ACM/IEEE 18th International Conference on
Model Driven Engineering Languages and Systems

September 28, 2015 • Ottawa (Canada)

2nd International Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp) 2015

Workshop Proceedings

Federico Ciccozzi, Patrizio Pelliccione, Etienne Borde (Editors)

Published in September 2015 v2.0

Editors' addresses:

Federico Ciccozzi
School of Innovation, Design and Engineering – Mälardalen University (Sweden)

Patrizio Pelliccione
Chalmers – Gothenburg University (Sweden)

Etienne Borde
Telecom ParisTech (France)

Organizers

Program Committee

Table of Contents

Preface

The design of modern software systems requires support capable of properly dealing with their ever-increasing complexity. In order to account for such a complexity, the whole software engineering process needs to be rethought and, in particular, the traditional division among development phases to be revisited, hence moving some activities from design time to deployment and runtime. Model-Driven Engineering (MDE) and Component-Based Software Engineering (CBSE) can be considered as two orthogonal ways of reducing development complexity: the former shifts the focus of application development from source code to models in order to bring system reasoning closer to domain-specific concepts; the latter aims to organize software into encapsulated independent components with well-defined interfaces, from which complex applications can be built and incrementally enhanced.

When exploiting these development approaches, numerous different modelling notations and consequently several software models are involved during the software life cycle. On the one hand, effectively dealing with all the involved models and heterogeneous modelling notations that describe software systems needs to bring component-based principles at the level of the software model landscape hence supporting, e.g., the specification of model interdependencies, and their retrieval, as well as enabling interoperability between the different notations used for specifying the software. On the other hand, MDE techniques must become part of the CBSE process to enable the effective reuse of third-party software entities and their integration as well as, generally, to boost automation in the development process.

An effective interplay of CBSE and MDE approaches could help in handling the intricacy of modern software systems and thus reducing costs and risks by: (i) enabling efficient modelling and analysis of extra-functional properties, (ii) improving reusability through the definition and implementation of components loosely coupled into assemblies, (iii) providing automation where applicable (and favourable) in the development process. In the last fifteen years, such a cooperation has been recognized as extremely promising; tools and frameworks have been developed for supporting this kind of integrated development process. Nevertheless, when exploiting interplay of MDE and CBSE, clashes arise due to misalignments in the related terminology but also, and more importantly, due to differences in some of their basic assumptions and focal points.

The goal of the workshop on Model-Driven Engineering for Component-Based Software Systems 2015 (ModComp'15) was to gather researchers and practitioners to share opinions, propose solutions to open challenges and generally explore the frontiers of collaboration between MDE and CBSE. ModComp'15 aimed at attracting contributions related to the subject at different levels, from modelling to analysis, from componentization to composition, from consistency to versioning; foundational contributions as well as concrete application experiments were sought.

The workshop was co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages & Systems, and represented a forum for practitioners and researchers. We received twelve papers out of which six papers were selected for inclusion in the proceedings. The accepted papers covers many different forms of intertwining of MDE and CBSE including, but not limited to:

– model integration;
– model transformations for analysis and code generation;
– modeling component interaction and component behaviors;
– model interoperability;

– modeling languages for components.

This was the second edition of the workshop and the high attention received once again in terms of submissions proves that the topics are relevant both in practice and in theory of model-driven engineering of component-based software systems. Thus, we would like to thank the authors – without them the workshop simply would not have taken place – and the program committee for their hard and precious work.

September 2015 Federico Ciccozzi, Patrizio Pelliccione and Etienne Borde

Keynote

Model-driven Analytics with Models@run.time: The Case of Cyber-Physical-Systems

Yves Le Traon

University of Luxembourg

Bits and bytes are governing an increasing number of areas in our lives and businesses. The exploration and simulation of what might happen and which action can be triggered is a fundamental part of intelligent systems such as smart grids, smart buildings, smart homes and any cyber-physical system. This new intelligence is supported by machine learning algorithms that, based on past data and runtime data, model the behavior of the system to predict its evolution. Recommendation systems, autonomous decision-support, prescriptive simulations have to be both scalable and highly accurate at runtime. It is paramount to develop new decision support services that should (at least partly) relieve the users from the overwhelming load of information and the growing number of decisions to be taken in time. In that perspective, model-driven engineering offers a bridge between the knowledge of experts who best know which data are relevant, and the monitoring and control of software components and sensors. The presentation is about how MDE, and specifically models@run.time, may become an enabler for designing and deploying easily domain-specific, scalable analytics for heterogeneous sources of timed data. Some problems still have to be solved and I will introduce some of them. Cyber-physical systems continuously analyze their surrounding environment and internal state, which together we refer to as the context of a system, in order to adapt itself to varying conditions. To yield accurate predictions, such systems not only rely on single numerical values, but also need structured data models aggregated from different sensors. Therefore, building appropriate context representations is of key importance. Over the past few years the models@run.time paradigm has shown the potential of models to be used not only at design-time but also at runtime to represent the context of cyber-physical systems, to monitor their runtime behavior and reason about it, and to react to state changes. However, reasoning about such contexts is a complex and time critical activity that needs to leverage near real-time analytics together with big data methods to quickly process the massive amount of data measured by these systems. Current modeling techniques do not allow to face all needed features for reasoning, such as distribution, large-scale and near real-time response time. In this talk I present two concepts that might push the limits of models@run.time for near-real time analytics a little further: 1) stream-based, distributed models and 2) historized models. I will present our results based on a real application on a smart grid scenario in joined work with the main electrical grid provider of Luxembourg.

Yves Le Traon *is professor at University of Luxembourg, in the Faculty of Science, Technology and Communication (FSTC). His domains of expertise are related software engineering and software security, with a focus on software testing and model-driven engineering. He received his engineering degree and his PhD in Computer Science at the "Institut National Polytechnique" in Grenoble, France, in 1997. From 1998 to 2004, he was an associate professor at the University of Rennes, in Brittany, France. During this period, Professor Le Traon studied design for testability techniques, validation and diagnosis of object-oriented programs and component-based systems. From 2004 to 2006, he was an expert in Model-Driven Architecture and Validation in the EXA team (Requirements Engineering and Applications) at "France Télécom R&D" company. In 2006, he became professor at Telecom Bretagne (Ecole Nationale des Télécommunications de Bretagne) where he pioneered the application of testing for security assessment of web-applications, P2P systems and the promotion of intrusion detection systems using contract-based techniques. He is currently the head of the Computer Science Research Unit at University of Luxembourg. He is a member of the Interdisciplinary Centre for Security, Reliability and Trust (SnT), where he leads the research group SERVAL (SEcurity Reasoning and VALidation). His research interests include software testing, model-driven engineering, model based testing, evolutionary algorithms, software security, security policies and Android security. The current key-topics he explores are related to Internet of things (IoT) and Cyber-Physical Systems (CPS), Big Data (stress testing, multi-objective optimization, analytics, models@run.time), and mobile security and reliability. He is author of more than 140 publications in international peer-reviewed conferences and journals.*

Challenges for Model-Integrating Components

Mahdi Derakhshanmanesh
University of Koblenz-Landau
Institute for Software Technology
Koblenz, Germany
Email: manesh@uni-koblenz.de

Jürgen Ebert
University of Koblenz-Landau
Institute for Software Technology
Koblenz, Germany
Email: ebert@uni-koblenz.de

Marvin Grieger
University of Paderborn
Department of Computer Science
Paderborn, Germany
Email: marvin.grieger@uni-paderborn.de

***Abstract*—Model-Integrating Software Components (MoCos) use models at runtime as first class entities within components to build flexible and adaptive software systems. Building such systems requires to design and implement the required domain-specific modeling languages. Insufficient design and realization of modeling languages raises the risk that they may not be optimized for their later use. Although various works on the use of models at runtime exist, they do not address the engineering of modeling languages to be used in software components at runtime. In this paper, we introduce the idea of Comprehensive Language Models (CLMs) which explicitly considers modeling language engineering as a part of the development of component based software systems. This is achieved by extending the modeling language specification, e.g., by a set of interfaces for models which are used for accessing models at runtime. We illustrate an initial solution concept along an insurance sales app case study on Android based on which we derive a set of key challenges for the community.**

I. Introduction

Models are no longer just used to design software but can become an integrated part of it, i.e., (executable) models and code coexist at runtime with equal rights. In previous works [1], [2], we proposed *Model-Integrating Software Components* (MoCos) as a concept for the design and development of such *model-integrating software systems*. Software engineers can choose to realize some parts of a system programmatically in code, while other parts are kept as models. No code is generated from the models but they are used at runtime. This concept yields flexible well-performing software that can be easily and systematically monitored, analyzed and modified.

The MoCo-approach combines models described using arbitrary *Domain-Specific Modeling Languages* (DSMLs) and embeds them within software components following a tailorable component design pattern that guides software engineers: the *MoCo Template*. It is depicted in Figure 1 and briefly introduced, next.

A. Model-Integrating Software Components

Each MoCo can have ports (`PFunction`, `PManage`) that are wired to the internal implementation, which is either encoded by a programming language (`MoCoCode`) or by a modeling language (`MoCoModel`). Conceptually, there may be sets of languages used on either side. In practice, a base technology such as the Java Virtual Machine (JVM) and its byte code format acts as a unifier. Programming languages are used on the code side, e.g., Java, and different – potentially integrated – DSMLs are used on the model side. Both constituents of a MoCo are encapsulated using *interfaces*. Optionally, a smart `Mediator` can manage any redirection from and to the MoCo's ports as well as communication between the model and code parts.

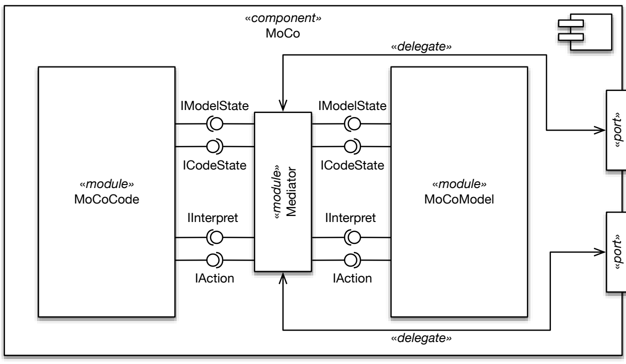


Fig. 1. Internal View on the MoCo Template (see [1])

The *expected advantages* of the MoCo approach are: (i) *enhanced flexibility* because the system and its individual components can be observed using model queries, can be modified by adapting models using an editor or model transformations and can be executed using *model interpreters* [3], (ii) *support of separation of concerns* because each model targets a concern, (iii) *understandability and maintainability* because models are assumed to be easier to understand and easier to handle than code, (iv) *self-documentation* because a well designed modeling language is assumed to be a documentation and (v) *no synchronization problem* because there is no redundancy between model and code unless it is introduced willfully, e.g., to realize reflection.

B. Research Problem

An essential difference between component-based and model-integrating software is the use of various *models at runtime* (not just reflective *models@run.time* [4]). Therefore, developing model-integrating software systems includes choosing existing modeling languages or designing and implementing adequate DSMLs. In fact, it must be possible to introduce new DSMLs easily and quickly to realize certain parts of a system as a model. In turn, the use of models of a given DSML requires support for the following core activities: (i) *building models*, e.g., using an application programming interface or an editor, (ii) *binding models into components* as building blocks, e.g., following the MoCo Template and (iii) *using models*, e.g., querying, transforming and interpreting them. These activities can only be supported if there is a powerful *technological space* [5] for modeling languages and models, which provides all relevant capabilities needed. In the context of MoCos, such a technological space needs to work together with the respective *component execution platform* [6] or even be part of it.

There are examples for such technological spaces like the Eclipse Modeling Framework (EMF) [7] or JGraLab [8] that deliver acceptable support for language design, especially for syntax and constraints. Additionally, many research works use models at runtime in various ways and this specific topic is still a very relevant research area [9].

While different approaches and solutions for modeling and models at runtime already exist in isolation, we observe that they do not comprehensively address the required capabilities for designing new modeling languages that shall be an integrated part of a software system. Moreover, the *challenges* associated with designing DSMLs that support the *symbiosis* of models at runtime and code within software components have to be inspected.

This paper tackles the following *research problems* and their associated challenges:

(Q1) How to specify modeling languages comprehensively for generating adequate runtime support for them?

(Q2) What are challenges for modeling languages in the context of MoCos?

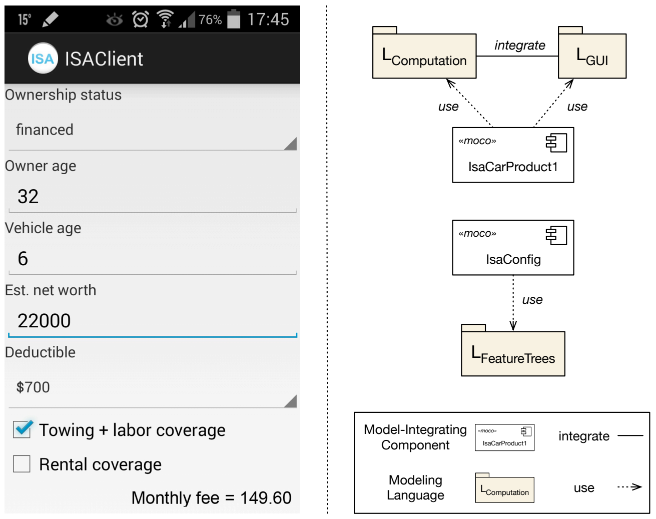
C. Contributions

In answering Q1, we propose that the introduction of a new DSML requires a *Comprehensive Language Model* (CLM), i.e., a specification of a modeling language to such an extent that all required activities on models are supported. We claim that each CLM should at least specify the following parts of DSML: (i) *syntax* (metamodel and constraints), (ii) *semantics* (dynamic state, constraints, state transitions) and (iii) *pragmatics* (at least facades [10]). We assume that a realization of a modeling language L_i will be derived from a CLM_i.

In answering Q2, we provide a description of selected *challenges* related to the seamless integration of models and code in software components, based on the *Insurance Sales App* (ISA) case study [1]. All in all, we aim to *raise awareness* for this topic and to initiate a fruitful discussion.

II. Running Example: Insurance Sales App

The *Insurance Sales App* (ISA) is a prototypical application for Google's Android mobile operating system. It has been developed to evaluate the feasibility of MoCos [1], [2]. It also serves as a *running example* throughout the rest of this paper. From the user's perspective, ISA's primary purpose is to support field staff in the insurance domain with their daily sales tasks. The system is built with MoCos, thus it


15° 76% 17:45

ISA ISAClient

Ownership status
financed

Owner age
32

Vehicle age
6

Est. net worth
22000

Deductible
$700

☑ Towing + labor coverage

☐ Rental coverage

Monthly fee = 149.60

Fig. 2. MoCo-based ISA Client App and an Architecture Excerpt

is dynamically extensible at the component level and single components' internals can be also monitored and modified at runtime. For example, insurance fee formulas are adapted, based on the current physical location of a customer. A screenshot of ISA is shown in Figure 2 (left), illustrating one of the views of the Graphical User Interface (GUI) specific to the *car insurance product*.

ISA's architecture consists of a mix of pure Java libraries and MoCos, i.e., a certain part of the running software is encoded in models that are used at runtime, e.g., by querying and transforming them. An excerpt is depicted in Figure 2 (right). The specific modeling languages used represent (i) *feature trees* (L_f) for architectural reconfiguration, (ii) *computation* (L_c) for insurance fee formulas and (iii) *graphical user interfaces* (L_g) for data presentation and user input capturing.

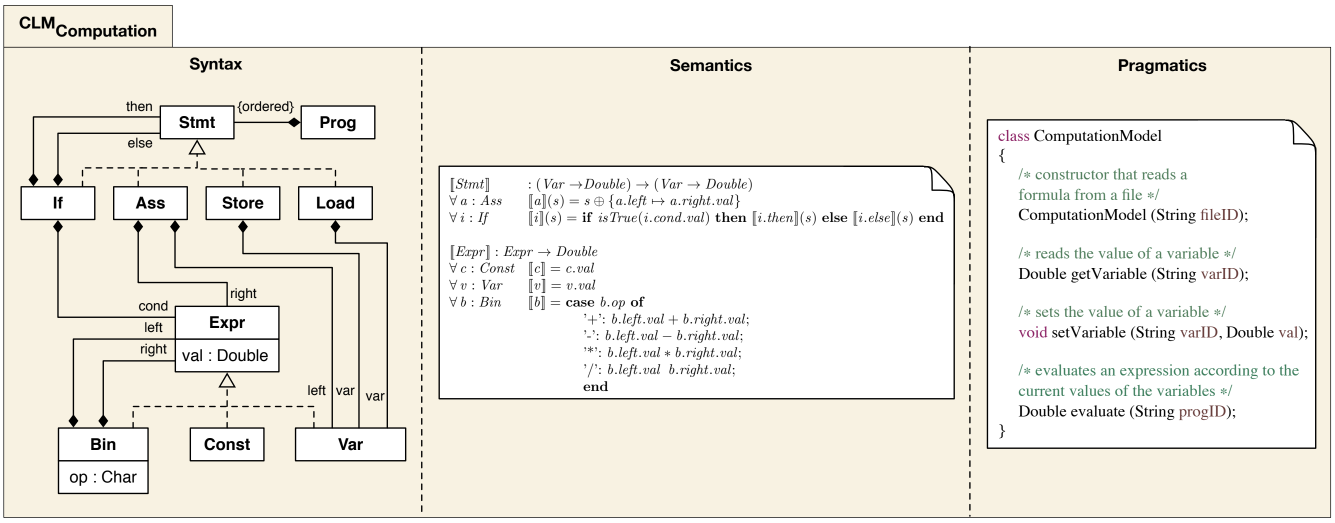
Regarding implementation, all MoCos conform to the structure proposed by the MoCo Template. For components, we used *OSGi's* [11] dynamic component technology, code was written in Java and models were developed in *JGraLab* [8]. The base execution platform is the Java Virtual Machine.

A. Comprehensive Language Model for L_c

As a clarification for what exactly a *Comprehensive Language Model* (CLM) is, we give an example in the context of the ISA case study. More concretely, we describe the CLM for L_c in the following and sketch how it relates to L_g. This background knowledge is required to understand some of the challenges described later in this paper.

CLM_c, i.e., the CLM that fully specifies the modeling language L_c, is depicted in Figure 3. It consists of three major parts: syntax, semantics and pragmatics.

1) Syntax Specification: L_c's *syntax* is specified using a UML-style metamodel and mostly represents the *static structure*. The language represents programs (`Prog`) consisting of statements (`Stmt`). There are special statements such as conditional (`If`), an assignment (`Ass`) and further specific



$$
\begin{array}{ll}
[\![Stmt]\!] & : (Var \rightarrow Double) \rightarrow (Var \rightarrow Double) \\
\forall\, a : Ass & [\![a]\!](s) = s \oplus \{a.left \mapsto a.right.val\} \\
\forall\, i : If & [\![i]\!](s) = \textbf{if } isTrue(i.cond.val) \textbf{ then } [\![i.then]\!](s) \textbf{ else } [\![i.else]\!](s) \textbf{ end} \\
 & \\
[\![Expr]\!] : Expr \rightarrow Double & \\
\forall\, c : Const & [\![c]\!] = c.val \\
\forall\, v : Var & [\![v]\!] = v.val \\
\forall\, b : Bin & [\![b]\!] = \textbf{case } b.op \textbf{ of} \\
 & \quad \text{'+'}: b.left.val + b.right.val; \\
 & \quad \text{'-'}: b.left.val - b.right.val; \\
 & \quad \text{'*'}: b.left.val * b.right.val; \\
 & \quad \text{'/'}: b.left.val \;\; b.right.val; \\
 & \quad \textbf{end}
\end{array}
$$

```
class ComputationModel
{
    /* constructor that reads a
    formula from a file */
    ComputationModel (String fileID);

    /* reads the value of a variable */
    Double getVariable (String varID);

    /* sets the value of a variable */
    void setVariable (String varID, Double val);

    /* evaluates an expression according to the
    current values of the variables */
    Double evaluate (String progID);
}
```

Fig. 3. CLM_c Comprising the Specification of Syntax (Dynamic Metamodel), Semantics (Dynamic Metamodel + State Transitions) and Pragmatics (Facade)

statements for loading (`Load`) and saving (`Store`) variables (`Var`). Variables and constants (`Const`) are expressions (`Expr`). Expressions have a value (`val`) as well as a left and right side of a specified binary operator (`Bin`).

2) Semantics Specification: L_c's *semantics* is specified (i) by extending the metamodel with information about the *dynamic state* and (ii) by adding a description of *state transitions* by following Plotkin's *Structured Operational Semantics* (SOS) approach [12]. This was an ad-hoc, pragmatic choice.

Like in *Dynamic Metamodeling* [13], L_c's metamodel elements depicted in Figure 3 also cover the *dynamic state* of its set of conforming models. The dynamic state is part of the semantics specification of L_c which is an essential part of any CLM. The attribute `val` belonging to the dynamic state can be changed during model execution.

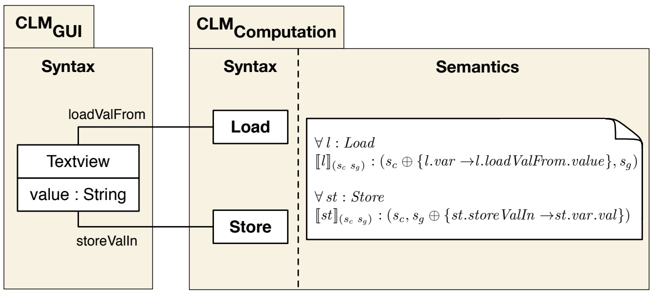
In terms of encoding the allowed *state transitions* in the dynamic state, we chose Plotkin's approach as a technology-independent precise and comprehensible formalism. For example, as shown in Figure 3, the semantics of a `Stmt` in L_c is that a given variable's value is replaced with another (potentially the same) value.

3) Pragmatics Specification: L_c's *pragmatics* is specified using a facade, e.g., using a notation similar to Java classes as illustrated in Figure 3. This approach facilitates the use of models similar to code objects. Moreover, the specification of available *services on models* of a given language, here L_c, supports communication between modeling language designers, software architects and software engineers. We use the term *services on models* to denote capabilities and functionalities specific to a modeling language that facilitate the use of models. These services are realized as *facades*.

For example, the `ComputationModel` facade allows to load a model from a file, to get and set a value for a variable and, importantly, to evaluate a model (e.g., representing an insurance fee formula in ISA) by starting model execution at a certain `Prog` element.

B. Integration of L_c and L_g

Besides the use of single modeling languages, it is particularly interesting when multiple languages are used together.[1] In the context of ISA, each insurance product MoCo carries the insurance fee formula (an instance of L_c) and its corresponding representation of the graphical user interface (an instance of L_g). The two modeling languages had to be *integrated*. For this purpose, additional associations (`loadValFrom` and `storeValIn` roles) were introduced and the specification of state transitions in CLM_c was extended to encode the semantics of loading values from the GUI (`Load`) and storing values from a formula in the GUI's `TextView` (`Load`). In Figure 4, the corresponding integration via additive extension of two CLMs is given.



$$
\begin{array}{l}
\forall\, l : Load \\
[\![l]\!]_{(s_c \; s_g)} : (s_c \oplus \{l.var \rightarrow l.loadValFrom.value\}, s_g) \\
 \\
\forall\, st : Store \\
[\![st]\!]_{(s_c \; s_g)} : (s_c, s_g \oplus \{st.storeValIn \rightarrow st.var.val\})
\end{array}
$$

Fig. 4. Integration of two Comprehensive Language Models

CLMs support the specification and design of modeling languages to be used within model-integrating software components as described. However, there are still many open challenges that need to be tackled.

