3\textsuperscript{rd} International Workshop on Interplay of Model-Driven and Component-Based Software Engineering (ModComp) 2016

Workshop Proceedings

Federico Ciccozzi, Ivano Malavolta (Editors)
Copyright © 2016 for the individual papers by the papers’ authors. Copying permitted for private and academic purposes. Re-publication of material from this volume requires permission by the copyright owners.

Editors’ addresses:

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

Ivano Malavolta
Software and Services – Vrije University (Netherlands)
# Table of Contents

Preface ........................................................................................................................................... 1

Keynote 1 – CBSE and MDE: Fitting the Pieces Together .............................. 4  
*Kung-Kiu Lau*

Keynote 2 – Component Models and Models of Components .................. 5  
*Jan Carlson*

Feature-Oriented Modelling in BIP: A Case Study ................................. 6  
*Cecylia Bocovich and Jo Atlee*

Towards a Meta-Model for Quality-aware Self-Adaptive Systems Design..... 12  
*Esma Maatougui, Chafia Bouanaka and Nadia Zeghib*

Consistent Extra-Functional Properties Tagging for Component and Connector Models ........................................................................................................... 19  
*Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe and Michael von Wenckstern*

Fault-aware Pareto Frontier Exploration for Dependable System Architectures . 25  
*Lukas Märtin, Hauke Baller, Anne Koziolek and Ralf Reussner*

Towards Concern-Oriented Design of Component-Based Systems .......... 31  
*Jörg Kienzle, Anne Koziolek, Axel Busch and Ralf Reussner*

Performance Modeling in the Age of Big Data: Some reflections on current limitations .......................................................................................................................... 37  
*Robert Heinrich, Holger Eichelberger and Klaus Schmid*
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 Interplay of Model-Driven and Component-Based Software Engineering 2016 (ModComp’16) 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’16 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 19th 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 interactions for quality assessment;
- concern-oriented modeling of components;
– modeling for self-adaptive systems
– modeling languages for components.

This was the third 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 2016
Federico Ciccozzi and Ivano Malavolta
Keynote 1

CBSE and MDE: Fitting the Pieces Together

Kung-Kiu Lau
University of Manchester

MDE and CBSE have not developed in tandem, at least in my opinion. However, it is clear that they can work together to their mutual benefit. In this talk I will give my take on their potential symbiosis and synergy, and illustrate it with my own views and experience.

Kung-Kiu Lau holds a PhD degree from the University of Leeds, UK. After a temporary appointment at Leeds, he moved to the University of Manchester, UK, where he is a senior lecturer. He is the series editor of a book series on Component-based Software Development and an area editor of the Journal of Applied Logic. He has served on numerous programme committees (e.g. the International Automated Software Engineering Conference, the International Symposium on Component-based Software Engineering, the International Symposium on Software Composition, and the International Conference on Generative Programming and Component Engineering). Similarly, he has delivered invited talks and tutorials at many international meetings (e.g. Twentieth IEEE/ACM International Conference on Automated Software Engineering 2005 and the Twenty-eighth International Conference on Software Engineering 2006). His main research areas are Component-based Software Development and Formal Program Development in Computational Logic.
Keynote 2

Component Models and Models of Components

Jan Carlson
Mälardalen University

We have worked in the intersection of component-based and model-based development, mostly targeting the domain of embedded systems, in a number of research projects ranging from development of new component models to model-level timing analysis and safety certification. This talk will present a few selected challenges as well as our general view of how a combination of models and components can be exploited in this domain.

Jan Carlson is an associate professor at Mälardalen University, Sweden. He received his M.Sc. degree in Computer Science from Linköping University in 2000, and his doctoral degree from Mälardalen University in 2007. His current research focuses on component- and model based development of embedded systems, addressing areas such as architectural decision support, allocation optimization, model-level timing analysis, and code generation. Other research interests include event pattern detection, and analysis of shared stack usage.
Feature-Oriented Modelling in BIP: A Case Study

Cecylia Bocovich  Joanne Atlee
University of Waterloo
Email: {cbocovic, jmatlee}@uwaterloo.ca

Abstract—In this paper, we investigate the usage of Behaviour-Interaction-Priority version 2 (BIP2), a component-based modelling framework, for specifying feature-oriented systems. We evaluate BIP2 in the context of the Feature Interaction Problem and quantify the amount of work needed to add features to an existing system (i.e., in terms of rework to existing features, and work to identify and specify interactions). We present the results of a case study on a telephony system with five optional features where we found that the amount of work depends heavily on how features are interconnected. We identify three different design methodologies for interconnecting features, and propose one that reduces the amount of work and rework needed to add new features to an existing system.

I. INTRODUCTION

In software engineering, an increasingly popular strategy to decompose a complex system into smaller subproblems is to perform feature-based decomposition, which is a type of functional decomposition of the system. A feature is a unit of functionality that can be developed and evolved independently. However, the composition of separately designed features to produce a final product often leads to unexpected or undesirable behaviours. A feature interaction (FI) occurs whenever the presence of one feature alters the behaviour of another. For example, a user may subscribe to a telephony feature that automatically forwards her calls to another number; she may also subscribe to a second feature that screens calls against a list of blocked numbers. If each feature is specified and developed without knowledge or consideration of the other feature, the outcome is not clear when both are activated in the same scenario. A call could be screened before it is forwarded, or it could instead be screened against the list of blocked numbers at the forwarding destination.

To be safe, a developer must consider how a new feature might interact with existing features. To be thorough, all combinations of existing features need to be considered. As the number of features grows, the number of feature combinations that must be analyzed for possible interactions grows exponentially — until the work of integrating a new feature is dominated by the analysis and resolution of feature interactions. In systems with high variability, the Feature Interaction Problem, the task of analyzing every possible combination of composed features and resolving any discovered feature interactions, becomes intractable with existing methods [1].

Many techniques and tools have been developed to minimize the work of the developer in discovering and resolving feature interactions [2]. One such strategy is the use of specialized modelling languages for the design and verification of composed systems. Behaviour-Interaction-Priority version 2 (BIP2) [3], [4] is a framework for the design of component-based systems. BIP2 allows the designer to decompose a complex system into a collection of interconnected components.

Given that the BIP2 formalism is designed to support component-based modularity, and given that BIP2 has explicit language constructs for specifying how feature combinations ought to synchronize and how conflicts and nondeterminism ought to be resolved, we investigated how to use BIP2 to address the Feature Interaction Problem. We performed a case study in which we used BIP2 to model a telephony system with five features. We aimed to answer the following questions in our investigation: (1) Is it possible to model features independently and integrate them into the system without changing existing features? (2) How much work (and rework) is required to integrate a new feature into an existing system model? (3) How much work is required to specify interactions among features, and what is the overall complexity of the resulting system model?

Answers to these questions depend heavily on the design methodology used to define component interfaces and to interconnect components. We identify three distinct design methodologies for composing features, and we evaluate the amount of developer work that is needed to integrate new features and resolve feature interactions in each approach. An interesting side effect of this work is that we have shown how BIP2 – whose strength is in modelling components that are designed to know about each other and to work together – can be used to model components that do not know about each other and to compose them so that they can work together.

II. OVERVIEW OF BIP

Behaviour-Interaction-Priority (BIP) is a component-based language for modelling complex systems [3]. In BIP, the behaviour of a system is modelled as a collection of individual components, each of which is responsible for a subset of the system’s behaviour. As the name suggests, BIP provides three layers of specification to the model: the Behaviour of system components, the Interactions between these components, and the Priorities between multiple possible execution paths. In this paper, we use the second iteration of this framework, BIP2 [4], and will refer to this version from this point forward.

1Given how the term interaction is overloaded, we use the acronym FI to refer to a traditional feature interaction (any difference in feature behaviour, intended or not, due to the presence of other features). We reserve the qualified term interaction to refer to a BIP2 interaction (an explicitly specified communication and synchronization among connected components).
A. The Behaviour Layer

Each component in a BIP2 model defines a subset of a system’s overall functionality. In this paper, our system consists of a base component that provides basic call-processing functionality (i.e., on-demand voice connections between two users), a set of optional feature components that extend or override this functionality, and a component that represents the system’s environment (i.e., telephone users).

The most basic BIP2 component is an atom. The internal operation of each atom is modelled as a Petri net. An atom’s current state is represented by the set of currently occupied places and the values of the atom’s variables. Transitions between places in the net update the atom’s variables and the set of occupied places. A transition from a set of previously occupied places to a set of newly occupied places may be optionally labelled with a guard, an update function, and a port. A guard is a predicate over the atom’s variables, and a transition is enabled and executed only if the system state satisfies the guard. After transitioning, the variables are updated as dictated by the update function. Ports trigger transitions in synchronization with other components, and are used in the specification of the interaction layer. Ports restrict transitions similar to guards; a transition labelled by a port relies on an interaction with another component to execute.

B. The Interaction Layer

Components interface with each other through ports that are linked together by connectors. A connector links at most one port from each of the two or more components it connects: the effect is to synchronize the transitions in each of the connected components that are labelled with the linked ports. The ports in a connector may be either triggering ports (i.e., senders) or synchronizing ports (i.e., receivers). When a transition labelled with a sender (denoted by a primed port name, e.g., busy’) is enabled, a synchronized execution step that involves a subset of the enabled receiving transitions in the connected components will execute. The subset of transitions that execute is determined by the guards and the priority ordering of the connector’s interactions.

Each interaction in a connector consists of a triggering port(s) and some subset of the connector’s synchronizing ports. Interactions may be labelled with guard and transfer functions in the same manner as component transitions, restricting which of the components will participate in the synchronized step. The variables in these functions are the data variables exported by the components’ ports. Upon execution, the interaction’s transfer function updates the variables in participant atoms, allowing components to exchange information.

For example, in a telephony model, the connectors between the basic-call service components of multiple users define the ways in which the services may interact throughout the process of a call. Likewise, the connectors between a user component and its basic-call component define how a user interacts with her own call service.

C. Priorities

To combat nondeterminism and enforce scheduling policies, BIP2 provides priorities as a means to choose between multiple enabled execution paths. Nondeterminism arises when there are multiple simultaneously enabled interactions, each leading to a different overall system state. Normally, if there is more than one connector with an enabled interaction, there are no guarantees about which interaction will execute. We can control the outcome by specifying priorities in one of two ways: (1) at the component level by specifying that port $p_1$ has a higher priority than port $p_2$ with $p_1 > p_2$, or (2) at the interaction level by specifying that interactions in the connector $C_1$ have priority over interactions in the connector $C_2$ with the priority $C_1 : * > C_2 : *$.

The simplest way to resolve all nondeterminism is to define a complete ordering on the transitions that lead from each state. Our basic-call service atom requires a total of 26 priorities to resolve conflicts from simultaneously enabled interactions and avoid inconsistent states. Priorities play a large role in the resolution of feature interactions.

III. Telephony Case Study

We conducted a case study on a telephony system to assess the extent to which BIP2 combats the Feature Interaction Problem. In this section, we outline the basic structure of our telephony system, the features involved, and the criteria we used to evaluate the design methodologies we developed.

A feature-oriented BIP2 telephony model consists of three parts: (1) a basic-call service (modelled as an atomic component), (2) a set of optional features to which a user may subscribe that extend or modify the functionality of the basic-call service (each of which is modelled as an atomic component), and (3) the user (modelled as an atomic component).

Each user’s basic-call service (BCS) allows that user to place and receive calls. The places in the BCS component, together with its variables, reflect the possible states of an outgoing or incoming call. The ports of the component reflect the ways in which users and features may interact with or extend the functionality of the BCS (e.g., taking the phone off the hook, or dialing a number), and the ways in which the BCS of one user interacts with the BCSs of other users (e.g., establishing a connection). Our case study includes five optional features, taken from the specifications for the Feature Interaction Contest [5]:

- **Call Forwarding (CF):** The subscriber may specify a forwarding number. All calls to the subscriber will then be forwarded to this number.
- **Call Forwarding on Busy (CFB):** If the subscriber receives a call when she is involved in another call, the feature will redirect the new call to a predetermined forwarding number.
- **Call Waiting (CW):** If the subscriber receives a call when she is involved in another call, she may choose to put the original call on hold, answer the new call, and then toggle between the two calls.
- **Terminating Call Screening (TCS):** This feature allows its subscriber to specify a list of blocked numbers. Any call
originating from a number on this list will be terminated automatically.

**Three-Way Calling (TWC):** This feature allows a subscriber to add a third user to an existing call. Once three-way communication has been established, any user may choose to leave, resulting in a traditional two-way call configuration.

The BIP2 framework claims to support component-based modelling with an emphasis on inter-component interactions. The primary goal of our case study was to assess these claims in the context of feature-oriented modelling and feature interactions (FIs). We evaluated BIP2’s suitability for modelling feature-oriented systems on three main points:

1) **Composed model complexity:** The overall complexity of a complete model of the telephony system (i.e., the BCS together with the user model and optional features for each user).

2) **New feature integration:** The amount of work that a developer must perform to add a new feature to an existing system. We look at the difficulty of design decisions when composing new features in terms of limitations on the number or type of ports in existing components, transitions within the BCS component, and the types of existing connectors. We strive to adhere to the principles of feature-oriented development. That is, the addition of a new feature to the system should not require the modification of the BCS or existing features.

3) **FI Resolution:** The difficulty of detecting and resolving FIs in terms of how the modeller discovers conflicting features and the number of changes they must make in the model to resolve these FIs.

Our secondary goal was to identify design methodologies or patterns for modelling and connecting BIP2 components in feature-oriented systems. In the next section we present three different feature-oriented modelling strategies and evaluate each of them based on the criteria above. For a more complete description of our modelling strategies complete with BIP2 models and code, see our extended technical report [6].

**IV. DESIGN METHODOLOGIES**

Each of our design methodologies approaches the problem of feature composition and integration with the base system in a different way, resulting in different interactions, different degrees of model complexity, and different types of decisions the modeller must make during composition. We give a summary of our evaluations in Table I.

**A. Reuse Approach**

In the reuse approach, new features are integrated into the base component by reusing existing components and expanding the connectors between basic-call services and users to include the new feature component, and replacing the default interactions with new ones that slightly alter the progression of a call. The inspiration for this approach stems from the idea that a feature overrides existing functionalities provided by the base service. Our case study features can naturally be described in terms of the BCS functionalities they override: CFB, CW, and TWC override the progression of a call when the subscriber is busy, while CF and TCS override the progression of an incoming call.

A call progresses through interactions with other basic-call services and users. To integrate a new feature in the reuse approach, we first identify the interactions in the existing components that it overrides. We then expand the connector(s) that contain these interactions to include ports in the new feature’s component. Interactions that involve the new feature’s synchronizing or triggering ports are then given higher priority than the pre-existing interactions.

We show the integration of CW to an existing BIP2 model in Figure 1. If User A is in a call, the (red) interaction normally terminates subsequent incoming calls by synchronizing the busy’ port of User A’s BCS with the isBusy port of the caller’s BCS, causing the caller to transition to its WAIT FOR OHOOK place. If User A subscribes to CW, this interaction is replaced with a new interaction (blue) that instead allows the caller to proceed to the INCALL state. The CW feature keeps track of which of the subscriber’s calls is currently on hold. The new interaction is given higher priority, thereby replacing the old functionality.

