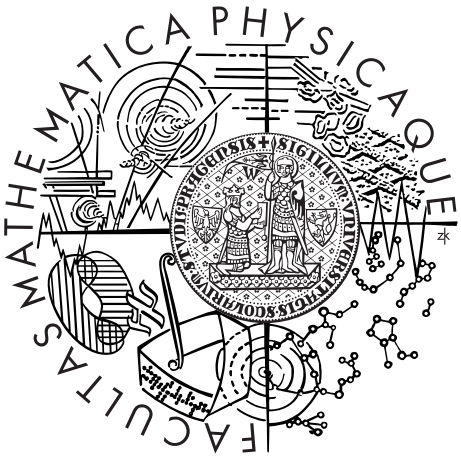
Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS



Michal Malohlava

Variability of Execution Environments for Component-based Systems

Department of Distributed and Dependable Systems

Thesis advisor:	RNDr. Tomáš Bureš, Ph.D.
Study program:	Computer Science
Specialization:	Software Systems

Prague 2012

Acknowledgment

This work is a result of being a member of Department of Distributed and Dependable Systems for the last five years. Therefore, my thanks go to all my colleagues who help me to finish the thesis. I very appreciate the help and support from my advisor Tomáš Bureš. I also would like to thank František Plášil for his patience and guidance in writing. I thank all my colleagues for creating an inspiring working environment; a particular thank goes to: Tom Poch, Ondra Šerý, Honza Kofroň, Pavel Parízek, Tomáš Kalibera, Petr Hnětynka, Vlasta Babka, Pavel Ježek, Jarda Keznikl, Tom Pop, and Vilo Šimko.

I am also grateful to Lionel Seinturier who allowed me to spend four amazing months in in the INRIA ADAM team. I would like to thank my French colleagues Aleš Plšek and Frédéric Loiret for creating not only an inspiring research environment, but also for their endless enthusiasm.

My thanks also go to the institutions that provided financial support for my research work. Through my doctoral study, my work was partially supported by the Czech Science Foundation grant 201/09/H057, by Charles University institutional funding SVV-2012-265312, by the Ministry of Education of the Czech Republic grant MSM0021620838, and by the Q-ImPrESS research project by the European Union under the ICT priority of the 7th Research Framework Programme.

All my friends deserve also a dedicated acknowledgment – thank you for being here. And last, but not least, I am in debt to my parents, sister, and overall family for their support and encouragement during my doctoral studies.

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature, and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague 7th June 2012 Author's signature

Název práce	Variabilita běhových prostředí komponentových systémů
Autor	RNDr. Michal Malohlava michal.malohlava@d3s.mff.cuni.cz (+420) 221 914 236
Katedra	Katedra distribuovaných a spolehlivých systémů Matematicko-fyzikální fakulta Univerzita Karlova v Praze
Vedoucí disertační práce	RNDr. Tomáš Bureš, Ph.D. tomas.bures@d3s.mff.cuni.cz (+420) 221 914 236
Adresa	Katedra distribuovaných a spolehlivých systémů Univerzita Karlova v Praze Malostranské náměstí 25 118 00 Praha

Abstrakt

Znovu použitelnost je jedním ze základních pilířů softwarového inženýrství. Tato vlastnost umožňuje vyvíjet systémy nejen rychleji ale také s menším úsilím. Proto tato dizertační práce zkoumá limity znovu použitelnosti v rámci komponentových systémů. Na základě analýzy současných komponentových systémů nachází jejich společné vlastnosti a rozdíly. Dosažené poznatky shrnuje do návrhu meta-komponentového systému – softwarové výrobní linky pro přípravu komponentových systémů na míru specifikovaným požadavkům.

Práce se dále soustředí na definici vlastního meta-komponentového systému a rozebírá jeho důležité aspekty, které jsou klíčové pro přípravu nového komponentového systému – (1) konfigurovatelné běhové prostředí a (2) generování kódu. Pro řešení (1) práce přináší modelem řízenou metodu přípravy konfigurovatelného běhového prostředí. V návaznosti na tuto metodu rozebírá generování kódu, definuje a vysvětluje roli interoperability doménově specifických jazyků v tomto procesu. Dále práce rozšiřuje koncept interoperability a definuje rodinu jazyků parametrizovanou obecným programovacím jazykem.

Klíčová slova

Komponenta, Komponentový systém, CBSE, Běhové prostředí

Title	Variability of Execution Environments for Component-based Systems
Author	Michal Malohlava michal.malohlava@d3s.mff.cuni.cz (+420) 221 914 236
Department	Department of Distributed and Dependable Systems Faculty of Mathematics and Physics Charles University in Prague
Advisor	RNDr. Tomáš Bureš, Ph.D. tomas.bures@d3s.mff.cuni.cz (+420) 221 914 236
Mailing address	Department of Distributed and Dependable Systems Charles University in Prague Malostranské náměstí 25 118 00 Prague, Czech Republic

Abstract

Reuse is considered as one of the most crucial software engineering concerns. It allows for delivering software systems faster with less effort. Therefore, the thesis explores limits of reuse in the context of component systems. It analyzes in depth contemporary component systems, finds their commonalities and variation points, and introduces a meta-component system – a software product line which allows for producing a tailored component system based on a set of requirements.

The thesis addresses the meta-component system definition and focuses on its crucial aspects which play the key role in component systems preparation – (1) a configurable execution environment and (2) generation of implementation artifacts. To address the first aspect, the thesis proposes a model-driven method for creating configurable execution environments. Motivated by creating execution environments, the thesis contributes to (2) by introducing a notion of domain-specific languages interoperability in the context of the code generation. Furthermore, the thesis elaborates the proposed notion resulting into a family of interoperable domain-specific languages which is parametrized by a general purpose language.

Keywords

Component, Component system, CBSE, Execution environment, Variability

Table of Contents

Chapter 1

Introduction

1.1 System Development and Reuse

During last decades software system development has become a craft which relies on many aspects covering system design, architecture decisions, implementation, documentation, testing. Those aspects are typically specified by an overall development methodology clarifying all factors of system development.

One of the crucial aspects having an immediate impact on fast system delivery is reuse of development artifacts. It represents a natural demand which can be directly expressed by the popular statement "don't repeat yourself" (often referenced shortly as a DRY concept) [HT99] and which forces developers to divide development artifacts into well-defined reusable modules encapsulating given assets (*e.g.*, implementation, documentation, or architecture artifacts) and defining their provisions and requirements. The strategy does not only help to reuse the artifacts in different contexts but also forces developers to systematically separate concerns and define relations among them. Therefore, it is considered as one of the key factors of proper software development.

1.2 Component-based Software Engineering

One of development methodologies which consider reuse as a first-class concept is called *Component-Based Software Engineering* (CBSE) [HC01, Szy02]. Component-based development has been adopted in many domains – ranging from simple configuration systems (*e.g.*, Google Guice [12], Spring [31], iPOJO [EHL07]), desktop and web applications (*e.g.*, JavaBeans [26], Java Server Faces [29]), enterprise applications (*e.g.*, EJB [27], CCM [13]) to embedded systems (*e.g.*, Koala [OLKM00], MyCCM-HI [VPK05a]).

It allows for building software from well-defined building blocks called *components* communicating via precisely specified communication ports. The component represents an entity of reuse which can participate in various contexts. Component semantics can be based on a non-formal definition based on common conventions (*e.g.*, Spring [31], OSGi [4]), or it can be semi-formally defined by a rigorous *component model* (SOFA [BHP06], Fractal [BCL+06], CCM [13]) which can be further refined

by a formalism (*e.g.*, BIP [BBS06], FracToy [TMS10]). The component model states rules for component definition including, for example, a form of their structure and composition, behavior, data-flow, and control-flow. However, in order to actually develop, deploy and execute a component system, the model itself is not enough. The model has to be supplemented with various tools and runtime libraries which give the model meaning and enables system development and execution. The following three constituents typically participate in building component-based systems:

Design and development tools They enable to "draw" component and application architectures and support their validation and implementation. The tools understand the component model and implement its rules and constraints. Such a group of tools can include a graphical editor of component architectures, compiler, validator, processor of a component architecture description language, generator of component implementation skeletons, repository for storing components, analysis and verification tools, debugger, etc.

Deployment tools are responsible for transformation of designed component-based architectures to a form ready to be executed. The tools involve support for planning of component distribution, adjusting and preparing execution infrastructure including generation of connectors for inter-component communication, transformation of design components to runtime components (*e.g.*, in the case of embedded systems when design components are merged on code level and runtime components are produced), compiling and bundling prepared artifacts.

Execution environment and tools allow for instantiating and executing deployed components. The execution environment typically provides an infrastructure which satisfies components dependencies including runtime libraries. Furthermore, it can support components execution by managing their lifecycle, distribution, providing additional services (*e.g.*, monitoring, persistence, transaction management), and assuring other components' non-functional requirements (NFRs). The infrastructure can have different forms varying from an advanced container provisioning a rich set of services, component interpreter instantiating a given component assembly (*i.e.*, a collection of components that together form a component application), to a library providing only API which is compiled with implementation of components.

The joint product of these four constituents (the *component model* and the three categories of tools) is typically referred to as a *component system (or framework)* [CL02].

There are currently a number of different kinds of component systems. They are typically tailored for a specific application domain or use. For example, in the enterprise domain, the stress lies more on a richer set of features (*e.g.*, transactions, persistence, transparent distribution, fault-tolerance), while in the domain of embedded systems, the driving concerns are determinism, predictability, low resource consumption, formal-based verification, rigorous testing, simulation, and hardware interaction.

Although, component systems tend to have significant differences in the component model, tools and execution environment, there are unifying concepts and features (*e.g.*, encapsulation, reuse, composition, methodology) that all component systems share.

Moreover, as the use of components spreads and software becomes more pervasive, it is typical to find applications that combine even several component systems (*e.g.*, one for user interface declaration, one for implementing business logic and one for driving an embedded device controlled by the application). Such mixing of component systems means however combining different models, tools and execution environments, which is typically not very well-supported. Also in fact it means duplicating the support for the common component concepts in each utilized component system (*e.g.*, system designer, deployment tools).

Therefore, the primary demand is to remove duplication among different component systems and share the common assets. The commonalities lead to an immediate idea of sharing selected common parts of a component system (*e.g.*, model, component repository, deployment tools, runtime, management tools) or at least generic elements which would be adaptable for a particular usage. These common building blocks would serve to assemble a component system according to specified requirements.

The concept of reusing various systems' assets is not new. It can be found in many previous works – software product lines [PBVDL05, CN02], software factories [GS03a], even components themselves represent reuse as its primary property [HC01, Szy02, CL02]. Nevertheless, these systems focus only on reuse in the scope of one selected application domain and do not support reuse crossing domains. Evidently, reuse among multiple application domains is considered as difficult or even impossible because of their rich variance. However, the mentioned commonalities in the structure of component systems could help to increase reuse crossing domains.

Therefore, the vision is to have a product line or rather a *"meta-component system"* – a framework allowing construction of component systems for a particular domain or a combination of domains by addressing domains' needs.

The idea would bring several advantages. The meta-component system would be beneficial in environments with specific domain-requirements (*e.g.*, resource constraints, limited programming model) where utilization of existing component systems would be difficult and require their non-trivial adaptation. Furthermore, the prepared component system would permit its user to focus on system development without distraction of unnecessary adaptation to address domain requirements not targeted by the too general component system.

Another advantage of using the meta-component system would be fast prototyping of applications. In this sense, the meta-component system would prepare a component system which would enable to produce functional prototypes of a planned component-based application.

In spite of the mentioned benefits, the meta-component system idea still brings many questions concerning system variability, configurability, adaptation, modeling, assembling, and execution of applications. Many of them are already at least partially solved by existing approaches: feature-oriented design and programming [CE00], aspect-oriented programming [KLM+97] and modeling [SVB+06], product lines [CN02, PBVDL05], software factories [GS03a]). However, to our best knowledge, the overall concept reflecting component systems is still missing. Therefore, the primary motivation for this thesis is to elaborate the idea of meta-component system in more details and focus on missing parts which cannot be covered by existing approaches.

1.3 Problem Statement

Considering the presented idea of producing tailored component systems (and potentially a combination of them), according to our best knowledge, there is no approach which would clarify an overall concept or even implement it. There are several potential reasons including complexity of integration, adaptation, and generation of modeling and implementation artifacts, as well as ensuring their synergy and interoperability. Furthermore, a missing rigorous notion of component systems tools and execution environments further increases the difficulty of the problem.

Here, it is necessary to stress the key role of the execution environment and related deployment tools. The execution environments of contemporary component systems miss a rigorous structure and semantics. They are often neglected as an "implementation part". However, according to our opinion based on SOFA 2 experience and conducted case-studies, the execution environment encodes an important part of component-model's execution semantics and often ensures non-functional properties required for correct system execution. Furthermore, a deployment process of component-based application frequently requires generation and adaptation of artifacts which become part of the execution infrastructure. But without knowing the structure of the execution environment it is hard to perform the process.

Hence, if correct deployment and execution of component-based system are to be ensured, the execution environment and related deployment tools cannot be encoded as a big knot of code without rigorous structure and semantics. Instead of that, their preparation and structure need to be explicitly described, modeled, and constructed. That involves a structural design of the execution environment, its behavior, and a process of adaptation and generation of artifacts which need to be prepared by deployment tools. These requirements directly fit the idea of the meta-component system which suggests assembling the execution environment and deployment tools on demand from reusable artifacts.

Another problem which needs to be considered is preparation of the implementation (*e.g.*, a Java class, XML configuration, persistence descriptor). Both mentioned topics – meta-component system and execution environment – directly rely on it. The preparation can have different forms – artifact configuration (*e.g.*, configuration via shell environment properties), its adaptation (*e.g.*, source code optimization), or generation artifact from scratch (*e.g.*, interceptors, configuration files). In all the cases, the process of preparation typically depends on a set of models describing various concerns – structure, data-flow, control-flow, behavior, code patterns, etc. However, having the set of various modeling assets brings a direct demand for their synchronization and interoperability (often referenced as cooperation or coordination [JVB+10, BCC+10]). And the interoperability plays a key role in the process of artifact preparation based on code generation when modeling assets influence the form of resulting code. However, in this context, the notion of interoperability among modeling assets and model of code still misses a rigorous specification and it is replaced by ad-hoc solutions based on different kinds of template languages.

1.4 Research Goals

Compared to existing approaches (*e.g.*, ACME [GMW97], xADL [DHT01], AADL [VPK05b], UML [RJB04, BRJ05]), which focus only on component modeling, the thesis puts equal stress on component-based application modeling as well as on the execution environment, and corresponding tools, which in our experience are crucial points for general use of component systems.

The thesis addresses the meta-component system approach and corresponding process of preparation of tailored component systems. The primary intention is to adopt ideas of the product line engineering [CN02] and generative software development (GSD) [Cza05b] with focus on producing families of component frameworks for different target application domains rather than just for one domain.

Instead of providing a complete realization of the meta-component system, the thesis aims at clarifying crucial aspects concerning the execution environment and its preparation. The thesis targets the following goals:

G1 The first goal is to clarify the concept of the meta-component system based on an analysis of contemporary component systems and case-studies utilizing different component systems.

G2 The second goal is to introduce a configurable execution environment and corresponding deployment process which would be suitable for the construction of the meta-component system. The environment should be well-defined and clearly express a structure of running applications and the underlying infrastructure. Furthermore, the designed environment should be capable of reflecting differences in application deployment, which varies from advanced component containers to a simple library linked with the application code.

G3 The last goal focuses on the process of execution environment preparation, which typically copes with artifacts generation. The goal is to elaborate the role of models participating in code generation and clarify the notion of their interoperability.

1.5 Overview of Contribution

The overall contribution of the thesis can be summarized in the following four points:

C1 specification of the meta-component system based on rigorous analysis of domain of component systems

C2 model-driven approach for execution environment construction

C3 models (described via DSLs) interoperability in the context of code generation

C4 parameterization of code generation by a target implementation language

The first part of the thesis clarifies the meta-component system idea based on three case-studies and experience obtained during development of SOFA 2 component system (C1). The case-studies focuses on different application domains to cover various

requirements, functional and non-functional properties which can appear. The case-study jPapaBench has been partially published in [KPMS10] and used for exhaustive testing of safety critical Java (SCJ) [HHL+09] with help of Java Pathfinder [22],[HP00]. Besides identified domain attributes, jPapaBench has shown to be a suitable benchmark for evaluation of real-time Java virtual machines (it was used in the following publications [TPNV11, Puf11, ZM11]). The second case-study clarifies properties required to be considered during designing an infrastructure responsible for remote communication among real-time Java-based applications [MPL+08]. The case study has been developed in the scope of Fractal component model and serves as a basis of further research of producing domain specific component frameworks [LPM+09]. And finally the third case-study shows a compact way of extending the SOFA 2 execution environment to satisfy a specific non-functional requirement, particularly primitive components implemented in a scripting language [KMBH11]. The case-study has served to verify suitability of aspect-oriented injection of functionality into an execution environment infrastructure.

All the facts obtained during domain analysis and performed case-studies have been summarized in a definition of the meta-component system [BHM09]. The contribution is the clarification of its notion and declaration of its stakeholders and corresponding process of compiling new component systems (C1).

The second goal of the thesis is covered by a μSOFA (C2) – a model-driven approach to assemble the component-based execution environment. It represents a part of the presented meta-component system. The novelty of the approach lies in a parallel construction of a tailored execution environment and corresponding deployment process with help of aspect-oriented modeling.

The described construction heavily relies on code generation which serves to create and complete code artifacts which cannot be prepared in advance due to their dependency on actual usage context. Hence, the last part of the thesis introduces a code generation process treating a family of domain-specific languages describing application models and patterns of code to be generated [BMH08, MPBH12]. The part stresses the demand for interoperability among these languages in the context of code generation and proposes how it can be achieved with help of a small query language [MPBH12] ensuring languages cooperation (C3). Furthermore, the thesis formalizes the interoperability concept by describing semantics of the query language. And finally, the proposed code generation process depending on the family of interoperable languages is parameterized by a target implementation language (C4) which directly fits into the meta-component approach.

1.6 Publications

The thesis content (Chapter 3, Chapter 4, Chapter 6) is directly based on the following reviewed publications:

[MPBH12] Malohlava M., Plášil F., Bureš T., Hnětynka P.: *Interoperable DSL Families for Code Generation*, In Software: Practice and Experience, John Wiley & Sons, Ltd, ISSN: 1097-024X, DOI: 10.1002/spe.2118, April 2012.

[KMBH11] Keznikl J., Malohlava M., Bureš T., Hnětynka P.: *Extensible Polyglot Programming Support in Existing Component Frameworks*, In Proceedings of 37th Euromicro

Conference on Software Engineering and Advanced Applications, Oulu, Finland, Aug 2011.

[BHM09] Bureš T., Hnětynka P., Malohlava M.: *Using a product line for creating component systems*, In Proceedings of the 2009 ACM symposium of Applied Computing (SAC'09), Honolulu, Hawaii, USA, ACM, ISBN:978-1-60558-166-8, Mar 2009.

[MPL+08] Malohlava M., Plšek A., Loiret F., Merle P., Seinturier L.: *Introducing Distribution into a RTSJ-based Component Framework*, In Proceedings of 2nd Junior Researcher Workshop on Real-Time Computing, Rennes, France, Oct 2008.

Apart from the papers directly included in the thesis, the following co-authored publications relate to component systems, their structure, and preparation including code generation:

[PPO+12] Pop T., Plášil F., Outlý M., Malohlava M., Bureš T.: *Property Networks Allowing Oracle-based Mode-change Propagation in Hierarchical Components*, Accepted for publication in Proceedings of the 15th international ACM Sigsoft symposium on Component based software engineering, 2012.

[MHB] Malohlava M., Hnětynka P., Bureš T.: *SOFA 2 Component Framework and Its Ecosystem*, Extended abstract of the tutorial – accepted for publication in Proceedings of the 9th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA), 2012.

[PLM12] Plšek A., Loiret F., Malohlava M.: *Component-Oriented Development for Real-Time Java*, A book chapter in the book *Distributed, Embedded and Real-time Java Systems*. Editors: T. Higuera-Toledano and A. Wellings, ISBN 978-1-4419-8157-8, DOI 10.1007/978-1-4419-8158-5_11, February 2012.

[PKH+11] Pop T., Keznikl J., Malohlava M., Bureš T., Hnětynka P., Hošek P.: *Introducing Support for Embedded and Real-time Devices into Existing Hierarchical Component System: Lessons Learned*, In Proceedings of 9th ACIS International Conference on Software Engineering Research, Management and Applications (SERA2011), Baltimore, Maryland, USA, pp. 3-11, ISBN 978-1-4577-1028-5, DOI 10.1109/SERA.2011.14, August 2011.

[BJM+11] Bureš T., Ježek P., Malohlava M., Poch T., Šerý O.: *Strengthening Component Architectures by Modeling Fine-grained Entities*, In Proceedings of 37th Euromicro Conference on Software Engineering and Advanced Applications, Oulu, Finland, pp. 124–128, ISBN 978-1-4577-1027-8, DOI 10.1109/SEAA.2011.27, August 2011.

[KPMS10] Kalibera T., Parizek P., Malohlava M., Schoeberl M.: *Exhaustive testing of safety critical Java*, In Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'10), Prague, Czech Republic, 164–174, ISBN 978-1-4503-0122-0, DOI 10.1145/1850771.1850794, 2010.

[HPB+10] Hošek P., Pop T., Bureš T., Hnětynka P., Malohlava M.: *Comparison of Component Frameworks for Real-time Embedded Systems*, In proceedings of CBSE 2010, Prague, Czech Republic, LNCS 6092, Springer, pp. 21-36, ISSN 0302-9743, ISBN 978-3-642-13237-7, June 2010.

[LPM+09] Loiret F., Malohlava M., Plšek A., Merle P., Seinturier L.: *Constructing Domain-Specific Component Frameworks through Architecture Refinement*, In Proceedings of the 35th Euromicro Conference, Patras, Greece, Aug 2009.

[BMH08] Bureš T., Malohlava M., Hnětynka P.: *Using DSL for Automatic Generation of Software Connectors*, In Proceedings of International Conference on Composition-Based Software Systems (ICCBSS 2008), Madrid, Spain, IEEE Computer Society Press, ISBN 0-7695-3091-5, pp. 138-147, Feb 2008.

[MB08] Malohlava M., Bureš T.: *Language for reconfiguring runtime infrastructure of component-based systems*, In Proceedings of MEMICS 2008, Znojmo, Czech Republic, ISBN 978-80-7355-082-0, November 2008.

1.7 Thesis Roadmap

The thesis is divided into the following chapters:

Chapter 2 It gives an overview of the current state of the art. It surveys domains which have an overlap with the intended meta-component system and identifies commonalities which would be beneficial for meta-component system realization as well as weak points which the meta-component system should fulfill.

Chapter 3 Since the meta-component system idea is motivated by the concept of software product lines, the chapter performs a domain analysis whose goal is to explore a domain of component systems and find common parts and variation points which the meta-component system realization has to consider. The chapter also provides three case-studies demonstrating use of different forms of execution environment.

Chapter 4 Based on the analysis, the chapter proposes an overall design of the meta-component system.

Chapter 5 The chapter focuses on an execution environment as the crucial part of the meta-component system. It describes μSOFA– the model-driven method to construct an execution environment and assemble corresponding deployment process according to application requirements.

Chapter 6 Since implementation artifacts constituting the execution environment often need to be generated, the chapter elaborates the topic in more details. It explains the role of domain-specific languages in code generation and clarify the notion of their interoperability with respect to code generation. Furthermore, it also shows how the code generation process can be parameterized by a target technology.

Chapter 7 The evaluation designs the three case-studies introduced in Chapter 3 with help of μSOFA. The chapter also stresses the notion of interoperability in this context.

Chapter 8 The last chapter summarizes the thesis, its contribution, and proposes possible future plans and directions.

1.7.1 Notes on Conventions

Selected parts of the thesis are based on the published papers (Chapter 3, Chapter 4, Chapter 6). In this case, the paper content is adapted to fit into the thesis text flow and the corresponding part of the thesis is marked by a vertical bar on the side of the text. Moreover, the part is introduced by a citation of the paper. If the part contains a major change, the changed text is considered as new and the vertical bar is ommited.

Chapter 2

State of the Art

The meta-component system idea is not unique and its characteristics, such as production of tailored systems, sharing and configuring assets, can be observed in various existing approaches. Therefore, the objective of the chapter is to explore related approaches and finds similarities which would be beneficial for the meta-component system realization. Essentially the related work can be categorized into four main topics:

- *Configurable component systems* – the idea of tailoring a component system to application requirements can be observed in existing component systems;
- *Reflective middleware* – it can be adjusted according to requirements specified by the environment and applications participating in communication;
- *Software product lines and factories* – they share the idea of producing systems which conform to specified requirements;
- *Component systems and their execution environments* – respecting the goal **G2** it is necessary to explore properties of contemporary component systems and their execution environments.

The following sections are dedicated to each of these topics and elaborate them in more details.

2.1 Configurable Components

The idea of producing dedicated component systems or at least their parts according to domain requirements can be found in several works. They frequently focus on preparation of one particular aspect of a component system while neglecting the rest – for example, only a domain-specific component model is prepared [LT09, Taw11], or extending an existing component system to support domain-specific concerns [PLMS08, PPK+11]. To our best knowledge, there is no solution which would prepare an entire component system. This section brings an overview of selected approaches considering different ways of components configuration.

The Soleil approach [PLMS08], which is built on the top of the Fractal component model [BCL+06], focuses on development of real-time systems with help of *Real-time*

Specification For Java (RTSJ) [BGB+]. The Soleil's driving aspect is a limited programming model of RTSJ which extends the Java language by restrictions considering different kinds of memories, threads, and scheduling primitives. These concepts require to be modeled at the architecture level to mitigate code generation and to introduce early error detection. Therefore, the Soleil approach brings a notion of *domain components* abstracting the RTSJ programming concepts – particularly the notion of memory area representing heap, immortal and scoped memories and threading domain symbolizing various kinds of RTSJ threads.

The domain component can be considered as an orthogonal concept to regular functional components – a functional component can be encapsulated in different kinds of domain components prescribing non-functional properties. Each domain component has associated additional attributes (*e.g.*, thread timing, size of a memory area) and defined semantics via Alloy [Plš09]. During application deployment, the domain components are translated into RTSJ code which manages encapsulated functional components (*i.e.*, their content).

For transformation of the architecture into implementation, the Hulotte framework ([LPM+09],[LRS+11]) is utilized. The Hullote represents a general framework which supports modeling application non-functional requirements and their translation into runtime infrastructure. It introduces a process which allows for annotating an architecture description with well-defined annotations. Annotations have associated semantics in the form of container aspects which inject components into the architecture description resulting into fine-grained architecture of runtime container. The resulting architecture is further transformed into a form of code, compiled, and executed. In the context of Soleil, domain-components are translated into a form of Hullote annotations and then overall RTSJ application is generated.

To summarize, the overall combination of Soleil and Hulotte frameworks focuses on preparation system realization, but it neglects the rest of artifacts participating in system development (*e.g.*, component designers, deployment tools). Although, the approach of architecture refinement is highly configurable, it assumes the predefined form of execution environment (a container) with its predefined deployment process which cannot be changed. Furthermore, the container implementation technology is fixed to the Java language only and there is no obvious way of introducing another implementation or configuration technology (for example, C language, Spring/Guice).

The MICOBS project [PPK+11] shares the idea of preparation specific platform specific models and corresponding executable system. It is a multi-platform multi-model component-based development framework. Its main goal is to introduce a framework to realize systems built from components (even heterogeneous) which would be possible to deploy on various target platforms. Hence, it brings an explicit concept of a target platform which requires to be modeled to reflect platform-dependent non-functional properties. In the MICOBS view, the platform is defined by a particular operating system, its API, CPU (and its architecture), and board.

On the architecture level, MICOBS defines a general component model called MCAD describing the elementary system artifacts (component, component type, interface, port). To achieve independence on a particular component model, the MCAD model introduces a concept of a *domain* which can be of two kinds. The *implementation-oriented domain* has a reference domain implementation (*e.g.*, a contemporary component system) and therefore can be directly utilized for system development and deployment. On the other hand, the *application-oriented domain* represents an application

specific component model without any direct implementation. Hence, it needs to be transformed into implementation-oriented domain with help of model-to-model and model-to-text transformations.

The MICOBS introduces also a concept of *complex implementation-oriented domains* which composes multiple "primitive" implementation-oriented domains. From the architecture perspective, it represents a system composed of multiple heterogeneous component-models. In this case, additional transformation and code generation is required to overcome the differences among the utilized models.

The MICOBS also copes with non-functional properties in a general way. It defines a separated model decorating the MCAD model elements with platform independent as well as dependent properties. The system-analysis model then defines a way of transforming the properties into a form of system report.

The MICOBS approach does not specify a new runtime environment. It always relies on an existing component-framework providing executable environment for component-based systems. However, the MICOBS allows modeling with help of ad-hoc models which needs to be transformed into a selected component model which would allow deployment and execution.

Taweel's Phd thesis *An Approach to the Definition of Domain-specific Software Component Models* [Taw11] considers component models as an elementary approach to model systems. It states that the models are used in various domains from enterprise to embedded systems, which require not only adapting the model itself but also providing domain-specific modeling artifacts to mitigate development effort.

However, in Taweel's perspective the contemporary component models are designed to be general-purpose to cover a large set of systems. And, even if a component model is considered as domain-specific (*e.g.*, the case of embedded systems) it does not fully integrate domain knowledge of a particular domain. The thesis focuses on deriving domain-specific component models based on a domain analysis producing functional as well as a feature model of a selected domain. Based on a survey of selected component models (PECOS, Koala, SaveCCM) from the perspective of domain-modeling (considering extra-functional properties, support for product lines), the thesis introduces a generic component model which serves as a core of derived domain-specific component models. The core of a proposed generic component model consists of a set of predefined exogenous connectors with precisely defined semantics which serve as a base for deriving new types of connectors as well as for defining control and data flows. The generic model further clarifies the model notion of data encapsulation and its relation to component composition as well as support of extra-functional properties which are managed by connectors.

Thus the Taweel's approach includes a description of methodology for deriving domain-specific component models based on domain knowledge including functional and features models. The derivation methodology specifies rules of: (i) selecting primitive components as elements in the domain, (ii) composing and specifying new connectors reflecting data and control flows in the domain, and (iii) mapping domain data stores to encapsulated data.

To summarize, the proposed approach focuses only on preparation of a domain-specific component model, which represents in our interpretation only one part of a component system. It means that a specification and adaptation of tools and execution environment is missing. Nevertheless, the proposal of generic component model considers not only structural composition, but it also respects the data as well as control

flow among components.

2.2 Reflective Middleware

The similar intention can be, for example, recognized in the area of reflective middleware [LQS04, BCA+01]. In this context, middleware can be straightforwardly specialized to a range of domains including multimedia, embedded systems, and mobile computing. The main idea behind them is to provide a highly configurable middleware layer which would reflect (statically or dynamically) requirements of an application built on the top of this layer. To build such a layer, projects typically utilize some ideas of component systems that ensure the demanded adaptability and dynamic reconfiguration.

A well-known member of the reflective middleware family is OpenORB [BCA+01]. It assembles a middleware layer with the help of the OpenCOM component system [CBCP01] and it focuses on adaptation and runtime reflection. OpenORB provides multiple meta-layers for describing interfaces, architectures and resources. These layers are fully configurable statically as well as dynamically (at runtime).

Another example of the reflective middleware is Dream [LQS04], which is built using the Fractal component system [BCL+06]. Dream allows for building middleware that offers messaging, scheduling, task creation, etc. For adaptation and runtime reconfiguration it uses the principle of controllers provided by the Fractal component model.

Both the mentioned examples partially realize the idea of the meta-component system, however they focus on a single domain only and allow configuration just in the scope of their domain. Also, they aim at adaptation and dynamism of resulting systems and therefore they do not consider requirements of embedded systems for static configuration or efficient use of system resources.

Nevertheless, the important property of reflective middleware is that its realization is encapsulated and independent – *i.e.*, it contains all services required for its execution as well as execution of applications built on the top of them and does not require any kind of external support (naturally except operating system). These properties we consider crucial to achieve a well-defined execution environment.

2.3 Software Product Lines and Factories

Obvious recurring patterns, demands to make software development cheaper, faster, and easier are main reasons why reuse stands as one of primary intentions of software development since its beginning [Par76, GS04] . Increasing software complexity has brought several paradigms for rigorous systematic reuse. The most known paradigm is represented by *software product lines* [Par76, CN02, PBVDL05] which establish a scope for systematic assets reuse by defining a common vocabulary, reusable set of assets, and strategies to produce members of a desired product family. The intention of the product line is to identify common problem features for a desired product family in advance and to prepare reusable implementation assets which would mitigate production of family members. The product line itself is defined by its analysis, design, and implementation [CN02, GS04]. The product line analysis is based on a survey of the problem space and identification of problems which the desired product

line is expected to solve. Based on the identified problems, their significant features (*e.g.*, stakeholders, commonalities, variation points) are selected and composed into a domain model. The domain model describes desired problems and typically is expressed with help of a feature diagram [CE00, KCH+90].

The product line design specifies how the line will develop a product. Hence, it is composed of its architecture describing high-level design of products emphasizing common features and variation points and product line development process which prescribes how particular product is developed. The design has to also provide a mapping among domain model features and designed product architecture variation points.

And finally, the product line implementation provides common implementation assets (*e.g.*, components, configuration files, documentation) required by the product line architecture and development process.

During product development, the domain model allows for selecting features of the desired product. The selection configures the product line architecture and process. And the product line implementation is utilized to derive product architecture and corresponding production plan prescribing product assembly. During assembly the selected product line implementation assets are integrated together to provide the desired product.

It is necessary to stress that the concept of product lines does not enforce to utilize a given implementation and modeling technology. It only specifies a vocabulary and set of strategies which mitigate reuse. The most known utilization of the product line concept is called a *software factory*.

A software factory is defined as a model-driven product line [GS03b, GS04] – *i.e.*, a product line which is driven by models defined with help of domain-specific languages. The definition of software factory says [GS04]:

> *"A software factory is a software product line that configures extensible tools, processes, and content using a software factory template based on a software factory schema to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring framework-based components."*

The factory schema prescribes a set of artifacts which are necessary to be developed to produce a software product. The schema actually describes a family of products but does not provide implementation. To build members of the family, the schema needs to be implemented resulting into the software template. It contains assets (DSLs, tools, frameworks) which are required to produce a family member. The template is then loaded into Interactive Development Environment (IDE) which becomes a particular software factory to produce members of the product family defined by the schema. In contrary to the general idea of product lines, the software factory stresses the IDE as a central part of the product line which serves not only to produce new products but also to configure and produce a software template.

In the presented context, the meta-component system can be considered as a dedicated software product line (respectively a factory, but we do not stress importance of the IDE) which copes with a family of component systems.

2.4 Component Systems and Their Execution Platforms

Contemporary component systems can be classified according various criteria. Most of systems comparisons focus only on capabilities of component model with respect to composition [LW07], component syntax and semantics [LW07, LW06], modeling extra-functional properties [CCSV07, CSVC11], capability of modeling individual design stages of component-based system [LW06, LW07, CSVC11, CCSV07], and component reusability and persistence [Sam97].

In the majority of publications, an execution environment (called also a component platform or container) is neglected due to the general opinion that the component model can be run without any effort. But, contrary is the case. The common view of running component-based system [CL02] is depicted in Figure 2.1. The overall *system execution infrastructure* involves a *component platform* enabling component deployment and execution (we call it also an *execution environment*). The component platform is running on the top of an operating system (or a virtual machine). It can utilize a dedicated middleware for component communication. With respect to the component platform, its boundaries are not fixed. The component platform can be a dedicated application communicating with operating system via its API (*e.g.*, OSGi [4], Fractal [BCL$^+$06], SOFA 2 [BHP06], EJB [27]). On the other hand, the component platform can constitute a part of an operating system as in the case of COM [21]. The same goes for middleware: it can be incorporated in the component platform or have a form of external provider which the component platform utilizes via a well-defined interface. This freedom in the component platform definition leads to various variants.

Furthermore, the concept of the running system is tightly connected to deployment tools which prepare components and the component platform for execution. The deployment tools have the major influence on the form in which components are installed into the component platform. They have to know platform's provided services to satisfy components dependencies. The tools can be also responsible for preparation an ad-hoc infrastructure ensuring components distribution [Bur06, BP04]. In this case, the deployment tools have to consider not only provided middleware services but also component platform particularities such as a component instance registry [BHP$^+$07]. Although the relation between the component platform and deployment tools is obviously close, it is often not considered.

Regarding the execution environment, it is necessary to clarify its form and how it is prepared. In the component-based development process [Szy02, CCL06], the deployment stage is identified as a point in an application lifecycle, when it is being prepared for execution. The goal of the deployment process is to realize inter-connections among components and ensure their requirements. The deployment process takes components implementations and glues them together with respect to an application architecture (fulfilling required services). The glue can be in a form of generated code, library, or it can be provided by a container.

According to our best knowledge, a full-fledged survey of execution environments has not been written yet. Nevertheless, limited overviews considering only selected features exist. For example, the survey [LU07] elaborates execution environments of desktop, and web systems regarding transient state, concurrency, and resource management. However, the survey neglects component systems containing an execution environment in a form of a container or embedded systems execution environments. Furthermore, the survey does not even consider other non-functional prop-

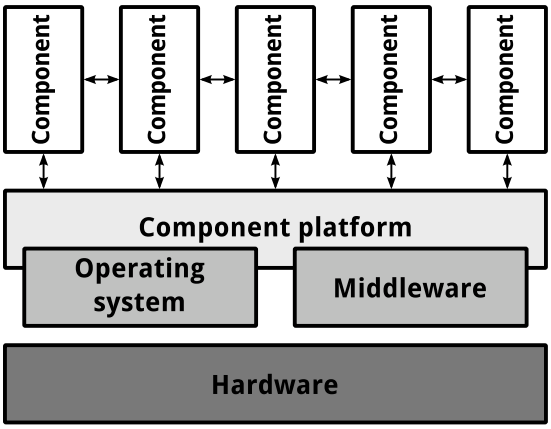


Figure 2.1: Conceptual view of running component-based system

erties which are often demanded by component-based systems (*e.g.*, dependability, distribution). Likewise, deployment tools are omitted.

Another survey [CSVC11] distinguishes two different kinds of component deployment: (i) *compile-time deployment* and (ii) *runtime deployment*. During compile-time deployment the components are compiled and linked together. This kind of deployment is typical for embedded systems which are frequently deployed into an environment with limited resources (*e.g.*, Koala [OLKM00], Pecos [GCW+02], Pin [HIPW05], MyCCM-HI [VPK05a], ProCom [BCC+08]), for GUI builders (*e.g.*, JavaBeans [26]), or simple configuration frameworks (*e.g.*, Google Guice [12], Spring [31]).

On the other hand, the runtime deployment allows components to be added, and often to be replaced during system execution. This is a typical case for enterprise systems which need continuous execution and do not have strict resource constraints (*e.g.*, CCM [13], EJB [27], OSGi [4], SOFA [BHP06], Fractal [BCL+06]).

In both cases of component deployment, there is only a minimal intention to rigorously clarify a role and structure of the execution environment and its constructions. Nevertheless, for example, the Fractal component model [BCL+06] brings a concept of a component *control membrane*. Each Fractal component is composed of its content which is encapsulated by a control membrane. The membrane is composed of several controllers which can be controlled via dedicated control interfaces. Furthermore, a controller can expose or require internal interfaces to communicate with other controllers.

