6th Workshop on Models@run.time at MODELS 2011

Wellington, New Zealand, October 17th 2011

Proceedings

Editors

Nelly Bencomo
Gordon Blair
Betty Cheng
Robert France
Cédric Jeanneret

Organization Committee

Nelly Bencomo (Program Chair)
INRIA Paris, France

Gordon Blair (Program Chair)
Lancaster University, UK

Betty Cheng
Michigan State University, USA

Robert France
Colorado State University, USA

Cédric Jeanneret (Program Chair)
University of Zurich, Switzerland

Program Committee

Uwe Assman
Dresden, Germany

Peter J. Clarke
Florida International University, USA

Franck Fleurey
SINTEF, Norway

Jeff Gray
University of Alabama, USA

Paola Inverardi
Università dell'Aquila, Italy

Brice Morin
SINTEF, Norway

Rui Silva Moreira
Universidade Fernando Pessoa, Portugal

Mario Trapp
Frauenhofer IESE, Germany

Liliane Pasquale
LERO, Ireland

Franck Chauvel
Peking University, China

Fabio M. Costa
Federal University of Goias, Brazil

Holger Giese
Universität Postdam, Germany

Gang Huang
Peking University, China

Jean-Marc Jezequel
IRISA, France

Hausi Muller
University of Victoria, Canada

Arnor Solberg
SINTEF, Norway

Thaís Vasconcelos Batista
Federal University of Rio Grande do Norte, Brazil

Additional Reviewers

Basil Becker
Universität Postdam, Germany

Preface

Welcome to the 6th Workshop on Models@run.time at MODELS 2011!

This document contains the proceedings of the 6th Workshop on Models@run.time that will be co-located with the ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (MODELS). The workshop will take place in Wellington, New Zealand, on the 17th of October 2011. The workshop is organized by Nelly Bencomo, Gordon Blair, Betty Cheng, Robert France and Cédric Jeanneret.

From a total of 8 papers submitted 6 full papers were accepted. This volume gathers together all the 6 papers accepted at Models@run.time 2011. After the workshop, a summary of the workshop will be published to complement these proceedings.

We would like to thank a number of people who contributed to this event, especially the members of the program committee and additional reviewers who provided valuable feedback to the authors. We also thank to the authors for their submitted papers, making this workshop possible.

We are looking forward to having fruitful discussions at the workshop!

October 2011

Nelly Bencomo
Gordon Blair
Betty Cheng
Robert France
Cédric Jeanneret

Content

Session 1

Session 2

Language and Framework Requirements for Adaptation Models

Thomas Vogel and Holger Giese

Hasso Plattner Institute at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany
{thomas.vogel|holger.giese}@hpi.uni-potsdam.de

Abstract. Approaches to self-adaptive software systems use models at runtime to leverage benefits of model-driven engineering (MDE) for providing views on running systems and for engineering feedback loops. Most of these approaches focus on causally connecting runtime models and running systems, and just apply typical MDE techniques, like model transformation, or well-known techniques, like event-condition-action rules, from other fields than MDE to realize a feedback loop. However, elaborating requirements for feedback loop activities for the specific case of runtime models is rather neglected.
Therefore, we investigate requirements for *Adaptation Models* that specify the analysis, decision-making, and planning of adaptation as part of a feedback loop. In particular, we consider requirements for a modeling language of adaptation models, and for a framework as the execution environment of adaptation models. Moreover, we discuss patterns for using adaptation models within the feedback loop regarding the structuring of loop activities. The patterns and the requirements for adaptation models influence each other, which impacts the design of the feedback loop.

1 Introduction

Self-adaptation capabilities are often required for modern software systems to dynamically change the configuration in response to changing environments or goals [5]. *Models@run.time* are a promising approach for self-adaptive software systems since models may provide appropriate abstractions of a running system and its environment, and benefits of model-driven engineering (MDE) are leveraged to the runtime phases of software systems [3].

Most models@run.time efforts to self-adaptive software systems focus on causally connecting models to running systems, and just apply typical or well-known techniques from MDE or other fields on top of these models. These techniques are used for engineering a feedback loop that controls self-adaptation by means of *monitoring* and *analyzing* the running system and its environment, and the *planning* and *execution* of changes to the running system [13].

For example, the causal connection has been a topic for discussions at the last two workshops on models@run.time [1, 2], or the work of [17] particularly addresses the causal connection, and it just applies MDE techniques, like model transformation, on top to show their technical feasibility. We proposed an approach to use incremental model synchronization techniques to maintain mul-

tiple, causally connected runtime models at different abstraction levels, and thereby, we support the monitoring and the execution of adaptations [18, 19].

While causal connections provide basic support for monitoring and for executing changes, they do not cover the analysis and planning steps of a feedback loop, which decide *if* and *how* the system should be adapted. For these steps, techniques originating from other fields than MDE are used. Most approaches [4, 7, 8, 11, 12, 14] employ rule-based mechanisms in some form of event-condition-action rules that exactly specify when and how adaptation should be performed, and thus, the designated target configuration is predefined. In contrast, search-based techniques just prescribe goals that the system should achieve. Triggered by conditions or events and guided by utility functions they try to find the best or at least a suitable target configuration fulfilling these goals [10, 15].

All these approaches focus on applying such decision-making techniques for the analysis and planning steps, but they do not systematically investigate the requirements for such techniques in conjunction with models@run.time. Eliciting these requirements might help in engineering new or tailored decision-making techniques for the special case of models@run.time approaches to self-adaptive systems. Therefore, we elaborate requirements for such techniques by taking an MDE perspective. The techniques should be specified by models, which we named *Adaptation Models* in an attempt to categorize runtime models [20]. However, the categorization does not cover any requirements for runtime models.

In this paper, we discuss requirements for adaptation models, and in particular requirements for languages to create such models and for frameworks that employ and execute such models within a feedback loop. By language we mean a broad view on metamodels, constraints, and model operations, which are all used to create and apply adaptation models. Moreover, we discuss patterns for using adaptation models within the feedback loop. The patterns and the requirements for adaptation models influence each other, which impacts the design of the feedback loop by providing alternatives for structuring loop activities.

The rest of the paper is structured as follows. Section 2 discusses related work, and Section 3 sketches the role of adaptation models in self-adaptive systems. Section 4 discusses the requirements for adaptation models, while Section 5 presents different patterns of employing adaptation models within a feedback loop. Finally, the paper concludes and outlines future work in Section 6.

2 Related Work

As already mentioned in the previous section, most models@run.time approaches to self-adaptive software systems focus on applying techniques for decision-making and do not systematically elaborate on their requirements [4, 7–12, 14, 15]. A few approaches merely consider the requirement of performance and efficiency for their adaptation mechanisms to show and evaluate the applicability at runtime [10, 11, 15]. Likewise, several decision-making mechanisms are presented in [16] that primarily discusses their specifics for mobile applications in ubiquitous computing environments by means of performance and scalability regarding the size of the managed system and its configuration space. In general,

rule-based mechanisms are considered as efficient since they exactly prescribe the whole adaptation, while for search-based approaches performance is critical and often improved by applying heuristics or by reducing the configuration space.

This is also recognized by [9] that attests efficiency and support for early validation as benefits for rule-based approaches. However, they suffer from scalability issues regarding the management and validation of large sets of rules. In contrast, search-based approaches may cope with these scalability issues, but they are not as efficient as rule-based approaches and they provide less support for validation. As a consequence, a combination of rule-based and search-based techniques is proposed in [9] to balance their benefits and drawbacks.

To sum up, if requirements or characteristics of decision-making techniques are discussed, these discussions are limited to performance, scalability, and support for validation, and they are not done systematically. One exception is the work of Cheng [6] who discusses requirements for a self-adaptation language that is focused on specifying typical system administration tasks. However, the requirements do not generally consider self-adaptive software systems and they do not address specifics of models at runtime. Nevertheless, some of the requirements that are described in this paper are derived from this work.

3 Adaptation Models

Before discussing requirements for adaptation models, we sketch the role of these models based on a conceptual view on a feedback loop as depicted in Figure 1.

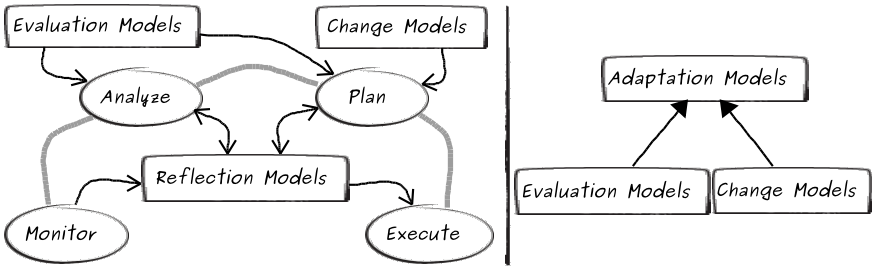


Fig. 1. Feedback Loop and Runtime Models (cf. [20])

The steps of monitoring and analyzing the system and its environment, and the planning and execution of changes are derived from the autonomic computing element [13], while we discussed the different models and a usage scenario of models in the loop in [20]. *Reflection Models* describe the running system and its environment and they are causally connected to the system. According to observations of the system and environment, the monitor updates the reflection models. Reasoning on these models is done by the analyze step to decide whether the system fulfills its goals or not, and thus, whether adaptation is required or not. The reasoning is specified by *Evaluation Models*, which can be constraints that are checked on reflection models. If adaptation is required, the planning step devises a plan defining how the system should be adapted, which is guided

by *Change Models* to explore the system's variability or configuration space. Deciding on the designated target configuration is guided by evaluation models to analyze different adaptation options, and the selected option is applied on reflection models. Finally, the execute step involved in the causal connection performs the adaptations on the running system to move it to the target configuration.

By *Adaptation Models* we generally consider evaluation and change models regardless of the concrete rule-based or search-based techniques that are employed for the analysis and planning steps, and thus, for the decision-making. This view on adaptation models is similar to [14], which just presents one adaptation model for the specific approach, but no general discussion of such models.

4 Requirements for Adaptation Models

In this section we describe requirements for adaptation models to be used in self-adaptive software systems to analyze and decide on adaptation needs, and to plan and decide on how to adapt the running system. We assume that the self-adaptive system employs runtime models, which influences the requirements for adaptation models. At first, we discuss requirements for a modeling language that is used to create adaptation models. Then, we elaborate the requirements for a framework as the execution environment for adaptation models. Being in the early requirements phase, we take a broad MDE view on the notion of languages as combinations of metamodels, constraints, and model operations, which are all used to create and apply adaptation models.

Likewise to the common understanding that requirements for real-world applications cannot be completely and definitely specified at the beginning of a software project, we think that the same is true for the requirements discussed here. It is likely that some of these requirements may change, become irrelevant, or new ones emerge when engineering concrete adaptation models for a specific self-adaptive system and domain. Thus, we do not claim that the requirements are complete and finalized with respect to their enumeration and definitions.

4.1 Language Requirements for Adaptation Models

Language requirements (LR) for adaptation models can be divided into functional and non-functional ones. Functional requirements target the concepts that are either part of adaptation models or that are referenced by adaptation models. These concepts are needed for the analysis, decision-making, and planning. Thus, functional requirements determine the expressiveness of the language. In contrast, non-functional language requirements determine the quality of adaptation models. At first functional, then non-functional requirements are discussed.

Functional Language Requirements

LR-1 ***Functional Specification/Goals***: Enabling a self-adaptive system to continuously provide the desired functionality to users or other systems, adaptation models have to know about the current functional specification or goals of the system. The functional specification or goals define *what* the system should

do, and this information needs to be available in an operationalized form to relate it with the actual behavior of the running system. This is the foundation for adapting the functional behavior of the system.

LR-2 ***Quality Dimensions***: While LR-1 considers *what* the system should do, quality dimensions address *how* the system should provide the functionality in terms of quality of service (QoS). To support QoS-aware adaptations, quality dimensions, like performance or security, must be characterized by adaptation models (cf. [6]).

LR-3 ***Preferences***: Since multiple quality dimensions (LR-2) may be relevant for the managed system, preferences across the dimensions must be expressed to trade-off and balance competing qualities (cf. [6]). Likewise, preferences for goals (LR-1) are necessary if several valid behavioral alternatives are feasible and not distinguished by the quality dimensions.

Thus, the language for adaptation models must incorporate the concepts of goals (LR-1), quality dimensions (LR-2), and preferences (LR-3) in an operationalized form, such that they can be referenced or described and automatically processed by concrete adaptation models. This operationalized form should be derived from the requirements of the self-adaptive system. Goals, quality dimensions, and preferences serve as references for the running system as they state what the system should do and how it should be.