[1]The *GEMOC initiative* (http://gemoc.org/) provides related work on the coordinated use of heterogeneous modeling languages.

TABLE I
INITIAL LIST OF CHALLENGES FOR MODELING LANGUAGES IN MOCOS

ID	Challenge Description
Syntax	
C1	How to modularize metamodels?
C2	How to integrate two metamodels?
C3	How to establish links between different models?
C4	How to provide a context-dependent view on sets of models?
Semantics	
C5	How to specify model semantics?
C6	How to realize model execution?
C6.1	How to manage the dynamic state of a model?
C6.2	How to reuse model interpreters?
C6.3	How to support the interplay of different model interpreters?
C7	How to support variants of semantics for the same modeling language?
Pragmatics	
C8	How to establish control and data flow between models and code?
C9	How to design language-specific and usage-specific services on models?
C10	How to control access to models?

III. CHALLENGES

Building on the use of CLMs, we elaborate on an initial list of *challenges* for modeling languages in the context of MoCos using the ISA running example. For readability, we formulate each challenge as a question and cluster them according to (i) *syntax*, (ii) *semantics* and (iii) *pragmatics* as summarized in Table I. A detailed description follows subsequently.

A. Syntax Challenges

1) Modularization of Metamodels: Software architects follow a divide-and-conquer approach and split larger systems in smaller pieces, e.g., into software components and connectors. A standardized approach for the modularization of modeling languages is missing, though. A main part of any DSML's definition is the specification of its syntax with a metamodel. While package-structures and import mechanisms are available, depending on the concrete modeling technology, the modeling language designer cannot orchestrate modeling languages and their metamodels in a black-box fashion (C1). In ISA for example, feature models (L_f) are managed and used by a MoCo called `IsaConfig` and insurance fees models (L_c) as well as GUI models (L_g) are managed and used by insurance products like the `IsaCarProduct1` MoCo.

2) Integration of Metamodels: Integration means that at least two existing modeling languages shall be merged. In contrast to distributed, potentially not connected models conforming to different metamodels, this approach conveniently enables full access, e.g., via model queries, to the conforming models of this integrated modeling language (C2). In ISA for example, insurance fee models and GUI models are used tightly together within the `IsaCarProduct1` MoCo, e.g., values from computed fees are directly associated with elements of the user interface.

3) Links between Different Models: In a MoCo-based software system, the architectural decomposition based on functionality dictates a clean separation of concerns between the various MoCos. As in any other component-based software systems, MoCos are connected with each other via provided and required interfaces. While separation of concerns has many well-known advantages, it has one major disadvantage in the context of MoCos: the flexibility that comes with the ability to query and transform a single (possibly large) interconnected model can no longer be leveraged if single models are distributed and encapsulated across individual MoCos (C3). There are no associations between them on the model-level. The ability to establish links and to access these distributed models is especially helpful to debug MoCos and their interdependencies. In ISA for example, it is interesting to know the available insurance fee formulas (contained in `IsaCarProduct1`) for a specific feature configuration (contained in `IsaConfig`).

4) Context-Dependent Views on Models: Modeling languages and their metamodels are only partly used, i.e., the available expressiveness is not required and a restricted metamodel will be sufficient. Moreover, only some data may be relevant for a certain use case and MoCo. The ability to define an adequate and context-specific *view* on sets of models (C4) helps to reduce complexity and supports ease of use of DSMLs. In fact, a view can be seen as a specification for the model parts that can be used by another MoCo. In ISA for example, an adaptation manager MoCo may need to control certain location-dependent variables of the insurance fees and their associated GUI elements. These parts could be encoded by a special adaptation view.

B. Semantics Challenges

1) Specification of Model Semantics: The models in MoCos are not only used as pure data (similar to databases) but some are also executed. Therefore, the semantics of modeling languages needs to be precisely specified (C5). There is a multitude of options available and one can choose between a spectrum of rather informal and very formal approaches [14]. It is important to choose a formalism that is both sufficiently formal but also adequately practical for software engineers and modeling language designers. In ISA for example, all three modeling languages are executable: L_f is interpreted for architectural reconfiguration, L_c is interpreted to compute an insurance fee and L_g is interpreted to create and synchronize an Android-specific graphical user interface for each insurance product represented by a single MoCo.

2) Realization of Model Execution: Given a specification of semantics for a modeling language, this specification needs to be implemented (C6) and related decisions need to be taken carefully. In general, there has been no commonly accepted proposal for the realization of model semantics/model interpreters, yet. Besides the development of stand-alone interpreters and model interpreters embedded into the metamodel, code generation is another option. Each approach has its advantages and disadvantages, e.g., with regards to performance, complexity and reusability. In ISA for example, there is at least one model interpreter for each modeling language.

One sub-challenge that is critical in the case of interpretation is the way to deal with the parts of a model that may change during interpretation (C6.1). We refer to them as the model's *dynamic state*, in contrast to the rest of the model, the *static structure*. While these parts can be regarded as the execution context of a stand-alone model interpreter, it can be also seen as a part of the actual model. In ISA for example, models of the kind of L_f, i.e., feature configurations, are used for runtime reconfiguration. This implies that the state of a feature (selected or not selected) varies. This information can be stored separately by the model interpreter (e.g., technically as a hashmap) or it can be a Boolean attribute in the L_f metamodel.

A second sub-challenge is related to the reuse of model interpreters (C6.2). In this specific case, one needs to distinguish between (i) reuse of model interpreter implementations and (ii) their instances at runtime. Depending on the chosen type of implementation, the reuse potential varies. In ISA for example, the same feature model can be interpreted by different model interpreters concurrently if the dynamic state, i.e., the feature selection flag, is stored by the model interpreters themselves. On the contrary, if the dynamic state is part of each model but needs to be different per semantics, then the model needs to be duplicated or an embedded model interpreter needs to instantiate the dynamic state multiple times.

A third sub-challenge is related to the interdependencies of semantics and, hence, the interplay of model interpreters (C6.3). Ideally, each modeling language comes with its own set of model interpreters. In case that two modeling languages need to be used together, it is required that not only their metamodels are integrated (see C2) but it is also necessary that their semantics fit. In the simplest form, one model interpreter invokes another model interpreter, which asks for a more sophisticated management of dependencies – especially if these shall be dynamic. In ISA for example, each insurance product MoCo encapsulates an insurance fee formula model and a GUI model. Given that their metamodels were previously integrated, the model interpreter of L_c (i) may access metaclasses of L_g and operate on them (e.g., store a computed insurance fee in a text field), or (ii) may invoke the model interpreter of L_g to perform the task.

3) Variants of Semantics: We experienced that while for some modeling language (especially general-purpose modeling languages) a single semantics specification is sufficient, in the case of DSMLs, multiple semantics for the same modeling language need to be supported (C7). Therefore, in this context, a modeling language becomes a *software product line*. Reuse is critical to deal with complexity and to avoid redundancy and duplication. In ISA for example, there may be behavioral semantics (dynamic reconfiguration), constraint checking semantics and visualization semantics for L_f.

C. Pragmatics Challenges

1) Data and Control Flow between Model and Code: In MoCos, models and code coexist and realize the functionality of the component together. It is important to be able to invoke models from code and vice versa (C8). The MoCo Template already defines a pattern with its `Mediator` and sketched interfaces. We deem it important to further standardize these interfaces and to provide realization guidelines, e.g., in the scope of our *reference implementation* (MoCo API) [1]. The design and development of *language-specific facades* encapsulating required services as a part of a CLM needs to be researched. In ISA for example, there is code for sending an email report in the `MoCoCode` module of the `IsaCarProduct1` MoCo that receives data from the `MoCoModel` module (insurance fee model, GUI model).

2) Services on Models: When talking about services on models, two categories need to be distinguished: (i) *foreseen services* that are provided by the modeling language designer and (ii) *unforeseen services* that are specific to a certain user of a modeling language (e.g., a system or a component). Moreover, modeling languages – especially DSMLs as primarily used in MoCos – need to be compact and adequately expressive. A strategy and a set of mechanisms is required to specify *context-specific services*, realize them and to manage the resulting variability (C9). In ISA for example, different insurance products, i.e., different MoCos, require similar special services, like email reporting. This particular service was not initially foreseen when developing L_c and L_g. Therefore, it was first developed as a part of the respective `MoCoCode` module, resulting in clones across the different insurance product MoCos. To solve such issues, often required and DSML-specific services need to be offered by a facade as a part of the CLM.

3) Access Control for Models: Given the power of models in the MoCo concept, any kind of analysis or manipulation needs to be controlled (C10). Models need to be accessible only in predefined authorized, i.e., safe and secure, ways. It needs to be decided whether models can be accessed using the MoCos' interfaces only or if there is a more powerful role that can inspect everything, i.e., all models within all MoCos across the system architecture. Indeed, there is a tradeoff between flexibility and encapsulation. In ISA for example, obviously insurance fee formulas should not be editable by anyone, even though this is technically possible at any time using model transformations. An adaptation manager MoCo that adjusts the fee formulas according to the geolocation of the sales person and potential customer requires exactly this ability, though.

IV. Related Work

While there are works that deal with individual challenges described in this paper, we observe that a *comprehensive solution approach* is missing. The work on Model-Integrated Computing, initiated by *Sztipanovits and Karsai* [15], targets similar issues but lacks the runtime aspect. Due to limited space, we can only hint at an excerpt of related work here.

Regarding the topic of *syntax*, *Heidenreich et al.* [16] present a generic approach for the composition of models that is based on invasive software composition and the Reuseware Composition Framework. *Krahn et al.* [17] use an extended grammar format that supports language inheritance and embedding for the modular development of textual domain-specific languages. *Bae et al.* [18] propose to modularize a large metamodel into a set of small metamodels and present their idea of model slicing along the UML. In contrast to modularization approaches, *Atkinson et al.* propose a Single-Underlying-Model (SUM) [19] that serves all users.

Regarding the topic of *semantics*, *Plotkin* [12] proposes a structural approach to operational semantics. *Engels et al.* [13] describe dynamic metamodeling as a graph-based approach to the specification of semantics for (behavioral) modeling languages. The *Object Management Group* (OMG) [20] provides a specification of the semantics of a foundational subset for executable UML models (fUML) using activity diagrams and a dedicated action language. *Mayerhofer* [14] comprehensively describes the state of the art in model execution.

Regarding the topic of *pragmatics*, *Balz et al.* [21] discuss the embedding of behavioral models (state machines) into object-oriented source code. *Ecore Facade* [22] is a textual domain-specific language for annotating existing Ecore metamodels. This mechanism can be used to define multiple views for a single metamodel via Ecore facade models. The survey by *Szvetit and Zdun* [23] covers existing research on models at runtime and software architecture in detail.

V. Concluding Remarks

In this paper, we introduced *comprehensive language models* as a way to specify modeling languages in the context of *model-integrating software components*. Moreover, we gave concrete examples along the insurance sales app study and elaborated on a first set of *challenges*.

We conclude that an infrastructure is needed that provides all model-specific services in a light-weight, homogeneous, formally founded, easily understandable, and efficient manner using a comprehensive *technological modeling space* supplying full modeling and metamodeling support, and coherent interoperable services based on a powerful data structure.

Regarding future work, we plan to carry out additional case studies to identify further challenges for the infrastructure needed to support model-integrating software components. In the long run, we aim to manifest our lessons learned in a systematically derived engineering method [24].

Acknowledgment

This work is supported by the Deutsche Forschungsgemeinschaft (DFG) under grants EB 119/11-1 and EN 184/6-1. The authors would like to thank Gregor Engels for his valuable feedback and Thomas Iguchi for his implementation support.

References

[1] M. Derakhshanmanesh, J. Ebert, T. Iguchi, and G. Engels, "Model-Integrating Software Components," in *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, ser. Lecture Notes in Computer Science, J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfrán, Eds., vol. 8767. Springer, 2014, pp. 386–402.

[2] M. Derakhshanmanesh, *Model-Integrating Software Components - Engineering Flexible Software Systems*. Springer, 2015.

[3] M. Derakhshanmanesh, M. Amoui, G. O'Grady, J. Ebert, and L. Tahvildari, "GRAF: Graph-based Runtime Adaptation Framework," in *Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems - SEAMS '11*. New York, NY, USA: ACM Press, May 2011, pp. 128–137.

[4] G. Blair, N. Bencomo, and R. B. France, "Models@run.time," *Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[5] I. Kurtev, J. Bézivin, and M. Aksit, "Technological Spaces: An Initial Appraisal," in *International Symposium on Distributed Objects and Applications, DOA 2002*, 2002.

[6] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron, "A Classification Framework for Software Component Models," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 593–615, 2011.

[7] "Eclipse Modeling Framework Hompage," https://eclipse.org/modeling/emf/ (accessed July 16th, 2015).

[8] "JGraLab Hompage," http://jgralab.uni-koblenz.de (accessed July 15th, 2015).

[9] S. Götz, N. Bencomo, and R. France, "Devising the Future of the Models@Run.Time Workshop," *SIGSOFT Softw. Eng. Notes*, vol. 40, no. 1, pp. 26–29, Feb. 2015.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[11] The OSGi Alliance, "OSGi Core Release 5," The OSGi Alliance, Tech. Rep. March, 2012, http://www.osgi.org/Download/File?url=/download/r5/osgi.core-5.0.0.pdf (accessed July 15th, 2015).

[12] G. D. Plotkin, "A Structural Approach to Operational Semantics," 1981. [Online]. Available: http://homepages.inf.ed.ac.uk/gdp/publications/

[13] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer, "Dynamic Meta-Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML," in *Proceedings of the 3rd international conference on the Unified Modeling Language (UML 2000), York (UK)*, ser. LNCS, B. S. A. Evans, S. Kent, Ed., vol. 1939. Berlin/Heidelberg: Springer, 2000, pp. 323–337, third International Conference.

[14] T. Mayerhofer, "Defining Executable Modeling Languages with fUML," Ph.D. dissertation, Institute of Software Technology and Interactive Systems, 2014.

[15] J. Sztipanovits and G. Karsai, "Model-Integrated Computing," *Computer*, vol. 30, no. 4, pp. 110–111, Apr. 1997.

[16] F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler, "On Language-Independent Model Modularisation," in *Transactions on Aspect-Oriented Software Development VI*, ser. Lecture Notes in Computer Science, S. Katz, H. Ossher, R. France, and J.-M. Jézéquel, Eds. Springer Berlin Heidelberg, 2009, vol. 5560, pp. 39–82.

[17] H. Krahn, B. Rumpe, and S. Völkel, "MontiCore: Modular Development of Textual Domain Specific Languages," in *Objects, Components, Models and Patterns*, ser. Lecture Notes in Business Information Processing, R. Paige and B. Meyer, Eds. Springer Berlin Heidelberg, 2008, vol. 11, pp. 297–315.

[18] J. H. Bae, K. Lee, and H. S. Chae, "Modularization of the UML Metamodel Using Model Slicing," in *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*, April 2008, pp. 1253–1254.

[19] C. Atkinson, R. Gerbig, and C. Tunjic, "A Multi-level Modeling Environment for SUM-based Software Engineering," in *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, ser. VAO '13. New York, NY, USA: ACM, 2013, pp. 2:1—-2:9.

[20] The Object Management Group, "Semantics of a Foundational Subset for Executable UML Models (fUML)," p. 441, 2012. [Online]. Available: http://www.omg.org/spec/FUML/

[21] M. Balz, M. Striewe, and M. Goedicke, "Embedding Behavioral Models into Object-Oriented Source Code," *Proceedings of "Software Engineering 2009"*, 2009.

[22] "Ecore Facade," 2015. [Online]. Available: http://www.emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_Ecore_Facade

[23] M. Szvetits and U. Zdun, "Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime," *Software & Systems Modeling*, pp. 1–39, 2013.

[24] G. Engels and S. Sauer, "A Meta-Method for Defining Software Engineering Methods," in *Graph Transformations and Model-Driven Engineering*, ser. Lecture Notes in Computer Science, G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, and B. Westfechtel, Eds. Springer Berlin Heidelberg, 2010, vol. 5765, pp. 411–440.

Towards a deep metamodelling based formalization of component models

Antonio Cicchetti
School of Innovation, Design and Engineering (IDT)
Mälardalen University, Västerås, Sweden
email: antonio.cicchetti@mdh.se

***Abstract*—Component-based software engineering (CBSE) is based on the fundamental concepts of components and bindings, i.e. units of decomposition and their interconnections. By adopting CBSE, a system is built-up by means of a set of re-usable parts. This entails that system's functionalities are appropriately identified so that implementing components can be accordingly selected. In turn, this means that each component-based design is at least made-up of two different instantiation levels, i) one for designing the system in terms of components and their interconnections, ii) and one for linking possible implementation alternatives for each of the existing components. In general, this twofold instantiation is managed at the same metamodelling level through the use of relationships. Despite such solutions are expressive enough to model a component-based system, they cannot represent the instantiation relationship between, e.g., a component and its implementations. As a consequence, validity checks have to be hard-coded in a tool, while the interconnection between component and implementation have to be managed by the user.**

In this paper we propose to exploit deep metamodelling techniques for implementing CBSE mechanisms. We revisit CBSE main concepts through this new vision by showing their counterparts in a deep metamodelling based environment. Interestingly, multiple instantiation levels enhance the expressive power of CBSE approaches, thus enabling a more precise system design.

***Index Terms*—model-driven engineering; component-based software engineering; component models; deep metamodeling; instantiation level;**

I. Introduction

The increasing complexity of contemporary software systems and the growing pressures to deliver products faster while still keeping high quality attributes demands appropriate development solutions. Component-based software engineering CBSE [1] is a well-established methodology that proposes to alleviate software development intricacy by studying the target application as an assembly of composable units (indeed, software components), each one addressing a particular aspect of the system. In this way, the complexity of the initial problem can be reduced through its partitioning into smaller sub-problems. Moreover, time devoted to development and testing can be narrowed by promoting the reuse of already existing components across several software development projects [2].

Component-based system (CBS) specifications are intrinsically hierarchical: i) on the one hand, a component might be realised as the composition of several nested components; ii) on the other hand, a component might have multiple implementations distinguished by quality attributes, target platform, and so forth. Usually, modelling languages support such hierarchical structure in terms of relationships between a component and its sub-components, or between a component and its realisations, respectively. Despite this approach is powerful enough to represent complex CBSs from the expressiveness point-of-view, it requires a careful management of system validation. Notably, type correctness checking, that is verifying whether a component realisation is a valid instance of the component specification, has to be hard-coded in the tool. Moreover, this check should be re-executed each time changes were performed in the component specification and/or in its realisation. Besides, the relationship solution becomes quickly intricate with the growth of hierarchical decomposition levels. Practically, supporting more than two levels of component nesting poses relevant representation issues, as distinguishing the quality attributes of a parent component from the ones of its nested children.

Deep metamodelling [3] is a recent technique introduced in the model-driven engineering (MDE) research field to cope with multiple instantiation levels. It enhances the usual 4-layered metamodelling architecture [4] (also known as two-level metamodelling) by providing a recursive language extension/instantiation structure. In this respect, the deep metamodelling vision fits perfectly with CBSE methodology and its hierarchical decomposition of software systems [3]. In fact, deep metamodelling allows to represent a system and its components by means of arbitrary decomposition/instantiation levels.

This paper investigates the implementation of a component model by means of deep metamodelling mechanisms with the aim of verifying the feasibility of such a solution. The initial results illustrated in this work confirm the feasibility of the approach and meet the expectations of exploiting deep metamodelling mechanisms. Notably, hierarchical component structures can be represented in an easier way, while the conformance check of a component instance against a component specification is obtained by-construction. Despite both the component model and the deep metamodelling solution are specific, the discussion is kept generic enough to be reproducible with other component models and deep metamodelling approaches.

The paper is organised as follows: next section introduces CBSE together with a running example, which will be exploited in Section III to clarify the issues raising in considering multiple instantiation levels. Section IV discusses the proposed formalisation of CBSE concepts through a deep metamodelling framework. Eventually, related works are discussed and conclusions are drawn in Sections VI and VII, respectively.

II. Instantiation relationships in CBSE

CBSE methodology relies on the notion of component, that is "*a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party.*" [5]. Depending on the application domain, technological platform, and so on, the concept of component might include disparate characteristics, which are typically defined in a corresponding component model [6]. Therefore, a CBS is specified by adhering to a well-defined component model, that prescribes how components, their interconnections, and their deployment, look like.

For example, let us consider a simple Personal Navigation Assistant (PNA)[1] CBS as depicted in Figure 1: it includes `GPS Receiver`, `Power Management`, `Navigation System`, and `UI` components (represented as boxes with names). For the purpose of this paper, it is sufficient to know that the `Navigation System` retrieves geo-positioning information from a `GPS Receiver` and delivers navigation data to a user interface (`UI` component). These interconnections are represented by means of named relationships linking component ports. More precisely, a triangle shaped port represents an (provided) output of a certain component, while a square represents an (required) input. Therefore, `Navigation System` gets `Position` information from `GPS Receiver` and, after computing relevant `Navigation Data`, it delivers them to the `UI`.

[1]The example has been taken from [7] and readapted for the purpose of this paper.

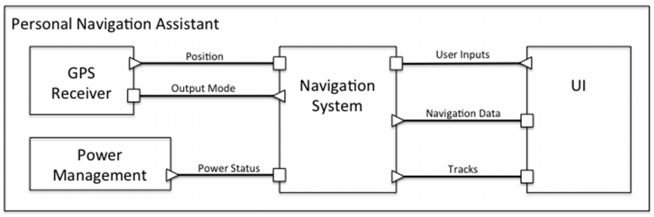


Fig. 1. A simple Personal Navigation System.

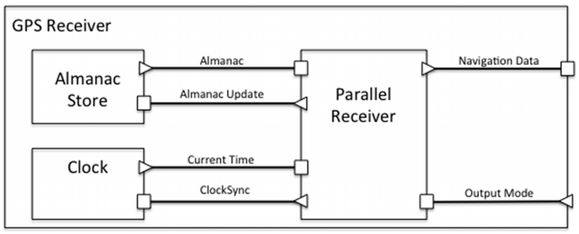


Fig. 2. A simple GPS receiver component.

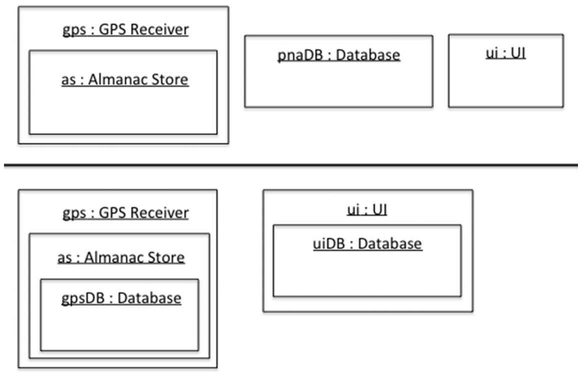


Fig. 3. Two (excerpts of) possible implementations for the PNA system.