The controllers serve for managing component content (*e.g.*, injection of binding, attributes' values, lifecycle) and non-functional requirements such as component internals introspection, distribution, or reconfiguration. The Fractal component model specification does not intentionally clarify a notion of controllers and their semantics is only weakly specified – it describes them as "objects" exposing interfaces. Hence, the Fractal model implementation called *Julia* utilizes regular Java objects to assemble the component membrane. On the other hand, the implementation of the Fractal specification called AOKell [SPDC06] introduces a *componentized membrane*. In this case, the membrane is represented as a composite Fractal component providing control interfaces and containing sub-components standing for particular controllers. This concept enables to define the component membrane in a modular way and reuse it among systems realizations. Nevertheless, the Fractal membrane definition omits interface

interceptors which play a key role in specification of controllers observing calls on interfaces (typical example is the lifecycle controller). As the result, this separation limits reuse of componentized membrane. Furthermore, AOKell does not address a kind of deployment – it implicitly relies on the runtime deployment. This deficiency in deployment configuration is solved by *Juliac* framework (Julia Compiler) enabling compile-time deployment by specifying a process generating and compiling the entire system runtime infrastructure into a runnable binary.

The SOFA 2 component system [BHP06] adopts a similar approach for describing an infrastructure encapsulating a component. The SOFA component model introduces a micro-component model which is simple, flat, and without distribution, The model serves to describe non-functional aspects of components. The model introduces a *micro-component* providing and requiring interfaces. The micro-components are composed into so called *aspects* representing a non-functional requirement (*e.g.*, management of lifecycle). Every aspect has a precisely defined location in the component infrastructure which it can affect. During deployment time, the aspects are woven around components contents and as a result they compose the entire application infrastructure. SOFA 2 supports only the runtime deployment that means the described process of weaving is performed in the scope of the SOFA runtime container.

2.5 Lessons Learned

The previous sections have shown four principal research areas which in some manner overlap with the meta-component system idea. They have confirmed that there is no solution which would implement the overall idea of the meta-component system as has been introduced in Section 1.3. However, the state of the art discussion has shown that there are approaches which could be useful for meta-component system realization.

The part devoted to configurable components has shown that only partial solutions exist bringing limited variability. They are either only focused on configurability of a dedicated part of a component system – typically on the component model. Or the solutions are tightly connected with a selected component system and do not go beyond its boundaries. The analysis has also exhibited that components can be configured with help of various forms of modeling aspects bringing mainly benefits of concerns separation and reuse. On the other hand, the aspect-based approaches typically suffer from loosely described dependencies among aspects (*i.e.*, different weaving orders of dependent aspects can lead to different results) and from inaccurate structural definitions of aspects.

Reflective middleware can be considered as a working example of a configurable system typically composed of components, which is adapted according to domain requirements. In the scope of the meta-component system, the reflective middleware could play a role of a highly configurable communication layer. Except that, the part has stressed the reflective middleware is also encapsulated and independent – it models and implements all the necessary services and functionality to support application communication and its own execution and management.

The closest area to the meta-component system is represented by product lines and software factories. They bring a concept producing members of a described product family based on desired features. Their important property is that they do not depend

on any application domain or application kind and only describes rules, strategies, and concepts to improve reusability while producing software products. Since most of component systems cope with various kinds of models (*e.g.*, architecture, deployment, behavior), the model-driven product line introduced by the software factories concept looks promising with respect to the meta-component system. Therefore, in the design of meta-component system, we will follow best-practices recommended by the product lines concept.

Chapter 3

Domain Analysis

To clarify the motivation and understand the variability of component systems, the thesis presents an analysis of contemporary component systems (Section 3.1) and three case-studies (Section 3.2) which were implemented in the scope of different application domains.

The motivation for the analysis also comes from the concept of product lines (summarized in Section 2.3) which prescribes to perform a domain analysis to identify common features as well as variation points of a product family. Therefore, the presented analysis focuses on existing component systems supporting the whole development cycle [CCL06]. It identifies typical application domains, where these component systems are utilized. For each domain, the analysis discusses a typical usage scenario of components and identifies requirements and specifics of the domain with respect to component design, deployment, and execution. The analysis designates a scope which will be covered by the proposed meta-component system as well as it specifies a common vocabulary (in words of generative system development [Cza05b], the analysis constitutes a *problem space*).

With respect to the thesis goals, the case-studies continue the analysis by demonstrating three different forms of the execution environment and its construction. Each case-study is supplemented by lessons learned which stress the relevant concepts important for the meta-component system and execution environment.

3.1 Component Model Domains

Based on [BHM09]

To better understand the commonalities and differences of the existing component systems, we have analyzed a diverse set of existing component systems (EJB [27],Koala [OLKM00], Fractal [BCL+06], OSGi [4], Gravity [CH04], Pin [HIPW05], PECOS [NAD+02], ProCom [BCC+08], CCM [13], JavaBeans [26], JavaServer Face [29], Robocop [Maa05], ProActive [BCM03]). The analysis focuses only on component systems which support a complete application lifecycle, *i.e.*, from design till execution. Also, we have taken into account our experience, which comes from the past decade of developing the SOFA 2 component system [BHP06].

The general analysis of the component systems has led us to identify four main application domains (in other words, what is modeled using the components). These domains are:

D1 enterprise applications

D2 user interfaces

D3 configuration frameworks

D4 embedded systems.

We elaborate on each domain separately in the rest of the section. For each domain we discuss a typical usage scenario of components and identify requirements and specifics of the domain. We relate these specifics to the component systems typically used in the domain and study what features the component systems in the domain typically posses.

3.1.1 Enterprise Application Domain

This domain represents a large number of applications, typically used in a commercial environment. An enterprise application often operates over a large amount of data stored in databases – it collects data from users, saves them into a database, does computation over stored data and presents the results. The application is typically divided into multiple layers where one layer is used for presenting information to the users (*e.g.*, a thin client), next layer takes care of storing the data (*e.g.*, into a database) and another encapsulates the business logic of the application.

A component system is often used in the business logic layer (*e.g.*, EJB [27], CCM [13]), where the logic is divided into (often flat) components. Simple component systems can be also identified in a view layer (*e.g.*, Java Server Faces [29]) where they are used to compose the user interface. The view layer however falls in the user interface domain as discussed in Section 3.1.2.

From the component system point of view, a component in the enterprise application is an encapsulated entity which communicates with other components through local interfaces or through remote interfaces, which allow distribution. The communication is typically implemented by a procedure call but messaging is also possible (*e.g.*, via message sinks and sources in CCM, or via message-driven beans in EJB). Middleware (such as RMI or CORBA) is often employed.

A component is typically represented by a set of classes that are deployed (*i.e.*, loaded and instantiated) in a *container* which contains an execution environment for components. In this sense a component is a well-identifiable unit even during runtime. It is further possible to deploy and undeploy a component basically independently.

The enterprise orientation of this application domain requires support of various services typically provided by the execution environment of the component system. These services typically include database access, persistence, transactions, naming, trading, web-service support, fault-tolerance (replication), etc. Each service has a standardized API defined by the component system (*e.g.*, J2EE in EJB) and a reference to it is looked up by the component implementation using naming service. The binding of the services to particular entries in the naming registry can be typically configured in the deployment descriptor of a component application.

In summary, the stress in the enterprise domain lies on having a rich set of services (database, transaction, persistence, etc.) that ease the development of enterprise applications. Components tend to be rather independent units existing at runtime. The

composition of components (especially in the industrial component systems such as EJB or CCM) is typically quite simple, presumably owing to the fact that the component systems are usually flat. A lot of complexity in enterprise component systems is concentrated in the execution environment, which has to provide all the services and has to be able to manipulate components at runtime. To manage the complexity of the deployment and management of deployed components, graphical tools are often provided for this task.

3.1.2 User Interface Application Domain

This application domain represents component systems which are used to build graphical-oriented user interfaces from predefined UI components (often called *widgets*). UI components are not limited to desktop applications (*e.g.*, JavaBeans component system [26]), but they can be used to build web interfaces too (*e.g.*, JSF [29]).

The UI components typically provide a standardized API that allows generic handling of them (methods as *show*, *hide*, *paint*, etc.). The relations among components are mainly of parent–child (corresponding to the graphical nesting of components) and publisher–subscriber (making possible to react and reflect on changes done in another component).

In this sense the component models are hierarchical and messaging (publish-subscribe) is used as the primary communication style. Interesting feature is that the messaging usually disregards parent component boundaries (*i.e.*, it connects components that do not reside in the same parent component).

The composition of UI components is typically performed on running components, so as the result of the composition may be immediately visible (*e.g.*, using the Bean Builder [1]). That requires components to be runtime entities, which are composable and replaceable at runtime.

Important features of UI components are introspection and persistence. The former allows querying the component to obtain a list of its attributes and messaging interfaces, which enables the UI design tools to compose and configure components in a generic way without having previous knowledge of them. The latter feature allows serializing a composition of configured components to a file and using the file later to instantiate the user interface at runtime.

In summary, UI component systems are relatively light-weight and do not require any elaborate services (as opposed to enterprise systems). The stress lies on easy and generic composition of running components. UI component systems are typically local (*i.e.*, not distributed) with no or very simple deployment.

3.1.3 Configuration Frameworks

Configuration frameworks are component systems typically used for configuring, customizing and extending an application. They span a wide range from simple plug-in frameworks or flat component systems (such as OSGi [4]) up to elaborate hierarchical component systems (such as Fractal [BCL+06] or OpenCOM [CBCP01]).

[1] JavaBean Builder – `http://java.sun.com/developer/onlineTraining/Beans/Beans1/builder-tools.html`

The common denominator of these systems is that components are well-defined runtime entities, which means that they can be instantiated and composed at runtime; basic introspection mechanism is also often present.

Typically, configuration frameworks do not rely on a rich execution environment. Rather, the services usually provided by the execution environment are modeled as application components – it often results in a need of accessing the same service component from several other components. There are different ways of supporting this – in OSGi a component may expose part of its functionality as a service and other components may resolve a reference to the service dynamically at runtime, while Fractal has the concept of the *shared component*, which allows using the same instance of a component at several places in a nested architecture.

Component systems for the configuration are typically local, since for simple configurations the distribution is usually not required. If the distribution becomes an issue, it is realized by components wrapping middleware and acting as *connectors*. However, in configuration frameworks aiming at grid computing [BCM03], the distribution is the core concept together with special multicast and gathercast interfaces.

As the configuration frameworks address customization and extensibility of an application, they also sometimes contain means allowing extensions of the component system itself. For example in Fractal this is supported by having the possibility of customizing *component controllers*, which act as management interfaces of a component. This way, it is easily possible to switch on/off several checks (*e.g.*, blocking calls to a component that has not been started yet), alter existing functionality and implement new one (*e.g.*, verifying that calls to a component follow a particular behavior protocol).

In summary, the component systems for configurations tend to have a simple and quite flexible execution environment. The environment does not offer rich services itself, however these services may be introduced in the form of components as there is typically a way of viewing a component instance as a shared service. The stress lies on having a strong concept of a component that can be manipulated (instantiated, composed, bound, replaced, etc.) at runtime. The example of Fractal also shows that it is advantageous if component systems in this domain are hierarchical and extensible.

3.1.4 Embedded Systems Domain

This domain covers software for embedded systems – such as those found in cars, mobile phones, home appliances, industry automation, etc. This area is rather wide and the particular requirements of the embedded systems may quite differ, however the common requirements typically include:

Efficient resource usage: The devices running embedded systems are typically small with restrictions on power usage and overall cost (save for automation where this is typically not a problem). This means that the embedded system must perform correctly even with a slow CPU and little memory. As the result, the coupling between hardware and software is quite high.

High demands on dependability: Embedded systems are often used for controlling safety critical tasks (*e.g.*, brakes in a car), where the cost of failure is very high (it can even cost human lives).

Real-time: The correct function of embedded systems often depends not just on the correct result of the computation but also on delivering the result in time. That classifies many of the embedded systems as soft real-time or even hard real-time.

Component systems in this domain (*e.g.*, Koala [OLKM00], Pecos [GCW+02], Pin [HIPW05], ProCom [BCC+08], Robocop [Maa05], AUTOSAR [7]) reflect the hardware limitations by assuming a thin execution environment, which provides only a basic abstraction layer between hardware, a real-time operating system (RTOS), and components. The execution environment typically offers only limited services such as a persistent storage [Maa05].

In many cases components are transformed during deployment to RTOS concepts (tasks, processes) and they are statically linked with the execution environment. Because of the absence of runtime components, components are mostly composed and configured statically at deployment, which often includes creating a real-time schedule for component execution based on the data/control-flow in the component architecture [BCC+08]. Configuration at runtime is missing or is very restricted (*e.g.*, only to setting attributes via a previously defined interface) [OLKM00]. The static configuration, however, allows for dramatic optimizations such as discarding unused components in an application [OLKM00].

The requirements on dependability put a lot of emphasis on analysis (such as of resource consumption, worst-case execution time, reliability). This concern is often reflected in the component systems by having a simple semantics for components, which are sometimes seen as being purely reactive (*i.e.*, having no own thread of activity) [NAD+02, BCC+08]. The communication is also kept simple by supporting the procedure call or the asynchronous message delivery. In some cases exogenous connectors are employed to explicitly capture the data and control-flow [BCC+08]. To allow for analysis, component models often support annotations for expressing resource usage, real-time requirements, reliability attributes, etc.

In summary, component systems for embedded devices have often no explicit execution environment running in a target device, but the environment is represented by a bundle of libraries statically linked to components during deployment. Important is the role of tools, which comprise a transformation of components to RTOS concepts and linking with an execution environment, simulator and so on. The stress further lies on analyzability and support for real-time.

3.1.5 Lessons Learned

Based on the analysis, it is possible to identify and analyze common characteristics of component systems used in the listed application domains. The analyzed component system significantly differ in three aspects:

(i) component model

(ii) supported non-functional requirements (including various runtime services)

(iii) form of execution environment and corresponding deployment process

It is necessary to mention that tools as a remaining component system constituent also differ among individual component systems. However, there are either directly

dependent on a component model and specified non-functional requirements (*e.g.*, graphical designers, modelers, architecture analyzers, memory footprint analyzers) or independent in the sense that they can be utilized without major changes by different component systems (*e.g.*, behavior checkers). Therefore, the tools are not considered here as a significant aspect which would distinguish component systems.

Component model variability. Component models mainly differ in a set of features supporting component modeling and development. The set is influenced by an application domain which can enforce various features such as:

- vertical composition – hierarchical versus flat components
- horizontal composition – explicit versus implicit bindings
- operation-based v. port-based interfaces
- complete v. incomplete interface definition
- meta-class levels for components – (i) component as a singleton, (ii) component and its instance, (iii) component type, component, and instance
- additional component kinds (*e.g.*, active, passive components)
- partial versus fully-fledged development cycle
- endogenous v. exogenous connectors
- behavior modeling support
- data modeling support
- and more [CSVC11, LU07, LW06, CL02]

For example, enterprise applications require a rich set of modeling features including hierarchical components, support for modeling non-functional services, and full-fledged description of component assembling and deployment. On the other hand, configuration frameworks and user interfaces are often supported by a simple (either flat or hierarchical) component model which is mainly concerned with a component assembly. And finally, the embedded component systems are typically based on a rigorous component model distinguishing between active and passive components and stressing an importance of extra-functional properties (*e.g.*, timing properties, memory allocation) which can be further used for system validation or verification.

To support the meta-component system idea, this wide component model variability needs to be considered. The meta-component system should allow for preparing a component model with a defined set of modeling features. Fortunately, there are many existing approaches which permit to handle modeling heterogeneity and produce a model including desired features. Thus, for example, aspect-based modeling [VG07, SSK+06, GV07, KTG+06, PRJ+03] injecting desired features into a common model can serve as a possible solution to cope with component model heterogeneity. Also a template based method preparing the desired component model according to specified features can serve as a solution [Cza05a]. Another possible way is to employ the UML profiles mechanism [FPR00, RJB04] or EMF profiles [LWWC12] to create a specialized component model.

Supported non-functional requirements. The second main difference among component systems lies in supported non-functional requirements (NFRs). Each component systems copes with its specific set of requirements which are primarily influenced by a kind of developed applications. For example, enterprise systems support a rich set of additional services including persistence, transaction management, transparent distribution, load-balancing, etc. On the other hand, configuration frameworks ensure only basic execution services (*e.g.*, lifecycle management, introspection) and all additional NFRs need to be managed by applications themselves.

This unbounded variability of possible NFRs which need to be reflected puts stress upon the component system participants and also directly impacts the meta-component system. Nevertheless, the survey [CSVC11] brings a classification of non-functional requirements (in the publication referenced as extra-functional properties) according their management, specification, and composability. Regarding component system realization, the most important classification is according NFRs management which directly impact the component system and particularly its execution environment. For meta-component system realization it is necessary to consider NFRs managed endogenously (only system wide) and exogenously (per collaboration as well as system wide). Both of these NFRs groups directly influence component system's execution environment. The properties managed endogenously per collaboration does not need to be further considered, since the component implementation is responsible for them.

Form of execution environment. The execution environment form is the last identified major difference. The analysis of four applications domains brings three different forms of execution environments:

EE1 *Ad-hoc execution environment* constructed in parallel with an application. This kind of the environment is typically utilized by simple ad-hoc configuration frameworks and control systems with a pre-generated off-line schedule. In this case, the execution environment has to be encoded manually (or generated) as a glue code connecting the components. The result of the deployment process is a binary image of application which can be directly launched.

EE2 *Execution environment as a library* is used by component systems for embedded applications domain, configuration frameworks, and user interfaces. In this case, glue code connecting components is supplemented in a form of a static library providing all necessary services (*i.e.*, programming API). During deployment, the components (and also generated glue code) are compiled with the library. As in the previous case, the result of deployment is a binary image of application.

EE3 *A container* providing a rich set of execution services is characteristic for the domain of enterprise systems. Contrary to the previous execution environments, the deployment process bundles the assembled applications into a form of package and installs it in the running container. The container is responsible for satisfying all application's requirements including bindings among components and non-functional properties.

These three forms of execution environment need to be supported by the meta-component system. That also includes a corresponding deployment process which transforms components into their runnable form.

To summarize the identified differences in component systems, there are three fundamental variation points. Since the differences in component models can be addressed by existing approaches and support of NFRs depends mainly on the capabilities of the components' execution environment, the remaining variation point – form of execution environment – and the execution environment itself require more detailed elaboration. Therefore, additional case-studies are conducted to understand variability of execution environments.

3.2 Case-studies

The presented case-studies directly continue in the analysis and focus on the identified forms of execution environment. The case-studies employ execution environments in the same order as they were listed in Section 3.1.5 – *i.e.*, the jPapaBench case-study utilizes the simplest EE1 form of execution environment, the RTSJ connectors case-study employs a library-based execution environment (EE2), and finally the last case-study (SOFA 2 runtime extension) demonstrates the container-based form of the execution environment (EE3). The objective of the section is to demonstrate the role of execution environment during the system construction and identify important properties which need to be considered by the meta-component system.

jPapaBench represents an on-board control system for a unmanned aerial vehicle (UAV) implemented on the top of real-time Java virtual machine. The case-study is designed to support two kinds of real-time Java virtual machines represented by *Real-time Specification for Java* (RTSJ) [BGB+] and *Safety Critical Java* (SCJ) [17]) as well as regular Java virtual machine. The utilization of three kinds implementation technologies with their specificities has a direct impact on a form of used component system and its execution environment. The case-study uses ad-hoc module system – the modules are encapsulated and assembled with help of selected target implementation technology. Hence, there is no explicit execution environment and further due to real-time domain constraints the case-study does not utilize any kind of a library. The resulting system is self-contained and can be directly run in a virtual machine. Therefore, the ad-hoc modules are more runtime entities than architectural elements. Another important point, which needs to be emphasized, is the configurability of modules. The module implementation has to reflect differences in programming models of plain Java, RTSJ and SCJ.

The case-study has been also utilized as a real-time benchmark for evaluation different aspects of real-time virtual machines – for example, the paper [TPNV11] utilizes jPapaBench to perform a static analysis of code to detect violation of SCJ rules, while the paper [Puf11] elaborates hard real-time garbage collecting with help of jPapaBench, and finally authors of [ZM11] have utilized jPapaBench as an inspiration for their C++-based benchmark and further analysis.

RTSJ connectors case-study focuses on proposing a component model for design and implementation connectors for a clone of Fractal component system implemented with help of real-time Java. The case-study uses regular Fractal components to model and implement communication bindings and express real-time Java extra-functional properties. In this case, components build a runtime infrastructure which directly implements desired communication style reflecting the RTSJ specific programming model. The components have associated code generators producing RTSJ code. For execution, compilation with the Fractal library is necessary.

SOFA 2 runtime extension case-study focuses on SOFA 2 component system suitable for designing enterprise systems. The case study introduces support of components implemented with help of a scripting language. It demonstrates how the SOFA runtime composed of micro-components can be extended to support scripting languages and how the extension mitigate differences between the Java-based runtime model and its scripted extension. In this case, the micro-components express runtime infrastructure which is directly instantiated by the SOFA container providing the execution environment.

3.3 jPapaBench

The jPapaBench[2] case-study represents the domain of embedded systems, particularly the domain of Java-based real-time control systems. It utilizes an ad-hoc component system including a non-formal component model and the simplest variant of the execution environment (*i.e.*, variant EE1). The goal of the case-study is not only to show the simplest variant of the execution environment, but it also demonstrates reusability of components respecting differences in a technology used for execution environment implementation.

The case-study was originally presented as a real-time benchmark in the paper [KPMS10]. However, the section brings an extended version of the jPapaBench description published as the technical report [Mal12].

> [Mal12] Malohlava M.: *jPapaBbench – a Realtime Benchmark*, Technical Report n. 2012/1, Charles University in Prague, Department of Distributed and Dependable Systems, 2012. Published in the condensed form in [KPMS10].

3.3.1 Motivation

The Java programming language is well-adopted in various domains of software development due to its simplicity, huge set of additional libraries, large community and long-term experience.

However, the language itself does not guarantee any determinism or predictability of timing properties which represent important requirements for systems requiring

[2]The jPapaBench implementation can be downloaded from `http://code.google.com/p/jpapabench/`

real-time behavior (*e.g.*, control systems, embedded systems, large-scale trading systems).

Therefore, dedicated language specifications – *Real-Time Specification for Java* (RTSJ) [BGB+] and *Safety Critical Java* (SCJ) [17] – targeting development of real-time Java applications were initiated.

The shared attribute of these specifications is that they follow the original Java language specification, but extend and restrict the Java language programming model to achieve determinism and predictability of timing properties. These properties are guaranteed by modified garbage collector as well as by dedicated construct including explicit memory allocation, non-heap real-time thread un-interruptible by garbage collecting, etc.

Because of specificities of real-time domain, specification implementations in the form of dedicated Real-time Java Virtual Machine (RT-JVM) require to be extensively validated and verified if its execution characteristics and performance corresponds to the specification.

Such testing can be typically provided in the form of an application benchmark which would allow to characterize the properties of RT-JVM implementation under real-time workload. Nevertheless, the domain of real-time Java lacks a suitable benchmarks which would be based on real-life applications. On the other hand, many C-based real-time benchmarks exists ([NCS+06], [GRE+01], [SWR+99], [20]) targeting different real-time aspects (*e.g.*, WCET, response time, jitter measurements, distribution). Therefore, it would be beneficial to re-implement a selected C-based benchmark with help of different kinds of real-time Java not only to allow RT-JVM validation but also allow comparison to runtime characteristics of C-based implementation.

One of C-based benchmarks is a project called Papabench [NCS+06]. The Papabench project represents an example of non-trivial real-life real-time application benchmark. It implements a control system of unmanned aerial vehicle (UAV) originating from the Paparazzi project [1]. The C-based benchmark targets WCET-measurements and provides corresponding static characteristics of C-code (memory allocation rate, branching).

From the real-time Java perspective, the Papabench project seems as an ideal benchmark for RT-JVM because it: (i) provides a real-life control system of UAV which (ii) has real-time requirements; (iii) includes several tasks performing non-trivial computation; and (iv) has analyzable data- and control-flows. Furthermore, existence of Java and C version of the benchmark would permit to compare static and runtime characteristics of both implementations.

Therefore, this section introduces Java-based implementation of the Papabench benchmark called jPapaBench and presents its design and code characteristics. In summary, the section contribution is:

- real-life real-time benchmark for testing and verification of RT-JVM;
- design clarification with respect to supported three kinds of Java specification: plain Java, RTSJ, and SCJ;
- characterization of jPapaBench code.

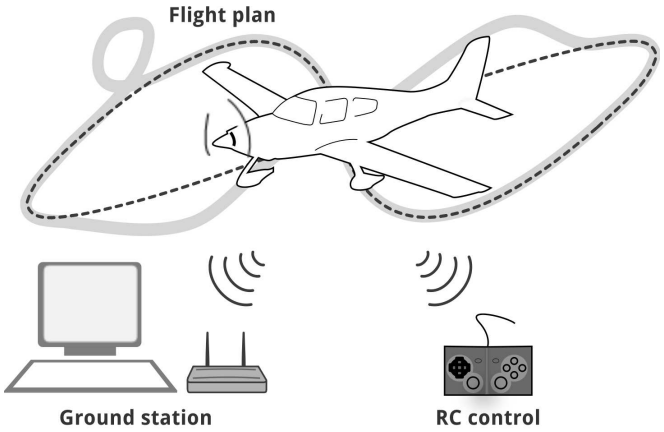


Figure 3.1: PapaBench application scheme.

3.3.2 PapaBench Overview

The Papabench benchmark [NCS+06] implements a control system of UAV which was originally deployed in the scope of the Paparazzi project [1]. While the Paparazzi project support several airplanes and their configurations, the Papabench project selects a particular airplane and relies on the corresponding software and hardware configurations. Nevertheless, the Papabench strictly follows the original Paparazzi software as well as hardware design.

Usage Scenario

The scenario in which the benchmark is deployed and executed is depicted on Figure 3.1. The autonomous flight of the airplane is driven by a predefined *flight plan*, which is loaded into the plane in advance and contains a description of the desired flight route and corresponding way-points which should be visited, eventually operations which should be performed (*e.g.*, , taking photos). The whole flight is monitored by a *ground station* which receives *telemetry* data from the airplane and provides their visualization. Furthermore, if a critical situation occurs during an autonomous flight, the airplane can be also controlled by a regular *radio control* (RC).

Airframe Design

Aircraft on-board hardware and software have to reflect the described usage scenario including autonomous flying, airframe configuration, and manual airplane control.

Regarding the hardware design, the airplane configuration consists of two computational $MC0$ and $MC1$ units connected via *SPI bus* and additional sensors and actuators. The first unit ($MC0$) controls actuators and manages received commands from radio control. That is why it is connected to actuators (airframe flaps, engine gas) and radio control receiver. It also communicates with the second computation unit $MC1$ via the SPI bus.

This unit is responsible for management and computation of the autonomous flight according to a loaded flight plan and present flight telemetry. Furthermore, the unit reports telemetry information to a ground station via a *datalink* typically in

a form of a modem. To estimate an airplane position and attitude, the autopilot unit utilizes GPS and IR sensors.

From the software perspective, each computational unit contains a deployed control system responsible for computation and communication with attached devices. The *fly-by-wire* control system, which is deployed on the $MC0$ unit, is responsible for configuration of actuators according to received commands from the autopilot unit and the radio control.

The second computational unit $MC1$ contains an *autopilot* implementation which manages the autonomous flight. It contains a navigator implementation which estimates the airplane position and attitude according to data from GPS and IR sensors and computes a new configuration for airplane actuators which is sent via SPI bus to the fly-by-wire unit.

Each control system includes several running tasks with priorities varying from 25ms to 250ms. The Papabench implementation uses a simple timeline scheduling without preemption. The side effect of the scheduling strategy is that no synchronization is placed in Papabench implementation.

The whole airplane control system works in two independent modes: (i) manual in which the fly-by-wire unit receives RC commands, passes them to autopilot which performs airframe stabilization and returns the commands for actuators; and (ii) automatic mode which is fully driven by autopilot unit according to the flight plan. The implementation also supports a fall-back mode, which is activated only when the autopilot does not response. Its activation causes setup up of actuators to a predefined fall-back configuration assuring safe landing of the airplane. In these modes all tasks are running but their computation paths are different.

3.3.3 jPapaBench Design

To bring Papabench into Java-based world, the resulting Java-benchmark has to satisfy several requirements. With respect to real-time Java, the resulting benchmark should fulfill common requirements summarized in Kalibera et al. [KHM+11] including object-oriented design, code size and complexity, memory management, or multithreading.

Additionally, in our case jPapaBench implementation should also fulfill the following aspects:

- preserve behavior of the original Papabench project, but use common Java constructs for the implementation. The resulting Java implementation should contains the same data-flow and control-flow as the original C-implementation;
- provides an implementation in plain Java and different flavors of real-time Java – RTSJ, SCJ Level 0, and SCJ Level 1;
- utilize common object-oriented and component design practices – separation of concerns via modularization, encapsulation, inheritance and polymorphism (which are not provided by the Papabench implementation);
- highly configurable (*e.g.*, change memory allocation strategy);
- verifiable in the sense of worst-case execution time (WCET). That means, for example, to avoid for unbounded loops.

The jPapaBench benchmark implementation stems from the original benchmark Papabench which is implemented in the pure C-language. The design of jPapaBench follows the structure of the original implementation to preserve its properties – number of tasks and their timing properties, synchronization, interrupts, number of external devices, and computation paths. However, the jPapaBench implementation has to also reflect object-oriented/component design as well as target selected Java-kinds.

Therefore, the high-level jPapaBench design follows a layered approach [SVB+06], where the implementation is divided into layers – only a higher layer can reference a lower layer (see Figure 3.2). The actual implementation contains four layers: a jPapaBench Core including entire benchmark computation logic, and three Java-kind specific layers encapsulating the jPapaBench Core into corresponding constructs. The jPapaBench Core design and implementation has to be flexible enough to enable such encapsulation, but should not change computation characteristics. The purpose of the design is to share the same computation characteristics among implementations in selected Java-kind.

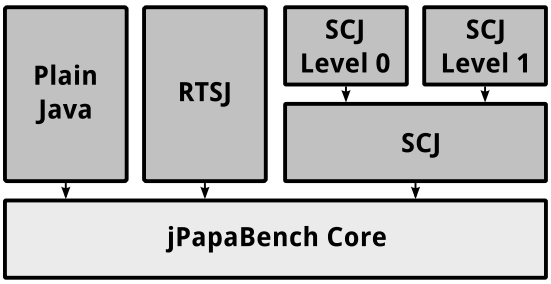


Figure 3.2: Layered jPapaBench structure.

The overall jPapaBench design relies on two kinds of components – *modules* and *tasks* which allow separation of individual concerns spread over original Papabench implementation. The module represents a static building block of the jPapaBench system including abstraction of hardware devices. The modules can reference each other and communicate via well-defined interfaces. The main objective of such separation is to explicitly locate a place in the control system where the data are stored (*e.g.*, aircraft position, attitude, telemetry data). Furthermore, component-based design is highly flexible and configurable – it permits to preserve the layered design, but, on the other hand, it also allows to a Java-kind specific implementation to introduce its specific implementation of a module if the original provided by jPapaBench Core cannot be adapted or encapsulated.

The tasks represent sources of computation. They operate over the modules and their data.

From the component-based perspective, the module represents a passive component, the task stands for an active component. Both have well-defined required, provided interfaces.

Module Design

The overall design of the jPapaBench Core depicted on Figure 3.3 follows the structure of the original Papabench implementation and identifies the components which are spread over original C-implementation.

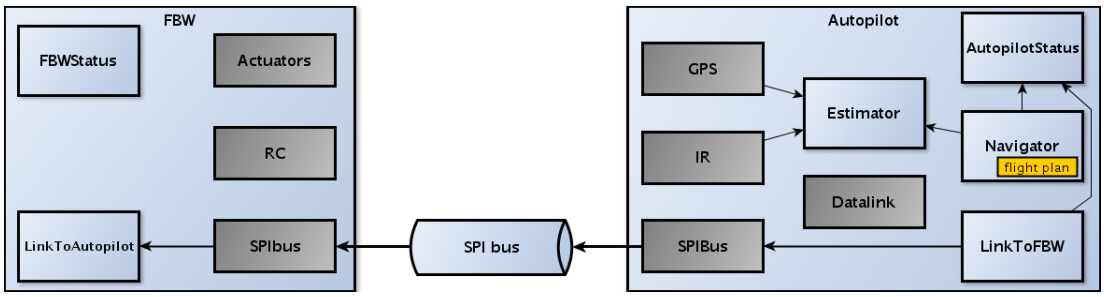


Figure 3.3: Overall jPapaBench design.

Autopilot		FBW		Simulator	
Task	T (ms)	Task	T (ms)	Task	T (ms)
Radio Control	25	ReceiveRC	25	Environment	25
Stabilization	50	SendToAutopilot	25	GPS interrupt	250
Fly-by-wire link	50	CheckFailsafe	50	IR interrupt	50
Reporting	100	CheckAutopilot	50		
Navigation	250				
AltitudeControl	250				
ClimbControl	250				

Table 3.1: jPapaBench tasks and their periods.

Two top-level modules – fly-by-wire and autopilot– correspond to software parts deployed on computational units $MC0$ and $MC1$. Each has a reference to additional modules representing hardware devices or sub-units.

The fly-by-wire module is composed of two sub-modules $FBWStatus$ (holding fly-by-wire status information) and $LinkToAutopilot$ (responsible for communication with the autopilot) and two devices – remote control commands receiver (RC) and SPI bus end-point ($SPIBus$).

The design of the autopilot includes the following four sub-modules:

- $Estimator$ holding estimation of aircraft position;
- $Navigator$ storing a current flight-plan;
- $LinkToFBW$ responsible for management of a message channel to the fly-by-wire unit;
- $AutopilotStatus$ storing a flight data (*i.e.*, telemetry).

Furthermore, the autopilot module controls three hardware devices: GPS, IR sensors and a datalink transmitting telemetry data to the ground station.

The modules reflect the static design of the jPapaBench core. However, to ensure correct airplane control (periodic) tasks needs to be running on each computational node. The jPapaBench contains periodic tasks and interrupts handlers. Their periods vary from 25-250ms (see Table 3.1).

The fly-by-wire module contains four computation tasks. Two of them – $CheckFailsafe$ and $CheckAutopilot$ – are responsible for checking the consistency of the whole system including autopilot responsiveness. If any error is detected

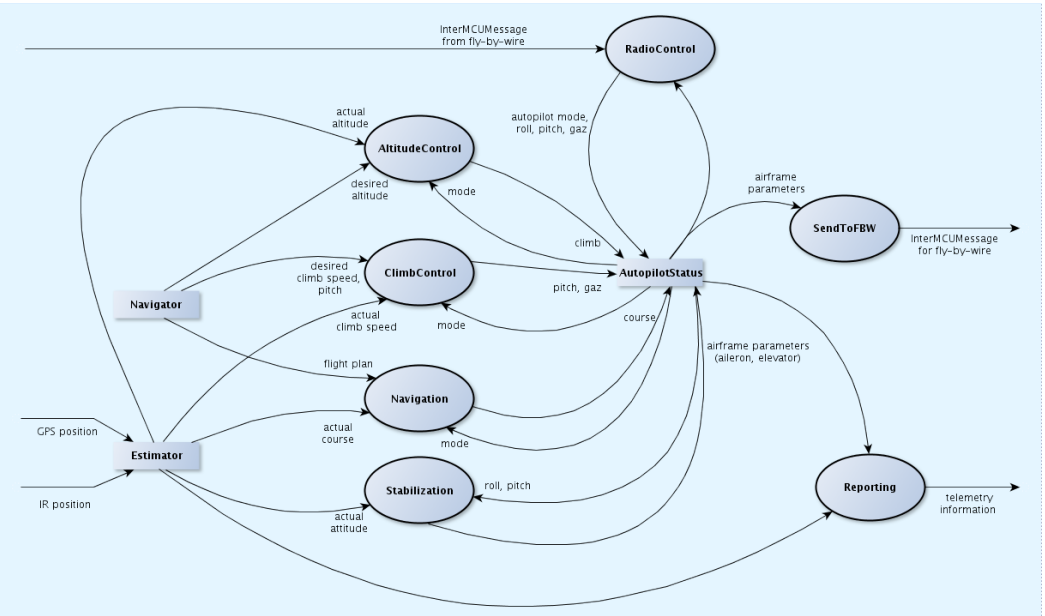


Figure 3.4: Autopilot unit data-flow.

the fail-back mode is triggered, which causes sending default commands to actuators and which should ensure safe landing of airplane. The $ReceiveRC$ task is responsible for receiving commands from radio controller which are stored in module $LinkToAutopilot$. It can also trigger switching between manual and auto mode of flying. The last fly-by-wire's task $SendToAutopilot$ sends messages to the autopilot unit.

A set of tasks deployed on the autopilot unit includes tasks responsible for autonomous flying, telemetry reporting and communication with the fly-by-wire unit. A collection of navigation tasks contains the $Navigation$ task which setup desired flying parameters according to a given flight plan. The parameters serves as an input for the consequent computation tasks $ClimbControl$ and $AltitudeControl$ which compute real-flying parameters based on actual plane position and attitude. The $Stabilization$ task translates computed flying parameters into an airframe configuration (flaps, gaz) which are then send by the $SendToFBW$ task to the fly-by-wire unit.

To give a deeper view of jPapaBench tasks, the following sections describes a data and control flow of the implementation.

Data-flow

Data-flow diagrams (Figure 3.4 and Figure 3.5) visualizes tasks and their data dependencies including inputs/outputs and data-stores in form of modules. The diagram shows data-flows for both units.

The autopilot itself contains three main data-flows (see Figure 3.4). The first reflects the computation of flight parameters according to estimate position of airplane and a desired flight plan. The second handles communication with the fly-by-wire unit and receiving commands from the unit. And the last one collects telemetry data and reports them to the ground station.

In case of the fly-by-wire unit, it involves two data-flows – the former handles

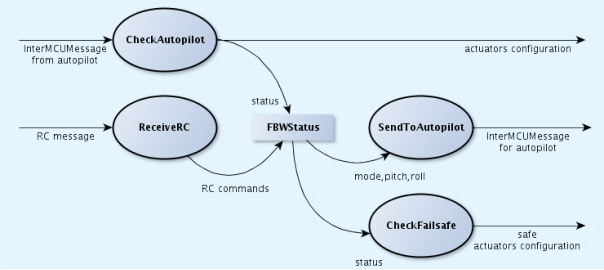


Figure 3.5: Fly-be-wire unit data-flow.