LR-4 ***Access to Reflection Models***: Adaptation models must reference and access reflection models to obtain information about the current situation of the running system and its environment for analysis, and to change the reflection models to effect adaptations. Thus, a language for adaptation models must be based on the languages of reflection models.

LR-5 ***Events***: Adaptation models should reference information from events emitted by the monitor step when updating the reflection models due to runtime phenomena of the system. Besides serving as a trigger for starting the decision-making process, events support locating the phenomena in the system and reflection models (LR-4). Thus, evaluating the system and its environment (LR-6) may start right from the point in the reflection models where the phenomena have occurred. Events provided by the monitor step and signaling changes in the running system support reactive adaptation, while the decision-making process for proactive adaptations can be triggered periodically.

LR-6 ***Evaluation Conditions***: A language for adaptation models must support the specification of conditions to evaluate the running system and its environment (cf. [6]). These conditions relate the goals (LR-1), quality dimensions (LR-2), and preferences (LR-3) to the actual running system represented by reflection models (LR-4). Therefore, conditions may refer to events notifying about runtime phenomena (LR-5) as a starting point for evaluation, and they should be able to capture complex structural patterns for evaluating the software architecture of the running system.

LR-7 ***Evaluation Results***: Adaptation models must capture the results of computing the evaluation conditions (LR-6), because these results identify and decide on adaptation needs especially when the conditions are not met by the

system. Adaptation models may annotate and reference the evaluation results in reflection models (LR-4) to locate adaptation needs in the running system.

LR-8 ***Adaptation Options***: Adaptation models must capture the variability of the system to know the options for adaptation. These options define the configuration space for the system and how reflection models (LR-4) can be modified to adapt the running system.

LR-9 ***Adaptation Conditions***: Adaptation models must consider adaptation conditions since not all adaptation options (LR-8) are feasible in every situation. Thus, conditions should constrain all adaptation options to applicable ones for certain situations (cf. [6]). To characterize a situation for an adaptation option, conditions should refer to reflection models (LR-4), events (LR-5), evaluation results (LR-7), or other adaptation options. Likewise to such pre-conditions for adaptation options, post-conditions and invariants should be considered.

LR-10 ***Adaptation Costs and Benefits***: Adaptation models should characterize costs and benefits of adaptation options (LR-8) as a basis to select among several possible options in certain situation (cf. [6]). Costs should indicate that adaptations are not for free, and benefits should describe the expected effects of options on the goals (LR-1) and quality dimensions (LR-2) of the system. By relating costs and benefits to the preferences of the system (LR-3), suitable adaptation options should be selected and applied on the reflection models.

LR-11 ***History of Decisions***: Adaptation models should capture history of decisions, like evaluation results (LR-7) or applied adaptation options (LR-8) to enable learning mechanisms for improving future decisions.

Non-functional Language Requirements

LR-12 ***Modularity, Abstractions and Scalability***: An adaptation model should be a composition of several submodels rather than a monolithic model to cover all concepts for decision-making. For example, evaluation conditions (LR-6) and adaptation options (LR-8) need to be part of the same submodel, and even different adaptation options can be specified in different submodels. Thus, the language should support modular adaptation models. Moreover, the language should enable the modeling at different abstraction levels for two reasons. First, the level depends on the abstraction levels of the employed reflection models (LR-4), and second, lower level adaptation model concepts should be encapsulated and lifted to appropriate higher levels. For example, several simple adaptation options (LR-8) should be composable to complex adaptation options. Language support for modularity and different abstractions promote scalability of adaptation models.

LR-13 ***Side Effects***: The language should clearly distinguish between concepts that cause side effects on the running system and those that do not. For example, computing an evaluation condition (LR-6) should not affect the running system, while applying an adaptation option (LR-8) finally should. Making the concepts causing side effects explicit is relevant for consistency issues (FR-1).

LR-14 ***Parameters***: The language should provide constructs to parameterize adaptation models. Parameters can be used to adjust adaptation models at runtime, like changing the preferences (LR-3) according to varying user needs.

LR-15 ***Formality***: The language should have a degree of formality that enables online and offline validation or verification of adaptation models, e.g., to detect conflicts or thrashing effects in the adaptation mechanisms.

LR-16 ***Reusability***: The core concepts of the language for adaptation models should be independent of the languages used for reflection models in an approach. This leverages the reusability of the language and adaptation models.

LR-17 ***Ease of Use***: The design of the language should consider its ease of use, because adaptation models are created by software engineers. This influences, among others, the modeling paradigm, the notation, and the tool support. Preferably the language should be based on a declarative modeling paradigm, which is often more convenient and less error-prone than an imperative one. Imperative constructs should be deliberately used in the language. Likewise, appropriate notations and tools are required to support an engineer in creating, validating or verifying adaptation models.

4.2 Framework Requirements for Adaptation Models

In the following we describe framework requirements (FR) for adaptation models. By framework we consider the execution environment of adaptation models, which determines how adaptation models are employed and executed in the feedback loop. Thus, only requirements specific for such a framework are discussed. Typical non-functional requirements for software systems, like reliability or security, are also relevant for adaptation mechanisms, but they are left here.

FR-1 ***Consistency***: The execution or application of adaptation models should preserve the consistency of reflection models and thus, the consistency of the running system. For example, when adapting a causally connected reflection model, the corresponding set of model changes should be performed atomically and correctly. Thus, the framework should evaluate the invariants, pre- and post-conditions (LR-9) for adaptation options (LR-8) at the model level, before adaptations are executed to the running system.

FR-2 ***Incrementality***: The framework should leverage incremental techniques to apply or execute adaptation models to promote efficiency. For example, events (LR-5) or evaluation results (LR-7) annotated to reflection models should be used to directly locate starting points for evaluation or adaptation planning, respectively. Or, adaptation options (LR-8) should be incrementally applied on original reflection models rather than on copies. Incrementality could avoid costly operations, like copying or searching potentially large models.

FR-3 ***Reversibility***: Supporting incremental operations on models (FR-2), the framework should provide the ability to incrementally reverse performed operations. For example, the configuration space has to be explored for adaptation planning by creating a path of adaptation options (LR-8) applied on reflection models. Finding a suitable path might require to turn around and to try alternative directions without completely rejecting the whole path. Thus, *do* and *undo* of operations leverages, among others, incremental planning of adaptation.

FR-4 ***Priorities***: The framework should utilize priorities to organize modular adaptation models (LR-12) to efficiently and easily identify first entry points

for executing or applying adaptation models. For example, priorities can be assigned to different evaluation conditions (LR-6) based on their criticality, and the framework should check the conditions in decreasing order of their criticality.

FR-5 ***Time Scales***: The framework should simultaneously support different time scales of analysis and adaptation planning. For example, in known and mission-critical situations quick and precisely specified reactions might be necessary (cf. rule-based techniques), while in other situations comprehensive and sophisticated reasoning and planning are feasible (cf. search-based techniques).

FR-6 ***Flexibility***: The framework should be flexible by allowing adaptation models to be added, removed and modified at runtime. This supports including learning effects, and it considers the fact that all conceivable adaptation scenarios could not be anticipated at development-time. Moreover, it is a prerequisite of hierarchical control where the adaptation mechanisms as specified by adaptation models are managed by another, higher level control loop [13, 20].

Using these language and framework requirements for adaptation models, we investigate their dependencies on different patterns or designs of feedback loops.

5 Feedback Loop Patterns for Adaptation Models

In the following we discuss feedback loop patterns for adaptation models and how the functional language requirements (cf. Section 4.1) map to these patterns, while considering the framework requirements (cf. Section 4.2). The non-functional language requirements are not further addressed here because they are primarily relevant for designing a language for adaptation models and not for actually applying such models. The patterns differ in the coupling of the analysis and planning steps of a feedback loop, which influences the requirements for adaptation models. Moreover, the adaptation model requirements likely impact the patterns and designs of the loop. Thus, this section provides a preliminary basis for investigating dependencies between requirements and loop patterns.

5.1 Analysis and Planning – Decoupled

The first pattern of the feedback loop depicted in Figure 2 decouples the analysis and planning steps as originally proposed (cf. Section 3). The figure highlights functional language requirements (LR) at points where the concepts of the corresponding requirements are relevant. This does not mean that adaptation models must cover all these points, but they must know about the concepts.

In response to events notifying about changes in the running system or environment, the monitor updates the reflection models and annotates the events (LR-5) to these models. The analyze step uses these events to locate the changes in the reflection models and to start reasoning at these locations. Reasoning is specified by *evaluation models* defining evaluation conditions (LR-6) that relate the goals (LR-1), qualities (LR-2), and preferences (LR-3) to the characteristics of the running system. These characteristics are obtained by accessing reflection models (LR-4). Analysis is performed by evaluating the conditions and probably enhanced by consulting past analyses (LR-11). This produces analysis results (LR-7) that are annotated to the reflection models to indicate adaptation needs.

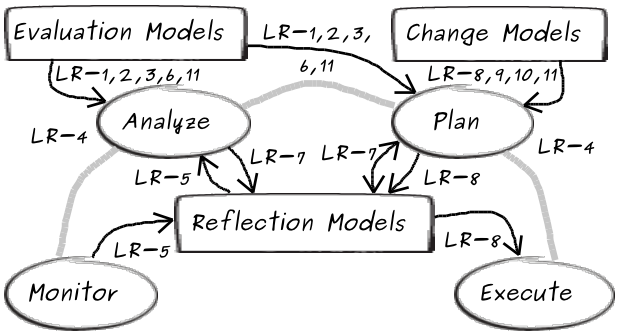


Fig. 2. Decoupled Analysis and Planning Steps

The planning step uses these results (LR-7) attached to reflection models (LR-4) to devise a plan for adaptation. Planning is based on *change models* specifying adaptation options (LR-8) and their conditions (LR-9), costs and benefits (LR-10). This information and probably plans devised in the past (LR-11) are used to find suitable adaptation options to create potential target configurations by applying these options on reflection models. These reflection models prescribing alternative target configurations are analyzed by applying evaluation models to select the best configuration among them. In contrast to the analyze step that uses evaluation models to reason about the current configuration (descriptive reflection models), the planning step uses them to analyze potential target configurations (prescriptive reflection models). Finally, the selected adaptation options (LR-8) are effected to the running system by the execute step.

This pattern is similar to the generic idea of search-based approaches, since planning is done by exploring adaptation options (LR-8, 9, 10) that are evaluated (LR-6, 7, 11) for their fitness for the preferenced system goals (LR-1, 2, 3) based on the current situations of the system and environment (LR-4). Explicitly covering all language requirements for adaptation models, this pattern rather targets comprehensive and sophisticated analysis and planning steps working at longer time scales (FR-5), while efficiency concerns could be tackled by incrementality.

This pattern leverages incrementality (FR-2) since the coordination between different steps of the loop is based on events, analysis results, and applied adaptation options, which directly point to location in reflection models for starting analysis, planning, or executing changes. Moreover, analysis and planning steps may incrementally interleave. Based on first analysis results that are produced by evaluation conditions with highest priorities (FR-4), a planning process might start before the whole system and environment have been completely analyzed. However, incrementality requires the reversibility of performed operations (FR-3) to ensure consistency of reflection models (FR-1), e.g., when alternative adaptation options are tested online on reflection models and finally discarded.

In our categorization of runtime models, we distinguished two kinds of adaptation models based on the feedback loop steps: evaluation models for the analyze step, and change models for the planning step [20]. This distinction is backed by the different language requirements each of these kinds of models are addressing.

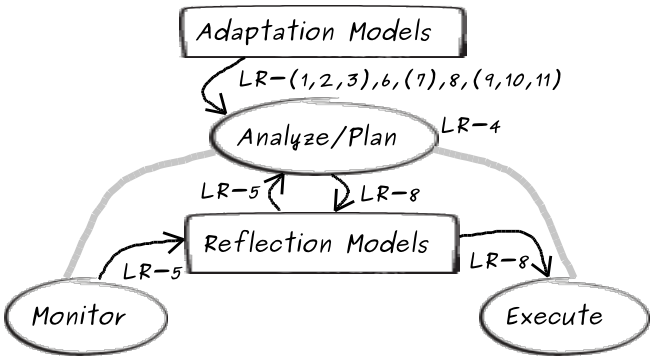


Fig. 3. Coupled Analysis and Planning Steps

5.2 Analysis and Planning – Coupled

In contrast to decoupling the analyze and planning steps, they can be closely integrated into one step, which is sketched in Figure 3. Based on events (LR-5) the integrated analyze/plan step computes evaluation conditions (LR-6) that are directly mapped to adaptation options (LR-8). If a condition is met, the corresponding adaptation options are applied on the reflection models and finally executed to the running system. Access to reflection models (LR-4) is realized by the analyze/plan step as a link between adaptation and reflection models.

