROS supports side-by-side existence of all kinds of overlapping concepts without an abstract representation of the core features and properties. ROS lacks a component model representing its node concept independently of a particular implementation.

3.3.2. Separation of Roles and Separation of Concerns

ROS is missing mechanisms and structures to support different stakeholders such as component developer and system integrator. Component developers are typically technology experts and require a more detailed view on components (white-box view) including the component's internal details whereas system integrators are domain experts and thus typically require a more high level view (black-box view) in order to compose their applications in a buildings blocks manner.

In the ecosystem of ROS, a white-box view on parts of the system is seen as a huge advantage (e.g. [43], to name a recent opinion of a ROS user). The requirement to look inside of building blocks and to adjust them by programming (either because the interfaces are not fitting to the system or because of custom adaptations) is relevant for ROS. However, separation of roles requires black-box view in order to provide building blocks that allow for system composition (putting together black boxes) of new applications, even by third-party integrators.

3.3.3. Model-Driven Software Development

ROS is not using any MDSD techniques. Introducing MDSD in ROS would potentially allow to generate stable structures, thus supporting component developers to use established and tested structures for the component implementation. In addition, component models could be represented in an abstract view for system integrators who on the other hand can use the models for system composition.

Using MDSD for ROS would require a description of the ROS concepts independent from the implementation. The current status of ROS is that there are only code-examples and documentation for usage, but no meta-models.

3.3.4. Building Blocks: Market of Components

Beyond the aforementioned issues, one of the main problems that prevents ROS from being able to establish a component market is their too strong focus on rapid prototyping while underestimating and ignoring typical industrial requirements. One of such requirements is for example resource awareness and defined communication semantics.

Another issue in ROS is related to their custom middleware. Although, ROS core developers invested great efforts to port the underlying communication middleware onto various platforms, yet they are completely ignoring established standards and optimized implementations from the middleware community. This issue is also discussed [44] on the ROS mailing list, where it is proposed to use the Data Distribution Service (DDS) as a basis. Using an established standard such as DDS would not only make ROS much more compatible to other domains, but in particular use industrially accepted technologies. However, at the time of writing, the ROS community seems not to be willing to make this important step in near future.

3.3.5. ROS on Android

Among various platforms that support ROS is android via the java-implementation of ROS [45]. There are individual apps that can connect to ROS nodes on robots such as an app that makes android sensors accessible to ROS [46] or an app that makes a Turtlebot rotate and take panoramic images [47]. The main method however is an app-in-app infrastructure [48]. An special app (ROS app chooser [49]) is able to access ROS apps that provide "android services". ROS uses android only together with a robot for remote control or UI. It targets Android as client platform but does not see Android as a single target platform.

Supporting ROS on android contributes to the ROS ecosystem by providing a new client, platform and even sensors. However, ROS on android does not contribute to system composition in the Smartphone domain. With respect to FIONA, the missing concepts in ROS for system composition prove the necessity of FIONAs goals.

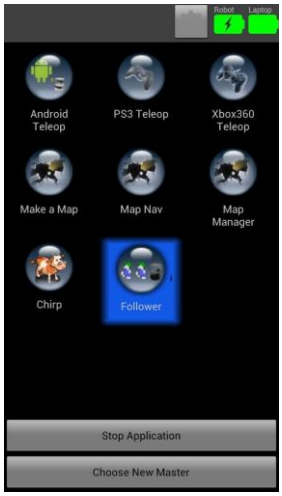

Figure 8: The Android app "ROS App Chooser" [49]

3.4. OMG Robot Technology Component (RTC) / RT-Middleware

Robot Technology Component (RTC) is an OMG standard [50] driven by the Japan's National Institute of Advanced Industrial Science and Technology. RTC formally defines a component model and has several reference implementations, one of the most prominent ones being OpenRTM-aist (aka "RT-Middleware", based on CORBA) [51] [52].