**B. Rewire Approach**

While the reuse approach allows for the independent development of features and resists changes to the BCS components, the design and integration of a feature is limited by the ports and transitions of existing components. Furthermore, a system with many features that override the same functionalities may result in very large connectors that contain ports from many different components. These connectors are more difficult to specify and define priorities for, as all combinations of enabled ports must be considered. We designed the rewire approach to give the modeller more freedom to modify existing features and code, see our extended technical report [6].
components with the expectations of easier design decisions and simple components and interaction specifications.

In the rewire approach, new features may entail new functionality (i.e., new ports and transitions) in the pre-existing model of the BCS. When integrating a feature, we first decide the changes the feature makes to the progression of states inside the BCS, and add new transitions and label them with new ports that will be connected to the new feature component. Finally, we design the feature component, and specify the interactions of a new connector that synchronizes transitions in the modified BCS components and the feature component.

In Figure 2, we give an example of the changes made to a BCS component when integrating CF and TCS, both of which modify the progression of an incoming call. In TCS, a new call interacts with the TCS feature component through ports that first check and then allow or block the call. New transitions and new ports (shown in purple) are involved in new interactions with the connected TCS component.

The rewire approach results in feature-specific connectors that are small and similar in behaviour. Fortunately, BIP2 allows modellers to specify connector types to ease the specification of many, similar connectors. This further reduces the work of the modeller and the complexity of the overall model in the rewire approach. Unfortunately, the advantages of the rewire approach come at the cost of violating the principles of feature-oriented development: existing components must be extended with new transitions that react to events on new ports.

C. Pipe-and-Filter Approach

The reuse and rewire approaches exemplify the challenge of feature-oriented modelling in BIP. There is a trade-off between modelling freedom versus modularity; by refusing to change existing components, we restrict the ways in which other components can interact with them. To bridge the gap between these two strategies, we adapted an approach that standardizes how components interact with each other.

We took inspiration for our third approach from the Distributed Feature Composition (DFC) architecture developed by Zave and Jackson [7] for the development and composition of telephony features. In DFC, each user’s features are connected sequentially in a pipeline, and communications from one user to another propagate through a sequence of features as a call is placed from one BCS to another. Thus, the execution of features is serialized, with each feature triggering the next feature in the pipeline. As a result, DFC provides a default resolution of FIs by imposing a priority ordering on the execution of features, determined by the feature’s positions in the sequence (e.g., the last feature in the pipeline provides a final response to a user request).

In our pipe-and-filter approach, we standardize the triggering and synchronizing ports on each feature, making it much easier to interconnect features without knowledge of their internal structure. Synchronized transitions within components are triggered not just by communications on the ports of connectors, but by the specific data conveyed in the communications. Specifically, we designed a new BCS that standardizes the messages that are sent among components. Messages fall into one of three main types: messages that establish a call, busy messages that indicate the other service is currently unavailable, and disconnect messages that indicate one of the participants wishes to terminate a call. Every component has two ports: a synchronizing port in for receiving incoming messages, and a triggering port out for sending outgoing messages. Every interaction between an out and in port passes the following data: (1) The enumerated message type (CONN, BUSY, or TERM), (2) the id of the component that sent the message, and (3) the id of the component that is the designated recipient of the message.

In Figure 3, we show the composition of two BCS components with a TCS feature component. A user’s features are arranged and connected in a sequence between her BCS and the feature sequences of other users. Messages “flow” through the pipeline one component at a time. Each component synchronizes with the previous component in the chain; decides whether to react to the received data by modifying the message; and then propagates the message further, either by passing it to the next feature or back to the previous feature.

The standardization of port types and interactions, along features’ compliance to the rule that all components must propagate messages either forward or backward through the pipeline, allows features and BCS components to be oblivious of the behaviour and existence of other components, while still reacting predictably to received communications. Features can be designed independently and in parallel. This provides a greater degree of modularity than the rewire approach, which requires modifications to existing components, as well as the reuse approach, which requires knowledge of existing components. Additionally, every feature has the same ports and is linked to other components with the same connectors, further reducing the work of the modeller.
D. Discussion

We performed a case study to evaluate each strategy on three main points: the complexity of the overall model (in terms of the number of feature places and transitions, as well as modifications to the BCS and the number and complexity of connectors used to compose the overall model), the work of integrating a new feature into the existing model (in terms of additional feature components, connectors, priorities, and design decisions that require knowledge of existing components), and the difficulty of detecting and resolving feature interactions (in terms of analyzing existing components and the rework required to remove undesired behaviour). We summarized our quantitative data from the case study in Table I.

We found that each approach exhibits complexity in a different aspect of the modelling process, as shown in Table I. The reuse approach has more complex connectors and interaction specifications, whereas the rewire approach adds model complexity in the form of monolithic implementations of features in the BCS component, which violates the principles of feature-oriented design and increases the chance of introducing errors in BCS behaviour. The pipe-and-filter approach introduces complexity in yet another area, requiring more complex feature components to formulate specialized behaviour in response to standardized messages. Feature components in the pipe-and-filter approach require more data variables, and transitions require guard and update functions that react to and modify the component and message data variables.

The integration and resolution of new features require varying amounts of knowledge, work, and design decisions in each of the three approaches. The reuse approach requires the modeller to design new features within the constraints of existing ports and transitions in the BCS. In contrast, the rewire approach affords the modeller more freedom, yet complicates the BCS model and violates the principles of feature-oriented design. In fact, both of our first two approaches require significant knowledge of, and possible modifications to, existing components. In feature-oriented systems with a continuously evolving set of features, it is advantageous for a feature developer to not know about the other features in the model. It is this obliviousness and separation of concerns that allows features to be developed in isolation and by third parties, and to be more easily integrated into an existing system without requiring significant rework of existing features or their connectors. The pipe-and-filter strategy is the most effective in supporting feature obliviousness. Not only does every feature have the same interface, but the connector types and their interactions are standardized. What is left to the modeller is to determine the order of connected features in the pipeline, and to instantiate the connectors to realize this pipeline. As a result, the composition of features and resolution of FIs was almost trivial.

We have shown that in BIP2, where specifications of ports, connectors, and interactions require some knowledge of the internal workings and ports of other components, feature-oriented modelling is possible with the pipe-and-filter approach. In this approach, individual features may remain agnostic to other features, only requiring knowledge of the base component during their development and composition.

V. Related Work

Since the framing of the Feature Interaction Problem in 1989 [1], there have been myriad attempts to minimize the effort of the developer in composing systems that are prone to a large number of FIs [2], [8]. Off-line techniques aid the developer during the design and development of the system.

- Techniques for detecting FIs reduce the effort of the developer in discovering problematic compositions of features and pinpointing the sources of undesired behaviour that need to be resolved [9], [10], [11], [12].
- Filtering approaches limit the variability of a system by removing problematic or unlikely combinations of
features from analysis, thereby reducing the number of FIs a developer needs to consider to those in a small set of feasible products [13], [14], [15].

On-line techniques for coordinating feature execution resolve FIs as they occur at runtime. Hay and Atlee proposed a specification and composition model that uses feature priority to automatically resolve FIs during composition [16]. Distributed Feature Composition (DFC) developed by Zave and Jackson [7] connects features in a pipe-and-filter architectures, avoiding FIs architecturally by serializing the features’ executions. In contrast, BIP provides modellers with the flexibility to specify how features are connected and prioritized.

There have been previous case studies to evaluate the modelling capabilities of BIP. Basu et al. performed a case study on wireless sensor networks [17] to assess the suitability of BIP2 in modelling distributed systems with heterogeneous components. Bourgos et al. conducted a case study on the modelling of a MJPEG decoder [18] to test the use of BIP2 in analyzing the performance of embedded applications on different hardware platforms. While these case studies provide evidence for the flexibility of BIP and its applicability to a wide variety of hardware and software systems, to our knowledge, there are no existing studies that analyze the use of BIP in the context of the Feature Interaction Problem. We provide both an analysis of its use to compose and analyze features and a comparison of design strategies for specifying feature-oriented systems in BIP.

VI. CONCLUSION

In summary, we investigated the effectiveness of BIP2 for modelling feature-rich systems, with particular attention to the amount of work needed to compose features, the amount of re-work needed to evolve a model to integrate new features, and the degree of complexity of the resulting model. Each of the three strategies that we studied has its advantages and its weaknesses. Ultimately, when considering which strategies help to address the Feature Interaction Problem, the pipe-and-filter approach is the more effective design methodology: (1) it supports and preserves feature modularity, even when new features are added to the system, and (2) the amount of work and re-work needed to add a new feature is substantially less than in the other two strategies.

REFERENCES


Towards a Meta-Model for Quality-aware Self-Adaptive Systems Design

Esma Maatougui, Chafia Bouanaka, Nadia Zeghib
LIRE Laboratory
University of Constantine 2-Abdelhamid Mehri
Constantine, Algeria.
esma.maatougui@univ-constantine2.dz, chafia.bouanaka@univ-constantine2.dz, nadia.zeghib@univ-constantine2.dz

Abstract—Self-adaptation is a promising technique to manage software systems maintainability and evolution. A self-adaptive system is able to adapt its structure and behavior autonomously at run-time in response to changes in the context in which it is actually running to achieve particular quality goals. However, designing and verifying quality-aware self-adaptive systems remains a challenging task. In this paper, we propose a formal approach that combines the advantages of both component-based modeling (e.g., reduces model complexity), MDE (e.g., facilitates the development process) and Maude (a formal language) to define a development process for quality-aware self-adaptive software. We particularly focus on the specification of quality-aware adaptation strategies required to ensure continuous satisfaction of non-functional requirements (Quality of Service).

Index Terms—Self-adaptive systems; QoS; Component-Based Software Engineering; Model-Driven Engineering; Maude.

I. INTRODUCTION

Nowadays, users extensively rely on software systems quality, especially in the presence of parametric and variable execution contexts. However, ensuring the required qualities of software systems that might operate in dynamic environments, poses severe engineering challenges, since they must become more versatile, flexible, resilient, dependable, energy-efficient, recoverable, customizable, configurable, and self-optimizing by adapting themselves to changes that may occur in their operational contexts, environments and system requirements. Self-adaptation [1] is generally considered as a promising solution to manage the complexity of such software systems since it enables the system to adapt itself to internal dynamics and changing conditions in the runtime environment to achieve particular quality goals automatically.

A key characteristic of self-adaptive systems engineering is to provide guarantees about the required runtime quality properties. Nevertheless, the central role of QoS requirements has to be considered at the early stages of design. Hence, the emergence of the software system architecture provides the right level of abstraction, sets the basis to achieve both functional and non-functional requirements, and needs to be supported by methodologies and tools to capture these two dimensions of the product at the same time which generally can deal with the challenges of self-adaptation [3]. The component-based approach can provide an appropriate level of abstraction to describe dynamic changes in a system structure and increase the reusability and portability of software pieces. However, a key issue to be faced concerns the assessment of self-adaptive systems effectiveness, in terms of their ability to meet the required QoS under different context conditions. In particular, this assessment should take into account the cost of the adaptation process itself. Since, adapting a system can require time and system resources to be carried out, and this cost could even outweigh the potential benefit [3]. In addition to component-based software engineering (CBSE) [4], Model-driven engineering (MDE) [5] is an emerging approach to address these and other challenges.

MDE advocates the use of models, not only for capturing high-level design ideas and documenting the final product, but as key artefacts throughout the development process. The goal is to reduce the development time and efforts, and to increase product quality by raising the level of abstraction and automating some time consuming and error prone activities, e.g., by generating code directly from detailed models instead of implementing it manually [6].

One major advantage of MDE is the opportunity to automatically transform design models into analytical ones, thus enabling formal verification of system properties; including non-functional ones. A largely adopted approach is the combination of MDE and formal methods to ensure and guarantee functional correctness of the adaptation logic. This provides a rigorous means for modeling, specifying and reasoning about self-adaptive systems’ behavior, both at design time and at runtime.

A variety of research work has been realized and significant efforts invested to propose models for QoS-aware self-adaptive systems. However, existing techniques for non-functional properties analysis rely on very specific quality-related formalisms such as Petri Nets (PNs), or Markovian models, but software systems are rarely represented in these terms [3]. Besides, most of these approaches do not take into account the separation of concerns between user requirements in terms of QoS contract and system QoS parameters. Moreover, designers, who usually lack sufficient experience in requirements engineering, prefer design-oriented formalisms such as UML [7] which reflects more the modeling intent.

In this paper, we present a component-based contractual approach to define a model for designing, specifying and verifying self-adaptive systems with respect to QoS contracts. To address this problem, we define a model for QoS contracts as a natural and effective way for user requirements.
The remainder of the paper is organized as follows. Section 2 discusses some models for self-adaptive systems that are relevant to our work. Section 3 is dedicated to the presentation of our model and the generation of the corresponding formal specification. Section 4 illustrates our proposal via a case study to validate our model. Finally, Section 5 rounds up the paper.

II. RELATED WORK

A variety of models for Self-adaptive systems have been proposed and various modeling methodologies have been adopted, including MDE [7, 8], requirements engineering [9] and component-based development [12].

Vogel and Giese [8] propose a MDE-based model for Self-Adaptive Software with EUREMA approach that realizes self-adaptation using the so-called executable runtime mega-models. In [7], a UML-based modelling language called Adapt Case Modeling Language (ACML) is presented. The language allows a separate and explicit specification of self-adaptivity concerns using the concept of the MAPE-K loop. Based on formal semantics, they apply quality assurance techniques to the modeled self-adaptive system.

Brown and Cheng [9] adopt a Goal-Oriented Requirements Engineering to present the Awareness-Requirement and propose a way to elicit and formalize such requirements using the OCL language. A methodology for generating feedback from such requirements, as well as fragments of a prototype implementation founded on an existing requirements monitoring framework is proposed. Elkodary et al., present an approach, named FUSION [10], which uses feature diagrams as a system model where self-adaptation is realized by switching between different system configurations. The self-adaptation in FUSION is goal-driven, i.e., relying on predefined functional or non-functional goals. Each goal consists of a metric and a utility. While the metric is a measurable entity as response time, a utility is a feature which has influence on the metric, and is triggered when FUSION detects that a goal is violated. The violation of a goal is detected via defined monitoring functions.

DYNAMICO [11] is a reference model for engineering adaptive software aligned with the vision of self-adaptive systems, where dynamic adaptation is necessary to ensure the continuous satisfaction of their functional requirements while preserving the predefined conditions on Quality of Service levels. These QoS levels are usually represented in the form of Service Level Agreements (SLAs), and their enforcement mechanisms are based on contracts and policies. Castaneda Bueno designs a component-based reference architecture [12] with distribution and extensible capabilities for self-adaptive systems according to the reference model DYNAMICO.

In the present work, we propose a component-based contractual approach for quality-aware self-adaptive software systems specification that supports system and QoS contracts modeling together with the corresponding adaptation logic. The proposed approach defines QoS constraints in an independent way from system QoS parameters. This separation of concerns reduces system modeling complexity and increases model reusability and maintainability. Our model for quality-aware self-adaptive systems provides a clear satisfaction of QoS contacts by applying adaptation strategies in case of violation of QoS constraints.

III. A COMPONENT-BASED CONTRACTUAL APPROACH FOR SELF-ADAPTIVE SYSTEMS

We adopt a component-based contractual approach to define a model for designing and formally specifying self-adaptive systems with respect to QoS contracts CBSE can help in the development of self-adaptive software in two ways. First, it is easier to design and implement adaptable software relying on component models. Second, the adaptation engine needs to be modular and reusable. Additionally, CBSE can also be adopted in the development phase of the self-adaptive system. However, the Component-oriented paradigm still requires comprehensive and sound QoS contract-aware self-adaptation theories, models and mechanisms further trustworthy, extensible and reusable in order to realize its contract. Moreover, in the CBSE vision, contracts play a fundamental role, as they must capture the functional and extra-functional user requirements.