In general, a component can include nested subcomponents, referred to as composite components [6]. This is the case of the `GPS Receiver`, which has a complex internal structure. As shown in Figure 2, GPS antennas (`Parallel Receiver`) have to coordinate their tasks with `Clock` and `Almanac Store`. In particular, satellite availabilities depend on the current time and are stored in an almanac.

Eventually, components are attached with one or more implementations, which can be distinguished by quality attributes, supported platforms, and so forth. Notably, for the PNA one might want to prioritise power consumption versus precision in a mobile phone while doing the opposite for a rescue device. Figure 3 illustrates two implementation alternatives for the PNA system: in the one shown on the top half of the picture, a single `Database` is shared between the implementations of `Almanac` and `UI` components, whereas the realisation shown on the bottom half exploits separate databases.

It becomes quickly evident that CBSE methodologies are intrinsically hierarchical: the generic notion of component assembly is instantiated by means of a specific component model (e.g., the simple one used in the example), which in turn is instantiated into a particular CBS (the PNA system). Even further, components can be realised in terms of other components and/or through implementations (as shown in Figure 2 and 3, respectively). In this respect, it is expectable that each CBS specification is made-up of only valid instances for the component model, the components defined in the system together with their implementations. Some of these instantiation relationships are managed by-construction: notably, a CBSE tool is built-up on a well-defined component model, hence the tool will support the design of CBSs by means of all and only the concepts offered by the selected component model (i.e. there is no need to verify that a CBS specification conforms to the component model).

A number of instantiation relationships however have to be checked case-by-case, and this validation step has to be addressed either by the designer, through appropriate constraints at modelling level (e.g. by means of OCL [8]), or hardcoded

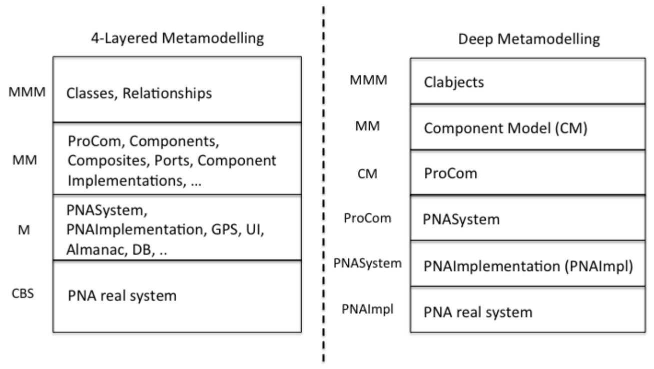


Fig. 4. A comparison between 4-layered and deep metamodelling architectures.

into the tool. For example, when specifying that the `GPS Receiver` composite component in Figure 1 is decomposed as in Figure 2, the tool should at least verify that input and output ports of the latter component specification are compatible with input and output ports of the former composite component (e.g., matching types). A similar reasoning has to be done when considering the interconnection between components and corresponding implementations. More specifically, every implementation of `GPS Receiver` should be compatible with every implementation of `Navigation System` when considering the exchange of `Position` and `Output Mode` data (e.g., the implemented setter and getter methods should match with their types). Regardless whether specified by the designer or if hardcoded in the tool, keeping consistent and up-to-date validity checks can be time-consuming and error-prone, especially when considering complex CBSs. Notably, if the system needed a more precise tracking of power status, the `Power Management` component could be refined as providing more details. In turn, these refinements should be propagated at implementation level by choosing appropriate component implementations for both `Power Management` and `Navigation System`.

III. On the need of a deep metamodelling solution

Current modelling techniques are usually based on a 4-layered metamodelling architecture [4]: a software system is represented by means of a model, that is an abstraction of reality for a given purpose. The model is created by following a set of well-formedness rules stated in a language definition, referred to as the metamodel. In other words, a metamodel defines the set of legal abstractions for a certain system. A model is said to conform to a metamodel if it adheres to the defined well-formedness rules. At the top of the 4-layered architecture there is the meta-metamodel, i.e. a unique minimal set of concepts needed to create all the possible languages. In this respect, the specification for the example introduced in Section II would be supported as shown on the left side of Figure 4: the `MMM` layer would be exploited to define a CBSE language based on a specific component model (at level `MM`), while the PNA system, its (sub-)components, bindings, and component implementations, would all be represented at the modelling level (i.e., `M`).

The conformance validity issues mentioned in Section II are due to the fact that a certain entity either pertains to the metamodel or to one of the models conforming to it. Moreover, at language level, realisation links defined between composite components and sub-components, and analogously between components and implementations, cannot guarantee conformance (i.e., they cannot impose type instantiation constraints). Technically, these relationships link concepts pertaining to different metamodelling layers that however cannot be represented in the typical 4-layered metamodelling architecture [3]. More specifically, the PNA system in Figure 1 is an instance of a certain component model, and at the same time the implementations in Figure 3 are instances of PNA components. In other words, a certain entity should play the role of a concept definition (`MM` level in Figure 4) and instance (`M` level in Figure 4) at the same time.

Multiple metamodelling layers allow to appropriately represent instantiation hierarchies, as depicted on the right side of Figure 4: a model can be equally considered as an instance conforming to the metamodel on the layer above and as a language definition (i.e. as a metamodel itself), for the layer below. In this way, it is be possible to define a component model as a metamodel a certain CBS specification conforms to, like it happens for `CM` and `ProCom` levels. In turn, the CBS specification would constitute a metamodel for which (sub-)component instances could be created (see `ProCom` and `PNASystem` levels, respectively). Eventually, implementations would be represented in a model conforming to a metamodel including simple component definitions (i.e., `PNAImpl`).

IV. A deep metamodelling formalisation for CBSE

This section illustrates the proposed formalisation of CBSE methodologies into a deep metamodelling framework. The formalisation proceeds step-by-step, from higher abstraction level concepts towards more and more concrete instantiations of them. In particular, we leverage a specific component model, namely ProCom [9], to implement the example presented in Section II. Moreover, we exploit MetaDepth [10] as support for concretising the formalisation proposal on a specific deep metamodelling environment. It is worth noting that, despite the component model and deep metamodelling solution are specific, the discussion is kept generic to be extensible to arbitrary component models and other deep metamodelling solutions.

In order to develop a system through CBSE methodologies, it is necessary to preliminarily adopt a specific component model [6]. In the most generic terms, a component model is made-up of components, bindings, and a platform. By adopting MetaDepth syntax, these concepts are specified as shown in Listing 1. In particular, `Component` nodes are bound by means of directional `Binding` edges (the direction is identified through attributes `bindingOut`, `bindingIn`, respectively). A similar reasoning can be done for the `Deployment` relationship between `Component` and `Platform` nodes.

It is worth noting that, already at this stage it is possible to put modelling constraints: the `noSelfBinding` expression at line 18 prescribes that a component cannot be bound to itself. Moreover, `child` multiplicity at line 9 establishes that a `Composite` must have at list one nested component.

```
 1 Model ComponentModel@*{
 2  ext Node Component@*{
 3   bindingIn: Component[*];
 4   bindingOut: Component[*];
 5   deployment: Platform[0..1];
 6  }
 7
 8  ext Node Composite@*: Component{
 9   child: Component[1..*];
10  }
11
12  ext Node Platform{
13   in: Component[*];
14  }
15
16  Edge Binding(Component.bindingOut,Component.bindingIn) {}
17  Edge Deployment(Component.deployment,Platform.in) {}
18  noSelfBinding@* : $Component.allInstances()->forAll(src,tgt
        | Binding(src.bindingOut,tgt.bindingIn) implies src!=
        tgt)$
19 }
```

Listing 1. Encoding of a generic component model.

The generic definition given in Listing 1 introduces the necessary CBSE concepts to create a specific component model. Notably, if we would like to define the ProCom component model, we would need to refine the generic bindings as ports, since ProCom adopts port-based interfaces. In particular, we introduce data ports and trigger ports, as illustrated in Listing 2, lines 4–7. Moreover, bindings have to be refined correspondingly (lines 18–19). It is important to notice that ProCom component model is defined in terms of, or better instantiates, the generic component model defined in Listing 1. This ensures, for instance, that `DataConnection` correctly binds a pair of `ProComComponents` through their `in_dataPort` and `out_dataPort`, respectively. Other alternatives, e.g. connecting a port with a child, would have raised type mismatch issues at validation time due to the type relationships defined before.

```
 1 ComponentModel ProCom{
 2  Component ProComComponent{
 3   name: String {id};
 4   in_dataPort: ProComComponent[*] {bindingIn};
 5   out_dataPort: ProComComponent[*] {bindingOut};
 6   in_triggerPort: ProComComponent[*] {bindingIn};
 7   out_triggerPort: ProComComponent[*] {bindingOut};
 8   parent: ProComComposite[0..1];
 9  }
10
11  Composite ProComComposite: ProComComponent{
12   child: ProComComponent[1..*];
13  }
14
15  Platform ProComPlatform{
16  }
17
18  Binding DataConnection(ProComComponent.out_dataPort,
        ProComComponent.in_dataPort) { name: String {id}; }
19  Binding TriggerConnection(ProComComponent.out_triggerPort,
         ProComComponent.in_triggerPort) { name: String {id};
        }
20  Edge isChildOf(ProComComponent.parent, ProComComposite.
        child) {}
21  Deployment ProComDeployment(ProComComponent.deployment,
        ProComPlatform.in) {}
22 }
```

Listing 2. Encoding of (a subset of) the ProCom component model.

Once the component model has been defined, it is possible to model a CBS. In our case, we specify the PNA system introduced in Section II through ProCom, as shown in Listing 3[2]. In particular, the Navigation System, Power Management, and UI components in Figure 1 are modelled as `ProComComponents`, while the GPS receiver as a `ProComComposite` (see lines 2–13). Moreover, `DataConnections` are specified to bind the components appropriately, and implicitly define data ports for the corresponding components (lines 16–21).

Since GPS is defined as a composite, it is possible to define it as an assembly of sub-components. In this respect, Listing 3 shows the definition of `Almanac Store` at lines 23–25 according to the description of the GPS receiver depicted in Figure 2. Furthermore, by choosing the implementation alternative at the bottom of Figure 3, the almanac is defined as composite, thus allowing the introduction of a nested database component together with its quality attributes (lines 29–34). The nesting specification is completed with the definition of `isChildOf` relationships, as visualised at lines 36–38. Eventually, a platform is introduced to allow the deployment of the PNA system, and component deployments are specified accordingly (lines 41–47).

```
 1 ProCom PNAModel{
 2  ProComComposite GPS{
 3   name = "GPS Receiver";
 4  }
 5
 6  ProComComponent NS{
 7   name = "Navigation System";
 8  }
 9  ProComComponent UI{
10   name = "UI";
11  }
12  ProComComponent PM{
13   name = "Power Management";
14  }
15
16  DataConnection(GPS.out_dataPort,NS.in_dataPort){name="
        Position";}
17  DataConnection(NS.out_dataPort,GPS.in_dataPort){name="
        OutputMode";}
18  DataConnection(PM.out_dataPort,NS.in_dataPort){name="
        PowerStatus";}
19  DataConnection(UI.out_dataPort,NS.in_dataPort){name="
        UserInputs";}
20  DataConnection(NS.out_dataPort,UI.in_dataPort){name="
        NavigationData";}
21  DataConnection(NS.out_dataPort,UI.in_dataPort){name="
        Tracks";}
22
23  ProComComposite AS{
24   name = "Almanac Store";
25  }
26
27     ...
28
29  ProComComponent DB{
30   name = "DB";
31   encryption: String = "NotDefined";
32   queryLanguage: String = SQL;
```

[2]Due to space limitations, some portions of the specification are omitted. The interested reader can download the full specification at http://www.es.mdh.se/~acicchetti/PNASystem.php .

```
33  WCET: int = 22;
34  }
35
36  isChildOf innerASDB(DB.parent,AS.child);
37  isChildOf innerAlmanac(AS.parent,GPS.child);
38  isChildOf innerUIDB(DB.parent,UI.child);
39  ...
40
41  ProComPlatform PNAPlatform{
42   name: String = "PNAPlatform";
43   CPU: String = "FPGA";
44   BUS: String = "EtherNet";
45  }
46
47  ProComDeployment GPSDeployment(GPS.deployment, PNAPlatform
        .in) {}
48  ...
49 }
```

Listing 3. Specification of the PNA system through ProCom.

An excerpt of the implementation of the PNA system is specified as shown in Listing 4. In particular, it illustrates the details for GPS, almanac, and database components (lines 2–18), together with the ones for UI and its nested database (lines 20–21), consistently to the implementation choice depicted at the bottom of Figure 3. Moreover, it shows the declaration of a platform and corresponding deployments at lines 33–34.

```
1 PNAModel pna{
2  GPS gpsImplementation{
3   name = "GPS1";
4  }
5
6  AS asImplementation{
7   name = "AS1";
8  }
9
10 DB dbImplementation1{
11  name = "DB1";
12  encryption = "none";
13  queryLanguage = "SQL";
14  WCET = 13;
15 }
16
17 innerDBAS(dbImplementation1,asImplementation);
18 innerAlmanac(asImplementation,gpsImplementation);
19
20 UI uiImplementation{
21  name = "UI1";
22 }
23
24 DB dbImplementation2{
25  name = "DB2";
26  encryption = "none";
27  queryLanguage = "SQL";
28  WCET = 22;
29 }
30
31 innerDBUI(dbImplementation2,uiImplementation);
32
33 PNAPlatform platform {}
34 GPSDeployment(gpsImplementation.deployment, platform.in);
35 ...
36 }
```

Listing 4. An excerpt of the specification of the PNA system implementation.

V. Discussion

At this point it is important to remark several relevant aspects related to the PNA system specification. From an instantiation procedure point-of-view, the deep metamodelling framework introduces correctness by-construction. Notably, once a system is defined as shown in Listing 3, it will be only possible to introduce component implementations as instances of the defined types (as in Listing 4). Even more important, the implementations have to obey the constraints set in the specification: `innerAlmanac` can only connect an implementation for the almanac with an implementation of a GPS (see line 18), while `GPSDeployment` can only be instantiated with an implementation for the GPS (see line 34). The check of such constraints comes "for free" by the system specification itself, which acts as a metamodel for the system implementation; on the contrary, the 4-layered metamodelling techniques would require additional coding and/or correctness rule definitions to check relationships consistency.

Another relevant aspect to notice is the ease of identification of type instances, which allows to set properties by component implementation, and link each of them to the appropriate component types. In particular, the two different implementations for the database are equipped with different quality attributes and can be included into different composites accordingly. Moreover, the deep metamodelling framework naturally supports the extension of attributes, making it possible to provide additional implementation details for component implementations (e.g. cost, size, and so forth) depending on target platform sensitiveness.

From a higher level of abstraction perspective, the deep metamodelling approach enables the definition of advanced modelling constraints. Notably, the component model might define modelling patterns/styles that later on will have to be preserved by system specifications in order to be successfully validated. This could include the number of components, the kind/number of allowed bindings, and so on. It is important to notice, once again, that similar constraints could be implemented also in the usual 4-layered metamodelling architectures. However, such a need would require implicit checks that in the long run can become time-consuming and error-prone.

As a drawback, the hierarchical arrangement of CBSs specification over multiple metamodelling levels could result as less intuitive and become less usable when dealing with complex systems. In this respect, it is very important to notice that the formalisation is intended to be transparent to the CBS designer, and should be considered as the underlying infrastructure over which a CBS tool would be implemented. MetaDepth is a text-based deep metamodelling environment, and as a consequence this work adopts the same approach. Nonetheless, other existing deep metamodelling tools have already demonstrated the implementability of diagrammatic layers over a base deep metamodelling technology (notably Melanee [11] and the DPF [12]).

The current description of this formalisation in inherently top-down, whereas an ideal CBSE approach would promote a bottom-up development, where useful pre-existing components are identified and picked-up from a repository [13]. In this respect, the component model can be still described as including a repository, and a valid CBS as being a collection of component repository elements. In this case, it would be up to the CBSE tool to create an appropriate instantiation hierarchy based on the selected components.

VI. Related Works

A preliminary choice in adopting a modelling language is deciding whether opting for a general purpose or a domain-specific language [14]. In general, the former solutions have embedded extension mechanisms (like the prototyping mechanisms for the UML [15]), while the latter demand proper language extensions through metamodelling activities. With respect to this paper, the former mechanisms provide more expressiveness through model instances (by inheritance), while the latter ones act on the metamodel to provide appropriate refinements. In both cases, the extensions are limited to the 4-layered metamodelling architecture that does not allow to introduce multiple instantiation levels.

The general need for better addressing multiple instantiation levels has been recognised in the last decade and corresponding solutions have been identified under the name of multilevel (or deep) modelling [10], [12], [16], [17]. In some cases, multilevel modelling techniques have been even used to implement domain-specific component-based systems, notably robots [18] and cloud services [19]. Nonetheless, to the best of our knowledge this is the first work that proposes a general formalisation of CBSE concepts, and in particular of component models, with the aim of enhancing current CBSE techniques.

The problem of managing multiple instantiation levels in CBSE has been already tackled by several works, as [7], [20], [21]. In general these works adopt a 4-layered metamodelling solution, that is, they typically exploit inheritance or other recursive relationships to provide support for containment/refinement modelling [7]. Therefore, they leave open the instantiation problems described throughout the paper.

VII. Conclusion and future works

This paper presents the first steps towards the formalisation of CBSE concepts in a deep metamodelling framework. Component-based systems have an intrinsic hierarchical structure and frequently exploit the "type-instance" pattern [3]. These characteristics have been identified as problematic to be implemented in the usual 4-layered metamodelling architecture and require better support. In this respect, the formalisation illustrated in this work shows promising improvements and gains with regard to both expressiveness and correctness checking.

Future investigation directions will include a more extensive experimentation of deep metamodelling techniques, especially focusing on the adoption of different component models, in order to verify the malleability of deep metamodelling in component adaptation/reconfiguration scenarios [13]. Moreover, the formalisation will have to be embedded in a CBSE tool to better evaluate the usability/scalability aspects related to both modelling and analysis tasks.

Acknowledgements

The author would like to thank Jan Carlson and Severine Sentilles for the interesting preliminary discussions around the topic covered in this paper.

References

[1] I. Crnkovic, "Component-Based Software Engineering for Embedded Systems," in *LMO*, 2006, p. 13.

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

[3] J. D. Lara, E. Guerra, and J. S. Cuadrado, "When and how to use multilevel modelling," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, pp. 12:1–12:46, Dec. 2014.

[4] J. Bézivin, "On the Unification Power of Models," *Software and System Modeling*, vol. 4, pp. 171–188, 2005.

[5] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[6] I. Crnkovic, S. Sentilles, V. Aneta, and M. R. V. Chaudron, "A classification framework for software component models," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 593–615, Sep. 2011.

[7] T. Lévêque and S. Sentilles, "Refining extra-functional property values in hierarchical component models," in *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering*, ser. CBSE '11. New York, NY, USA: ACM, 2011, pp. 83–92.

[8] Object Management Group (OMG), http://www.omg.org/spec/OCL/2.0/PDF.

[9] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, and I. Crnković, "A Component Model for Control-Intensive Distributed Embedded Systems," in *Proceedings of CBSE*. Springer Berlin, 2008, pp. 310–317.

[10] J. de Lara and E. Guerra, "Deep meta-modelling with metadepth," in *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, ser. TOOLS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 1–20.

[11] C. Atkinson and R. Gerbig, "Melanie: Multi-level modeling and ontology engineering environment," in *Proceedings of the 2Nd International Master Class on Model-Driven Engineering: Modeling Wizards*, ser. MW '12. New York, NY, USA: ACM, 2012, pp. 7:1–7:2.

[12] Y. Lamo, X. Wang, F. Mantz, W. MacCaull, and A. Rutle, "Dpf workbench: A diagrammatic multi-layer domain specific (meta-)modelling environment," in *Computer and Information Science 2012*, ser. Studies in Computational Intelligence, R. Lee, Ed. Springer Berlin Heidelberg, 2012, vol. 429, pp. 37–52.

[13] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *J. Syst. Softw.*, vol. 82, no. 1, pp. 3–22, Jan. 2009.

[14] T. Kosar, N. Oliveira, M. Mernik, J. M. Pereira Varanda, M. Črepinšek, D. Da Cruz, and P. Henriques Rangel, "Comparing general-purpose and domain-specific languages: An empirical study," *Computer Science and Information Systems*, vol. 7, pp. 247–264, 2010.

[15] Object Management Group (OMG), "UML Superstructure Specification V2.3," http://www.omg.org/spec/UML/2.3/Superstructure/PDF/, 2011, [Online. Last access: 11/04/2012].

[16] C. Atkinson, M. Gutheil, and B. Kennel, "A flexible infrastructure for multilevel language engineering," *IEEE Trans. Softw. Eng.*, vol. 35, no. 6, pp. 742–755, Nov. 2009.

[17] B. Neumayr, K. Grün, and M. Schrefl, "Multi-level domain modeling with m-objects and m-relationships," in *Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modeling - Volume 96*, ser. APCCM '09. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2009, pp. 107–116.

[18] C. Atkinson, R. Gerbig, K. Markert, M. Zrianina, A. Egurnov, and F. Kajzar, "Towards a deep, domain specific modeling framework for robot applications," in *Proceedings of the First Workshop on Model-Driven Robot Software Engineering (MORSE)*. CEUR-WS, 2014. [Online]. Available: http://ceur-ws.org/Vol-1319/

[19] A. Rossini, J. de Lara, E. Guerra, and N. Nikolov, "A comparison of two-level and multi-level modelling for cloud-based applications," in *Modelling Foundations and Applications*, ser. Lecture Notes in Computer Science, G. Taentzer and F. Bordeleau, Eds. Springer International Publishing, 2015, vol. 9153, pp. 18–32.

[20] J. Odell, "Power types," *JOOP*, vol. 7, no. 2, pp. 8–12, 1994.

[21] R. C. Goldstein and V. C. Storey, "Materialization," *IEEE Trans. on Knowl. and Data Eng.*, vol. 6, no. 5, pp. 835–842, Oct. 1994.

Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions

Lars Hermerschmidt, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann
Software Engineering, RWTH Aachen University, http://www.se-rwth.de/

***Abstract*—Component-based software engineering (CBSE) decomposes complex systems into reusable components. Model-driven engineering (MDE) aims to abstract from complexities by lifting abstract models to primary development artifacts. Component and connector architecture description languages (ADLs) combine CBSE and MDE to describe software systems as hierarchies of component models. Using models as development artifacts is accompanied with the need to evolve, maintain and refactor those models, which can be achieved by model transformations. Domain-specific transformation languages (DSTLs) are tailored to a specific modeling language as the modeling language's concrete syntax is used to describe transformations. To automate the development of DSTLs for ADLs, we present a framework to systematically derive such languages from domain-specific C&C language grammars. These DSTLs enable to describe such model transformations concisely in vocabulary of the underlying ADL. These domain-specific transformations are better comprehensible to ADL experts than generic transformations.**

I. Motivation and Problem Statement

Engineering non-trivial software systems demands techniques to reduce development effort. Component-based software engineering (CBSE) aims to reduce complexity by composing systems from reusable components. Ideally, these components can be developed independently by domain experts and reused off-the-shelf - increasing component maturity along the way. Components of CBSE usually are source code artifacts, which gives rise to "accidental complexities" [1] (dealing with programming instead of domain issues). Model-driven engineering (MDE) aims to abstract from these by lifting abstract models to primary development artifacts. Such models are typically formulated in terms of a domain-specific language (DSL) that reduces noise and trades expressiveness for comprehensibility. In addition, such models can be better reusable, analyzable, and automatically transformable into executable systems. Component and connector (C&C) architecture description languages (ADLs) [2] combine CBSE and MDE to model systems as hierarchies of components.