A survey on RTC/RT-Middleware can be found in [28].

3.4.1. Integration Platform

Before the BRICS Component Model and RobotML has been published, RTC and SmartSoft were the only available initiatives for specifying the structures and semantics of a robotic component. Even so, the RTC standard had an impact on the robotics community worldwide and has raised the awareness of needing component models and structures for robotics. Yet, the standard itself was not widely accepted in Europe and USA and remained mainly used in Japan. The reasons for that are manifold.

For example, the CORBA based OpenRTM-aist implementation does not fully hide the CORBA middleware details from the component builder. Thus, the user code contains CORBA code fragments and has therefore a binding to this specific implementation of the specification. The RTC specification is strongly influenced by use-cases requiring a data-flow architecture. Thus, its component model in the current stage is strongly influenced by a strict internal automaton structure that is tightly coupled with the activity model inside a component. For example, it does not easily allow multiple tasks inside a component. Nevertheless, providing an abstract component model is the only way to discuss and compare different robotic concepts and component models with the overall aim of harmonizing the various models without getting stuck in implementation details and at the level of code fragments.

The RTC specification has once been the most advanced concept of a component model and MDSD in robotics. However, the activities are not state of the art anymore, especially after activities like BRICS and SmartSoft.

3.4.2. Separation of Roles and Separation of Concerns

Separation of concerns allows for separation of component model and middleware implementation. Since there are several implementations available for RTC, separation of concerns is adequately addressed. The tools behind RTC generates skeletons for implementations which both guide and restrict developers to stick to the standard and ensure standard compliance.

3.4.3. Model-Driven Software Development

Alongside with RT-Middleware, OpenRTM-aist also provides an Eclipse based IDE. However, this IDE does not use the Eclipse Modelling Tools but customized GUIs and hidden generators which makes it difficult to reuse or adjust (even parts) of the tools.

3.4.4. Building Blocks: Market of Components

RTC has proven that components and meta-models are the right approach for robotics, which is evident through standardisation and the variety of standard compliance industry implementations. However, RTC does not address the needs as identified nowadays in UCM and OMG pushes towards more advanced approaches in UCM, even though RTC was standardized in OMG.

3.5. RobotML

The Project Platform for RObotic modelling and Transformations for End-Users and Scientific communities (PROTEUS) [53] is a recently finished national project in France. The main objective of PROTEUS was to provide a common platform, where results from French academic institutions related to robotics are collected and the transfer to French industrial partners is simplified and supported. Among other tasks, within the scope of PROTEUS two main outcomes are:

- Robotic Ontology to identify and formalize terminologies and requirements in robotics [54]
- DSLs and tools to support the development process, especially RobotML [55] [56]

The main objective of Robotic Ontology was to identify and define a common terminology in the domain of robotics harmonizing deviating terms from similar approaches and projects. The idea is that follow-up projects can use this ontology to create DSLs and tools exposing this terminology that are useful and accessible to other institutions and projects. It is accessible through the Protege tool.

RobotML (Robot Modelling Language) is a Modelling environment based on Eclipse and Papyrus UML tools. In particular, RobotML defines a UML profile that implements ontology aspects related to robotic software architectures (including a component definition called System), robot behavior (mainly based on finite state machines), robotic communication (defining DataFlowPort, ServicePort and connectors) and deployment (abstractly defining the capability to map on different target platforms such as Orocos and ROS).