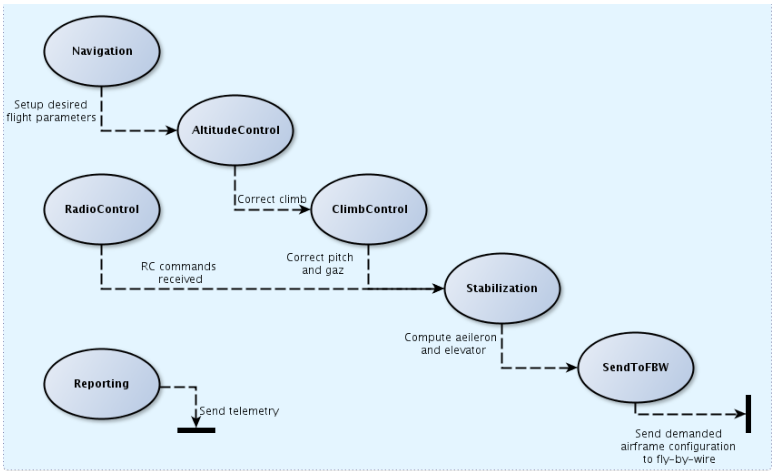


Figure 3.6: Autopilot unit control-flow.

radio control messages; the latter is interested in detecting and propagating fail-back values to actuators.

Control-flow

The jPapaBench control-flow for both units is depicted on Figure 3.6 and Figure 3.7. From the implementation perspective, the control-flow inherently influences tasks dependencies and hence it has to be reflected to assure the right computation. In the implementation the correct order is ensured by associating the right task priorities (in plain Java, RTSJ, SCJ Level 1), respectively by construction of a correct schedule (in SCJ Level 0).

The control-flow of the autopilot is based on a common control loop: the $Navigation$ tasks setups a desired flight course according to the flight plan stored by $Navigator$ module (see Figure 3.6). The desired flight course is then consulted with the estimated airframe position by $AltitudeControl$, $ClimbControl$ and $Stabilization$ tasks resulting into a configuration correcting the state of airframe. The configuration is then reported to the fly-by-wire unit.

The fly-by-wire unit includes two control-flows (see Figure 3.7). The first ensures that the commands received by the $ReceiveRC$ task are correctly passed to the autopilot unit. The second flow manages a fail-safe mode of the airplane – if a system inconsistency is detected by the $CheckFailsafe$ task, it directly configures actuators. Otherwise, the $CheckAutopilot$ task validates status of the autopilot and in the case

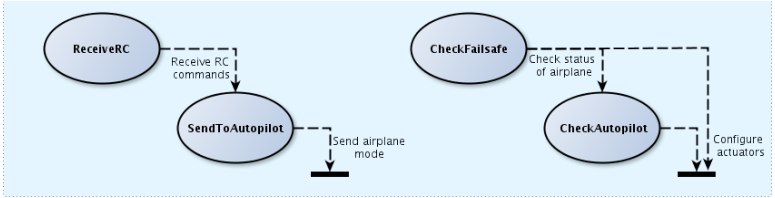


Figure 3.7: Fly-be-wire unit control-flow.

jPapaBench version	Java kind	Scheduling
jPapaBench$_{PJ}$	plain Java	Executor service (fixed rate)
jPapaBench$_{RTSJ}$	RTSJ	Default – fixed-priority preemptive scheduling
jPapaBench$_{SCJ-L0}$	SCJ Level 0	Default – timeline scheduling (cyclic executive)
jPapaBench$_{SCJ-L1}$	SCJ Level 1	Default – fixed-priority preemptive scheduling

Table 3.2: jPapaBench scheduling policies.

that autopilot is not accessible, actuators are configured to fail-safe values too.

3.3.4 Technology Mapping

The core design is written in the plain Java and does not assume any real-time properties. Thus, to provide a kind of RT-benchmark, the core has to be encapsulated into constructs of a selected RT-Java which ensure correct implementation of real-time properties. The current jPapaBench version provides three kinds of encapsulation(see Table 3.2).

Plain Java

The plain Java encapsulation focuses on rapid-prototyping, testing, and verification of the implementation. For periodic task execution it utilizes a common Java *executor service* with fixed execution rate.

RTSJ

The version encapsulating the core into RTSJ constructs and utilizes $RealtimeThread$ for periodic tasks execution. There are two implementations of the thread – the former does not allow any allocation in task implementation, the later uses scope memory for task's allocated data.

SCJ

The SCJ version of jPapaBench provides implementations for SCJ levels 0 and 1. Each level's design encapsulates jPapaBench initialization into the $Mission$ construct. While the level 0 provides an explicit tasks schedule with minor cycle length equal to 25ms and major cycle with 250ms length, the level 1 utilizes priority-based tasks scheduling.

3.3.5 Environment Simulator

To run and test jPapaBench, an environment simulator is provided. As the rest of the system it is also implemented in conformity with the existing Papabench simulator written in C and OCaml. However, in the case of jPapaBench, the simulator is a part of running system and it is deployed in the same VM as jPapaBench implementation.

The simulator implementation consists of three tasks managing a *flight model*. The flight model carries data necessary for environment simulation – *e.g.*, last position and attitude of airplane. The task $EnvironmentSimulator$ periodically recomputes the flight model according to the actual configuration of actuators and the last computed flight model. The periodic tasks GPS and IR simulate behavior of GPS and IR devices interrupts. Their computations depend on the computed flight model and provides GPS position and IR attitude data with respect to the C-implementation characteristics.

3.3.6 Workload

The workload is specified in the form of a flight plan. It declares way points which should be visited by the aircraft. Furthermore, the flight plan can directly describe desired parameters of the flight such as climb, attitude, or speed. If the flight plan specifies the parameters, the $Navigation$ task has to correct the real flight parameters respectively. The flight plan itself is divided into so called *navigation blocks* representing a navigation target (*i.e.*, to achieve the given way-point, altitude, or speed). The transition between blocks are driven by specified conditions reflecting the position, time, or current number of navigation cycles.

For the purpose of benchmarking, the whole flight plan can be bound to a given number of navigation cycles. The purpose of the flight plan is to test different computation paths of jPapaBench implementation.

3.3.7 jPapaBench Code Characteristics

Code Complexity and Size

To characterize the complexity of jPapaBench implementation we follow the strategy proposed by Blackburn et al. [BGH+06] and further adopted also in CD_x benchmark evaluation [KHM+11]. The Chidamber and Kemerer [CK94] metrics (*CK-metrics*) characterize the complexity regarding various aspects. In the case of jPapaBench, *ckjm* tool [30] was utilized to obtain CK-metrics. It provides the following metrics:

WMC represents weighted methods per class. For ckjm the value represents the number of methods declared by class.

DIT determines a depth of classes inheritance tree. However, the metric does not compute with interfaces. The value is connected with the metric NOC.

NOC is a number of class's immediate children. The values show, that the jPapaBench core heavily uses inheritance, while the Java-kind specific modules do not. That is caused by the fact, that they encapsulate the core with help of composition which is more advantageous from the perspective of configurability than using a plain inheritance.

CBO characterizes coupling between classes based on a number representing a number of classes coupled to the given class (*e.g.*, via inheritance, field access). In case of jPapaBench, higher coupling is typical for helper classes or factories, lower for tasks.

RFC determines how many methods can be invoked if a method is executed.[3]

LCOM represents a number counting class's methods which are not related through of sharing a class field. Higher number can signal that the class should be split because it seems to be interested in different concerns.

The resulting measurements of jPapaBench are shown in Table 3.3. In comparison with existing open-source project, the jPapaBench object-oriented metrics determines satisfiable object-oriented design with good separation of concerns. Comparison the values with measurements of existing Java benchmarks (DaCapo, CDx) shows that jPapaBench is slightly more complex.

Additionally, to characterize real size of jPapaBench we have utilized *Sonar* tool [2] extending CK-metrics by providing class-size statistics:

Classes contains a number of classes which are declared.

NCLOC/Physical lines identify the number of lines of executed code versus number of physical lines corresponding to the number of carriage returns. Such comparison gives also intuition how well is code commented.

Statements characterizes the number of Java statements as they are defined in *Java Language Specification* [25]. The number is increased when the expression or statement (if, else, while, do, for, switch, break, continue, return, throw, synchronized, catch, finally) is processed. The lower number of statements characterizes classes which carry only data (typically POJOs), higher number of statements is typical for classes which contain computation.

Complexity determines a cyclomatic complexity (McCabe metric [McC76]) for a class. It is defined as a sum of cyclomatic complexities of class's methods. The number itself determines the possible control paths within a method which is useful for WCET analysis or program verification.

Table 3.4 identifies that in comparison with CD_x benchmark the jPapaBench has slightly lower number of code lines. Moreover, cyclomatic complexity detects that the majority of jPapaBench methods contains only one control-flow path. However, there are several methods which are complex containing more than twenty control-paths. Typically, such a method includes a case-statement causing high number of cyclomatic complexity.

With respect to these measurements, jPapaBench can be considered as a non-trivial project with a medium size code base. Its design follows best-practices of object-oriented design and splits different concerns into separated modules. As a result, the jPapaBench represents a non-trivial object-oriented real-time Java benchmark.

[3] The ckjm tool computes only methods within a class body

	WMC			DIT			NOC		
	Med	Max	Sum	Med	Max	Sum	Med	Max	Sum
jPapaBench Core	2	33	795	1	3	258	0	40	59
jPapaBench$_{PJ}$	5	12	25	1	1	5	0	0	0
jPapaBench$_{RTSJ}$	3	11	36	1	4	17	0	1	1
jPapaBench$_{SCJ}$	5	10	62	2	6	32	0	0	0
Simulator	2	17	66	1	2	18	0	0	0
All code	2	33	984	1	6	630	0	40	62

	CBO			RFC			LCOM		
	Med	Max	Sum	Med	Max	Sum	Med	Max	Sum
jPapaBench Core	4	24	673	9	48	1708	0	435	3800
jPapaBench$_{PJ}$	11	25	62	14	52	100	2	46	59
jPapaBench$_{RTSJ}$	1	26	70	8	46	133	2	41	67
jPapaBench$_{SCJ}$	3	40	89	16	47	203	8	45	164
Simulator	4	8	57	6	33	129	0	110	191
All code	4	40	951	9	52	2273	1	435	4281

WMC	*Weighted methods per class*	CBO	*Coupling between object classes*
DIT	*Depth of inheritance tree*	RFC	*Response for a class*
NOC	*Number of children*	LCOM	*Lack cohesion in methods*

Table 3.3: Chidamber and Kemerer Java Metrics for jPapaBench

	NCLOC/Physical lines			Statements		
	Med	Max	Sum	Med	Max	Sum
jPapaBench Core	18/61	195/367	3451/7700	3	79	1106
jPapaBench$_{PJ}$	25/62	123/197	213/439	8	46	75
jPapaBench$_{RTSJ}$	24/62	110/180	266/584	4	42	83
jPapaBench$_{SCJ}$	51/105	96/163	499/1029	11	29	145
Simulator	20/57	114/171	404/921	4	55	124
All code	21/62	195/367	4833/10673	4	79	1533

	Complexity			Classes
	Med	Max	Sum	
jPapaBench Core	4	32	718	98
jPapaBench$_{PJ}$	1	13	20	5
jPapaBench$_{RTSJ}$	4	9	33	8
jPapaBench$_{SCJ}$	9	12	65	9
Simulator	2	11	44	13
All code	4	32	880	133

Table 3.4: Class properties

3.3.8 Lessons Learned

The jPapaBench represents a non-trivial case-study dealing with implementation of a real-time system. Its design and implementation is constrained by characteristics of the original C-based implementation – the jPapaBench implementation respects predefined data- and control-flow. However, the resulting system is also well designed with respect to component-based development.

Utilization of multiple implementation technologies brings a new level of variability which needs to be reflected by the system architecture. Furthermore, the implementation itself manages manually assembly and instantiation of application components. There are two reasons for this – (i) each implementation technology has a slightly different way to assemble and execute the system, (ii) respecting explicit memory management of RTSJ/SCJ, it is almost impossible to utilizes existing containers, libraries, and methods due to they often rely on unmanageable hidden memory allocation (*e.g.*, typically creation and manipulation with $java.lang.String$, $java.util.List$ instances).

From the component perspective, the jPapaBench implementation demonstrates the following properties:

- **Ad-hoc modules** – the design utilizes a lightweight flat component framework with explicit bindings. The system introduces two kinds of components – active (called tasks) and passive (called modules).
- **Manual assembly of the system** – the resulting system is assembled from components which are encapsulated into a selected implementation technology. The technology has a direct impact on the assembly process – for example, there are different ways for active component instantiation.
- **No explicit runtime environment** – resulting assembly is directly executable on the top of a JVM.
- **Non-functional requirements** – the jPapaBench involves timing and memory management. The former one is reflected by active modules, the second needs to be reflected by a structure encapsulating a component. There is no explicit mechanism to inject or declare these properties. They are directly encoded in the implementation of the execution environment.
- **Configurability** – the jPapaBench implementation exposes two configuration points. The first point considers modular design and allows for exchanging a module with a new module satisfying substitutability principle. The second configuration point is enforced by employing different kinds of implementation technologies to produce an executable system. The technologies directly influence the glue code assembling components together.

3.4 Real-time Java Connectors for Fractal Component System

The section introduces an example utilizing a rigorous component system in the scope of Java-based real-time embedded systems. Nevertheless, in this case the component system is not used for application development, but it supports design and implementation of bindings among component. While the previous case-study demonstrates

utilization of the simplest variant of the execution environment, this example employs an execution environment which is partially generated and supported by a library (variant EE2). The content of this section is based on the paper:

> [MPL+08] Malohlava M., Plšek A., Loiret F., Merle P., Seinturier L.: *Introducing Distribution into a RTSJ-based Component Framework*, In Proceedings of 2nd Junior Researcher Workshop on Real-Time Computing, Rennes, France, Oct 2008.

3.4.1 Introduction

An upcoming era of massively developed real-time systems brings a challenge of developing large-scale, heterogeneous and distributed systems with variously stringent QoS demands. To keep a complexity of such systems at reasonable levels, emerging solutions in this area are recently based on RTSJ [BGB+] since it embeds real-time properties such as predictability and determinism into a general-purpose programming language.

However, the aspect of distribution in such systems still represents a challenge and brings many open issues. The state-of-the-art of distributed and real-time Java lies at its very beginning. A few proposals introducing specifications, profiles or frameworks [AJ06, WCJW02, TAdM07] have been conducted, however, there is still a need of a comprehensive solution proposing a full-fledged approach that would mitigate complexities of real-time programming in distributed systems.

The previous work [PLMS08] proposes a component framework for development of RTSJ-based systems. The framework provides a continuum between design and implementation of such systems and offloads burdens from developers by automatically generating an execution infrastructure of RTSJ-based systems. Nevertheless, the aspect of distribution have not been addressed there. We however envisage that supporting development of distributed real-time systems is a highly desired feature, therefore as the key contribution of this work we focus on extensions of the framework towards distribution support.

Related Work

The research area of distributed programming in the scope of real-time Java includes several research directions. The leading initiative is represented by an integration of Remote Method Invocation (RMI) into the RTSJ [WCJW02] and solving the task related issues such as handling real-time properties [BW03, WCJW02] or memory allocation [BVGVEA05, BW03]. The results of these projects are reflected in a status report of *JSR 50* [AJ06] which tries to cover all aspects of distribution (real-time properties handing, failure semantics, distributed threads and their scheduling). A similar approach proposes a profile for distributed hard real-time programming [TAdM07], however, a framework addressing a comprehensively challenge of developing such a complex system still has not been proposed.

Another research area covers the *Real-time CORBA specification*,[4] which can serve

[4] OMG, Real-time CORBA, v1.2, `http://www.omg.org/technology/documents/formal/real-time_CORBA.htm`.

as a particular base for a requirements analysis of real-time distributed systems. Its main implementor in the RTSJ world is RTZen [RZP+05]. Although it is a middleware implementing almost all parts of the Real-time CORBA specification within the scope of RTSJ, it only focuses on a core of communication and does not provide any suitable modeling abstraction.

From our point of view, all these projects focus only on low-level communication issues and their integration into the scope of RTSJ, they do not address any higher abstraction of the real-time communication. It could however be beneficial to reflect distribution in different stages of the application lifecycle (design, implementation, runtime).

3.4.2 Challenges of Distributed RTSJ-based Designing and Programming

Integration of distribution into a real-time component framework [PMS08] is a challenge involving an analysis of requirements dedicated to real-time systems as well as requirements coming from used RTSJ. All these requirements affect not only a way of specifying model artifacts (components, bindings and their properties) but also its runtime structure and the process of its initialization.

In this section we therefore determine a scope of requirements which have to be reflected by a distributed system within a real-time environment at all stages of the application lifecycle.

Requirements and Challenges

Real-time properties. Since real-time programming introduces specific requirements on distributed systems (*e.g.*, priorities of running tasks, computational deadlines), they play a substantial role during their development. These properties influence remote connections and superimpose new constraints over them. Some realtime properties have to be propagated between remote parts of applications (*e.g.*, priority of a client thread) and others have to be reflected during creation of the connection (*e.g.*, end-to-end time).

RTSJ requirements. Moreover, employing RTSJ in development of distributed RT-Java-based systems is also affected by the particularities of its specification. It distinguishes between a heap memory and non-heap memory and specifies how they can be accessed by the different threads. These facts enforce different memory allocation ways as well as a specific utilization of schedulable entities (threads, timers, events). However, the specification silences about distribution aspects and therefore these complexities need to be resolved by the developers.

Integration level. Furthermore, the integration of distribution into a component framework yields a decision at which level of abstraction the distribution will be incorporated into the framework and how a component-application developer will manipulate with real-time properties. Whether to hide the manipulation from the developer or not.

These questions were discussed in the scope of RMI integration into RTSJ presented in [WCJW02]. It distinguishes three basic levels of RMI integration (denoted as

L0, L1 and L2) from different views.[5] L0 is the minimal level of the integration with no support for real-time properties from underlying technology. The level L1 requires a transparent manipulation with scheduling parameters or timing constraints and finally L2 declares semantics for the distributed thread concept [AJ06] which represents a fully transparent real-time programming model.

We partially adopt this idea of integration levels in our approach. The primary objective is an integration of distribution into the RTSJ-based component system at a level corresponding to L1. We however generalize the idea of the level L1, originally tightly coupled with RMI, to address the full span of possible communication middlewares (RMI, CORBA) in distributed environments. We therefore address the following contributions to meet this generic goal:

(i) **Scheduling Parameters.** To handle transparently scheduling parameters which are associated to component threads. The task involves a transportation of parameters from a client to a server where it is required, configuration of an underlying middleware (*e.g.*, in case of CORBA, creating priority lanes), pre-reservation of connections for selected priorities, etc.;

(ii) **Determinism.** To ensure that the generated runtime infrastructure does not affect determinism and timely delivery assured by a used underlying middleware;

(iii) **RTSJ Rules and Restrictions.** To handle memory and thread differences between components and a used middleware. This also covers handling of a memory allocation of call parameters (*e.g.*, . CORBA parameter holders) and auxiliary artifacts (*e.g.*, adaptors, call serializers),

(iv) **Communication Styles** To provide different communication styles [Bur06, MMP00] which are common in the real-time and embedded systems world (synchronous and asynchronous method call, asynchronous messaging).

Goals

Our philosophy postulated in [PMS08] states that the RTSJ concerns influence the architecture of applications and therefore must be considered at early stages of the system development lifecycle. We have applied this in [PLMS08] where we propose a framework clarifying all the steps of the system development lifecycle. In this work we follow the same principles, there are therefore two key objectives:

(i) **Development Methodology** that clarifies specification of model artifacts and properties that will cover distributed real-time requirements and create an abstract layer which will hide low-level distribution concerns from component designers, and

(ii) **Execution Infrastructure** that manages transparent deployment and run of distribution support inside the execution infrastructure.

[5] *Programming model* (identification of remote objects), *development tools* and *implementation model* (real-time properties transport mechanism). In our case, the first two models are realized by the component framework [PMS08] therefore the following text is interested in the third one.

Although distribution has to be captured at all stages of the system development lifecycle, in this section, we focus on the design and generation of an execution infrastructure.

3.4.3 Supporting Distribution in Real-time Java

The basic idea of our approach is inspired by a solution in which components communicate through architecture-level software connectors that are implemented using a middleware [MDT03]. This approach preserves the properties of the architecture-level connectors while leveraging the beneficial capabilities of the underlying middleware. Moreover, we integrate this approach into the *Soleil* framework proposed in [PLMS08].

Soleil is the execution infrastructure generator that generates a system's infrastructure on the basis of a given architecture. Thus we automatically obtain connector implementations, consequently mitigating complexities of the system development. Additionally, the process of designing and implementing connectors addresses the real-time challenges identified in Section 3.4.2.

From the high-level point of view, we adopt a general approach to a connector generation presented in [Bur06], we however focus on more lightweight and especially RTSJ tightly coupled solution.

Applying Component Connectors

Design Time. At design time we perceive connectors as representations of bindings between functional components. A binding has attached non-functional properties such as monitoring, enforcement of a dedicated communication channel or prescribed utilization of a given middleware. Furthermore, the binding connects components which also have associated properties (*e.g.*, call deadlines for interface operations) or they receive derived properties from non-functional components in which they are placed (*e.g.*, memory allocation context, thread priorities). All these properties are reflected in the connector architecture representing the binding.

The proposed connector architecture is based on a concept of *chains of interceptors* which are connected to concerned interfaces, as illustrated in Figure 3.8. Each interceptor in the chain symbolizes one non-functional concern which reflects communication *in situ* (*e.g.*, monitoring) or modifies communication (*e.g.*, adaptation of method call parameters). The presence and position of the interceptor in the chain is influenced by properties of the modeled binding and also by a presence of other interceptors.

This representation of connectors allows us to build them easily with different functionalities – by selecting relevant interceptors and their order in according to specified properties. As well as, the division of the connector architecture into separated interceptors permits handling real-time specific properties separately in dedicated interceptors. The chosen architecture also brings advantages in dealing with issues triggered by using RTSJ [PMS08] such as memory scopes crossing or copying between memory areas.

Generation Process. The connector generation process includes:

i **Chain Structure Selection.** Which involves selecting interceptors and their order

in according to binding properties (specified and derived) and also to RTSJ requirements, *e.g.*, selecting memory allocation areas and adapting memory or thread differences;

ii **Interceptor Code Generation.** The task involves generation of interceptors and of a selected middleware specific code (*e.g.*, initialization of middleware, setting connection parameters).

Furthermore, different optimizations in the chain or in its selected parts are possible, similarly as proposed in [PLMS08].

Runtime. The preservation of the connector architecture at the runtime level permits modification of connector attributes. Either simple attribute modifications affecting only one interceptor are possible (*e.g.*, modification of middleware threads priority) or even more advanced adaptations of the connector structures can be performed (*e.g.*, update of interceptors in a chain, change of the interceptors order).

Illustration Example

The proposed concept was applied in an implementation of a simple example presented in [PLMS08]. Concretely, we model a real-time communication between two active components – *ProductLine* and *MonitoringSystem* allocated in a non-heap memory. Both components have associated properties defining components' thread priorities. The binding between these components is modeled as a remote binding with two associated non-functional properties — the first one enforces utilization of a distribution enabling technology (in our case we use *RTZen* middleware [RZP+05]) and the second one identifies an asynchronous method call.

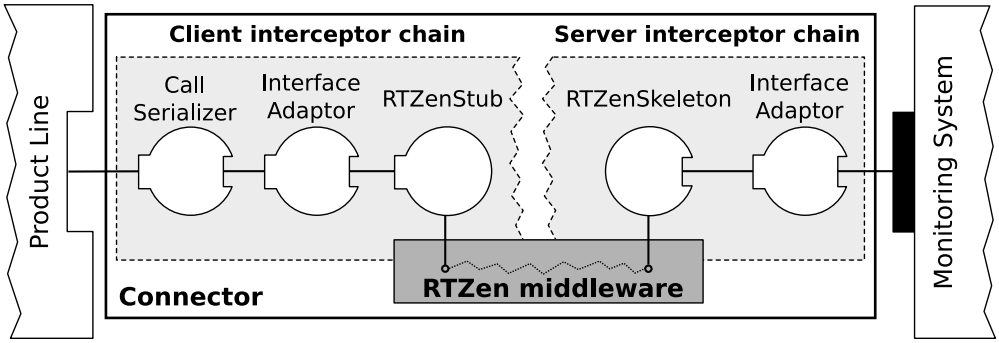


Figure 3.8: Structure of RTZen-based connector.

These simple properties involve several tasks which has to be covered by the generated connector and its *chain* architecture:

(i) implementation of core distribution with help of RTZen. This also involves generation of low-level CORBA interfaces, helpers, value holders in according to a specified IDL;

(ii) configuration of underlying middleware – adjustment of CORBA policies to reflect components' thread properties;

(iii) asynchronous method calls in case the underlying middleware does not support them; and

(iv) adaptation between memory areas;

(v) adaptation between functional and internally generated interfaces.

The core of the distribution implementation is generated in interceptors called *RTZenStub* and *RTZenSkeleton* which mediate the communication with help of RTZen middleware. At the server side, *RTZenSkeleton* registers itself as a remote object in RTZen and serves like a proxy which delegates calls to a following interceptor which adapts an internally generated interface to the server component's functional interface. The *RTZenSkeleton* interceptor also configures a priority with which remote calls will be handled.

At the client side, *RTZenStub* obtains, via calling the encapsulated RTZen middleware, a reference to the remote object and delegates all incoming calls to it. However, this reference implements the internally generated interface, therefore it has to be adapted to the functional interface by another interceptor called *Adaptor*. Finally, the *Serializer* interceptor arranges asynchronous semantics for method calls – each call on its provided interface is stored in a local queue and then served by a thread associated with the queue.

3.4.4 Lessons Learned

The case-study incorporates the Fractal component system for modeling and development bindings among components in the context of RTSJ-based systems. As in the previous case-study, the selected implementation technology (*i.e.*, RTSJ) directly enforces the kind of non-functional properties which need to be reflected (*i.e.*, memory allocation, priorities, deadlines). Even thought, the approach utilizes a general component model based on Fractal to design a binding structure. Another important identified property is utilization of adaptation in large scale. The component interfaces, component content, and execution infrastructure are adapted according to the kind of binding. There are two kinds of adaptation – the former is based on code generation, the latter employs code adjustment (by code merging). Both of methods are used during execution environment construction – in this case-study, the prepared code is linked with the Fractal library supporting the code.

To summarize, the case-study shows following properties:

- **Utilization of a component model with explicit bindings and control layer.** The component model is based on the Fractal component model introducing hierarchical components and explicit bindings. Moreover, each component has also its control membrane. The membrane serves to manage non-functional properties and memory allocation scopes for components composing the binding.
- **Binding's chains of components directly becomes a part of runtime environment.** A binding is represented as a chain of components (they play role of interceptors). The chain is a composite component which is managed by the Soleil tool which generates necessary binding infrastructure. It is necessary to point out, that the generated binding infrastructure directly becomes a part of

the overall application infrastructure. Thus, during deployment and runtime there is no difference between binding's components and business-level components – the approach utilizes the same formalism and infrastructure to manage both.

- **Adaptation and code generation of component interfaces, content, and infrastructure.** The component system heavily relies on code adaptation and generation. However, the code is always linked with the Fractal runtime library (contrary to the previous case-study which does not contain any runtime library).
- **Implementation technology impact.** The utilized RTSJ has an impact on design and implementation of the component system. The approach uses the Soleil framework which brings into the Fractal component model a notion of domain components to express RTSJ specificities. Domain components serve to model memory allocation scopes and thread domains which directly reflect RTSJ concepts and which need to be considered during the execution environment preparation. Moreover, control membranes of utilized components have to be tailored to satisfy RTSJ rules.
- **Multiple component system.** The interesting property of the case-study is utilization of the dedicated component system to implement bindings in the context of another component system. The combination demonstrates benefits of modeling execution infrastructure (*i.e.*, a binding in the context of this case-study) with help of components.

3.5 SOFA 2 Runtime Extension

The last case-study is concerned with the SOFA 2 [BHP06] component system focusing on development enterprise systems. It provides an advanced container to launch and execute component-based applications (*i.e.*, EE3 variant). The case-study focuses on the container configurability and demonstrates its extension providing new functionality. The section is based on the following paper:

> [KMBH11] Keznikl J., Malohlava M., Bureš T., Hnětynka P.: *Extensible Polyglot Programming Support in Existing Component Frameworks*, In Proceedings of 37th Euromicro Conference on Software Engineering and Advanced Applications, Oulu, Finland, Aug 2011.

3.5.1 Introduction

Nowadays, agile development methodologies often incorporate rapid prototyping as a method of early deployment and fast delivery of working solutions. Furthermore, prototyping is often utilized for demonstrating new ideas, writing system tests, simulations or mocks. The cornerstone of rapid prototyping is an "elastic technology" which can be easily utilized in different contexts, for different concerns. Furthermore, such technology have to be inherently dynamic to allow on-the-fly changes of the implementation and re-deployment.

In the context of the systems without inherent support for rapid prototyping, *polyglot programming* ([32], [WC10]) incorporating multiple languages for building systems represents a well-adopted direction. Its idea is to utilize dedicated, especially scripting/dynamic, programming languages for particular parts of system representing different concerns. A typical example is JavaScript [GME07] which is primarily used in web development to improve HTML based user interfaces and incorporate asynchrony into the stateless HTTP protocol; or the Groovy programming language which is largely utilized, for example, for writing system unit tests. Utilization of scripting languages permits fast development without a long cycle between change and its deployment in a running system.

From the rapid prototyping perspective, the polyglot programming with scripting languages is preferably used during the development phase in order to derive the correct implementation of the individual system parts, or to implement tests and mocks. Production-ready code is then implemented in a particular programming language according to the prototypes.

The rapid prototyping based on polyglot programming with scripting languages is particularly important in the domain of component-based systems as they have a long development cycle involving a dedicated cycle of component development [CCL06], and require complex testing environments for a particular component as well as whole system.

However, the existing advanced component-based frameworks, such as EJB [27], OSGi [4], Fractal [BCL+06], SOFA 2 [BHP06], JBoss [FR03], COM [21] – *i.e.*, those with existing runtime environment often called as *component container* – lack the required dynamic features as they are based on statically typed programming languages. Therefore, to allow rapid prototyping in these frameworks the support for polyglot programming is necessary.

Despite the spread of polyglot programming in other domains [18, 16], as far as the mentioned advanced component-based frameworks are concerned, the possibility of combining components implemented in different programming languages in a single application is still not usual. Therefore, it is not only difficult to rapidly prototype new components, but also to write test- and mock-components for building testing environments. This is primarily caused by the fact that (i) either the component frameworks were not designed with such heterogeneity in mind (*e.g.*, Fractal, EJB), or (ii) the component frameworks support heterogeneity, but require a dedicated component container for each language (*e.g.*, CORBA and CCM [13] or SOFA 2), which turns out to be very heavyweight and difficult to manage and use.

The need of polyglot programming support in advanced component frameworks is partly satisfied by introducing virtual machines and common runtime mechanism (*e.g.*, .NET CLR or JVM). These techniques allow for programming language binding on the byte-code level, thus making the combination of different languages invisible for the concerned component framework and components. However, the problem still persists when a particular language does not have a compiler for compilation to the byte-code – *e.g.*, it is purely interpreted, possesses features which cannot be mapped to the byte-code or the compiler just does not exist.

These problems are especially true for JVM, for which only a few languages may be integrated on the byte-code level (currently Java, Groovy, Scala and Clojure), but many other languages are already supported by interpreters implemented in Java (*e.g.*, JavaScript, PHP, Python, Ruby, Erlang, Prolog) or via JNI binding (C, C++, R, Fortran,

Ada, MATLAB). The result of it is that Java-based component frameworks still need some support for business code provided in different programming languages, which in turn means the necessity to partially re-implement the concerned component frameworks.

Therefore, we propose an alternative approach dealing with the problem. A number of advanced component frameworks inherently support extension mechanism, which allows extending their runtime environments by introducing interceptors around the component's business code. The interceptors are responsible for relaying calls coming to and going out of an instantiated component. By reacting and possibly modifying the calls, interceptors may influence of the behavior of the component framework and enrich it of additional capabilities without having to change its core implementation.

Goals

The section contributes to the problem of rapid prototyping of component-based systems by introducing a component system extension to address the issue of polyglot programming with scripting languages in existing Java-based component frameworks that support user-defined interceptors. We provide a solution that allows seamless and transparent runtime integration of components implemented using different programming languages inside one component container. Furthermore, the solution inherently allows easy updating of a component implementation at runtime.

The presented solution is generally applicable for Java-based component frameworks with the extension mechanism based on custom interface interceptors (*e.g.*, Fractal, SOFA 2, JBoss, Spring, Castle). The solution is a pure extension, which does not require modification of component framework internals and it can be easily extended to support additional programming languages. Furthermore, the solution stresses separation of concerns by separating the support for different languages into component interceptors. Therefore, the component developer does not have to deal with any language integration and scripting frameworks – he or she just provides component implementation in a particular scripting language.

In addition to describing the solution concepts, we present results of a performance benchmark to assess the performance impact of our approach, and we discuss the lessons learned from implementing the solution for SOFA 2 component framework.

Section Structure

The rest of the section is organized as follows. In the next section the component systems for which the presented approach is applicable are defined. In Section 3.5.3 analyzes and generally describes the presented approach. Section 3.5.5 presents a case implementation allowing scripting support in the SOFA 2 component system, evaluates the component developer experience and gives a brief performance analysis of the approach.

3.5.2 Prerequisites and Definitions

Our approach allows introduction of components written in scripting languages into an existing component framework and their combination with the components native

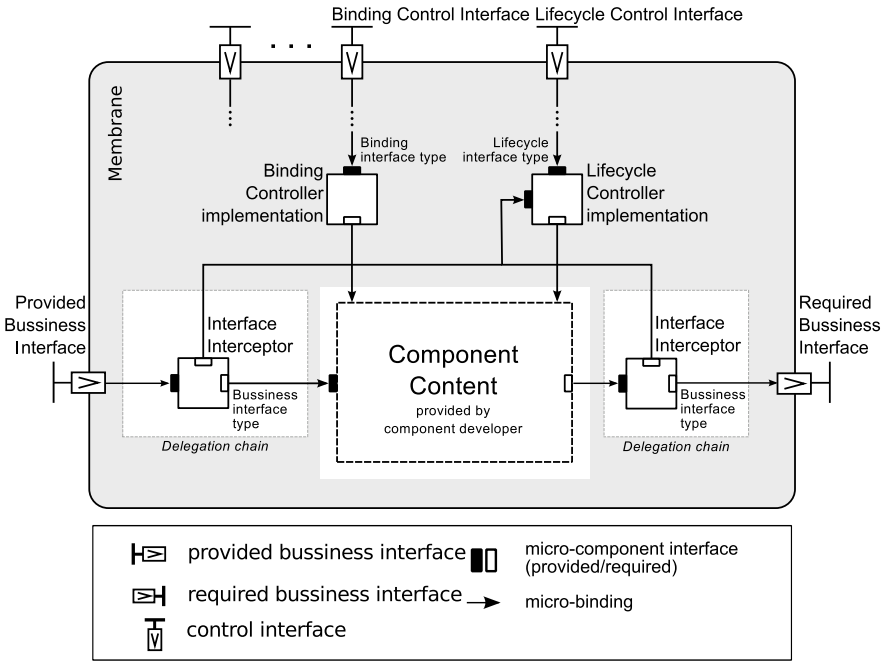


Figure 3.9: Platform component overall view.

to the framework. This allows for rapid prototyping of selected parts of a component application.

We assume that the associated component container is extensible in the way that it provides the possibility of introducing custom component interface *interceptors*. This has already become a de-facto standard way of extension, which is currently supported by many component frameworks (*e.g.*, all Fractal-based component systems [BCL+06, SPDC06], SOFA 2 [BHP06], JBoss [FR03], Spring [JHRS05], Castle). Although, the terms used in each component framework differ to some extent, the essential feature of custom interceptors is the same – relaying of component interface calls. In this way, we can recognize a family of interceptor-based component frameworks, for which the solution presented in the section is generally applicable.

To alleviate explanation, we use the terms featured in the Fractal specification [BCL+06] – we recognize a *component content*, which is the business logic of a component provided by the component developer, and a *component membrane*, which is the wrapper around the content that contains the interceptors (see Figure 3.9). The membrane thus displays the *component interfaces* on its outer boundary and relates them to the content. There are essentially three main kinds of interfaces: *provided business interfaces*, *required business interfaces*, and *provided control interfaces*. Business interfaces reflect the core business functionality of the component provided by the developer, thus they typically differ for each component. On the other hand, control interfaces (also termed *controllers*) reflect the management side of the component – *e.g.*, lifecycle management, component bindings, interface queries (similar to Microsoft COM's `IUnknown` interface), etc. Controllers provide the component container and deployment and monitoring tools with an unified management interface for each component.

Interceptors in the membrane may intercept calls on the business interfaces as well

as on the control interfaces in order to allow custom implementation of the interfaces being called. This is particularly important in the case of control interfaces, where the interceptors implement the majority of functionality offered by the control interface, relaying only little (if anything) to the component content. The interceptors together form a simple component architecture around the content. Thus we consider them as a special kind of platform components, which are however at a different level of abstraction compared to the application components. To emphasize the component nature of interceptors, we term them *micro-components* (this term is used interchangeably with *interceptor*) and distinguish provided and required interfaces on micro-component boundaries.

Addressing the polyglot programming for components (*i.e.*, the ability to develop components in dedicated languages), we essentially deal with two types of programming languages in our approach:

(i) There is a language native to the component container. We call this programming language *container language*.

(ii) By the introduction of polyglot programming, the component business functionality (embodied in the component content) may be provided in a language different to the container language. We call this programming language *content language*.

Referring back to the goals of the section, we primarily focus on Java as the container language and either on Java or on a scripting language as the content language.

3.5.3 Analysis and Solution Design

The problem of integration of scripting languages into existing component frameworks (by means of the polyglot programming) consists of two independent parts:

(i) integration of the container language with the content languages, and

(ii) integration with the component container.

The first part of the problem, the language-level integration, covers inherently two issues – data and control flow integration. With regard to polyglot support, the data flow integration comprises data type mapping, value conversion, integration of individual language concepts, access to objects in the other language, mapping of interfaces, etc. The control flow integration comprises integration of method conventions, handling of method return parameters (by value, by reference) and exception handling.

All these concerns are typically handled by an *adaptation framework*. For integration of scripting languages in Java-based containers, which we are primarily concerned with, such an adaptation framework is the Java Scripting API (JSA) [24]. It provides an unified interface for different scripting engines which are integrated in Java [19]. Each engine allows loading and executing code in its respective scripting language. Furthermore, it ensures data type mapping, access to Java classes from the scripting language, and their instantiation.

The second part of the problem of polyglot programming for components refers to the integration of the adaptation framework (*e.g.*, JSA) into the component system so that it can be used to execute the component code.