In Figure 3, the language requirements written in brackets are not explicitly covered by adaptation models, because this pattern precisely specifies the adaptation mechanism by directly relating evaluation conditions to the application of adaptation options. Thus, this relation or mapping implicitly covers some of the language requirements listed in brackets. For example, it is assumed that the applied adaptation options modify the running system's configuration in a way that fulfills the desired goals, qualities and preferences (LR-1, 2, 3).

Considering the events and the mapping of evaluation conditions to adaptation options, this pattern is similar to rule-based approaches using event-conditions-action rules. Likewise to such rules covering the whole decision-making process, and due to the integration of analysis and planning into one step, the clear distinction between evaluation and change models is blurred. Therefore, both kinds of models are combined to adaptation models in Figure 3.

Thus, this pattern targets adaptation mechanisms requiring quick reactions to runtime phenomena by enabling adaptation at rather short time scales (FR-5). Moreover, efficiency is improved by incrementality (FR-2) and priorities (FR-4). The steps may incrementally coordinate each other through locating events and applied adaptation options in reflection models in order to incrementally evaluate conditions and execute adaptation options to the running system. Priorities may be used to order evaluation conditions for quickly identifying critical situations that need urgent reactions, while conditions for non-critical situations can be evaluated without strict time constraints.

The framework requirement of consistency (FR-1) is not explicitly covered, since it is assumed that the mapping of condition to adaptation options preserves

consistency by design of such rule-based mechanisms. Since these mechanisms strictly prescribe the adaptation, there need not to be any options left that have to be decided at runtime. This reduces the need for reversible operations (FR-3).

5.3 Discussion

Regarding the two different feedback loop patterns and their effects on adaptation models, we can make two observations. First, it might be necessary to combine both patterns in a self-adaptive system if simultaneous support for different time scales (FR-5) is required, or if the nature of a self-adaptive system requires both flavors of rule-based and search-based decision-making mechanisms. Second, we think that these two patterns span a range of several other patterns. By explicitly covering more and more language requirements, the adaptation models and thus, the adaptation mechanisms get more elaborate, and we may move stepwise from the coupled pattern (cf. Section 5.2) toward the decoupled one (cf. Section 5.1). Which pattern and adaptation models suit best depends on the concrete self-adaptive system, especially on the system's domain requirements.

Finally, the requirement of flexibility (FR-6) has not been discussed for the two patterns. However, it is relevant for both of them since it is usually not possible to anticipate all adaptation scenarios at development-time. Thus, changing adaptation models at runtime is required to adjust the adaptation mechanisms.

6 Conclusion and Future Work

In this paper we have elaborated the requirements for adaptation models that specify the decision-making process in self-adaptive software systems using models@run.time. In particular, requirements for a modeling language incorporating metamodels, constraints, and model operations for creating and applying adaptation models have been discussed, as well as requirements for a framework that executes adaptation models. Moreover, we discussed patterns of a self-adaptive system's feedback loop with respect to the requirements for adaptation models.

As future work, we plan to analyze existing approaches to self-adaptation regarding their fitness to the requirements presented in this paper. This analysis is challenging since it requires in-depth descriptions of approaches, which are often not available. However, it would give us feedback on the relevance and completeness of these requirements, and it may identify further need for research on adaptation models. Moreover, we want to engineer a language and framework for adaptation models, which are suitable for our approach [18, 19]. A particular challenge is to engineer a single language that fulfills most of the requirements presented in this paper, and it likely will be required to integrate several languages into the framework. However, having profound knowledge about the requirements is a promising start to systematically engineer adaptation models for self-adaptive software systems based on models@run.time techniques.

References

1. Bencomo, N., Blair, G., Fleurey, F., Jeanneret, C.: Summary of the 5th International Workshop on Models@run.time. In: Dingel, J., Solberg, A. (eds.) MODELS'10 Workshops, LNCS, vol. 6627, pp. 204–208. Springer (2011)

2. Bencomo, N., Blair, G., France, R., Munoz, F., Jeanneret, C.: 4th International Workshop on Models@run.time. In: Ghosh, S. (ed.) MODELS'09 Workshops, LNCS, vol. 6002, pp. 119–123. Springer (2010)
3. Blair, G., Bencomo, N., France, R.B.: Models@run.time: Guest Editors' Introduction. Computer 42(10), 22–27 (2009)
4. Chauvel, F., Barais, O.: Modelling Adaptation Policies for Self-Adaptive Component Architectures. In: M-ADAPT'07. pp. 61–68 (2007)
5. Cheng, B.H., Lemos, R., Giese, H., Inverardi, P., Magee, J. et al.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer (2009)
6. Cheng, S.W.: Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, USA (2008)
7. Dubus, J., Merle, P.: Applying OMG D&C Specification and ECA Rules for Autonomous Distributed Component-based Systems. In: Models@run.time'06 (2006)
8. Fleurey, F., Dehlen, V., Bencomo, N., Morin, B., Jézéquel, J.M.: Modeling and Validating Dynamic Adaptation. In: Chaudron, M. (ed.) MODELS'08 Workshops, LNCS, vol. 5421, pp. 97–108. Springer (2009)
9. Fleurey, F., Solberg, A.: A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In: Schürr, A., Selic, B. (eds.) MODELS'09. LNCS, vol. 5795, pp. 606–621. Springer (2009)
10. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E.: Using Architecture Models for Runtime Adaptability. Software 23(2), 62–70 (2006)
11. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. Computer 37(10), 46–54 (2004)
12. Georgas, J.C., Hoek, A., Taylor, R.N.: Using Architectural Models to Manage and Visualize Runtime Adaptation. Computer 42(10), 52–60 (2009)
13. Kephart, J.O., Chess, D.: The Vision of Autonomic Computing. Computer 36(1), 41–50 (2003)
14. Morin, B., Fleurey, F., Bencomo, N., Jézéquel, J.M., Solberg, A., Dehlen, V., Blair, G.: An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability. In: Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M. (eds.) MODELS'08. LNCS, vol. 5301, pp. 782–796. Springer (2008)
15. Ramirez, A.J., Cheng, B.H.: Evolving Models at Run Time to Address Functional and Non-Functional Adaptation Requirements. In: Models@run.time'09. CEUR-WS.org, vol. 509, pp. 31–40 (2009)
16. Rouvoy, R.: Requirements of mechanisms and planning algorithms for self-adaptation. Deliverable D1.1 of MUSIC (EU-FP6 project) (2007)
17. Song, H., Huang, G., Chauvel, F., Sun, Y.: Applying MDE Tools at Runtime: Experiments upon Runtime Models. In: Models@run.time'10. CEUR-WS.org, vol. 641, pp. 25–36 (2010)
18. Vogel, T., Giese, H.: Adaptation and Abstract Runtime Models. In: SEAMS'10. pp. 39–48. ACM (2010)
19. Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B.: Incremental Model Synchronization for Efficient Run-Time Monitoring. In: Ghosh, S. (ed.) MODELS'09 Workshops, LNCS, vol. 6002, pp. 124–139. Springer (2010)
20. Vogel, T., Seibel, A., Giese, H.: The Role of Models and Megamodels at Runtime. In: Dingel, J., Solberg, A. (eds.) MODELS'10 Workshops, LNCS, vol. 6627, pp. 224–238. Springer (2011)

Towards Adaptive Systems through Requirements@Runtime*

Liliana Pasquale[1], Luciano Baresi[2], Bashar Nuseibeh[1,3]

[1] Lero - Irish Software Engineering Research Centre, Ireland
{liliana.pasquale|bashar.nuseibeh}@lero.ie
[2] Politecnico di Milano, Italy
baresi@elet.polimi.it
[3] The Open University, Milton Keynes, United Kingdom
b.nuseibeh@open.ac.uk

Abstract. Software systems must adapt their behavior in response to changes in the environment or in the requirements they are supposed to meet. Despite adaptation capabilities could be modeled with great detail at design time, anticipating all possible adaptations is not always feasible. To address this problem the requirements model of the system, which also includes the adaptation capabilities, is conceived as a runtime entity. This way, it is possible to trace requirements/adaptation changes and propagate them onto the application instances. This paper leverages the FLAGS [1] methodology, which provides a goal model to represent adaptations and a runtime infrastructure to manage requirements@runtime. First, this paper explains how the FLAGS infrastructure can support requirements@runtime, by managing the interplay between the requirements and the executing applications. Finally, it describes how this infrastructure can be used to adapt the system, and, consequently, support the evolution of requirements.

1 Introduction

Software systems must be able to adapt to continue to achieve their requirements while they are executing. The need for adaptation may be triggered by different events: the application reaches a particular execution state, the context changes, established requirements are not satisfied, or the objectives of the system change. Some of these events can be foreseen in advance, and thus the corresponding adaptation capabilities, *foreseen adaptations*, can be planned and designed carefully, while others cannot and adaptation capabilities, *unforeseen adaptations*, must be devised completely at runtime. Note that, besides predictability, there is a tradeoff between the number of different events the system is able to react to (completeness) and the cost of embedding these reactions from the beginning. Furthermore, even if all possible adaptations could be anticipated

* This research has been funded by Science Foundation Ireland grant 03/CE2/I303_1 and the European Commission, Programmes: IDEAS-ERC, Project 227977 SMScom, and FP7 Network of Excellence 215483 S-Cube.

from the beginning, some of them may become useless when requirements evolve, due to new business objectives or users' needs.

In the last years, different modeling notations [2,3,1] have been proposed to engineering adaptations at the requirements level. RELAX [2] is a notation to express uncertain requirements, whose assessment is affected by the imprecision of measurements. Adaptation can be designed by relaxing non-critical requirements for guaranteeing that the critical ones are still satisfied. Awareness requirements [3] are used to represent the requirements of the activities in the feedback loop. They may trigger a set of changes in the requirements model and must be aware of the satisfaction of the other requirements of the system. However, these research contributions are mainly focused on modeling foreseen adaptations, and neglect requirements evolution. In our previous work we propose the FLAGS [1] methodology. Similarly to the other approaches, it allows the designer to elicit adaptation together with the other conventional (functional and non functional) requirements of the system. Adaptation capabilities are represented as adaptation goals that are added to the KAOS [4] goal model. However, FLAGS also conceives requirements as runtime entities [5], and provides a suitable infrastructure to dynamically support their evolution.

This paper clarifies how FLAGS supports requirements@runtime. In particular, the goal model is conceived as a runtime entity and is fed by the data coming from the running application instances. The goal model can dynamically change to accommodate new/changing requirements. We assume that applications are expressed as BPEL [6] processes. Adaptation actions modeled at requirements level can affect the underlying architecture/execution environment (e.g., by restructuring the process activities or changing some partner services). They can also modify the process definition and the goal model, thus changing the actual requirements of the system. Finally the paper exemplifies how the FLAGS methodology supports the requirements evolution. In particular, every time the requirements change, their modifications must be propagated onto the process and adaptations must evolve accordingly. Note that we do not focus on the automatic identification of new requirements/adaptations capabilities, and assume that they are manually added by the designer.

The paper is structured as follows. Section 2 describes how the FLAGS model can be used to represent adaptations. Section 3 illustrates the runtime infrastructure to support requirements@runtime. Section 4 describes how the infrastructure activates adaptations at runtime and supports the interplay between the requirements and the running application. Section 5 discusses some related approaches and concludes the paper.

2 Modeling Adaptations with FLAGS

This section classifies different kinds of adaptation, and demonstrates how they can be elicited through FLAGS. We use an example of a web portal, *Click&Eat*, which collects information regarding different restaurants that deliver food at home and allows customers to order food from one of them.

2.1 Adaptation Classification

Adaptations can be classified along several dimensions [7]. For the objectives of this paper we just consider their *effect* and *anticipation*.

- **Effect.** Adaptations may just change the underlying implementation or may also modify the actual requirements of the system. In the first case, they can be performed automatically and may affect one or all application instances.
  - **Single Instance.** These adaptations are suitable to cope with transient events and can modify the execution flow of a process instance or one of its partner links. Hence, the effects of these adaptations are temporary and do not have an impact on the process definition.
  - **All Instances.** They restructure the process by changing the definition of all (executing and future) process instances. Hence, the effects of these adaptations are permanent.

  Despite it can be feasible to apply temporary actions to all process instances or permanent actions on a single instance, we did not find useful to introduce this further distinction for our purposes.