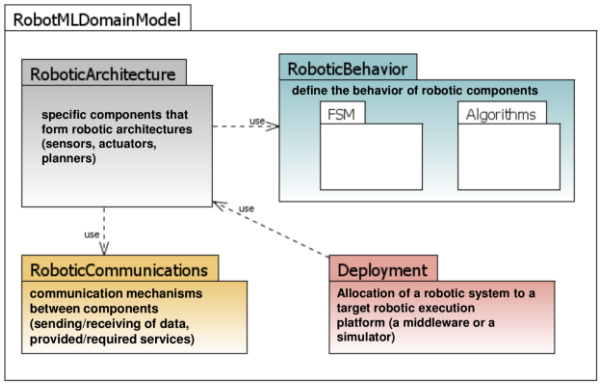

Figure 9: RobotML Domain Model [55]

3.5.1. Integration Platform

On the one hand, RobotML is a reasonable approach to formalize, explicate and abstract over common (often hidden) concepts in frameworks such as ROS and Orocos. On the other hand, RobotML has not achieved to decouple from the ideas and concepts in ROS to a degree that would allow to become independent of the target platform. Instead, RobotML directly maps the terminology defined in the robotic ontology from PROTEUS with the implementation in ROS or Orocos.

One interesting aspect of RobotML is the concept of connectors. At the moment it just defines the synchronization and the buffering policies for communication between components. However, this concept could be extended by more generally formalizing the communication semantics and policies such as for example in SmartSoft. At the time of writing an exchange of ideas between SmartSoft and RobotML developers takes place at the European Robotics Forum within the Robotics Modelling Initiative.

3.5.2. Separation of Roles and Separation of Concerns

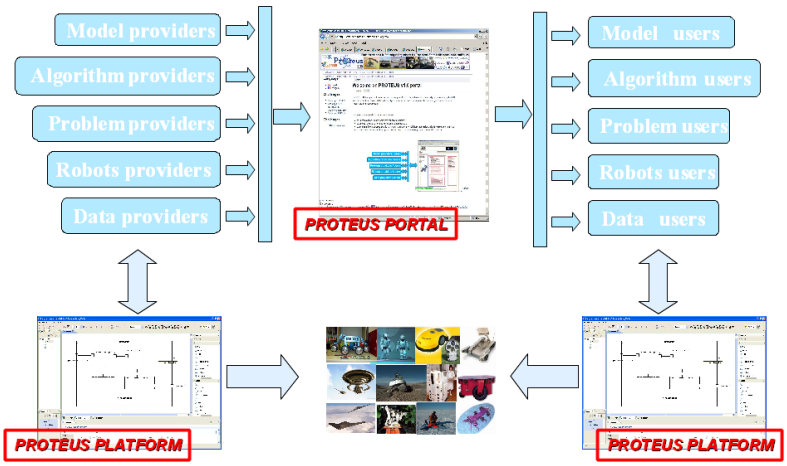

Figure 10: PROTEUS Rationale [57]

In the sense of separation of roles RobotML distinguishes between "Provider" and "User". Thereby exchange of "elements" between them takes place using the PROTEUS portal. The element can be system descriptions modelled using RobotML, algorithms (resp. libraries) or other data. Even so, this is generally aligned with the goals of FIONA, the main problems are that at the end of the PROTEUS project, RobotML remained in an early prototypical state. Beyond the few promising publications such as [58] it remains unclear how to extend the RobotML models such that a component model is able to support black-box and white-box views, the communication characteristics are distinctly defined and finally the code generators can be implemented for more than just ROS, Orocos and Morse.

3.5.3. Model-Driven Software Development

One strong point in PROTEUS is their explicit usage of MDSD from the very beginning. Similar as in SmartSoft, RobotML uses Eclipse and Papyrus to define models. However, at the time of writing, the tooling behind RobotML has not yet reached a sufficient maturity level to be able to build on top of it. In particular, many conceptual questions remain how to use the tooling beyond the scope of ROS and Orocos platforms.

3.5.4. Building Blocks: Market of Components

PROTEUS provides a web portal [56] to exchange models and implementations, which to a certain extent can be seen a robotics market.

4. Domain Specific Software Component Models

There are several domains that apply component based software engineering, e.g. in automotive and avionics that have come up with solutions meeting domain specific requirements. We use AUTOSAR as a representative example.