Using ADLs in MDE gives rise to needs for multiple types of model transformations, such as: i) preprocessing: translate ADL keywords into equivalent component structures or flatten the component hierarchy prior to code generation, rearrange the subcomponent hierarchy for deployment. ii) refactoring: find architectural anti patterns and replace these with established solutions. iii) refinement: replace platform-independent with platform-specific components.

K. Hölldobler is supported by the DFG GK/1298 AlgoSyn.

Describing transformations either requires handcrafting code to transform a model based on its representation, such as an abstract syntax tree (AST), in a general purpose programming language or modeling with a generic transformation language such as ATL [3]. The former is tedious and error prone. The latter requires learning a new language, which might provide adequate transformation descriptions, but cannot rely on the original DSL's notations.

Domain-specific transformation languages (DSTLs) also called "transformations in concrete syntax" [4]–[7] reduce the effort of learning a transformation language as they employ the familiar DSL's syntax. In addition they allow a more concise definition of transformations as the AST is not involved. Producing such DSTLs however requires the same effort as developing a DSL. To approach this, we have developed a framework to generate DSTLs from DSLs while retaining their vocabulary. With this framework, developers can efficiently describe model transformations in well-known form and the overhead of learning additional modeling elements is minimized.

In the following, Sect. II presents the language workbench MontiCore on which our framework, and the ADLs we generate DSTLs for, build. Afterwards, Sect. III describes the framework before Sect. IV illustrates the resulting DSTLs and their application. Sect. V presents related work. Finally, Sect. VI discusses the approach and Sect. VII concludes.

II. Preliminaries

The DSTL generation framework relies on the language development and integration mechanisms of the language workbench MontiCore [8]. With this, it parses the grammars of MontiCore DSLs and generates domain-specific transformation languages. MontiCore provides a language to describe the integrated concrete and abstract syntax of DSLs in terms of context-free grammars and means to generate model processing infrastructure, such as tools to parse textual models into an abstract syntax tree (AST), frameworks for language integration and well-formedness checking [9], as well as code generation [10]. Language integration enables aggregation, inheritance, and embedding between DSLs. For the latter, the host DSL provides extension points filled by modeling elements of the embedded DSL.

We apply our approach to MontiArc [11], a C&C ADL build with MonitCore and its extension MontiSecArc. Both describe logically distributed software architectures as hierarchies of connected components. Components are black-boxes with interfaces of typed, directed ports. The behavior of atomic components is defined by source code artifacts and the behavior of composed components emerges from their subcomponents. MontiSecArc introduces the *trust level* to distinguish components that might be influenced by an adversary from those which are not that easy to reach. qAs modeling something unknown like an adversary is hard, the trust level describes (physical) protection measures which hinder an adversary to compromise a component. The trust level abstracts from individual measures like locked doors, fences, and video surveillance to focus the model on IT security. A subcomponent's trust level is denoted relative to its containing component and the surrounding of a system is assumed as insecure and hence has the trust level -1.

A classical measure to hinder adversaries to access a resource is access control, such as role based access control, or access control lists (ACLs) [12], which is noted by the keyword *access* in MontiSecArc. Access is limited to certain policies, such as roles or ACLs, for specific incoming ports or complete components, where the later is equivalent to access control for all incoming ports of the component. Assigning users to roles or ACLs is left to run-time, such that new users' access rights are defined by the access policy.

To avoid naming problems with policies when composing components, policies of different components are independent, even if they have the same name. When interconnecting components, e.g., client and server as depicted in Fig. 4, where one role has access to different components an *identity* link connects these components. When interconnecting identities, the process of authentication, where a user from a proving component claims to have a role at the verifying component, ensures that only users, which possess this role are able to claim it. In Fig. 4 `Client'` is the proving component and `Server'` the verifying one. To specify proving and verifying component, the `identity` link is directed from the former to the latter.

III. DSTL Generation Framework

The DSTL generation framework is able to create DSTLs that, in conjunction with additional generated and provided parts of the framework, realizes a graph transformation approach. In such approaches, complex transformations are composed of small transformation rules where transformation rules usually are described by a left-hand side (LHS) - the model part before applying the rule - and a right-hand side (RHS) - the same model part after being transformed [13]. The following sections explain the framework to automatically derive DSTLs from the grammar of a modeling language as well as the resulting DSTL and the application of domain-specific transformations.

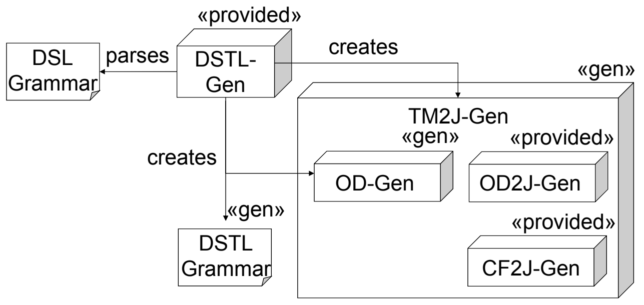


Fig. 1. Overview of the generation of a DSTL including provided and generated generators.

A. From DSL to DSTL

The automatic derivation of DSTL is solely based on the grammar of the DSL. Thus, by taking the modeling language's grammar as input the DSTL generator produces the grammar for describing transformation rules following the derivation rules described in [14] and a generator (`OD-Gen`, Fig. 1) to translate those transformation rules to a LHS and a RHS of a transformation in form of object diagrams (OD notation). Furthermore, the framework provides a generator to translate this OD notation to Java (`OD2J-Gen`), a control flow language to control the application of transformation rules and a generator (`CF2J-Gen`) to translate the control flow to Java.

As complex transformations usually are decomposed to transformation rules combined by some kind of application strategy ([15]), the control flow language uses the transformation rule language via language embedding to allow the description of complex transformations in form of so called transformation modules (Sect. III-B4). Finally, to ease the use of the generators (`OD-`, `CF2J-`, and `OD2J-Gen`) the DSTL generator creates glue code that combines those generators to a single generator (`TM2J-Gen`) able to translate transformation modules to executable Java transformations.

B. Generated DSTLs

A DSTL created by the generator described above reuses the concrete syntax of the DSL to describe patterns. In addition, the DSTL provides a replacement operator for modifications, allows to bind elements to variables, and to specify negative elements and application constraints. With this, the DSTL is able to describe endogenous in-place transformations [15], [16]. In contrast to the typical transformation form consisting of LHS and RHS, we use an integrated notation of LHS and RHS. Combining these in a single model avoids repeating unchanged model parts on the RHS. The transformation operators, such as the replacement operator or negative elements, are provided for every model element defined by a nonterminal such as components and ports. The following explains those operators.

1) Pattern and Schema Variables: The DSTL uses concrete DSL syntax to describe patterns, thus, a pattern resembles the model part it describes and omits parts that do not constrain the pattern. For example, the model in Lst. 1 could also serve as a pattern. However, every component that has the depicted structure and arbitrary additional structures, such as additional ports or subcomponents, would be a suitable match for this pattern. There also is no need to start a pattern at the top-level element of a model. Instead, all elements can be top-level elements in a pattern. For instance, if a transformation is defined for a port and the containing component is irrelevant, the pattern may only define the port and its modification.

In many cases transformations need to be more general, thus, for abstraction purposes as well as binding model elements to variables (for instance to move them), the generated DSTL provides a concept called *schema variables*. Those variables consist of a type, i.e., the name of the nonterminal that defines the model element and a name starting with a $-sign. There are black box and a white box schema variables: Black box variables end with a semicolon ("$ElementType\ SchemaVar$;"), while white box variables allow to define the element's structure within double square brackets ("$ElementType\ SchemaVar$ [[$Element$]]"). An example black box variable is depicted in line 6 of Lst. 3 for an `access` definition. Line 9-12 of Lst. 3 show a white box variable for a component.

To ease the use of variables for names the type `Name` can be omitted. A schema variable for a name is displayed in Lst. 2 (`$name` in l. 7, `$sp` in l. 8). If a schema variable is used for a model element the corresponding element is bound to this variable during pattern matching. Thus, using the same variable twice refers to the same model element in both cases. However, for names we relaxed this such that two occurrences of a schema variable for a name require equality instead of identity. When using variables for abstraction, without the need for referencing them later, the anonymous `$_` variable may be employed. It does not bind the model element and, hence, two occurrences neither require identity nor equality (Lst. 2, l. 9).

2) Modifications: The generated DSTL uses an integrated notation of the LHS and RHS of a transformation rule. To achieve this the DSTL provides the replacement operator :- that acts on element level ("[[$Element?$:- $Element?$]]"). The element left of :- is replaced by the one right of it. If the LHS is left blank an element is created and added. Leaving the RHS blank deletes an element. A modification is illustrated in line 6 of Lst. 2.

3) Negative Elements, Application Constraints and Assignments: Negative application conditions [17] are provided in form of negative elements with the following syntax: `not` [[$Element$]]. A negative element is an element that must not occur in the model. Furthermore, a where-block is provided that allows formulating application constraints and assignments of schema variables. The *where*-block is structured as follows:

`where {` $Assignment*\ \ BooleanExpression?$ `}`

It starts with the assignment of schema variables that are not assigned during pattern matching (i.e., parts of the RHS of a transformation). Within the $BooleanExpression$ the elements of the transformation bound to schema variables can be used to formulate the constraint. Thereby, the signature of the abstract syntax of the model elements can be used as well as any static Java method. Listing 2 shows a negative element (l. 9) and a *where*-block (l. 11). An example of an application constraint is shown in line 14 of Lst. 3. A transformation will only be applied if all positive elements are found, no match for the negative elements is possible and the application constraint holds.

4) Transformation Modules: To control and combine the transformation rules to transformation modules, the generated DSTL is combined with a generic control flow language via language embedding.

A transformation module, as shown in Lst. 2, consists of instructions and transformation methods (introduced by the keyword `transformation`) where the body of a transformation method is a transformation rule. The instruction methods define the application order of transformation methods. The instruction method `main()` is the starting point of a transformation module. Within instructions, Java syntax extended by a special `loop` statement can be used to specify the control flow in an imperative manner. The loop statement applies the following transformation rule until no further match for the pattern can be found.

C. Translation and Application of a Transformation

A transformation module is defined using the control flow language and its embedded transformation rule language (Fig. 2), which have to be translated to Java code for execution. This translation is performed by the composed and generated generator `TM2J-Gen` (Fig. 1 and Fig. 2). `TM2J-Gen` takes a transformation module as input and internally uses its three subgenerators to translate it to an executable Java transformation. The latter reads a model and applies the transformation described by the transformation module.

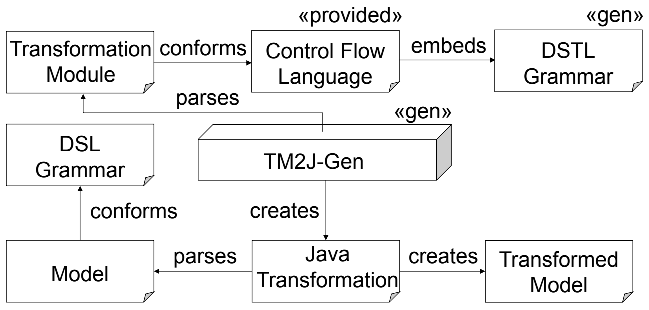


Fig. 2. Overview of the translation and application of a transformation.

IV. Applying the Transformation Language

With the DSTL derived from the DSL's grammar, the description of model transformations is greatly facilitated as the transformation developers are familiar with the DSLTs vocabulary. The following sections illustrate application of model transformations to MontiArc and MontiSecArc with the DSTLs generated for each.

A. Preprocessing: Adding Structural C&C Elements

A common challenge for the development of distributed systems is dealing with the unforeseeable run-time issues.

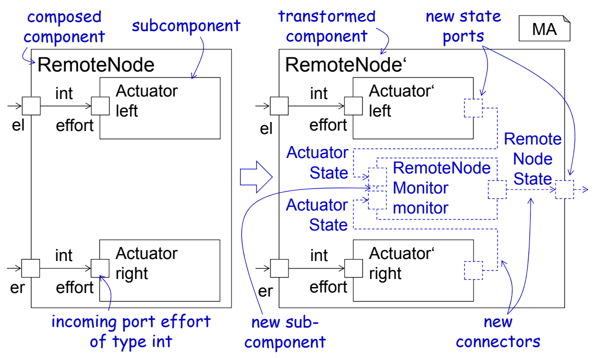


Fig. 3. Applying the monitoring transformation to a composed component RemoteNode with a single subcomponent.

```
1  component RemoteNode {
2    port in int el , in int er ;
3    component Actuator left , right ;
4    connect el -> left . effort ;
5    connect er -> right . effort ;
6  }
```

Listing 1. Textual syntax of composed component RemoteNode with subcomponent Actuator.

To this effect, MontiArc introduces component monitoring. Every component of the architecture is monitored by a specific monitor per composed component. Instead of handcrafting the monitoring infrastructure for each component, it is conveniently integrated via model transformations. These transformations introduce new subcomponents, ports, and connectors, such that all composed components and their direct subcomponents are observed by a new subcomponent. That subcomponent receives status messages from its neighboring subcomponents, calculates an overall component status, and emits this via a new outgoing port. Applying this transformation to a composed component RemoteNode (Fig. 3) requires that (a) RemoteNode and all its subcomponents receive a new port to emit status messages, (b) RemoteNode receives a new subcomponent of type RemoteNodeMonitor that provides appropriate input ports for all new state ports and emits messages on the overall state of RemoteNode, and (c) the state ports of the subcomponents of RemoteNode are connected to RemoteNodeMonitor, which itself is connected to the new state port of RemoteNode.

As the DSTL's syntax is derived from the DSL, Lst. 1 describes the textual syntax of the untransformed MontiArc component RemoteNode for comprehension. The keyword component (l. 1), followed by a name and curly brackets declares a component definition (ll. 1-6). The components interface is defined by the keyword port and a list of directed, typed ports (l. 2). Furthermore, a composed component contains a set of subcomponents (l. 3), each starting with the keyword component, followed by its type and name. The ports of subcomponents are connected via unidirectional connectors (ll. 4-5).

Handcrafting these transformations in terms of AST API calls requires considerable effort. Instead, the three transformation rules given in Lst. 2 describe this transformation. The main block (ll. 2-4) invokes the three transformations addPorts(), addMonitor(), and connect(), where addPorts() (ll. 6-12) adds state ports to all components of the software architecture. To this effect, it iterates over all components (denoted by concrete MontiArc syntax component followed by a name $name) and adds a new outgoing port state to each of the component's ports rule (l. 8), where no such port already exists (l. 9). The port's type is defined by $sp as calculated by the where-block (l. 11). The transformation addMonitor() (ll. 14-22) adds a new subcomponent monitor (l. 16) to each composed component - enforced by requiring that the component contains a subcomponent (l. 19) - that does not already contain a monitor (l. 17). The type of monitor is calculated via $type (l. 21). Finally, the transformation connect() (ll. 24-31) adds new connectors to each composed component to connect its subcomponents to its new monitor. This is better comprehensible than a lengthy program exploiting the AST API and less susceptible to errors arising from accidental complexities of AST programming.

```
1  module AddMonitoring {
2    main() { loop addPorts();
3             loop addMonitor();
4             loop connect(); }
5
6    transformation addPorts() {
7      component $name {
8        port [[ :- out $sp state ]] ;
9        not [[ out $_ state ]]
10     }
11     where { $sp = $name.concat("State"); }
12   }
13
14   transformation addMonitor(){
15     component $name {
16       [[ :- component $type monitor;]]
17       not [[ component $_ monitor; ]]
18       [[ :- connect monitor.state -> state; ]];
19       component $_ {}
20     }
21     where { $type = $name.concat("Monitor") }
22   }
23
24   transformation connect(){
25     component $_ {
26       component $type $name;
27       [[ :- connect $name.state -> monitor.$sp; ]];
28       not [[connect $name.state -> monitor.$_;]]
29     }
30     where {$sp = $name.concat("State");}
31   }
32 }
```

Listing 2. The transformations required to add a monitor, related ports, and connectors to a software architecture.

B. Refactoring: Resolving Anti-Patterns

Architects need to consider security as one out of many nonfunctional requirements. There are numerous commonly known anti-patterns and design flaws [18]. We consider the anti-pattern of client-side authentication [19, p. 687], which is depicted in Fig. 4 using the MontiSecArc language. In this case a client with low trust level enforces access control and a server which has a higher trust level relies on that client. Hence, an attacker able to impersonate the client can bypass access control and compromise the server, as it relies on the client.

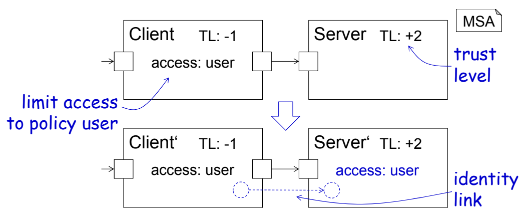


Fig. 4. Applying the transformation to an insecure client server setup introduces access control to the server to make it secure.

```
1  module ClientAuth {
2    main(){ loop accessPort(); }
3
4    transformation accessPort() {
5      SecArcComponent $C [[ component $client {
6        Access $A;
7      } ]]
8      connect $client.$_ -> $server.$someInPort;
9      SecArcComponent $S [[component $server {
10       port in $someInPort;
11       [[ :- access $someInPort ($policy) ]];
12     } ]]
13     where {$policy = $A.getPolicy();
14     $C.getTrustlevel() < $S.getTrustlevel()}
15   }
16 }
```

Listing 3. The transformation moves access control enforcement from client to server components.

We use the transformation depicted in Lst. 3 to identify client components which have this anti-pattern (ll. 5-7) and add access control to the server (ll. 9-12). Client components within this anti-pattern have one of the statement starting with `access`, so we use a black box schema variable `$A` for the common super type `Access` to match both (l. 6). Furthermore, there is a connection (l. 8) to a more trustworthy server (l. 9-12). We use the white box variant of schema variables for the client (`$C` in l. 5) and the server (`$S` in l. 9). In the `where`-block we first retrieve the access policy from the client by utilizing the method `getPolicy()` from `$A` and assign it to the `$policy` variable (l. 13). Finally, to ensure that the client has a lower trust level then the server, we use another method `getTrustlevel()` accessible via the variables `$C` and `$S` (l. 14). Using a combination of keywords and abstract syntax of MontiSecArc in the DSTL makes the patterns precise and comprehensible to domain experts.

V. Related Work

Similar to PROGRES [20], Fujaba [21], eMoflon [22], and Henshin [23], the transformations of our approach are endogenous, and in-place [16]. However, these approaches do not employ the concrete syntax of the underlying DSL. There are approaches for transforming software architectures [24]–[26], however, they either introduce their own notation, operate on the abstract syntax or provides less functionality e.g. do not allows to remove elements [25]. Existing approaches to derive DSTLs from DSLs focus graphical languages [6], [27] and do not provide the concrete syntax of the transformation language. Another approach to circumvent generic transformation languages is to infer LHS and RHS of model transformations from examples [28], [29]. To generalize these examples, developers have to use abstract syntax. Term rewriting [4] works on concrete syntax as well by applying rewriting rules to manipulate rather small connected model parts as compared to graph transformations. T-Core [30] and others [31] introduce transformation primitives which, similar to term rewriting, do not automate the process of deriving an DSTL but are combined and configured to create it. Thus, they do not propose a systematic and automated way of deriving a DSTL, but provide building blocks to create them. Our previous work on delta languages, which describes small changes for models in concrete syntax of the modeling language, shares the underlying generative approach of deriving those languages we use here and we first applied those deltas to architectural models [32].

VI. Discussion

A generated DSTL relies on the concrete syntax of its base DSL. However, for typing schema variables the nonterminal names of the modeling language are used and, thus, this abstract syntax information become part of the concrete syntax of the DSTL. This cannot be avoided completely as for the black box variant of schema variables the type cannot be inferred whenever there is an alternative of nonterminals in the base DSL. Furthermore, keywords such as `not` or `where` and delimiters might conflict with the DSL's concrete syntax. However, these problems can be solved by using MontiCore's language inheritance to redefine the concrete syntax of the DSTL. Allowing every model element of the base DSL as a top level element in transformation rules leads to problems if the model element does not have any mandatory concrete syntax. Restructuring the DSL will solve this issue.

VII. Summary

We presented a framework to generate DSTLs from the grammars of DSLs. The resulting DSTLs consist of declarative transformation rules that employ patterns based on the DSL's concrete syntax to describe both what is to be replaced and how it is to be replaced. These transformation rules are embedded into a control flow language to describe complex, imperative transformation modules. As such DSTLs reuse the well-know vocabulary of the underlying DSL, modeling individual transformations require less effort from domain experts. The control flow language is very compact and learning their combination is less complex than learning a general transformation language. The framework has been applied to the C&C ADLs MontiArc and its descendant MontiSecArc. We currently examine the application of the DSTL generation framework to other ADLs in ongoing case studies.

References

[1] R. France and B. Rumpe, "Model-Driven Development of Complex Software: A Research Roadmap," in *Future of Software Engineering 2007 at ICSE.*, 2007.

[2] N. Medvidovic and R. N. Taylor, “A Classification and Comparison Framework for Software Architecture Description Languages,” *IEEE Trans. Software Eng.*, 2000.

[3] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, “ATL: A model transformation tool,” *Science of Computer Programming*, 2008.

[4] E. Visser, “Meta-programming with Concrete Object Syntax,” in *Generative Programming and Component Engineering*, 2002.

[5] T. Baar and J. Whittle, “On the Usage of Concrete Syntax in Model Transformation Rules,” in *International Andrei Ershov memorial conference on Perspectives of systems informatics (PSI)*, 2007.

[6] R. Grønmo, “Using Concrete Syntax in Graph-based Model Transformations,” PhD thesis, University of Oslo, 2009.

[7] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. V. Mierlo, and H. Ergin, “AToMPM: A Web-based Modeling Environment,” in *MODELS’13: Invited Talks, Demos, Posters, and ACM SRC.*, 2013.

[8] H. Krahn, B. Rumpe, and S. Völkel, “MontiCore: a framework for compositional development of domain specific languages,” *International Journal on Software Tools for Technology Transfer (STTT)*, 2010.

[9] A. Haber, M. Look, P. Mir Seyed Nazari, A. Navarro Perez, B. Rumpe, S. Voelkel, and A. Wortmann, “Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components,” in *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, 2015.

[10] M. Schindler, *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Shaker Verlag, 2012.

[11] A. Haber, J. O. Ringert, and B. Rumpe, “MontiArc – Architectural Modeling of Interactive Distributed and Cyber-Physical Systems,” RWTH Aachen, Tech. Rep., 2012.

[12] R. Sandhu and P. Samarati, “Access control: principle and practice,” *Communications Magazine, IEEE*, 1994.

[13] M. Nagl, *Graph-Grammatiken: Theorie, Anwendungen, Implementierung*. Vieweg, 1979.