  In case some requirements cannot be satisfied, due to, for example, lack of resources or premature design choices, or to accommodate context changes, adaptation actions modify the requirements model —including the adaptation capabilities— and propagate their effect on all application instances.
- **Anticipation.** Adaptations can be planned ahead of time or not. Foreseen adaptations are modeled by the designer along with the other conventional requirements of the system. This way the underlying process can adapt autonomously when a specific scenario takes place (e.g., a goal is violated, a specific event happens). Unforeseen adaptations are determined by changes in the requirements that may take place after an application is already on the market, due to new users needs, or mutations in the organization, laws and business opportunities. Requirements changes may also be due to the execution of an adaptation that modifies the requirements model. In all these cases, new adaptations must be identified, or some of the existing adaptations may no longer be useful or may need to be modified. Despite requirements changes cannot be identified automatically, the way new requirements impact on the underlying application can be detected semi-automatically through requirements traceability.

2.2 Adaptation Elicitation

Adaptation goals must be specified over the conventional requirements of the system. Figure 1(a) shows a FLAGS model of *Click&Eat*. The general objective of the system is to manage the customers' requests (goal G1). Customers want to browse available restaurants to view offered food items and their cost (goal G1.1). To this aim they must search a set of potential restaurants (goal G1.1.1) and select one among them (goal G1.1.2). The search can be performed by name (operation *SearchByName*) or by kind of provided food (operation *SearchByType*). All customers must register (goal G1.2) to be able to make an order (goal

G1.3). This application also aims to collect the customers' feedbacks regarding a

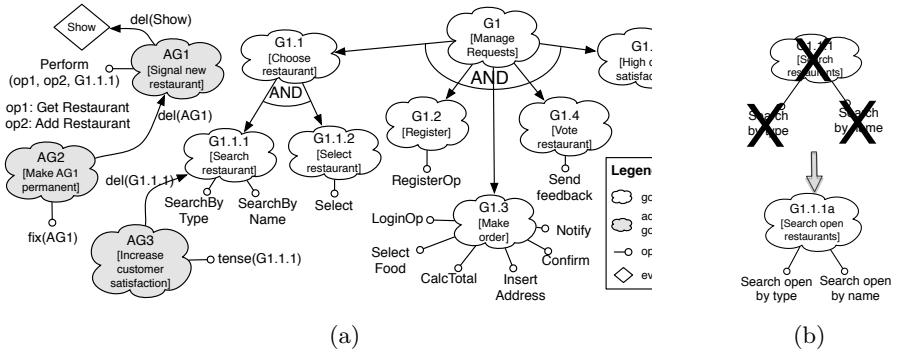


Fig. 1. The FLAGS model (a) and its modifications (b) for *Click&Eat* web portal.

Adaptation goals define the adaptation capabilities embedded in the system at requirements level. The operationalization of these goals defines the actions to be carried out when adaptation is required. Each adaptation goal is associated with a trigger and a set of conditions. The trigger states when the adaptation goal must be activated. Conditions specify further necessary constraints that must be satisfied to allow the corresponding goal to be executed. Conditions may refer to properties of the system (e.g., satisfaction levels of goals, or adaptation goals already performed) or domain assumptions.

Adaptation goals can embed process-level actions that simply change the way goals are achieved (process activities, partner services), or goal-level actions that modify the goal model. Process level adaptations have conditions that do not depend on the satisfaction of goals, but are just expressed through untimed formulas over runtime data. Furthermore, they can be performed any number of times, while goal level adaptations must be performed at most once. Process-level actions are: *perform([*o_1*,..., *o_n*], g/o)*, may optionally execute a sequence of operations (o_1, ..., o_n) and resumes the execution before operation o or goal g is executed; *substitute(*a_1*, *a_2* [, o])*, substitutes agent a_1 with a_2 for all operations performed by a_1 (in case the third parameter is not specified) or only when a_1 executes operation o (otherwise); *fix(ag)*, includes the effects of an adaptation goal (ag) in the process definition[4]. Goal level actions are: *add/remove(g)*, adds or removes (conventional/adaptation) goal g; *add(o, g)*, adds operation o to goal g; *remove(o)*, removes operation o; *add/remove(e/ev/a)*, adds or remove an entity (e), or an event (ev), or an agent (a); *relax(g, constr)*, modifies the definition

[4] Action $fix(ag)$ can be applied in case ag temporarily modifies the process execution flow, without changing the process definition or the goal model.

of a leaf goal (g) making it less strict (e.g., by adding a new disjunct constraint or relaxing the corresponding membership function); *tense(g, constr)*, modifies the definition of a leaf goal (g) by making it stricter (e.g., by adding a new conjunct constraint or tensing the corresponding membership function). Adaptation goals are associated with other elements of the goal model (dependencies). In case at least one element in the dependencies is removed or modified, the adaptation goal must be consequently removed. This way adaptations can be automatically adjusted to comply with the modifications of the goal model.

To handle those cases in which a desired restaurant cannot be found, a temporary adaptation goal (AG1) is defined. It aids goal G1.1.1, is triggered when event *Show* takes place (i.e., after operation *SearchByName*), under the condition that attribute *list* of event *Show* does not contain any restaurant. As actions, AG1 performs operations *Get Restaurant* and *Add Restaurant*, which respectively retrieve the information of a restaurant from a user and add it to the list of available restaurants. This adaptation depends on event *Show*: in case this event is removed from the the model, AG1 must be consequently removed. In case AG1 is triggered too many times, a permanent adaptation goal (AG2) is applied. It performs action *fix(AG1)*, which removes AG1 and adds the actions performed by AG1 to the process definition. It is triggered when AG1 is performed and under the condition that AG1 has been triggered more than 10 times. When some goals (G1.5) are not satisfied enough, due to wrong design choices, the requirements of the system should change. For example, adaptation goal AG3 may be applied to tense goal G1.1.1 by adding a conjuntive constraint. This new constraint asserts that only those restaurants with a feedback greater than 40% must be shown to the customer.

The customers may change their requirements while the system is executing. For example, they may want to visualize only those restaurants that are open at the moment when the search is performed. To address these changes it is necessary to modify the goal model manually, by substituting goal G1.1.1 with G1.1.1a, as shown in Figure 1(b). Consequently, new unforeseen adaptations must be automatically applied at the applications level to propagate these modifications to all instances.

3 The Runtime Infrastructure

As shown in Figure 2, FLAGS provides a conceptual infrastructure to enact requirements at runtime. Its components operate respectively at the process and the goal level, whereas the *Process Reasoner* handles the interplay between them. To support requirements evolution the infrastructure manages two models at runtime: the *FLAGS model* and the *implementation model*. The former includes the requirements and the adaptation capabilities performed at the goal level, while the latter includes the definition of the process together with the data collection, monitoring and adaptation capabilities necessary to support the adaptations at the process level. These models are managed respectively by the components at the goal and process level.

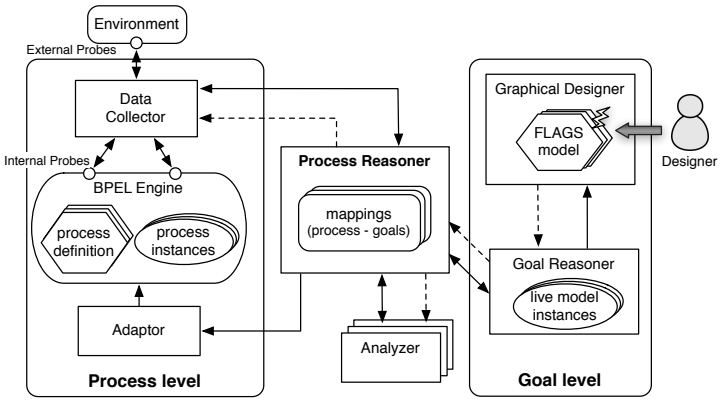


Fig. 2. Runtime infrastructure.

At the process level a *BPEL engine* supports the execution of the process instances. We use a modified version of ActiveBPEL [8] engine, which is augmented with Aspect Oriented Programming [9]. It provides a customizable "aspect" that can be used to intercept the process execution at specific points, for example to collect the process internal state or apply some adaptation actions. The *Data Collector* manages the collection of runtime data from the process or the environment. The process state is collected by the *Internal Probes* that are activated when the process reaches specific execution points. Conversely, the data from the surrounding environment are gathered by the *External Probes* that expose a proper interface to be configured and enacted at runtime.

Runtime data are sent to the *Goal Reasoner*, through the Process Reasoner, and are used to update the elements of the goal model (i.e., events, entities, satisfaction of leaf goals and performed adaptations). The Process Reasoner also evaluates the satisfaction of leaf goals, by invoking a specific *Analyzer*, depending on the kind of constraint (i.e., temporal or untimed) that must be checked. When the goal model changes, the Goal Reasoner notifies the Process Reasoner, which propagates these modifications on the running and next process instances. To manage the interplay between the goal model and the process, the Process Reasoner uses a bidirectional *mapping* between the elements of the FLAGS model, and those of the implementation model. Note that the adaptations that are applied at runtime may generate different versions of the implementation model and the FLAGS model. For this reason, the Process Reasoner must store different mappings, depending on the versions of the models that are in use.

The Process Reasoner orchestrates the adaptation at the process level, according to the directives of the implementation model. It monitors the runtime data to check if the trigger and conditions of an adaptation are satisfied. In this case, it activates proper adaptation actions through an *Adaptor*, which can temporarily change the execution flow of a single process instance or may change

the implementation model, by modifying the definition of the process and the adaptation capabilities. To adapt the running process instances, the Adaptor intercepts the execution at a "safe" point, where the modifications can be correctly applied, since they do not break any conversation or transaction. To adapt the next process instances the Adaptor transparently deploys a new version of the process in the BPEL engine.

At the goal level, the *Graphical Designer* [10] allows the designer to create a new version fo the FLAGS model and modify it at runtime. It also stores all versions of the FLAGS model that are actually in use. The Goal Reasoner manages the live instances of the FLAGS model. It creates a new instance every time the Process Reasoner signals that a process instance is started. This live instance must conform to the last version of the FLAGS model. Every time a live instance of the FLAGS model is updated, a set of rules recompute the satisfaction of high level goals, evaluate the triggers and conditions of the adaptation goals, and decide which adaptation goal must be applied, if necessary. In our implementation the Goal Reasoner is based on JBoss rule engine [11].

Every time the last version of the FLAGS model is modified at runtime, a new version is created. Consequently, all live instances of the FLAGS model and their corresponding process instance must migrate to the new version of the model, if possible. A new version of the implementation model is inferred. In particular, the new version of the process is inferred and deployed on the BPEL Engine, the new mappings between the goal model and the process are generated and stored at the Process Reasoner, and the components at the process level must be properly re-configured to update and apply process level adaptations for the modified instances of the FLAGS model. A live instance of the FLAGS model is canceled every time the Process Reasoner signals that the corresponding process instance has terminated. In case there is no running instance of a process associated with an old version of the FLAGS model, that version is removed together with its corresponding mappings at the Process Reasoner.

4 Managing Adaptations @Runtime

This section describes how adaptations are applied at runtime and their impact on the implementation and the FLAGS model. The implementation model may change when process level adaptations are performed. The FLAGS model can change when the designer manually modifies it or when goal level adaptations are applied, and, in these cases, the implementation model must evolve accordingly.

4.1 Process Level Adaptations

Adaptations at the process level are applied on a single process instance and may perform a small set of actions. They can change the process execution flow (action $perform$) or substitute a partner service with another one (action *substitute*). These adaptations require to keep the process blocked at specific execution points, while their conditions are checked and their actions are applied,

if necessary. The trigger of these adaptations must clearly identify the execution point where the adaptations are performed. Since conditions must be evaluated "on-the-fly", while the process is blocked, they must be expressed as untimed formulas. The Process Reasoner can verify them by invoking one of the Analyzers that have been plugged in the infrastructure.

To apply action *perform*$([o_1, \ldots, o_n], g/o)$, the operations included in the first argument $(op_1, \ldots, op_n)$ are translated into a sequence of activities that must be temporarily performed. These activities can be executed, for example, by invoking an external partner service. The second argument is associated with a concrete execution point where the process execution must be restored. This action can be applied at runtime through a suitable Adaptor, such as Dynamo [12], which provides recovery actions call(wsdl, operation, ins) and restore(destLocation). The former calls an operation exposed by an external web service, which is identified by its wsdl. The third parameter (ins) represents the data that are to be sent to the service. Action restore takes the process back to the point of execution immediately prior to the destLocation (indicated with an XPath expression), and resumes the process execution from there.