Key statements:

- Automotive is an industry which has high demands for software and software engineering processes.
- New developments in automotive are driven by composition of components.
- Automotive has an established ecosystem of component suppliers and value chains.
- AUTOSAR is an established standard that uses components for composition and relies on component models.

Consequence to FIONA:

- AUTOSAR shows the necessity of taking frameworks to the next level of component models and meta models in order to manage the complexity and composability of a system.
- FIONAs decision to ground on component based engineering and meta models is supported by the success story of AUTOSAR.
- FIONA must not leave freedom for interpretation (e.g. in semantics) which otherwise limits composability and interoperability as in AUTOSAR.

4.1. Automotive Open System Architecture (AUTOSAR)

The Automotive Open System Architecture (AUTOSAR) development cooperation "is a worldwide development partnership of car manufacturers, suppliers and other companies from the electronics, semiconductor and software industry." [59]. It is the primary software component model in the automotive industry. Its goal is to provide a platform for implementation and standardisation of vehicular systems by OEMs as well as the integration of functional modules from multiple suppliers.

The developments behind AUTOSAR show good tendencies such as targeting system composition using commercial off-the-shelf components (COTS) through a defined/standardized component model, QoS and more. The standard is very heavy and much code is being generated for implementations. However, tools of different vendors are not always compatible due to some freedom for interpretation in the standard (e.g. for messages), therefore implementations are not always compatible which limits composability. The standard defines only interfaces, but does not standardize message exchange, which may cause inconsistencies between suppliers. AUTOSAR misses a flexible lifecycle automaton and focuses on (static) configuration during development/deployment but does not address (dynamic) configuration and reconfiguration or adaptation at runtime.

More complex software systems in AUTOSAR nowadays are typically hidden beyond a "complex driver" and are thus not natively built via AUTOSAR.

AUTOSAR is a good example that shows the necessity of taking frameworks to the next level of component models and meta models in order to manage the complexity and composability of a system.

With AUTOSAR, the automotive industry has established a software ecosystem of component suppliers within a value chain.

5. Smartphone Domain

Within mobile computing platforms, Smartphones are the outstanding platforms in the recent years. The most widely used operating systems and integration platforms are Android, iOS and Windows Phone. They have distinct features (especially UI and touch HMI) compared to standard embedded devices and have advanced concepts for application development and distribution in markets (iOS app store, android market).

Key statements:

- Integration platforms in the Smartphone domain, such as Android, iOS have no component model in the sense of UCM.
- Reuse takes place on the level of app-parts and code/libraries rather than on system level.
- Tool support focuses on rapid GUI development rather than classical software engineering.
- It appears that the complexity behind the goals of FIONA are greater than the typical complexity in the Smartphone domain.

Consequence to FIONA:

- In order to deal with complexity, reuse on a higher level than libraries is a valid goal for FIONA.
- Separation of roles (developer vs. integrator) is missing and has to be considered for FIONA.
- FIONAs goals on a technical and systematic view are still valid and are confirmed by the Smartphone domain.

The complexity of software in the Smartphone domain is less complex than in robotics. Robots are dedicated systems (autonomous and autarkic) which act and interact in and with their environment (e.g. driving on their own, perception, object manipulation). There is a huge variety of different motion platforms (e.g. wheeled, legged, flying), sensors (e.g. camera, 3d, touching) and algorithms. Smartphones are focused on being a tool for humans, therefore having its focus on user interaction (mostly graphical and touch/gestures) with less complexity in functionality and algorithms. Most Smartphone applications can be limited to presentation of data which is retrieved through webservices, stored and processed remotely.

5.1. Software Integration Platform

Mobile development is supported by a variety of frameworks (e.g. Appcelerator, Phonegap, Rhodes and Xamarin) which mainly distinguish in their nature, being natively implemented, in mobile apps powered by HTML/JavaScript or hybrid applications. An overview can be found in [60]. These frameworks focus on helping to speed up development of graphical user interfaces across several platforms (iOS, Android, etc.), some of them also provide an abstraction layer to access the devices of the Smartphone (gyro, location service, compass). These frameworks and their sophisticated tools speed up the development process of custom apps. However, it lacks a common understanding and solution on what level exchange and reuse can be made other than UI elements. There are no activities known in which reuse is being made at a higher level than software libraries. It does not address the composition of new applications at that level.