We define a QoS-aware component-based model for self-adaptive systems where context and functional entities are viewed as components that interact via adaptation strategies, and designed in an entirely independent manner and only relationships between them are specified, thereby simplifying the adaptation mechanisms. To achieve this goal, we model an adaptation strategy as a pair of elements: an action associated with the notification of events that violate their contracted QoS constraints. The adaptation strategy adapts system functionalities according to context changes in terms of variations on system structure and/or behavior.

The model is designed with a focus on the separation of concerns between the specification of QoS parameters; defining user quality requirements, and software components quality parameters (see Figure 1). The first ones are specified in the QoS contract while the second ones are directly defined of the component specification.

![Figure 1. An Overview of the proposed model.](image)

QoS contracts comprise a number of quality of service constraints that might be satisfied and preserved by a managed system. These QoS constraints are specified for each of the different context conditions that the managed system is faced with while it is running. Thus, the continuous satisfaction of a
QoS contract (i.e., its preservation) implies satisfying each of the QoS constraints that the user expects, under each of the corresponding varying conditions of execution contexts. At runtime, once these conditions actually occur in the execution context of the managed application, the respective QoS constraints must be monitored, and their fulfillment enforced.

To be able to automatically ensure QoS contracts, a component-based self-adaptive system requires (i) to maintain a structural representation of itself (ii) to have a representation of the contracted QoS constraints under the different context conditions; (iii) to be self-monitoring, that is, to identify and notify events on the QoS constraints violations; and (iv) to apply the dynamic reconfiguration in response to events notifying imminent violation of QoS constraints, as specified in the QoS contracts.

Based on the previous considerations, we build our component-based QoS-aware model for self-adaptive systems. We first present our meta-model-based definitions for component-based self-adaptive software structure and QoS contracts respectively. Then, we define transformation rules to be applied to generate automatically a Maude formal specification of models instantiating the already defined meta-model.

3.1. Model-based self-adaptive systems design

Our model exploits the MDE techniques to provide a solution for self-adaptation via meta-models which describe concepts that can be used for constructing models that conform to its definition, and describes in an abstract way, the possible structure of the underlying models. The meta-model of Figure 2, specifies the various concepts that intervene to define the structure of quality-aware self-adaptive systems together with their pertinent relationships. It is structured in four parts:

A. The first part contains four meta-classes representing a quality of service contract. A QosContract is defined by its name and a set of QoS properties. A QoS property denotes a specific non-functional characteristic of the considered system such as its performance, reliability, and cost. A QoSproperty is defined by a name and a weight reflecting the relative importance of the QoSproperty with regards to the user preferences. To facilitate the specification of user preferences, three weight values are predefined in the Weight Enumeration (high, low, medium). Each QoSProperty needs one or more metrics to be quantitatively measured. A QosMetric, defined by its idMetric, represents a non-functional property which belongs to a domain of values as response time. Finally, we associate a QosConstraint to the entire or a subset of QoS properties in different conditions of context. In general, a QosConstraint consists of a relational operator (e.g., <, >, =) and a value representing a threshold.

B. The second part contains two meta-classes representing context sensors used to model context sources and values. The ContextSensor meta-class is defined by its SensorID and sensor type. Three types of sensors are identified in [13]: Physical, Virtual and Logical sensors. Sensor types are represented via the SensorTypes Enumeration. The Context meta-class defines anything that interacts and affects the target or functional system. The Context is defined by its ContextID and the corresponding possible values.

C. The third part contains the AdaptationStrategy meta-class, which represents scenarios of adaptation that will be applied in the case of violation of the QoS Constraints. These scenarios are defined by a set of adaptation rules that can be of the following types: (i) add a component to the actual system configuration, (ii) remove a component, and (iii) replace one component by another.

D. The last part of the meta-model contains necessary concepts to define the functional system configuration, viewed as a set of components which require or provide services to each other through specific interfaces. These components are represented by the Component meta-class and defined by a name specified in the CName attribute. A component comprises a set of Quality attributes (quality attributes of the running service), and a set of provided interfaces (ProvidedInterface) and Required ones (RequiredInterface). Each interface exposes a set of services that are required or provided by the component. Connections in our model are dynamic and only established whenever one component is providing the service and the other one is requesting it.

3.2. Model Transformation for Generating Maude specifications

Albeit, MDE tries to facilitate software development and simplify the design process by specifying meta-models focusing on the structural and static semantics of software systems, it lacks necessary concepts to define the semantics or behavior of software systems and thus verification mechanisms that are among the major issues in specifying self-adaptive systems. A reasonable and desirable formal method to be adopted for this scope should be powerful enough to capture the principal models of computation and specification methods, and endowed with a meta-model-based definition conforming to the underlying meta-modeling framework. Additionally, the formal approach should allow working at different levels of abstraction, and be executable, in order to validate the meta-model semantics. Rewriting logic [14] via its implementation language Maude [15] is an adequate candidate for the definition of the semantics basis of our meta-model for many reasons. First, the versatility of rewrite theories can offer the appropriate level of abstraction for addressing the specification, modelling and analysis of self-adaptive systems and their environment within one single coherent framework. Second, since Maude is a rule-based language, the adaptation logic can be naturally expressed as a subset of the available rules, and the meta-programming capability of Maude can be exploited to enforce the execution of a given adaptation rule to maintain QoS parameters via Maude strategies. Third, the formal analysis toolset of Maude can support simulations and analysis over the self-adaptive system.
The bridge between MDE and formal methods is established via model transformation techniques, realized via a set of transformation rules. A model transformation consists in general of a computation that applies repeatedly a set of transformation rules to a model, where the model represents the structure of a sentence in a given formal language, defined by a meta-model. EMF (Eclipse Modeling Framework) [16] and specially Acceleo [17] are used in our case as a modeling framework and code generator implementation of the OMG’s Model-to-text specification for building tools and applications based on models defined in the Ecore meta-model. This tool provides the capability to define advanced code generators for transforming models to a target code by defining transformation templates.

Table 1 illustrates some results of transformation rules defined between the self-adaptive meta-model and the formal semantics. The meta-model and the imposed constraints provide the capability to achieve a formal specification generation through template models. Our goal is to transform EClass, EAttribute, EReference and EOperation of the self-adaptive model to Maude constructs to facilitate self-adaptive systems specification.

Since Maude offers two possible representations, the algebraic and the object-oriented ones, we have adopted an object-oriented representation in order to reflect the hierarchical structure of self-adaptive systems and avoid the flat structure while adopting algebraic terms. In addition, all structural concepts are transformed to Maude classes while behavioral concepts as Adaptation Rules and Adaptation Strategies are transformed to rewriting rules and Maude Strategies respectively. The first mapping of Table 1 concerns structural concepts that can be defined as an Acceleo template as follows:

```
[template public generateElement(Package : EPackage)
 [comment @main/]
 [file (Package.name.concat('.maude'), false, 'UTF-8')]
 (omod [Package.name.toUpperFirst()] is
 for (c : EClass | Package.eAllContents(EClass))
 separator('\n')
 [if c.name.equalsIgnoreCase('AdaptationStrategy')=
 false]
 [if(c.name.equalsIgnoreCase('AdaptationRule')=
 false)]
 class [c.name.toString()] | [for (a : EAttribute
 | c.eAttributes) separator(' , ')] [a.name] :
 [if (a.eAttributeType.name='EString')
 a.eAttributeType.name] [if] [/if]
 [if (c.eReferences<>null)]
 c.eReferences->first().name] [if] : OidList.
 [/if]
 [if]
 [/for]
 endom)
[/template]
```

The template for structural concepts generates a Maude file, using a tag [file] to specify the output file, that contains the various classes and their attributes as specified in Table 1. It begins by testing if the considered element is not a behavioral concept, i.e., neither an adaptation rule nor an adaptation strategy. Such verification is realized via the conditional statement [if]. Then, it generates a class from each EClass of the meta-model via the [for] bloc, together with the corresponding attributes.
TABLE 1. Transformation results.

<table>
<thead>
<tr>
<th>Eclass</th>
<th>Maude specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>QosContract</td>
<td>class QosContract {name: String, QosProperties: OidListe}</td>
</tr>
<tr>
<td>QosProperty</td>
<td>class QosProperty {name: QosPropertyName, Weight: Weight, QosMetrics: OidListe}</td>
</tr>
<tr>
<td>QosMetric</td>
<td>class QosMetric {idMetric: String, QosConstraints: OidListe}</td>
</tr>
<tr>
<td>QosConstraint</td>
<td>class QosConstraint {value: Float, operator: String, contextValue: Oid}</td>
</tr>
<tr>
<td>FonctionnelSys Component</td>
<td>class FonctionnelSystem {Components: OidListe}</td>
</tr>
<tr>
<td>ProvidedInterface</td>
<td>class Component {Cname: String, QualityAttribute: Oid, ProvidedInterfaces: OidListe, RequiredInterfaces: OidListe}</td>
</tr>
<tr>
<td>RequiredInterface</td>
<td>class Service {Servicename: String, QualityAttribute: Oid, isActive: Bool, Parameters: OidListe}</td>
</tr>
<tr>
<td>Service</td>
<td>class QualityAttribute {name: String, value: Float}</td>
</tr>
</tbody>
</table>

Behavioral concepts

AdaptationRule

crl [ReplaceComponent] :< F : FonctionnelSystem | Components: C Cl >
< C : Component | Cname | name, QualityAttribute: Q1, RequiredInterfaces: I PIL, RequiredInterfaces: RIL >
< C' : Component | Cname | name2, QualityAttribute: Q2, ProvidedInterfaces: I PIL, ProvidedInterfaces: L >
< Q1 : QualityAttribute | name | QN, value: V1 >
< Q2 : QualityAttribute | name | QN, value: V2 >
=>
< F : FonctionnelSystem | Components: (del(C, (add(C’, CL ))) ) >
< Q2 : QualityAttribute | >
< C' : Component | Cname | name2, QualityAttribute: Q2, ProvidedInterfaces: I L, RequiredInterfaces: L >
if V2 < V1

AdaptationStrategy

{fmod SelfAdapt-STR is pr REN-SEQ .
op SelfAdaptStrat : -> List{Tuple{Qid, Substitution}} [memo].
eq SelfAdaptStrat = (‘ReplaceComponent, ‘F:Oid <- ’F.Qid ; ‘C:Oid <- ’FireManComp.Qid ;
’C’.Qid <- ’FireEngComp.Qid’).

IV. MOTIVATING ADAPTATION SCENARIO

The scenario of a firefighting system [18, 19] is used as an example. Fire fighters often work in dangerous and dynamic environments. Moreover, a fire accident is one of the most frequent incident types. The early detection and timely preventive measures are effective methods for limiting fire damage and reducing casualties. In this example, the firefighting system is a component-based software system designed to detect fire signals and make effective fire-management strategies. When fire danger occurs, these components dynamically restructure into a firefighting plan by choosing appropriate firefighting resources from the component library. These well-structured components then drive the corresponding fire-extinguishing installations to perform the firefighting plan.

The Firefighting System automatically takes effective measures to prevent the fire disaster (Goal). This goal can be further decomposed into: (G1) detect fire signals in the early stage and (G2) assemble a set of fire-fighting devices in response to a real-time fire situation. To achieve these self-adaptation objectives, we should identify detectable contexts reflecting the software running state or physical environment, and then identify adaptive actions that can be performed at runtime to change the system behavior. In this example, the detectable fire signals (contexts) are various, such as CO, CO2, along with high temperature, and strong flame. Therefore, the context to be chosen depends on the occurring place and the fire disaster type.

Self-adaptive Firefighting System is used to monitor indoor fire disasters. It is composed of two essential parts, see Figure 3: context layer and functional one. We identify Temperature, Smoke Concentration, CO Concentration and Infrared Wavelengths as different contexts. The corresponding Maude specification of the available contexts is given by the following fragment of code:

< 'CTXS1' : ContextSensor | SensorID : 'FireMonitor_TEM'. Type : Temperature, Context : 'CTX1' >
< 'Temperature': Context | ContextID : 'Temperature', ContextValue : "65" >
< 'CTXS2' : ContextSensor | SensorID : 'FireMonitor_CO'. Type : COConcentration, Context : 'CTX1' >
< 'CO-Con': Context | ContextID : 'CO-Con', ContextValue : "70%" >

16
We also identify three types of components: Fireman, Fire Engine and Extinguisher. In the example, fire-prevention measures are made by dynamically restructuring the firefighting components. The corresponding Maude specification of these components is given by the following fragment of code:

The “FireManComp” component has “Q1” as a quality attribute which represents the response time of 50 sec and a Provided Interface “FM_Interface” that proposes a unique running service “StartCompFM”.

In the firefighting system, we are concerned with the analysis of the performance quality parameters in terms of the response. For this reason, we identify the Firefighting_Contract which comprises the Performance as a QoSProperty and ResponseTime, see Figure 3, as a metric that is used to evaluate the performance. We propose two QoSConstraint in this example: The response time in the Temperature context must not exceed 30 sec. But, in the context of CO-Concentration, the response time might not exceed 20 sec. The corresponding Maude specification of this QoSContract is given by the following fragment of code:

As an example of adaptation strategies application, we consider the case of a violation of the response time in the Temperature context by the actually running component “FireManComp”. In this case, the system detects a violation of QoS Constraint and applies the adaptation strategy that replaces the “FireManComp” by “FireEngineComp” component. Figure 4 shows the result of the adaptation strategy. “FireEngineCom” component that respects the QosConstraint “C1” (response time of FireEngineComp = 20ms) is added to the list of components in the functional system and its service “StartCompFE” becomes running (isActive : true). It replaces “FireManComp” which does not meet the quality requirements.

Maude> ... 
result Configuration : ...

Figure 3. Self-adaptive Firefighting System model.

Figure 4. A strategy application result.
V. Conclusion

In this paper, we have proposed a component-based contractual approach for designing and specifying self-adaptive systems with respect to Quality of Service contracts. The approach establishes a clear separation of concerns between the specification of user-definable QoS quality parameters and quality parameters of the software components. To implement the proposed approach, we have combined the MDE techniques and a formal method in order to provide an intuitive modeling notation, supporting a graphical view, but still having a rigorous syntax and semantics. Such combination also facilitates the use of formal methods in many stages of the development process including the analysis phase that includes validation and verification techniques.

As future work, we intend to exploit main characteristics of formal methods to rigorously verify the behaviors of model-based self-adaptive systems, formal specifications are automatically generated. We will mainly adopt a stochastic model-checking technique to ensure quality properties of self-adaptive systems. Besides, we plan to develop a modeling tool that facilitates the creation and the implementation of quality-aware self-adaptive systems. We aim to integrate formal techniques within the MDE ones. The role of MDE is the definition of system graphical models and formal methods serve to validate and verify the self-adaptive system in order to guarantee that system model satisfies global properties and particularly quality ones. Furthermore, we aim to apply our approach on supplementary case studies in the goal of optimizing the existing quality properties modeling, the verification and implementation capabilities of the self-adaptive systems modeling framework.

ACKNOWLEDGMENTS

This work is published through funding provided under the CMEP/TASSILI project N°08MDU945.