To apply action *substitute*$(a_1, a_2$ *[, o]*$)$, the BPEL partner links p_1, p_2 that are associated with agents a_1, a_2 must be identified. This action substitutes p_1 with p_2 for all process activities in which p_1 is used, in case the third parameter is not specified. Otherwise, it is necessary to identify the process activities associated with operation o, and substitue p_1 with p_2 for all of them. Both adaptation actions can be supported by Dynamo. In the first case, we can use recovery action rebindPartnerLink(name, wsdl), which changes partner link p_1, identified by a name, with p_2, identified by its wsdl. In the second case we can apply action rebind(wsdl, operation), which substitutes partner link p_1, which performs activity operation, with p_2, identified by its wsdl.

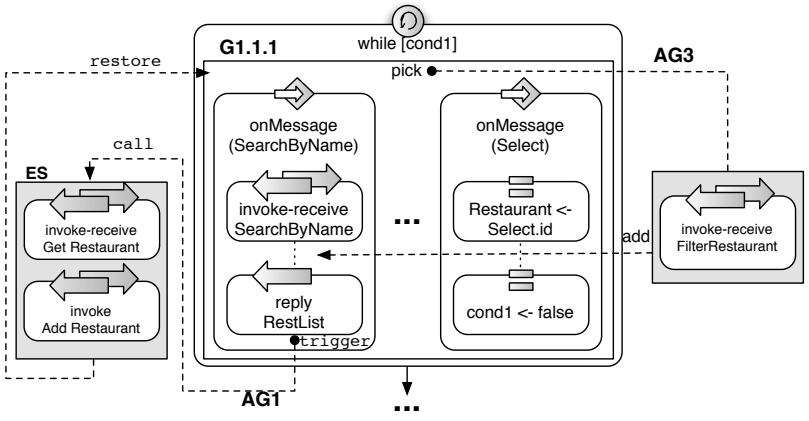


Fig. 3. BPEL process for the Click&Eat application.

For example, AG1 is an adaptation goal that applies action $perform(GetRestaurant,\ AddRestaurant,\ G1.1.1)$ on a single process instance. To support its enactment at runtime, the Internal Probes must be instrumented to intercept the process execution at the points when event *Show* (trigger) takes place. In our example (see Figure 3) this corresponds to the execution point after activity reply *RestList*. This way, every time the process terminates this activity, its execution is blocked, and, in case the condition is satisfied, an external service (ES) is invoked. It performs a sequence of activities associated with the operations *GetRestaurant* and *Add Restaurant* and restores the process execution before activity pick, which is the execution point where G1.1.1 starts to be activated.

All adaptations that are applied on all process instances have a permanent effect on the implementation model. The trigger of these adaptations only identifies the moment when the condition can be verified. A set of "safe points" must be identify the process point at which an adaptation must be applied. In case a running process instance has already passed all safe points, it cannot be migrated. All the other adaptations at the process level are temporarily deactivated, until all instances complete the migration to the next version of the process, when possible. The new mappings between the new version of the process and the last version of the FLAGS model must be also added to the Process Reasoner. Finally the adaptation capabilities at the Data Collector and Process Reasoner may also change.

AG2 falls in this category, since it applies action *fix(AG1)* to modify the process definition. After the Process Reasoner performs AG1 (trigger) for one of its process instances, the condition associated with AG2 must be verified. AG2 applies a set of modifications at the Process Reasoner, since it removes AG1 and all the directives necessary to support its execution. AG2 also modifies the process definition by inserting a set of activities after the execution point identified by the trigger of AG1 (i.e., after activity reply *RestList*). These activities are: an if activity, which verifies the condition of AG1, and the activities that were temporarily performed by external service *ES*. A new version of the process that includes all these modifications is deployed as well in the BPEL Engine.

4.2 Goal Level Adaptations

For goal level adaptations things work slightly differently. Their trigger and conditions are checked by the Goal Reasoner every time a new element of the goal model is updated. When an adaptation at the goal level is applied, the Goal Reasoner suspends all further updates of the goal model and stores all runtime data received from the Process Reasoner in a buffer, until the adaptation completes. It also sends a notification to the Process Reasoner to block all running process instances at the end of the activity they are currently executing. An adaptation at the goal level generates a new version of the FLAGS model, and must apply the corresponding modifications onto its live instances. Note that only those instances that comply with the last version of the FLAGS model can potentially migrate to the new version. The migration can take place only if

the corresponding process instance has not already passed the safe points where modifications must be applied.

A new version of the implementation model is created accordingly. In particular, the Process Reasoner creates a new version of the mappings between the process and the goal model and indicates what are the process instances that must actually follow this mapping. Consequently the probes and the Process Reasoner change their configuration to respectively intercept the process at different points, collect different data, evaluate different leaf goals (specified with different constraints) and activate different adaptation actions. Finally the definition of all running and next process instances must change to comply with the modifications of the goal model. For example, if some goals or operations are added/removed, the corresponding activities in the process definition must be added/removed. In case agents are added/removed, the corresponding partner links must be added/removed from/to the process definition and so on.

After all aforementioned actions are performed, the execution of all process instances can be restored. All runtime data received by the Process Reasoner that are not associated with any element at the goal level are automatically removed. The Goal Reasoner will start to process the data stored in its buffer only after all process instances correctly migrated to their new version (if possible). The Goal Reasoner will discard all the data that are associated with leaf goals, entities and events that do not exist anymore or have been modified. Goal level adaptations can be triggered while a process level adaptation, which modifies the process definition, is being performed on the process. In this case, it is necessary to guarantee that only the previous adaptation is applied, or both of them are applied. In other cases a goal level adaptation can be triggered while one or more process instances are performing a process level adaptation that just temporarily modifies the process execution flow. In this case the process instance must terminate the execution of the temporary activities and must be blocked before performing the activity associated with the restore point.

In our example, AG3 applies a goal level adaptation when the satisfaction of G1.5 is low (trigger). It changes the definition of goal G1.1.1 and its operationalization. As an effect, the Goal Reasoner creates a new version of the FLAGS model, manage the migration for the live instances of the model, and adds a new version of the mappings to the Process Reasoner. At the process level the next process instances can be migrated by deploying a new version of the BPEL process, which complies with the new version of the FLAGS model. The running process instances can be migrated by intercepting their execution before activity pick and adding activity invoke-receive *FilterRestaurant* after invoke-receive *SearchByName*, as shown in Figure 3. A similar procedure is followed when a new version of the FLAGS model is manually created by designer, who directly modifies the last version of the FLAGS model. This can happen when, for example, the requirements of the system change, as shown in Figure 1(b). In this case, the Graphical Designer notifies the Goal Reasoner that propagates the modifications on the live instances of the FLAGS model and on the process. A goal model at runtime allows us to reshape the adaptations at the process level,

depending on the modifications applied on the process. In this example all the adaptation goals that have been defined can persist, since goal G1.1.1 and event *Show* are not removed. However the way adaptations are supported at runtime changes. For example, at the process level AG3 will be added after another activity (invoke-receive *SearchOpenByName*), and will modify the new definition of G1.1.1 by adding its conjunct constraint to its modified consequent.

5 Discussion

Different solutions have been already proposed to support requirements evolution. Courbis and Finkelstein [13] analyze the requirements in a number of possible environments where the system can execute. This analysis is used to identify alternative requirements definition and architectural choices to foster "design for change". Note that identifying all possible changes in advance is not always possible. For this reason requirements@runtime [5] are fundamental to trace the modifications of both the requirements and the operative environment where the system is executing. Ali et al. [14] use requirements at runtime to support their evolution when wrong assumptions are discovered. Requirements evolution can be performed automatically, by changing the priority given to software variants, or can be manually performed by the designer. However this work neglects how requirements changes are applied onto the system.

To support requirements at runtime the link between the requirements and the underlying implementation cannot be lost. The changes in the system requirements at runtime may trigger the execution of a set of analysis (e.g., consistency check, requirements verification) that have been traditionally performed offline. Baresi and Ghezzi [15] foster this idea. In particular, they claim that the rigid boundary between development time and runtime must be broken and more support must be provided to analyze and re-design the software at runtime. According with this idea, Epifani et al. [16] use a live model of the system to reason about its reliability. They use runtime data to feed the model and perform probabilistic analysis to improve its accuracy. The updated model can be used to detect or predict if a reliability property will be violated in the running implementation. Our work also maintains the link between the requirements model and the implementation model and apply unforeseen requirements and application features through aspects, as already proposed by Courbis and Finkelstein [17]. A similar idea has been also proposed in the DiVA project [18], which models variability dimensions as aspects and performs dynamic aspect weaving as model transformations when an adaptation need occurs.

Our approach needs further improvements. It lacks a mechanism to verify the consistency and correctness of the models updated by the designer. It should better handle conflicts between adaptations that are activated at the same time. We are also planning to exploit requirements@runtime in the security domain. Security is a critical property whose violation must be avoided. The countermeasure used to support this property depend on the context and, for this reason, it may be fundamental to detect changes that can take place in the assets to be

protected or in the environment. These changes may trigger suitable analysis at runtime and activate new countermeasures necessary to continue to guarantee the satisfaction of security requirements.

References

1. Baresi, L., Pasquale, L., Spoletini, P.: Fuzzy Goals for Requirements-Driven Adaptation. In: Proc. of the 18th Int. Requirements Eng. Conf. (2010) 125–134
2. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C.: RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In: Proc. of the 17th Int. Requirements Eng. Conf. (2009) 79–88
3. Silva Souza, V.E., Lapouchnian, A., Robinson, W.N., Mylopoulos, J.: Awareness Requirements for Adaptive Systems. In: Proc. of the 6th Int. Symposium on Software Eng. for Adaptive and Self-Managing Systems. (2011) 60–69
4. van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. John Wiley (2009)
5. Sawyer, P., Bencomo, N., Whittle, J., Letier, E., Finkelstein, A.: Requirements-Aware Systems: A Research Agenda for RE for Self-adaptive Systems. In: Proc. of the 18th Int. Requirements Eng. Conf. (2010) 95–103
6. OASIS WSBPEL TC: Web services business process execution language. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`
7. B. Cheng et al.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Software Engineering for Self-Adaptive Systems. Volume 5525. Springer Berlin / Heidelberg (2009) 1–26
8. Active Endpoints: The activebpel engine. `http://www.activevos.com/community-open-source.php`
9. G. Kiczales et al.: Aspect-Oriented Programming. In: Proc. of the 11th European Conference on Object-Oriented Programming. (1997) 220–242
10. Baresi, L., Pasquale, L.: An Eclipse Plug-in to Model System Requirements and Adaptation Capabilities. In: Proc. of the 6th IT-Eclipse Workshop. (2011) (to appear)
11. JBoss Drools Team: Drools expert. `http://jboss.org/drools`
12. Baresi, L., Guinea, S.: Self-Supervising BPEL Processes. IEEE Trans. on Software Eng. **37**(2) (2011) 247–263
13. Bush, D., Finkelstein, A.: Requirements Stability Assessment Using Scenarios. In: Proc. of the 11th Int. Conf. on Requirements Eng. (2003) 23–32
14. Ali, R., Dalpiaz, F., Giorgini, P., Souza, V.E.S.: Requirements Evolution: From Assumptions to Reality. In: Proc. of the 12th Int. Conf. BMMDS/EMMSAD. (2011) 372–382
15. Baresi, L., Ghezzi, C.: The Disappearing Boundary Between Development-time and Run-time. In: Proc. of the Workshop on Future of Software Eng. Research. (2010) 17–22
16. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: Proc. of the 31st Int. Conf. on Software Eng. (2009) 111–121
17. Courbis, C., Finkelstein, A.: Towards Aspect Weaving Applications. In: Proc. of the 27th Int. Conf. on Software Eng. (2005) 69–77
18. DiVA-Dynamic Variability in complex, adaptive systems. `http://www.ict-diva.eu/`

Model-based Situational Security Analysis

Jörn Eichler and Roland Rieke

Fraunhofer Institute for Secure Information Technology SIT, Darmstadt, Germany
{joern.eichler,roland.rieke}@sit.fraunhofer.de

Abstract. Security analysis is growing in complexity with the increase in functionality, connectivity, and dynamics of current electronic business processes. To tackle this complexity, the application of models in pre-operational phases is becoming standard practice. Runtime models are also increasingly applied to analyze and validate the actual security status of business process instances. In this paper we present an approach to support not only model-based evaluation of the current security status of business process instances, but also to allow for decision support by analyzing close-future process states. Our approach is based on operational formal models derived from development-time process and security models. This paper exemplifies our approach utilizing real world processes from the logistics domain and demonstrates the systematic development and application of runtime models for situational security analysis.