5.1.1. Android

Android uses terms like services and component. Four different types of components [61] exist (Activities, Services, Content providers and broadcast receivers), for example for a graphical user

interface, background operation, providing data or consuming broadcasts. Some of them are accessible to other applications.

The idea of android components is that every app can make use of another app's components. For example, an image gallery app might be able to share images with other people and there are apps that provide a sharing functionality (e.g. via email clients, social network apps). The app will hand over the image to share to one of these apps which brings the app (e.g. email client with image attached) in the foreground, guides the user through the sending process within the email app and finally returns to the image gallery. Requesting actions of other components (communication between components) is done via asynchronous messaging (android intents). These intents define actions (e.g. view or send something) and additional data (e.g. URL to open in browser or URI of file to be attached). The intent might return results, such as the URI of a picture that was taken by a camera app. There are mechanisms within the Android system to match the intents to activities that other apps provide. There is no formal component model for Android.

Apple plans to introduce inter-app communication [62] in iOS 8. This is welcome within the view of FIONA, however, not yet implemented.

5.2. Separation of Roles and Separation of Concerns

Separation is a principle and general guideline in iOS and Android development. Developers often apply design patterns (e.g. MVC, Template Method, Observer, MVP, etc.) for separation between Data, UI graphic and UI logic (e.g. iOS makes heavy use of MVC [63]). [64] provides an overview on typical problems in the mobile domain and used design patterns. However, even there are different interpretations and practices for these patterns and this separation only happens at the code level on functional parts. Patterns are a design philosophy that helps a company for in-house projects but rarely contributes to reuse within a market idea.

Activities towards explicit separation of roles in an integrated approach seem to be missing.

5.3. Building Blocks: Market of Components

Most of the Smartphone Apps apps are applications for the end user, provided through a graphical UI. In contrast, components are building blocks or entities of applications which can be put together to form a new application. Apps might contain components but apps are not components itself and therefore repositories for apps (Android Market, iOS App Store) are not what is called a component market in this document.

Most comparable to the composition view is Xamarin [65]. It provides C# bindings for native app development of multiple platforms based on MONO, the open-source implementation of .NET. Its "component store" [66], offers UI and functional components such as colour chooser dialogs, slider, buttons, signature input, UI themes and cloud services in an integrated tooling. A similar concept is "androidlibraries" [67] offering for example diagram or list views.

There are a lot of services that offer composing apps from building blocks in a drag and drop way, [68] provides an overview. However, these apps only display such as contact information, product catalogues in a way similar to a website. They are limited in their functionality, but offer composition of apps on a very non-technical level even for novices without any programming.

6. Conclusion

Several approaches and frameworks for integration platforms have been presented. However, only few contain explicit component models, are based on Service-Oriented Architectures and omponent based software engineering.

Especially in robotics, the need for software development and system composition has been recognized and is a field of active research, especially within Europe. Within the Smartphone domain, efforts towards reuse and composition exist, however, these activities are so far focused on cross-platform reuse and UI elements. Actual reuse on implementation is made on the level of source code and libraries.

The Unified Component Model initiative driven by OMG has defined requirements and characteristics of the next-generation state of the art in component models. The SmartSoft Service-Oriented Software Component Model meets these characteristics to a large extend. The intention of FIONA to use and adapt the SmartSoft from the robotics domain is therefore in line with the state-of-the-art. Its aspects are even addressed and in line with the requirements that the OMG lists in the request for proposals of the "Unified Component Model", the next-generation state-of-the-art. Even more, concepts behind this approach have contributed to UCM and shape the activities related to CBSE for robotics within Europe.