[14] K. Hölldobler, B. Rumpe, and I. Weisemöller, “Systematically Deriving Domain-Specific Transformation Languages,” in *International conference on Model Driven Engineering Languages and Systems (MoDELS)*, 2015.

[15] T. Mens and P. V. Gorp, “A Taxonomy of Model Transformation,” *Electronic Notes in Theoretical Computer Science*, 2006.

[16] K. Czarnecki and S. Helsen, “Feature-based Survey of Model Transformation Approaches,” *IBM Systems Journal*, 2006.

[17] A. Habel, R. Heckel, and G. Taentzer, “Graph grammars with negative application conditions,” *Fundamenta Informaticae*, 1996.

[18] IEEE Computer Society Center for Secure Design, “Avoiding the top 10 software security design flaws,” 2014.

[19] M. Howard and D. E. Leblanc, *Writing Secure Code*, 2nd. Microsoft Press, 2002.

[20] A. Schürr, *Operationales Spezifizieren mit Programmierten Graphersetzungssystemen: Formale Definitionen Anwendungsbeispiele and Werkzeugunterstützung*. Wiesbaden: Deutscher Universitäts-Verlag, 1991.

[21] T. Fischer, J. Niere, L. Torunski, and A. Zündorf, “Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java,” in *Theory and Application of Graph Transformations*, 2000.

[22] E. Leblebici, A. Anjorin, and A. Schürr, “Developing eMoflon with eMoflon,” in *Theory and Practice of Model Transformations*, 2014.

[23] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, “Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations,” in *Model Driven Engineering Languages and Systems*, 2010.

[24] J. M. Barnes, D. Garlan, and B. Schmerl, “Evolution styles: foundations and models for software architecture evolution,” *Software & Systems Modeling*, 2014.

[25] O. Barais, A. F. Le Meur, L. Duchien, and J. Lawall, “Software Architecture Evolution,” in *Software Evolution*, 2008.

[26] L. Grunske, “Formalizing Architectural Refactorings as Graph Transformation Systems,” *SNPD/SAWN ’05*, 2005.

[27] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer, “Explicit Transformation Modeling,” in *Models in Software Engineering*, 2010.

[28] E. Kindler and R. Wagner, “Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios,” Software Engineering Group, Department of Computer Science, University of Paderborn, Tech. Rep., 2007.

[29] P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger, “An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example,” in *Model Driven Engineering Languages and Systems*, 2009.

[30] E. Syriani, H. Vangheluwe, and B. LaShomb, “T-Core: a framework for custom-built model transformation engines,” *Software & Systems Modeling*, 2013.

[31] J. Sánchez Cuadrado, E. Guerra, and J. de Lara, “Towards the Systematic Construction of Domain-Specific Transformation Languages,” in *Modelling Foundations and Applications*, 2014.

[32] A. Haber, K. Hölldobler, C. Kolassa, M. Look, K. Müller, B. Rumpe, and I. Schaefer, “Engineering Delta Modeling Languages,” in *Proceedings of the 17th International Software Product Line Conference (SPLC’13)*, 2013.

Towards a Generic Modeling Language for Contract-Based Design

Johannes Iber, Andrea Höller, Tobias Rauter, and Christian Kreiner
Institute for Technical Informatics
Graz University of Technology
Inffeldgasse 16, Graz, Austria
{johannes.iber, andrea.hoeller, tobias.rauter, christian.kreiner}@tugraz.at

***Abstract*—Component-based and model-driven engineering are key paradigms for handling the ever-increasing complexity of technical systems. Surprisingly few component models consider extra-functional properties as first class entities.**

Contract-based design is a promising paradigm, which has the potential to fill this shortage of methods for dealing with extra-functional properties. By defining the concept of using assumptions in order to determine the environment, and by using the concept of guarantees to state what a component provides to the environment, it enables the analyzability of components and compositions in advance and during system execution.

With this work, we aim to create the base for a pragmatic model-driven method that provides reusable modeling concepts for contracts targeting arbitrary extra-functional properties. Furthermore, we expand the current state-of-the-art of contract-based design by introducing the concept of a finite state machine, where single states consist of several valid contracts. It is also assumed that these modeling language features will ease the use of contract-based design. Additionally, we demonstrate the applicability of the presented modeling concepts on an exemplary use case from the automotive domain.

***Index Terms*—Metamodeling, contract-based design, extra-functional properties, component models**

I. Introduction

Numerous industrial sectors are currently confronted with massive difficulties originating from managing the increasing complexity of systems. The automotive industry, for instance, has an annual increase rate of software-implemented functions of about 30% [1]. This rate is even higher for avionics systems [2]. Additionally, this development of systems is not restricted to software, as we are facing a so-called Internet of Things, where the number of physical devices is expected to expansively explode [3]. New challenges regarding complexity of systems emerge caused by this dramatic increase of diverse hardware/software, possible interactions and distributed intelligences [4].

Component-based engineering is today a widely recognized and well-established paradigm for tackling complexity of systems [5]. Together with model-driven engineering, it forms a potentially powerful union to construct, analyze, and deploy systems.

But still, modern component models are flawed. As shown by Crnković et al. [5], astonishingly few (software) component models are addressing extra-functional properties (e.g. timing, safety, memory consumption, etc.) as first class entities. However, these properties are essential for composing a component-based system predictable and safe. Management of extra-functional properties is thus still one of the core challenges faced by component-based design [6].

Contract-based design is a promising paradigm for filling or narrowing this gap, [7]. It captures the behavior of a specific functional or extra-functional property in relationship with the environment of a component. Despite the existence of a mathematical groundwork [7] [8] and exemplary applications, a standard and generic metamodel for contract-based design does not yet exist.

With this work, we provide pragmatic modeling concepts that pave the way for integrating contract-based design into component models of systems. We present a metamodel fragment for contracts which target arbitrary single extra-functional properties. Furthermore, we introduce the concept of a finite state machine, where single states constitute valid contracts. This concept extends the current state-of-the-art regarding contract-based design. We show the applicability of these modeling concepts by using an example from the automotive domain. The target component of the use case is a simplified electronic steering column lock, which we examine with respect to the extra-functional properties safety and timing.

The remainder of this paper is structured as follows: the next Section provides a brief overview of the background to this work. In Section III the proposed modeling concepts are introduced. Subsequently, a use case demonstrating the applicability of these concepts is described in Section IV. Finally, concluding remarks and future research opportunities are given in Section V.

II. Background and Related Work

Here, we give an overview of system abstractions and properties. After this, we briefly explain contract-based design. Finally, we summarize the related work concerning contract-based design, which is also the motivation setting for this work.

A. System Abstractions and Properties

According to Jantsch [9], there are four main different abstraction models or views concerning embedded system engineering. First is the *computational model*, which describes the observable behavior of a system or of its single parts (hardware, software components), i.e. the relationship between inputs and outputs [10]. Second, a *data model* exists that provides notations for information (e.g. integer, boolean). Third, a *time model* is needed to constitute the causality of events. Fourth, a *communication model* is established to specify how components interact. This model forms the top-level system behavior.

In the context of the properties of systems the literature distinguishes between functional and extra-functional (also known as non-functional) properties. Functional properties describe the function of a system or component, i.e. behavior, input or output data types. Extra-functional properties provide additional information and give a better insight into the behavior and capability of a system or component [6]. A wide range of such properties exists, e.g. safety, security, portability, performance. Since these issue from humans, there is no method to determine a priori which extra-functional properties exist in a system [6] [11].

B. Contract-based Design

Contract-based design usually sees a component as an abstraction, a hierarchical entity that represents a single unit of design [8] [12]. Therefore in the context of contract-based design a component can represent, for instance a module, a composition, a complex system or even a physical device.

The essence of this paradigm is to decompose a component into different independent views referred to as contracts, which capture the behavior of a target functional or extra-functional property under certain conditions [12] [13]. This approach significantly reduces the complexity of design and verification, because the single properties become manageable.

Informally, a contract is a set of assumptions and guarantees.

An assumption asserts what a contract expects from the component environment (this can include interactions with other components). Additionally, an assumption provides a certain context for the guarantees. The condition contained in an assumption can reference for instance input data, events or system properties. In general, the available variables are set or inferred by the analysis environment.

A guarantee describes what a component provides to the environment if the corresponding assumptions become valid. In the simplest case a guarantee states that a component just works under the constrained context. More complex contracts define limits for instance for output data, environment characteristics or extra-functional properties such as timing.

Historically, contract-based design is influenced by Meyer's design-by-contract principle [14] for object-oriented software [7]. The main difference is that contract-based design goes much further and provides means to integrate components in the design hierarchy [10]. This is achieved through capturing the context by assumptions (which may include platforms, other components, etc.), under which a component behaves as specified by the guarantees. Furthermore, a system can be viewed by selecting only appropriate contracts of interest.

Fig.1 illustrates that contract-based design not only allows the analyzing of components on a horizontal design level (e.g.

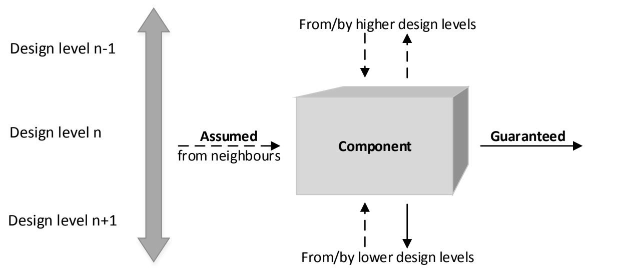


Fig. 1. Contract assumptions and guarantees for a component (Adapted from [15])

interaction between software modules, hardware devices, etc.). It also enables analyzing to take place on a vertical level between different kinds of abstraction [7].

A solid mathematical groundwork already exists for this as provided by several authors, including Benveniste et al. [7], and Sangiovanni-Vincentelli et al. [8].

Promising applications of contract-based design have been shown for several domains. For instance, this paradigm has been demonstrated for smart integrated energy management systems [16], aircraft electric power systems [12], mixed-signal integrated circuits [17], and automotive [18] [7]. Despite these examples, contract-based design is still at its infancy [19].

Little work has been done towards establishing a generic standard metamodel for contract-based design. Warg et al. [20] presents a prototype modeling tool for contracts, but their work solely focuses on safety integrity levels.

C. Summary of Contract-Based Design

There exist a few approaches for realizing contract-based design, for instance the contract-based model developed in the framework of the SPEEDS project [13]. The problem is that state-of-the-art approaches either tackle single extra-functional properties, or take a relatively theoretical approach without concrete modeling examples or tool implementations. A survey concerning the certification of safety-relevant systems, carried out by the SafeCer consortium [21], shows that only a few companies are actually using contracts for components. And where this is the case they are relying on Meyer's design-by-contract principle on a programming language level.

III. Proposed Modeling Language Concepts

In this section, we explain concepts which are necessary for a pragmatic modeling language that targets contract-based design.

A. Target System Abstractions and Properties

With the following concepts, we aim at enriching the computational, time, and communication models of a system. Furthermore, the data model plays an important role, as it provides data types and notations, which could be used by contracts.

In the context of properties, our intention is to capture extra-functional properties and not necessarily functional behavior. We take the view that functional behavior is better described by other well-established methods than by the use of many different contracts.

The issue of what extra-functional properties we are aiming at, is dependents on the specific use case or context under which the following language features are used. These concepts may be applied for a wide range of different extra-functional properties (e.g. security, safety, timing, expected hardware/platform, memory consumption, many-core environment, etc.). But certainly not for all of them, since no silver bullet exists for dealing with every extra-functional property [11].

B. Pragmatic Modeling Langugage Features

In the following, we present a modeling concept for contracts. Additionally, we introduce the concept of a finite state machine for contracts.

1) Contract: Fig.2 illustrates our proposed metamodel for contracts. We separate a contract into two parts. A *Contract Declaration* represents a type for *Contract Definitions*. It states the available parameters, assumptions and guarantees. Furthermore, it represents the target extra-functional property. A *Contract Definition* captures the unique behavior concerning the target extra-functional property of a component in relationship to its environment.

Parameters can represent properties of the execution environment, data ports or events. They can be used by *Constraint Definitions* in order to set the specific assumption or guarantee. *Parameter Declarations* are used to specify that a variable of a specific data type may exist, but the concrete value has to be defined by the realizing *Contract Definition*. This can be useful for data arrays where the data points contained are individual for each component.

In the context of assumptions and guarantees, it is possible for a *Constraint Declaration* to set expected data types. The associated *Constraint Definition* must provide an expression where the resulting data type equals one of the expected types.

As we can see in Fig.2, we use the placeholders *Variable* for parameters, *DataType* for data types, and *Expression* for constraint expressions. These elements should be provided by a suitable constraint language or referable by the language that is used for the *Constraint Definition* expressions.

2) Finite State Machine for Contracts: Single contracts are sometimes not adequate for representing extra-functional properties. As we explain with our presentation in the following Section IV, cases exist where the behavior of a component - including extra-functional properties - changes over time or as a result of specific events. We thus expand the theory of contract-based design and capture such differences concerning contracts by applying the concept of a finite state machine. The idea is to have a finite state machine, where the single states may contain several currently valid contracts. The state machine itself operates on parameters provided by the environment or the internal states of a component.

Fig.3 illustrates our proposed metamodel for such a state machine. We again use the concept of declaration and definition in order to separate the specification and actual instance of a so-called contract state machine.

A *Contract State Machine Declaration* constitutes allowed *Contract Declarations*, concrete parameters and declarations of parameters which need to be defined by corresponding *Contract State Machine Definitions*.

Parameters are supposed to be used by *Contract State Machine Events* within constraint expressions, which trigger transitions to other *Contract State Machine States*. Such a state contains zero to infinite *Contract Definitions*.

Again, the metamodel elements *Variable*, *DataType* and *Expression*, refer to an arbitrary constraint language.

The actual semantics of a contract state machine depends on the target extra-functional properties and is determined by convention. It may be that entering a state implies that only those *Contract Definitions* it contains are valid. An alternative convention would be, that all visited *Contract Definitions* are valid except that a current *Contract Definition* overrides a former visited one by using the same *Contract Declaration*.

IV. Use Case

In this Section we show the application of our modeling concepts as presented on an exemplary use case from the automotive domain. First we give an overview of the target component and system. After that, we apply contracts together with a contract state machine. Finally, we discuss the use case presented.

A. Example - Electronic Steering Column Lock

Fig.4 illustrates a simplified electronic steering column lock (ESCL). Such locks are mandatory for cars in many countries. The Electronic Control Unit (ECU) decides whether to lock the steering column based on the input signals *Key State* and *Velocity*. These signals may be transmitted by a CAN bus or separate connections. If the ECU decides to lock the steering column, an actuator is activated which inserts the bolt into the steering column. Otherwise, the ECU decides to hold or eject the bolt.

There are several extra-functional properties which are worth considering in a system of this kind. In the following, we apply the modeling concepts presented for the extra-functional properties safety and timing. In the safety context we capture the data on whether the component ESCL is performing *normally*, is in a *failure* state, or *recovering* from a failure state. A failure state can be induced for instance by faulty transmitted data or other misbehaving components. Further to this we capture the data on how long it takes to execute the lock or unlock mechanism in two separate contract definitions.

B. Declarations

According to our metamodel concepts, the first step is to specify general declarations for components. Such declarations are known to contract checkers, interpreters or model transformers in advance. Fig.5 illustrates declarations for a

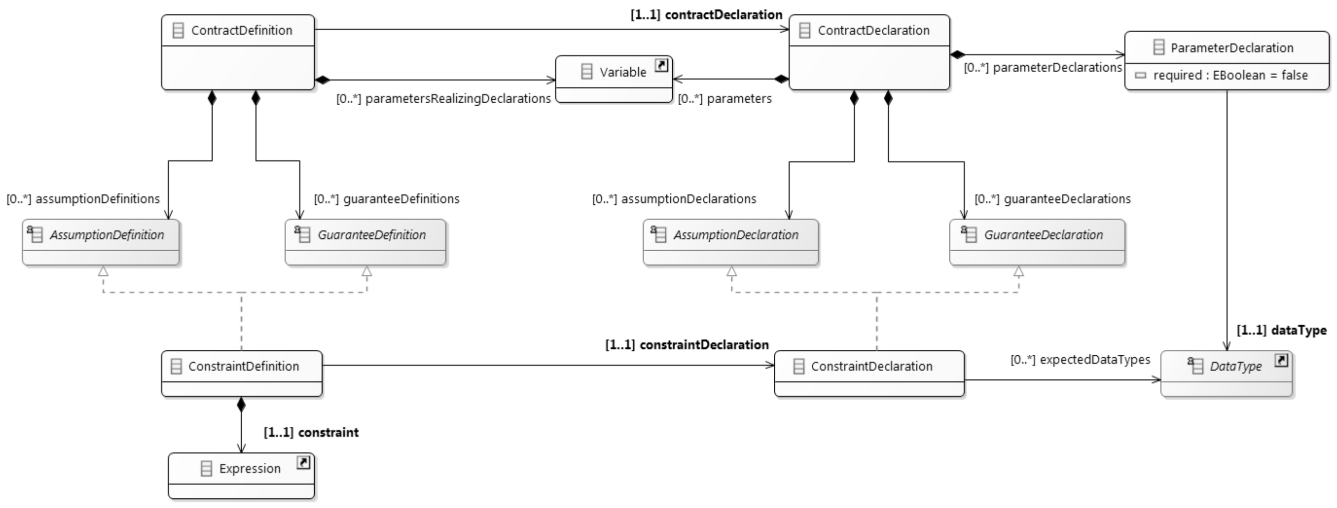


Fig. 2. Proposed Metamodel for Contract Declarations and Definitions

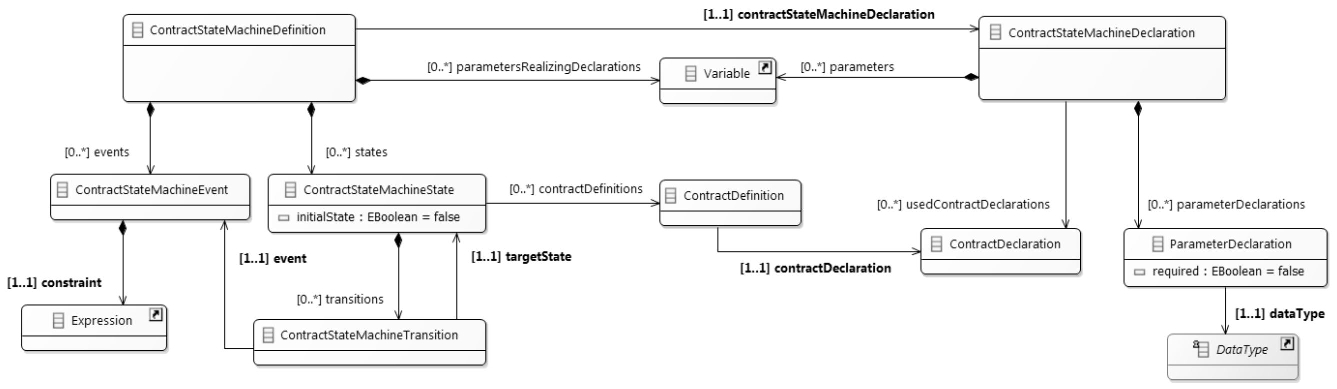

ContractStateMachineDefinition
[1..1] contractStateMachineDeclaration
ContractStateMachineDeclaration
[0..*] parametersRealizingDeclarations
Variable
[0..*] parameters
[0..*] events
[0..*] states
ContractStateMachineEvent
ContractStateMachineState
initialState : EBoolean = false
[0..*] contractDefinitions
ContractDefinition
[0..*] usedContractDeclarations
[0..*] parameterDeclarations
ContractDeclaration
ParameterDeclaration
required : EBoolean = false
[1..1] contractDeclaration
[1..1] constraint
[1..1] event
[1..1] targetState
[0..*] transitions
Expression
ContractStateMachineTransition
[1..1] dataType
DataType

Fig. 3. Proposed Metamodel for Contract State Machine Declarations and Definitions

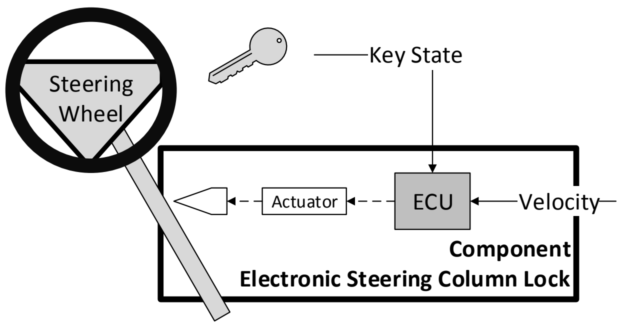
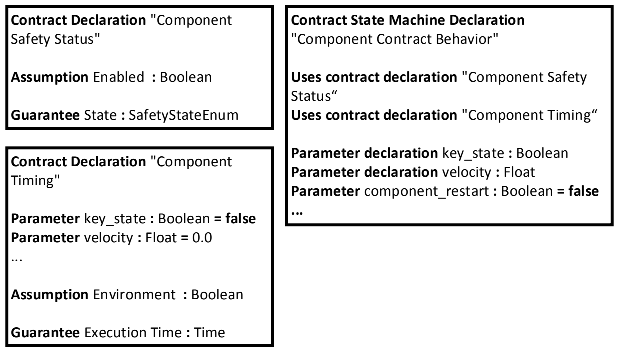


Fig. 4. Example Component - Electronic Steering Column Lock


Contract Declaration "Component Safety Status"

Assumption Enabled : Boolean

Guarantee State : SafetyStateEnum

Contract Declaration "Component Timing"

Parameter key_state : Boolean **= false**
Parameter velocity : Float = 0.0
...

Assumption Environment : Boolean

Guarantee Execution Time : Time

Contract State Machine Declaration "Component Contract Behavior"

Uses contract declaration "Component Safety Status“
Uses contract declaration "Component Timing“

Parameter declaration key_state **:** Boolean
Parameter declaration velocity **:** Float
Parameter component_restart **:** Boolean **= false**
...

Fig. 5. Use Case Declarations for the target extra-functional properties

component's safety status and timing. Additionally, we specify a contract state machine declaration that is used to capture the behavior of a component in order to set valid contracts.

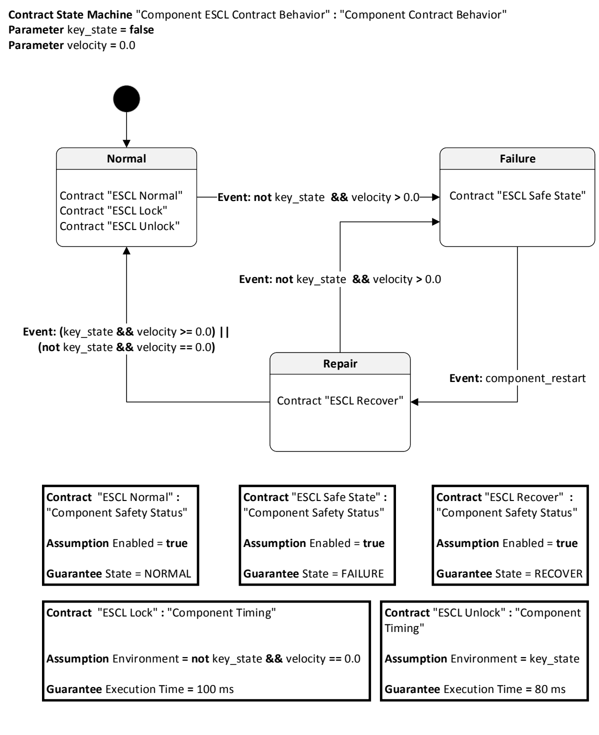
The contract declaration *Component Safety Status* assumes whether the component of interest is enabled and guarantees a certain safety state to the environment. The available types for this guarantee are restricted by the data type *SafetyStateEnum*, which contains the literals *NORMAL*, *FAILURE*, and *RECOVER* (not shown in Figure 5).