Keywords: security requirements elicitation, predictive security analysis, analysis of business process behavior, security modeling and simulation, security monitoring

1 Introduction

Electronic business processes connect many systems and applications. This leads to an increasing complexity when analyzing distinctive properties of those business processes. Additionally, frequent changes to business process models are applied to address changing business needs. Current approaches apply changes to those models at runtime [4]. This situation challenges operators and participants in electronic business processes as the assessment of the status of business process instances at runtime becomes difficult. An example for these difficulties is the assessment whether instances of business processes violate security policies or might violate them in the near future.

Traditionally, approaches to security analysis of electronic business processes are executed at development-time. In this perspective, the analysis of possible violations of security policies is part of the requirements engineering process [13]. To cope with the growing complexity of the electronic business processes, the application of security models in the course of the requirements engineering process is becoming a common strategy [5]. Nevertheless, the requirements engineering process is generally limited to development-time.

Contributions. To support security analysis at runtime we utilize formal models based on development-time process and security models. On the basis of

sound methods for the elicitation and modeling of security requirements provided in [7] and an architectural blueprint described in [18], we document in this paper our approach to analyze the security status of electronic business processes. The security analysis consumes events from the runtime environment, maps those events to security events and feeds them to our runtime model, an operational finite state model. This allows to match and synchronize the state of the real process with the state of the model. Annotations of security requirements to the states of the model can now be used to check for security violations and possibly generate alarms. These alarms are then in turn converted to events and sent to the running business process. Furthermore, a computation of possible close-future behavior, which is enabled by the model of the business process, is used to evaluate possible security critical states in the near future at runtime. This knowledge about possibly upcoming critical situations can be used to raise respective predictive alarms.

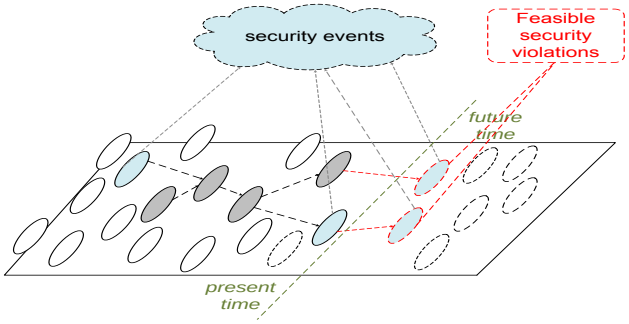


Fig. 1. Predict feasible security violations

In section 2 we provide an application scenario from the logistics domain and elicit security requirements. The formalization of the scenario is given in section 3. Section 4 analyzes the runtime operation and exemplifies generated security alerts. Section 5 reviews shortly related work to our approach. Concluding remarks and further research directions are given in section 6.

2 Application Scenario

In order to demonstrate what kind of security requirements we are able to consider and how our model-based runtime analysis is applied, we have chosen a small part of a "*Pickup*" target process which is analysed in the project Alliance Digital Product Flow (`http://www.adiwa.net/`).

2.1 Process and Event Model

The "Pickup" process is initiated when the truck driver is notified about new pickup orders. He accepts the received list of orders and the system calculates a route plan based on the addresses. When the driver arrives at a pickup address, he checks visually the packages for deviations with regard to the description in the order. In case of deviations he consults with the sender whether this package is to be transported. If the truck driver accepts the new package, the package description in the list of orders is updated accordingly. For each accepted package the system receives a confirmation that it has been loaded. The system links each loaded package and its transporting truck using the corresponding radio-frequency identification (RFID) tag identifiers. An Event-driven Process Chain (EPC) flowchart of the considered subprocess is depicted in Figure 2. Rectangles with rounded corners denote actions and chamfered rectangles denote events.

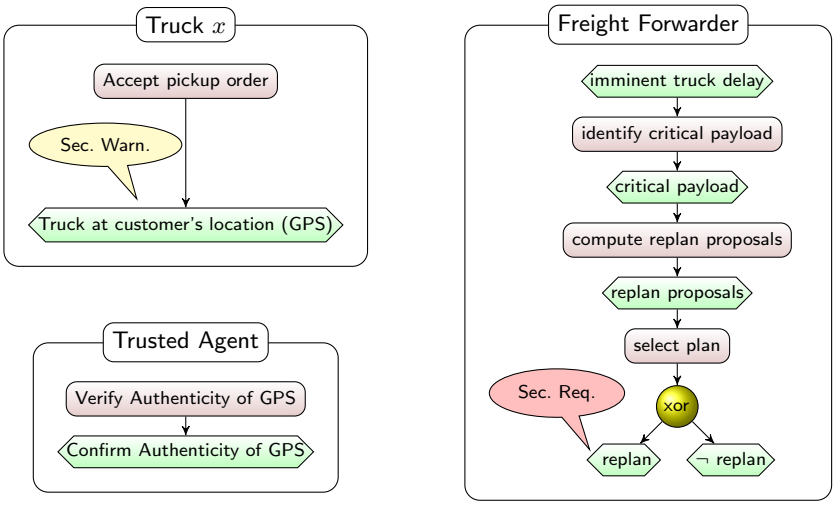


Fig. 2. Model of a part of the Pickup Process (EPC notation)

As an example of a security threatening misuse case, we consider a situation where the system performs a rescheduling because of a delay of one or more trucks on the basis of not confirmed Global Positioning System (GPS) locations. In this case there is a possibility for an attacker to send false GPS data to the system, which may result in ineffective rescheduling and possible time loss in completing the orders.

2.2 Security Requirements Elicitation

In order to derive the security requirements in the given scenario, we follow the scheme described in [7]. We assume that the functional dependencies between the actions in our scenario are given by Fig. 3.

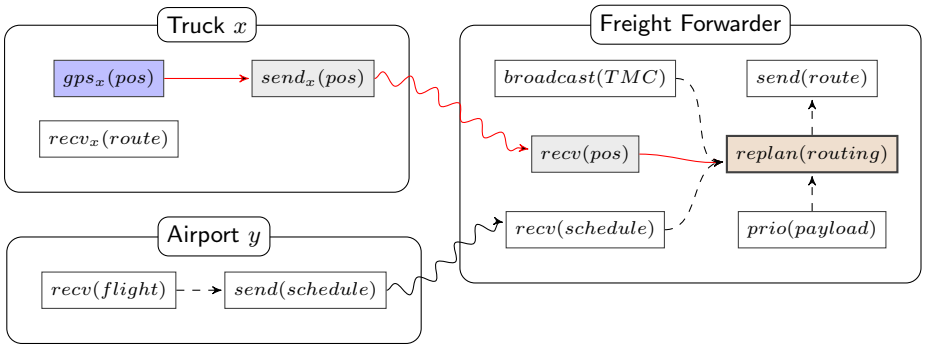


Fig. 3. Functional dependencies

We apply a general security goal: *Whenever a certain output action happens, the input actions that presumably led to it must actually have happened.* As an example for a specific security goal, in the following we will use the authenticity requirement: *Whenever a rescheduling action is performed, the GPS coordinates of each truck should be authentic for the dispatcher in terms of origin, content and time.* The formal syntax to describe these requirements in parameterized form is defined as (see [8]):

Definition 1. *$auth(a, b, P)$: Whenever an action b happens, it must be authentic for an Agent P that in any course of events that seem possible to him, a certain action a has happened.*

Therefore, our selected authenticity requirement can be written as:

$$auth(gps_x(pos), replan(routing), dispatcher). \qquad \text{(Auth 1)}$$

We will use the authenticity requirement (Auth 1) to describe the reasoning process with the help of an appropriate operational model.

There are of course many other security requirements necessary in this scenario. For example, *while loading a package on the truck the RFID data and the truck driver should be authentic in terms of content and identification number.* Analysis and application in our situational security analysis follow the same procedure as for (Auth 1). Therefore, we will exemplify only (Auth 1) in the following.

3 Formal Model

In order to analyze the system behavior with tool support, an appropriate formal representation has to be chosen. In our approach, we use an operational finite state model of the behavior of the given process which is based on *Asynchronous Product Automata (APA)*, a flexible operational specification concept

for cooperating systems [16]. An APA consists of a family of so called *elementary automata* communicating by common components of their state (shared memory). We now introduce the formal modeling techniques used, and illustrate the usage by our application example.

Definition 2 (Asynchronous Product Automaton (APA)). *An* Asynchronous Product Automaton $\mathbb{A} = ((Z_s)_{s\in\mathbb{S}}, (\Phi_t, \Delta_t)_{t\in\mathbb{T}}, N, q_0)$ *consists of a family of* state sets $Z_s, s \in \mathbb{S}$, *a family of* elementary automata (Φ_t, Δ_t), *with* $t \in \mathbb{T}$, *a* neighborhood relation $N : \mathbb{T} \to \mathfrak{P}(\mathbb{S})$ *and an* initial state $q_0 = (q_{0s})_{s\in\mathbb{S}} \in \bigtimes_{s\in\mathbb{S}}(Z_s)$. $\mathbb{S}$ *and* $\mathbb{T}$ *are index sets with the names of state components and of elementary automata and* $\mathfrak{P}(\mathbb{S})$ *is the power set of* $\mathbb{S}$. *For each elementary automaton* (Φ_t, Δ_t) *with* Alphabet Φ_t, *its* state transition relation *is* $\Delta_t \subseteq \bigtimes_{s\in N(t)}(Z_s) \times \Phi_t \times \bigtimes_{s\in N(t)}(Z_s)$. *For each element of* Φ_t *the state transition relation* Δ_t *defines state transitions that change only the state components in* $N(t)$. *An APA's (global)* states *are elements of* $\bigtimes_{s\in\mathbb{S}}(Z_s)$. *To avoid pathological cases it is generally assumed that* $N(t) \neq \emptyset$ *for all* $t \in \mathbb{T}$. *An elementary automaton* (Φ_t, Δ_t) *is* activated *in a state* $p = (p_s)_{s\in\mathbb{S}} \in \bigtimes_{s\in\mathbb{S}}(Z_s)$ *as to an* interpretation $i \in \Phi_t$, *if there are* $(q_s)_{s\in N(t)} \in \bigtimes_{s\in N(t)}(Z_s)$ *with* $((p_s)_{s\in N(t)}, i, (q_s)_{s\in N(t)}) \in \Delta_t$. *An activated elementary automaton* (Φ_t, Δ_t) *can execute a state transition and produce a* successor state $q = (q_r)_{r\in\mathbb{S}} \in \bigtimes_{s\in\mathbb{S}}(Z_s)$, *if* $q_r = p_r$ *for* $r \in \mathbb{S} \setminus N(t)$ *and* $((p_s)_{s\in N(t)}, i, (q_s)_{s\in N(t)}) \in \Delta_t$. *The corresponding state transition is* $(p, (t, i), q)$.

A simplified model of the part of the "Freight Forwarder" business process shown in Fig 2 contains the APA state components *pstate* and *event* representing the current process state and event. Formally, $\mathbb{S} = \{pstate, event\}$, with $Z_{event} = \{imminent\_truck\_delay, \ldots, replan, \neg replan\}$, $Z_{pstate} = \ldots$.

The *elementary automata* $\mathbb{T} = \{identify\_critical\_payload, \ldots, select\_plan\}$ represent the possible actions that the systems can take. The neighborhood relation between elementary automata and state components of the APA model is depicted by the edges in Fig. 4.

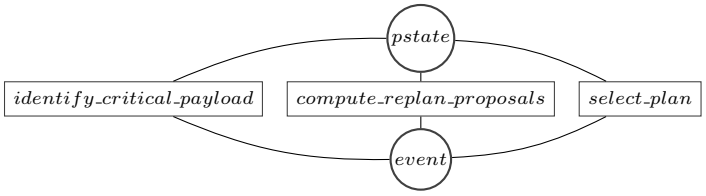


Fig. 4. Elementary automata and state components in the APA process model

Formally, the behavior of our operational APA model of the business process is described by a reachability graph. In the literature this is sometimes also referred to as labeled transition system (LTS).

Definition 3 (Reachability graph). *The behavior of an APA is represented by all possible coherent sequences of state transitions starting with initial state* q_0. *The sequence* $(q_0, (t_1, i_1), q_1)(q_1, (t_2, i_2), q_2) \ldots (q_{n-1}, (t_n, i_n), q_n)$ *with* $i_k \in \Phi_{t_k}$ *represents one possible sequence of actions of an APA. State transitions* $(p, (t, i), q)$ *may be interpreted as labeled edges of a directed graph whose nodes are the states of an APA:* $(p, (t, i), q)$ *is the edge leading from* p *to* q *and labeled by* (t, i). *The subgraph reachable from* q_0 *is called* reachability graph *of an APA.*

We use the *SH verification tool* [16] to analyse the process model. This tool provides components for the complete cycle from formal specification to exhaustive validation as well as visualisation and inspection of computed reachability graphs and minimal automata. The applied specification method based on APA is supported. The tool manages the components of the model, allows to select alternative parts of the specification and automatically *glues* together the selected components to generate a combined model of the APA specification.