The need for an integration platform for systematic reuse and composition in order to reduce development time and enhance maintainability has been recognized by recent activities. Therefore, the relevance of FIONAs goals have been confirmed and even have increased and FIONAs decision to base on component based engineering and meta models is supported by these new activities.

7. Bibliography

[1] C. Szyperski, "Component-Software: Beyond Object-Oriented Programming," Boston, ISBN 0-201-74572-0, 2002.

[2] D. Brugali and P. Scandurra, "Component-based Robotic Engineering. Part I: Reusable Building Blocks," *IEEE Robotics & Automation Magazine*, vol. 16, no. 4, pp. 84-96, 2009.

[3] D. Brugali and A. Shakhimardanov, "Component-based Robotic Engineering. Part II: Models and Systems," *IEEE Robotics & Automation Magazine*, vol. 17, no. 1, pp. 100-113, 2010.

[4] Christian Schlegel, Andreas Steck, and Alex Lotz, "Model-Driven Software Development in Robotics: Communication Patterns as Key for a Robotics Component Model," in *Introduction to Modern Robotics*.: iConcept Press, 2011.

[5] D. Sprott and L. Wilkes, "CBDI Forum," 2004. [Online]. http://msdn.microsoft.com/en-us/library/aa480021.aspx

[6] Christian Schlegel, Andreas Steck, and Alex Lotz, "Robotic Software Systems: From Code-Driven to Model-Driven Software Development," in *Systems - Applications, Control and Programming*.: InTech, 2012.

[7] Christian Schlegel et al., "Model-Driven Software Systems Engineering in Robotics: Covering the Complete Life-Cycle of a Robot," in *Workshop Roboter-Kontrollarchitekturenlink, co-located with Informatik 2013*, Koblenz, 2013.

[8] Matthias Minich, Bettina Harriehausen-Mühlbauer, and Christoph Wentzel, "Component Based Development in Systems Integration," in *Informatik 2011, Workshop Vorgehensmodelle in der Praxis*, Bonn, 2011.

[9] (2013) Eclipse Website: Eclipse Modelling Project. [Online]. http://www.eclipse.org/modeling/

[10] (2013) Eclipse Website: Papyrus UML. [Online]. http://www.eclipse.org/papyrus/

[11] (2013) Eclipse Website: Xtext. [Online]. http://www.eclipse.org/Xtext/

[12] "Unified Component Model for Distributed, Real-Time and Embedded Systems," Object Management Group, OMG Document: mars/2013-09-10, 2013.

[13] (2014) OMG Website: Unified Component Model Wiki. [Online]. http://www.omgwiki.org/ucm/doku.php?id=start

[14] (2014) RemedyIT UCM Wiki. [Online]. https://osportal.remedy.nl/projects/ucm/documents

[15] RemedyIT. (2013) Evolution from LwCCM to UCM. [Online]. http://www.omgwiki.org/ucm/lib/exe/fetch.php?media=cid_evolutionlwccm2ucm.pdf

[16] E. A. Lee. (2010) MODELS Keynote Talk: Disciplined Heterogeneous Modeling. [Online]. http://models2010.ifi.uio.no/material/Heterogeneous_Models.pdf

[17] Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan, "Overview of the CORBA Component Model," http://www.cs.wustl.edu/~schmidt/PDF/CBSE.pdf,.

[18] "CORBA Component Model Specification," 2006. [Online]. http://www.omg.org/spec/CCM/

[19] (2008) The CORBA & CORBA Component Model (CCM) Page. [Online]. http://www.ditec.um.es/~dsevilla/ccm/

[20] Douglas Schmidt, "Tutorial on the Lightweight CORBA Component Model (CCM)," in *OMG Workshop on Distributed Object Computing for Real-time and Embedded Systems*, Arlington, VA (USA), 2003.