REFERENCES

Consistent Extra-Functional Properties Tagging for Component and Connector Models

Shahar Maoz1, Jan Oliver Ringert1, Bernhard Rumpe2,3, and Michael von Wenckstern2
1 School of Computer Science, Tel Aviv University, Israel http://www.cs.tau.ac.il
2 Software Engineering, RWTH Aachen University, Germany http://www.se-rwth.de
3 Fraunhofer FIT, Aachen, Germany http://www.fit.fraunhofer.de

Abstract—We present a model-driven approach for adding extra-functional properties to component and connector (C&C) models. The approach is based on a tagging mechanism that allows non-invasive extensions of existing languages and their models, here C&C models, with attributes for extra-functional properties. Importantly, our language extension provides means for integrated formal analyses of the consistency of tagged values. Consistency ranges from type-safety and units of quantitative measures to complex dependencies across component hierarchies as well as between component definitions and their instances. We provide a framework for defining and checking rich consistency rules of extra-functional property values based on selection, aggregation, and comparison operators. Our work allows for independent definition and organization of tagged properties to support reuse across models and development stages. The approach is implemented within the MontiCore framework for the C&C architecture description language MontiArc.

I. INTRODUCTION

Component and connector (C&C) models are used in many application domains of software engineering, from cyber-physical and embedded systems to web services and enterprise applications. Their structure consists of components at different containment levels, their typed interaction points, and connectors between them [17]. In addition to expressing functional properties, also extra-functional properties (EFPs) play an important role in successful development of C&C models [10], [24]–[26]. Typical examples of EFPs include worst-case-execution-time, memory and power consumption, security properties, and traceability [1], [21], [23].

We are interested in consistent definition of EFPs for C&C models, commonly expressed in C&C architecture description languages (C&C ADLs) [17]. Previous works in this direction have extended the meta-models of languages or defined special language profiles for adding EFPs to ADLs [4], [26]. These extension mechanisms are typically invasive to the language definition and thus impede extensibility. One approach for extending modeling languages without modifying their meta-model are tagging languages [9]. The first contribution of this paper is a tagging language for non-invasive extension of C&C models with EFPs. This new language supports the tagging of C&C elements in component definitions as well as in their instances.

The consistency of EFP values is crucial throughout the life-cycle of a system. Consistency checks of EFPs have been investigated for various development steps of C&C models, e.g., property definition [26], refinement and subtyping [14], evolution [2], and deployment [25]. The second contribution of this paper is a framework for defining rich consistency rules for tagged EFP values in the context of C&C models. Our consistency rules are specific to an EFP and C&C element and consist of selection, aggregation, and comparison operators. Rules select relevant C&C model elements, aggregate their EFP values, and compare them to determine the consistency of the checked element’s EFP value.


The next section presents our running example. Sect. III presents necessary background. The two main contributions, our tagging language and our consistency framework, are presented in Sect. IV and Sect. V. We present the implementation and a discussion in Sect. VI, related work in Sect. VII, and a conclusion in Sect. VIII.

II. RUNNING EXAMPLE

As running example we use an excerpt of a wind turbine example adapted from [25], [28], as shown in Fig. 1 and Lst. 1. The turbine controller (component type TurbineCtrl) contains two brake controllers (component type BrakeCtrl) to compute force on the turbine and merge the calculated result with brake commands from the main controller MainCtrl.

A team of engineers developing the controller deals with various EFPs including the traceability of component implementations to the requirements catalog and the power consumption of components. The component BrakeCtrl is a safety relevant component and thus requires traceability for safety audits. This extra-functional property is added to the component type definition. As a requirement the power consumption of TurbineCtrl is limited to 4W.

The team created an implementation of component type BrakeCtrl that consumes power of 1W and is traceable. In an attempt to improve safety and provide different instances for subcomponents brCoA and brCoB, the team uses an existing implementation that consumes 2010mW. The chief architect decides to update the power consumption defined for type BrakeCtrl to the new value of 2010mW.

A team member is unsure whether these updates are enough and the C&C model is consistent with respect to its EFPs. She is right to doubt consistency: on the component type level the power consumption of TurbineCtrl (4W) is smaller than two times the power consumption of BrakeCtrl.
Greifenberg et al. [9] presented a mechanism to derive tagging languages for domain specific modeling languages.
The mechanism uses two languages: the tag model, which decorates the domain specific model with additional information, and the tag schema, which defines the tag types that are used in the tag model.

Defining a tagging language based on this mechanism has the following advantages: (1) The model will be kept clean and, therefore, easy to read, not polluted with extra information; (2) Inherent separation of concerns [3], as different people can decorate the domain specific model with their own separated tagging models; and (3) The tagging language references the elements by their concrete syntax as they are defined in the domain specific model.

IV. C&C TAGGING LANGUAGE

This section presents our first contribution, namely, the C&C tagging language. Similar to Greifenberg et al. [9], we have the following tag types: (1) simple tags, when one only cares whether a C&C element is or is not tagged with this information, similar to Java’s marker interface; (2) valued tags, decorating a C&C element with a tag containing a value, such as Boolean, Number, String, enumeration value or a JScience2 quantity (e.g. Power or DataAmount); and (3) complex tags to store several values, such as estimated worst-case-execution time [26] \( \text{wcet} = \{ \text{time}=800\text{ms}, \text{confidence}=50\% \} \). Since complex tags consist of several simple or valued tags, the rest of this paper handles only the simple and value tags.

The EmbeddedTagSchema in Lst. 2 defines a tag schema for C&C models used in an embedded context. It contains one simple tag traceable, which can be applied for component instances and definitions, and three valued tags power, encryption, and reliability. All defined tags start with tagtype, have a name, and end with for plus the C&C element to which the tag type can be applied; the valued tag types have after the name additionally a colon followed by a data type. The data type of a valued tag can depend on the element decorated with it. This is the case for the encryption tag, where port definitions are tagged with a set (+ sign in L. 4) containing one or more values of the enumeration defined inside square brackets; whereas, in contrast, tags for port instances contain exactly one enumeration value. This is due to the fact that a defined port can support multiple encryption modes, whereas a concrete port instance encrypts its data using one concrete algorithm.

B. Tag Model Definitions

After the tag schema is defined, C&C elements can be tagged with it. Listings 3 and 4 tag C&C element definitions (Example1) and instances (Example2). Tag models have a header and body (e.g., Lst. 3, ll. 1-2 and ll. 3-8), containing additional information which is added to the C&C elements. Every header starts with conforms to followed by the tag schema name to which the tag model definition conforms to. The header also includes the tags keyword and the tag model’s name (e.g., Example1). Additionally, the header may contain an optional list of C&C elements (a comma separated list of names after the for keyword) to address these C&C elements’ children directly in the body.

A body is a container for several tag definitions starting with a tag keyword and ending with a semicolon. Each tag definition has at least one C&C element name and one tag type name separated by the with keyword (e.g., Lst. 3, l. 3).

The tag model enables tagging of all C&C elements from Defs. 1 and 2:

1) component definitions CDef, e.g., BrakeCtrl, and instances Cmps, e.g. brCoA;
2) port definitions Pports, e.g., BrakeCtrl.pitchBrake, and port instances Ports; and
3) connector definitions CCons, e.g., brCoA.brakeControl -> parkController.brakeControlA, and connector instances Cons.

A. Tag Schema Definitions

The tag schema defines the types of the tags used to decorate C&C models. One may view tag schemes as meta-models.

Since complex tags consist of several simple or valued tags, the rest of this paper handles only the simple and value tags.

The EmbeddedTagSchema in Lst. 2 defines a tag schema for C&C models used in an embedded context. It contains one simple tag traceable, which can be applied for component instances and definitions, and three valued tags power, encryption, and reliability. All defined tags start with tagtype, have a name, and end with for plus the C&C element to which the tag type can be applied; the valued tag types have after the name additionally a colon followed by a data type. The data type of a valued tag can depend on the element decorated with it. This is the case for the encryption tag, where port definitions are tagged with a set (+ sign in L. 4) containing one or more values of the enumeration defined inside square brackets; whereas, in contrast, tags for port instances contain exactly one enumeration value. This is due to the fact that a defined port can support multiple encryption modes, whereas a concrete port instance encrypts its data using one concrete algorithm.

B. Tag Model Definitions

After the tag schema is defined, C&C elements can be tagged with it. Listings 3 and 4 tag C&C element definitions (Example1) and instances (Example2). Tag models have a header and body (e.g., Lst. 3, ll. 1-2 and ll. 3-8), containing additional information which is added to the C&C elements. Every header starts with conforms to followed by the tag schema name to which the tag model definition conforms to. The header also includes the tags keyword and the tag model’s name (e.g., Example1). Additionally, the header may contain an optional list of C&C elements (a comma separated list of names after the for keyword) to address these C&C elements’ children directly in the body.

A body is a container for several tag definitions starting with a tag keyword and ending with a semicolon. Each tag definition has at least one C&C element name and one tag type name separated by the with keyword (e.g., Lst. 3, l. 3).
Valued tag types must additionally include the tag type’s value, preceded by an equals sign (e.g., Lst. 3, l. 4). Multiple values are assigned by using a comma separated list inside square brackets (e.g., Lst. 3, l. 6).

Line 3 in Lst. 3 adds the `traceable` tag to the component instance `turbineCtrl.brCoA`, because the context is `turbineCtrl`\(^3\) (l. 1). Line 8 in Lst. 4 shows that the domain expert decorating C&C elements needs no knowledge about the C&C meta-model and directly uses the concrete syntax of the C&C model (e.g., Lst. 1, l. 8).

V. CONSISTENCY OF EXTRA-FUNCTIONAL-PROPERTIES

We now present the second contribution of our paper, namely, a framework for the definition of rich consistency rules for tagged extra-functional property values.

We distinguish between the consistency of EFP tags with their tag schema and the more interesting consistency of tagged EFP values in the context of the C&C model. Our rules for checking the consistency of tags and their schema are independent of the specific semantics of the respective EFP. In contrast, the consistency rules for values in the context of the C&C model are very specific to the expressed EFP.

A. Consistency of Tags and Tag Schema

The following rules check for consistency of tags and their tagging schema:

1) tag type names are unique per C&C model element kind
2) tagged C&C elements exist uniquely and are of the kind defined in the schema
3) every C&C element is tagged at most once per tag type
4) the tag value is of the data type defined in the schema
5) the unit of the tag is compatible with the unit in the schema, e.g., `W` and `mW` but not `W` and `s`
6) for complex tags the above applies to every value.

Note that rule 3 does not allow to tag a C&C element twice for the same EFP. While one could define strategies to resolve possible inconsistencies, e.g., considering maximal or minimal values, we added this rule to avoid inconsistencies.

B. Consistency of Tags and C&C Models

In addition to the consistency of tags with their tag schema, the consistency of a tagged EFP value may also depend on its context in the C&C model. More advanced examples of consistency relate to component instantiation and composition in C&C models. It is important to note that the consistency of a tagged EFP value may depend on multiple other C&C model elements and their relation. In addition, consistency may be very specific to the EFP type, e.g., allowing subsets of values or defining their bounds.

To address the challenge of ensuring consistency of tagged EFP values, we define a general framework based on consistency rules. First, each rule defines what tag of which kind of C&C model element it checks. Second, the rule specifies how to select relevant C&C model elements for the check. Third, the rule defines how to aggregate tagged values over the selected elements. Finally, the aggregated value is compared to the value of the checked element, to determine its consistency.

We summarize the structure of consistency rules in Def. 3

Definition 3 (EFP Value Consistency Rule): A consistency rule is a structure consisting of:

- **checks** name of tag and element checked by rule;
- **selection** selects relevant C&C elements to check consistency;
- **aggregation** aggregates values of selected elements; and
- **comparison** compares values to decide consistency.

The next two subsections illustrate consistency definition rules according to Def. 3. Sect. V-B1 presents example rules for the consistency of EFP values in the context of component type instantiation. Sect. V-B2 presents example rules in the context of composition.

1) Instantiation Consistency Examples: Instantiation consistency checks whether the EFPs of C&C model instances conform to the EFPs of their type definitions. To simplify the definition of rules, we employ a general operator `typeOf : Cmps \rightarrow CTDefs`, which given a component instance returns its uniquely determined component type.

**Rule 1 (InstTrace):** If the component type definition is traceable, all instances have to be traceable:

- checks: tag `traceable` of `c ∈ Cmps`
- selection: `t := typeOf(c) ∈ CTDefs`
- aggregation: `v := t.traceable`
- comparison: `v ⇒ c.traceable`

In our example, component type `BrakeCtrl` is tagged as `traceable` in Lst. 3, l. 3. While component instance `brCoB` is not tagged as `traceable`. It will thus be reported by Rule 1 as inconsistent.

**Rule 2 (InstPower):** The power consumption of an instance is at most the power consumption of its type:

- checks: tag `power` of `c ∈ Cmps`
- selection: `t := typeOf(c) ∈ CTDefs`
- aggregation: `v := c.power`
- comparison: `v ≤ t.power`

In our example, both component instances of type `BrakeCtrl` pass the check of Rule 2 with `1W ≤ 2010mW` for `brCoA` and `2010mW ≤ 2010mW` for `brCoB` (see Lst. 3, l. 5 and Lst. 4, l. 4-5).

**Rule 3 (InstEncryption):** The encryption of a port instance must be in the encryption set of the port definition:

- checks: tag `encryption` of `p ∈ Ports`
- selection: `pt := THE\(^4\)pt ∈ typeOf(p.parent\(^5\)).CPorts : pt.name = p.name`
- aggregation: `v := pt.encryption`
- comparison: `p.encryption ∈ v`

In our example, the port instances `main.pitchBrake` and `brCoB.pitchBrake` pass Rule 1, while port instance `brCoA.pitchBrake` violates Rule 3: `AES \notin v = {DES, 3-DES}` (see Lst. 4, l. 6 and Lst. 3, l. 7).

\(^3\) `turbineCtrl` is the top-level instance of the component type `TurbineCtrl` defined in Lst. 1

\(^4\) Definite description operator `THE : P(x)` returns `x` satisfying `P(x)`.

\(^5\) The parent of a port is the component it belongs to.
2) **Composition Consistency Examples**: Composition consistency checks whether the EFPs of C&C model elements are consistent across their composition. The following example rules address consistency on the type level. Similar rules can be defined on the instance level.

**Rule 4 (CompPower)**: The combined power consumption of all subcomponents is at most the power consumption of the composed component:

- checks: tag power of $c \in CTDefs$
- selection: $S := c.CSubs$
- aggregation: $v := \sum_{(name,ct) \in S} d.power$
- comparison: $v \leq c.power$

In our example, the component type **TurbineCtrl** contains subcomponents **brCoA** and **brCoB** of type **BrakeCtrl** and $v = 2010mW + 2010mW + \ldots \leq 4W$, i.e., the tagged value of 4W violates Rule 4 and is thus inconsistent.

**Rule 5 (CompEncryption)**: A receiver port must support at least one encryption of its sender ports.

- checks: tag encryption of $p \in CPorts$
- selection: $P := \{p' | \exists con \in CCons : (p' = con.src \land p = con.tgt)\}$
- aggregation: $v := \bigcap_{p' \in P} p.encryption$
- comparison: $v \cap p.encryption \neq \emptyset$

In our example, the port **BrakeCtrl.pitchBrake** supports encryption DES and 3-DES (Lst. 3, l. 7), and is a receiving port for **MainCtrl.pitchBra**ke with encryption AES and RSA (Lst. 3, l. 6). Rule 5 will report port **BrakeCtrl.pitchBrake** as inconsistent.