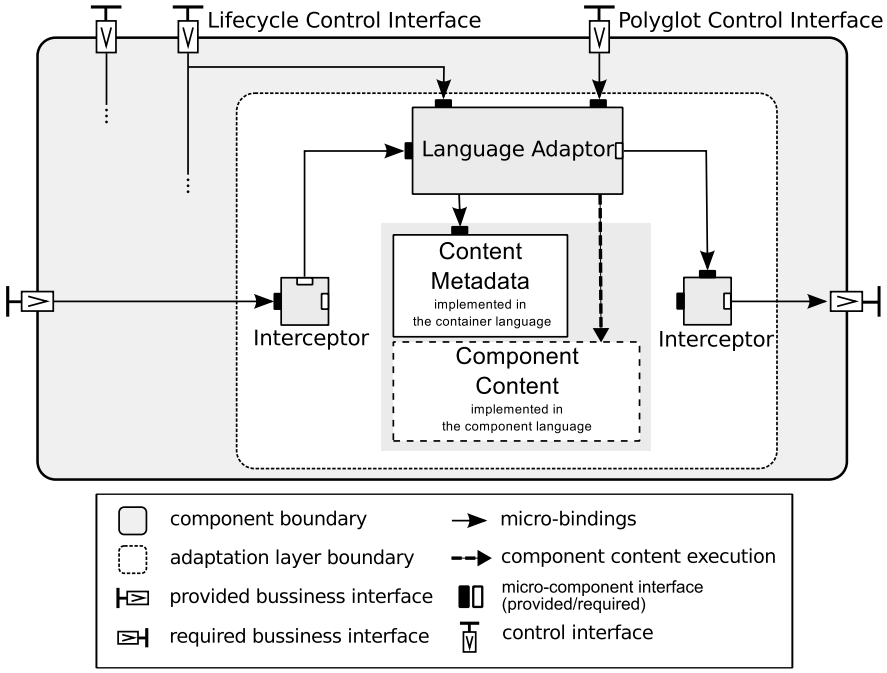


Figure 3.10: Basic idea.

Since, the first part of the problem has been already sufficiently solved by adaptation frameworks (*e.g.*, JSA in the case of Java), we focus on the second part of the problem only in the rest of the section.

3.5.4 Overall Design

Our approach introduces a specialized component runtime extension – further referred as the *polyglot extension* – which forms an adaptation layer between the component container and the component implementation in the content language. It creates a special environment inside the component membrane that allows executing code in the content language, which is different to the container language, using an adaptation framework. This environment is built using a set of membrane's controllers and interceptors and its general structure is shown in Figure 3.10. In particular, the *polyglot extension* ensures redirection of the control flow to the adaptation framework, and integration of this framework with the related component controllers (*e.g.*, life-cycle controller).

The component content, *i.e.*, the code in the content language, is not executable by the container directly. It is managed by the *language adaptor*, which loads the component content based on location provided by the *content metadata* and reroutes calls with the help of interceptors, which "surround" the content. The adaptor transforms calls into invocations of the adaptation framework that is part of the adaptor. The framework then executes the code in the content language and returns a result (or an exception), which is passed back via the interceptor. For required interfaces, the language adaptor similarly transforms and redirects calls from the component content to the corresponding interceptor. In the same manner, the polyglot extension allows

propagation of control interfaces calls to the component content (from the low-level view, a control interface is just another provided interface).

Both the caller and the callee of the component, as well as the rest of the component controllers, do not note any difference between a common component and the one with the polyglot extension (except for possible type incompatibilities of interface call arguments and exception handling, which should be resolved by the author of the prototype implementation).

In the rest of the section, we describe each of the polyglot extension elements in more detail.

Provided Interface Interceptors

The interceptor on a provided interface redirects calls to the language adaptor. To do this, the interceptor passes the method identification and values of arguments of the called method. During the method call return, it returns the return value or an exception.

Each interceptor has three interfaces. Two of them – one provided and one required – have the same type as the intercepted interface. Using these interfaces the interceptor is connected in the membrane to the chain of interceptors for the particular interface. In normal case, the interceptor receives the call on the provided interface, performs its functionality, and forwards call on the required interface. However in our case, the interceptor stops the call and redirects it, *i.e.*, the required interface is never used and it can be omitted if the component system supports it.

As the interceptor is based on the type of the intercepted interface, *i.e.*, it is unique for each interface type, its implementation has to be either reflection-based or dynamically generated at deployment or instantiation time.

The remaining interface of the interceptor is the required interface via which the method call is redirected to the language adaptor. The type of this interface is the same for all provided interface interceptors. Basically, it contains one method which has two arguments, one for the intercepted method name and one for the original argument array, and which returns a generic return value (*e.g.*, an `Object` reference). The interceptor implementation handles required type conversions.

Required Interface Interceptors

The required interface calls are (similarly to provided interfaces) redirected to the interceptor corresponding to the particular required interface.

As in the case of the provided interfaces interceptor, there are three interfaces on each required interface interceptor. Two of them – one provided and one required – are used for connecting the interceptor to the interceptors chain (again, the provided one in this case can be omitted if the component model allows it) and they have the type of the intercepted interface. The remaining interface has also the same type as the intercepted interface and is used for call redirections from the language adaptor.

Again, as the interceptor is unique for each interface type, its implementation has to be reflection-based or dynamically generated.

Control Interfaces

Certain controllers – such as the life-cycle management controller – may require access to the component content, *i.e.*, call methods on it. For example in the case of the life-cycle control interface, it has methods for notifying the content about starting/stopping the component and the content can react for example by starting/stopping its internal threads.

As from the low-level view, the control interfaces are the same as provided interfaces, the situation is almost the same as in the case of the business provided interfaces. The difference is that both the control interface interceptors and the language adaptor have to cope with these control interfaces in order to invoke the associated functions on the content (in case of the business interfaces, the calls are forwarded only based on their names and types of arguments).

Component Content

The presented approach allows achieving maximal transparency and simplicity of component implementation. The code implementing the particular component is represented by a regular source file containing purely code in the content language. The developer only provides this file and in the component description (*e.g.*, in an ADL description based on the particular component model) he/she defines the identification of the used content language.

For languages without interfaces and/or object support, the methods of the provided interfaces have to be implemented as global functions. The language adaptor can find the particular function using for example prescribed name convention.

The required interfaces are available via dependency injection which is managed by the language adaptor. For example for scripting languages, the required interfaces can be available as predefined variables, which are filled during the component instantiation by the language adaptor.

Content Metadata

Content metadata is a unit implemented in the container language. The role of content metadata is twofold:

1. it wraps component-specific information needed by the language adaptor, and
2. by itself forms a unit that is recognized by the component container as a component.

Thus, from the container point of view, the content metadata has the role of component content and in fact the content metadata is typically a class implemented in the container language providing all the interfaces prescribed by the particular component framework. From the language adaptor point of view, the content metadata holds information about the real component content, *i.e.*, at least the identification of the content language and location of the content. The language adaptor uses this information to properly configure the adaptation framework and to load the particular business code.

Moreover, the existence of separate content metadata allows keeping the language adaptor independent of a particular component implementation. Additionally, since

the content metadata in not directly involved in execution of the code in the content language (it only points to its location), the content metadata class is independent of the particular content language.

The content metadata may be generated automatically by an interceptor attached to component instantiation if the component container allows such an extension or by a dedicated tool during component development.

3.5.5 Evaluation

SOFA 2 Case–Study

To validate the presented approach, we have implemented it as an extension of the Java-based SOFA 2 runtime (in SOFA 2 terminology, such an extension is called a component aspect), further referred as the *script aspect*, allowing component implementation in scripting languages using JSA as the adaptation framework.

The aspect architecture is shown in Figure 3.11. The aspect consists of several micro-components implementing the language adaptor controller (`Call Transceiver`, `Script Invoker` and various `Proxy` micro-components). These micro-components are responsible for integration with other component aspects, as well as for implementation of the language adaptor logic. The aspect also comprises designated micro-components representing the interface interceptors (which are referred as *script interceptors* in scope of polyglot support). The content meta-data has a form of a generic component content created and initialized by the component container for all components implemented in programming languages other than Java.

The language adaptor is implemented as follows. The *Script Invoker* micro-component contains and manages the JSA scripting engine and transforms the incoming calls of the provided interfaces into the scripting engine invocations. The *Call Transceiver* micro-component manages all the interceptors of the associated component and redirects their notifications to the Script Invoker.

In SOFA 2 case, the interceptor implementation is dynamically generated using a designated interceptor generator (with help of the ASM [6] bytecode manipulation library).

For interaction with other component aspects (*e.g.*, accessing the component content or reacting to the life cycle changes of the component), the script aspect intercepts the `Component` and the `Lifecycle` control interfaces. The former allows tracking changes of the component content in case of dynamic updates. The latter allows tracking lifecycle changes of the component (and therefore notifying the script code).

The script aspect introduces a new `Script` control interface (see Listin 3.1) which allows manipulation of the encapsulated scripting engine and the associated component implementation. Moreover, the Script interface is remotely accessible via Java RMI.

Together with a designated tool which we had developed in order to access this interface from command line (and thus to enable changes of implementation of a running component), we have successfully enabled the rapid prototyping of component implementation using scripting languages in our Java-based component system.

The script aspect implementation can be found on the official SOFA 2 website[6].

[6] `http://sofa.ow2.org/extensions/index.html#dynamic`

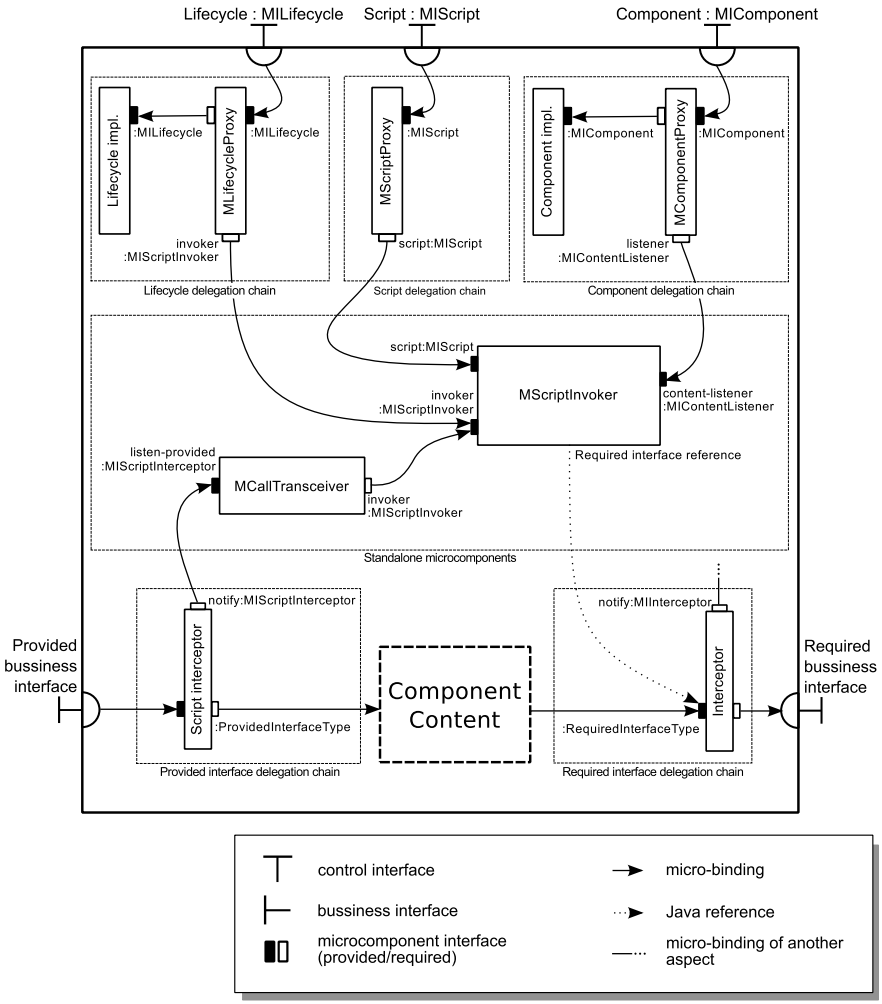


Figure 3.11: SOFA 2 Script aspect architecture.

Listing 3.1: Script control interface (Java).

```
1 public interface Script {
2     void evaluateScriptText(String text) throws SOFAException;
3     void reset() throws SOFAException;
4     void setValue(String name, Object value)
5         throws SOFAException;
6     Object getValue(String name) throws SOFAException;
7 }
```

Component Developer Experience

To demonstrate the achieved language integration, we present a prototype implementation of a component from the CoCoMe [HKW+08] application in the SOFA 2 system. We focus on the *ReportingLogic* component. Its main goal is to provide reports (using its provided interface, see Listing 3.2) which are computed from data in a shared data storage (accessed using required component interfaces). When this component

is being developed, its interfaces are already defined and thus it is possible to use the proposed polyglot support for rapid-prototyping of its implementation.

Listing 3.2: Reporting interface definition (Java)

```
1 public interface ReportingIf {
2   ReportTO getStockReportForStore(StoreTO sTO);
3   ReportTO getStockReportForEnterprise(EnterpriseTO eTO);
4   ReportTO getMeanTimeToDeliveryReport(EnterpriseTO eTO);
5 }
```

Listing 3.3 we present a fragment of the ReportingLogic component prototype implementation using Python which demonstrates the basic concepts of scripting language integration in SOFA 2. In scope of component implementation, it is possible to use Java classes using regular Python import statements (served by designated classloader). It is also possible to use additional Python modules and classes.

First we focus on provided interface implementation. In SOFA 2 case, the provided interfaces of scripted components have to be implemented as global script functions. This makes the implementation shorter than using Java (see `getStockReportForStore`).

Listing 3.3: Prototype component implementation (Python).

```
1  from java.lang import Thread
2  from org...inventory.application.reporting import *
3  # Downloading additional python modules using SOFA importing API
4  SOFAPythonImporter.loadCodeBundle(
5    'org ... application. reportinglib '
6    )
7  from org... application. reportinglib import HTMLReportPrinter
8  ...
9  def getStockReportForStore(storeTO):
10   result = ReportTO()
11   pctx = persistmanager.getPersistenceContext()
12   ...
13   store = storequery.queryStoreById(storeTO.getId(), pctx)S
14   store_name = store.getName()
15   items = storequery.queryAllStockItems(storeTO.getId(), pctx)
16   printer = HTMLReportPrinter()
17   ...
18   for si in stockitems:
19     printer.addRow(...)
20   ...
21   result.setReportText(printer.getText())
22   return result
23 ...
24 class PeriodicReportGenerator(Thread):
25   def run(self):
26     while not stop_reporting:
27       report = getStockReportForStore(find_store())
28       periodic_publisher.publish_report(report)
29       Thread.sleep(REPORT_PERIOD)
30 ...
31 def start():
32   thread = PeriodicReportGenerator()
33   thread.start()
```

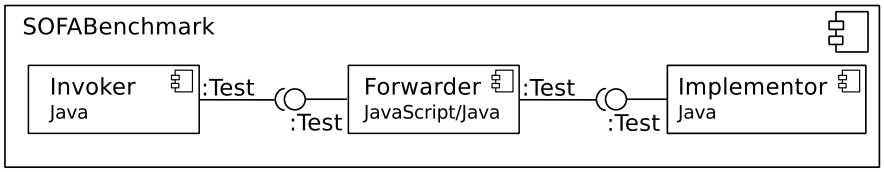


Figure 3.12: SOFA benchmark architecture

Next we focus on required interface access. The scripted component implementation uses automatic dependency injection where variables holding the provided interface references are created and assigned automatically during component initialization (using names in the associated component definition, see `storequery`, `persistmanager` and `periodic_publisher`).

To manage internal threads, a component content in SOFA 2 has to implement the `SOFALifecycle` interface. This interface provides automatic notifications when the component is started or stopped. This interface also has to be implemented using global script functions (see `start` function managing the `PeriodicReportGenerator` thread).

To summarize, the use of scripted components is transparent, without any significant adaptation code needed. It is also generally shorter than Java implementation, since the scripting languages are typically more expressive and there is no service code needed for typical tasks such as implementation of provided interfaces, acquisition of required interfaces, lifecycle management, etc. Therefore, the implemented polyglot support is suitable for the rapid prototyping task.

Performance Evaluation

The proposed polyglot support introduces a performance overhead caused by scripting controller infrastructure and by execution of the script code in the scripting engine. To asses this overhead, we have created a simple benchmark which uses a JavaScript component to forward interface calls between two Java components (see Figure 3.12). Because the script code is restricted to minimum, this benchmark minimizes the performance slowdown caused by slower execution of scripting language code. We have repetitively measured the execution time of 10000 forwarded interface calls (before that, we have used 5000 calls to eliminate initialization and JIT overhead). Then we have calculated the average execution time of one invocation for each of the 40 measurements.

We have compared the results with measurements of the same scenario which uses a standard Java component instead of the scripted one to forward the calls. For each measurement, we computed the difference between the corresponding average execution time of script forwarder and median of all Java-only measurements. The computed overhead, which is approximately 0.147 ms per one call, comprises both scripting engine overhead and the scripting extension infrastructure overhead. To get an overhead approximation of the scripting controller infrastructure only, we have created a simple regular Java application implementing the previous scenario (both Java-only and Java/JavaScript version) using simple objects in place of components. In this case, the measured overhead comprises only the scripting framework overhead. For

JSA using Mozilla Rhino version 1.6 release 2 it is approximately 0.059 ms per one call. By subtracting this value from the average component measurement we get the average overhead of the scripting controller infrastructure – approximately 0.088ms per one call. This is 40% of the default SOFA controller overhead. To summarize, a prototyped scripted component has 40% bigger overhead than regular Java components, which is still acceptable in the development environment. The measurements were performed on the following configuration: Intel(R) Core(TM)2 Duo P8600, 2.4GHz, 4GB DDR2 RAM, Windows 7 Professional OS 32bit, JVM 1.6.0_22-b04, SOFA 2 revision 1194.

3.5.6 Lessons Learned

The contribution of the case-study is twofold. It introduces a simple component model (called micro-components) for modeling execution environment extensions. The important property of the used component model is its simplicity in which sense it is similar to configuration frameworks as Spring [JHRS05] or Guice [12]. Furthermore, the model inherently expects that a content of micro-component can have two forms – a direct implementation or code generator producing desired implementation.

Moreover, the case-study also shows how an execution environment in a form of an advanced container can be extended in a systematic and non-invasive way. The presented approach is applicable not only for programming language integration but generally for integration of distinct technologies into the container (for example, integration of different component system implementations). This is an important property with regards to construction of tailored execution environments.

To summarize, characteristic features of the case-study are:

- **Utilization of a flat component model.** To preserve simplicity of the execution environment, there is no intention to model the extensions with help of a hierarchical component model. The flat model allows for expressing important execution aspects and it is easier
- **Explicit model of the execution environment.** The execution environment is explicitly modeled – it is specified by a composition of micro-component and bindings among them. Furthermore, the environment publishes locations where its infrastructure can be extended.
- **Role of interceptors.** The case-study heavily relies on call interception which is performed by dedicated micro-components in the execution infrastructure called interceptors. The interface interceptors are typically composed into a chain which corresponds to a component interface. The relation and order of interceptors inside the chain need to be well-defined to avoid unwanted malfunction.
- **Execution environment extension dependencies.** It is often necessary for a execution environment extension to use features of other already-existing extensions. This requires dealing with various extension dependencies.
- **Multiple content of micro-component.** The case-study demonstrates importance of having different forms of micro-component content to prepare the right execution infrastructure. In the context of the case-study, the micro-component

has a pre-defined implementation or its content is defined by a code generator preparing actual micro-component implementation. The combination permits to control reuse and adaptation of the infrastructure.

- **Component system is utilized in the scope of another component system.** As in the previous case-study, the execution environment of the component system is extended with help of another component system. The property makes both systems dependent. In this case-study, the dependency is evident mainly during the deployment, when the developed extension has to be injected into the existing execution environment.
- **Reflection-based versus generated interceptors.** To avoid unnecessary runtime overhead interceptors can be generated in advance. However, the generation process increases a complexity of system deployment. On the other hand, code reflection during runtime degrades execution performance. The right decision between these options is driven by a target domain and its constraints.

3.6 Summary

The presented analysis of contemporary component systems and case-studies bring several observations to be considered by the meta-component system.

Application domains. The analysis has identified four important application domains which heavily utilizes component systems.

(i) enterprise applications

(ii) user interfaces

(iii) configuration frameworks

(iv) embedded systems

Each domain has its characteristic properties which have impact on component system and its capabilities. As has been mentioned in Section 3.1.5, the most different part of the component system are its component model, execution environment, and support of non-functional requirements.

Execution environments. The analysis of component systems has also identified three different forms of execution environment. They differ in their complexity as well as in a deployment process which prepares component-based application to be launched and executed.

To summarize, there are three kinds of the execution environment:

EE1 Ad-hoc execution environment

EE2 Execution environment as a library

EE3 Container

Execution environment modeling and development. The case-studies have stressed the importance of execution environment modeling and rigorous development. The real-time connectors and SOFA 2 runtime extension case-studies have demonstrated benefits of utilization of a component system to develop and extend an execution environment infrastructure. In both cases a simple but well-defined component model has been employed and it has allowed for representing environment structure. The models have typically contained two kinds of components – regular components encapsulating predefined logic controlling the infrastructure and interceptors participating in calls among component interfaces. The regular components have been associated with an implementation, while the interceptors have been defined by a code generator. The both case-studies have shown the importance of generators which prepare pieces of the infrastructure as well as its implementation. On the other hand, the first case-study jPapaBench has demonstrated a demand for manual preparation and modification of the execution infrastructure.

Chapter 4

Towards Meta-component System

The chapter elaborates the idea of the meta-component system in more details. It states its structure and a process preparing a new tailored component system. From this perspective, the chapter is a realization of the goal G1. The text of this chapter is based on the following paper:

> [BHM09] Bureš T., Hnětynka P., Malohlava M.: *Using a product line for creating component systems*, In Proceedings of the 2009 ACM symposium of Applied Computing (SAC'09), Honolulu, Hawaii, USA, ACM, ISBN:978-1-60558-166-8, Mar 2009.

4.1 Introduction

This chapter aims at building a meta-component system, which shall serve as a software product line for creating customized component systems, each addressing a particular domain or a combination of them. Our intention is to adopt ideas of the product line engineering and generative software development (GSD) [Cza05b] while focusing on producing families of component systems for different target application domains rather then just for one domain. In more detail, the objectives are as follows:

1. To identify variation points which a meta-component system has to provide. The analysis presented in Chapter 3 designates a scope which will be covered by the proposed meta-component system as well as it specifies common vocabulary (in words of generative system development, the analysis constitutes a *problem space*). Based on results of the analysis component systems' characteristics are identified, which in turn lead to variation points that our envisioned meta-component system has to accommodate.

2. To demonstrate how the meta-component system is to be used throughout the whole development life-cycle and how it eases the component development.

3. To discuss how to build the meta-component system and how to achieve its variability. In this sub-goal we focus on the last two constituents of a component system, namely *deployment tools* and *execution environment and tools*. This

choice is motivated by the fact that in these two the biggest differences in current component systems can be found and as [LU07] states, the execution environment significantly influences component semantics. Additionally the variability of the *component model* part has been at least partially addressed by existing approaches [GMW97, DHT01, VG07, SSK+06, GV07, KTG+06, PRJ+03, RJB04, LWWC12]. In terms of GSD we want to specify basic building elements of *solution space* as well as transformation process from the problem space to the solution space.

4.1.1 Structure of the Chapter

The rest of the chapter is organized as follows. Section 4.2 describes design of the meta-component system together with its lifecycle, while Section 4.3 presents realization of the proposed system in the scope of an existing component system.

4.2 Meta-component System

As shown in Chapter 3, there is a broad range of component systems targeting different domains. Although these systems differ in many aspects, they share a lot of commonalities regardless of their particular domain. The most significant commonalities include:

- All component systems have the concept of an encapsulated component that communicates with other components only via designated interfaces.
- The communication between components is typically realized by a procedure call or a kind of messaging. The communication is local or it is realized by a kind of middleware.
- Components may be composed horizontally by connecting their interfaces or vertically by the parent component—subcomponent relationship.
- Components themselves require a kind of a *component container*, which is an entity providing them various services (such as naming, transactions, persistence, operating system API, etc.). The container is also responsible for managing the component lifecycle and managing connections among components.
- The container itself is controlled by a set of tools that allow configuring it, creating it, and destroying it. Also, there are often tools available for controlling components running in the container.
- The components are typically deployed according to a deployment plan that describes the allocation of components to the containers and concretizes the resources used by components. Tools for creating the deployment plan and for performing the actual deployment are typically utilized.

All the concepts and activities related to components are reified not only in the component model and the execution environment, but also in tools, which cover the whole development process and typically include an IDE for design and development, a repository for storing and reusing components, various deployment tools and tools

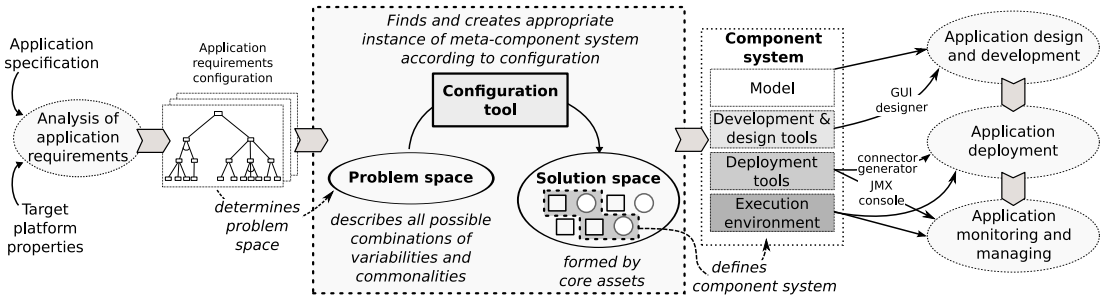


Figure 4.1: Usage scenario of the meta-component system from the point of view of application developer.

for managing components at runtime. In fact, it is the tool support where common component system concepts are evident even more than in the component model.

In our approach we propose taking advantage of the commonalities among component systems and build a meta-component system (*i.e.*, a kind of product line) for creating custom component systems. In this section we elaborate this idea in detail and show a typical use case of the meta-component system.

As an example, we consider development of an application for control and management of a manufacturing line. The application consists of two parts – an embedded system part controlling the move of the line and other actuators, and an enterprise system part providing an interface for retrieving the status of the line and history of the production and planning for new production. Development of such an application using the proposed meta-component system consists of the following steps:

1. **Gathering and analyzing the requirements of the application.** This step is the same as in the classical CBSE. In the case of the example it means identifying the need of transaction support, distribution, replication and monitoring in the enterprise part and support for real-time and emphasis on a low memory and CPU footprint in the embedded part.

2. **Configuring and generating the component system.** This step is new compared to the classical CBSE. The system architect specifies the requirements of the desired component system in a *configuration tool* of the meta-component system. The *configuration tool* uses common vocabulary specified by the analysis presented in Chapter 3 and permits selection and configuration of variation points. Based on a selected configuration the corresponding component system is generated/configured – that means the component model, execution environment(s) as well as tools and IDE for designing, developing, deploying components and managing components at runtime. The step of configuration and regeneration of the component system may be performed also later during the development process when new requirements arise. From the point of view of the generative programming [Cza05b], the *configuration tool* can be seen as a transformation of a problem space determined by the domain and application analysis to a solution space which contains basic building elements of component systems.

   In the case of the example it means specifying an application spanning the enterprise system and embedded system domains with the requirements as stated

in the previous step. The *configuration tool* evaluates the requirements and creates the component system consisting of two mutually interconnected component models (one for enterprise systems and one for embedded systems), an IDE, an execution environment (container) for the enterprise components and a synthesis tool for creation of the executable image out of embedded system components. The *configuration tool* omits unnecessary features – *e.g.*, it excludes support for distribution in the embedded part to decrease memory footprint.

3. **Designing and developing the application.** This step is similar to the classical CBSE. The component developer uses the provided IDE and the models to design the application by components and to implement the components. The whole process is simplified by the fact that the developer has available only the features requested when configuring the component system. When new requirements are discovered, the component system may be reconfigured to include new features.

   In the case of our example, it means designing and developing components of the enterprise part and the embedded part. The component system contains support for interconnecting these two parts. The enterprise part is developed in Java, using JTA for transactions, RMI for distribution, etc. The embedded part is developed in C. Existing domain specific tools are utilized (*e.g.*, simulator of an embedded device) – they are actually selected by the *configuration tool*.

4. **Application deployment.** This step is again similar to the classical CBSE, only the tools provided by the created component system are used. In many cases it means reusing existing tools, which were selected and pre-configured by the *configuration tool*.

   In the case of the example, the deployment comprises allocation of enterprise components to containers and generating connectors for their distribution. On the embedded side, the components are merged and a binary image is synthesized. The binary image contains the RTOS, basic API for the components and the actual components turned into tasks and processes of the RTOS. The image is uploaded to the target device using tools provided by the created component system. The communication between the enterprise part and the embedded part of the application happens through CAN-bus abstracted by special generated connectors.

5. **Monitoring and managing the application at runtime.** This step is again similar to the classical CBSE. Only the management and monitoring is performed using the tools provided by the generated component system. In the case of the example it means using a JMX console to access the enterprise components, which are automatically exposed by the pre-configured container via JMX. The tooling also contains support for starting, stopping and updating components.

In summary, the proposed meta-component system inserts the step 2 (configuring and generating the component system) into the classical CBSE process. The other steps remain the same or similar, only they utilize the component system (*i.e.*, models, tools and execution environment) configured and generated at the step 2 (see Figure 4.1). An important feature is also the ability of reconfiguring the component system at any

stage of the development lifecycle, when new requirements are discovered or some existing requirements are evaluated as not needed.

The benefits of using the meta-component system lie in reducing the duplication of functionality and providing a common approach to different component features and application domains. Another strong benefit is the automated evaluation of application's requirements by the *configuration tool*, which guides the choice of used technologies by ruling out those which do not satisfy the requirements. The ability of customizing the component system also reduces the footprint of the unused features, which would be present in the component system otherwise. Finally, the fact that the generated component system provides support only for the required features makes design and development easier and more straightforward as the developer has available only the features required for building the application.

The production of customized component systems is also very interesting for cross-domain applications (as shown in the example). In this respect, the meta-component system instantiates an interrelated family of component systems (*e.g.*, having one model for enterprise systems and one model for embedded systems). The tooling provided in this case supports the whole family, thus making possible to design, develop and deploy components in the same way (preferably through a single IDE).

4.3 Realization of Deployment and Execution Environment

The previous section analyzes the commonalities of component systems, proposes the meta-component system and demonstrates its utilization. Although the many commonalities, the meta-component system has to accommodate also the differences that stem from different domain requirements. In this section we focus on the differences and show how they can be addressed in deployment tools and the execution environment. When inspecting them closer, it is possible to identify the following areas of variations.

- *Component semantics* – it is defined by the component model and comprises decisions such as whether components are flat or hierarchical, what are the possible connections and if they can be distributed, whether shared components are permitted, what is the component lifecycle, etc.

- *Target platform* – it influences the choice of underlying technologies used by the execution environment and it defines activities to be performed in the deployment – *e.g.*, generation of connectors, transforming components of an embedded application to RTOS tasks and processes.

- *Services required by components* – they comprise the services required for a proper functioning of the components, such as a transaction support, a persistence, a database access, etc.

Thus, with respect to the target domain and specified requirements, the execution environment has to support different functionality and services. The deployment is also influenced by the domain and requirements as it is totally dependent on the execution environment (*e.g.*, deployment of enterprise applications consists of dynamic

uploading of code to a container, while in embedded applications the container's functionality must be merged with the application).

In our proposed approach, we achieve the necessary level of variability of the deployment process and execution environment by composing them from components. We demonstrate this on the SOFA 2 component system and in the rest of this section we show how SOFA's deployment and runtime environment can be extended to become fully configurable.

SOFA 2 [BHP06] (shortly SOFA in the following text) is a hierarchical component system. In addition to component model, SOFA comes with a set of deployment tools and an elaborate runtime environment, which is responsible for component instantiation, life-cycle management, distribution of a using software connectors, dynamic update, etc. SOFA contains two concepts which are very important for the variability and configurability of the deployment process and execution environment. Although they do not directly implement all the envisioned variability yet, they serve as the proof of the concept and represent a viable way of achieving the variability. These two concepts are the *connector generator* and *microcomponents*.

The connector generator [Bur06] is responsible for creating connectors, which are entities realizing the communication among components. The connector generator takes on the input a declarative description of communication requirements – component interfaces, a communication style and required non-functional properties (*e.g.*, security, monitoring); and based on these it creates the requested connector.

Internally, the connector generator contains two main components – an *architecture resolver* and a *code generator*. The architecture resolver uses constraint solving techniques (based on Prolog) to derive a connector architecture that satisfies specified requirements. The connector architecture is built out of hierarchical connector components (so called *connector elements*). In addition to composing and connecting the connector elements, the architecture resolver also parametrizes each element in the architecture (*e.g.*, adapting an interface, setting attributes). The code generator follows the resolved architecture and builds the connector implementation. It performs source code level adaptations of the connector elements to reflect element parametrization, then it compiles the connector elements and assembles them to form the connector.

From the point of view of the meta-component system, the connector generator represents a skeleton of the configurable deployment tool. It is itself modular – the knowledge about the particular deployment and runtime environment is introduced in the form of available connector elements and actions, which are plug-ins that take care of source code adaptations, compilation, generating middleware stub and skeletons, packaging, etc. Thus by extending the connector generator and configuring its parts, it is possible to achieve simple packaging of components (*e.g.*, in configuration frameworks) as well as elaborate transformation of components to RTOS concepts (*e.g.*, in embedded systems). The Prolog-based constraint solver in the connector generator additionally provides a ground for planning the transformation and optimizations.

Microcomponents [MB05] are used to provide runtime variability with respect to component semantics and services. The idea behind microcomponents is to divide an application component into the *control part* and the *content*. While the content is the business code written by the application developer, the control part is provided by the component runtime. In fact the control part wraps the *content* and exhibits component interfaces. In addition to business interfaces, the *control part* also implements

control interfaces (*e.g.*, lifecycle interface, lookup interface) that allow the execution environment to manage the component.

The innovative approach of SOFA is that the control part of a component is modeled using the *microcomponents*, which are flat local components with a simple lifecycle. Each microcomponent is responsible for a specific control functionality (*e.g.*, looking up an interface, intercepting calls, or blocking calls to a business interface when the component is stopped).

Since a particular component feature (*e.g.*, component lifecycle) is realized by a number of microcomponents (*e.g.*, one implementing the start/stop method, others attached to business interfaces to block incoming calls when the state is "stopped"), the microcomponents are grouped to *aspects*. Each aspect defines a consistent extension of the control part introducing a particular feature. Thus, the runtime component semantics and services are configured by a selection of aspects to be applied to the application components (or to a subset of them).

Microcomponents share a lot of commonalities with connector elements, which makes natural and advantageous to unify them in one common concept that may be used both during deployment and runtime. Thus, coming out from the two concepts of the connector generator and microcomponents, we propose the configurable deployment process and execution environment to be based on two main constituents – a *synthetizer* and a *microcomponent system* (see Figure 4.2).

The synthetizer is a part of deployment tools responsible for transforming business components to runtime artifacts. It follows the concept of SOFA connector generator and allows adapting, building and assembling microcomponents (see Figure 4.2), which represent various parts of the execution environment, connector functionality, and business code of application components. The output of the synthetizer is either a microcomponent architecture (together with microcomponents) that is to be instantiated, or it is a binary image to be uploaded to an embedded device and executed there. In the former case, the microcomponent architecture produced by the synthetizer contains only the business code of application components merged with control parts and connectors. These microcomponents are to be deployed in the execution environment, which consists of a minimalistic core of the microcomponent system and preconfigured microcomponents providing remaining services (*e.g.*, naming, transactions, remote interface to component deployment). In the latter case the produced binary images contain not only microcomponents for the business code, control parts and connectors, but also the microcomponents implementing the services of the execution environment.

With regard to the current status of SOFA, we have to enhance the connector generator to use the unified concept of connector elements and microcomponents and to handle whole application components, not just connectors. Further, we have to advance in componentizing the deployment container with the execution environment, *i.e.*, the container in which components are executed. Currently, SOFA provides only a single type of container, suitable for rather large-scale applications. To follow the meta-component system concepts, the container needs to be reimplemented also as a set of microcomponents that can be configured together to provide a required functionality. Finally, with all infrastructure prepared, it is necessary to provide a *configuration tool*, which allows the configuration of the synthetizer and the execution environment.

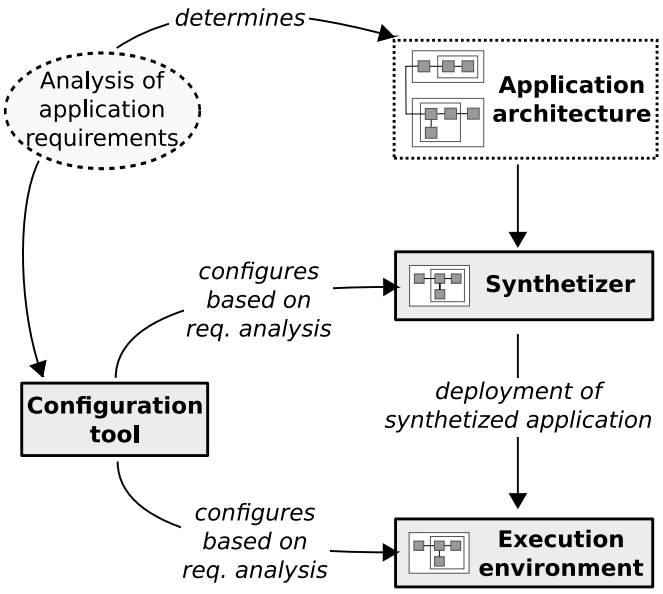


Figure 4.2: Deployment and execution environment.

4.4 Conclusion

In this chapter we have presented an approach of instantiating component systems for different domains using a common meta-component system. We have shown the usage of the meta-component system, which basically follows the classical CBSE process, only inserts a component system configuration step at the beginning (after collecting requirements). Based on identifying typical application domains of component systems and discussion of their characteristics, we have proposed how to achieve the configurability of the deployment process and the execution environment.

Chapter 5

μSOFA – Model-driven Method for Creating Configurable Execution Environment

Since we consider the execution environment as a crucial part of every component system, it naturally plays also a key role in the meta-component system. Chapter 4 has introduced an approach of synthesizing an execution environment and corresponding deployment tools. In this chapter, the approach is elaborated in more details resulting in a model-driven method called μSOFA preparing a configurable execution environment (achieving goal G2).

5.1 Motivation

One of the meta-component system goals is to prepare a tailored execution environment which would support execution of a component-based application described in a constructed component model and which would be able to satisfy its non-functional requirements and utilized implementation technology. Therefore, our main objective in this chapter is to have the execution infrastructure and its preparation well-defined with explicitly defined variability. That includes (i) an execution infrastructure model which would express all necessary functional and non-functional concepts, and (ii) a rigorously defined transformation from an input component assembly (*i.e.*, a collection of application components that together form a component application) to the infrastructure model and further to its realization. Moreover, motivated by reflective middleware, we would like also stress a demand on infrastructure encapsulation and independence – the infrastructure model should include all aspects which are necessary to translate it into its realization. And finally, the execution infrastructure as well as the corresponding transformation process should not be constrained by any particular technology.

To elaborate our motivation in more details, it is also necessary to discuss variation points of component systems as identified in Chapter 3 – (i) component model, (ii) supported non-functional requirements (including various runtime services), and (iii) a form of execution environment. Since the (i) component model variability can be

solved by existing approaches, Section 3.1.5 has shown how it can be achieved. Nevertheless, the component model has to be also reflected by an execution infrastructure, because the component assembly of an application (which is described in term of the component model) serves as an input for the realization of the whole system (as it will be seen later in this chapter).

Non-functional requirements (NFRs) can impact any part of the resulting application – architecture, execution environment, implementation, configuration of hardware infrastructure, usability of application, etc. In the context of this chapter, the most interesting are NFRs which have impact on the execution environment infrastructure (*e.g.*, reliability, monitoring, persistence support, life-cycle). However, it is impossible to enumerate beforehand all possible NFRs which need to be considered. Therefore, we would like to have the execution infrastructure highly configurable to be able to capture the variety of non-functional requirements.