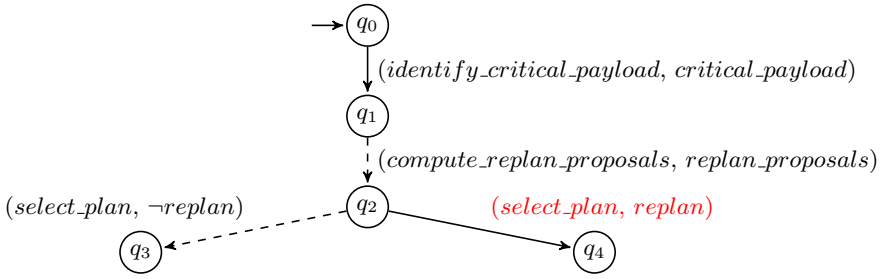


Fig. 5. Close-future (3 Steps) Reachability Analysis

Figure 5 shows the initial part of the reachability graph resulting from the analysis of the model when reaching the part of the business process of the freight forwarder shown in Fig. 2. An example for a state transition of the model in this situation is: (q_0,($identify\_critical\_payload$, $critical\_payload$),q_1). Please note that there are two different transitions from the state q_2 because the interpretation of a variable can have the values $replan$ or $\neg\ replan$, respectively.

4 Runtime Operation and Generated Alerts

During runtime, the events from the business process are used to synchronize the state of the model with the real process. In our exemplary setup, the events are produced by a Complex Event Processing (CEP) engine which is provided by one of the project partners. The events are described by an XML schema and communicated by the Java Message Service (JMS). The events from the event bus are used to provide the information about the state and input to the business process. In our finite state model, this information is represented in the state

components *pstate* and *event* (cf. Fig. 4). This constitutes the initial state of the model from which a simulation is then started. In addition to the predicted system behavior, we also need the information on the security requirements in order to identify critical situations. In [18] we proposed to use APA to specify meta-events, which match security critical situations, to generate alerts. However, since this is slow and not easily usable by end-users, we decided to build the matching algorithm directly into the SH verification tool. We use monitor automata [22] to specify the security requirements graphically. These automata monitor the behaviour of the abstract system during the run of the simulation and provide interfaces to trigger alerts. This concept could be further extended to make use of the built-in temporal logic based reasoning component if more complex reasoning is necessary.

4.1 Security Reasoning – No Authenticity Approval of GPS Event

In order to demonstrate the use of process models at runtime, let us assume the following situation. We are currently at logical time 0 as depicted on the timeline in Figures, 7, 8, 9, 10. We further assume that the trusted agent inspects the events generated by GPS units of the trucks and sends additional events which attest to the authenticity of each GPS event within a timeframe of 2 logical time units. Please note that it is also a possibility that the trusted agent would filter the events and only let authentic events pass the filter. We furthermore assume that we know from the analysis of dependencies of actions and specifically from the requirement (Auth 1) that whenever a rescheduling action is performed, the GPS coordinates of each truck should be authentic for the process planner in terms of origin, content and time.

We now describe the reasoning process where the authenticity of the GPS event is not approved by the trusted agent. In the diagrams we use pentagon symbols to depict events on the event bus such as GPS information and we use triangles to depict Security Warnings (SW), Predictive Security Alerts (PSA) and Security Alerts (SA) generated by the reasoning process.

Step 1: A GPS event is received

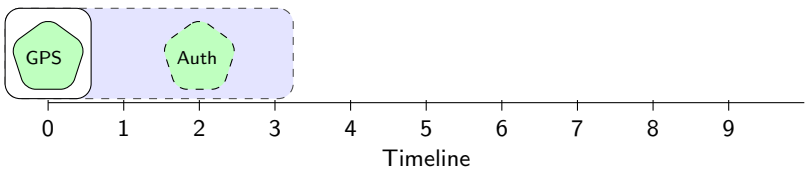


Fig. 6. Security Reasoning - Authenticity Approval of GPS Event - step 1

Figure 6 shows the situation when a GPS event is received. This event is matching a precondition in the requirement pattern: *GPS needs confirmation*

in 2 steps. This requirement (warn-level) is triggered by the GPS event. The reachability analysis reveals no critical actions within the scope (3 steps) of the analysis. We conclude from Fig. 6 that everything is OK at this point. A future event might confirm authenticity of the GPS location received.

Step 2: Confirmation of GPS event not received

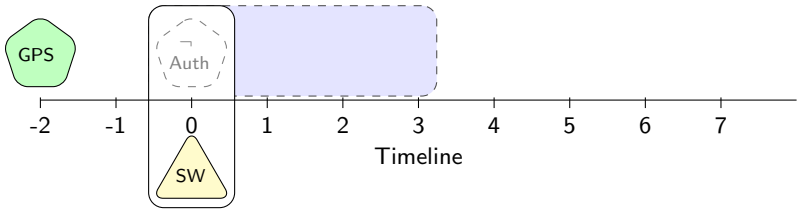


Fig. 7. Security Reasoning - No Authenticity Approval of GPS Event - step 2

Figure 7 shows the situation when an expected event from the trusted agent, namely the authenticity approval of this GPS event is recognized as missing. The missing event indicates a broken security requirement: *GPS needs confirmation in 2 steps*. The reachability analysis in this situation shows that no other security requirement will be triggered within the scope of the analysis. However, some forthcoming security relevant action might require authenticity of this GPS event. Therefore, an alert action associated with a broken warn-level requirement, such as issuing a security warning (SW), is now triggered.

Step 3: Replan event in analysis scope

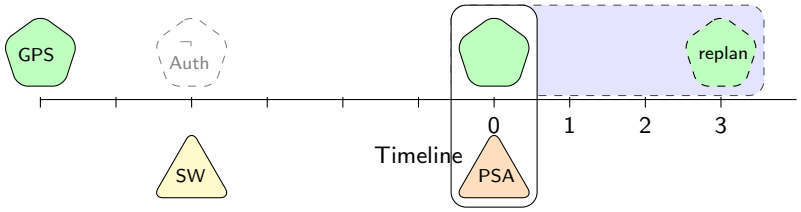


Fig. 8. Security Reasoning - No Authenticity Approval of GPS Event - step 3

In Fig. 8, an arbitrary event is received from which, in one possible execution sequence of the business process, a *replan* event is reachable within the scope of the analysis. In our scenario *imminent_ truck_delay* is such an event. The reachability graph is similar to the one depicted in Fig. 5. It shows that the *select_plan* action may happen in the future if *replan* is chosen. But there

is another possible path in the graph where *replan* is not chosen. The *replan* event in the prediction scope is matching a precondition in a requirement pattern: $auth(GPS, replan, dispatcher)$, but the GPS event is not approved to be authentic. Therefore, a *replan* event with broken security requirement is possible. An action associated with this (possibly) broken alert-level requirement, such as issuing a predictive security alert (PSA), is now executed.

Step 4a: Expected *replan* event received

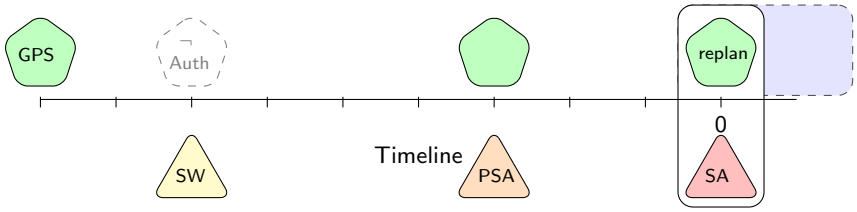


Fig. 9. Security Reasoning - No Authenticity Approval of GPS Event - step 4a

Figure 9 shows the situation when a *replan* event is received as predicted (cf. Fig. 5 transition $q_2 \rightarrow q_4$). At this time we know that the security requirement (Auth 1) is broken. Therefore, an action associated with a broken alert-level requirement, such as issuing a security alert (SA), is now executed.

Step 4b: Predicted *replan* event not received after step 3

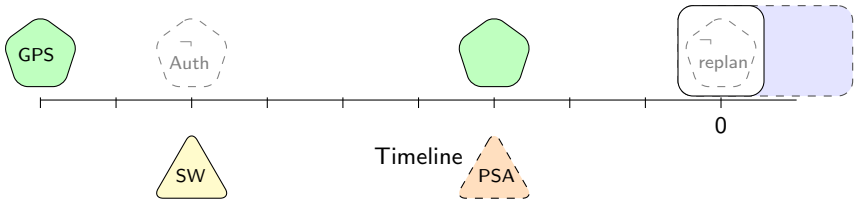


Fig. 10. Security Reasoning - No Authenticity Approval of GPS Event - step 4b

Figure 10 shows the situation when a *replan* event is not received as expected (cf. Fig. 5 transition $q_2 \rightarrow q_5$). In this case, we know that the issued predictive security alert (PSA) was a "False Positive", so a corrective action may be necessary. Corrective actions might be the reduction of a general security warning level or lifting of restrictions on the business process depending on the operating environment. However, the security warning issued in step 2 is still valid because some future event might require authenticity of the GPS event.

5 Related Work

The work presented here combines specific aspects of security analysis with generic aspects of process monitoring, simulation, and analysis. The background of those aspects is given by the utilization of models at runtime [6]. A blueprint for our architecture of predictive security analysis is given in [18].

Security analysis *at development-time* to identify violations of security policies is usually integrated in the security requirements engineering process. An overview of current security requirements engineering processes is given in [5,13]. The security requirements elicitation methods developed in [7] are used in section 2 to derive the requirements which are needed to assess possible security policy violations at runtime. A formalized approach for security risk modeling in the context of electronic business processes is given in [21]. It touches also the aspect of simulation, but does not incorporate the utilization of runtime models. Approaches that focus security models at runtime are given in [14] or in [12]. Morin et. al [14] propose a novel methodology to synchronize an architectural model reflecting access control policies with the running system. Therefore, the methodology emphasizes policy enforcement rather than security analysis. The integration of runtime and development-time information on the basis of an ontology to engineer industrial automation systems is discussed in [12].

Process monitoring has gained some popularity recently in the industrial context prominently accompanied with the term Business Activity Monitoring (BAM). The goal of BAM applications, as defined by Gartner Inc., is to process events, which are generated from multiple application systems, enterprise service buses or other inter-enterprise sources in real-time in order to identify critical business key performance indicators and get a better insight into the business activities and thereby improve the effectiveness of business operations [11]. Recently, runtime monitoring of concurrent distributed systems based on linear temporal logic (LTL), state-charts, and related formalisms has also received a lot of attention [9,10]. However, these works are mainly focused on error detection, e.g., concurrency related bugs. A classification for runtime monitoring of software faults is given in [1]. Patterns to allow for monitoring security properties are developed in [20]. In the context of BAM applications, in addition to these features we propose a *close-future* security analysis as it is detailed in section 4. Our analysis provides information about possible security policy violations reinforcing the security-related decision support system components.

Different categories of tools applicable for simulation of business processes including process modeling tools are based on different semi-formal or formal methods such as Petri Nets [3] or Event-driven Process Chains (EPC) [2]. Some process management tools such as FileNet [15] offer a simulation tool to support the design phase. Also, some general-purpose simulation tools such as CPNTools [19] were proven to be suitable for simulating business processes. However, independently from the tools and methods used, such simulation tools concentrate on statistical aspects, redesign and commercial optimization of the business process. On the contrary, we propose an approach for *on-the-fly* dynamic simulation and analysis on the basis of operational APA models detailed in section 3. This

includes consideration of the current process state and the event information combined with the corresponding steps in the process model.

6 Conclusions and Further Work

In this paper we demonstrated the application of runtime models to analyze the security status of business processes and to identify possible violations of the security policy in the near future. Therefore, we started with a business process model from the logistics domain and analyzed corresponding security requirements. Utilizing both development-time models we derived a runtime model. The runtime model consumes events from the runtime environment, evaluates current violations of the security policy, and identifies close-future violations of the security policy. Within the logistics domain we applied our approach to identify situations in which an attacker might try to disrupt or degrade the process performance. By issuing predictive security alerts, users or operators (in this case: the dispatcher in the logistic process) are able to act securely without the need to understand the security policy or infrastructure in detail.