The above examples of Rule 1 to Rule 5 show the expressiveness of consistency rules of our framework that cover various EFPs and C&C element relations.

**VI. IMPLEMENTATION AND DISCUSSION**

**A. Implementation**

We implemented the tagging language together with its EFP consistency checks in MontiArc [11], a textual C&C modeling framework. The workflow of processing MontiArc models is as follows: (1) The parser converts the textual input model to an abstract syntax tree (AST); (2) The AST is traversed to store all model definitions as symbols in the symbol table (ST); (3) Based on the AST and the model definition symbols, all C&C instances’ symbols are created by (a) resolving the component extension chains, (b) binding all generics, and (c) recursively instantiating all subcomponent hierarchies.

Although the MontiArc tagging language is based on the same concepts and concrete syntax as the one presented in Greifenberg et al. [9], our approach adds tags to the ST while their method enriches the AST directly. Decorating the ST has the following advantages: (1) The ST represents the semantic model [6], [19], and, thus, in contrast to the AST (see **ArcConnector** and **ArcSimpleConnector** in [11]), each C&C element in Def. 1 and Def. 2 is represented by exactly one symbol; (2) Since MontiArc’s AST is a mixture of C&C model definitions and instantiations for readability purposes, it is not possible to tag chains of instances (e.g., **turbineCtrl.brCoA.brakeControl**) differently as both of them are represented by the same AST node; the ST solves this problem by providing separate symbols for C&C definitions and for all C&C instances; and (3) Contrary to the AST, the ST is a graph with additional references making it possible to easily navigate through it and execute more complex selections as needed for consistency checks, e.g., in Rule 5.

**B. Discussion**

The ST’s resolving mechanism – containing bidirectional navigation together with symbol filtering and adaption – as well as the ability to add several different EFP tags to the same C&C element symbol, allows to check constraints between different EFPs, such as time vs. data amount. We consider this to be a nice property of our work.

Regarding composition, the complex tag is a composition of tags (which can be complex themselves). This way, it is possible to logically group EFPs (e.g., power consumption and confidence) as is suggested by Shaw [27] and implemented by Sentilles et al. [26].

It is important to note that extra-functional properties may evolve, together with knowledge about a system [2], [14], [18]. They may also depend on the viewpoint of stakeholders [4], which can be solved by tagging the extra-functional tags again with the stakeholders name. Since it is possible to tag elements or tags several times, one tag can be tagged by different stakeholders. Since all the tags (including tags of tags) are stored in the same ST, consistency constraints can also be expressed between tags of tags with the same concepts shown in Sect. V. For simplicity, in this paper we did not discuss the tagging of one element multiple times. Rather than multiple tagging, we consider organizing ownership by using separate tag models. Meta-model approaches cannot easily do it but it is natural in our approach.

Note that depending on the EFP type and its composition and instantiation semantics, the consistency of composition on the type level and the consistency of instantiation, do not guarantee consistency of instance composition. Our presented port encryption semantics in Rule 3 and Rule 5 still allow for inconsistent instance compositions. We believe that a unified framework, as the one we have presented, is a first step towards reasoning about EFP consistency.

**VII. RELATED WORK**

Espinoza et al. [4] annotate UML/MARTE models with quantitative EFPs. One of their main goals is to distinguish sources of EFPs, e.g., requirement vs. measurement during test. This could be done with complex tags containing the actual value and the value source.

Grunske [10] presents an evaluation framework for EFPs consisting of four elements: usage profile, evaluation model, composition algorithm, and evaluation algorithm. We focus on solutions for the composition and evaluation parts.
Sentilles et al. [26] present a meta-model for integrating non-functional properties into C&C models. Their model allows to specify multiple values per attribute with validity conditions, dependencies, and version information. Since our models are all text-based, external version control mechanisms such as Git or SVN can handle EFP's history. All the other information, e.g., validity condition or dependencies, can be expressed via complex tags. Finally, the complete information tagging is available in the symbol table for consistency checks.

Leveque et al. [14] present a way to express refinement of attribute values for instances and subtypes of components. Similar rules can be defined in our framework. For MontiArc, one can formulate EFP consistency constraints for refinements of port types (Java-like inheritance and generics) as well as for refinements of components (inheritance of component types).

Cicchetti et al. [2] introduce a framework for evolution of EFP values and present, as an example, how the change of the worst-case-execution time (WCET) of a component requires updating the WCET of its parent component. It is possible to extend our framework to support similar evolution scenarios.

Sapienza et al. [25] motivate the benefit of composing EFPs of components in embedded system design. They present a general classification of property composability and provide many examples. In their terminology, our rules for consistency of composed components compute non-emergent, directly composable properties. Additional types of composition identified by Sapienza et al. [25] require further information beyond the tagging language and C&C model.

VIII. CONCLUSION

We presented a mechanism to enrich existing C&C models with consistent extra-functional properties. The strengths of our approach are: (1) All EFPs are stored in separate files to avoid model pollution, (2) The tag schema is used to validate tag models to avoid tagging mistakes (typos, wrong units, wrong C&C element, etc.), (3) EFP-specific consistency rules between tagged C&C elements (C&C definitions as well as C&C instantiations) can be defined and verified.

We illustrated the two main contributions, our tagging language and consistency rule framework, using several examples. The examples cover many scenarios of consistency defined also in related work. As future work we consider the application of extra-functional property tags to C&C views, with corresponding verification between views and models, extending [16].

Acknowledgements Part of this work was done while Shahar Maoz was on sabbatical as visiting scientist at MIT CSAIL. This research was supported by a Grant from the GIF, the German-Israeli Foundation for Scientific Research and Development.

REFERENCES

Fault-aware Pareto Frontier Exploration for Dependable System Architectures

Lukas Märtin*, Hauke Baller*, Anne Koziolek†, Ralf Reussner†
*TU Braunschweig, Germany
Email: {maertin,baller}@ips.cs.tu-bs.de
†Karlsruhe Institute of Technology, Germany
Email: {koziolek,reußner}@kit.edu

Abstract—While designing dependable systems, a large number of asset combinations (system configurations) with contrary quality objectives needs to be investigated. Basically, each feasible configuration should be investigated. For fault-tolerant embedded systems this problem is extended by anticipating hardware faults leading to changed deployments of stressed resources in redundant constellations. The identification and evaluation of the best-fitting configuration remains a computationally intensive and difficult task at all.

We propose a multi-stage approach (1) to sample Pareto-optimal configurations for redundant system designs within hostile environments, (2) to check satisfiability of structural constraints and (3) to measure and identify quality degradation in fault scenarios. Thus, allowing developers to identify design flaws, leading to large quality degradations in case of emerging faults. We use genetic algorithms (NSGA-II) for sampling a wide range of system designs and demonstrate our approach by means of an exemplary fault-tolerant system.

I. INTRODUCTION

In fault-tolerant software design, the provision of dependable systems is charged with high expenses. In particular, a diversity of replacement units (redundancies) needs to be specified and addressed in redundancy methods and distributed well to successfully maintain faults [1]. Thus, developers are concerned with distinguishing many feasible system configurations, mostly equipped with redundant hardware resources. To meet the functional requirements and simultaneously optimize multiple dimensions of system’s quality, expensive explorations of the design space are inevitable to find a best-fitting system variant for deployment.

This challenge is getting even more complicated by further considering further variants for reconfiguration upon hardware resource faults. Such potential faults of stressed resources in hostile environments, e.g., cosmic radiation harming space crafts and satellites, might be predicted by methods like Fault Tree Analysis, but the consequences on quality and functional validity are still expensive to inspect appropriately for rich design spaces. Here, each feasible configuration should be evaluated in face of all alternatives, leading to an exponential complexity of comparisons. Even if the initial commit of a fault-tolerant system is usually more expensive than the initial commit of a regular system, the exploration of the design space has to be done in a systematic manner.

From an architecture-oriented point of view, the separation and modeling of software components and hardware resources lifts the exploration to an abstract level of component-based software engineering for embedded systems [2]. Here, the deployment of components describes use-relations to the platform (resources for execution). Figure 1 shows an example for a redundant system design with an excerpt of feasible configurations, defined as the reconfiguration space by the developer. Each configuration requires a different subset of hardware resources from the platform and result in varying ratings (values) for quality attributes. Figure 1 depicts the differing uses of hardware resources (R1, R2) by software components (C1, C2, C3) during execution. Configurations (rows) are defined by selecting (marked by 1) elements. Some selections are optional, denoted with dashed lines. Each defined configuration is rated, leading to rising, falling, or constant quality changes (arrow directions). Here, we consider the quality attributes energy, performance and maintenance costs. Each configuration is also validated against a set of structural constraints, representing the basic functional relations in the architecture model. As soon as one of the hardware resources is marked as faulty (defined as fault scenario), some configurations including fault-affected components may no longer be executable. To handle this partial loss of fault-tolerance, the developer needs to extend the former reconfiguration space by additional configurations, not relying on the faulty hardware resources. Each configuration has to be identified, quality-rated and compared to the existing (valid) configurations. This procedure supports the developer in identifying possible
alternative configurations for an assumed fault scenario. In
order to focus on significant degradations of quality attributes, a
user-definable threshold for quality degradation is desirable from a developer’s point of view.

In previous work [3], we arranged alternative configurations as nodes in a graph structure called Architecture Relation Graph. Edges result from the reduction of available hardware resources caused by faults. For edge prioritization, the qualities of each configuration are investigated. Such a strict hardware-oriented procedure is not feasible to evaluate alternative configurations efficiently while considering quality attributes. In this paper, we therefore investigate the measurement of quality distances between configurations, including validity checking according to required hardware resources.

**Foundations and Related Work**

Our work relates to the concept of Degrees of Freedom [4] to define and evaluate variability in architecture design. Possible variation points are specified as explicit part of the architecture model. A genetic algorithm explores the design space to find Pareto-optimal solutions, i.e., the supremum of all feasible solutions with respect to contrary objectives, respecting a set of quality attributes. This procedure can be applied to support design decisions and to explore potential reconfiguration options [5]. To apply the approach to its full extend, we need to create rich design models to gather ratings for quality attributes by simulation. Instead in this paper, we use simplified representations of design models and abstract quality measurements in order to provide a lightweight implementation of the basic concepts in our approach.

The sampling of system configurations is performed by the genetic algorithm NSGA-II [6]. Echtle et al. [7] also apply such algorithms to identify fault-tolerant system designs on a high level of abstraction. This work is focused on finding critical fault combinations leading to invalid system designs. We describe the variation space of the system explicitly to check validity of many sampled designs in a short period of time. More precisely, we use a propositional logic formula to describe the design space of examined systems and imitate faults by disabling operation-critical hardware resources to restrict feasible variations in configurations. Technically, we represent relevant parts of the architecture model as features in a feature model. Feature models provide a comprehensible graphical representation of a variant rich system. Relations between functional components and hardware resources are defined by constraints in the feature model. The fitness of a feasible solution, i.e., a configuration validated by the feature model, is based on quality assignments annotated as property to each feature.

Frey et al. [8] inspect reconfigurations as deployment options for cloud-based systems derived by genetic algorithms. The authors predefine rules at design time for systematically modified deployments of a system upon changed circumstances in operation, e.g., system overloads. Similar to that, Jung et al. [9] adapt a running system by policies derived at design time. For that, a decision-tree learner is trained with feasible system configurations, generated from queuing models. Both approaches guide the developer to identify alternatives for reconfiguring a system. However, the reconfiguration space is not explicitly explored to identify quality drops upon faults in unstable hardware/software systems. Our approach identifies such gaps and prioritizes near-by alternative configuration for recommendation and decision support. In relation to our distance measurement in neighborhood of faulty configurations, Barnes et al. describe relations between architectures as candidate evolution paths [10]. These paths specify a search-based reconfiguration process from a source to a pre-defined target architecture via a sequence of transient architectures. The goal is to shorten the paths to minimize reconfiguration efforts. However, we aim to retain as much system quality as possible without defining a target architecture manually. J. R. Schott [11] defines a metric called spacing to measure how well non-dominated individuals on the Pareto-frontier are distributed with respect to their neighbors. Following that idea, Gong et. al [12] also use a neighbor-based technique to inspect the crowding distances of non-dominated individuals and select minority isolated individuals. Thus, they refine recombination and mutation by determining nearest neighbors of less-crowded individuals for the next optimization iteration. In our approach, we also explore dominated neighbors to find design alternatives for individuals that became infeasible due to resource faults by comparing and minimizing distances in the objective space.

In the area of search-based approaches, Garvin et al. [13] combine heuristic search with Feature Modeling. By using simulated annealing, the authors extend a test generation algorithm to determine valid feature configurations. Based on an array representation of a feature model, the algorithm perform pair-wise changes of feature selections. After each change a SAT check on the feature model is done. The fitness function of the optimization tries to maximize coverage of feature pairs. Similarly to that, Ensan et al. [14] apply a genetic algorithm to generate products (configurations) in accordance to a feature model. In both approaches, each gene of a chromosome represents a feature. The fitness of a product is coverage-oriented by evaluating the variability points and their constraints from the feature model. In our approach, a feature model provides the structure for variation points and restricts the selection of configurations by constraints. However, we do not consider coverage measurements, but guide our approach by minimizing distances between configurations. Furthermore, we assume, that the number of variation points is decreased by faulty features, potentially leading to faulty configurations. Several other tools from literature apply genetic algorithms to generate products from a feature model in testing Software Product Lines, e.g., PLEDGE [15].

**II. Multi-Stage Architecture Design Analysis**

Our approach searches for appropriate architectural design alternatives for reconfiguration under the assumption of predictable hardware resource faults. The resulting set of configurations needs to be ordered and prioritized by multiple quality
In the first two stages individual optimization runs are performed with different settings. In stage one, no faults are considered and each locally optimal and valid alternative configuration is added to the Pareto-frontier. Although fault-prone configurations of the first run might be detected for reevaluation easily, another run is needed to determine previously dominated non-faulty configurations. A second Pareto-frontier without any faulty configurations results from the second run performed in stage two. In stage three, the comparison of both Pareto-frontiers is prepared by injecting the same fault scenarios in the results of the first run. Therefore, all faulty configurations are separated from the healthy (still fully operational) ones. Faults in resources might lead to side effects in quality evaluation, e.g., if a faulty resource is cold-redundant to another still healthy resource of the configuration. Thus, an a posteriori re-calculation of qualities of each healthy configuration is performed. In a reconfiguration process, an alternative configuration for a faulty configuration seems to be optimal if minimal quality losses is archived. As measurement for comparing neighbors of faulty configurations, the Euclidean distances between the faulty configurations and healthy ones are determined in stage four. Next the distance measurement between faulty configuration of the first run and the newly sampled configurations from the second run is performed to find new nearest neighbors again. From both comparisons distance matrices result. To figure out the best-fitting alternative configurations, the matrices are merged to find in the combined results the nearest neighbors for each faulty configuration. This action already leads us to a basic transition structure to judge reconfiguration decisions. Our approach is intended to support design and maintenance activities. Therefore, the final stage five refers to the presentation of results. This allows developers of fault-tolerant systems to identify design flaws leading to potentially large quality degradations in case of faults. The presentation consists of statistics about the amount of nearest neighbors of faulty solutions and quality differences in distance matrices. Furthermore, large degradations are highlighted to identify needs for design improvements. Ultimately, the developer has just to set a threshold for distance values as the upper limit for acceptable degradations in each quality dimension. In the result all best-fitting alternatives for a configuration, addressed by a fault scenario, are presented including quantified quality differences.