The contract declaration *Component Timing* is used to guarantee a specific execution time for certain assumed environments. The parameters *key_state* and *velocity* are provided by the analysis environment. The boolean parameter *key_state* indicates whether the ignition system is activated (boolean value true), while the parameter *velocity* states the current speed of the car. A comprehensive contract declaration would provide several other parameters, which may be obtained for instance by a CAN bus or observed from the condition of a system. The issue of which of these parameters are actually used by the assumption *Environment* depends on the component. When this assumption results in a boolean true, the guarantee *Execution Time* becomes valid.

Furthermore, contract definitions of these declarations can be used by the single states of the contract state machine *Component Contract Behavior*. Here again the parameters contained are obtained by the analysis environment or transmitted by the available connections. For instance, the parameter *component_restart* must be set by the analysis environment or by the described component. These parameters are used by a contract state machine definition in order to specify the events for state transitions.

C. Definitions

We now present how the declarations from above are used.



Contract State Machine "Component ESCL Contract Behavior" : "Component Contract Behavior"
Parameter key_state = false
Parameter velocity = 0.0
Normal
Contract "ESCL Normal"
Contract "ESCL Lock"
Contract "ESCL Unlock"
Event: not key_state && velocity > 0.0
Failure
Contract "ESCL Safe State"
Event: not key_state && velocity > 0.0
Event: (key_state && velocity >= 0.0) || (not key_state && velocity == 0.0)
Repair
Contract "ESCL Recover"
Event: component_restart
Contract "ESCL Normal" : "Component Safety Status"
Assumption Enabled = true
Guarantee State = NORMAL
Contract "ESCL Safe State" : "Component Safety Status"
Assumption Enabled = true
Guarantee State = FAILURE
Contract "ESCL Recover" : "Component Safety Status"
Assumption Enabled = true
Guarantee State = RECOVER
Contract "ESCL Lock" : "Component Timing"
Assumption Environment = not key_state && velocity == 0.0
Guarantee Execution Time = 100 ms
Contract "ESCL Unlock" : "Component Timing"
Assumption Environment = key_state
Guarantee Execution Time = 80 ms

Fig. 6. Contract State Machine and Contract Definitions of the ESCL Example

Fig.6 illustrates a contract state machine definition which sets the valid contract definitions according to the current state. The parameters are realizations of the parameter declarations declared by the contract state machine declaration *Component Contract Behavior* and are initialized to default values.

The initial state of this example is state *Normal*. Within this state, we can guarantee the execution time in respect to the locking and releasing mechanism. Furthermore, the contract *ESCL Normal* determines the safety state *NORMAL* to the environment. Whenever an abnormal event occurs such as there is no key but the car is moving, the contract state machine changes to the state *Failure*. In this state we cannot constitute the execution time of the ECSL and the contract *ESCL Safe State* becomes valid. After the component ESCL restarts, the state machine changes to the state *Repair*, which is reflected by the contract *ESCL Recover*. When the recover procedure was successful, the state machine changes to the state *Normal*, where the contained contracts become valid again, otherwise the state machine switches back to state *Failure*.

D. Discussion of the Use Case

We have shown how our contract modeling features can be used as presented on a simplified use case. It is imaginable that this example can be further advanced to capture the target and other extra-functional properties in more detail.

Note that we do not capture the actual functional behavior of the component ESCL. We rather use the functional behavior of the environment in order to determine how the target extra-functional properties timing and safety status of the component are changing and what guarantees are valid in that state. The semantics of the contract state machine we present is such that a new state invalidates the former visited contracts. The assumptions and guarantees of the *Contract Definitions* must be either automatically gathered by a measurement software or issued by humans.

Such a contract state machine can be used for two purposes.

One purpose is that a system becomes analyzable in advance, also with respect to composability. A model checker could simulate such a system and calculate the different expected safety states. Another model checker would be able to estimate the overall timing of a system.

The second purpose would be that a detection mechanism observes and constitutes the single states during runtime of a system and takes appropriate action based on predetermined contracts.

V. Conclusion and Future Work

In this paper, we presented concepts for modeling contracts and showed in a use case how these concepts can be applied.

The vision is to have a generic modeling language for specifying contract types and contract instances. By using the term generic we mean contracts that are suitable for at least a substantial number of extra-functional properties.

We introduced the concept of splitting a contract into a declaration and a definition. For analysis purposes a specific contract declaration would be known by a model checker or code generator beforehand. It declares the available parameters, assumptions and guarantees, while a contract definition uses such a declaration to define the actual behavior of a target extra-functional property.

Furthermore, we introduced the concept of a contract state machine which is basically a finite state machine where the single states represent different contract definitions. This concept is necessary, because a component may behave in different ways depending on the input data, environment properties or specific events. For instance, the timing of a component may be different depending on its previous processed data. It may also be different if the environment has changed. Such changes may require different valid contracts.

Concerning our future work, we are currently working on a configurable constraint modeling language, inspired by OCL [22], which we want to use for setting assumptions and guarantees. The idea is to have a constraint language where language elements, such as an if expression or a boolean operation, can be disabled and is afterwards not usable by an assumption or guarantee. This is useful, in our opinion, to simplify the construction of contract checkers or interpreters, because not all concepts of an expression language need to be considered and handled properly. It would also provide a user with direct feedback concerning what language elements are allowed for use.

Additionally, the presented modeling features for contracts do not consider composition, refinement, and conjunction of contracts as described theoretical by Benveniste et al. [7]. We are still working on finding pragmatic and usable metamodel solutions for these concepts.

After building this in a form suited to our use case metamodel for contract-based design, we are planning to develop a thin generic UML profile [23] for contracts and contract state machines.

This profile will be aligned with the existing OMG specifications MARTE [24] and SysML [25]. As mentioned by Selić and Gérard [26], a natural complementarity exists between these two profiles. We are of the view that a UML profile for contract-based design would benefit from concepts such as the physical types of MARTE or the constraint blocks of SysML. Not using such existing and standardized modeling concepts would be like reinventing the wheel.

The advantages of such a UML profile for contracts could be manifold. The most important one is, that it would allow the rise of specialized analyzing tools of different vendors which target single extra-functional properties. The input of such tools would depend, in such an ideal ecosystem, on the same UML profile for contract-based design.

References

[1] C. Ebert and C. Jones, "Embedded Software: Facts, Figures, and Future," *Computer*, vol. 42, no. 4, Apr. 2009.

[2] P. Feiler, J. Hansson, D. de Niz, and L. Wrage, "System Architecture Virtual Integration: An Industrial Case Study," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, CMU/SEI-2009-TR-017, 2009.

[3] M. Miller, *The Internet of Things: How Smart TVs, Smart Cars, Smart Homes, and Smart Cities Are Changing the World.* Pearson Education, 2015.

[4] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of things: Vision, applications and research challenges," *Ad Hoc Networks*, vol. 10, no. 7, Sep. 2012.

[5] I. Crnkovic, S. Sentilles, V. Aneta, and M. R. Chaudron, "A Classification Framework for Software Component Models," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, Sep. 2011.

[6] S. Sentilles, P. Štěpán, J. Carlson, and I. Crnković, "Integration of Extra-Functional Properties in Component Models," in *Component-Based Software Engineering*. Springer Berlin Heidelberg, 2009.

[7] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. L. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. Larsen, "Contracts for Systems Design," INRIA, Rennes, France, Tech. Rep., 2012.

[8] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems," *European Journal of Control*, vol. 18, no. 3, Jan. 2012.

[9] A. Jantsch, *Modeling Embedded Systems and SoCs: Concurrency and Time in Models of Computation.* San Francisco, Amsterdam: Morgan Kaufmann, 2004.

[10] N. Kajtazovic, "A Component-based Approach for Managing Changes in the Engineering of Safety-critical Embedded Systems," Ph.D. dissertation, Graz University of Technology, 2014.

[11] I. Crnkovic, M. Larsson, and O. Preiss, "Concerning Predictability in Dependable Component-Based Systems: Classification of Quality Attributes," in *Architecting Dependable Systems III.* Springer Berlin Heidelberg, 2005.

[12] P. Nuzzo, Huan Xu, N. Ozay, J. B. Finn, A. L. Sangiovanni-Vincentelli, R. M. Murray, A. Donze, and S. A. Seshia, "A Contract-Based Methodology for Aircraft Electric Power System Design," *IEEE Access*, vol. 2, 2014.

[13] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple Viewpoint Contract-Based Specification and Design," 2008.

[14] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, Oct. 1992.

[15] A. Rajan and T. Wahl, Eds., *CESAR - Cost-efficient Methods and Processes for Safety-relevant Embedded Systems.* Vienna: Springer Vienna, 2013.

[16] M. Maasoumy, P. Nuzzo, and A. Sangiovanni-Vincentelli, "Smart Buildings in the Smart Grid: Contract-Based Design of an Integrated Energy Management System," 2015.

[17] P. Nuzzo, A. Sangiovanni-Vincentelli, Xuening Sun, and A. Puggelli, "Methodology for the Design of Analog Integrated Interfaces Using Contracts," *IEEE Sensors Journal*, vol. 12, no. 12, Dec. 2012.

[18] N. Kajtazovic, C. Preschern, A. Höller, and C. Kreiner, "Constraint-Based Verification of Compositions in Safety-Critical Component-Based Systems," in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, ser. Studies in Computational Intelligence. Springer International Publishing, 2015.

[19] P. Nuzzo and A. Sangiovanni-Vincentelli, "Lets Get Physical: Computer Science Meets Systems," in *From Programs to Systems. The Systems perspective in Computing.* Springer Berlin Heidelberg, 2014.

[20] F. Warg, B. Vedder, M. Skoglund, and A. Soderberg, "Safety ADD: A Tool for Safety-Contract Based Design," in *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, Nov. 2014.

[21] O. Bridal, R. Mader, A. Geven, E. Schoitsch, H. Martin, M. Larramendi, A. Aristimuno, A. Fritsch, E. Vaumorin, M. Bordin, A. Solinas, A. Martelli, I. Korago, A. Levcenkovs, F. Joakim, R. Land, A. Söderberg, P. Conmy, and M. Illarramendi, "State-of-practice and state-of-the-art agreed over workgroup," Tech. Rep., 2011. [Online]. Available: http://www.safecer.eu/images/pdf/pSafeCer_D1.0.1StateOfThePracticeAndTheArt.pdf

[22] Object Management Group (OMG), "Object Constraint Language Version 2.4," 2014. [Online]. Available: http://www.omg.org/spec/OCL/2.4/

[23] ——, "Unified Modeling Language (UML)," 2015. [Online]. Available: http://www.omg.org/spec/UML/Current

[24] ——, "UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems Version 1.1," 2011. [Online]. Available: http://www.omg.org/spec/MARTE/

[25] ——, "OMG Systems Modeling Language (OMG SysML) Version 1.3," 2012. [Online]. Available: http://www.omg.org/spec/SysML/1.3/

[26] B. Selić and S. Gérard, *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE*, 2014.

Transforming Platform-Independent to Platform-Specific Component and Connector Software Architecture Models

Jan Oliver Ringert[2], Bernhard Rumpe[1] and Andreas Wortmann[1]
[1] Software Engineering, RWTH Aachen University, http://www.se-rwth.de/
[2] School of Computer Science, Tel Aviv University, http://www.cs.tau.ac.il/

Abstract—**Combining component & connector architecture description languages with component behavior modeling languages enables modeling great parts of software architectures platform-independently. Nontrivial systems typically contain components with programming language behavior descriptions to interface with APIs. These components tie the complete software architecture to a specific platform and thus hamper reuse. Previous work on software architecture reuse with multiple platforms either requires platform-specific handcrafting or the effort of explicit platform models. We present an automated approach to transform platform-independent, logical software architectures into architectures with platform-specific components. This approach introduces abstract components to the platform-independent architecture and refines these with components specific to the target platform prior to code generation. Consequently, a single logical software architecture model can be reused with multiple target platforms, which increases architecture maturity and reduces the maintenance effort of multiple similar software architectures.**

I. Introduction

Component & connector (C&C) architecture description languages (ADLs) [1] combine component-based software engineering with model-driven engineering (MDE) to describe complex software systems as interacting components. Describing component behavior with modeling languages enables to model great parts of software architectures platform-independently. Complex systems, however, require components that interface with APIs to access operating system functions or hardware drivers. Describing the behavior of such components with abstract modeling languages is hardly feasible. Instead, their behavior usually is defined in terms of general purpose programming languages (GPLs). Using GPL components in an architecture ties it to these GPLs and the interfaced APIs. This hampers reuse with different platforms.

Current approaches to generative MDE with C&C ADLs either do not take multi-platform reuse into account [2]–[6] or require explicit platform models [7]–[9]. The former requires duplicating the software architecture and changing the affected components manually, which introduces maintenance and evolution efforts as the duplicated architectures need to be fixed and progressed. The latter introduces complex notions to describe models of the target platform and the mapping of components to it. This introduces efforts in definition, maintenance, and evolution of platform models.

We present an approach to transform platform-independent, logical software architectures into platform-specific architectures of the same modeling language prior to code generation. With this, single logical software architectures can be reused with similar target platforms easily. This approach exploits the black-box nature of components by introducing *abstract components*. These provide stable interfaces to the software architecture, but omit behavior implementations to act as extension points for platform-specific components. Hence, generation of executable systems from such architectures is impossible. Prior to code generation, the abstract components are thus bound to compatible platform-specific components and the software architecture is transformed accordingly. The resulting platform-specific architecture is a well-formed, type-safe model available to further analyses and existing code generators can transform it into executable systems.

Our approach is implemented with the MontiArc-Automaton [10]–[12] C&C ADL and introduces a modeling language to describe *bindings* of software architecture models as well as different *library types*. It builds upon previous work presented in [13] and presents the following improvements:

- architectures and bindings are transformed to type-safe architectures before code generation instead of relying on special annotations of the abstract syntax,
- binding to platform-specific components may add platform-specific parameters,
- code generators need not be aware of replacement of implementations as we transform the architecture prior to code generation (generators process plain architectures),
- code libraries and library models are replaced with implementation libraries, which contribute platform-specific components instead.

This contribution presents the new approach and explains the model transformation to translate platform-independent architectures into platform-specific architectures. To this end, Sect. II describes the required preliminaries of MontiArc-Automaton before Sect. III motivates multi-platform generative MDE by example. Afterwards, Sect. IV introduces the new notions of bindings and libraries. Sect. V relies on these to describe the transformation from platform-independent to platform-specific software architecture models. Finally, Sect. VI discusses related work, including differences to our previous approach, and Sect. VIII concludes.

II. The MontiArcAutomaton C&C Architecture Modeling Framework

MontiArcAutomaton is a modeling framework for C&C software architectures with application-specific component behavior languages that features a powerful code generation framework. The modeling language comprises a C&C

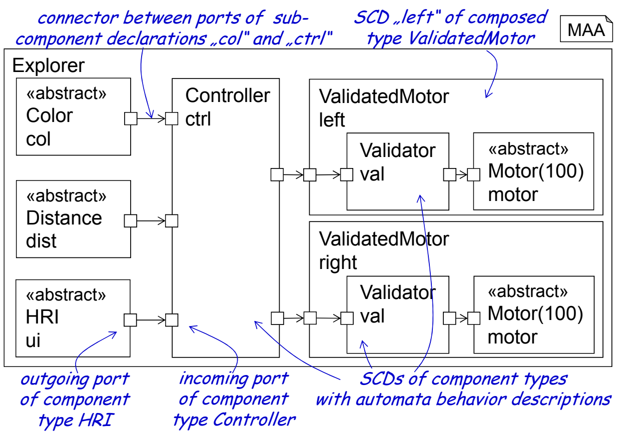
ADL [11], embeds a component behavior modeling language based on I/O$^\omega$ automata [11], and uses UML/P class diagrams [14] to model data types. It describes logically distributed software architectures in which components perform computations and connectors regulate communication. Components are black-boxes with stable interfaces of typed, directed ports and are either atomic or composed. Atomic components contain a component behavior description, either as a model of an embedded language [12], or as a reference to a GPL artifact. Composed components contain a hierarchy of subcomponents and their behavior emerges from subcomponent interaction. Components do not reveal whether they are composed or atomic or whether they feature a behavior model.

MontiArcAutomaton distinguishes component types from their instantiation and supports component configuration parameters. Component types define the interface and subcomponents of all their instances. Configuration parameters resemble constructors from object-oriented programming and serve component instantiation. Their arguments are passed by the containing component type. Component types may extend other component types and inherit their interfaces and component configuration parameters. Inheriting types may introduce new ports and configuration parameters. Atomic component types may extend composed component types and vice versa. Each atomic component type without behavior model is tied to a GPL behavior implementation - either via naming convention or explicit reference. Architecture models are parsed by MontiArcAutomaton, checked for well-formedness, and transformed into executable systems using generators for Java, Mona, and Python [10], [12].

III. Example and Problem Statement

Reusing the commonalities of C&C software architectures for multiple similar systems facilitates efficient modeling. Consider two robots for exploration of unknown areas: one cheap and for indoor educational purposes, the other expensive and rugged for outdoor missions. Both feature different sets of sensors to detect obstacles, actuators to propel two parallel motors, and a navigation to control the robot based on the sensors' inputs. The platform-independent base software architecture for such a robot is depicted in Fig. 1. It comprises a composed component type `Explorer` that declares three subcomponents `col`, `dist`, and `ui` for sensors, a navigation controller `ctrl`, and two subcomponents `left` and `right` to access the parallel motors. The latter are of composed component type `ValidatedMotor` which itself declares two subcomponents `val` and `motor` to validate inputs and access motor drivers. The subcomponent declarations (SCDs) of `left` and `right` parametrize their respective `motor` SCDs with argument `100` as the component type `Motor` requires an integer as configuration parameter.

The behaviors of component types `Controller` and `Validator` are modeled platform-independently with automata. Depending on the actual platform properties, the GPL behavior implementations of component types `Color`, `Distance`, `HRI`, and `Motor` differ. Therefore, they are declared *abstract* which prevents ties to platform-specific GPL behavior implementations. Reusing this software architecture with both platforms demands for integration of proper behavior implementations for subcomponent declarations of abstract component types. To achieve this under reuse of the existing code generators the following is required:



connector between ports of sub-component declarations „col" and „ctrl"
SCD „left" of composed type ValidatedMotor
MAA
Explorer
«abstract» Color col
«abstract» Distance dist
«abstract» HRI ui
Controller ctrl
ValidatedMotor left
Validator val
«abstract» Motor(100) motor
ValidatedMotor right
Validator val
«abstract» Motor(100) motor
outgoing port of component type HRI
incoming port of component type Controller
SCDs of component types with automata behavior descriptions

Fig. 1. C&C software architecture using abstract component types for different realizations of exploration robots. Port names and types are omitted for readability.

R1 Additional parametrization: platform-specific components might require additional configuration, such as the hardware port a sensor is connected to. Introducing this information to the base software architecture would tie it to specific platforms again. Hence, it may not be defined within the platform-independent software architecture.

R2 Behavior decomposition: Realizations of platform-specific components might be arbitrary complex and thus their decomposition is desired.

R3 Architecture validity: The resulting platform-specific architecture must be a valid MontiArcAutomaton model, hence the platform-specific behavior implementations for abstract component types must be compatible to the abstract component types' interfaces.

R4 Code generator compatibility: Retaining compatibility with existing code generators [10], requires integration to be performed completely prior to code generation and may not rely on generator specifics.

Exploiting the black-box nature of components to conceive subcomponent declarations of abstract component types as architecture extension points allows to fulfill these requirements with minor effort.

IV. Binding Platform-Independent Components

Our approach allows the development of logical, platform-independent architectures and their transformation to platform-specific ones by binding abstract SCDs to platform-specific component types. To this effect, the architecture modeler describes extension points for different platforms by using abstract component types from respective *model libraries*. Afterwards, she selects or develops proper *implementation*

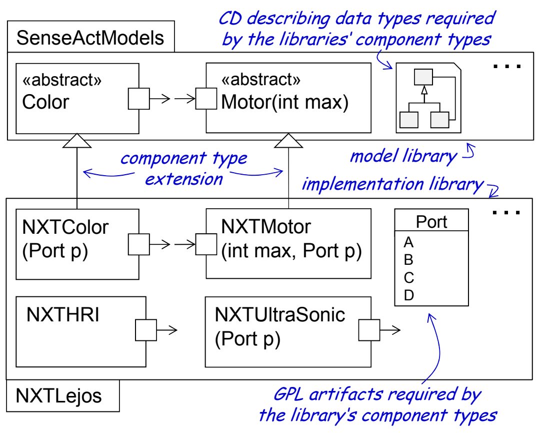

SenseActModels
CD describing data types required by the libraries' component types
«abstract» Color
«abstract» Motor(int max)
...
component type extension
model library
implementation library
NXTColor (Port p)
NXTMotor (int max, Port p)
Port
A
B
C
D
NXTHRI
NXTUltraSonic (Port p)
NXTLejos
GPL artifacts required by the library's component types

Fig. 2. Excerpt of the model library `SenseActModels` and the corresponding implementation library `NXTLejos` for NXT robots.

libraries that provide platform-specific realizations of the abstract component types. Modeling the *application configuration*, she defines how the SCDs of abstract types should be bound. Finally, MontiArcAutomaton processes the platform-independent software architecture, library components, application configuration model, and transforms the software architecture into a platform-specific model - without abstract components - according to the bindings. From this model, an executable system is generated.

Abstract component types are atomic and may not contain a behavior description, i.e., they are component interfaces with ports and configuration parameters. This follows the idea of abstract classes in object-oriented software engineering: they can be used during design time to describe properties expected from possible implementations, but they need to be extended and bound prior to code generation. To model a platform-independent software architecture, the abstract component types are imported from model libraries. Thus, a platform-independent software architecture may contain composed component types, atomic component types with behavior models, and abstract component types - all of which may use platform-independent data types only. Hence, the complete architecture is independent of GPLs and platforms.