[21] Object Management Group, "Real-time CORBA Specification," 2005. [Online]. http://www.omg.org/spec/RT/

[22] Pedro J. Clemente, Juan Herandez, and Fernando Sanchez, "Extending Component Composition Using Model Driven and Aspect-Oriented Techniques," *Journal of Software*, vol. 3, no. 1, 2008, http://dx.doi.org/10.4304/jsw.3.1.74-86.

[23] Philippe Merle, "OpenCCM: The Open CORBA Components Platform," in *3rd ObjectWeb Conference*, INRIA Rocquencourt, France, 2003.

[24] (2014) euRobotics AISBL: Topic Groups. [Online]. http://www.eu-robotics.net/ppp/objectives-of-our-topic-groups/

[25] Ayssam Elkady and Tarek Sobh, "Robotics Middleware: A Comprehensive Literature Survey and

Attribute-Based Bibliography," *Journal of Robotics*, vol. 2012, 2012, doi:10.1155/2012/959013.

[26] Christian Schlegel and Robert Wörz, "The software framework SMARTSOFT for implementing sensorimotor systems," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, Kyongju, Korea, 1999, pp. 1610-1616.

[27] Christian Schlegel et al. (2014) SmartSoft Project Sourceforge. [Online]. http://sourceforge.net/projects/smart-robotics

[28] Christian Schlegel, Alex Lotz, and Andreas Steck, "SmartSoft - The State Management of a Component," Berichte des ZAFH Servicerobotik, Ulm, Technical Report ISSN 1868-3452, 2011.

[29] (2014, Feb.) MORSE Simulator and iPad using SmartSoft Components. [Online]. http://youtu.be/rSFbq2AE_vg

[30] Alex Lotz et al., "Towards a Stepwise Variability Management Process for Complex Systems – A Robotics Perspective," in *International Journal of Information System Modeling and Design (IJISMD 2014)*, 2014, (accepted, to appear).

[31] (2014) Servicerobotik Ulm Website. [Online]. http://www.servicerobotik-ulm.de/drupal/?q=node/61

[32] (2014) RoboticsAtHsUlm Youtube Channel. [Online]. http://www.youtube.com/user/RoboticsAtHsUlm

[33] Servicerobotik Ulm. (2013, Dec.) Modular Production System with Robotino® 3, SmartSoft and SmartMDSD. [Online]. http://youtu.be/J3k-eN7BrBk

[34] (2013) BRICS Website. [Online]. http://www.best-of-robotics.org/home

[35] Herman Bruyninckx, "Step Changes by BRICS & Rosetta," in *eu Robotics Forum*, Rovereto, Italy, 2014.

[36] H. Bruyninckx et al., "The BRICS Component Model: A Model-Based Development Paradigm for Complex Robotics Software Systems," in *28th international Symposium On Applied Computing (SAC 2013)*, Coimbra, Portugal, 2013, pp. 1758-1764, http://doi.acm.org/10.1145/2480362.2480693.

[37] BRIDE - BRICS Integrated Developement Environment. [Online]. http://www.best-of-robotics.org/bride/

[38] Davide Brugali, Luca Gherardi, A. Biziak, Andrea Luzzana, and Alexey Zakharov, "A Reuse-Oriented Development Process for Component-Based Robotic Systems," in *Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, 2012, pp. 361-374, http://dx.doi.org/10.1007/978-3-642-34327-8_33.

[39] N. Hochgeschwender et al., "A model-based approach to software deployment in robotics," in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on* , Tokyo, 2013, pp. 3907-3914, http://dx.doi.org/10.1109/IROS.2013.6696915.

[40] M. Quigley et al., "ROS: An open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.