**Implementation**

Our approach was prototyped as an ECLIPSE plug-in\(^1\). Thereby, we combined the plug-in FEATUREIDE [16] for variant-rich feature modeling and validity analysis and the JMetal [17] library for multi-objective optimization with meta-heuristics. Each problem-essential software component and hardware resource of the system is identified with a feature within a feature model. Furthermore, constraints in the model describe cross-cutting concerns between components and resources, i.e., implications or excludes. In order to improve readability and to represent an architecture-oriented structure, abstract features with no corresponding architectural elements are used. To define objectives for the optimization, the root of the feature model is annotated with a list of considered quality attributes. Based on this list, each concrete feature holds discrete ratings of one or more of these quality attributes. During optimization these assignments are evaluated and summed up\(^2\) for each feature contributing to the configuration under the fitness analysis. We do not consider additional side constraints to restrict the optimization objectives beyond the ones from the feature model.

We apply the NSGA-II implementation from the JMetal framework to sample binary decision vectors. For each 1 occurring in that vector, a corresponding feature in the feature model is selected; without any propagation guided by the rules in the feature model. The whole selection is validated by the SAT checker of the FEATUREIDE core engine. If the

\(^1\)https://github.com/lmaertin/modcomp

\(^2\)Function for aggregation can be customized, e.g., Mean or Median
sampled configuration is valid, the solution is rated by the quality assignments of the configuration, gathered from the feature annotations before. If the SAT check fails, the solution is downgraded in each quality dimension.

For the given fault scenarios, we mark faulty resources by deselecting features that are related with deployments to such resources. During the first optimization run, such faults are initially ignored. After the run is completed, faults are injected and all configurations from the first run are re-evaluated in FeatureIDE. By rechecking satisfiability, some alternative might be no longer valid. The resulting faulty (non-valid) and healthy (still valid) configurations are stored in two independent sets for further processing. In addition, also the quality assignments of healthy configurations are reevaluated according to the potentially changed number of addressed quality attributes affecting the aggregated sums.

The presentation of results provides all data about feasible alternative configuration to the developer in a comprehensive manner. In addition to general statistics (number of solutions of both runs, ratios faulty vs. healthy and faulty vs. second run), the data of the new reconfiguration space is aggregated for decision support.

Because of usually varying qualities in multi-objective optimization, it is reasonable to let the user define a threshold for qualities for alternative configurations. In this way, the identification of a rich neighborhood set of configurations for each faulty configuration is promoted. Without a distance threshold, just the (one) best-fitting neighbor would be computed. After the data processing is completed, all distances are ordered beneath the threshold. On the one hand, the subset of results is shown as a distance matrix and new neighbors from the second run are highlighted. On the other hand, gaps between the Pareto-frontiers are investigated to identify most significant quality impacts. For that, a list of largest distances between faulty configurations and nearest neighbors is created.

The aggregated information can be used by the developer to optimize the design, e.g., by adding additional resources, and to derive rules for reconfigurations during self-maintenance.

III. EVALUATION

For evaluation purposes, we applied our tool-supported approach to a fault-tolerant vending machine. For simplicity, the system deals with a well-known application scenario enhanced by redundancies and a fair reconfiguration space. Thus, the scenario addresses the domain of fault-tolerant embedded system design regarding redundant sensors and actuators. In addition, the system relies on software-intensive sensing and control, instead of pure mechanical solutions.

Fault-tolerant Vending Machine

The vending machine offers still water, sparkling water and coke in cups, optionally chilled. Payments are accepted by coins, notes and money card. Some sensors and actuators can partially emulate other ones to support a high degree of fault-tolerance without cost-intensive replication of resources. For instance, water can alternatively be served by the coke injector after that injector was cleaned by an additional resource. Thus, each of these reproductions leads to changes in required resources and system’s quality.

The system contains the following sensors and actuators.

**Sensors:** Buttons (still water, sparkling water, coke, return money), counters (coins, notes), a money card terminal and filling-level meters (water, coke-mix, collector tray for cleaning, cups), and a thermometer for chilling-control.

**Actuators:** Mixers (coke, CO₂), flow controllers (pump, gravity), valves (water, coke) and injectors (water, coke), money changers (coins, notes).

As a baseline for complexity analysis and satisfiability checks during configuration validations, we specified our design by a feature model, depicted in Fig. 3. The resources are represented as concrete features (dark blue boxes) in the model and labeled with indexes from 0 to 20. In total, our system provides a design space of about 7,700 valid configurations with a variety of degradations in all quality dimensions. We assume that customers prefer to drink chilled drinks and like coke more than sparkling water as well as sparkling water more than still water.

**Quality Attributes and Fault Scenarios**

For optimization, we defined a set of quality attributes to be minimized. (1) pollution to observe the compliance of hygienic value limits, (2) taste deviation according to company’s standards, (3) response time representing the time to drink delivery, and (4) energy consumption of the machine.

In order to evaluate our approach, we use a fault scenario with significant impact on the design space, i.e., affecting more than just optional features. By defining the resources CokeInjector and Pump as faulty, one half of initially sampled configurations is no longer valid. Thus, the developer has to figure out which alternative configurations are best-fitting.

**Results and Findings**

The evaluation is performed on an Intel i5 CPU at 2.5GHz with 16GB of memory running Mac OS El Capitan, Java 8 and Eclipse Neon (FeatureIDE 3.0.1, jMetal 4.5.2). The NSGA-II was set to a population size of 100 and 250 iterations for demonstration purposes.

After the first three stages are performed, the following statistics result from our experiment.

- #Solutions first run: 4
  - #Faulty: 2
  - #Healthy: 2
- #Solutions second run: 2

Thus, the comparison of distances between faulty and healthy as well as faulty and new sampled is done both times in ratios of 2:2. We pruned duplicate solutions, leading to a small set of unique optimal solutions for the example system.

With a threshold set to a maximal distance of 0.65, a distance matrix for each faulty solution is generated. Due to lack of space, in Table I the neighbors for only one faulty configuration are shown. The configurations are shown as binary vectors (optimization solution) representing the
selection of concrete features in order of depth-first search corresponding to the indexes given in Fig. 3. Newly appearing nearest neighbors from the second run are highlighted in bold font. The quality assignments refer to the quality attributes and their order introduced before. Further neighbors with distances above the threshold value are hidden by “...”.

By inspecting the values, the developer can figure out the configuration (1000011001001101010100) as the nearest therefore and best-fitting neighbor of the faulty configuration. According to the feature model, the resulting vending machine sells still water via card payment, monitors water fill level and number of remaining cups, supports CO₂ mixing and uses a gravity-based water flow control towards a water value and a water injector. In this case, this configuration was randomly sampled in first and second optimization run. To assure fault-tolerance, the nearest neighbors shall also be considered in the rule set for run-time reconfiguration. Thus, all neighbors under the given distance threshold are added to a new set of solutions, representing the reconfiguration space. For optimization of the design space, the largest distances between solutions are also investigated. Our implementation performs pair-wise comparisons to find the largest distance between quality dimensions in objective space, i.e., gaps in Pareto-frontiers. In our experiments, we figured out the largest distances in solutions for all quality dimensions as listed in Table II. Subsequent to those results a developer may use an appropriate tool to visually explore gaps in the solution, e.g., by hierarchical cluster analysis in GNU R.

To visually present our idea, we performed an optimization with two quality dimensions (Response Time and Energy Consumption), resulting in the 2D-plot in Figure 4. Faulty solutions are colored in red, still healthy solutions are green and new solutions from second sampling are shown in blue. The plot shows a large gap between the Pareto-frontier of faulty solutions and the frontier containing all alternative solutions from the second run. Here, we suggest to recapture the design to minimize that gap by resource changes contributing to meeting the objectives.

Discussion

During evaluation we were faced with some algorithmic characteristics in optimizing multiple objectives. Despite of differing assignments of quality attributes and a complexity of about 7,700 feasible configurations, the generation led to just a few unique configurations and many duplicates. We presume, that this is caused by the small-scaled application scenario and side-effects by similarities in quality assignments. Furthermore, we do not consult constraints for objectives, e.g., minimal acceptance values as lower bounds. Nevertheless, our gap investigations were also applicable by considering only widely spread objective values. Following the idea of Deb et al. [18], an ϵ-dominance might support the reduction of such gaps by a better diversity to be maintained in a population.

Table I: Distances for Faulty Solution (1001001001001101010101)

<table>
<thead>
<tr>
<th>Dis.</th>
<th>Qual. Assignments</th>
<th>Solution Vector</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.58</td>
<td>0.3 1.4 1.1 0.0</td>
<td>100001100100101101000</td>
<td>healthy</td>
</tr>
<tr>
<td>0.58</td>
<td>0.3 1.4 1.1 0.0</td>
<td>100001100100101101000</td>
<td>second</td>
</tr>
<tr>
<td>0.64</td>
<td>0.3 0.9 1.2 0.5</td>
<td>1001001001001011010101</td>
<td>healthy</td>
</tr>
<tr>
<td>0.64</td>
<td>0.3 0.9 1.2 0.5</td>
<td>1001001001001011010101</td>
<td>second</td>
</tr>
</tbody>
</table>

Table II: Largest Gaps in Value Assignments to Objectives

<table>
<thead>
<tr>
<th>Objective</th>
<th>Gap</th>
<th>Assignment 1</th>
<th>Assignment 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00</td>
<td>0.3 1.4 1.1 0.0</td>
<td>0.3 1.4 1.1 0.0</td>
</tr>
<tr>
<td>2</td>
<td>0.00</td>
<td>0.3 1.4 1.1 0.0</td>
<td>0.3 1.4 1.1 0.0</td>
</tr>
<tr>
<td>3</td>
<td>0.09</td>
<td>0.3 0.9 1.2 0.5</td>
<td>0.3 1.4 1.1 0.0</td>
</tr>
<tr>
<td>4</td>
<td>0.54</td>
<td>0.3 0.9 1.2 0.54</td>
<td>0.3 1.4 1.1 0.0</td>
</tr>
</tbody>
</table>
IV. CONCLUSIONS

Even if existing techniques for fault-tolerant system design assist the developer in identifying necessary redundancies, additional best-fitting configurations have to be figured-out. Our approach guides the developer through the subset of the remaining design options after a hardware resource fault is injected. Under the consideration of such a fault scenario, Pareto-optimal solutions are sampled and decision support to identify nearest alternates to a faulty configuration is provided in five process stages. In addition, large gaps in a system’s quality can be shown with our tooling to find flaws in redundant system design.

Future Work

We provide lightweight tooling for mass-generating solution set and gap exploration of Pareto-frontiers. Based on our previous experience, we plan to integrate our exploration concept within the Palladio toolset and its add-ons. For full-fledged architecture modeling [19] we will apply PALLADIO BENCH3 and PEROPTERYX4 to define variability in models. Also the sampling with genetic algorithms is performed there. Ratings of quality attributes are gathered by the simulation engine SIMUCOM. We will make use of the results for distance comparison and gap exploration proposed in this paper. In particular, the findings of this paper will contribute to the improvement of our decision structure Architecture Relation Graph [3] and on-going work in area of hierarchical cluster analysis. The final selection of which configurations from the Pareto-frontiers are added to the graph still relies on trade-off settings preferred by the developer. Such trade-off analysis is supported by our tool AREVA5. Based on the work of Florentz et al. [20] contrary quality properties are normalized by conversion function and ordered hierarchically with weightings. This analysis needs to be integrated with the distance measurements from our tool prototype presented here.

We plan to comprehensively evaluate the integrated tool-supported approach at whole with a case study. For that, we will build upon our previous findings in the domain of space systems [21]. As a real-world case study, we have access to a system design of a micro-satellite provided by one of our industrial partners. The system has a high degree of inherent availability implemented by autonomy mechanisms and a large number of redundant hardware resources. We will extend our idea from previous work [22] in enhancing the system by replication-redundant capabilities as addressed here.

ACKNOWLEDGMENT

This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future Managed Software Evolution. The authors would like to give special thanks to Axel Busch for fruitful discussions on tool integration within PEROPTERYX.

REFERENCES


3http://www.palladio-simulator.com
4https://sdqweb.ipd.kit.edu/wiki/PerOpteryx
5https://github.com/lmaertin/areva2
Towards Concern-Oriented Design of Component-Based Systems

Jörg Kienzle
McGill University
Montreal, Canada
Email: joerg.kienzle@mcgill.ca

Anne Koziolke, Axel Busch, Ralf Reussner
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: {koziolke, busch, reussner}@kit.edu

Abstract—Component-based software engineering (CBSE) is based on defining, implementing and composing loosely coupled, independent components, thus increasing modularity, analyzability, separation of concerns and reuse. However, complete separation of concerns is difficult to achieve in CBSE when concerns crosscut several components. Furthermore, in some cases, reuse of components is limited because component developers make certain implementation choices that are incompatible with the non-functional requirements of the application that is being built. In this paper we outline how to integrate CBSE and concern-oriented reuse (CORE), a novel reuse paradigm that extends Model-Driven Engineering (MDE) with best practices from aspect-oriented software composition and Software Product Lines (SPL). Concretely, we outline how to combine the Palladio Component Model (PCM) capable of expressing complex software architectures with CORE class and sequence diagrams for low-level design. As a result, multiple solutions for addressing concerns that might even crosscut component boundaries can be modularized in a reusable way, and integrated with applications that reuse them using aspect-oriented techniques. Additionally, thanks to CORE, component developers can avoid premature decision making when reusing existing libraries during implementation.

I. INTRODUCTION

With the ever increasing complexity of software, traditional approaches of building software from scratch become more and more inefficient in terms of productivity and cost. Component-based software engineering (CBSE) [1] is a reuse-promoting way of developing software that is based on defining, implementing and deploying loosely coupled, independent components. Each component forms a modular and cohesive unit and provides clearly defined functionality, encapsulated behind provided interfaces. In case the implementation of the component depends on functionality provided by other components, these dependencies are documented with required interfaces. Ideally, when building an application with CBSE, only application-specific components have to be implemented from scratch, and general functionality is provided by reusing existing components from the component repository. A running application is obtained by selecting the appropriate components based on their interfaces, assembling them following a well-defined software architecture, and deploying them onto computational, communication and storage resources.

CBSE has multiple advantages. For example, it promotes modularity [2] and separation of concerns [3], as components can be designed by different developers, potentially using different programming languages, and implementation details hidden behind well-defined interfaces. Since components can be developed independently, developers can work on components according to their specific business or technical knowledge, isolating them from the complexity and unrelated details of the remaining part of the application, and allowing them to focus on the development tasks that they are experts in.

Another advantage of CBSE is that thanks to the provided and required interfaces, replaceability is promoted. A required component can be replaced by other compatible ones that implement the same functionality in different ways. This in turn increases maintainability, since component implementations that are no longer suitable can be substituted easily.

Furthermore, the explicit software architecture present in CBSE forms an additional layer of structuring that makes it possible to reason about software at a higher level of abstraction. In particular, model-driven engineering (MDE) approaches can be applied at the software architecture level to analyze, verify and predict desired properties of applications under development. For example, automated MDE-based approaches such as [4] can exploit the degrees of freedom that component substitution and flexible allocation strategies offer to perform multi-criteria design exploration.