Moreover, according to Chapter 3 the execution environment has one of three forms (iii). The forms differs in their complexity and in the corresponding process which prepares them. Each form is constructed by a process transforming a component assembly and specified NFRs into the form which can be then launched. Hence, such process has to understand the inputs which can vary according to utilized component model and constructed application. Furthermore, the process has to know how to refine the inputs into an execution infrastructure and further into an executable form built on the top of an implementation technology.

And finally, every execution environment is built on the top of an implementation technology which also directly influences environment capabilities. The technology can be a programming language or also another component system or configuration framework (*e.g.*, Google Guice [12], iPOJO [EHL07]).

To summarize our motivation, we would like to have a method which would allow for defining the execution infrastructure of component-based applications and corresponding process transforming it into one of the identified forms. The method should be able to reflect variability in:

- the component model and particularly in the component assembly which stands for method input
- non-functional requirements
- identified forms of execution environments
- target implementation technology

5.2 Outline of the Solution

This section presents our major ideas to achieve a proper solution. The input of the method is a component assembly representing an application architecture, specified non-functional requirements with a description of their impact on the execution infrastructure. Additionally, to prepare a target executable form of execution environment, a particular implementation technology is specified.

The result of the method depends on the specified implementation technology and application's NFRs and it is one of identified execution environment forms – *i.e.*, the method outputs a binary which can be directly executed, or a package, configuration

which can be loaded into a component container. The method thus combines creation (or configuration) of the execution environment with packaging of the component application to be executed within the runtime. This allows us to treat uniformly the three principal kinds of the execution environment.

During the process of the infrastructure preparation, interpretations of individual NFRs have to be injected into an infrastructure of the execution environment. However, the NFRs can enforce a preparation of new artifacts (*e.g.*, interface interceptors) or code generation (*e.g.*, interceptors implementation, software connectors).

To denote infrastructure concepts including application components, their execution infrastructure, and execution infrastructure of underlying component platform, we use a dedicated *execution infrastructure model* (EIM) based on micro-components [MB05]. To avoid preserving and maintaining two models (one for describing application components and another capturing the architecture of the execution infrastructure) during execution environment modeling, we use the micro-components for decomposition of both the execution infrastructure and the application components. This is however done without loosing traceability to concepts of the application component model (*i.e.*, application component's interfaces, bindings). To clarify the utilization of micro-components, we refer to Section 3.5 which has shown the micro-components as well-defined and appropriate concepts to model execution environment artifacts (interface interceptors, infrastructure managers).

Based on this fundamental idea, it is possible to define individual high-level stages of the μSOFA method, their inputs, and outputs (Figure 5.1). The first stage (called *front-end*) refines the given assembly of application components with the help of specified NFRs. The result of the stage is the execution infrastructure model. The second stage (called *back-end*) transforms the model into an implementation. Moreover, the first stage can demand code generation which need to be executed in the scope of the second stage. it means the front-end configures the back-end.

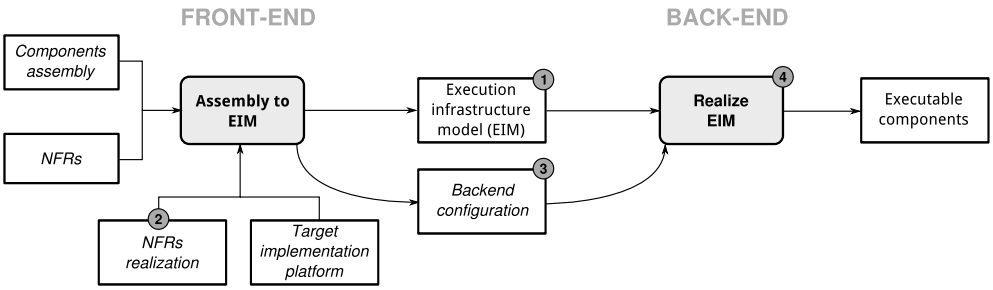


Figure 5.1: The outline of the μSOFA method.

Figure 5.1 outlines the solution. To detail the μSOFA method it is necessary to clarify:

1. Execution infrastructure model (EIM) based on the concept of micro-components.
2. Representation of NFRs and their realization in the execution infrastructure.
3. Impact of NFRs on the generation back-end.
4. The front-end process of refining the component assembly into an execution infrastructure model.

5. The back-end process of translating the execution infrastructure model into realization based on a selected technology.

5.2.1 Illustrative Example

To clarify the outlined μSOFA method an illustrative example of a component-based application is presented. The example demonstrates a simple watch control system employing two components. The objective is to demonstrate an application component-based architecture, its underlying execution environment infrastructure, and the process of its realization with help of a selected target technology.

Functional requirements Figure 5.2 shows an application assembly composed of three components. The top-level composite component `Watch` includes two primitive sub-components `Ticker` and `Display`. The `Ticker` is an active component which generates periodic ticks and calls the `Display` interface. The call on the interface causes the `Display` component to update its time counter. The component `Display` is responsible for visualization the current time by serving a web page with the current time. The implementation of the component provides a J2EE servlet which needs to be registered by a HTTP service. In this case, the behavior of both the primitive components is not specified rigorously by any formalism due to their simplicity, but both of them are implemented in the Java language as POJOs.[7]

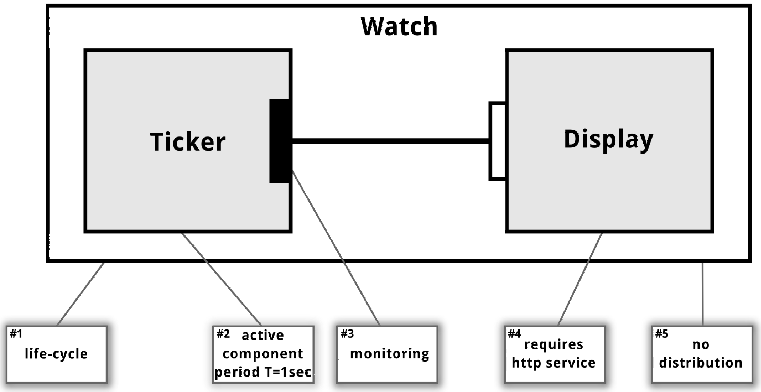


Figure 5.2: The illustrative watch example.

Non-functional Requirements To provide a meaningful architecture, non-functional requirements, their meaning, and impact on execution infrastructure have to be specified.

The first non-functional requirement is stated on the top-level component and it expresses that application life-cycle management has to be enabled. The second requirement marks the component `Ticker` as active with periodic execution enabled every second. The third requirement demands monitoring on the `Ticker` required interface. The fourth NFR expresses a required service – the component `Display` requires an HTTP service for correct execution – via the service the `Display` component registers its provisioned servlet showing actual time.

[7] POJO – Plain Old Java Object

Furthermore, for system execution it is also necessary to specify distribution of components.[8] In the case of the example, only a local scenario is considered and it is expressed by the fifth non-functional requirement.

5.2.2 Front-end: From Component Assembly to Execution Infrastructure

Independently of a particular component implementation technology, it is possible to identify an infrastructure of the example's execution environment. Figure 5.3 shows the overall system execution infrastructure including components' and platform's infrastructures. In this case, the overall infrastructure is composed of micro-components and bindings among them representing communication channels. The micro-components allow for expressing application components as well as execution infrastructure in the same way.

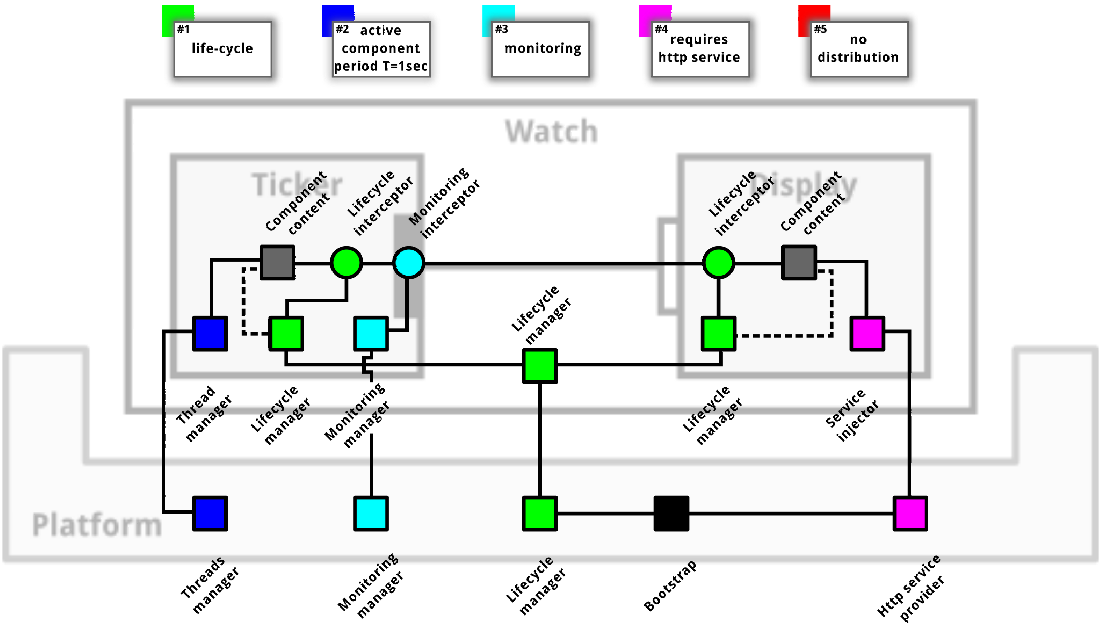


Figure 5.3: Watch example execution infrastructure.

First of all, we need to clarify a relation between an application component and its representation in EIM. Regarding a primitive component, its implementation (*i.e.*, content) has to be encapsulated in the infrastructure ensuring its correct instantiation and execution. The component content is represented as a micro-component, since it participates in system execution. In the case of a composite component, which has no implementation, the execution infrastructure has to adopt a strategy to represent them. Either the hierarchical structure is "flattenized" (as in the case of the example). Or the execution environment can preserve hierarchical components with help of dedicated micro-components storing a notion of component hierarchy – for example, to enable runtime introspection.

[8] For simplicity, a specification of an application deployment plan is omitted here and it is stated in a form of non-functional requirement. However, in a rigorous development cycle, it should be specified and it should describe component distribution and corresponding properties.

Non-functional requirements inject into a component infrastructure new micro-components and bindings among them. The example's NFRs require life-cycle management. It is defined on the top-level component, however, if the top-level component have to be able to manage its life-cycle, it also has to manage life-cycle of its sub-components. Therefore, each sub-component has its own *life-cycle manager*, which informs the component implementation of pending life-cycle changes (if the implementation requires being informed). Furthermore, to control the life-cycle, the calls on the interfaces have to be blocked till the whole system is started. Hence, a concept of an *interface interceptor* managing calls on interfaces is introduced. The interceptors are controlled by the corresponding life-cycle manager.

The `Ticker` component is declared as an active component. Hence, it needs a periodic thread which will periodically invoke the component implementation according to the given period. The provision of a thread depends on `Ticker` implementation – it can be hidden in the component implementation, or the component's infrastructure can provide it. Regarding separation of concerns, in this case we prefer the solution that the component delegates the decision of using thread to its infrastructure which can manage directly thread properties or thread reuse. Therefore, the *thread manager* is incorporated in the infrastructure of the `Ticker` component.

Respecting the call monitoring requirement, the `Ticker` infrastructure contains a *monitoring interceptor* on the interface which will intercept all calls and send call information to the *monitoring manager*. The order of interface interceptors is important here (as has been also mentioned in Section 3.5). Respecting the method call semantics, the monitoring interceptor has to be put after the life-cycle interceptor in the interface interceptor chain to ensure that it monitors only calls which are invoked only during during system execution (not during the initialization of the system).

Finally, the component `Display` has to obtain a reference on an HTTP service. This reference has to be provided by a component infrastructure and needs to be injected into the `Display` implementation. There are two options how to achieve such injection – (i) the component implementation explicitly supports the injection by marking the location of injection by implementing a predefined interface, marking a method or attribute via an annotation; or (ii) the component implicitly expects the desired service – in this case aspect-oriented programming (AOP) [KLM+97] can be for example utilized to weave an aspect containing the desired reference (in this case, aspect still needs to know at least the structure of the code). Furthermore, the injection of a service can be static – *i.e.*, during compile time of the infrastructure implementation, or dynamic which ensures that the service is injected during system instantiation or execution. Hence, the *service injector* encapsulating a selected service injection strategy needs to be placed into the component infrastructure.

The introduced micro-components in the components' infrastructures can require platform services (*e.g.*, HTTP service) or they may need to be managed by a platform (*e.g.*, life-cycle of component is directly controlled by underlying platform life-cycle). That in fact means the injection of a micro-component into the component's infrastructure can have a direct impact on a structure of the underling platform. To clarify the platform infrastructure it is necessary to explain the elementary platform services. The platform typically contains a concept of an entry-point (or bootstrap) – code which enables platform initialization. The entry-point can cause a start of an application – hence, it cooperates with a platform life-cycle manager controlling life-cycle managers of application's components.

Regarding component requirements, the active component `Ticker` introduces a dependency in the platform infrastructure in a form of a *thread manager* (*e.g.*, thread pool) controlling threads assignment to components' thread managers. A similar manager can be also required for call monitoring – for example, in a form of a global message sink collecting output of all application monitors.

And finally, the `Display`'s required service has to be provisioned by the underlying infrastructure. That means in the case of the HTTP service, it has to employ a library containing an HTTP server which publishes the desired HTTP service. Furthermore, the HTTP server has to be started during initialization of the platform which is typically ensured by a bootstrap implementation.

Since the example does not introduce distribution of components, only local bindings among component interfaces are considered. However, in the case of distribution, additional micro-components composing software connectors would be necessary to incorporate into the infrastructure model ([Bur06], [BMH08]).

It is important to mention that the example shows only a limited set of possible non-functional requirements and infrastructure properties. Nevertheless, the presented example serves to give an idea of the infrastructure refinement.

5.2.3 Back-end: From Execution Infrastructure to Its Realization

Since the execution infrastructure model contains all necessary information, it can be transformed into its execution form. In the context of the example, let us consider that *Google Guice* [12] is utilized as the target implementation platform. Guice is a simple Java-based inversion of control library which automatically manages references and their injection respecting annotated Java sources. Therefore, the execution environment realization will adopt the identified EE2 form – *i.e.*, execution environment is provided as a library.

To transform the example's execution infrastructure model depicted on Figure 5.3, it is necessary to associate micro-components with their implementation – *i.e.*, to decide about micro-component implementations reflecting the selected implementation technology. There are three kinds of micro-components in the example's infrastructure – (i) micro-components which have already an associated implementation which fits to Guice rules (for simplicity all managers); (ii) micro-components which have already an associated implementation, but it needs adaptation to Guice (*e.g.*, content of functional components); and (c) micro-components whose implementations have to be generated reflecting the actual infrastructure (interceptors, platform bootstrap).

Furthermore, in case of the example, Guice modules configuring inversion of control have to be generated. The example can adopt several strategies according to the granularity of Guice modules – it can generate only one Guice module configuring the whole system (including the application and platform) or multiple modules configuring depicted components and platform infrastructures. The example selects the second option and generates five Guice modules – modules for `Ticker` and `Display` components, module for the `Watch` application, module for underling platform, and overall system module gluing the other modules together.

Since the Guice is only a static library it does not introduce any concept of entry-point. Hence, the bootstrap code needs to be prepared as well. It configures the Guice internals and instantiates the generated system module. And finally, code compilation and bundling is performed to deliver a binary fully realizing the system.

Figure 5.4 shows the overall generation process realizing EIM. It depicts individual stages including code generation, adaptation, compilation, and final bundling. Each stage has associated a set of actions which are introduced by NFRs (code generation of interceptors, bootstrap) or the selected implementation technology (generation of Guice modules). Based on the example, it is necessary to stress that NFRs can modify the generation process by introducing new activities.

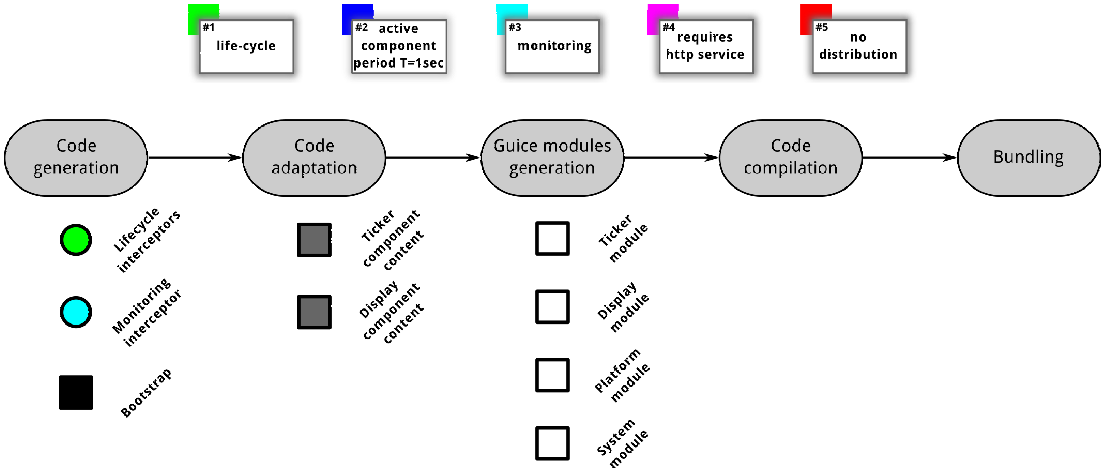


Figure 5.4: Schema of generation process for watch example execution infrastructure.

5.2.4 Lessons Learned

Based on the presented example, we can refine our understanding of the solution and state essential requirements regarding individual method's parts stated above.

Execution Infrastructure Model

The execution infrastructure model is a corner-stone of the method. The model has to allow for modeling all runtime concepts including representation of functional components, NFRs, and underlying component platform. We put stress on its expressiveness to describe clearly and in unique way the execution infrastructure which will be also beneficial for code generation. Therefore, the execution infrastructure model has to meet the following properties:

- the model should allow for separation of concerns;
- the micro-components should be reusable and well-defined. They should express their requirements, provisions, and realization. They should also represent a minimal set which is required to be modeled;
- the micro-components should be able to participate in infrastructure implementation directly (by providing an implementation) or indirectly (by providing a code generator, code or model transformation);
- the dependencies among micro-components' provisions and requirements should be explicitly expressed to provide a complete infrastructure to the back-end realization process;

- the model should not support distribution – the distribution should be modeled as non-functional property;
- the model should allow for expressing NFRs and platform services.

Non-functional Requirements Representation

The example has shown that a non-functional requirement can affect both component and component platform infrastructure by injecting at a specified location new micro-components and bindings among them. Hence, our demands on non-functional requirements representation are:

- NFRs representation should be well-defined – *i.e.*, it should have a unique interpretation in the context of the execution infrastructure;
- NFRs representation should adopt the same structure entities (*i.e.*, micro-components) as the execution infrastructure model to allow for easy interpretation of NFRs inside the execution infrastructure model;
- representation of NFRs should have an associated location in the execution infrastructure (and corresponding realization process) which it targets;
- representation of NFRS should allow for configuration of back-end realization process.

Transformation from Application Assembly to System Realization

The transformation process is a crucial part which creates an execution infrastructure by understanding component-based architecture and its non-functional requirements and further translates it into a realization based on selected implementation technology. Therefore, the overall transformation has to fulfill the following objectives:

- The transformation process should allow for refinement of component assembly into an execution infrastructure model by interpreting NFRs.
- The transformation process should understand NFR representation and know how to inject it into the execution infrastructure model.
- The transformation process should allow for refinement of the execution infrastructure model into a realization based on a selected implementation technology. The process should be technology agnostic – *i.e.*, it should not be directly connected to any technology.
- The refinement of the execution infrastructure into its realization should be configurable – NFRs representation should be able to reconfigure it.
- The overall transformation workflow should be well-defined with known structure and its "extension points".

5.3 Execution Environment Model

This section elaborates the stated requirements and proposes a suitable model for constructing execution infrastructure of component-based systems.

5.3.1 Micro-components

The model follows and adapts the micro-components concepts proposed in [MB05] since it provides suitable expressiveness to describe execution infrastructure and fulfill requirements stated in Section 5.2.4.

The micro-component as the key reusable entity is represented by the `UComponent` which has defined provisions and requirements via `UInterfaceable` (depicted on Figure 5.5). The `UComponent` has associated content which can have two forms – a generator or a particular implementation. Both kinds distinguish between source and binary code to permit deployment process to employ code post-processing (*e.g.*, optimization, compilation). The micro-component `UComponent` represents a reusable entity, hence, to reference its instance the `UComponentInstance` has to be utilized.

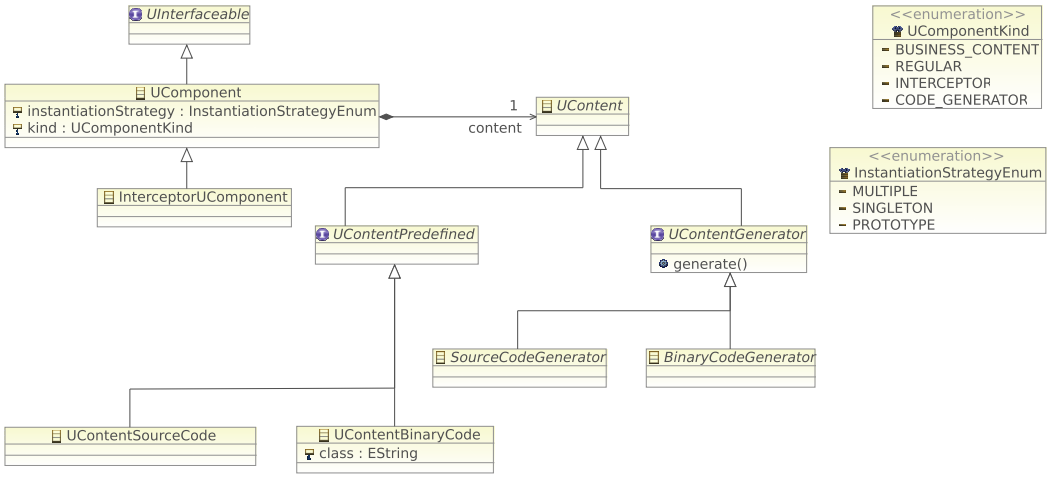


Figure 5.5: UComponent definition.

The requirements and provisions of `UComponent` are expressed via micro-interfaces `UInterface` (see Figure 5.6). Each micro-interface has its role (provided or required) and type `UInterfaceType`. Relations among micro-components are modeled as bindings `UBinding` which connect end-points `UBindingEndPoint`. The end-point can have different forms – *e.g.*, it can reference micro-component interface (`UBindingInterfaceEndPoint`), or it can employ *select* which finds target end point micro-component (`UBindingEndPointDeclarative`).

However, to model an execution infrastructure, additional transient concepts are needed (in the case of example, they are used to generate Guice configuration modules). These concepts should express high-level execution environment entities and as a result simplify refinement of component assembly. They only delineate well-defined areas in the execution infrastructure which are helpful during non-functional properties injection and code generation. Based on the motivation example presented in Section 5.2.1, it is possible to identify the following high-level entities:

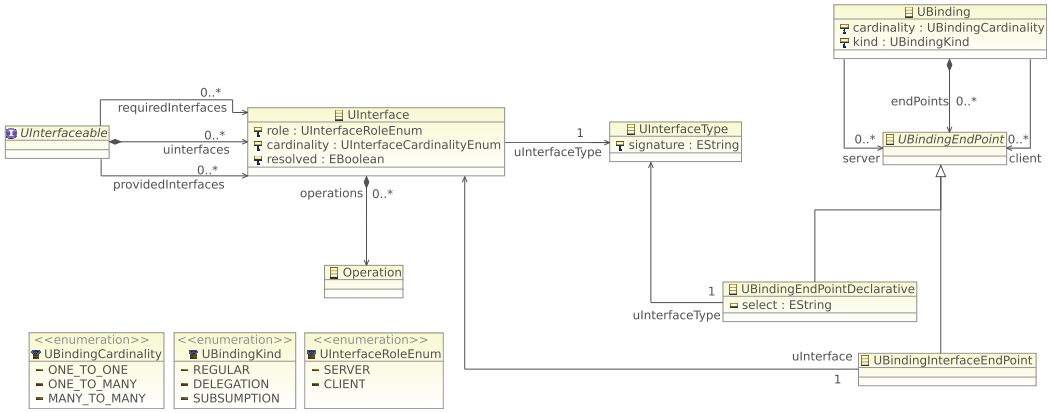


Figure 5.6: UComponent provisions and requirements.

- **Component instance** – represents an instantiated component including instantiated user ("business") code and also corresponding execution infrastructure supporting the code;

- **Application** – stands for a component assembly deployed into a container. The application does not involve only the assembled component instances, but also a corresponding part of runtime infrastructure encapsulating the assembly (*e.g.*, interceptors, controllers);

- **Execution platform** – represents a runtime environment providing services and corresponding support to ensure execution of deployed applications (*e.g.*, thread pool, transactions manager, timers manager, OS API);

- **System** – an abstract concept representing a set of applications deployed and executed on the top of a specified execution platform.

From modeling perspective, these high-level entities are modeled as micro-assemblies (`UAssembly`) of micro-component instances `UComponentInstance` and bindings among them (`UBinding`). These micro-assemblies allow for separation of execution environment concerns – component, application, platform, and system. A component instance is represented by `ComponentAssembly` and stands for the component execution infrastructure. Component instances are assembled into `UApplicationAssembly`. The assembly can, except for component instances, also contain micro-component instances and bindings among them. Such infrastructure standing outside component instances is useful for shared functionality extracted from selected component instances. Finally, the system modeled as `SystemAssembly` puts together multiple applications and a platform (expressed with `PlatformAssembly`).

To denote a required and provided services the assembly `UAssembly` specializes a concept of `UInterfaceable` which corresponds to a common base of all provided (resp. required) services. The service is bound to a micro-interfaces of an encapsulated micro-component instance. Thus, for example, `SystemAssembly` can bind application required services to platform provisions.

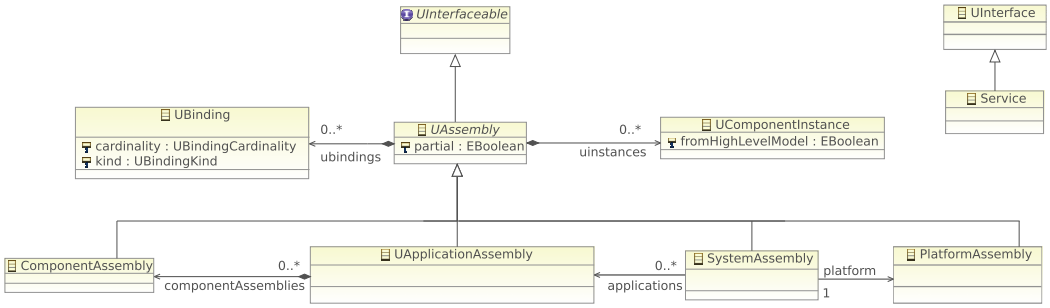


Figure 5.7: Execution infrastructure micro-assemblies.

5.4 Non-functional Requirements

General non-functional requirements with their vague semantics are hard to express in a generic way. Owing to their broad variance, it is impossible to capture all their difference kinds. However, in case of the execution environment, it is necessary at least to describe the requirements which affect the execution infrastructure – *i.e.*, according to Section 3.1.5 only NFRs managed endogenously (only system wide) and exogenously (per collaboration and system wide) have to be considered.

With respect to the stated requirements, a suitable representation of a non-functional requirement is a kind of *infrastructure aspect*. Its motivation comes from a domain of aspect-oriented programming (AOP) [KLM+97] and from the component configuration approaches shown in Section 2.1. Therefore, as a regular aspect the infrastructure aspect would be also composed of an *advice* and *point-cut*. The advice would specify an infrastructure change – which micro-components are injected and how they are bound with the existing context. The point-cut is able to identify a unique location in the infrastructure where the aspect advice should be weaved. To permit point-cut to find a desired location, crucial variation points in the infrastructure need to be determined – they are called *join-points* in the AOP vocabulary.

The concept of aspects perfectly suits into infrastructure building as has been already shown by the case-study presented in Section 3.5. Moreover, the concept have been already applied in existing projects [LRS+11] to express domain-specific concerns or as component aspects [MB05].

5.4.1 Infrastructure Aspects

A definition of the infrastructure aspect (`Aspect` – see Figure 5.8) is divided into an *advice* (`VariationPointAdvice`) and *select* (`VariationPointSelect`).

The advice shares the AOP principle and it is represented as a composition of actions which manipulates the infrastructure including adding new micro-components, bindings among them, and bindings to the rest of infrastructure, creating new join-points, publishing a service, etc. Furthermore, every advice is parametrized by its properties which need to be specified before aspect weaving.

The *select* (`VariationPointSelect`) shares the idea of the AOP *point-cut* and it identifies a variation point in the infrastructure where the advice should be weaved. In our context, the *select* is represented by an OCL query which selects the right infrastructure join-point. The relation between the advice and actual infrastructure is also

guaranteed by the select which query for end-points where advice's bindings should be connected.

System-level aspect – it serves for injection of system-wide NFRs. Typically it does not modify `SystemAssembly` but it can introduce new actions into the back-end.

Application-level aspect – it is utilized for injection of application-specific NFRs. As in the previous case, it is typically used to modify back-end rather than the `ApplicationAssembly`.

Component-level aspect – the aspect encapsulates the component implementation with micro-components participating in component management including interception on interfaces, wrapping component implementation, etc. The typical non-functional requirements solved by the kind of aspect are life-cycle, introspection, component properties injection, thread management. This kind of aspect can also reconfigure the transformation back-end – for example, by requiring post-processing component implementation.

Platform-level aspect The kind of the aspect is tightly related to the rest of aspects and serves to configure execution platform. Typically, weaving of a platform-level aspect is enforced by injection of a component-level aspect. This kind of aspect typically inject various managers, bootstrap, or configure underlying middleware.

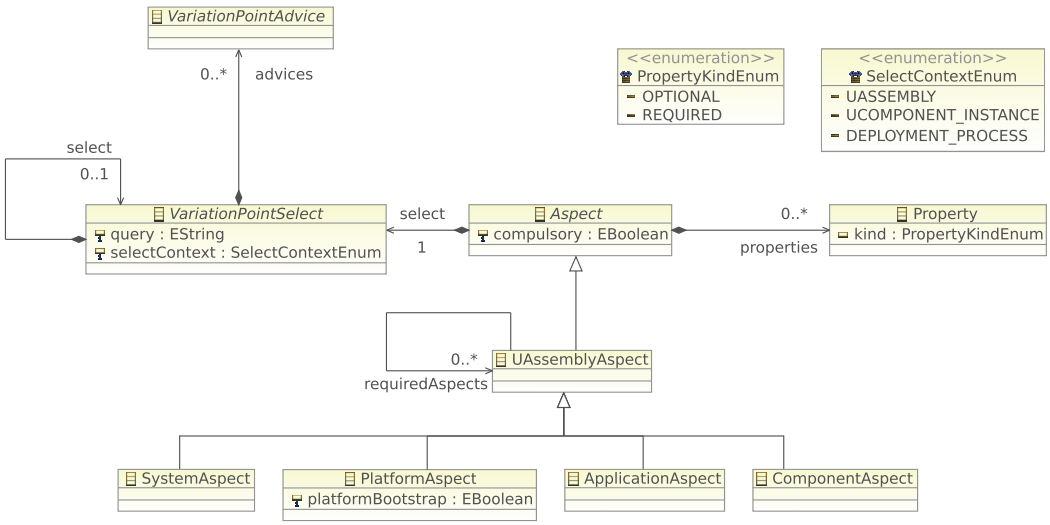


Figure 5.8: Infrastructure aspect.

5.4.2 Variation Points and Join-points

To identify well-defined locations in the execution infrastructure and to enable aspect injection, the concepts of infrastructure variation points is adopted. Two kinds of variation points are introduced: (i) implicit and (ii) explicit (see Figure 5.10). The implicit point refers entities composing a micro-component model – a micro-component,

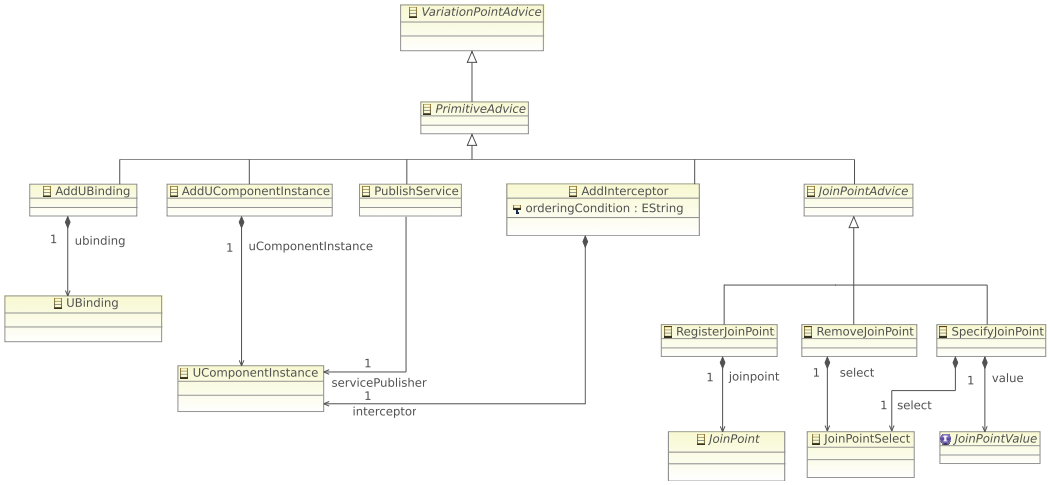


Figure 5.9: Advice hierarchy.

micro-binding, micro-interface, micro-assembly and conceptual entities as component, application, system, and platform.

The explicit variation point is called a *join-point* and it permits to exhibit a named location where other aspects can append their functionality. The join-point is declared by its name and micro-interface (and additional properties including the join-point priority). This kind of explicit variation point allows aspects to express their own "extension points". Therefore, several kinds of join-points exist. They serve to express different extension policies – for example, the `SingletonJoinPoint` join-point can bound only once, while the `ChainJoinPoint` join-point allows to be defined multiple-times. From this perspective, the join-point can be also considered as a way of ensuring dependencies among aspects. Here, it is important to mention, that join-points play a key role in defining and extending deployment process. They serve to determine a well-defined location in deployment process which can be extended by aspects.

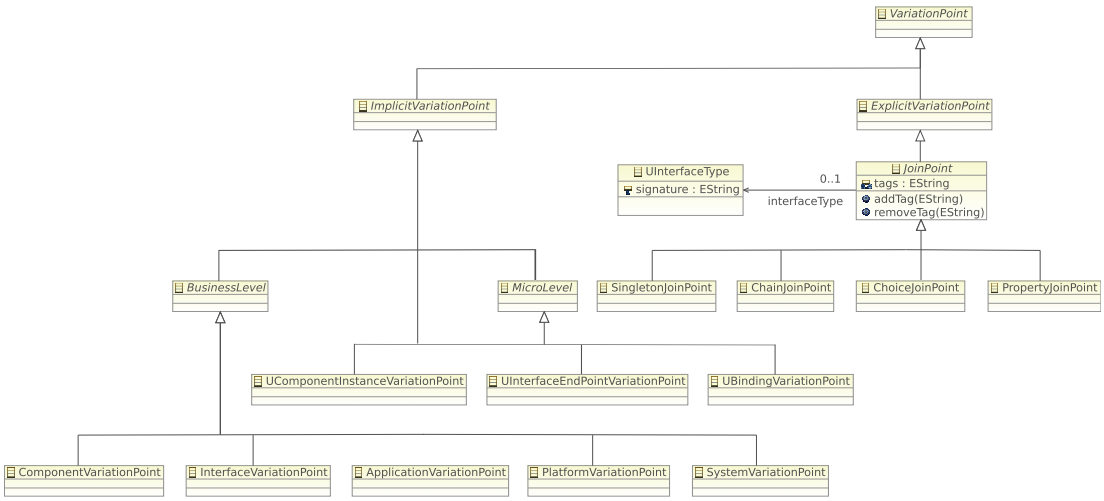


Figure 5.10: Variation points hierarchy.

5.5 Transformation Process

The objective of the process is to refine a given component assembly into its realization which can be launched. Our goal is to provide well-defined model-driven refinement which should be target technology agnostic and self-configurable. The motivation for an architecture of the transformation process design lies in compiler design [Muc97] – a common compiler is typically split into a part which is target platform independent (so called front-end) and a specific part which generates a machine code for a given platform (called back-end). The interoperability between both parts is achieved by a kind of inter-code which is independent on target platform and permits different optimizations.

Therefore, a transformation process is divided as depicted on Figure 5.11 into two parts – a front-end and back-end. The role of inter-code is played by execution infrastructure model which is at least partly target technology independent.

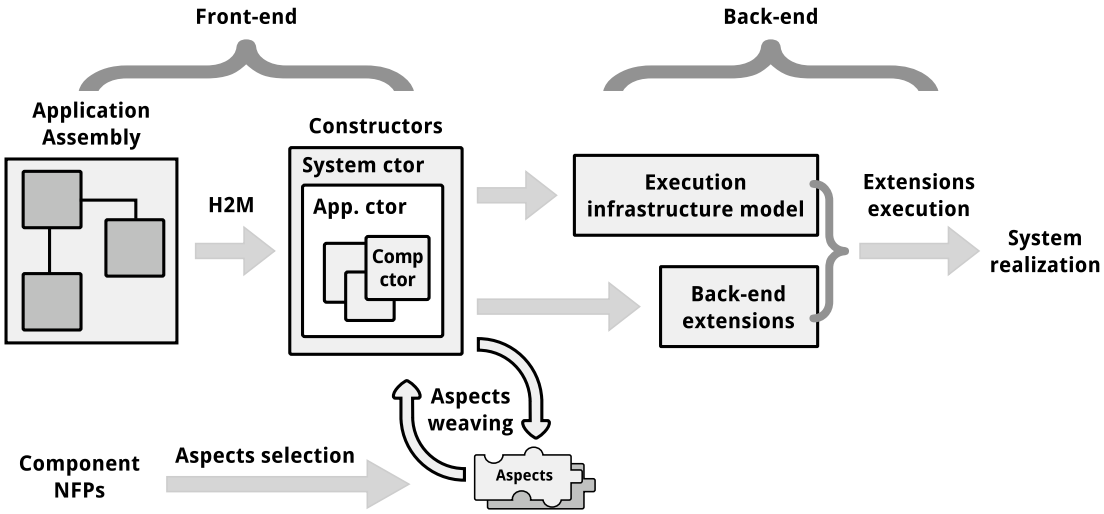


Figure 5.11: The scheme of the deployment process.

The front-end takes an assembly of components with its properties and expands it into a basis of an execution infrastructure and a set of aspects corresponding to NFRs. The result of aspect weaving is a model of execution infrastructure and extensions of the transformation process's back-end.

Realization of the EIM is the goal of the back-end. The stage processes the model and generates, adapts, configures, and compiles included micro-components. Moreover, it prepares all necessary configurations.