[41] (2014) ROS Website. [Online]. http://www.ros.org/

[42] TullyFoote. (2014) ROS Wiki: ROS/ Concepts. [Online]. http://wiki.ros.org/ROS/Concepts

[43] (2014, Mar.) here Website. [Online]. http://360.here.com/2014/03/18/open-source-robotics-foundation/

[44] Brian Gerkey. (2014, Feb.) ros-users Mailinglist: ROS & DDS. [Online]. http://lists.ros.org/pipermail/ros-users/2014-February/068179.html

[45] Daniel Stonier. (2013) RosJava. [Online]. http://wiki.ros.org/rosjava

[46] (2013) Google play: ROS Android Sensors Driver. [Online]. https://play.google.com/store/apps/details?id=org.ros.android.sensors_driver

[47] (2013) Google play: Turtlebot Panorama (Hydro). [Online]. https://play.google.com/store/apps/details?id=com.github.turtlebot.turtlebot_android.panorama

[48] Daniel Stonier. (2013, Sep.) ROS.org. [Online]. http://wiki.ros.org/ApplicationsPlatform/Clients/Android

[49] Open Source Robotics Foundation. (2013, Aug.) Google play. [Online]. https://play.google.com/store/apps/details?id=org.ros.android.android_app_chooser

[50] "Robotic Technology Component (RTC)," Object Management Group, http://www.omg.org/spec/RTC, 2012.

[51] Noriaki Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and Woo-Keun Yoon, "RT-middleware: distributed component middleware for RT (robot technology)," in *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2005. (IROS 2005)*, 2005, pp. 3933 - 3938, http://dx.doi.org/10.1109/IROS.2005.1545521.

[52] National Institute of Advanced Industrial Science and Technology. (2010) RT-Middleware Overview. [Online]. http://openrtm.org/openrtm/en/content/rt-middleware-overview

[53] Proteus Website. [Online]. http://www.anr-proteus.fr

[54] Jean-Loup Farges, "PROTEUS - Robotic Ontology and Modelling - 3rd version," 2012.

[55] (2014) RobotML documentation. [Online]. http://robotml.github.io/RobotMLModelingPlatform/index.html

[56] Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, and Mikal Ziane, "RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications," in *Simulation, Modeling, and Programming for Autonomous Robots*, vol. 7628, 2012, pp. 149-160, http://dx.doi.org/10.1007/978-3-642-34327-8_16.

[57] (2014) RobotML Life cycle. [Online]. http://robotml.github.io/IntroductionToRobotML/LifeCycle.html

[58] Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, and Mikal Ziane, "RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications," in *Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, vol. 7628, 2012, pp. 149-160, http://dx.doi.org/10.1007/978-3-642-34327-8_16.

[59] AUTOSAR Website. [Online]. http://www.autosar.org

[60] Diego Wyllie. (2012) App-Entwicklung: Die besten Open-Source-Frameworks für iOS, Android und Co. [Online]. http://www.muensolutions.com/de/app-entwicklung-die-besten-open-source-frameworks-fur-ios-android-und-co.html

[61] (2014) Android API Guides. [Online]. http://developer.android.com/guide/components/fundamentals.html#Components

[62] "Bericht: iOS 8 soll App-Datenaustausch erleichtern," *Mac & i*, vol. 2014, no. 11, Mar. 2014. [Online]. http://heise.de/-2146088

[63] (2012, Jan.) iOS Developer Library. [Online]. https://developer.apple.com/library/ios/documentation/general/conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html

[64] Fadilah Ezlina Shahbudin and Fang-Fang Chua, "Design Patterns for Developing High Efficiency Mobile Application," *Journal of Information Technology & Software Engineering*, vol. 3, no. 2, 2013, http://www.omicsgroup.org/journals/design-patterns-for-developing-high-efficiency-mobile-application-2165-7866-3-122.pdf.

[65] (2014) Xamarin Website. [Online]. http://www.xamarin.com

[66] (2014) Xamarin Component Store. [Online]. http://components.xamarin.com

[67] (2014) androidlibraries Website. [Online]. http://www.android-components.com

[68] Grace Smith. 10 Excellent Platforms for Building Mobile Apps. [Online]. http://mashable.com/2013/12/03/build-mobile-apps/