However, although CBSE constitutes an important step forward, reuse in the context of CBSE is in practice not as straightforward. For example, substitutability and reuse of components, in particular COTS components, is sometimes difficult to achieve [1]. Even if the provided functionality and required dependencies match, the non-functional requirements, e.g., performance or memory/power consumption, can prevent reuse in a particular application context. This is most often due to the fact that bottom-up development approaches such as CBSE force developers to make most implementation choices at development-time when the requirements of the reuse context are unknown.

Moreover, separation of concerns is difficult to achieve in CBSE when the concern does not align with the structure imposed by the software architecture [5]. This is predominantly the case for concerns that are inherently distributed, as by their very nature their functionality crosscuts multiple computational resources. Since classic CBSE is designed in such a way that the unit of reuse is the component, concerns that crosscut component boundaries can not easily be packaged. This hinders reuse, and additionally prevents the rigorous application of information hiding principles.
Last but not least, CBSE does not provide support for explicitly expressing the fact that there are often many solutions and possible software architectures to address specific functional or non-functional development issues. As a result, a software architect that wants to put together an application by reusing components from the component repository might directly select a specific solution, i.e., set of components and architecture, and oversee potential alternatives.

In this paper we outline how to integrate CBSE and concern-oriented reuse (CORE), a novel reuse paradigm that extends Model-Driven Engineering (MDE) with best practices from separation of concerns, goal modelling and Software Product Lines (SPL). With its explicit variation, customization and usage interfaces, advanced software composition based on aspect-orientation, and support for delayed decision making, CORE has the potential to enhance CBSE to overcome the shortcomings mentioned above.

The remainder of the paper is structured as follows. Section II illustrates the problem with a motivating example. Section III presents the most important concepts of CORE, compares CORE with CBSE, and outlines the main integration challenges. Section IV explains concretely how we envision to integrate the Palladio Component Model (PCM) [6], a modelling formalism capable of expressing complex software architectures with performance, cost and reliability impacts, into the existing CORE framework reference implementation that supports low-level design with class and sequence diagrams. The last section presents a perspective on the potential benefits that a successful CORE/CBSE integration might provide in a long run.

II. MOTIVATING EXAMPLE

Let us assume a software architect develops a web store. The web store is comprised of four main components, namely a store, a basket, checkout and shipping component. These components operate together when a customer processes an order: Store interacts with basket, while basket forwards its content to checkout (for payment). Finally, checkout triggers shipment.

Payment is a recurring issue in many commercial applications, and hence it is not surprising that multiple third parties offer payment processing systems that can be integrated into component-based systems. Such systems provide essential payment-related functionality, e.g., payment processing, which covers standard payment interactions and connection to multiple payment solutions, i.e., credit cards, PayPal, payment verification, which keeps track of payment status and provides reliable confirmation that payment was successful, and billing, which creates and distributes customer bills.

Including this functionality into the web shop application affects the software on multiple levels. The software architecture changes, since the payment system is constituted of additional COTS components that need to be considered during deployment. Furthermore, these new components need to be connected to the business components according to the new control flow that includes payment. For instance, after checkout, but before shipping, payment verification needs to occur. Similarly, a bill should only be sent out after successful shipping.

The internal design of some components also needs to be updated. For example, somewhere within the functionality of checkout, the payment processing needs to be invoked. This involves changing the internal behaviour of checkout, but also adding a required interface to checkout that needs to be linked to the component that provides payment processing.

While all third-party solutions for payment offer the main functionalities outlined above, they do so in different ways, with a varying number of components and interfaces, different control flows and dependencies on other components, and with different quality (e.g., performance, reliability, cost).

Integrating CBSE and CORE would make it possible to express the variability of available solutions, and for each solution describe the architectural variations, if any, and design integrations, if any. For each solution, the concern would encapsulate the architectural models and detailed design models specifying the solution, as well as how to apply the solution within an application. The CORE reuse process would assist the software developer in choosing a solution, and ensure that the solution is correctly and consistently integrated with the application architecture and design. Why this is the case is explained in the following section, which presents CORE and how it relates to CBSE in more detail.

III. CORE AND CBSE

A. Background on CORE

In concern-oriented reuse (CORE), software development is structured around modules called concerns that provide a variety of reusable solutions for recurring software development issues. Techniques from Model-Driven Engineering (MDE), SPL, and software composition (in particular aspect-orientation) allow concerns to form modular units of reuse that encapsulate a set of software development artifacts, i.e., models and code, describing relevant properties of a domain of interest during software development in a versatile, generic way [7]. Concerns decompose software into reusable units according to some points of interest [3], [2] and may have varying scopes, e.g., encapsulating several authentication choices, communication protocols, or design patterns. Consequently, the models within a concern can span multiple phases of software development and levels of abstraction (from requirements and analysis models, to design models to code).

The main premise of CORE is that recurring development concerns are made available in a concern library, which covers most recurring software development needs. Similar to class libraries in modern programming languages, this library should grow as new development concerns emerge, and existing concerns should continuously evolve as alternative algorithmic and technological solutions become available. Applications are built by reusing existing concerns from the library whenever possible, following a well-defined reuse process supported by clear interfaces [8]. The same idea is applied to the development of concerns as well: high-level/more specific concerns can reuse low-level/more generic concerns to realize the functionalities they encapsulate. In the end, the software
architecture of software developed with CORE takes the form of a concern hierarchy (directed, acyclic graph), thus supporting hierarchical modularity [9].

B. Comparing CORE and CBSE

While the description of CORE presented above shows many similarities with component-based development, one of the fundamental differences is that the unit of reuse in CORE – the CORE concern – is broader than the unit of reuse in CBSE – the component. Similar to SPLs, a CORE concern encapsulates a variety of solutions for a specific domain of interest, and expresses this variability explicitly in a variation interface (VI). The VI comprises a feature model [10], which describes the available functional- and implementation alternatives that the concern encapsulates as well as their dependencies, if any. The feature model expresses the closed variability, i.e., the set of solutions that the designers of the concern have realized. Standard CBSE does not provide a means to group a set of functionally equivalent components together, but extensions have been proposed to express variability within components, e.g., [11], and modules, e.g., [12].

The CORE VI also describes the impacts that the different solutions offered by the concern have on high-level stakeholder goals, system qualities, and non-functional properties using a variant of the Goal-oriented Requirement Language (GRL) [13]. Again, standard CBSE does not address non-functional properties, but extensions have been defined to address specific quality properties [14]. In particular, the Palladio Component Model (PCM) [6] that is discussed in more detail in the next section supports detailed performance, reliability and cost analysis for software architectures.

In CORE, each solution within a concern must define a customization interface (CI) that describes how the solution can be adapted to a specific reuse context. Since each solution is described as generally as possible to increase reusability, some structural and behavioural elements are only partially specified and need to be related or complemented with concrete elements from the reuse context. This enables open variability, similar to what is achieved at the programming level with generic or template classes. Standard components support coarse-grained open variability, as they can specify required interfaces for functionality that must be provided by the reuse context. Application-specific functionality can then be integrated during assembly by connecting the required interface to an application component with a compatible provided interface. Internal customization of components, while provided by extended component models such as Fractal [15] and BlueArX [16], is not supported by standard CBSE. In this case, object-oriented customization techniques, e.g., generics, inheritance and call-backs, can be used for customization purpose.

Last but not least, one important advantage of CORE is that it does not require a concern user to commit to a specific solution variant at the moment of reuse. When reusing a concern, a developer only needs to decide on the reusable functionality that is minimally needed to continue development, and can re-expose relevant alternatives of the reused concern in the reusing concern’s interface. This delays decision making to when more detailed requirements are known and further decisions can be made. To operationalize delayed decision making, CORE relies on additive, aspect-oriented software composition techniques that (re)compose concern realizations whenever additional decisions are made [17].

Table I summarizes the CBSE and CORE comparison.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Standard CBSE</th>
<th>Palladio PCM</th>
<th>CORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface for expressing variability</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Support for expressing impacts</td>
<td>no</td>
<td>performance reliability cost</td>
<td>variation interface feature model</td>
</tr>
<tr>
<td>Interface for customization</td>
<td>coarse grained interface</td>
<td>coarse grained required interface</td>
<td>variation interface customization interface</td>
</tr>
<tr>
<td>Interface for functionality</td>
<td>provided interface</td>
<td>provided interface</td>
<td>usage interface</td>
</tr>
<tr>
<td>Support for delaying decisions</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

C. Integrating CORE and CBSE

The CORE paradigm is general, and in theory any modelling or programming language can be used to describe properties of interest for a concern. Concretely, though, a modelling language that wants to be usable as a realization language within CORE must, for each model type it provides:

1) clearly identify customization and usage interfaces, and
2) offer a homogeneous model composition operator.

Furthermore, if the modelling language is supposed to be used in combination with other modelling languages to describe the realization of a concern following a multi-view approach, then consistency rules have to be specified between the views, and compatibility of the composition operators must be ensured.

IV. PALLADIO INTEGRATION

Palladio [6] is a model-driven approach to CBSE comprised of the Palladio Component Model (PCM), a metamodel for describing component-based software architectures, and the Palladio Bench, an Eclipse-based modelling environment. What sets Palladio aside from other component-based approaches is an extensive set of analysis tools for performance, reliability, and costs evaluation, with includes PerOpteryx, a tool for multi-criteria design space exploration.

This section outlines the steps involved in integrating Palladio with the CORE framework. For each Palladio model type, we discuss what would constitute the customization and usage interface, and explain how we envision the design of a homogeneous composition operator.

A. Structural View: Component Repository and Assembly

Just like other component-based approaches, behaviours (i.e., operations) are modularized structurally in the PCM with signatures, which have a name, parameters and return type. Signatures are logically grouped within interfaces. The
unit of reuse in the PCM is the component. There are three increasingly detailed ways of specifying a component: the Provides Type defines interfaces comprised of the signatures a component offers to be used by other components; the Complete Type extends the Provides Type with the interfaces that a component requires to realize the provided services. The Implementation Type goes further and provides an abstract behavioural description on the implementation of the signatures of a component. Signatures, interfaces and components are specified in the system-independent component repository model.

The assembly model describes the inner structure, i.e., the conceptual architecture, of composite components and systems. In an assembly model, an assembly contexts stands for a component, from which it inherits the provided and required interfaces as defined in the component repository. Assembly connectors link a required interface of one assembly context to a provided interface of another assembly context.

Integration of Structural View with CORE

Structural Customization Interface: As explained in section III, CORE concerns are broader units of reuse than components. It is therefore natural that a concern can encompass multiple components, i.e., an entire, though maybe partial, software architecture. Some of the components encapsulated in the concern will be basic Palladio components, i.e., they have provided and required interfaces, and their complete implementation is also contained inside the concern. However, in order to be able to capture architectural elements that crosscut component boundaries, it should be possible to define partial components, interfaces and signatures that can later be composed with the architecture of the reuse context. These elements comprise the architectural customization interface.

For example, imagine a simple Logging concern. From an architecture point of view, it would be comprised of a standard Logger component that provides a Log interface that would allow any other component in the system to write log entries to a common storage. Having a Logger, though, is not enough. There must also be at least one component that make use of the Logger. Structurally, we don’t know much about the components that use the Logger, apart from the fact that they should require the Log interface, and that they need to be connected to the logger. From an architectural point of view, the Logging concern should therefore also comprise a Logged partial component that must be composed with components in the system in order to augment their structure with the required Log interface and link them to the Logger component in the assembly. The customization interface for the Logging concern therefore consists of the Logged partial component.

Structural Usage Interface: The provided interfaces of Palladio components are the key to executing the component’s functionality. It therefore makes sense to use them as the architectural CORE usage interface. However, as some provided interfaces of components within a concern should never be directly connected to components in the reuse context, only specifically selected provided interfaces should be part of the usage interface.

For example, assume that the Logger component in our example above uses a Database component to store the logs. The provided interface of the Database component should not be part of the usage interface for the Logging concern. Information hiding principles [2] dictate that internal design decisions, in particular those that might change in the future, should not be visible to the outside world to reduce complexity and increase maintainability by avoiding unnecessary dependencies. Hence, we propose that the architectural CORE usage interface of a concern should be constituted of a subset of the provided interfaces of the components encapsulated within the concern.

Structural Composition Operator: We envision the definition of the composition algorithm that combines two structural architecture models to be straightforward. It essentially constitutes a symmetric merge of two repositories and two assembly contexts. The specifics of the merge are highly similar to what is done in the current CORE reference implementation for merging class diagrams. Components need to be treated just like classes, and assembly connectors just like associations. For details, the interested reader is referred to [18] for CORE class diagram composition.

B. Behavioural View: SEFFs and Usage Scenarios

Palladio uses an abstract behavioural description, called service effect specification (SEFF), for describing the internal behaviour of a component at a high level of abstraction. SEFFs model the abstract control flow of the service provided by a signature of a component in terms of internal actions (i.e., resource demands accessing the underlying hardware) and external calls (i.e., accessing functionality by invoking signatures in provided interfaces of connected components). All control flow elements of activity diagrams, e.g., conditional execution, iterative execution, parallel execution and synchronization, are supported. In order to support quality prediction, additional information is attached to SEFFs. For example, for performance prediction, resource demands for internal actions (in terms of CPU instructions to be executed) can be specified. Cost and reliability prediction are supported as well.

In addition to the intra-component behaviour specified using SEFFs, processing rates of hardware nodes can be specified. Furthermore, domain experts can specify the interaction between certain user types and the system under development with a usage model, typically expressed using a variant of activity diagrams. Workload annotations can be supplied for each scenario definition that defines the number of occurrences of a scenario within a certain time period in terms of probability. The combined information from resource demands, processing rates and usage models makes performance prediction of architectures possible.

Since usage scenarios focus on describing user interaction and workload with entire systems, we currently believe that it does not make sense to include usage scenarios in concerns. Concerns are meant to be reused in many systems, each subject to their own usage scenarios. Therefore, the remainder of this subsection focuses solely on integrating SEFFs with CORE.
Integration of Behavioural View with CORE

In the current reference implementation of CORE, the behavioural design of a concern is modelled using a variant of sequence diagrams that can describe control flow at the same level of detail as code, if desired. One single sequence diagram specifies an interaction that can involve many operations provided by multiple objects.

When integrating CORE and Palladio, a component implementor would use CORE sequence diagrams to specify the functionality of the component with an object-oriented design. In this case, SEFFs are simply a more abstract representation of the behaviour of a component that adequately represents the low-level detailed design described using CORE sequence diagrams. In other words, the SEFF view and the sequence diagram view should be consistent. If they are not consistent, then the Palladio quality estimation tools would not be able to accurately predict performance, reliability and cost of a design.

Integration Strategy 1 (IS1): One way of keeping SEFFs and sequence diagrams consistent is to define consistency rules or model checking algorithms that the CORE tool would use to verify consistency periodically. Whenever the consistency check fails, the modeller would be prompted to adjust the models. Additionally, model transformations should be defined to create a SEFF skeleton from an existing CORE sequence diagram, and vice versa. The former transformation would support bottom-up development, since it creates an abstract SEFF representation that is consistent with an already existing design. The latter transformation would support top-down development, since it generates a skeleton design that corresponds to the high-level behavioural description envisioned by a software architect and specified in a SEFF.