As in other generation frameworks ([BMH08], [RCGT09], [LPM+09]) the deployment process has a form of a pipeline. Each stage of the pipeline has well-defined inputs, output, and functionality. Here, it is possible to reuse micro-components to define and extend the transformation pipeline, since they have exactly defined requirements, provisions, and implementation (see Figure 5.12).

5.5.1 Front-end: From Component Assembly to Execution Infrastructure

The process translating the component assembly has to consider the complexity of the underlying component model. The model can introduce various features including

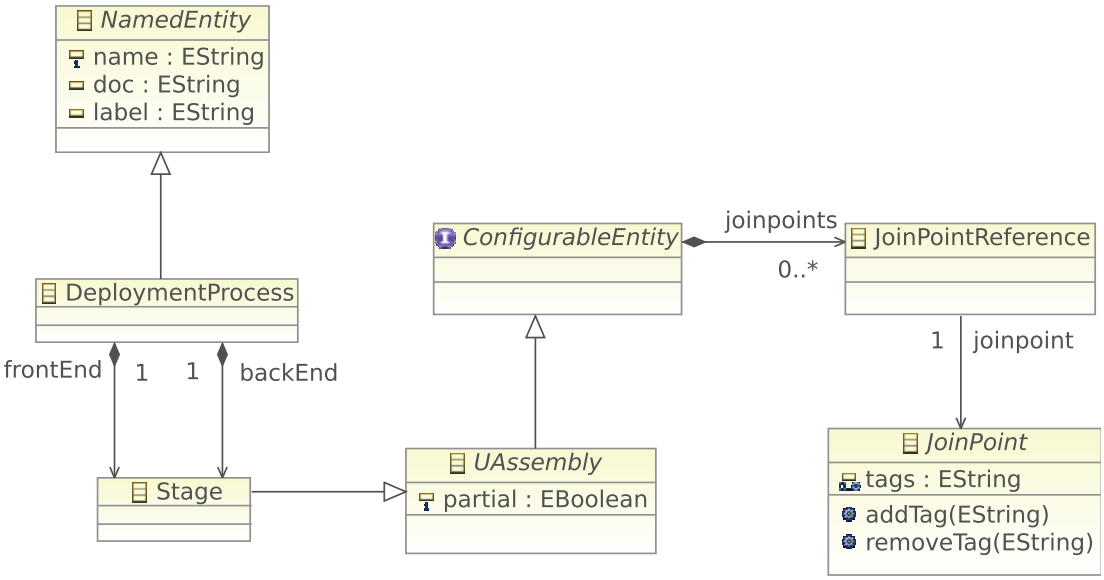


Figure 5.12: Deployment process meta-model.

flat/hierarchical components, exogenous/endogenous connectors, implicit/explicit bindings, behavior specification, etc. Furthermore, the component model contains a specification of non-functional requirements which can influence configuration of the component implementation, configuration of underlying component platform, and preparation of component infrastructure. The non-functional requirements can also enforce injection of new functionality into the platform infrastructure. All these factors are considered by the front-end during transformation the system architecture into its low-level form:

(i) The front-end transforms the assembly into a form of *constructors*. The constructor contains a partial micro-assembly as a seed for aspect weaving that represents a basis of the demanded execution infrastructure. The partial assembly contains a micro-components directly corresponding to component concerns – content and interfaces. The constructor further includes a set of aspects which require being applied. The aspects are resolved according to specified NFRs. The front-end prepares four kinds of constructor – for each component, overall application, system, and platform. These constructors directly correspond to micro-assemblies proposed in Section 5.3.1.

(ii) The next stage of the front-end, includes aspect weaving for each defined constructor. All constructors aspects are weaved into the defined micro-assembly. To mitigate complexity of aspects ordering, the weaving does not connect the injected artifacts with the rest of infrastructure. This is done after all aspects are weaved – then the dependencies are resolved and connected. This strategy implements a trade-off between usability and solving aspects constraints. During the aspect weaving the back-end is configured.

5.5.2 Back-end: From Execution Infrastructure to its Realization

The deployment back-end transforms the resolved execution infrastructure model into its realization with help of a selected implementation technology. Nevertheless, the back-end is constructed during creation of the execution infrastructure. The reason for such scheme lies in micro-components which typically require code generation, adaptation, compilation, or additional post-processing expressed by NFRs (*e.g.*, bundling). Therefore, we have designed the back-end as a set of join-points which the front-end can redefine. The back-end typically includes these join-points (it is important to stress, that every aspect can introduce a new join-point):

- `backend.architecture.optimization` – optimization of micro-architecture structure, *e.g.*, merging, sharing micro-components;
- `backend.architecture.fetch` – reuse of micro-components which are already implemented;
- `backend.code.generate` – code generation;
- `backend.code.optimization` – optimization of generated code, *e.g.*, code merging to reduce memory footprint;
- `backend.code.compilation` – code post-processing including its compilation;
- `backend.code.configuration` – preparation of configuration;
- `backend.code.bundling` – preparation of a system image which can be directly run or deployed into a target container;

Each of the defined join-points defines an extension interface containing a method `generate(SystemAssembly, Configuration)`, where `SystemAssembly` carries the resolved micro-architecture and `Configuration` represents additional technical details required for the back-end (*e.g.*, directories where generated code is stored).

The back-end itself only iterates over defined join-points according to their priority and call their interface. The output of the back-end depends on defined join-points – hence, it can differ from a non-structured source code to a bundle including compiled code and configuration for deployment into a container.

The back-end transformation is also affected by the demanded form of resulting system.

Moreover, the transformation can employ different strategies to assemble the system with respect to a selected implementation technology. For example, it can compose the platform with help of a selected configuration technology – *e.g.*, Guice, iPOJO. Or, it can adjust the micro-components to satisfy a given technology platform (*e.g.*, OSGi) which allows for deployment into a container. The adjustment can include modification of the micro-component content, generation of meta-information, and configuration files. The mapping of micro-components on a selected technology platform concepts corresponds to mapping *Platform Independent Model* (PIM) to *Platform Specific Model* (PSM) in MDA [MM03].

5.6 Discussion

The presented solution addresses the requirements stated in Section 5.2.4. However, there are still questions which are not directly evident. In this section we discuss them in more details.

Distribution The execution infrastructure model does not support distribution implicitly per se. We argue that to keep the core functionality of the execution platform minimal (which is among others important for embedded systems), the distribution should be treated as a NFR. Indeed it can be easily expresses via an aspect which injects into the execution infrastructure micro-components (typically interface interceptors) implementing distribution with help of a selected middleware. The same aspect enables generation of injected interceptors implementations and bundling of application parts respecting distribution boundaries.

In this respect, the aspect supplies a realization of a software connector managing communication. Regarding the software connectors, it is necessary to stress that the aspect can be automatically resolved from the specified application assembly to achieve the best solution reflecting the distribution and communication-related non-functional requirements. The topic is elaborated in more details in [Bur06].

Models at Runtime Since the execution infrastructure model serves only to express infrastructure concepts, there is a natural question how to preserve notion of application components at runtime. This behavior is useful in case of application introspection or reconfiguration at runtime. The proposed solution for this question is to utilize a dedicated aspect injecting into the infrastructure additional micro-components preserving notion of application components. Then micro-components' management interface allows for querying application component-based architecture. The same strategy is used to preserve notion of hierarchical components.

Realization of NFRs The method was designed to support wide range of non-functional requirements supporting application execution. However, it is obvious that it cannot support all existing NFRs including execution environment independent requirements such as usability, or accessibility, as such NFRs are to be captured at the application level rather than on the platform (or middleware) level.

Technology Independence The overall μSOFA method is designed to be implementation technology agnostic. Moreover, it does not require any particular kind of the technology. That means that the method can support any programming language (C, Java, C#), but also more advanced technology such as a configuration framework (*e.g.*, the case of the presented example which use Google Guice) or a component system to assemble the system realization. The method contains several locations where a target implementation technology needs to be known:

(i) application assembly refinement – the technology can influence the selection of aspects corresponding to application NFRs. For example, the aspect can expose that it supports life-cycle but only in the scope of the Google Guice configuration framework;

(ii) micro-component content – the micro-component content implementation is bound to the particular implementation technology;

(iii) micro-component content generator – the content generator itself does not need to be based on the selected target implementation technology, but the code, which it produces, has to be.

The fact that these locations are well identified allows us to manage the usage of the target implementation technology and produce system implementation in a fine-grained way.

Unanticipated Back-end Extensions The back-end structure introduced in Section 5.5.2 has a form of a pipeline with a predefined set of join-points. However, the presented set serves only for demonstration and can be arbitrary changed by the presented join-points mechanism. Therefore, the back-end supports unanticipated extensions by allowing for declaring a new join-point which is injected into the back-end's transformation pipeline according to the join-point's priority. In this way, the extension can alter the pipeline to support a new activity which other micro-components can extend too. Furthermore, apart from pipeline-driven communication among components, the micro-components can be also connected via regular bindings and hence exchange additional information. Respecting these extension strategies, the back-end transformation pipeline can be extended to more advanced transformation structures such as a parallel transformation pipeline.

Chapter 6

Models Interoperability

The chapter is motivated by preparation of implementation artifacts which has been emphasized in the previous chapter. It clarifies a role of code generation with respect to models describing its inputs and elaborates relations among these models in more details to achieve thesis goal G3. The text of this chapter is based on the following paper:

[MPBH12] Malohlava M., Plášil F., Bureš T., Hnětynka P.: *Interoperable DSL Families for Code Generation*,In Software: Practice and Experience, John Wiley & Sons, Ltd, ISSN: 1097-024X, DOI: 10.1002/spe.2118, April 2012.

6.1 Introduction

Nowadays software development takes advantage of the novel technologies which allow code generation from models at a variety of abstraction levels. These range from simple templates to advanced series of model transformations supporting backwards traceability. In this chapter, we show how such methods can be employed in the area of generating control elements of software components and connectors, focusing on the SOFA component model ([BHP06], [BHP+07]) where code generation is to be done both in Java and C (the chapter primarily focuses on Java). Nevertheless, the presented method can be employed in other code generation domains as well, specifically when several DSLs, multiple target languages, and/or optimization are to be considered.

Frequently, code generation into a target general-purpose programming language (*-language) is parameterized by the information not known statically, but available as late as the generation (model transformation converting a model to another model representing the code) is actually performed. Component-based systems are not an exception. Their development typically involves control elements which require to be tailored at the component assembly/deployment time or even runtime (*e.g.*, when dynamic architectures are supported). These elements include (i) functional units of software connectors bridging address spaces and handling differences in communication styles, and (ii) control elements of software components, such as interceptors handling

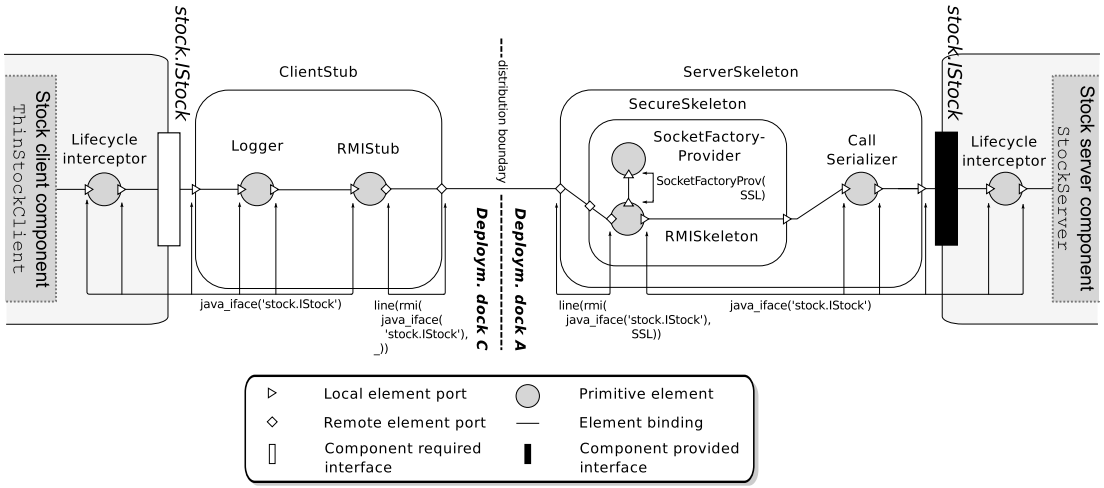


Figure 6.1: Element architecture and context.

calls on component interfaces in support of component lifecycle, QoS-related adaptation and monitoring. Having well-defined communication ports and despite being of relatively simple functionality, the elements may be either primitive or composed, forming a hierarchy.

Figure 6.1 shows a typical scenario in a component-based system consisting of two interconnected components – a client `ThinStockClient` and server `StockServer` – communicating by classic remote calls. Interconnection between the components is performed via a connector generated in an automatized way. An internal architecture of the connector is enforced by the desired communication method and extra-functional properties (in this case, communication has to be encrypted and monitored). The example illustrates the elements constituting this connector. The components `ThinStockClient` and `StockServer` communicate via the Java-interface of the type `stock.IStock`. To mediate communication, the connector is composed of `ClientStub` and `ServerSkeleton` top-level elements. `ClientStub` contains two primitive connector elements – `Logger` responsible for call logging and `RMIStub`, encapsulating the RMI middleware technology. Moreover, `ServerSkeleton` contains a composed connector element `SecureSkeleton` reflecting the given security requirement by encapsulating the `RMISkeleton` and the `SocketFactoryProvider`. Additionally, every call on both the required and provided interfaces "goes" through an *interceptor* to support lifecycle.

The original SOFA implementation [BHP06] generates code of all these elements, *i.e.*, interceptors in components and connector elements, in an automatized way [Bur06]; this is based on element specification in the form of:

(i) textual code *template (pattern)* of the element code,

(ii) description of the static *element architecture* (non-trivial in case of composed elements), and

(iii) description of the dynamic *element context* (*e.g.*, to determine the actual Java signatures of the bound interfaces in the context of the whole application).

Listing 6.1 shows an example of the textual code template for composed element

(such as `ClientStub` and `ServerSkeleton` in Figure 6.1). It is clearly visible that the template determines a skeleton of code where placeholders marked by double percent signs (*e.g.*, `%%GENERATE_ARCHITECTURE_INITIALIZATION_METHOD%%`) are to be substituted to reflect the element context (the actual interface types of the element) and element architecture, determining the internal, nested elements. Thus, the semantics of placeholders is determined by the specification of element architecture and context, which are encoded in a non-trivial, ad-hoc created, Java placeholder interpretation class plugged into the code generator. This way, the whole element specification is split between the textual code template and specific internals of the code generator. For example, in Listing 6.1 the implementation of the method `initializeArchitecture` (hidden by the placeholder `%%GENERATE_ARCHITECTURE_INITIALIZATION_METHOD%%` – line 19) traverses the element architecture and context and generates the code for creating instances of sub-elements and establishing bindings between them. All the code has to be encoded in the placeholder interpretation class.

Listing 6.1: Textual template for composed element.

```
1  public final class %%CLASSNAME%% implements
2      org ... ElementLocalServer,
3    /* ... */ {
4
5    /* Constructor */
6    public %%CLASSNAME%% (
7              ConnectorUnit parentUnit,
8              boolean isTopLevel) throws ElementLinkException {
9        this.parentUnit = parentUnit;
10       this.isTopLevel = isTopLevel;
11       dcm = DockConnectorManagerHelper. getDockConnectorManager();
12
13        initializeArchitecture () ;
14   }
15
16   /* Implementing the initializeArchitecture method
17    * Calling dedicated functionality of the code generator:
18    */
19   %%GENERATE_ARCHITECTURE_INITIALIZATION_METHOD%%
20
21   /* Depending on the element context:
22    * Calling dedicated functionality of the code generator:
23    */
24   %%GENERATE_PROVIDED_PORTS_METHODS%%
25 }
```

To summarize, the following three constituents are to be expressed in an element implementation specifications: ***E**lement Architecture*, ***C**ontext*, and *target code **P**attern*; therefore, we denote such specification as EPAC. An EPAC specification thus determines threee crucial constituents required for proper preparation of an element. By convention, when referring to a particular constituent, we will emphasize the corresponding letter by different font (*i.e.*, EPAC denotes specification of element's target code pattern, EPAC the architectural specification of an element and EPAC its context specification).

The EPAC specification determines three crucial aspects required for proper

preparation of the element: (i) its static structure via EPAC, (ii) the pattern of code (EPAC) delineating the element's implementation in a target language, and (iii) the environment (context) into which the element is to be deployed (EPAC). Moreover, each specification has its own lifecycle driven by its role in the system development process and has a dependency on *-language to different extent. The architecture specification EPAC is a part of the database containing pre-defined static architectures of the control elements to be generated. The database is usually prepared at a very early stage of system design (*e.g.*, together with decisions on interface communication styles) and it is not intended to be a subject to frequent changes during system development. In general, EPAC dependency on the *-language is just minor. On the contrary, being substantially dependent on *-language, the context specification EPAC can only be created as late as decisions of component assembly and deployment have been made. Along similar line, the EPAC specification (element code pattern) can be created right after EPAC specification is available. The bottom line is these three specifications are not typically created simultaneously and an EPAC and EPAC specification can be employed with a number of different EPAC specifications. With regard to the scenario in Figure 6.1, the EPAC specification of the `RMIStub` primitive element is represented by: (i) EPAC including a definition of the element structure containing a specification of input and output interfaces reflecting utilization of the RMI middleware; (ii) EPAC prescribing patterns of the Java code implementing the element functionality; and (iii) EPAC containing resolved signatures of input and output element's interfaces implied by the context of the whole connector.

Furthermore, the specifications have to cooperate (be *interoperable*) in terms of one can refer information provided by another. For example, an EPAC specification typically refers the both EPAC and EPAC specifications to learn on element's interfaces, and also an EPAC tightly adheres to an EPAC. It should be emphasized that in template-based code generation, this interoperability has to be hard-coded in a class interpreting a placeholder.

6.1.1 Problem Statement and Goals

In general, simple template-based code generation of control elements is a process inherently inflexible in four respects:

(i) EPAC and EPAC specifications and their processing have to be encoded as a plugin of the code generator;

(ii) porting to a new *-language means not only re-writing the code template (EPAC specification), but also modifying EPAC and EPAC specifications including their encoding in the code generator;

(iii) any further operations upon the resulting code cannot be easily integrated within the generation process (*e.g.*, code optimization by merging elements);

(iv) interoperability among specifications is hard-coded in the code generator.

The problem this chapter aims to address is this inflexibility which makes it very hard to accomplish element code generation from EPAC specification in a single framework, especially when multiple target code languages and code optimization

are to be considered. Thus a challenge is to overcome these obstacles by finding a way of employing multiple DSLs and modern code generation techniques, preferably based on model transformations, which would allow for DSL interoperability and easy porting to another *-language. From the perspective of code generation, the following three model transformation techniques suitable for code production have been identified ([CH06, MVG06]): (i) template-based, (ii) visitor-based, (iii) model-to-model. They can be characterized as follows:

In the context of element code generation, all these techniques would employ the EPAC and EPAC specifications similarly, but would differ in the way they make use of EPAC (code pattern) specification. The techniques (i) and (ii) differ in controlling code generation process. In the case of (i), the controlling process is based on traversing the code pattern in a text-based template where marks (such as our placeholders) trigger specific functionality of the code generator (Acceleo [23], JET [10], JavaServer Pages [28], Velocity [5], Xpand [11]). On the contrary, in case of (ii), such as JAMDA [8], the code generator would contain an encoded EPAC specification and visit the EPAC and EPAC specifications. Both techniques would fail when multiple target languages are to be considered, since, for each of them a dedicated code template or visitor would have to be created. As an aside, since both techniques (i) and (ii) lack a comprehensive representation of the code to be generated, code optimization would be hard to implement.

More promising is a technique of the (iii) category, supporting step-by-step refinement of the artifact model resulting into code ([JK06, dJVV01], [14]). This allows addressing multiple objectives in a sequence of transformation steps, including various optimizations. Obviously a key challenge (and the main goal of the chapter) is to propose an appropriate form of the model in each step and define efficient transformations reflecting the objectives adequately.

With respect to the goal, the chapter is structured as follows. Section 6.2 overviews the *Element Code Generation method* (ECoGen), while Section 6.3 focuses on DSL families. The next section presents details of ECoGen-J generation framework stressing also DSL interoperability including language assimilation yielding the target code. Section 6.5 evaluates the presented approach, while Section 6.6 describes related work. The last section concludes the chapter and sketches potential directions of future work.

6.2 ECoGen Method: Overall Strategy and Related DSL Families

The proposed code generation method follows the general MDD strategy involving multi-staged model transformations. It stems from the *MetaBorg method* [BV08] which allows embedding of specific DSL constructs into a hosting programming language (such as Java), and provides tools [dJVV01], [3] for assimilation (conversion) of these constructs into the hosting language. The proposed code generation method serves to transform an EPAC specification into the corresponding code in a selected *-language and extends MetaBorg by the employing several DSLs simultaneously. The presented method is generic enough to be easily applied for a new *-language.

Figure 6.2 shows the overall code generation strategy: Technically, the input is a triple of EPAC, EPAC and EPAC specifications, each of them in a dedicated DSL (in the same order): *element pattern language* EPLang-*, *architecture description language* ADL-

*, and *context description language* CDL-*. All of them are converted by a text-to-model transformation into the form of an abstract syntax tree (AST-ADL-* , AST-CDL-* , AST-EPLANG-*). Intentionally, the AST representation is chosen since it is an elegant model of a textual specification, allows further transformations, and is supported by several well-elaborated tools such as Stratego [3], TXL [CHHP91], or DMS [BPM04].

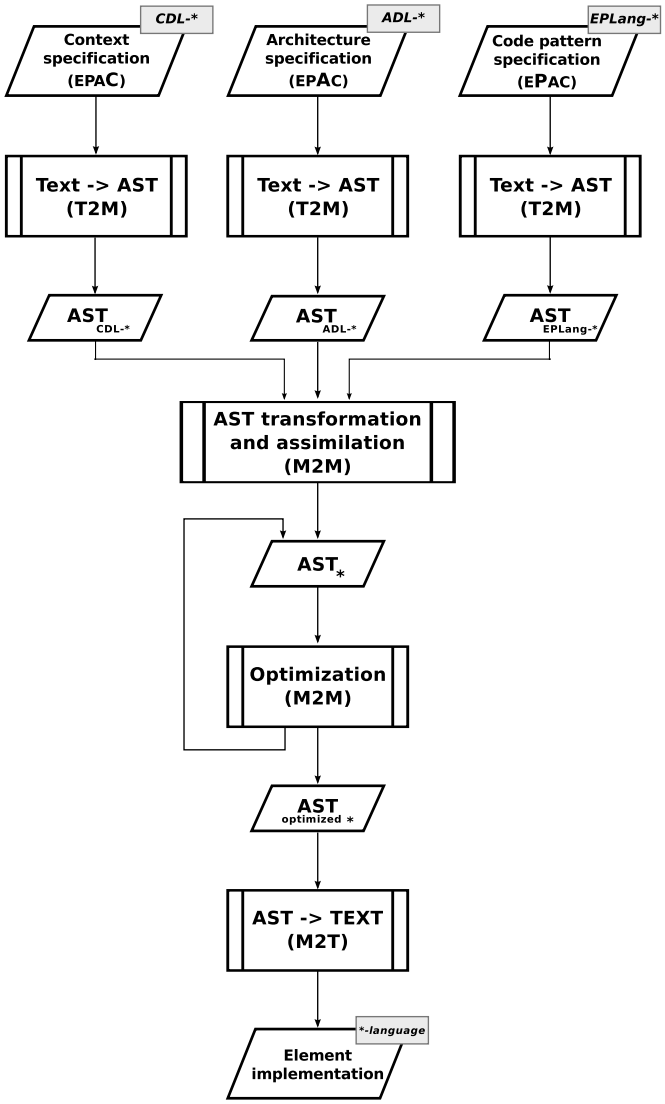


Figure 6.2: Overall generation strategy.

In particular, the EPAC specification is formed by constructs of the domain specific language EPLang embedded into a *-language (by convention this embedding yields the language EPLANG-*; similar convention on embedding applies to ADL-* and CDL-*). Next, driven by the transformations defined for the EPLANG-* constructs, AST-EPLANG-* is traversed several times and step-by-step converted (model-to-model transformation) into AST-* which already represents the element's code in the desired *-language. This way, the domain specific EPLang constructs are *assimilated* [BV08] into the *-language. In this process, the EPAC and EPAC specifications are taken into account by transforming and assimilating fragments of the AST-ADL-* and

AST-CDL-* during the AST-EPLANG-* transformation as described in Section 6.4. This is necessary to determine the actual types of the element"s ports from the context specification EPAC and to determine the sub-elements from the architecture specification EPAC. Optimizations of the generated code are done in an iterative way upon AST*; this includes code merging [BCL+06], inlining, and call indirections removal [9]. From the resulting AST-optimized *, the last transformation produces the resulting code in textual form (*i.e.*, in the *-language).

Obviously, the generation process of AST-* relies on interoperability of three DSL languages: ADL-*, CDL-*, and EPLANG-*. The key reason for having to accompany an EPLANG-* specification with the ADL-*, and CDL-* specifications, is that, typically, the information provided by the CDL-* and ADL-* specifications is not available at the moment the EPLANG-* specification is being written. Therefore, having different lifecycles, these specifications are provided as standalone entities and an interoperability of all corresponding DSLs is required.

6.3 EPLANG, CDL, and ADL Families

6.3.1 Why Three Domain-specific Languages and Their Families

In general, designing a DSL language requires identifying the related domain concepts and their relationships [KT08, MHS05] which are crucial for expressiveness and thus needed to be reflected in the language constructs. For element code generation, we identified three domain-related requirements which have to be taken into account – each of the three DSL languages has to:

(i) reflect the domain vocabulary;

(ii) be parameterized by the desired target *-language;

(iii) reflect the fact that EPAC parts are not created simultaneously (have different lifecycles).

While (i), domain vocabulary, is the central concept of domain-specific modeling [KT08, Eva03], the requirements (ii) and (iii) are specific in terms of flexible generation of element code in a *-language. The requirement (iii) is addressed by employing three different DSL languages. This also well complies with the component model domain conventions involving an architecture expressed in an ADL and context by a language specifying component assembly and potentially deployment. Therefore, it would be counterproductive for designer to break this conventions by striving to combine all of these specifications into a single DSL. Moreover, the requirement (ii) resulted in designing families of these DSL languages for each of the EPAC parts. These are depicted in Figure 6.3, where also the desired interoperability is graphically emphasized. Thus, for a specific target *-language, from each family its "*" member is to be selected (*e.g.*, EPLANG-J, ADL-J, CDL-J when Java is selected as the *-language). From the perspective of the final goal (code generation), this also means that such a family member provides constructs specific to the target code in *-language.

[9] In general, the optimization are *-language dependent, but there is a group of independent optimizations targeting the element architecture (*e.g.*, composite element flattening), which can be employed during early stage of generation. Details of optimizations are out of the scope of the thesis.

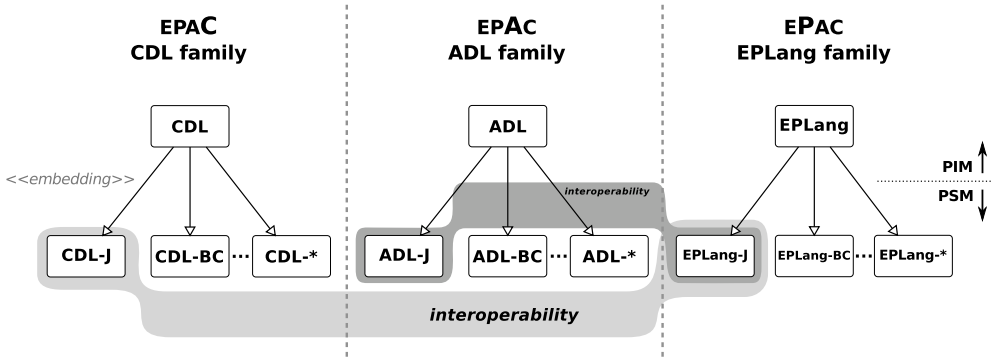


Figure 6.3: DSL families and required interoperability among languages.

Central to each of the families is its core language (CDL, ADL, and EPLANG). This concept follows the MDA approach [MM03]: A core language serves to specify a platform independent model (PIM). A member of the family (*e.g.*, CDL-J, ADL-J, EPLANG-J) serves to express a platform specific model (PSM). In general, a member of the family is created by embedding, which is achieved by both merging and restricting the core and target *-language grammars. The result of the embedding is a grammar of a new language. In the following subsections we focus always on the core language and its embedding to Java (and also bytecode in the case of EPLang family).

6.3.2 EPLANG Family

In essence, the core of the family is the EPLANG language (Element Pattern Language) – a DSL language allowing expressing, at an abstract level, the desired effect of an EPAC specification, which results into a complete element implementation based on the selected *-language. A primary requirement is that EPLANG constructs have to express key element-related concepts (*e.g.*, element ports), to query element architecture and context, and to determine other details of element implementation (*e.g.*, class and package names). Furthermore, EPLANG has to be easily embeddable into the *-language.

The example of the EPAC specification is illustrated by Listing 6.3. The example shows an implementation of `RMIStub` from Figure 6.1 declared in EPLANG-J. Lsting 6.2 demonstrates a skeleton of the same specification in pure EPLANG without Java code (in reality, this pure EPLANG code is not explicitly used – we show it here for illustration).

A key part of the element specification constitutes the `element` construct (lines 3-56), composed of a sequence of the `implements port` constructs (lines 14-17 and 20-55) expressing how ports become actual interfaces. Each of the implements port constructs references an interface type either directly by a string containing its signature, or by the result of a *query* (*e.g.*, `$query{ports.port(name=call)}:type`, line 20), which refers to the corresponding part in an EPAC or EPAC specification. Technically, both of these specifications are targeted by the query and the successful response is interpreted as the result. The query always returns a result, however, it the case of an incorrect query (*e.g.*, referring non-existing port), the result can be empty. In such a case, the generation process fails.

Syntactically, a query takes the form `$query<NAVIGATE>:<EXTRACT>`. Seman-

tically, the <NAVIGATE> part of the query selects a set of sub-trees in AST-ADL-* or AST-CDL-* , and <EXTRACT> part finds out the demanded information stored in the sub-trees. While the <NAVIGATE> part can be evaluated in AST-ADL-* or AST-CDL-* , just on syntactical basis, the <EXTRACT> part needs interpretation specific to each kind of the <EXTRACT> statement.

Overall, the following kinds of <EXTRACT> statements, which are used in queries targeting an EPAC specification, serve to obtain (i) a number and (ii) names of element ports and eventually, in case of a composed architecture, (iii) the names of sub-elements. In similar vein, <EXTRACT> statements used in the queries targeting an EPAC specification provide the names of the (iv) generated class, (v) related package, and (vi) actual element port types. Furthermore, in case of composite element, there is a variant of <EXTRACT> which provides (vii) the names of sub-elements' implementation types.

Implementation of the interface methods associated with a port is provided via the `method template` construct (lines 21-54). It determines the code pattern to be applied in each method of the interface the port is being turned into (recall that no interface type is known at the time of template creation because it has to be derived from the component interface the element is to serve); this information is to be sought in CDL-*. Typical examples of method template utilization are simple call delegation in an element and adaptation of two incompatible interface signatures via an element. For that purpose, method template declaration can refer the method's name and its parameters (`$method.name`) and `$method.args`, line 32). The latter being a list which can be manipulated by the `pop`, `append` and `count` statements for removing, appending and counting parameters. For manipulation with a method return parameter, the following three statements serve: `$method.declareReturnType` declares a temporary variable of a method return type (line 22), `$method.setReturnValue` assigns the temporary variable a given value (line 34), and finally `$method.returnStm` prescribes creating a return statement providing the value of the declared temporary variable (if necessary; line 52). Furthermore, the exception list stored in the variable `$method.exceptions` can be manipulated in the same way as method parameters with help of `pop`, `append`, `count` statements.

Variability of elements' implementation is supported by the extend constructs, extension points, and import statements. The `extend` construct provides a simple kind of specification inheritance, consequently allowing for code sharing between elements. The `extension point` serves to declare the location in a parent template which may be modified by the implementation of the inherited template (via the `extend` construct). Finally, the `import` statement (lines 6, 7) includes a predefined specification block.

Listing 6.2: Skeleton of EPAC specification in EPLANG for `RMIStub` element.

```
1 /*+ +++ +*/
2
3 element rmi_stub {
4
5   /*+ +++ +*/
6
7
8
9
```

```
10  $import("ElementMethodsImpl.eplang")$
11  $import("ReconfigurableElementImpl.eplang")$
12
13
14  implements port /*+ +++ +*/{
15    /*+ +++ +*/
16
17  }
18
19  /* Implementation of interface parameter */
20  implements port $query{ports.port(name=call):type} {
21    method template {
22      ${method.declareReturnValue}
23
24      $set encoderNeeded=...$
25
26      $if (encoderNeeded)$
27
28        $apply(...)$
29
30      $end$
31
32      $append(method.args, /*+ ++++ +*/)$
33
34      $setReturnValue(...)$
35
36
37
38
39      /*+ +++ +*/
40
41
42
43
44      $if (!method.returnType.isPrimitive) $
45
46
47      /*+ +++ +*/
48
49
50
51      $else$
52        ${method.returnStm}
53      $end$
54    }
55  }
56 }
```

The EPLANG language defines the following types and operations: integer (+, ==, !=), string (+ (concatenation), ==, !=), and associative dictionary of strings (indexed also by a string). Such a dictionary represents a set of key-value pairs. These can be manipulated by the indexing operator `[]`. Variables are of dynamic types. The operands in an expression can be only literals, variables, and queries. The names of a variable can be hierarchical – this helps navigate over composed elements. These language features are illustrated in the example of an EPLANG-J specification in Listing 6.3. This also includes the `apply` statement which applies a given expression to

each of the list members satisfying a given condition (*e.g.*, the `apply` statement on line 28 boxes each primitive type in the given argument list into the `RMIDecoder` call).

The EPLANG control statements include the `foreach` and `rforeach` (recursive foreach) cycles, and also the `if` (condition) and `set` (assignment) statements, Obviously, the purpose of these control statements is to specify sequencing of code generating actions (in detail illustrated in [BMH08]).

To demonstrate the potential of the EPLang family, we implemented two domain-specific languages (and corresponding generators): EPLANG-J and EPLANG-BC. The former generates Java-based elements, the later produces bytecode of elements at runtime to support efficiency.

Listing 6.3 shows an example of the full EPAC specification for `RMIstub` from Figure 6.1 in EPLANG-J. It specifies that the element implements a provided port (line 20-55). Furthermore, there is a method template for adapting and delegating an incoming call (lines 21-54; the actual method signature is determined from an EPAC specification by the algorithm described in Section 6.4.2).

EPLANG-BC is another member of the EPLANG family. The motivation for choosing Java bytecode as a *-language is to reduce compilation complexity when generating element code at runtime.

An EPLANG-BC specification of `RMIStub` is shown in Appendix C. Technically, such a specification is pre-prepared by compiling an EPLANG-J specification with a dedicated tool which first removes the EPLang constructs, compiles the pure Java code and returns back the EPLang constructs into the corresponding places of the resulting bytecode forming the EPLANG-BC specification.

Listing 6.3: EPAC specification of `RMIStub` element in EPLANG-J.

```
1  package ${package};
2
3  element rmi_stub {
4    /* Delegation target */
5    protected $query{ports.port(name=line):type} target;
6    /* Constructor */
7    public $query{:classname}(ConnectorUnit parentUnit) { /* ... */ }
8
9    /* Import common methods for all elements */
10   $import("ElementMethodsImpl.eplang")$
11   $import("ReconfigurableElementImpl.eplang")$
12
13   /* Implementation of a Java interface */
14   implements port ElementLocalServer {
15    public Object lookupElPort(String name) {
16          /* ... */ }
17   }
18
19   /* Implementation of interface parameter */
20   implements port $query{ports.port(name=call):type} {
21    method template {
22      ${method.declareReturnValue}
23
24      $set encoderNeeded=count(ARG in method.args
25                               where !method.args.ARG.type.isPrimitive)$
26      $if (encoderNeeded)$
27        RMIObjectEncoder rmiEncoder = new RMIObjectEncoder();
28        $apply(rmiEncoder.adaptObject(ARG) for ARG in method.args
```

```
29                    where !method.args.ARG.type.isPrimitive)$
30          $end$
31          try {
32            $append(method.args, SOFAThreadHelper.getCallContext())$
33            $setReturnValue this.target .${method.name}($implode(method.args)$)$
34          }
35          $if (encoderNeeded)$
36          catch (RMIObjectAdaptorException e) {
37            throw new ConnectorTransportException (e);
38          }
39          $end$
40          catch (RemoteException exc) {
41            throw new ConnectorTransportException (exc);
42          }
43          $if (!method.returnType.isPrimitive) $
44          try {
45            return (new RMIObjectDecoder()).adaptObject (${method.returnVar});
46          } catch (RMIObjectAdaptorException e) {
47            throw new ConnectorTransportException (e);
48          }
49          $else$
50            ${method.returnStm}
51          $end$
52        }
53      }
54  }
```

6.3.3 ADL Family

As mentioned in Section 6.2, an EPAC specification in EPLANG-* has to be complemented by an element architecture specification EPAC. Given an element, it describes its ports' roles (provided or required); furthermore, for a composed element it specifies its internal architecture – sub-elements and their bindings. In general, an EPAC specification is written in an ADL-* created by embedding the core ADL language into a (subset of) selected *-language. A dedicated ADL-* language is typically required because of the need to specify various extra-functional properties and dependencies among them in the *-language (*e.g.*, total memory consumption of a composite element is specified as a sum of memory consumptions of its sub-elements). The *-language serves here mainly as an "expression language".

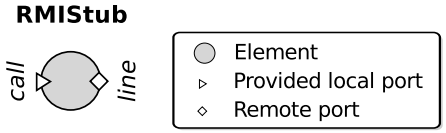


Figure 6.4: EPAC specification of the `RMIStub` in ADL-J.

Listing 6.4 shows an example of an ADL-J specification of the `RMIStub` element. The specification is split into: (i) definition of an element type, and (ii) definition of an internal architecture (implementing the element type). In principle, the element type defines a black box view of `RMIStub` (Figure 6.4) by specifying its provided and required ports (line 2-6). The architecture (lines 9-19) implements the given element

type (line 11) by clarifying schematic relations among ports' types (line 14, 16). Furthermore, the example shows a definition of the extra-functional properties (lines 19-24) used for computing element's memory consumption. An example of a composed element ADL-J specification can be found in Appendix A. In a similar vein, ADL-BC is provided (in the same style as in the case of EPLANG-BC – see Section 6.3.2).

Listing 6.4: The EPAC specification of the RMIStub element in ADL-J.

```
1  <!-- Black box view of element. -->
2  <element-type>
3    <name>stub</name>
4    <port name="call" role="provided" />
5    <port name="line" role="remote" />
6  </element-type>
7
8  <!-- Glass box view of element. -->
9  <element-architecture>
10   <name>RMIStub</name>
11   <type>stub</type>
12
13   <architecture>
14     <port name="call" type="I" />
15     <port name="line">
16       <signature-entry ref-name="rmi" role="client" type="rmi(I)" />
17     </port>
18   </architecture>
19   <efp>
20     <efp-def>java.lang.Long memoryConsumption</efp-def>
21     <efp-def>java.lang.Long baseConsumption = 4096</efp-def>
22     <efp-def>java.lang.Long totalConsumption</efp-def>
23     <efp-value>totalConsumption = memoryConsumption + baseConsumption</efp-def>
24   </efp>
25 </element-architecture>
```

6.3.4 CDL Family

Given an element instance, its context specification EPAC describes the actual types of its ports. In general, it is written in a CDL-* language created by embedding a core CDL language into a sub-set of *-language. A dedicated CDL-* is typically required, since the actual types of ports, values of extra-functional properties, and additional system initialization code are to be specified in *-language.