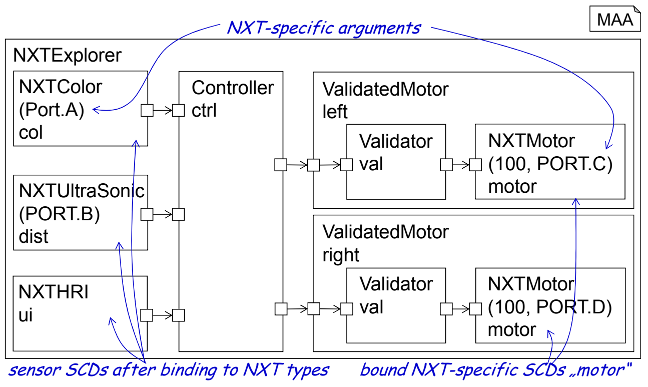
Similarly to software architectures, model libraries may only contain composed component types, component types with behavior model, abstract component types, and data types. This ensures that model libraries are platform-independent and consequently that the importing software architectures remain platform-independent as well. Abstract component types of model libraries are realized via extension by platform-specific component types of implementation libraries, which may also contain platform-specific data types. Fig. 2 illustrates the relation between abstract and platform-specific component types in the context of their libraries: The model library `SenseActModels` contains abstract component types for sensors and actuators as well as class diagrams describing the required data types. The implementation library `NXTLejos` contains the platform-specific component types `NXTColor` and `NXTMotor`, which extend the abstract component types `Color` and `Motor`, respectively. Similarly, `NXTHRI` and `NXTUltraSonic` extend the component types `HRI` and `Distance` of Fig. 1 assumed in `SenseActModels`. The `NXTMotor` also introduces a new configuration parameter of type `Port` that describes the physical port the motor's hardware is connected to. This type is specific to the NXT platform and thus not part of the abstract `Motor` interface but provided by `NXTLejos` instead. Component types for different platforms might require other configuration and thus extend `Motor` differently.

Implementation libraries are referenced by bindings defined in *application configuration models* [13]. These models describe how abstract SCDs will be bound before code generation. Such models reference a single software architecture and contain a set of *bindings*. These map the architecture's abstract SCDs to platform-specific, parametrized component types, such that the bound component types inherit from the SCD's component type and that the arguments match the bound component type's parameters. Hence, platform-specific parameters are part of the bound component type and the application configuration, but not of the platform-independent software architecture.

`ApplicationConfiguration`
```
1 import NXTLejosActuators.*;
2 application NXTExplorerApp for Explorer {
3   bind col to NXTColor(Port.A);
4   bind dist to NXTUltraSonic(Port.B);
5   bind ui to NXTHRI;
6   bind left.motor to NXTMotor(Port.C);
7   bind right.motor to NXTMotor(Port.D);
8 }
```
Listing 1. The application configuration `NXTExplorerApp` binds the abstract SCDs of architecture `Explorer` (Fig. 1) to platform-specific, parametrized types of `NXTLejos`.

Listing 1 illustrates the application configuration model `NXTExplorerApp`. It imports the implementation library `NXTLejos` (l. 1) before it declares its name and references the platform-independent software architecture `Explorer` (l. 2). Afterwards, it contains five bindings (l. 3-7) that describe how the abstract SCDs of `Explorer` should be replaced. Please note that the bindings for `left.motor` and `right.motor` (ll. 6-7) do not repeat the argument `100` passed to both `Motor` instances via their containing components (Fig. 1). Redefining arguments of the software architecture is prohibited and application configurations may define arguments for the platform-specific, bound component types only. Missing arguments are derived from the architecture and applied automatically.

With the libraries `SenseActModels` and `NXTLejos` and application configuration `NXTExplorerApp`, the platform-independent `Explorer` architecture can be transformed into the platform-specific software architecture depicted in Fig. 3. Here, the abstract component types used to describe the sensors and actuators have been bound to their platform-specific counterparts from the library `NXTLejos` and the



MAA
NXT-specific arguments
NXTExplorer
NXTColor (Port.A) col
Controller ctrl
ValidatedMotor left
Validator val
NXTMotor (100, PORT.C) motor
NXTUltraSonic (PORT.B) dist
ValidatedMotor right
Validator val
NXTMotor (100, PORT.D) motor
NXTHRI ui
sensor SCDs after binding to NXT types
bound NXT-specific SCDs „motor"

Fig. 3. NXT-specific architecture `NXTExplorer` with bound SCDs using platform-specific component arguments.

arguments defined in the application configuration model have been applied. With different implementation libraries and additional bindings, the `Explorer` software architecture can be used with multiple target platforms. Mapping SCDs to component types and with platform-specific configuration parameters entails the following, updated, notion of bindings: a *binding* is a mapping from an abstract SCD to a parametrized, platform-specific component type such that this type and its parameters are applied to the SCD. As such, it consists of a source, which identifies a SCD in the architecture's hierarchy to be replaced, and of a target, which describes how it is to be replaced. The latter consists of a platform-specific component type and configuration arguments.

A binding for a MontiArcAutomaton software architecture $\mathcal{A}$ is a tuple $(s, T(a_0, \dots, a_n))$, where:

- s is a qualified name in $\mathcal{A}$ that identifies a subcomponent declaration of abstract component type T_s with configuration parameters $p_0, \dots, p_k$,
- T is a platform-specific MontiArcAutomaton component type that inherits from T_s and possibly adds configuration parameters $p_{k+1} \dots, p_n$, and
- $a_0, \dots, a_n$ is a list of configuration arguments, such that a_i is of parameter type p_i.

Each element of $s = s_0 \dots s_m$ refers to a unique SCD name starting from $\mathcal{A}$ (MontiArcAutomaton prohibits multiple SCDs of the same name in the same composed component [15]). Examples of valid names in the software architecture depicted in Fig. 1 are `col`, `left.val`, and `right.motor`. We write a binding $(s, T(a_0, \dots, a_n))$ as s $\rightarrow$ T($a_0, \dots, a_n$).

This notion of bindings enables to add platform-specific arguments to the resulting software architecture without tying the platform-independent base architecture to target platform properties (Req. R1). Furthermore, bindings may map abstract SCDs to composed component types. Hence, complex platform-specific behavior can be expressed by multiple interacting components (Req. R2).

Given the software architecture depicted in Fig. 1 and the libraries illustrated in Fig. 2, the bindings `col` $\rightarrow$ `NXTColor()`, `left.motor` $\rightarrow$ `NXTMotor(10,Port.A)`, and `right.motor` $\rightarrow$ `NXTMotor(10,Port.B)` are valid bindings: the SCDs exist, the bound component types inherit from the SCDs abstract component types, and the arguments match. The following section describes how bindings are applied to a software architecture.

V. Binding Transformation

Bindings are defined in application configuration models (cf. Lst. 1) that are processed by MontiArcAutomaton prior to code generation. These models are checked for well-formedness to ensure each bound SCD is abstract, bound exactly once, the component it is bound to extends the SCD's component type, and the passed arguments are valid. Nevertheless, bindings bind abstract SCDs – not component types – to platform-specific types and binding a SCD of a specific type differently is desirable and supported. Naively, this entails a component type with a single SCD of different component types – which conflicts with the notion of types in MontiArcAutomaton. Our binding transformation resolves these conflicts.

MontiArcAutomaton requires that SCD `motor` of component type `ValidatedMotor` has the same type in each instance of `ValidatedMotor`. Fig. 4 illustrates this with an excerpt of component type `Explorer` that shows the subcomponent declaration `left` and `right` of component type `ValidatedMotor` after applying the bindings `left.motor` $\rightarrow$ `NXTMotor(10,Port.A)` and `right.motor` $\rightarrow$ `ROSMotor(10,Port.B)`, where `ROSMotor` is a component type applicable to be bound to `right.motor`. Afterwards, the component type `ValidatedMotor` is supposed to have a SCD `motor` of type `NXTMotor` (via `ValidatedMotor left`) and a SCD `motor` of type `ROSMotor` (via `ValidatedMotor right`). This naive transformation makes the type `ValidatedMotor` and with it the complete architecture invalid. We denote such type inconsistencies as *clashes*: There is a clash between two bindings $b_0 \dots b_n \rightarrow T_b\left(a_{b_1}, \dots, a_{b_x}\right)$ and $c_0 \dots c_m \rightarrow T_c\left(a_{c_1}, \dots, a_{c_y}\right)$ if they bind a SCD of a common parent component type to different component instantiations, i.e., SCDs b_{n-1} and c_{m-1} have the same type, b_n and c_m have the same name but $T_b \neq T_c$.

Desired bindings might clash and resolution prior to applying bindings is crucial to the resulting software architecture's validity. The following procedure takes care of clashes by replacing the types of all SCDs with new, unique types. To apply bindings, it conducts a breadth-first search through the component hierarchy defined by the root component type. During this search, the types and arguments of bound SCDs are replaced according to the bindings, i.e., bound. The types of unbound SCDs are replaced by copies of their original types with new and unique names to prohibit clashes. The corresponding procedure is depicted in Lst. 2.

Given a root component and a set of bindings, the procedure `BIND` visits all SCDs and either binds these according to the bindings or replaces their type with a new, unique type based

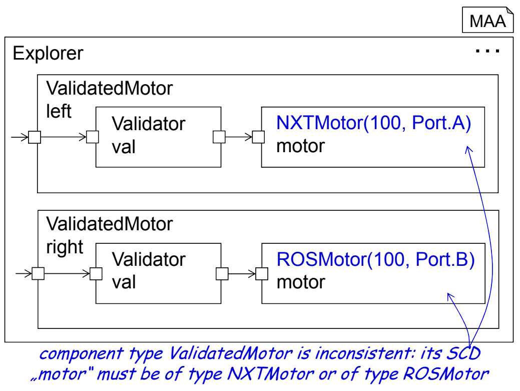


Fig. 4. Example for a clash between the two bindings `left.motor` → `NXTMotor(10,Port.A)` and `right.motor` → `ROSMotor(10,Port.B)`, which our transformation resolves.

```
                                       Pseudocode
1  BIND(ComponentType root, Bindings b)
2    Stack stack = new Stack()
3    newRoot = uniqueCopy(root)
4    stack.put("", newRoot)
5    while not s.isEmpty()
6      (pre, cmp) = stack.pop()
7      for each SCD (name,type(args)) of cmp
8        q = (pre==""?name:(pre+"."+name))
9        if b(q).exists()
10         type = b(q).type
11         args = append(args, b(q).args)
12       else
13         type = uniqueCopy(type)
14         stack.put(q, type)
15   return newRoot
```

Listing 2. The procedure `BIND` replaces the types of all SCDs with either bound types or new, unambiguous types.

on the original one. To this effect, the procedure utilizes a stack of tuples of names and component types. Initially the stack contains only the empty qualified name and a copy of the architecture's root component (ll. 2-4). The copy's type name is ensured to be unique by function `uniqueCopy()`. Afterwards it iterates over the stack's tuples and (ll. 5-14) inspects every SCD of the currently visited component type (such as `ValidatedMotor`). The qualified name `q` is updated with the current prefix and concatenated with the actual SCD's name using a ternary operator (l. 8, for instance to `left.val`) and it is checked whether a binding for the SCD indicated by `p` exists (l. 9). If a binding exists, the type and the arguments of the actual SCD are changed accordingly (ll. 10-11). As the replaced SCD's type must be abstract (and hence atomic) and the replacing component type must be platform-specific (it may be composed but not contain abstract SCDs), visiting the bound new component type is not necessary. In case there is no binding for the actual SCD, its type is set to a unique (in terms of its name) copy of itself (ll. 13-14). Finally, the currently updated hierarchy, as defined by `newRoot`, is returned (l. 15) for further analyses and code generation.

This procedure can be performed prior to any code generation and returns a valid MontiArcAutomaton software architecture (Req. R3) that describes the platform-specific architecture completely. Hence, the architecture can be processed by existing code generators without need for modifications (Req. R4). The procedure prohibits clashes but produces new component type definitions (l. 3 and l. 13) for each non-abstract subcomponent declaration. The number of new component types is thus bound by the number of subcomponent declarations. Whether this influences the number of artifacts in the generated system however depends on the employed code generators and their translation from component types to artifacts.

VI. Related Work

The presented approach is related to our previously introduced approach, deployment modeling, and other ADLs.

Our previous approach [13] relied on exchanging behavior implementation GPL artifacts instead of component types. Consequently, it could not produce software architecture models employing with different platform-specific component types. The architectures' components referenced to different behavior GPL artifacts instead. This prohibited to introduce new arguments to SCDs. Exploiting the notion of component inheritance lifts bindings completely to model level and enables such arguments while retaining a type-safe architecture. Handling references to different behavior GPL artifacts is no concern for code generators anymore and with code libraries, the library property models of [13] have become obsolete as well. These models described which abstract component types the contained behavior implementations belong to and identified the required run-time system (the GPL machinery required to enable system execution [12]). Now both is made explicit in the component types via inheritance and a new component property. Hence, libraries can also contain platform-specific component types for multiple run-time systems.

Bindings are related to deployment of C&C architectures to specific platforms [9], [16], but differ in the level of abstraction: deployment maps components to elements of the participating platforms and thus requires explicit platform models. Additionally, deployment may consider proper code generation for specific target platform elements, proper realization of connectors between physically distributed components, or mechanical and electrical properties of the target platforms. This imposes platform expertise on the application modeler.

The xADL [17] encourages including implementation details in component models. While omitting this allows describing platform-independent architectures, we are unaware of any similar pre-generation transformation. Relations to other ADLs and "abstract platforms" of MDA [7] are discussed in [13].

VII. Discussion

Application configuration models specify single bindings per SCD. For large architecture this is inconvenient, but can easily be solved by binding abstract component types and calculating the actually affected SCDs. This however is only part of improving the application configuration modeling language: additional features under consideration are conditional expressions over architecture properties and rewiring connectors for multiple interconnected bound component types. Also, interfaces of abstract component types need to be broad enough to support arbitrary platform-specific component types. They are by design, as the software architecture defines what is required. Furthermore, we do not bind non-abstract component types. While possible with this approach, this allows changing the architecture beyond recognition. This is not yet intended. Furthermore, we currently do not allow to bind SCDs of composed component types. While interesting, this leads to issues for abstract composed component types that contain abstract component types. The procedure BIND retains the processed software architecture's validity by introducing new component types to avoid clashes. Consequently, the resulting architecture contains redundant component types. We currently investigate a less invasive procedure that iteratively detects clashes and solves these introducing new components types only where necessary.

VIII. Conclusion

We have presented an enhanced approach to transform platform-independent into platform-specific software architectures. This approach builds upon previous work [13] and lifts it to model level completely. It applies bindings from abstract SCDs to parametrized, platform-specific component types of a software architecture and produces a valid software architecture again. The presented procedure is type-safe, allows to incorporate platform-specific configuration, reduces the complexity of MontiArcAutomaton code generators, and enforces a strict separation between platform-independent and platform-specific constituents. We are currently investigating the expressiveness of the new approach in further case studies.

References

[1] N. Medvidovic and R. Taylor, "A Classification and Comparison Framework for Software Architecture description languages," *IEEE Transactions on Software Engineering*, 2000.

[5] C. Schlegel, A. Steck, and A. Lotz, "Model-Driven Software Development in Robotics : Communication Patterns as Key for a Robotics Component Model," in *Introduction to Modern Robotics*, 2011.

[2] M. Geisinger, S. Barner, M. Wojtczyk, and A. Knoll, "A Software Architecture for Model-Based Programming of Robot Systems," *Advances in Robotics Research*, 2009.

[3] R. Bischoff, T. Guhl, E. Prassler, W. Nowak, G. Kraetzschmar, H. Bruyninckx, P. Soetens, M. Haegele, A. Pott, P. Breedveld, *et al.*, "BRICS - Best practice in robotics," in *Robotics (ISR), 2010 41st International Symposium on and 2010 6th German Conference on Robotics (ROBOTIK)*, VDE, 2010.

[4] D. Cassou, P. Koch, and S. Stinckwich, "Using the DiaSpec design language and compiler to develop robotics systems," in *Proceedings of the Second International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2011)*, 2011.

[6] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley, 2012.

[7] J. P. Almeida, R. Dijkman, M. van Sinderen, and L. F. Pires, "Platform-independent modelling in mda: supporting abstract platforms," in *Model Driven Architecture*, 2005.

[8] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, "RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications," in *Simulation, Modeling, and Programming for Autonomous Robots*, 2012.

[9] N. Hochgeschwender, L. Gherardi, A. Shakhirmardanov, G. K. Kraetzschmar, D. Brugali, and H. Bruyninckx, "A Model-Dased Approach to Software Deployment in Robotics," in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, IEEE, 2013.

[10] J. O. Ringert, B. Rumpe, and A. Wortmann, "From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems," in *Software Engineering 2013 Workshopband*, 2013.

[11] ——, *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Shaker Verlag, 2014.

[12] J. O. Ringert, A. Roth, B. Rumpe, and A. Wortmann, "Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems," in *1st International Workshop on Model-Driven Robot Software Engineering (MORSE 2014)*, 2014.

[13] J. O. Ringert, B. Rumpe, and A. Wortmann, "Multi-Platform Generative Development of Component & Connector Systems using Model and Code Libraries," in *1st International Workshop on Model-Driven Engineering for Component-Based Systems (ModComp 2014)*, 2014.

[14] M. Schindler, *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Shaker Verlag, 2012.

[15] A. Haber, J. O. Ringert, and B. Rumpe, "MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems," RWTH Aachen, Tech. Rep., 2012.

[16] L. Lednicki, I. Crnkovic, and M. Zagar, "Towards automatic synthesis of hardware-specific code in component-based embedded systems," in *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, 2012.

[17] E. M. Dashofy, A. Van der Hoek, and R. N. Taylor, "A highly-extensible, xml-based architecture description language," in *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, IEEE, 2001.

A Modular Reference Structure for Component-based Architecture Description Languages

Misha Strittmatter, Kiana Rostami, Robert Heinrich and Ralf Reussner
Chair for Software Design and Quality (SDQ)
Karlsruhe Institute of Technology
Karlsruhe, Germany
{strittmatter | rostami | heinrich | reussner}@kit.edu

***Abstract*—Metamodels are used to define languages, code generation and they serve as data structures for metamodel-centric software systems. In software engineering, these metamodels are crafted, evolved and extended, e.g., by further quality dimensions or structural features. However, an ad-hoc modeling approach does not properly support metamodel reuse by extension or composition. Nor does it enforce a proper modularization which helps with tackling complexity. We present an approach to design and extend metamodels for component-based architecture description languages in a modular way. The information which is to be metamodeled is divided into paradigm, domain, quality and analysis content. We constrain the usage of dependencies and give instructions how to modularize in accordance to concerns. Related approaches try to modularize and compose transformations, generators, and tools in general. However, in the field of metamodels, little support is given. Our approach is applied to several concerns of the Palladio Component Model and an extension thereof.**

I. Introduction

In model-based software engineering (e.g., model-driven software development or software performance engineering) and in general in many fields of computer science, software is described using models. These models capture different aspects like the object-oriented design, more coarse-grained architecture, deployment and so forth. Each discipline has its own focus and may add more information to this foundation. E.g., design decisions, implementation documentation, requirements and quality related information like service level agreements for performance or security.

A *metamodel* is a model which defines the structure of other models. If a model conforms to a metamodel, the model is considered an *instance* of the metamodel. Thus, metamodels are similar to grammars, as they define languages. A prominent example is the Unified Modeling Language (UML) metamodel [12]. A sequence diagram (instance) is a model which is an instance of the UML metamodel. Our approach primarily targets MOF [13] (i.e., the meta-metamodel of UML) conforming metamodels. However, we expect that it also directly applicable to metamodels conforming to meta-metamodels which feature similar concepts to: classes, containment-, inheritance- and association relations between classes.

This work was supported by the Helmholtz Association of German Research Centers and the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution.

There are multiple ways to found metamodel-based languages: 1) a new metamodel is developed. 2) an existing metamodel (e.g., UML) is extended by annotations or stereotyping. 3) a variant or branch of an existing language is created. The development of new metamodels is straightforward, if they do not evolve and are isolated. However, this is seldom the case. The major problem is that growing metamodels structurally degrade over time. Extensions by annotation or stereotyping are problematic as they may result in a flat, unstructured organization of information. Branches and variants are problematic, because duplicated parts have to be maintained when the original language evolves.

An example of this is the Palladio Component Model (PCM) [1]. It is a metamodel-based language which was initially developed for the specification of component-based software architectures and their resource demands to be able to predict their performance. With time, the research focus broadened and more structural features and quality dimensions were incorporated. Some of this information was directly built into the language [2]. Other aspects were specified as extensions or wound up in branches (e.g., the integration of business processes modeling and analysis [3] as well as modeling and analysis of maintainability [4]). This impedes the structure and reuse potential of inner parts of the PCM.

There are approaches, which put forward modular, composable or extensible concepts which are more or less related to metamodels. These concepts include components [5], [6], classes of object-oriented design [7], domain specific and general purpose languages [8], transformations [9], generators [10] and simulators [11]. However, for metamodels, little support is given.

Our approach aims to tackle these problems with a reference structure for metamodels for architecture description languages. The reference structure proposes a modularization of information into layers for paradigm, domain, quality and analysis information. The layers can further be divided: e.g., for separate quality dimensions or different domains. This leads to a modular, flexible and extensible structure, which satisfies separation of concerns and thus is better understandable and maintainable. It also increases the potential for reuse as a basis for new extensions. In addition, modularity leads to localization of change impact in the case of metamodel evolution. This does not only apply to the metamodel but to everything which is dependent on the metamodel (e.g., editors, analyzers, generators). The applicability of our reference structure is demonstrated on an selection of basic concerns and extensions of the PCM.

Our approach aims at the structuring of metamodels for the description of component-based software architecture and their qualities. However, we expect that it can also be applied to an even broader spectrum of metamodels, where the proposed decomposition is meaningful. These may be architecture description languages (ADLs) or even description languages of software-intensive systems in general.

This paper is outlined as follows: Section II describes the example scenario. Section III presents the reference structure and its concepts. Section IV applies the reference structure onto the example scenario. Section V presents related work. Section VI concludes the paper.

II. Problem Scenario

The scenario for our motivating example is the PCM. The PCM (i.e., a metamodel) is used for several analyses and simulations (see Figure 1). At the core of the PCM is a well formed construct to specify components, interfaces, and their composition. In the past, this part of the metamodel has served as a basis for new metamodel content which was emerging from new research. Initially this content was added directly to the PCM [14]. Examples of such intrusive extensions are reliability [15], event communication [16] and infrastructure components [17]. Later, as their number and diversity grew, extensions were no longer included directly in the PCM. They either came in the form of branches of the metamodel or metamodels which referenced into the PCM. Examples of such extensions are KAMP (Karlsruhe Architectural Maintainability Prediction) [4] and support for modeling business processes that interface with system services [3].

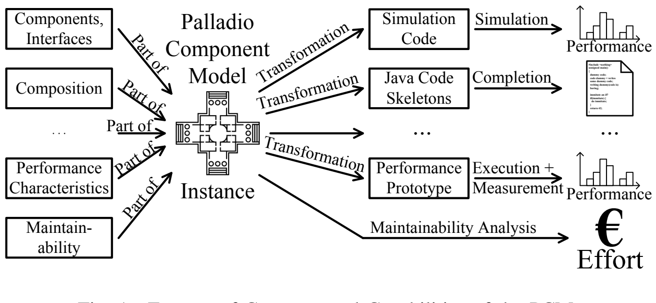


Fig. 1. Excerpt of Concerns and Capabilities of the PCM

Branches as well as intrusive extensions are problematic and have negative implication on users, developers, and researchers. Developers and researchers should have a clean base upon which they can build their extensions. If the metamodel is not modularized, it often contains content, which is irrelevant to the extension being made. This unnecessary complexity leads to a decline in understandability and may even lead to metamodeling mistakes. Further, both extension approaches are adverse to the maintainability of the metamodel. Intrusive extensions increase the complexity and external extensions often lack extension points in the base metamodel. Extension over time, regardless of intrusive or non-intrusive, increases either complexity or the amount of software artifacts that are dependent on the core metamodel. Thus, if the base metamodel is in need of restructuring but the restructuring is postponed, the cost of the refactoring will increase. This is especially critical in metamodel centric applications and can be called metamodel debt (cf. technical debt).