Integration Strategy 2 (IS2): Another way to ensure that SEFFs and sequence diagrams are consistent is to combine the information from SEFFs and sequence diagrams into one model. To this aim, we need to identify the information encoded in the SEFF metamodel that is not currently present in the CORE sequence diagram metamodel. The control flow structures in CORE sequence diagrams are at least as expressive as the ones in SEFFs, so they can replace the SEFF control flow structures. Similarly, whether an invocation of an operation of a class is a internal or external action can be determined by checking in the architecture model whether the operation is part of a class that belongs to the same component or not. On the other hand, performance, reliability and cost information, e.g., resource demands and failure rates, are only present in SEFFs. The sequence diagram metamodel would therefore need to be augmented so that the quality information from SEFFs can be attached to the right model elements of the sequence diagram.

SEFF Customization and Usage Interface: We believe that the customization and usage interface of SEFFs, in analogy with what is done for CORE sequence diagrams, should be inherited from the customization and usage interface of the structural view.

SEFF Composition Operator: If SEFFs and sequence diagrams were integrated by combining them into one model (IS2), then the current sequence diagram composition operator can be used as is to also compose the sequence diagrams with additional quality estimation information. This is true because the composition is solely based on the control flow.

If SEFFs are kept separately from the sequence diagrams (IS1), then a separate composition operator for SEFFs needs to be defined. Just like for any other behavioural modelling notation, the order and sequencing of model elements that represent internal and external actions in SEFFs is important. Hence, the composition operator for SEFFs would have to support the composition of behaviour from one SEFF before, after, around or in parallel of some behaviour in the other SEFF. Since CORE is based on additive composition, complete replacement of behaviour does not need to be supported. Instead, a limited form of substitution has to be provided, where the original behaviour is moved to some place within the substituting behaviour. The details of how this can be accomplished with CORE sequence diagrams are presented in [19], and for general sequence diagrams in [20].

When implementing the SEFF composition operator, care must be taken that the composition is compatible with the sequence diagram composition already present in the CORE reference implementation. Otherwise, when composing two concerns one could end up with an output that has inconsistent SEFFs and sequence diagrams, even in the case where the SEFFs and sequence diagrams of each input concern are initially consistent. Since the SEFF control flow structures are a subset of the control flow structures available in sequence diagrams, it should be fairly easy to create a compatible SEFF composition operator by adapting the algorithm of the existing sequence diagram composition operator.

SEFF Integration Strategy Discussion: From the considerations above it seems like it would be easier to choose IS1, i.e., combine sequence diagrams and SEFFs into one model. First, there would be no need to define a new composition operator, since the existing sequence diagram composition operator suffices. Second, concern realization models would be simpler, as there would not be two separate views describing the same behaviour at different levels of abstraction that must be kept consistent.

However, one of the advantages of Palladio is that it enables rapid design exploration. During development, SEFFs can be used to capture intended behaviour of architectural components at a high level of abstraction even before they exist. In doing so, it is possible to predict quality properties of the solution architecture under development before the detailed design of each individual component has been elaborated. If we want to continue to use Palladio for this purpose, then SEFFs have to be supported in CORE independently of CORE sequence diagrams.

C. Deployment View

The Palladio deployment view covers the specification of the execution environment and the allocation of software components on resources of this execution environment, e.g., processors, servers, hard disks, communication links. Resources are annotated with information used for quality prediction, e.g., processing and failure rates. At the current state of
research we believe that it is not necessary to integrate the Palladio deployment view into CORE. The reason for this is that currently, concerns only encapsulate software components.

Integrating the structural and behavioural views of Palladio into CORE as described above allows developers to define reusable software architectures and implementations that can be composed with an application architecture to yield a final system with an architecture that includes the components defined in the reused concern. Developers would then create a deployment view for this final system architecture, i.e., define resources and deploy the components onto them.

Adding deployment models to CORE concerns could make sense if a concern would also encapsulate reusable hardware. In that case, resource specifications for these hardware components should be provided in the concern, and different ways of allocating provided software components and functionality to these resources could be expressed with deployment models.

V. CONCLUSION

This paper presented a road map on how we are planning to integrate two highly complementary modelling approaches, Palladio and CORE. Palladio and the Palladio bench tools enable modellers to specify component-based software architectures and predict their quality by means of abstract behavioural specifications annotated with performance, reliability and cost information. CORE, in particular the current CORE reference implementation, provides facilities for encapsulating a variety of design solutions, and modelling the detailed designs of the solutions with class and sequence diagrams.

By integrating the Palladio architecture model with the CORE reuse mechanisms and low-level design models as described in this paper, reuse of existing architectural solutions is greatly simplified. First, different architectural solutions to the same development issue are easier to find, as they are grouped within a concern and their variability expressed with CORE feature models. Furthermore, the Palladio quality estimation tools for performance, reliability and cost, and the CORE impact models for any other relevant non-functional qualities enable informed decision making and even tradeoff analysis between the proposed solutions. Once a specific solution is chosen, the CORE customization interface clearly designates the architectural and design elements of the solution that need to be connected to application-specific elements. Once the developer specifies the mapping, the aspect-oriented composition operators of CORE take care of composing the chosen concern with the application. This includes adding interfaces to application components and linking them to components provided by the reused solution, and modifying the internal application design to invoke the services provided by the solution at the appropriate places. Finally, thanks to CORE’s support for delayed decision making, the developers of reusable architectural concerns can internally reuse other architectural and design concerns without having to make premature or default decisions regarding non-functional implementation alternatives. Ultimately this will increase reusability of the solutions, as the commitment to a specific implementation can be made once the requirements of the reuse context are known.

While the approach proposed in this paper has multiple advantages, it is also rather heavyweight, mainly because MDE and/or aspect-oriented programming has to be used for the detailed design of all solutions encapsulated within a concern. If MDE technology is going to be used exclusively at the architecture level for design exploration and quality prediction purpose, a more lightweight approach consisting of introducing a new concern-like grouping unit into the Palladio component model could be envisioned as outlined in [21].

REFERENCES

Springer, June 2016, pp. 122–137.
ACM, 2016.
Performance Modeling in the Age of Big Data
Some Reflections on Current Limitations

Robert Heinrich
Software Design and Quality
Karlsruhe Institute of Technology
Karlsruhe, Germany
robert.heinrich@kit.edu

Holger Eichelberger, Klaus Schmid
Software Systems Engineering
University of Hildesheim
Hildesheim, Germany
{eichelberger,schmid}@sse.uni-hildesheim.de

Abstract—Big Data aims at the efficient processing of massive amounts of data. Performance modeling is often used to optimize performance of systems under development. Based on experiences from modeling Big Data solutions, we describe some problems in applying performance modeling and discuss potential solution approaches.

Index Terms—Performance, modeling, Big Data, Palladio

I. MOTIVATION

Big Data solutions store, transfer, and process huge data sets. Modeling the architecture of Big Data systems in a component-based fashion and using architectural models for conducting performance analysis is essential to compare design alternatives. As Big Data solutions are typically realized as distributed systems, performance analysis can help to determine the required amount of resources and the distribution of analysis tasks over a cluster prior to implementation.

An established research area at the intersection of model-driven and component-based software engineering is using component models to conduct performance simulation and prediction. There are numerous approaches with varying focus in this area [3]. The key research that led to this paper was driven from experience with Palladio [4], however, we believe that most of our points hold for the broader field of model-based performance prediction of component-based systems.

In our research, we aim to apply model-based performance prediction to Big Data systems. Such systems typically use established infrastructures like Storm, Spark, etc. Our current focus is on performance models for near real-time computation for financial data streams [7]. As a consequence, our discussion of the approaches and their limitations is driven from this background. However, we believe that several of our points actually apply to a larger range of situations. As part of our discussion, we will make explicit which problems are particular to Big Data and which ones probably apply to a wider range.

The structure of the paper is: In Section II we describe the identified problems. This is the core of this contribution, as this is to our knowledge the first time these issues are made explicit. In Section III we discuss some initial ideas on how to address these issues. Finally, in Section IV we conclude.

II. IDENTIFIED PROBLEMS

Our analysis was mainly driven from attempts to model the performance of Big Data systems for achieving adaptive behavior. Our modeling studies were performed with Palladio.

Hence, in terms of experiences the points below must be regarded as Palladio-specific. However, as we will discuss, the points also apply to a broader range of modeling approaches and (partially) even beyond Big Data systems. First limitations of Palladio for modeling Big Data systems have been described in [7]. Our analysis goes beyond these limitations and is structured into 3+1 points: the first three issues refer to potential problems in modeling characteristics of Big Data systems, while the last point is of a more fundamental nature.

Flexible Component Distribution: In Big Data systems, it is often necessary to modify the distribution of worker implementations across different resources, e.g., switching from two servers responsible for a certain kind of processing to three servers hosting the same logical component. This can also be combined with adding servers (e.g., automatic scale-out). Such behavior is not specific to Big Data systems, as even web shops like Amazon do this. These distribution changes are triggered by changes in the workload of the system.

Today capabilities for describing arbitrarily complex dynamic component-resource binding do not exist in performance modeling and analysis approaches like Palladio. However, approaches like SimuLizar [1] and iObserve [2] are able to support some specific adaptations. SimuLizar considers deployment changes during analysis at design-time. iObserve focuses on reflecting observed changes in the running system in deployment like migration and (de-)replication.

Data-oriented Load Distribution: Big Data is centered on data processing and the data itself is used to control the applications, i.e., the processing component is selected based on the type of data (data-flow processing). In contrast, existing performance modeling approaches focus on the call-relation among components and do not consider data as first class entities. Thus, they do not provide capabilities for modeling data streams. Based on our experience, this makes it very hard to model Big Data applications. However, beyond ease and adequacy of modeling, it also leads to situations were specific, performance-relevant aspects cannot be modeled, e.g., if the amount of data stays the same over time, but the composition of data types change, this may lead to adaptations. It seems these issues are particularly relevant to Big Data applications.

Explicit Queuing Components: Big Data applications utilize queuing components for various purposes, but in particular, to organize the distribution of data across the application and to smooth peak loads. Thus, queuing has a very significant performance impact. The precise impact depends on specific
aspects of the queuing. Any performance models that omit such aspects are significantly insufficient, but current component-based modeling approaches like Palladio do not have modeling capabilities for queuing components. (Queuing exists, but is restricted to modeling resource usage). Thus, it is impossible to create adequate models of this aspect of system behavior. As internal queuing exists in other systems as well, we assume that this will be a problem for modeling these types of systems, too.

The previous three points describe three dimensions of modeling capabilities that are not – or not sufficiently – supported by existing performance modeling approaches like Palladio. However, there exists a broader and more fundamental problem to which we will turn now:

**Blackbox Infrastructure:** In Big Data applications, technologies like Storm, Spark, Hadoop, etc. play a central role, however, these are very large and complex infrastructures. There do not exist any models of their behavior and because of their size alone it would be a daunting task to construct one. The situation that large, unknown infrastructures are part of the models is not new. But, experience shows that for traditional systems, despite abstracting from classical infrastructures it was possible to get very adequate results [4]. This is different for Big Data, as the infrastructure operations may have a strong impact on system performance. This leads to the fundamental problem of how to derive models (at least of critical aspects) of such large infrastructures? Manual model construction seems out of scope, as the modeling alone would be often many times more complex than modeling the core application. Moreover, it would entail a significant reengineering project.

III. SOLUTION IDEAS

Based on the problems identified in Section II we will now discuss some potential solution approaches.

**Flexible Component Distribution:** Performance analysis of systems with flexible component distribution requires the inclusion of adequate modeling primitives to support this. An approach that already heads in this direction is SimuLizar [1]. It supports the modeling of self-adaptation rules, e.g. for load balancing. However, the expressiveness of its rules is not sufficient to support all relevant distribution adaptations.

In order to improve the capabilities of performance modeling approaches, we assume it is necessary to significantly enhance the capabilities for describing runtime adaptation rules and further enhance analysis approaches so the adaptation effects are taken into account in the analysis.

**Data-oriented Load Distribution:** As discussed earlier, the key issue is that the modeling of component-oriented approaches relies on call-relationships, while in Big Data systems, the key relation is the data flow. Hence, we assume a natural, but necessary extension, will be to extend the modeling approaches with explicit data-flow modeling primitives.

However, we are not the first to propose this. Seifermann proposed such an extension for the Palladio approach [6]. His motivation was completely different, i.e., it focused on analyzing systems for privacy or SLAs violations. We imagine that an appropriate extension for dataflow modeling could actually simultaneously serve both purposes.

**Explicit Queuing Components:** To the best of our knowledge queuing components are not yet considered as predefined model elements on architecture level in existing approaches to architecture analysis. They may be constructed manually using formalisms like Layered Queuing Networks (LQNs) and Queuing Petri Nets (QPNs). However, to support an integrated performance analysis in a component-based paradigm, it would be necessary to integrate these capabilities into the underlying component models. We regard this as a difficult, but mandatory challenge for the performance analysis of Big Data systems.

**Blackbox Infrastructure:** Even if the above extensions would be sufficient to model the characteristics of complex Big Data systems, the problem would remain that the underlying infrastructures would need to be modeled as well. Given existing performance-oriented reengineering approaches, this would require significant reengineering efforts that seem rather unrealistic. Hence, the challenge here is: how can we (semi-) automatically derive sufficient model information from such infrastructures? If these infrastructures would be once comprehensively modeled, these models could be reused as a single component or as a parameterizable pattern.

IV. CONCLUSION

In this paper, we provided an initial discussion of current shortcomings in model-driven performance engineering. While it was based on modeling Big Data systems with Palladio, we believe the experiences hold at least for the broader range of performance modeling of big data systems and some of them may hold for a much wider range of cases were complex off-the-shelf infrastructures are used in system development.

It is the goal of our ongoing research to address the identified shortcomings by augmenting modeling approaches and providing novel methods for model construction.

V. REFERENCES

Organizers

Federico Ciccozzi  Mälardalen University (Sweden)
Ivano Malavolta  Vrije University (Netherlands)

Program Committee

Marco Autili  University of L’Aquila (Italy)
Jan Carlson  Mälardalen University (Sweden)
Antonio Cicchetti  Mälardalen University (Sweden)
Peter Clarke  Florida International University (USA)
Ivica Crnkovic  Mälardalen University (Sweden)
Guglielmo De Angelis  CNR IASI/ISTI (Italy)
Juergen Dingel  Queen’s University (Canada)
David Garlan  Carnegie Mellon University (USA)
Sebastián Gerard  CEA List (France)
Jeff Gray  University of Alabama (USA)
Anne Koziolak  Karlsruhe Institute of Technology (Germany)
Patricia López Martinez  University of Cantabria (Spain)
Tanja Mayerhofer  TU Wien (Austria)
Julio Luis Medina  University of Cantabria (Spain)
Patrizio Pelliccione  Chalmers - GU (Sweden)
Dorina Petriu  Carleton University (Canada)
Alfonso Pierantonio  University of L’Aquila (Italy)
Pascal Poizat  Universit Paris Ouest Nanterre la Defense (France)
Ansgar Radermacher  CEA List (France)
Mehrdad Saadatmand  SICS (Sweden)
Lionel Seinturier  University of Lille/INRIA (France)
Bran Selic  Malina Software Corp. (Canada)
Eugene Syriani  University of Montreal (Canada)
Massimo Tivoli  University of L’Aquila (Italy)
Antonio Vallecillo  University of Malaga (Spain)
Tullio Vardanega  University of Padua (Italy)