Listing 6.5 shows an example of a CDL-J specification of the RMIStub element. It specifies the actual types of the ports (lines 10-14), technical details needed for code generation in Java (class name – line 7, class package – line 5), actual values of extra-functional properties (lines 17-19), and Java code for system initialization. It should be emphasized that a CDL-* specification refers only to the black box view of the element defined by ADL-* – no qualification of internal port types is provided in case of composed element.

Listing 6.5: The EPAC specification of the RMIStub element in CDL-J.

```
1  <element-context>
2    <!-- name of element -->
```

```
3   <name>RMIStub</name>
4   <!-- package for new element -->
5   <package>T00012</package>
6   <!-- class name for generated element -->
7   <classname>RMIStub</classname>
8   <!-- list of ports types -->
9   <ports>
10    <port name="call">
11      <type>stock.IStock</type>
12    </port>
13    <port name="line">
14      <type>stock.IStock</type>
15    </port>
16  </ports>
17  <efp>
18    <efp-value>memoryConsumption = 3680</efp-value>
19  </efp>
20  <system-bootstrap>
21    System.setSecurityManager(new ElementSecurityManager());
22  </sytem-bootstrap>
23 </element-context>
```

6.4 ECoGEN-J Generation Framework

6.4.1 Overview

Figure 6.5 illustrates the ECoGEN-J *generation framework* (*i.e.*, ECoGEN for Java) which is an instance of the overall generation strategy (Figure 6.2) where the *-language is Java. This framework is based on the Stratego/XT toolset ([dJVV01],[3]) supporting program transformation based on AST rewriting rules [Vis05]. The Stratego/XT represents an integrated tool for (i) defining DSL grammars and executing corresponding text-to-model (T2M) transformation; (ii) specifying and executing model-to-model (M2M) transformations upon ASTs; (iii) specifying and executing model-to-text M2T transformations. As to (i) T2M, the SDF (Syntax Definition Formalism) format ([HHKR89]) is employed which allows specifying context-free grammars and generation of scannerless generalized LR parsers for the languages. In principle, given languages L1, L2 with corresponding grammars G1, G2 defined in SDF, a new language L can by created by combining and restricting the rules of G1 and G2. Moreover, M2M transformations (ii) are done by a manually written program in the Stratego language expressing AST transformations (AST rewriting rules [dJVV01, Vis05]). Thus, with the help of a single toolset, all of the generation steps from Figure 6.2 can be implemented.

In Figure 6.5, the generation process begins by parsing each of the three parts of an EPAC specification. This is done by the Stratego tool *sglri* driven by definitions of the EPLANG-J, CDL-J, ADL-J grammars in the SDF format. The resulting parsers produce AST-ADL-J , AST-CDL-J , and AST-EPLANG-J corresponding to the respective parts of the EPAC specification.

Further, the generation process assimilates EPLANG statements into Java while employing the interoperability of EPLANG-J, ADL-J, and CDL-J languages. This is achieved by a Stratego program which works in three transformation stages:

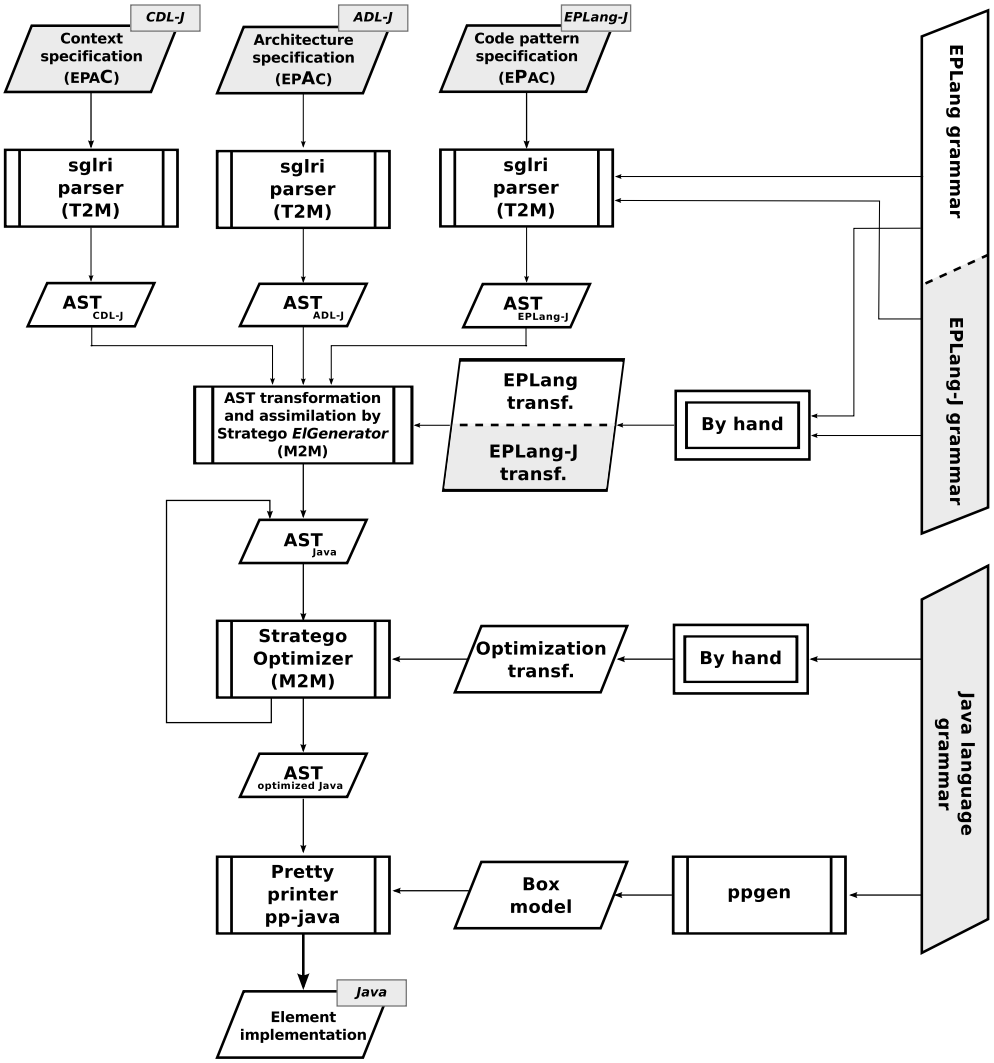


Figure 6.5: ECoGEN-J generation framework.

1. The first stage evaluates pure EPLANG statements [BMH08] as follows: code imports are done, then, to simplify later transformation to Java, statements are converted into a canonical form (this includes "inlining" of control statements whenever statically possible). This particular sequence of transformation steps ensures that the whole process converges: composed EPLANG statements are step-by-step simplified and finally transformed into primitive EPLANG statements which are further solely converted into Java.

2. The second transformation stage evaluates queries. They are handled with a simplification strategy similar to the on for handling composed statements. For this purpose, a specific algorithm merging information from three ASTs was designed as described in Section 6.4.2.

   In general, the first and second stages are Java independent, so that they can be reused for another choice of *-language.

3. During the last transformation stage, the Stratego program handles the Java-

specific EPLANG-J constructs and composes the resulting AST-*.

All in all, this three-phase approach well enforces step-by-step refinement of AST. To illustrate AST rewrite rules, Listing 6.6 shows a Stratego program `process-import` serving to assimilate `import` statements by rewriting them to the content of a file indicated by a parameter. Specifically, the rules on lines 5-8 parse the file name, while the rule `parse-EPLang-J` on line 12 parses the content of the file and returns it.

Listing 6.6: AST transformation rule assimilating the `import` statement.

```
1  process-import:
2      Import(Filename(X)) -> Y
3          where
4              if <is-relpath> X then
5                  get-template
6                  ; abspath ; dirname
7                  ; <strcat> (<id>, "/") ; <strcat> (<id>, X)
8                  ; ?finput
9              else
10                 where ( !X => finput)
11             end
12             ; <parse-EPLang-J> FILE(finput) => Y
```

Again, driven by a Stratego program, the *Optimizer* deals with optimizing the code represented by AST-Java. As mentioned in Section 6.2, this includes code merging, inlining, and call indirections removal.

Finally, the code represented now by AST-optimized Java is transformed by the Java pretty printer (*pp-java* – a part of the Stratego/XT toolset) into its textual form. Naturally, the produced Java code is to be further compiled, bundled, and deployed into a runtime environment.

For illustration, Listing 6.7 shows the code generated from the `RMIstub` EPAC specification (Listings 6.3-6.5) for the interface `stock.IStock` containing a single method `sell` with one integer parameter.

Listing 6.7: Generated Java code for `RMIStub` element.

```
1  class RMIStubImpl implements
2          stock.IStock, ElementLocalServer, ElementRemoteClient {
3    protected IStockRemote target;
4
5    public RMIStubImpl(ConnectorUnit parentUnit, ...) { /* ... */ }
6
7    /* Implementation of imported common methods */
8    public String getElementInfo(String indent) { /* ... */ }
9
10   /* Implementation of a particular Java interface */
11   public Object lookupElPort(String name) { /* ... */ }
12
13   /* Implementation of interface IStock */
14   void sell(int num) {
15     Object context = CallHelper.getCallContext();
16     this.target.sell(num, context);
17   }
18 }
```

6.4.2 Handling Queries – Basic Idea

For brevity, in the rest of the text, the ADL-*, resp. CDL-*, language is referred to as a *side-*-language* (and the corresponding AST as a *side-AST*).

To interpret a query, it is necessary to map the concepts in AST-EPLang-* to the concepts of a side-AST. In principle, it would be possible to express such mapping at the level of EPLang-* and side-languages grammars (*e.g.*, by technique [JVB+10] as discussed in Section 6.6 using intertwined grammars). However, such solution would bring a strong dependency among languages, which is not desired. Thus, the queries allowing expressing a loose dependency are introduced.

The actual technique of interpreting the query via AST transformations is as follows (see Figure 6.6). The target AST contains a query and its `<NAVIGATE>` part describes a path that determines two sub-trees in the side-AST. Further, the `<EXTRACT>` part extracts the desired nodes from both sub-trees. The result is integrated into a single tree as the outcome of the query. This tree is then assimilated back into the target AST.

In general, all the sub-trees determined by a query have to be integrated into a single AST sub-tree conforming to the structure of AST-EPLang-* tree as if it was the result of T2M transformation of its original EPLang-* specification. When an identified sub-tree of the side-AST is trivial, *e.g.*, carrying just a simple value (being a leaf associated with a string or number), it is assimilated into the EPLang-* AST simply by copying the corresponding node. On the contrary, a nontrivial set of sub-trees (*e.g.*, such as definition of all ports) is assimilated by iterative transformations into the EPLang-* AST as a sub-tree representing a list of values.

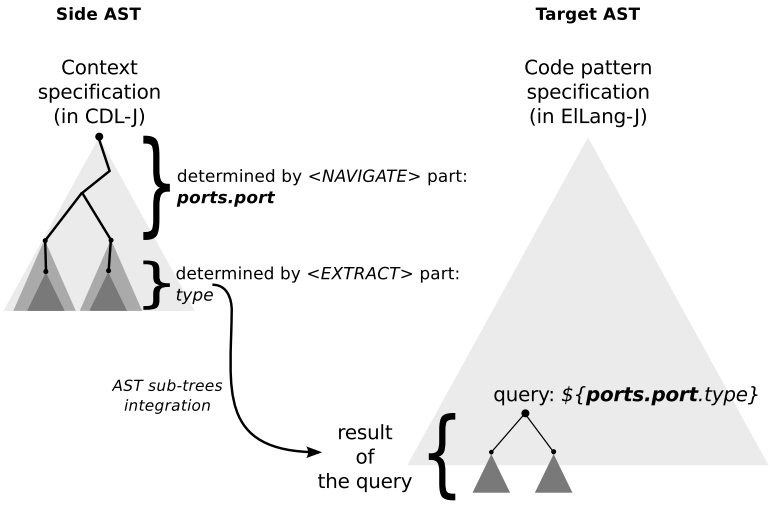


Figure 6.6: Handling a query – basic idea.

6.4.3 Assimilation and DSLs Interoperability

Basically, interoperability – in terms of the ability to interpret the queries issued in an EPLang-* AST – is required between each side-*-language and EPLang-*. This specifically means interoperability between EPLang-J and CDL-J and also EPLang-J and ADL-J. This is visualized in Figure 6.7. More precisely, query interpretation

is determined by applying a set of AST transformation rules the effect of which is modeled by the following mappings:

Definition 1. Let SLAST be the set of sub-trees of all ASTs in the side-*-language and ELAST the similar set for EPLANG-*. Also let $S \in SLAST$ and $T \in ELAST$ be ASTs and Q (sub-tree of T) be a query containing <NAVIGATE> and <EXTRACT>.

Interoperability of the languages is defined based on the following mappings (transformation rules):

$$nmap : SLAST \times ELAST \rightarrow 2^{SLAST} \tag{6.1}$$
$$emap : 2^{SLAST} \times ELAST \rightarrow 2^{SLAST} \tag{6.2}$$
$$integrate : 2^{SLAST} \rightarrow ELAST \tag{6.3}$$
$$replace : ELAST \times ELAST \times ELAST \rightarrow ELAST \tag{6.4}$$

such that the following holds:

$$nmap(S, <NAVIGATE>) = \mathbb{P}\ , \quad \text{where } S \in SLAST, \mathbb{P} \in 2^{SLAST} \tag{6.5}$$
$$emap(\mathbb{P}, <EXTRACT>) = \mathbb{R}\ , \quad \text{where } \mathbb{R} \in 2^{SLAST} \tag{6.6}$$
$$integrate(\mathbb{R}) = Q', \quad \text{where } Q' \in ELAST \tag{6.7}$$
$$replace(T, Q, Q') = T', \quad \text{where } T' \in ELAST \wedge T' \text{ is an AST} \tag{6.8}$$

The mapping $nmap$ transforms the Q query's <NAVIGATE> to a corresponding set of sub-trees $\mathbb{P}$ of the AST S. Then for each element of $\mathbb{P}$, the mapping $emap$ extracts its part relevant to <EXTRACT>. This yields a set of resulting sub-tress $\mathbb{R}$. Further, the set $\mathbb{R}$ of sub-trees is integrated via the mapping $integrate$ into a single AST sub-tree Q'. Finally, the target AST T containing the query Q gets modified into T' by replacing Q by Q'. The resulting T' has to be an AST in EPLANG-*.

Definition 2. The assimilation process of query Q defined in AST T and targeting side AST S is defined as a composition of mappings (6.5), (6.6), (6.7), and (6.8):

$$\begin{aligned} assimilation(T, S, Q) = replace(T, Q, \\ integrate(\\ emap(\\ nmap(S, <NAVIGATE>), \\ <EXTRACT>))) \end{aligned} \tag{6.9}$$

With regard to the mapping complexity, $nmap$ relies on the representation of a <NAVIGATE>. Obviously, interpretation of <NAVIGATE> is easy when it complies with the hierarchical namespace of constructs in a side-language. This is relatively simple to achieve when the side-*-language is XML-based, so that its constructs are inherently hierarchical. Nevertheless, <NAVIGATE> is expressed by the EPLANG-* means which do not have to comply with the side-language rules. If they do not, the sub-trees identified by <NAVIGATE> have to be transformed by $nmap$ into a final form $\mathbb{P}$ complied with the side-language. Fortunately, the Stratego framework allows

for very general AST restructuring. A penalty may be the complexity of the transformation rules, though.

While the mapping $emap$ which finds and extracts from $\mathbb{P}$ required information corresponding to <EXTRACT> is relatively simple, the integrate mapping is more complex. Obviously, assimilation of $\mathbb{R}$ into T in EPLang-* has to be based on mapping the concepts of a side-*-language to EPLang-*. This is specifically not simple when the result $\mathbb{R}$ of $emap$ is spread over elements of $\mathbb{P}$. This requires employing non-trivial iterative AST transformations as a part of $integrate$. Again, even though the Stratego framework allows for very general AST restructuring, a penalty might be the complexity of the transformation rules. Detailed specification of the $nmap$ and $emap$ mappings based on denotational semantics can be found in Appendix D.

The implementation of queries processing written with the help of the Stratego programming language is based on the strategy $eval$ which selects the demanded AST nodes (Listing 6.8). Its implementation directly corresponds to the denotational query semantics. It recursively traverses (line 9) the AST from the given node and looks for suitable nodes with help of the strategy $nmap$ (line 4). The strategy decides whether a node should be appended to the resulting set. It includes application of condition filter $apply-condition$ (line 22) which directly corresponds to the semantic definition of the function $Filter$. Lines 12-14 correspond to the function $Emap$ performing selection of desired values from the sub-trees selected by $nmap$ strategy.

Listing 6.8: Transformation strategies processing queries.

```
1  // query evaluation strategy
2  eval(|query) =
3      if <not(is-empty)> query then
4          where( !query => [head | tail] )
5          ; nmap(|head)
6          ; if <not(is-last)> tail then
7              map(\ Element(_,_, val) -> val \)
8              ; concat
9              ; eval(| tail)
10         else
11             // corresponds to emap mapping
12             map( \ Element(_, _, [Text(c)] ) -> Text(c) \ <+ id )
13             ; ( \[ Text(z)] -> Id(Text(z))\ <+ id )
14             ; ( \[ Lit(l)] -> Lit(l)\ <+ id )
15         end
16     end
17 // corresponds to nmap mapping
18 nmap(|e) =
19     switch id
20         case <?IdElement(Id(name), condition, operator)> e:
21             find-query-element-byname(|name)
22             ; apply-filter (|condition)
23         case <?IdElement(Id(name), index)> e:
24             /* ... */
25         otherwise: fail
26     end
```

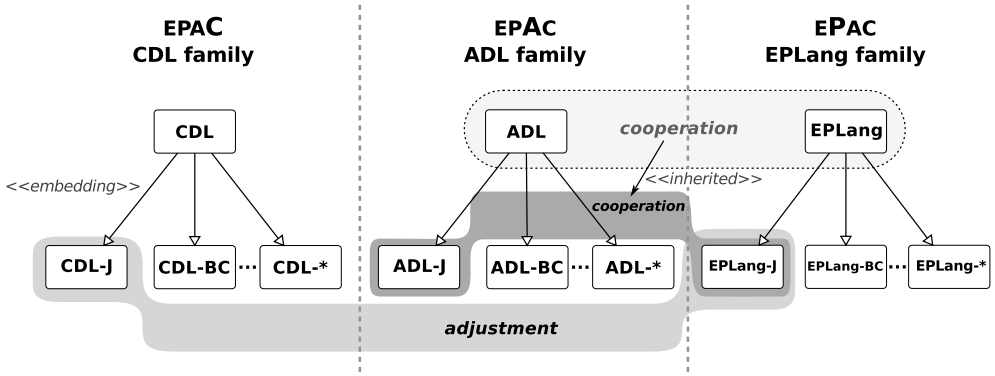


Figure 6.7: Interoperability among DSLs implemented by ECoGEN method.

6.5 Evaluation

In the light of the goals in Section 6.1.1, we evaluate the ECoGEN method from the perspective of:

(i) comparing the ECoGEN method with standard template-based code generation techniques,

(ii) DSL interoperability via MetaBorg tools and difference with classical MetaBorg method, and

(iii) the prospects of porting the ECoGEN method to other domains.

6.5.1 Comparison with a Standard Template-based Technique

As we mentioned in Section 6.1.1, template-based approach was originally used for specifying implementation of SOFA connector elements; however, such element generation proved to be inflexible not only in terms of modifications, but, first of all, of extensions. From implementation perspective, the code generator processed a code template which was composed of: (i) a "pure" textual template allowing only simple textual substitutions and (ii) "placeholders" to be interpreted by the internals of code generator (a hand-written placeholder interpretation class plugged into it). Overall, the template and interpretation of placeholders was the way EPAC specification was provided, while EPAC and EPAC specifications were available to the generator in the form of its internal classes created by a front-end tool.

In contrast, the EPLANG-based approach allows defining the whole EPAC specification via three DSLs, synergy of them makes it possible to generate an element's code via a number of AST transformations starting with its EPAC specification.

To illustrate the difference between the template and EPLANG-J based approaches, Appendix B shows the EPAC specification of the composite element in EPLANG-J which corresponds to the textual template shown in Listing 6.1. The method `initializeArchitecture` (Appendix A – lines 13-52) has similar functionality as the placeholder `%%GENERATE_ARCHITECTURE_INITIALIZATION_METHOD%%` in Listing 6.1, line 19. Obviously in the former case, the EPLANG-J constructs clearly, with much finer abstraction granularity indicate what actions have to be done with the EPAC and EPAC specifications to achieve architecture initialization. Furthermore,

the EPLANG-J EPAC specification determines with similar granularity the target code for provided interfaces (Appendix A, lines 55-74); in contrast, this is in the textual template hidden behind the placeholder `%%GENERATE_PROVIDED_PORTS_METHODS%%` (Listing 6.1, line 27). Naturally, the functionality specified in EPLANG-J is hidden inside the placeholder interpretation class which reads context information, finds an element's provided port type, and for each method included in the type has to generate code for call delegation to a corresponding sub-element port.

An important benefit of the ECOGEN method brought by a language definition via grammar embedding is the syntax correctness guaranteed at the source level. It means that the majority of syntactical errors in a code pattern can be detected directly in EPAC specification. On contrary, in template-based approach, syntactical errors are detected as late as code is generated and compiled.

As a case-study, consider a sub-set of connector elements for different communication styles in SOFA 2 [BHP06]. These are listed in Table 6.1, which compares the sizes of element specifications and implementations in both template-based and EPLANG-based approaches. For the former, two artifact sizes are indicated per each element: the size of the textual code template and of the placeholder interpretation class (each placeholder needs its own interpretation class, which cannot be shared among elements – see the motivation in Section 6.1.1). Here both the total class size and the size of its part which directly produces Java code are provided. For EPLANG approach, the size of EPAC specification and the size of Stratego transformation program (employed for all elements) are given. Along a similar line, both the total size of EPAC specification/Stratego program and the number of lines directly producing Java code are indicated. Here, the following should be emphasized:

(i) The actual context (EPAC specification) of an element practically does not influence any of these numbers, since it determines only the types of the interfaces/-ports of the element.

(ii) In a similar vein, the element architecture (EPAC specification) influences significantly only the numbers related to the composite element; the figures indicated in Table 6.1 hold for a composite element with two primitive sub-elements.

(iii) While a placeholder interpreter class has to be created for each element in the template-based approach, the Stratego program is achieving the similar goal by AST transformations common for all elements; a benefit is extensibility – adding a new element requires to provide only its EPAC specification, since the Stratego program depends only on EPLANG-J and side-languages' grammars, thus not being specific for an EPAC instance.

Table 6.1 also clearly illustrates that most of the complexity of the template-based approach lays in the placeholder interpretation which heavily manipulates with the EPAC and EPAC specifications. On the other hand, the EPLANG-J EPAC specification contains much more Java LOC than the pure EPLANG statements.

Overall, the size of EPLang specification including AST transformations is half of the size of textual templates and placeholder interpretation. As an aside, Table 6.1 indicates that the major part of the EPLANG-J specifications and Stratego AST transformations is Java language independent (compared to template-based approach, the ratio is (1551+496) / 7823). Specifically, in comparison to the template-based method,

this means the ECoGen method saves development effort, particularly in lines of code that need to be re-implemented, during porting to a new *-language. Further, it demonstrates that ECoGen method separates concerns (element code pattern, its architecture, and context) are clearly determined in standalone specification entities.

As far as porting to another *-language is concerned, porting to byte code was very smooth – it took only to modify the Stratego program for Java AST transformations (496 LOC were replaced by 630 LOC which, however, include support for the dedicated tool translating Java element ePac specification into byte code ePac). Naturally, porting to *e.g.*, C# would require in addition modifying the Java-specific ePac specifications (taking 1551 LOC).

Element	Template-based approach			EPLang-based approach				Resulting size of element in Java
	Textual code template	Placeholder interpretation in generator (LOC)		EPLang-J ePac (LOC)		Stratego program for AST transformation (LOC)		
		Total	Directly producing Java code	Total	Directly producing Java code	Total	Directly related to Java AST transformations	
Composite element	151	1134	37	320	280	2000	496	322
Local stub	207	686	20	216	180			219
Local skeleton	180	664	20	204	160			192
RMI stub	204	799	32	233	200			195
RMI skeleton	168	735	39	218	184			188
Logger element	142	686	20	159	130			162
Corba stub	206	841	20	241	211			213
Corba skeleton	210	810	24	233	206			201
Java dependent $\sum$	1468	6355	212	N/A	**1551**	N/A	**496**	
$\sum$	**1468**	**6355**		1824		2000		1692
Total	7823			3824				1692

Table 6.1: Template versus EPLang-based method
LOC – lines of code

6.5.2 DSL Interoperability via MetaBorg Tools

The cornerstones of the ECoGen code generation method are language embedding and assimilation via the MetaBorg method [BV08] based on combining language grammars and applying AST transformation rules. Using these means, the original MetaBorg method envisions embedding a dedicated "small" DSL into a general-purpose language. This embedding ends up by translating the DSL's statements,

when these are to be assimilated, into a general-purpose language. However, the ECoGen method employs the MetaBorg means differently:

(i) For the EPLang-* language the "embedding" and assimilation are parameterized by a target general-purpose language *. This way the ECoGen method considers the core EPLang language to be embedded into a set of target languages. Contrary to the MetaBorg method as applied in [BV08], [BGV05], the ECoGen method assumes restrictions of the target language making it "small" during the EPLang embedding process.

(ii) For assimilation of EPLang-* code, two additional "side DSLs" (ADL-*, CDL-*) are to be employed.

(iii) Each side DSL has its specific language core which is to be embedded into the *-language. Importantly, here "embedding" has a very different meaning: the side-language is embedded into a very small subset of general-purpose language. Thus, a "small" DSL is embedded into an even smaller language (technically, for example, in case of CDL-J a lot of Java grammar rules are marked as forbidden). Moreover, the assimilation of CDL-J to Java is done only indirectly via mappings into EPLang-J AST (as shown in Section 6.4.3).

6.5.3 Applying the EPLang Idea to Other Domains

The EPLang idea addresses interoperability among multiple DSLs while not explicitly requiring these DSLs to be declarative or imperative.

As another example requiring interoperability among several declarative DSLs, management of access control lists can be considered. In most cases such lists are written in a declarative DSL (WebDSL [GV08], Ponder [DDLS01]) which defines groups of users and their access rights to system artifacts. To control access, the DSL has to systematically refer other system entities which are also declared via dedicated DSLs (*e.g.*, pages, actions and their parameters). And here our approach can be applied. For example, access control for a web application specified with a dedicated DSL embedding EPLang could be specified in the form shown in Listing 6.9.

Listing 6.9: An example of ACL declaration based on the concept of queries.

```
1 $(for USER in users.user)
2     ${USER} has no access $end$
3 ${users.user(name="Joe")} can
4     operate ${actions.action(name="view")}
5     operate ${actions.action(name="edit")}
6     on ${pages.page(id="intro")}
```

Here, the concept of queries allows referencing users, pages, and actions without the need to explicitly integrate the access control DSL grammar with grammars of user/page/action languages. Hence,these DSLs would be less dependent on each other (*e.g.*, so far the DSL for access control contains dedicated constructs for referencing pages, actions, users, which could be easily eliminated by introducing a single concept of query where the <NAVIGATE> part would determine a cooperating DSL).

Technically, such porting of the EPLANG idea would require designing a new access control language embedding the EPLANG language and corresponding code generator assimilating the access rights specification into target code. Nevertheless, the queries evaluator engine itself can be reused as it depends only on the structure of DSLs' ASTs. Importantly, no interlinking of the DSLs' grammars would have to be considered.

As an example of requiring interoperability among declarative and imperative DSLs, the task of generating code from a textual representation of UML diagrams(*e.g.*, EMFText[10], TextUML toolkit[11]) and corresponding code patterns described via an imperative DSL similar to EPLANG could be considered. The example may include use-case, class, and state diagrams with corresponding code patterns interlinked via queries. For example, a query in class diagram code pattern could refer to a use-case diagram, extracting its details and producing the class's comments or constructing class's methods pre-/post- conditions to reflect the use-case. In a similar vein, the class diagram code pattern could query a state diagram to produce corresponding switch-case code reflecting transitions among states. Here, the implementation would be similar to the previous example of managing access control lists.

6.6 Related Work

The EPLANG idea relates to research in code generation and interoperability of DSLs. From the perspective of element code generation, its process typically involves two crucial tasks: (i) element specification and (ii) the actual code generation. The objective of (i) is to describe, in an abstract way, the required functionality of the element. This can include not only definition of code snippets, but also a structural and behavioral specification of the element. For element specification, several methods are employed, such as template-based methods (Acceleo [23], Xpand [11]), model-driven methods (Ecore [9], QVT [14]) based on various DSLs ([GHKV08, HV11]), and even their combinations ([RCGT09]).

The way actual code generation (*i.e.*, (ii)), is done, is tightly related to the form of element specification: templates are typically processed by a program substituting placeholders and unfolding simple macros [23, 11]. In the case of model-driven methods, textual element specification is converted to a model (T2M) which is further processed by M2M model transformations (QVT [14], ATL [JK06], Stratego [3]) and M2T transformations (Acceleo [23], Xpand [11]) to produce source code as the result. The published case-studies using these technologies include *e.g.*, JavaJava, JavaSwul, or JavaRegex [BGV05]. The toolchain for generating software connectors [RCGT09] is an example of combining previous two approaches; it follows model-driven approach by specifying various elements (infrastructure elements, CCM connectors) via UML MARTE system design language [15]. Conceptually, this approach follows the idea of the template-based method with placeholders as employed originally in SOFA ([Bur06], [BHP06]). Here, however, the interpretation of placeholders is defined by an UML MARTE model.

Interoperability of domain-specific languages has been identified as an emerging challenge of MDE [BCC+10, JVB+10]. Most of the existing approaches focus on

[10] `http://www.emftext.org/`
[11] `http://sourceforge.net/apps/mediawiki/textuml/`

formalization of relations between domain-specific languages, including coordination of language change propagation; this is mostly done by model transformations and aspect weaving [JVB+10]. In contrary to EPLANG idea, these approaches are based on interlinking the corresponding DSL grammars. For example, interoperable domain-specific languages are corner-stones of *WebDSL* [GHKV08] and *mobl* [HV11] approaches targeting development of web, resp. mobile, applications. These two approaches are close to our method. They are also implemented on the top of Stratego/XT toolset, and, furthermore, provide transparent integration into Eclipse platform with the help of Spoofax []. Both approaches integrate languages for architecture, data, access control, and user interface for producing executable applications via transformations. The *WebDSL* and *mobl* provide consistency checking which heavily relies on early error detection thanks to the statically defined interlinks among languages' grammars. Such strategy, however, requires management of interlinks and their coordination after grammar modifications. This makes the key difference to the DSL interoperability proposed by the EPLANG idea, where the languages are combined only at the level of ASTs (as the result of query evaluation). Hence, the proposed query-based interoperability does not statically restrict relations among languages, but the consistency of interlinks needs to be checked dynamically during queries processing.

Another key issue solved by the ECOGEN method and not considered by *WebDSL* and *mobl* is the porting a set of interoperable DSLs (ADL-*, CDL-*, EPLANG-*) to a new *-language (like porting from Java to C#). In such a case, the interlinks among grammars would not be of any help since they would have to be rewritten from scratch. Even though, porting in our case implies rewriting the AST transformations implementing queries, the process of adjusting DSL interoperability requires less effort compared to adjusting interlinks among grammars. This is mostly because AST provides a higher level of abstraction over the language grammars and because only the targets of queries are to be handled in this adjustment. As an aside in case of *WebDSL* and *mobl*, it is not clear if (and how) the generation process could be modified in order to generate code for a new platform (*e.g.*, producing Ruby or Scala web applications in case of *WebDSL*).

6.7 Conclusion

This chapter presents the ECOGEN method – a general control element generation approach based on AST transformation strategies and DSLs interoperability. The chapter explains the method of employing multiple element specifications defined in several DSL languages to generate an element implementation in a selected target language. The presented element specification languages comply with classical specification convections in the component-based systems and reflect element architecture, its context and implementation pattern. Further, the languages constitute a family of DSL languages which is parameterized by a target language. The chapter thoroughly analyses the interoperability among the languages and explains how it is realized via a dedicated language construct called query. The benefits of the ECOGEN method in comparison with classical template-based code generation techniques are demonstrated on a case study where control elements are generated in Java. The evaluation part of the chapter shows that the method mitigates development effort when a new element is introduced. From the perspective of future research, there is still a room

for ECoGen framework improvements comprising not only a new language families but also template pre-compilation in order to speed up code preparation. This could be achieved, *e.g.*, by allowing bytecode fragments in the code pattern specification. As to domain-specific languages interoperability, deeper formal analysis involving static reasoning about queries correctness at the level of DSL grammars and AST transformation rules would be also desirable.

Chapter 7

Evaluation

The objective of the chapter is to evaluate the proposed μSOFA method and the notion of interoperability in the context of the case-studies presented in Chapter 3. The chapter elaborates in depth the jPapaBench case-study – outlines the μSOFA transformation process, and stresses the parts which are affected by languages interoperability. The remaining two case-studies – RTSJ connectors and SOFA 2 runtime extension – adopt partially the same ideas of the configurable execution environment as the μSOFA method. Thus, for these two case-studies, the chapter brings only a discussion of interesting aspects with respect to the μSOFA method and interoperability.

7.1 jPapaBench

The jPapaBench case-study described in Section 3.3 prepares a control system of UAV reflecting three implementation technologies – plain Java, RTSJ, and SCJ. The case-study utilizes a simple flat ad-hoc component model whose realization utilizes the EE1 form of the execution environment – *i.e.*, manually written glue code which assembles components together. The section describes main aspects of realization jPapaBench with the help of the μSOFA method.

μSOFA Method Inputs In the context of the case-study, the input for the μSOFA method involves:

- *application assembly* – the assembly is composed of two kinds of primitive application components – static modules and active tasks. Each component has associated implementation in the Java language. The overall assembly corresponds to the architecture depicted on Figure 3.3 shown in Section 3.3.
- *NFRs* – jPapaBench specifies the following non-functional requirements:
  - **R1** a deployment scenario says the resulting system will be launched on single hardware unit;
  - **R2** each task has associated timing properties including period and priority;
  - **R3** in the case of RTSJ, memory allocation areas are specified for each tasks and module;

R4 SPI bus's internal buffer length needs to be configured to a given value;

R5 application life-cycle management is not required;

R6 the target implementation technology (for simplicity we include a technology among NFRs). The case-study targets three different implementation technologies (plain Java, RTSJ, SCJ). Each technology prescribes a different kind of (R6a) platform entry-point and (R6b) management of active components (*i.e.*, tasks).

- *aspects* – aspects corresponding to the stated NFRs are provided.

μSOFA Method Output For jPapaBench the methods produces a compiled code bundled into a jar file. The file can be directly launched on the top of a JVM.

Resolved aspects According to the NFRs and the selected technology a set of aspect is resolved:

A0 aspect resolves existing implementation of components (modules and tasks).

A1 aspect encapsulates all tasks into technology specific thread implementation (*e.g.*, `RealtimeThread` in case of the RTSJ).

A2 aspect injects the entry-point into a the platform infrastructure.

A3 aspect assembles the components.

A4 aspect configures the SPI bus.

A5 aspect compiles source code and bundles resulting binaries.

The relation among the aspects and non-functional requirements is shown in Table 7.1. The table also shows the impact of individual aspects.

Aspect	Corresponding NFRs	Modifies infrastructure	Modifies back-end
A0	implicit	NO	YES
A1	R2, R6b	NO	YES
A2	R5, R6a	YES	YES
A3	R6	NO	YES
A4	R4	YES	YES
A5	R6	NO	YES

Table 7.1: Impact of aspects.

7.1.1 Back-end: From EIM to System Realization

The back-end preparing realization of the jPapaBench case-study is composed of the following join-points:

- `fetch` The aspect A0 fetches the application components content.

- `adaptation` The aspect A1 adapts tasks implementation to inherit from the specified thread implementation.

- `generation` The aspect A2 generates entry-point which implements given technology specific code (*i.e.*, main method in case of plain Java, and RTSJ, `Mission` in case of SCJ). The aspect A3 generates code instantiating all modules, all tasks and passing references between them according to the infrastructure model represented by the model element `SystemAssembly`.

- `compilation` The aspect A5 compiles all the code including adjusted content of application components as well as generated entry-point and assembly code.

- `bundling` The aspect A5 also collects all compiled binaries and bundle them into jar file.

7.1.2 μSOFA dvantages and Disadvantages

In comparison to original manually written implementation, the μSOFA method mainly simplifies realization of changes in the jPapaBench architecture. If proper aspects are specified, the methods easily allows for changing NFRs which are automatically propagated into the system realization.

On the other hand, the preparation of reusable aspects is more complex than writing ad-hoc glue code assembling the components and reflecting NFRs.

Hence, there is a trade-off between well-defined and configurable infrastructure and system fast prototyping. If the system architecture and stated NFRs are expected to be changed frequently then the μSOFA method will be beneficial. However, if the system is required to be implemented from scratch without any reuse of existing aspects and the system is not intended to be changed, then the μSOFA method does not help. On the other hand, if a rich repository of aspects exists, it allows for fast system prototyping.

7.1.3 Role of Interoperability

In the context of the jPapaBench case-study, the μSOFA method incorporates code generation preparing the implementation of the (i) entry-point and (ii) glue code assembling components together. Regarding generation of entry-point, its implementation is defined by a simple template in a selected implementation technology. The template is only completed by a name of a class which assembles components together. Hence, with respect to Chapter 6, there is only a simple relation between code pattern prescribing entry-point implementation and a execution infrastructure model which stores actual names of generated classes.

The second case is more interesting. During generation code which glues all modules and tasks together, the code pattern describing the glue code has to typically traverse the execution infrastructure model. Hence, the interoperability between a language describing the code pattern and EIM is required. Furthermore, the code pattern benefits from the parameterized language family and corresponding code generation process as has been proposed in Chapter 6. The code pattern for the glue is parameterized by the selected implementation technology.

7.2 RTSJ Connectors

μSOFA method applied The RTSJ connectors case-study follows the same strategy as the previous section has described. However, there are two major differences: (i) the RTSJ connectors case-study utilizes a rigorous component system (*i.e.*, Fractal) as the target implementation technology and (ii) the aspects build the infrastructure of a binding according to binding's NFRs. To reflect the first difference, the micro-components are mapped to the Fractal components.

Role of Interoperability In the case-study, the binding is composed of a chain of interceptors which are generated. The generation in fact adopts the same scenario as Chapter 6 utilizes to generate software connector infrastructure. In this sense, the case-study and the software connector infrastructure share the same notion of interoperability. That means the code pattern describing desired interceptor implementation needs to cooperate with the execution infrastructure model and also with the Fractal component model.