Negative effects of unstructured extensions on users are twofold. Users have specific needs with regards to the features of the program. They may be confused or overwhelmed when confronted with too much content irrelevant to them. Further, to support the additional metamodel content, the code of all extensions has to be shipped. For these issues, there are technical workarounds to minimize their negative impact. Shipped code of extensions may be reduced to a minimum by only including the model code. Tools and editors may be configurable to hide content which is not wanted by the user. However, this is only a workaround, as each extension needs to intrusively modify the tool in question.

For the means of the running example, the relevant concepts of the PCM will be explained in a condensed way (illustrated in Figure 2). For a complete and detailed description of the PCM, please consult the technical report [18]. At the core of the PCM are components and interfaces (I). Both are first class entities and a components may implement or require interfaces (II). Components can be placed within an architecture by creating an assembly context (III). The interfaces of these assembly contexts can be linked by connectors (IV). A component may be either atomic or composite (V). Composite components contain further assembly contexts and connectors. For the modeling of performance and reliability, atomic components contain abstractions of control flow (VI) (similar to activity diagrams or flow charts). These control flow specifications carry resource demands and failure probabilities in some of their states. To conduct performance simulation [19] of such a model, there is further information needed like the maximum simulation time or the maximum measure count.

Another part of the running example is KAMP [4]. It is an extension for the PCM that deals with maintainability. It is a tool-supported approach to semi-automatically predict change propagation in a software system. In contrast to other approaches, KAMP considers not only the architecture of a software system, but also organizational aspects (e.g., management or various roles such as tester or deployer) as well as technical artifacts (e.g., testing, build configurations, or deployment). KAMP comprises two phases: 1) *Preparation Phase*: After the user models a software architecture using the PCM (e.g., Figure 2), she or he enriches the model with context annotations such as test cases and build configuration. 2) *Change Request Analysis Phase*: After the user has modeled

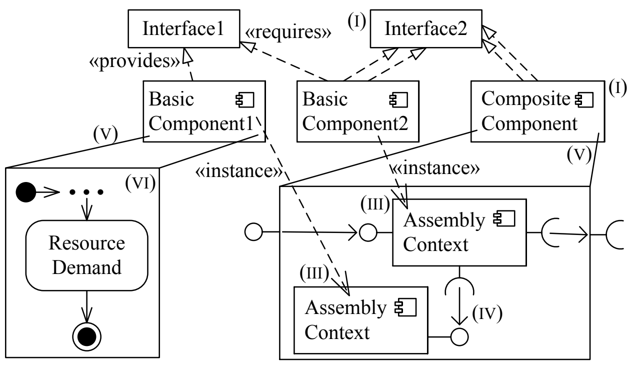


Fig. 2. Simplified Concepts of the PCM

the change in the enriched architecture model, KAMP semi-automatically calculates the change propagation and derives a task list, which is composed of tasks needed to implement the change request. This step calculates all affected artifacts annotated in the model, too.

The overall scenario of the running example is illustrated in Figure 3 and contains several main concerns: the PCM is a language to specify component-based software architecture as the common basis; performance characteristics, which are included in the PCM and partly in external sources (configurations of analysis launches); maintainability information, which is contained within an external extension. The PCM is internally structured. The primary decomposition runs along its view types (submodels). However, here it is displayed as one module with the exception of the performance results, which are stored in a modular way. This stems from the fact that due to dependency cycles within the PCM [2], it is only possible to use it as a whole. Thus, all the problems from above follow for developers and users: unnecessary complexity and its implications.

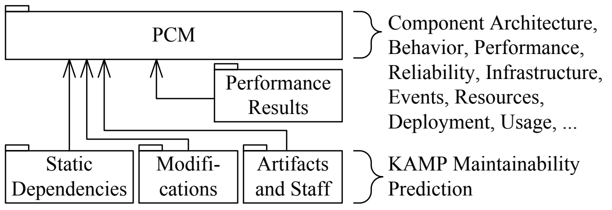

PCM
Performance Results
Component Architecture, Behavior, Performance, Reliability, Infrastructure, Events, Resources, Deployment, Usage, ...
Static Dependencies
Modifi-cations
Artifacts and Staff
KAMP Maintainability Prediction

Fig. 3. Status Quo of the Scenario: Metamodel Modules

These problems are tackled by our approach. By modularizing the meta content and structuring it according to our reference structure, the concerns are separated and the extensions have a clean base. In the following sections we will present the layers of the reference structure, then build an ideal and modular version of the scenario meta model according to the reference structure.

III. Reference Structure Overview

Within our reference structure, the metamodel is subdivided into smaller parts which we call metamodel modules or just *modules*. On the technical level, these modules may manifest in their own metamodels, each of which is persisted in its own file, or they may just manifest within the package structure or comparable subdivisions within one metamodel. The contents (e.g., classes) of one module may depend on the contents of another module. This is either in the form of simple reference, containment, inheritance, or stereotype application. If at least one dependency from one module to another exists, we will regard that as the one module being *dependent* on the other.

Our reference structure organizes modules into *layers* as the topmost unit of decomposition. A layer is a set of modules. The layers are ordered with regard to the dependencies of their modules. The modules of a layer may only depend on the modules of the same or more basic layers. More basic here means, that they depend on fewer other layers. Within the scope of this paper, basic layers will be illustrated at the top. If a module of a layer depends on a module of another layer, we will regard that as the one layer being dependent on the other one. Layers may depend on all higher layers. However, it is advisable to confine the dependencies on the next higher level where possible. Circular dependencies between modules (as well as between layers) are not allowed. A circular dependency is either a result of a dependency which should be reversed, or of a strong cohesion between the modules. These modules, thus, should be considered for merging or restructuring.

A module encapsulates a set of concerns. When modularizing, two main rules should be applied. It should be meaningful to use the basic module (or modules) with and without the concerns which have been factored out. If this is not the case, the modularization is still meaningful if the basic module serves as a common foundation for multiple further modules.

The information which is formalized in metamodels can be grouped into categories depending on the type of information. Multiple decomposition dimensions exist and it not always clear which decompositions to apply and in which order. Intuitive design might modularize information in ways of infrastructure (abstract class hierarchies) vs. concrete content, views types or sub models, or semantic cohesion. With our reference structure, we propose as the primary decomposition a layering into: paradigm, domain, quality and analysis content. *Paradigm* (π) is the most basic layer. It lays the the foundation of the language by providing structure but without semantics. The *domain* (Δ) layer builds upon the paradigm and assigns semantics to its abstract structure. The *quality* (Ω) layer adds quality properties to the domain concepts. Lastly, the *analysis* (Σ) layer provides modules for input-, output- and internal state and configuration options for analyses. We settled for these layers, because they provided an intuitive primary decomposition in our research when inspecting metamodels for component-based architecture description languages and their intrusive and external extensions. The concern constellations of these layers (whereas not as explicit as proposed by our reference structure) can be found in metamodels like UML MARTE, the Descartes Metamodel and the PCM.

IV. Applying the Reference Structure

In the following subsections, we will explain the layers of the reference structure and gradually construct a modular metamodel of our example (PCM extended by maintainability) according to these layers. Please keep in mind, that this is a simplification and a reconstruction of the PCM. Thus, modeling of concepts and dependency directions partly do not adhere to the current PCM.

A. Paradigm

The most basic layer is the paradigm layer (π). It defines foundational structure but without semantics. Thus, it is not directly usable. Not even for purposes which do not need dynamic semantics (e.g., documentation and communication). The minimal configuration of the metamodel which is meaningfully instantiable is π and the domain layer put together. First class entities in π should be abstract and no top level container (root) should be provided to avoid instantiation. Modules which cannot be instantiated directly, will be considered abstract modules. In our case π constitutes components, interfaces and their composition. However, any π is possible. It is dependent on the subject matter which is to be captured. E.g., object oriented design and behavioral formalisms.

In our example, π encompasses the modules CoreEntities and Composition. This is illustrated in Figure 4. Please keep in mind that for the sake of presentation this is an extremely simplified depiction. Wherein Composition is dependent on CoreEntities. CoreEntities contains the abstract metaclasses Component and Interface. Like in the standard UML notation, abstract classes and abstract modules are indicated by a name in italic letters. A Component may reference various Interfaces in two ways: provide and require. The Composition module defines ComposedStructures which contain AssemblyContexts and Connectors. Connectors link Interfaces of two AssemblyContexts. An AssemblyContext represents an instance of a Component.

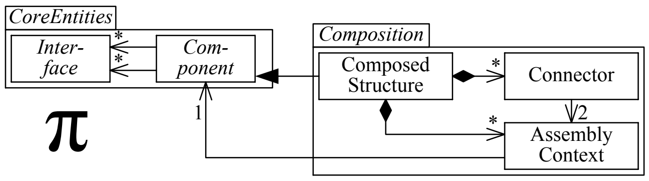


Fig. 4. The π (Paradigm) Layer

The ComposedStructure *extends* Component. In the illustration, the UML notation for stereotype application (a filled arrow) is used. However, the implementation of this relation does not necessarily have to be stereotyping. A wide range of different extension mechanisms is available with all their pros and cons. Further ones are annotation, plain inheritance, and the application of patterns like the decorator pattern or aspect-oriented extension [20].

The π layer builds the foundation of the remainder of the metamodel. It is important, that is has no outgoing dependencies into other layers. This way, it can be reused on its own.

B. Domain

The domain layer (Δ) extends π and assigns domain semantics to its abstract first class entities. By doing so, new information (i.e, attributes and relations) can be added to the derived classes. New domain concepts may be created as well. However, if these new domain concepts have an overlap with classes of other modules in Δ, or even of π it should be considered to factor that content out into a higher, more general module or even a higher layer (i.e., in this case π). In the scope of this paper, Δ will capture software systems. In general, any Δ layer is possible. E.g., embedded and mechatronic systems or cyber-physical systems. The Δ, however, has to fit the underlying π layer.

The Δ layer excerpt of the example scenario is illustrated in Figure 5. It features the ComponentRepository module, where the modeler can specify the Components and Interfaces of an architecture. Please keep in mind that the relation between Components and Interfaces is already defined in the π layer. The ComponentRepository module assigns domain semantics to the abstract CoreEntities module of the π layer providing concrete subclasses for its abstract classes. This separates the concerns of the structure of the paradigm (here component architecture) from its domain semantics, enabling the reuse of the paradigm structure. The Behavior module extends SoftwareComponents by a behavior abstraction, which is similar to a flowchart. The Behavior module is performance agnostic and is later used as a foundation for performance and reliability modeling. The Modification module is essential to the maintainability analysis. It defines Modifications of the architecture. Each concrete Modification references an element of the architecture. It further defines Propagations, which express how Modifications spread through an architecture.

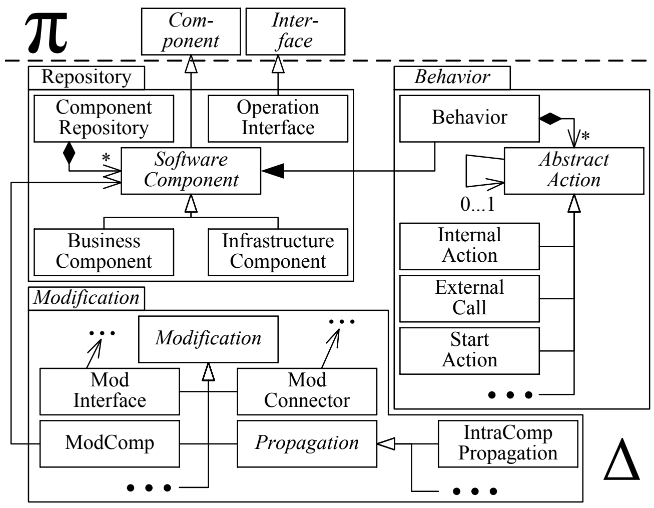


Fig. 5. Excerpt of the Δ (Domain) Layer

C. Quality

The quality layer (Ω) defines the inherent quality properties of Δ concepts. It contains primarily second class entities, which enrich first class entities of Δ. In the scope of this paper, the Ω layer constitutes performance and reliability characteristics. Possible other Ω modules which fit the π and Δ are: security and availability. Not in every circumstance a Ω layer is needed. E.g., when the metamodel is only needed for specification of software design or for static analysis. This is the case for the KAMP maintainability analysis, which does not require information in the Ω layer.

The quality properties have to be inherent and not derived. Inherent with respect to the analyses which can be performed on the model. E.g., when considering the PCM and performance, the response time of an operation depends on many things, like resource demands within the operation, response time of external calls and so on. Thus, the response time of an operation in this consideration is not an intrinsic value, but a derived one. Therefore, the response time of operations should be moved to a lower layer. There may be quality concepts which are needed to model derived properties. These should be specified in the Ω layer. However, their instances should be contained in the Σ layer. On the other side, the resource demand (not in terms of time but in load) is independent. It may be derived from an estimate or a real word software artifact, but with regards to the analyses it cannot be derived. Therefore, it belongs in Ω and should be contained and specified there. If a model is used for solving, analysis or simulation, the model content from Ω should remain static during execution. If that is not the case, the information belongs to the analysis layer, as it is derived or state information.

Figure 6 shows an excerpt of the exemplary Ω layer. It contains the modules for Performance and Reliability. They extend InternalAction, and therefore the internal behavior of a Component, by ResourceDemand and FailureOccurence. A FailureOccurence has a probability, as well as a type. Here, an important advantage of the modularization shows. The behavior metamodule is free of performance and reliability data. As required, it can be extended by one of the two, or even both. Tools and developers are not bothered by irrelevant content. Potential future extensions can be created without the need for prearrangement.

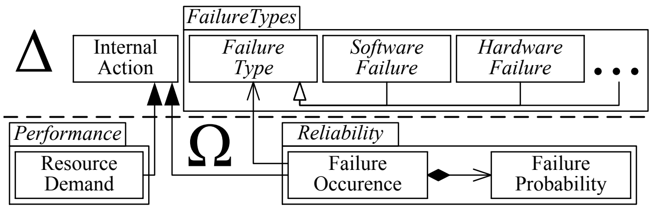


Fig. 6. Excerpt of the Ω (Quality) Layer

D. Analysis

The analysis layer (Σ) is only relevant, if models are used as a basis for analysis, solving or simulation (hereafter only referred to as analysis). Σ provides new views to specify input state, configuration, run-time state and output of an analysis. These are all possible views, but only a subset may be required by an analysis. It is possible for multiple analyses to be founded on the upper layers. Several analyses may share modules, but also posses their own ones. In the focus of this paper, the Σ layer is concerned with the performance simulation and the change impact analysis KAMP.

In Figure 7 an excerpt of the Σ layer is shown. As a really simplified version of the performance result, here, a set of OperationResponseTimes is modeled. An OperationResponseTime has a unit, which is specified in the Ω layer (the corresponding module was not shown in Figure 6). It further references the AssemblyContext and the Interface where the response time was recorded.

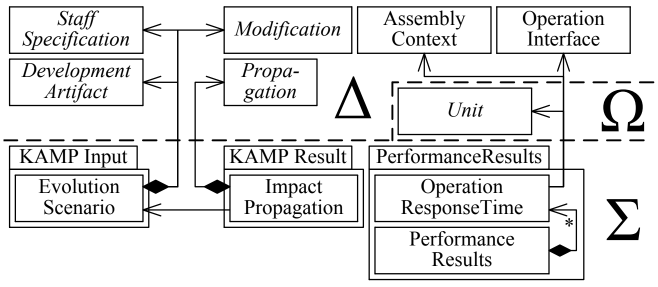


Fig. 7. Excerpt of the Σ (Analysis) Layer

The in- and output for the KAMP maintainability prediction is on the left side of Figure 7. In this circumstance, the quality layer is not needed. Both modules merely hold a root container and reference directly into the Δ layer. The EvolutionScenario of KAMP Input contains everything the analysis requires: seed Modifications, StaffSpecifications and DevelopmentArtifacts that belong to the architecture's elements. The ImpactPropagation contains the predicted extend of the architecture change in the form of Propagations. The ImpactPropagation further references the input of the scenario to enable inference of affected artifacts and staff.

E. Overall Module Structure

For an overview of all the modules of the scenario and their dependencies, see Figure 8. The modules which convey specific concerns are indicated by different levels of gray (maintainability is dark, performance is medium, reliability is light). The remaining modules cover more fundamental concerns: component-based architecture and behavior. They are the intersecting set of the metamodel elements of these analyzes. The illustration may seem much more complex, than the initial illustration in Figure 3. This is because in the initial figure, all the concerns of performance, reliability, behavior, and component architecture are contained in the big PCM module. The original structure cannot be subdivided into modules. At least not into modules according to the definition in this paper, as the subpackages of the PCM have cyclic dependencies (see [2]).

This modular structure brings many benefits. External extensions now have a clean base to build on. E.g., a security extension can now be developed without having to deal with performance and reliability. In the modular structure, extension developers have to only understand the modules they extend and to some degree modules from which they are indirectly dependent. This increases the potential of reuse of the more basic (higher) modules. When evolving the metamodel, change impact can be traced down the graph, following the depen-

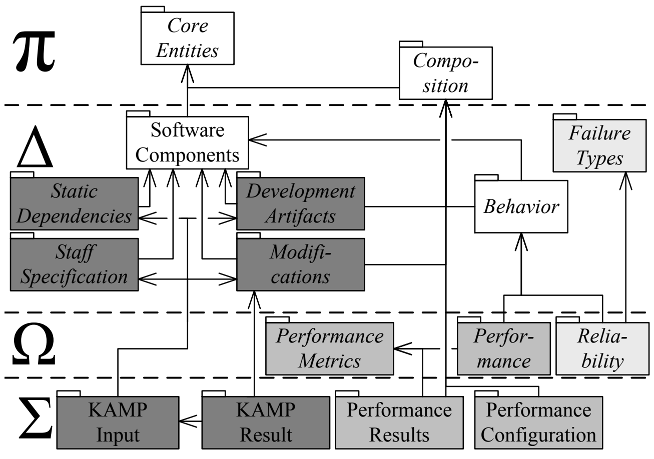


Fig. 8. The Scenario Metamodel Restructured In Concordance to the Reference Structure

dency relations. This way changes can be assessed much faster and more accurately.

V. Related Work

There are approaches, which deal with the modularization, categorization or structuring of related concepts. Coad's UML archetypes [7] for object-oriented design are used to classify classes into things, temporal concepts, roles and descriptions. Atkinson et al. propose a view-based approach for software engineering. The underlying model captures every concern into orthogonal dimensions, which are assessed through views [5]. Further, with deep modeling [21] they propose to enable instance relations within models. These deep models are layered with regards to instance levels. JetBrains MPS [8] features capabilities for the extension of DSLs. Siedersleben [6] proposed a reference structure for software architectures, where components are categorized into so-called blood types (technical, domain and library). Yet these approaches cannot be transferred to work on metamodels or they do do not offer support for metamodel extension and reuse.

There is also related work, which aims at modularizing concepts which are in direct interplay with metamodels. Jung [10] proposes a composition approach for generators. Rentschler [9] developed an approach for modular transformations. Föhrdes [11] presents a modularization into components of a performance simulator which operates on the PCM (metamodel). These approaches are especially interesting, as they deal with artifacts which are used in conjunction with metamodels. However, theses approaches cannot be directly transferred onto metamodels.

VI. Conclusion

In this paper, we proposed a reference structure for modular metamodels for component-based architecture description languages. It proposes a layering of the information into paradigm (π), domain (Δ), quality (Ω) and analysis (Σ) content. The reference structure is applied to the PCM [1] and the KAMP [4] maintainability extension. However, a remake of the PCM is not the contribution of this paper. These models were chosen to demonstrate the applicability of the reference structure. Our approach aims to be applicable to component-based architecture description languages which also express quality properties.

Our reference structure propose modularization into layers and further into modules as well as making the dependencies explicit. Dependencies are constrained to avoid dependency cycles and improve modularization. Guidance for the modularization with regards to concerns is given. The reward is a more modular metamodel which allows for better compositionality of extensions. The improved modularity leads to a reduced complexity and all its benefits: better understandability, maintainability and reusability.

Future work include an in-detail examination of the possible extension mechanisms which can be used for the extension relation. Also, we plan to develop decision support for the grouping of classes and modules into the layers. Further, an extensive metamodel will be remade according to the reference structure and its correctness and completeness proven (e.g., by finding an isomorphism). An interesting question worth investigating is if different types of extension with regards to the interplay of abstract and concrete module have implication onto potential roots elements of view types.

References

[1] S. Becker et al. "The palladio component model for model-driven performance prediction," *JSS*, Elsevier, 82(1):3–22, 2009.
[2] M. Strittmatter et al. "Identifying semantically cohesive modules within the palladio meta-model," *SSP*, 160–176, 2014.
[3] R. Heinrich et al. "Integrating business process simulation and information system simulation for performance prediction," DOI 10.1007/s10270-015-0457-1, *SoSyM*, Springer, 1–21, 2015.
[4] K. Rostami et al. "Architecture-based assessment and planning of change requests," *QoSA*, ACM, 21–30, 2015.
[5] C. Atkinson et al. "Orthographic software modeling: a practical approach to view-based development," *ENASE*, Springer, 69:206–219, 2010.
[6] J. Siedersleben *Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar*, dpunkt, 2004.
[7] P. Coad *Java modeling in color with UML*, Prentice Hall, 1999.
[8] M. Voelter et al. "Language modularity with the mps language workbench," *ICSE*, IEEE, 1449–1450, 2012.
[9] A. Rentschler "Model Transformation Languages with Modular Information Hiding," Ph.D. thesis, Karlsruhe Institute of Technology, 2015.
[10] R. Jung "Geco: Generator composition for aspect-oriented generators," *Doctoral Symposium - MODELS*, 2014.
[11] C. Föhrdes "Simulation components for software quality simulation in eclipse," Master's thesis, Karlsruhe Institute of Technology, 2014.
[12] OMG "UML Infrastructure Specification 2.4.1," 2011.
[13] OMG "MOF Core Specification 2.4.2," 2014.
[14] M. Strittmatter et al. "Towards a modular palladio component model," *SSP*, CEUR, 49–58, 2013.
[15] F. Brosch "Integrated software architecture-based reliability prediction for it systems," Ph.D. thesis, Karlsruhe Institute of Technology, 2012.
[16] C. Rathfelder "Modelling Event-Based Interactions in Component-Based Architectures for Quantitative System Evaluation," Ph.D. thesis, Karlsruhe Institute of Technology, 2013.
[17] M. Hauck "Extending Performance-Oriented Resource Modelling in the PCM," Diploma thesis, University of Karlsruhe, 2009.
[18] R. Reussner et al. "The Palladio Component Model," Tech. Rep., Karlsruhe Institute of Technology, 2011.
[19] M. Becker et al. "Performance analysis of self-adaptive systems for requirements validation at design-time," *QoSA*, ACM, 2013.
[20] R. Jung et al. "A method for aspect-oriented meta-model evolution," *VAO*, ACM, 19–22, 2014.
[21] C. Atkinson et al. "Melanie: Multi-level modeling and ontology engineering environment," *MW*, ACM, 7:1–7:2, 2012.