7.3 SOFA 2 Runtime Extension

μSOFA method applied Originally SOFA 2 extension case-study has been performed to verify μSOFA concepts and ideas. Therefore, the SOFA 2 extension case-study described in Section 3.5 directly follows the μSOFA method. The case-study proposes two aspects corresponding to non-functional requirements – (i) support for component content implemented in a scripted language, (ii) support for scripted content updates. The main difference lies in the way aspects are weaved. In case of the case-study, weaving is encoded as a part of the SOFA 2 container which refines application components into a form of execution infrastructure model, generates bytecode-based implementation of interceptors, and directly instantiates the resulting model. In this respect, the SOFA 2 container can be considered as a limited realization of the μSOFA method with static back-end configuration which cannot be extended.

Role of Interoperability Since the case-study's aspects cope only with simple interceptors which only redirect calls, the interoperability is not considered in this case. The implementations of interceptors are simply generated by the parameterized ASM-based generators associated with the interceptors.

Chapter 8

Conclusion

The thesis has described and elaborated the concepts of the *meta-component system* which allows for producing a component system based on a set of requirements (*e.g.*, target application domain, non-functional requirements involving distribution, support of services). In this respect, the goals (G1)—(G3) stated in Chapter 1 have been fulfilled. The thesis has analyzed existing component systems and performed three case-studies resulting into a definition of the meta-component system, its structure, and the corresponding process of the preparation of new component system (the goal G1). As stated in the goals, the thesis focuses on the issues connected with deployment and runtime aspects of the meta-component system, which come out to be centered about two main topics: (i) relation among DSLs and their interoperability in the context of code generation and (ii) specification and preparation of the execution environment. While the former topic addresses the goal (G3) and additionally describes the parameterized family of DSL languages, the latter topic describes the μSOFA method which enables model-driven specification and preparation of configurable execution environment (the goal G2).

Key Achievements and Benefits While regular component systems are focused on a particular domain, the thesis has clarified a notion of the meta-component system and, based on the performed analysis and case-studies, it has discussed requirements and capabilities of the system.

In the scope of meta-component system, the thesis has elaborated in depth the execution environment as a crucial part of every component system. Here, the thesis has proposed a configurable model of execution environment, which is a refinement of application architecture and specified non-functional requirements. Furthermore, the thesis has designed a self-configurable transformation process preparing the execution environment; the structure of the process is influenced by the inputs including application architecture, non-functional requirements, and target implementation technology. In contrast to contemporary approaches including limited transformation processes, the self-configurability enables powerful extensibility.

The last achievement of the thesis is tightly connected to generation and adjusting of implementation elements which occur during execution environment preparation. The thesis has clarified (i) a concept of languages interoperability and (ii) a family of interoperable domain-specific languages which is parameterized by a general-

purpose language. Both these ideas are novel with respect to nowadays state-of-the-art.

To summarize, the thesis does not bring an overall realization of meta-component system. However, it has clarified two processes which play key roles in component-based development and generally in software development.

Future Work and Open Issues There are several open issues which would deserve to be elaborated in more details. The meta-component system comprises preparation of tailored tools which are not discussed in the thesis. That represents challenging software engineering tasks to automatically build tools, especially graphical modelers, according to domain requirements.

With regard to the execution environment, the proposed infrastructure describes only structural concerns. However, it neglects control, as well as data-flows. Therefore, it would be beneficial to integrate them into the infrastructure assembling process. Furthermore, there are still open issues connected to infrastructure development and execution, such as on-the-fly infrastructure debugging, runtime infrastructure reconfigurations, and providing proofs considering infrastructure timing (or memory allocation) properties.

Finally, as models interoperability is identified as a general open challenge [BCC+10, JVB+10], there are still open questions which need to be clarified. In the context of the thesis, it would be useful to specify parameterization of the proposed language family more formally with regard to involved language grammars. Furthermore, it would be nice to have static analysis tools which would identify queries which have no sense respecting the referenced language grammars.

Bibliography

[AJ06] J.S. Anderson and E.D. Jensen. Distributed Real-Time Specification for Java: a Status Report (digest). In *Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '06)*, pages 3–9, New York, NY, USA, 2006. ACM.

[BBS06] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[BCA+01] G.S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. M. Costa, H.A. Duran-Limon, T. Fitzpatrick, L. Johnston, R.S. Moreira, N. Parlavantzas, and K.B. Saikoski. The design and implementation of Open ORB 2. *IEEE Distributed Systems Online*, 2(6), 2001.

[BCC+08] T. Bureš, J. Carlson, I. Crnkovic, S. Sentilles, and A. Vulgarakis. ProCom – the Progress Component Model Reference Manual, version 1.0. Technical Report, Mälardalen University, June 2008.

[BCC+10] H. Brunelière, J. Cabot, C. Clasen, F. Jouault, and J. Bézivin. Towards model driven tool interoperability: bridging eclipse and microsoft modeling tools. In *Proceedings of the 6th European conference on Modelling Foundations and Applications*, ECMFA'10, pages 32–47, Berlin, Heidelberg, 2010. Springer-Verlag.

[BCL+06] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.B. Stefani. The Fractal Component Model and its Support in Java. *Software: Practice and Experience*, 36:1257 – 1284, 2006.

[BCM03] F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In Robert Meersman, Zahir Tari, and Douglas Schmidt, editors, *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 1226–1242. Springer Berlin / Heidelberg, 2003. 10.1007/978-3-540-39964-3_78.

[BGB+] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. The real-time specification for java 1.0.2. Available online: `http://www.rtsj.org/specjavadoc/book_index.html`.

[BGH+06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, volume 41, pages 169–190, October 2006.

[BGV05] M. Bravenboer, R. De Groot, and E. Visser. Metaborg in action: Examples of domain-specific language embedding and assimilation using stratego/xt. In *In Participants Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*. Springer Verlag, 2005.

[BHP06] T. Bureš, P. Hnětynka, and F. Plášil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *Proceedings of SERA 2006*, pages 40–48, Seattle, USA, August 2006.

[BHP+07] T. Bureš, P. Hnětynka, F. Plášil, J. Klesnil, O. Kmoch, T. Kohan, and P. Kotrč. Runtime support for advanced component concepts. *Software Engineering Research, Management and Applications, ACIS International Conference on*, 0:337–345, 2007.

[BP04] T. Bureš and F. Plášil. Communication style driven connector configurations. In C. V. Ramamoorthy, Roger Lee, and Kyung Whan Lee, editors, *Software Engineering Research and Applications*, volume 3026 of *Lecture Notes in Computer Science*, pages 102–116. Springer Berlin / Heidelberg, 2004.

[BPM04] I.D. Baxter, C. Pidgeon, and M. Mehlich. Dms: Program transformations for practical scalable software evolution. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 625–634, Washington, DC, USA, 2004. IEEE Computer Society.

[BRJ05] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[Bur06] T. Bureš. *Generating Connectors for Homogeneous and Heterogeneous Deployment*. PhD thesis, Department of Software Engineering, Mathematical and Physical Faculty, Charles University, Prague, 2006.

[BV08] M. Bravenboer and E. Visser. Models in software engineering. chapter Designing Syntax Embeddings and Assimilations for Language Libraries, pages 34–46. Springer-Verlag, Berlin, Heidelberg, 2008.

[BVGVEA05] P. Basanta-Val, M. Garcia-Valls, and I. Estevez-Ayres. Towards the integration of scoped memory in distributed real-time java. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0:382–389, 2005.

[BW03] A. Borg and A. Wellings. A real-time rmi framework for the rtsj. *Real-Time Systems, Euromicro Conference on*, 0:238, 2003.

[CBCP01] M. Clarke, G. Blair, G. Coulson, and N. Parlavantzas. An efficient component model for the construction of adaptive middleware. In Rachid Guerraoui, editor, *Middleware 2001*, volume 2218 of *Lecture Notes in Computer Science*, pages 160–178. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45518-3_9.

[CCL06] I. Crnkovic, M. Chaudron, and S. Larsson. Component-based development process and component lifecycle. *Software Engineering Advances, International Conference on*, 0:44, 2006.

[CCSV07] I. Crnkovic, M. Chaudron, S. Sentilles, and A. Vulgarakis. A classification framework for component models. In *Proceedings of the 7th Conference on Software Engineering and Practice in Sweden*, October 2007.

[CE00] K. Czarnecki and U.W. Eisenecker. *Generative programming - methods, tools and applications*. Addison-Wesley, 2000.

[CH04] H. Cervantes and R.S. Hall. Autonomous adaptation to dynamic availability using a service-oriented component model. *Software Engineering, International Conference on*, 0:614–623, 2004.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, July 2006.

[CHHP91] J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow. Txl: a rapid prototyping system for programming language dialects. *Comput. Lang.*, 16(1):97–107, January 1991.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Soft. Eng.*, 20(6), 1994.

[CL02] I. Crnkovic and M. Larsson. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[CN02] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, 2002.

[CSVC11] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M.R.V. Chaudron. A classification framework for software component models. *Software Engineering, IEEE Transactions on*, 37(5):593 –615, sept.-oct. 2011.

[Cza05a] K. Czarnecki. Mapping features to models: A template approach based on superimposed variants. In *GPCE 2005 – Generative Programming and Component Enginering. 4th International Conference*, pages 422–437. Springer, 2005.

[Cza05b] K. Czarnecki. Overview of generative software development. In Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, pages 97–97. Springer Berlin / Heidelberg, 2005. 10.1007/11527800_25.

[DDLS01] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, POLICY '01, pages 18–38, London, UK, UK, 2001. Springer-Verlag.

[DHT01] E.M. Dashofy, A. Van der Hoek, and R.N. T.aylor. A highly-extensible, xml-based architecture description language. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, WICSA '01, pages 103– , Washington, DC, USA, 2001. IEEE Computer Society.

[dJVV01] M. de Jonge, E. Visser, and J. Visser. Xt: A bundle of program transformation tools system description. *Electronic Notes in Theoretical Computer Science*, 44(2):79 – 86, 2001.

[EHL07] C. Escoffier, R.S. Hall, and P. Lalanda. ipojo: an extensible service-oriented component framework. In *Services Computing, 2007. SCC 2007. IEEE International Conference on*, pages 474 –481, july 2007.

[Eva03] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

[FPR00] M. Fontoura, W. Pree, and B. Rumpe. *The Uml Profile for Framework Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[FR03] M. Fleury and F. Reverbel. The JBoss extensible server. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Middleware '03, pages 344–373, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[GCW+02] T. Genßler, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, P. Müller, and C. Stich. Components for embedded software: the pecos approach. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '02, pages 19–26, New York, NY, USA, 2002. ACM.

[GHKV08] D.M. Groenewegen, Z. Hemel, L.C.L. Kats, and E. Visser. Webdsl: a domain-specific language for dynamic web applications. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA Companion '08, pages 779–780, New York, NY, USA, 2008. ACM.

[GME07] D. Goodman, M. Morrison, and B. Eich. *Javascript bible, sixth edition*. John Wiley & Sons, Inc., New York, NY, USA, 2007.

[GMW97] D. Garlan, R. Monroe, and D. Wile. Acme: an architecture description interchange language. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '97, pages 7–. IBM Press, 1997.

[GRE+01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[GS03a] J. Greenfield and K. Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 16–27, New York, NY, USA, 2003. ACM.

[GS03b] J. Greenfield and K. Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 16–27, New York, NY, USA, 2003. ACM.

[GS04] J. Greenfield and K. Short. *Software factories: assembling applications with patterns, models, frameworks, and tools*. Wiley Application Development Series. Wiley Pub., 2004.

[GV07] I. Groher and M. Voelter. Xweave: models and aspects in concert. In *Proceedings of the 10th international workshop on Aspect-oriented modeling*, AOM '07, pages 35–40, New York, NY, USA, 2007. ACM.

[GV08] D.M. Groenewegen and E. Visser. Declarative access control for webdsl: Combining language integration and separation of concerns. In D. Schwabe, F. Curbera, and P. Dantzig, editors, *ICWE*, pages 175–188. IEEE, 2008.

[HC01] G.T. Heineman and W.T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison Wesley Professional, 2001.

[HHKR89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdfreference manual. *SIGPLAN Not.*, 24(11):43–75, November 1989.

[HHL+09] T. Henties, J.J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek. Java for safety-critical applications. *Electronic Notes in Theoretical Computer Science*, 2009.

[HIPW05] S. Hissam, J. Ivers, D. Plakosh, and K. C. Wallnau. Pin component technology (v1.0) and its c interface. Technical Report, CarnegieMellon Software Engineering Institute, 2005.

[HKW+08] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, B. Hummel, M. Meisinger, and C. Pfaller. CoCoME - The Common Component Modeling Example. In Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors, *The Common Component Modeling Example*, volume 5153 of *Lecture Notes in Computer Science*, pages 16–53. Springer Berlin / Heidelberg, 2008.

[HP00] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *STTT*, 2(4), 2000.

[HT99] A. Hunt and D. Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[HV11] Z. Hemel and E. Visser. Declaratively programming the mobile web with mobl. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 695–712, New York, NY, USA, 2011. ACM.

[JHRS05] R. Johnson, J. Hoeller, T. Risberg, and C. Sampaleanu. *Professional Java development with the Spring Framework*. Programmer to programmer. Wiley Pub., 2005.

[JK06] F. Jouault and I. Kurtev. Transforming models with atl. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer Berlin / Heidelberg, 2006. 10.1007/11663430_14.

[JVB+10] F. Jouault, B. Vanhooff, H. Bruneliere, G. Doux, Y. Berbers, and J. Bezivin. Inter-dsl coordination support by combining megamodeling and model weaving. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2011–2018, New York, NY, USA, 2010. ACM.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[KHM+11] T. Kalibera, J. Hagelberg, P. Maj, F. Pizlo, B. Titzer, and J. Vitek. A family of real-time java benchmarks. *Concurrency and Computation: Practice and Experience*, 23(14):1679–1700, 2011.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0053381.

[KT08] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enablink Full Code Generation*. Wiley-IEEE Computer Society Press, 2008.

[KTG+06] I. Krechetov, B. Tekinerdogan, A. Garcia, C. Chavez, and U. Kulesza. Towards an Integrated Aspect-Oriented Modeling Approach for Software Architecture Design. In *Software Architecture Design. 8th Workshop on Aspect-Oriented Modelling (AOM.06), AOSD.06*, 2006.

[LQS04] M. Leclercq, V. Quéma, and J.-B. Stefani. Dream: a component framework for the construction of resource-aware, reconfigurable moms. In *Proceedings of the 3rd workshop on Adaptive and reflective middleware*, ARM '04, pages 250–255, New York, NY, USA, 2004. ACM.

[LRS+11] F. Loiret, R. Rouvoy, L. Seinturier, D. Romero, K. Sénéchal, and A. Plšek. An aspect-oriented framework for weaving domain-specific concerns into component-based systems. *Journal of Universal Computer Science*, 17(5):742–776, mar 2011. `http://www.jucs.org/jucs_17_5/an_aspect_oriented_framework`.

[LT09] K.-K Lau and F.M. Taweel. Domain-specific software component models. In *Proceedings of the 12th International Symposium on Component-Based Software Engineering*, CBSE '09, pages 19–35, Berlin, Heidelberg, 2009. Springer-Verlag.

[LU07] K.-K. Lau and V. Ukis. A study of execution environments for software components. In H.W. Schmidt, I. Crnkovic, G.T. Heineman, and J.A. Stafford, editors, *CBSE*, volume 4608 of *Lecture Notes in Computer Science*, pages 107–123. Springer, 2007.

[LW06] K.-K. Lau and Z. Wang. A survey of software component models. Available online: `http://www.cs.man.ac.uk/cspreprints/PrePrints/cspp38.pdf`, 2006. Second edition, Pre-print CSPP-38, School of Computer Science, The University of Manchester, May 2006.

[LW07] K.-K. Lau and Z. Wang. Software component models. *IEEE Trans. Softw. Eng.*, 33(10):709–724, 2007.

[LWWC12] P. Langer, K. Wieland, M. Wimmer, and J. Cabot. Emf profiles: A lightweight extension approach for emf models. *Journal of Object Technology*, 11(1):1–29, April 2012.

[Maa05] H. Maaskant. A Robust Component Model For Consumer Electronic Products. In *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, volume 3, pages 167–192. Springer Netherlands, 2005.

[MB05] V. Mencl and T. Bureš. Microcomponent-based component controllers: A foundation for component aspects. *Asia-Pacific Software Engineering Conference*, 0:729–737, 2005.

[McC76] T.J. McCabe. A complexity measure. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[MDT03] N. Medvidovic, E.M. Dashofy, and R.N. Taylor. The Role of Middleware in Architecture-Based Software Development. *International Journal of Software Engineering and Knowledge Engineering*, 13(4):367–393, 2003.

[MHS05] M. Mernik, J. Heering, and A.M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

[MM03] J. Miller and J. Mukerji. Mda guide version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[MMP00] N.R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 178–187, New York, NY, USA, 2000. ACM.

[Muc97] S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[MVG06] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, March 2006.

[NAD+02] O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. P. Black, P. O. Müller, C. Zeidler, T. Genssler, and R. Born. A component model for field devices. In *CD '02: Proceedings of the IFIP/ACM Working Conference on Component Deployment*, pages 200–209, London, UK, 2002. Springer-Verlag.

[NCS+06] F. Nemer, H. Cassé, P. Sainrat, J.P. Bahsoun, and M. De Michiel. Papabench: a free real-time benchmark. In *Proceedings of 6th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2006.

[OLKM00] R. Ommering, F. Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33:78–85, March 2000.

[Par76] D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2:1–9, 1976.

[PBVDL05] K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[PLMS08] A. Plšek, F. Loiret, P. Merle, and L. Seinturier. A component framework for java-based real-time embedded systems. In Valérie Issarny and Richard Schantz, editors, *Middleware 2008*, volume 5346 of *Lecture Notes in Computer Science*, pages 124–143. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-89856-6_7.

[Plš09] A. Plšek. *SOLEIL: An Integrated Approach for Designing and Developing Component-based Real-time Java Systems*. PhD thesis, Université des Sciences et Technologie de Lille – Lille I, 2009.

[PMS08] A. Plšek, P. Merle, and L. Seinturier. A real-time java component model. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0:281–288, 2008.

[PPK+11] P. Parra, O. R. Polo, M. Knoblauch, I. Garcia, and S. Sanchez. MICOBS: multi-platform multi-model component based software development framework. In *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*, CBSE '11, pages 1–10, New York, NY, USA, 2011. ACM.

[PRJ+03] J. Perez, I. Ramos, J. Jaen, P. Letelier, and E. Navarro. Prisma: towards quality, aspect oriented and dynamic software architectures. In *Quality Software, 2003. Proceedings. Third International Conference on*, pages 59 – 66, nov. 2003.

[Puf11] W. Puffitsch. Hard real-time garbage collection for a java chip multiprocessor. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 64–73, New York, NY, USA, 2011. ACM.

[RCGT09] A. Radermacher, A. Cuccuru, S. Gerard, and F. Terrier. Generating execution infrastructures for component-oriented specifications with a model driven toolchain: a case study for marte's gcm and real-time annotations. In *Proceedings of the eighth international conference on Generative programming and component engineering*, GPCE '09, pages 127–136, New York, NY, USA, 2009. ACM.

[RJB04] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[RZP+05] K. Raman, Y. Zhang, M. Panahi, J.A. Colmenares, R. Klefstad, and T. Harmon. RTZen: Highly Predictable, Real-Time Java Middleware for Distributed and Embedded Systems. In *Middleware 2005*, pages 225–248, December 2005.

[Sam97] J. Sametinger. *Software Engineering With Reusable Components*. Springer, 1997.

[SPDC06] L. Seinturier, N. Pessemier, L. Duchien, and T. Coupaye. A component model engineered with components and aspects. In Ian Gorton, George Heineman, Ivica Crnkovic, Heinz Schmidt, Judith Stafford, Clemens Szyperski, and Kurt Wallnau, editors, *Component-Based Software Engineering*, volume 4063 of *Lecture Notes in Computer Science*, pages 139–153. Springer Berlin / Heidelberg, 2006. 10.1007/11783565_10.

[SSK+06] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, M. Wimmer, and G. Kappel. A Survey on Aspect-Oriented Modeling Approaches. Technical report, Johannes Kepler University Linz, 2006.

[SVB+06] T. Stahl, M. Völter, J. Bettin, A. Haase, S. Helsen, and B. Von Stockfleth. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley Software Patterns Series. John Wiley, 2006.

[SWR+99] B. Shirazi, L.R. Welch, B. Ravindran, C. Cavanaugh, B. Yanamula, R. Brucks, and E. Huh. Dynbench: A dynamic benchmark suite for distributed real-time systems. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 1335–1349, London, UK, UK, 1999. Springer-Verlag.

[Szy02] C. Szyperski. *Component Software: Beyond Object-Oriented Programming (2nd Edition) (Hardcover)*. Addison-Wesley Professional, 2002.

[TAdM07] D. Tejera, A. Alonso, and M.A. de Miguel. RMI-HRT: Remote Method Invocation - Hard Real Time. In *Proceedings of the 5^{th} International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '07)*, pages 113–120, New York, NY, USA, 2007. ACM.

[Taw11] F.M. Taweel. *An Approach to the Definition of Domain-specific Software Component Models*. PhD thesis, School of Computer Science, The University of Manchester, 2011.

[TMS10] A. Tiberghien, P. Merle, and L. Seinturier. Specifying self-configurable component-based systems with fractoy. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves, editors, *Abstract State Machines, Alloy, B and Z*, volume 5977 of *Lecture Notes in Computer Science*, pages 91–104. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-11811-1_8.

[TPNV11] D. Tang, A. Plšek, K. Nilsen, and J. Vitek. A Static Memory Safety Annotation System for Safety Critical Java. Available online: `http://sss.cs.purdue.edu/projects/checker/RTSS11-extended-version.pdf`, 2011.

[VG07] M. Voelter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pages 233–242, Washington, DC, USA, 2007. IEEE Computer Society.

[Vis05] E. Visser. A survey of strategies in rule-based program transformation systems. *J. Symb. Comput.*, 40(1):831–873, July 2005.

[VPK05a] T. Vergnaud, L. Pautet, and F. Kordon. Using the aadl to describe distributed applications from middleware to software components. In *Ada-Europe*, pages 67–78, 2005.

[VPK05b] T. Vergnaud, L. Pautet, and F. Kordon. Using the aadl to describe distributed applications from middleware to software components. In *Proceedings of the 10th Ada-Europe international conference on Reliable Software Technologies*, Ada-Europe'05, pages 67–78, Berlin, Heidelberg, 2005. Springer-Verlag.

[WC10] D. Wampler and T. Clark. Guest editors' introduction: Multiparadigm programming. *Software, IEEE*, 27(5):20 –24, oct 2010.

[WCJW02] A. Wellings, R. Clark, D. Jensen, and D. Wells. A framework for integrating the real-time specification for java and java's remote method invocation. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0:0013, 2002.

[ZM11] C. Zimmer and F. Mueller. A fault observant real-time embedded design for network-on-chip control systems. Technical Report TR-2011-13, North Carolina State University. Dept. of Computer Science, 2011.

Author's References

[BHM09] T. Bureš, P. Hnětynka, and M. Malohlava. Using a product line for creating component systems. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 501–508, New York, NY, USA, 2009. ACM.

[BJM+11] T. Bureš, P. Ježek, M. Malohlava, T. Poch, and O. Šerý. Strengthening component architectures by modeling fine-grained entities. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pages 124 –128, sept. 2011.

[BMH08] T. Bureš, M. Malohlava, and P. Hnětynka. Using dsl for automatic generation of software connectors. *Composition-Based Software Systems (ICCBSS 2008), International Conference on*, 0:138–147, 2008.

[HPB+10] P. Hošek, T. Pop, T. Bureš, P. Hnětynka, and M. Malohlava. Comparison of component frameworks for real-time embedded systems. In Lars Grunske, Ralf Reussner, and Frantisek Plasil, editors, *Component-Based Software Engineering*, volume 6092 of *Lecture Notes in Computer Science*, pages 21–36. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-13238-4_2.

[KMBH11] J. Keznikl, M. Malohlava, T. Bureš, and P. Hnětynka. Extensible polyglot programming support in existing component frameworks. *Software Engineering and Advanced Applications, Euromicro Conference*, 0:107–115, 2011.

[KPMS10] T. Kalibera, P. Parizek, M. Malohlava, and M. Schoeberl. Exhaustive testing of safety critical java. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 164–174, New York, NY, USA, 2010. ACM.

[LPM+09] F. Loiret, A. Plšek, P. Merle, L. Seinturier, and M. Malohlava. Constructing domain-specific component frameworks through architecture refinement. *Software Engineering and Advanced Applications, Euromicro Conference*, 0:375–382, 2009.

[Mal12] M. Malohlava. jPapaBench – a realtime benchmark. Technical Report 2012/1, Charles University in Prague, Department of Distributed and Dependable Systems, 2012.

[MB08] M. Malohlava and T. Bureš. Language for reconfiguring runtime infrastructure of component-based systems. In *Proceedings of MEMICS 2008*, November 2008.

[MHB] M. Malohlava, P. Hnětynka, and T. Bureš. Sofa 2 component framework and its ecosystem. Extended abstract of the tutorial – accepted for publication in *Proceedings of the 9th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA)*, 2012.

[MPBH12] M. Malohlava, F. Plášil, T. Bureš, and P. Hnětynka. Interoperable domain-specific languages families for code generation. *Software: Practice and Experience*, 2012.

[MPL+08] M. Malohlava, A. Plšek, F. Loiret, P. Merle, and L. Seinturier. Introducing Distribution into a RTSJ-based Component Framework. In *2nd Junior Researcher Workshop on Real-Time Computing (JRWRTC 2008)*, pages 1–4, Rennes, France, 2008.

[PKH+11] T. Pop, J. Keznikl, P. Hosek, M. Malohlava, T. Bures, and P. Hnetynka. Introducing support for embedded and real-time devices into existing hierarchical component system: Lessons learned. In *Software Engineering Research, Management and Applications (SERA), 2011 9th International Conference on*, pages 3 –11, aug. 2011.

[PLM12] A. Plšek, F. Loiret, and M. Malohlava. Component-oriented development for real-time java. In M. Teresa Higuera-Toledano and Andy J. Wellings, editors, *Distributed, Embedded and Real-time Java Systems*, pages 265–292. Springer US, 2012. 10.1007/978-1-4419-8158-5_11.

[PPO+12] T. Pop, F. Plášil, M. Outlý, M. Malohlava, and T. Bureš. Property networks allowing oracle-based mode-change propagation in hierarchical components. In *Proceedings of the 15th international ACM Sigsoft symposium on Component based software engineering*, CBSE '12, New York, NY, USA, 2012. ACM. Accepted for publication.

Web References

[1] Paparazzi: The free autopilot. http://paparazzi.enac.fr/, 2010.

[2] Sonar. Available online: http://www.sonarsource.org/, 2011.

[3] Stratego/XT transformation language and toolset. Available online: http://www.strategoxt.org/, 2012.

[4] OSGi Alliance. OSGi service platform core specification, release 4. Available online: http://www.osgi.org/Specifications/HomePage,, 2012.

[5] Apache Software Foundation. The apache velocity project. Available online: http://velocity.apache.org/, 2012.

[6] ASM, Java bytecode manipulation library, v. 3.1. Available online: http://asm.ow2.org/.

[7] AUTOSAR GbR. Autosar-technical overview. Technical Report, AUTOSAR GbR, 2005.

[8] Paul Boocock. JAMDA: Java Model Driven Architecture. Available online: http://sourceforge.net/projects/jamda, 2012.

[9] Eclipse Foundation. EMF Ecore. Available online: http://www.eclipse.org/modeling/emf/, 2012.

[10] Eclipse Foundation. JET – Java Emitter Templates. Available online: http://www.eclipse.org/emft/jet/, 2012.

[11] Eclipse Foundation. Xpand. Available online: http://wiki.eclipse.org/Xpand, 2012.

[12] Google. Google Guice. Available online: http://code.google.com/p/google-guice/.

[13] Object Management Group. Corba Component Model v4.0. Available online: http://www.omg.org/spec/CCM/, 2012.

[14] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) v1.1. Available online: http://www.omg.org/spec/QVT/, 2012.

[15] Object Management Group. UML Profile for MARTE: Modeling and Analysis of Real-time Embedded Systems v1.1. Available online: http://www.omg.org/spec/MARTE/, 2012.

[16] Iron Python. Available online: http://ironpython.net/.

[17] Java Expert Group. Java specification request JSR 302: Safety critical java technology. Available online: http://jcp.org/en/jsr/detail?id=302.

[18] The Jython Project. Available online: http://www.jython.org/.

[19] T. Künneth. Making scripting languages JSR-223-aware. Available online: http://today.java.net/pub/a/today/2006/09/21/making-scripting-languages-jsr-223-aware.html, September 2006.

[20] Archi Lab. Snu real-time benchmark suite. Available online: http://www.cprover.org/goto-cc/examples/snu.html.

[21] Microsoft. Component Object Model Technologies. Available online: http://www.microsoft.com/com/.

[22] NASA. Java Path Finder. Available online: http://babelfish.arc.nasa.gov/trac/jpf/, 2010.

[23] Obeo. Acceleo project. Available online: http://www.acceleo.org/, 2012.

[24] Oracle. Java Scripting API. Available online: http://java.sun.com/javase/6/docs/technotes/guides/scripting/.

[25] Oracle. Java Language Specification Java SE 5.0 / SE 6.0. Available online: http://docs.oracle.com/javase/specs/, 2012.

[26] Oracle. JavaBeans Specification,. Available online: http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html, 2012.

[27] Oracle. JSR 220: Enterprise JavaBeans^{TM} v3.0,. Available online: http://jcp.org/en/jsr/detail?id=220, 2012.

[28] Oracle. JSR 245: JavaServer Pages 2.1. Available online: http://jcp.org/en/jsr/detail?id=245, 2012.

[29] Oracle. JSR 314: JavaServer Faces 2.0. Available online: http://www.jcp.org/en/jsr/detail?id=314, 2012.

[30] D. D. Spinellis. ckjm – A Tool for Calculating Chidamber and Kemerer Java Metrics. http://www.spinellis.gr/sw/ckjm/, 2010.

[31] SpringSource. Spring Framework. Available online: http://www.springsource.org/.

[32] D. Wampler. Polyglot Programming. Available online: http://www.polyglotprogramming.com/.

Appendices

Appendix A

Example of EPAC for Composite Element

Listing A.1: The EPAC specification of the composite element in ADL-J.

```
<!-- Black-box view of the element -->
<element-type>
  <name>rpc-client-unit</name>
  <port name="call" role="provided" />
  <port name="line" role="remote" />
</element-type>
<element-type>
  <name>logger</name>
  <port name="in" role="provided" />
  <port name="out" role="required" />
</element-type>
<element-type>
  <name>stub</name>
  <port name="call" role="provided" />
  <port name="line" role="remote" />
</element-type>

<!-- Glass-box view of the element -->
<element-architecture>
  <name>LoggedClientUnit</name>
  <type>rpc-client-unit</type>

  <inst  name="logger" type="logger"/>
  <inst  name="stub" type="stub"/>

  <binding port1="call" element2="logger" port2="in"/>
  <binding element1="logger" port1="out" element2="stub" port2="call"/>
  <binding element1="stub" port1="line" port2="line"/>

</element-architecture>
```

Appendix B

Example of EPAC for Composite Element

Listing B.1: EPAC specification of a composite element in EPLANG-J.

```
1  element composite_default {
2    /* Constructor */
3    public $query{:classname}(ConnectorUnit parentUnit, boolean isTopLevel)
4     throws ElementLinkException {
5      this.parentUnit = parentUnit;
6      this.isTopLevel = isTopLevel;
7      dcm = DockConnectorManagerHelper.getDockConnectorManager ();
8
9       initializeArchitecture    () ;
10   }
11
12   /* Composite element's methods */
13   void  initializeArchitecture  ()  throws ElementLinkException {
14     subElements = new Element[$query{elements:element#count}];
15
16     try  {
17       /* create sub-elements */
18       $set i  = 0$
19       $foreach(ELEMENT in $query{elements:element})$
20         subElements[${i}] =
21           new ${ELEMENT.class as
                 (org.objectweb.dsrg.connector.ConnectorUnit,boolean)}(parentUnit, false);
22       /* remember ELEMENT index in a dictionary */
23       $set el[ELEMENT.name] = i$
24       $set i  = i  + 1$
25       $end$
26
27       /* create bindings */
28       $foreach(BINDING in $query{bindings:binding})$
29         $if  (BINDING.type == "BINDING") $
30         ((ElementLocalClient) subElements[${el[BINDING.from.element.name]}])
31           .bindElPort("${BINDING.from.port}",
32                       ((ElementLocalServer) subElements[${el[BINDING.to.element.name]}])
33                           .lookupElPort("${BINDING.to.port}")
34           ) ;
35         $end$
```

```
36        $end$
37      } catch (Exception e) {
38        throw new ElementLinkException(e);
39      }
40
41      /* boundings to remote references */
42      remoteTargetRefs = new RemoteRefBundle[$query{ports:port(type=REMOTE)#count}];
43
44      $set i = 0$
45      $foreach(REMOTE_PORT in $query{ports:port(type=REMOTE)})$
46        remoteTargetRefs[${i}] = null;
47        $set ref[REMOTE_PORT.name] = i$
48        $set i = i + 1$
49      $end$
50
51      distributeReconfigurationHandler();
52    }
53
54    // implements provided port as delegation
55    $foreach(PORT in $query{ports:port(type=PROVIDED)})$
56    implements interface ${PORT.type} {
57      method template {
58        // get an interface
59        Object target = ((ElementLocalServer)
              subElements[${el[PORT.boundedTo.element.name]}])
              .lookupElPort("${PORT.boundedTo.port}");
60
61        // call the method
62        ${method.declareReturnValue}
63        Object context = CallHelper.getCallContext();
64        $append(method.parameters, context)
65
66        $if (${method.returnVar})
67         ${method.returnVar} =
68           (($PORT.type}) target.${method.name}($implode(method.parameters));
69        $else$
70           (($PORT.type}) target.${method.name}($implode(method.parameters));
71        $end$
72      }
73    }
74  $end$
75  }
```

Appendix C

EPLang-BC Example

Listing C.1: The EPAC specification of *RMIStub* element in EPLANG-BC.

```
1  element rmi_stub {
2    // delegation target
3    protected $query{ports.port(name=line):type} target;
4
5    // input interface
6    implements port $query{ports.port(name=call):type} {
7      method template {
8        ${method.declareReturnValue}
9        Object context = CallHelper.getCallContext();
10
11       $if (${method.returnVar})
12         LABEL(0);
13         NEW("java.lang.StringBuilder");
14         DUP();
15         INVOKESPECIAL("java.lang.StringBuilder", "<init>", "void()");
16         ASTORE(1);
17         LABEL(1);
18         ALOAD(1);
19         LDC(S("${INFO}"));
20         INVOKEVIRTUAL("java.lang.StringBuilder",
21           "append",
22           "java.lang.StringBuilder(java.lang.String)"
23         );
24         POP();
25         LABEL(2);
26         ALOAD(1);
27         INVOKEVIRTUAL("java.lang.StringBuilder",
28           "toString",
29           "java.lang.String()");
30         ARETURN();
31       $else$
32         LABEL(0);
33         NEW("java.lang.StringBuilder");
34         DUP();
35         INVOKESPECIAL("java.lang.StringBuilder", "<init>", "void()");
36         ASTORE(1);
37         /* ... */
38       $end$
```

```
39          }
40      }
41  }
```

Appendix D

Denotational Semantics of Queries

The definition of query semantics consists of a definition of query abstract syntax and its interpretation in the context of AST. The abstract syntax of a query is defined as follows:

$$
\begin{aligned}
QUERY &:= \bot \mid NAVIGATE\ ":"\ EXTRACT \\
NAVIGATE &:= LNAVIGATE \mid LNAVIGATE\ "."\ NAVIGATE \\
LNAVIGATE &:= ID \mid ID"("CONDITION")" \\
CONDITION &:= ID\ "="\ ID \\
EXTRACT &:= "name" \mid "type" \mid "port" \mid "binding" \mid \ldots \\
ID &:= [:\ alphanum\ :]+
\end{aligned}
$$

The following functions denotes interpretation of a query in the context of an AST node `x`:

$$Eval: \quad QUERY \rightarrow node_{root} \rightarrow Set(node_{root}) \tag{D.1}$$

$$Eval[\bot, x] = \emptyset \tag{D.2}$$

$$Eval[n : e, x] = \{x_2 | x_1 \in Nmap[n, x] \wedge x_2 \in Emap[e, x_1]\} \tag{D.3}$$

$$Emap: \quad EXTRACT \rightarrow node_{root} \rightarrow Set(node_{root}) \tag{D.4}$$

$$Emap[e, x] = \{y | x \rightarrow y \wedge TypeOf(y) = e\} \tag{D.5}$$

$$Nmap: \quad NAVIGATE \rightarrow node_{root} \rightarrow Set(node_{root}) \tag{D.6}$$

$$Nmap[id, x] = \begin{cases} \{x\} \Leftrightarrow TypeOf(x) = id \\ \emptyset \end{cases} \tag{D.7}$$

$$Nmap[id(c), x] = \begin{cases} \{x\} \Leftrightarrow TypeOf(x) = id \wedge Filter[c, x] \\ \emptyset \end{cases} \tag{D.8}$$

$$Nmap[id.tail, x] = \{y | TypeOf(x) = id \wedge y \in \bigcup_{x \rightarrow d} Nmap[tail, d]\} \tag{D.9}$$

$$Nmap[id(c).tail, x] = \{y | Filter[c, x] \wedge y \in Nmap[id.tail, x]\} \tag{D.10}$$

$$Filter: \quad CONDITION \rightarrow node \rightarrow Boolean \tag{D.11}$$

$$Filter[attr = val, x] = attr \in Attr(x) \wedge Val(attr)_x = val \tag{D.12}$$

$$Attr(x) = \text{attributes of node } x \tag{D.13}$$

$$Val(attr, x) = \text{value of attribute } attr \text{ in node } x \tag{D.14}$$

$$TypeOf(x) = \text{type of node } x \tag{D.15}$$

The equations (1)-(15) define interpretation of the query abstract syntax rules. In principle, a query refers to the part of side-AST, where the required information is stored. As its signature (1) indicates, the interpretation function `Eval` operates upon the query and the side-AST determined by its root given as another `Eval` argument. The result of `Eval` is a set of AST sub-trees with the roots reachable from the root. Obviously, for an empty query, `Eval` returns empty set (2). The equation (3) determines that to process a query with a navigate `n` and extract `e`, a combination of functions `Nmap` and `Emap` is utilized. First, the function `Nmap` is applied (as specified by (6)-(10)) to navigate along `n` part of the query starting from the root `x`; to the resulting sub-trees determined by their roots, `Emap` is applied with argument extract `e`. The result of `Emap` is a set of AST sub-trees determined by their roots. In (5), the function `Emap` evaluates the required `e` and returns the set of direct children of root `x` which are of the type `e` (with help of function `TypeOf` – 14). For `Nmap`, equations (7)-(10) determine semantics with respect to the hierarchical structure of navigation part. If the navigate part contains a condition ((8), (10)), then the `Filter` function (11) is applied to decide whether a node satisfies a given condition. Currently, only conditions regarding values of AST node attributes are considered (12)-(14).