Other novel uses of such models at runtime can enable anticipatory impact analysis, decision support and impact mitigation by adaptive configuration of countermeasures. The project MASSIF (`http://www.massif-project.eu/`), a large-scale integrating project co-funded by the European Commission, addresses these challenges within the management of security information and events in service infrastructures. In MASSIF [17] we will apply the presented modeling concept in four industrial domains: (i) the management of the Olympic Games IT infrastructure; (ii) a mobile phone based money transfer service, facing high-level threats such as money laundering; (iii) managed IT outsource services for large distributed enterprises; and (iv) an IT system supporting a critical infrastructure (dam).

Acknowledgments. The work presented here was developed in the context of the project MASSIF (ID 257475) being co-funded by the European Commission within the Seventh Framework Programme and the project Alliance Digital Product Flow (ADiWa) (ID 01IA08006F) which is funded by the German Federal Ministry of Education and Research.

References

1. Delgado, N., Gates, A., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Transactions on Software Engineering 30(12), 859–872 (2004)
2. Dijkman, R.M.: Diagnosing differences between business process models. In: Business Process Management (BPM 2008). LNCS, vol. 5240, pp. 261–277. Springer (2008)
3. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Information and Software Technology 50(12), 1281–1294 (2008)

4. Döhring, M., Zimmermann, B., Karg, L.: Flexible workflows at design- and runtime using BPMN2 adaptation patterns. In: Business Information Systems (BIS 2011), LNBIP, vol. 87, pp. 25–36. Springer (2011)
5. Fabian, B., Gürses, S., Heisel, M., Santen, T., Schmidt, H.: A comparison of security requirements engineering methods. Requirements engineering 15(1), 7–40 (2010)
6. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Future of Software Engineering. pp. 37–54. IEEE (2007)
7. Fuchs, A., Rieke, R.: Identification of Security Requirements in Systems of Systems by Functional Security Analysis. In: Architecting Dependable Systems VII, LNCS, vol. 6420, pp. 74–96. Springer (2010)
8. Gürgens, S., Ochsenschläger, P., Rudolph, C.: On a formal framework for security properties. Computer Standards & Interfaces 27, 457–466 (2005)
9. Kazhamiakin, R., Pistore, M., Santuari, L.: Analysis of communication models in web service compositions. In: World Wide Web (WWW 2006). pp. 267–276. ACM (2006)
10. Massart, T., Meuter, C.: Efficient online monitoring of LTL properties for asynchronous distributed systems. Tech. rep., Université Libre de Bruxelles (2006)
11. McCoy, D.W.: Business Activity Monitoring: Calm Before the Storm. Gartner Research (2002)
12. Melik-Merkumians, M., Moser, T., Schatten, A., Zoitl, A., Biffl, S.: Knowledge-based runtime failure detection for industrial automation systems. In: Workshop Models@run.time. pp. 108–119. CEUR (2010)
13. Mellado, D., Blanco, C., Snchez, L.E., Fernndez-Medina, E.: A systematic review of security requirements engineering. Computer Standards & Interfaces 32(4), 153–165 (2010)
14. Morin, B., Mouelhi, T., Fleurey, F., Le Traon, Y., Barais, O., Jézéquel, J.M.: Security-driven model-based dynamic adaptation. In: Automated Software Engineering (ASE 2010). pp. 205–214. ACM (2010)
15. Netjes, M., Reijers, H., Aalst, W.P.v.d.: Supporting the BPM life-cycle with FileNet. In: Exploring Modeling Methods for Systems Analysis and Design (EMMSAD 2006). pp. 497–508. Namur University Press (2006)
16. Ochsenschläger, P., Repp, J., Rieke, R., Nitsche, U.: The SH-Verification Tool Abstraction-Based Verification of Co-operating Systems. Formal Aspects of Computing 10(4), 381–404 (1998)
17. Prieto, E., Diaz, R., Romano, L., Rieke, R., Achemlal, M.: MASSIF: A promising solution to enhance olympic games IT security. In: International Conference on Global Security, Safety and Sustainability (ICGS3 2011) (2011)
18. Rieke, R., Stoynova, Z.: Predictive security analysis for event-driven processes. In: Computer Network Security, LNCS, vol. 6258, pp. 321–328. Springer (2010)
19. Rozinat, A., Wynn, M.T., van der Aalst, W.M.P., ter Hofstede, A.H.M., Fidge, C.J.: Workflow simulation for operational decision support. Data & Knowledge Engineering 68(9), 834–850 (2009)
20. Spanoudakis, G., Kloukinas, C., Androutsopoulos, K.: Towards security monitoring patterns. In: Symposium on Applied computing (SAC 2007). pp. 1518–1525. ACM (2007)
21. Tjoa, S., Jakoubi, S., Goluch, G., Kitzler, G., Goluch, S., Quirchmayr, G.: A formal approach enabling risk-aware business process modeling and simulation. IEEE Transactions on Services Computing 4(2), 153–166 (2011)
22. Winkelvos, T., Rudolph, C., Repp, J.: A Property Based Security Risk Analysis Through Weighted Simulation. In: Information Security South Africa (ISSA 2011). IEEE (2011)

Runtime Monitoring of Functional Component Changes with Behavior Models *

Carlo Ghezzi, Andrea Mocci, and Mario Sangiorgio

Politecnico di Milano, Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci, 32, 20133 Milano (MI), Italy
{ghezzi,mocci,sangiorgio}@elet.polimi.it

Abstract. We consider the problem of run-time discovery and continuous monitoring of new components that live in an open environment. We focus on extracting a formal model—which may not be available—by observing the behavior of the running component. We show how the model built at run time can be enriched through new observations (dynamic model update). We also use the inferred model to perform runtime verification. That is, we try to identify if any changes are made to the component that modify its original behavior, contradict the previous observations, and invalidate the inferred model.

1 Introduction and Motivations

Modern software systems increasingly live in an open world [6]. In the context of this paper, we assume this to mean that the components that can be used to compose new application may be dynamically discovered and they may change over time. New components may appear or disappear; existing components that were already available may change without notice. Indeed, in an open world context, software components can be developed by different stakeholders, for which there might be no control from the point of view of their clients. New applications may be developed in a way that they rely on third party components, often called *services*, that are composed to provide a specific new functionality[1]. In this setting, *models*, play the role of formal specifications and have a crucial importance. In fact, to be able to compose components in applications and make sure they achieve ascertainable goals, one needs to have a model of the components being used. Such model, in practice, mays not exist. For example, in the case where components are Web services, suitable notations (e.g., WSDL) exist to specify the syntax of service invocations, but no standard notation exists to specify the semantics (i.e., model the behavior) of the components. In this context, it becomes relevant to be able to infer a model for the component dynamically, at run time, by observing how the component behaves.

* This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

[1] Although the terms "component" and "service" can be (and should be) disinguished, in this paper the terms are used interchangeably

In addition to the previous problems, one must consider the fact that the component may change at run time in an unannounced manner. In other words, even if a model were initially provided together with the exposed service, it may become unfaithful and inconsistent because the implementation may change at run time. For this reason, in open-world context the role of models is twofold. It may be necessary to infer it initially and it becomes then necessary to use the (inferred) model at run time to verify if changes invalidate the assumptions we could make based on the initial observations.

In conclusion, in the case where the model is initially absent, we need techniques to infer a formal model (a formal specification) for the components we wish to combine. We then need to keep the (inferred) model to analyze the run-time behavior of the component and detect wether the new observed behaviors indicate that a change has occurred in the component which invalidates the model.

In this paper, we propose a technique for run-time monitoring of component changes that relies on the use of a particular class of formal models, *behavior models*. The proposed approach requires a *setup phase*, in which the component to be monitored must be in a sort of *trial phase* in which it can be safely tested to extract an initial specification. This phase integrates techniques to infer formal specifications [7] with a new kind of behavior model, the *protocol behavior model*. This model enables the main phase of the approach — a *run-time validation activity* —, which consists of monitoring the component behavior and detecting a particular class of component changes, which will be precisely described in the following sections. The approach is also able to distinguish likely new observations against component changes.

The paper is structured as follows. Section 2 presents the formalisms used in the approach, that is, the kind of behavior models that we can infer and synthesize. Section 3 describes how these models are constructed to enable the setup step of our technique, while Section 4 describes their use at runtime to detect component changes. A simplified running example is used to give a practical hint on how the approach works. Finally, Section 5 discusses related approaches and Section 6 illustrates final considerations and future work. Space limitations only made it possible to explain the innovative approach and to provide basic examples. Additional details and more complex examples are available online [2].

2 Behavioral Equivalence and Protocol Models

We consider software components as *black boxes*, that is, their internals cannot be inspected and they are accessible only through their API, which consists of operations that might modify or not the internal state. Thus, each operation can be a *modifier* or an *observer*, or it can play both roles. Operations may also have an exceptional result, which is considered as a special observable value. As a running example, we consider a simple component, called StorageService, inspired by the ZipOutputStream class of the *Java Development Kit* [1], which models a storage service where each stored entry is compressed. The component

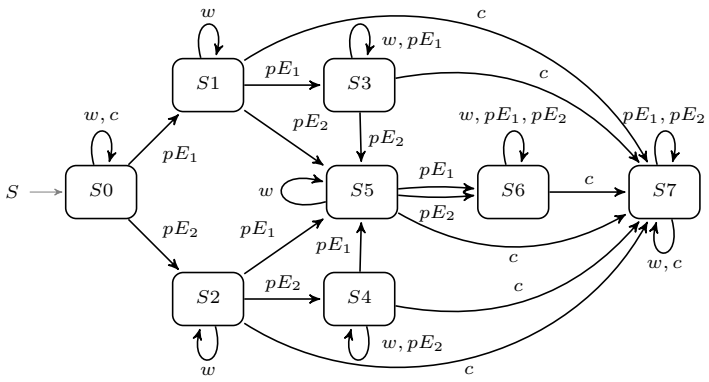
mixes container behaviors with a specific protocol of interaction. We consider the following operations: i) *putNextEntry*, which adds a new entry with a given name; ii) *write*, which adds data to the current entry; and iii) *close*, which disables any further interaction.

We now introduce the formal models used in our approach, which belong to the class of so-called *behavior models*. To accomplish the main task of the approach, that is, the runtime detection of component changes, we first need to define behavior models that they can "summarize" all the possible interactions with the software components, thus providing a description of the behaviors observed by its clients.

We start with *Behavioral equivalence models (*Bem [7]); i.e., finite state automata that provide a precise and detailed description of the behavior of a component in a limited *scope*. In a Bem, states represent *behaviorally equivalent* classes of component instances; that is, a set of instances that cannot be distinguished by any possible sequence of operations ending with an observer. Each state is labeled with observer return values and each transition models a specific modifier invocation with given actual parameters. The *scope* of the model defines the set of possible actual parameters used in the model (called *instance pool*), and the number of states we restrict to. Intuitively, these models define the component behaviors within a limited scope. Figure 1 represents a possible Bem for the StorageService component. We built it limiting the scope to two entries (e_1 and e_2) which are used as parameters for operation *putNextEntry*. Each transition represents a specific operation invocation. The table in Figure 1 describes observer return values; in this specific case, they are only exceptional results.

To describe every possible component interaction outside the Bem scope, we introduce a second kind of behavior model that generalizes the Bem through an abstraction: the *protocol behavior models (*Pbm*)*. Pbms provide an abstracted, less precise but generalized description of the interaction protocol with the component as opposed to the precise description in a limited scope provided by Bems. The new model is still based on a finite state automaton, but now states encode whether the results of observers are normal or exceptional[2]. States also abstract the behavior of modifiers as *variant* or *invariant*. A modifier behavior is variant if there exists a possible invocation with specific actual parameters that brings the component in a different behavioral equivalence state. Otherwise, the modifier behavior is invariant. This abstraction is usually (but not always) associated with an exceptional result of the operation: it is the typical behavior of a removal operation on an empty container or a add operation on a full bounded container. Pbm transitions instead keep track only of the operation they represent, ignoring the values of the parameters. Thus they model the behavior of every possible modifier invocation; they synthesize the behavior of possibly infinitely-many behavior changes induced by the possible operation invocation.

[2] If observers have parameters, then the abstraction can be i) always (i.e., for every parameter) normal; ii) always exceptional; iii) any intermediate situation, that is, for some parameters the result is normal and for others is exceptional .



Legend: *S:StorageService(), w:write(0), c:close()*
pE1:putNextEntry(e1), pE2: putNextEntry(e2)

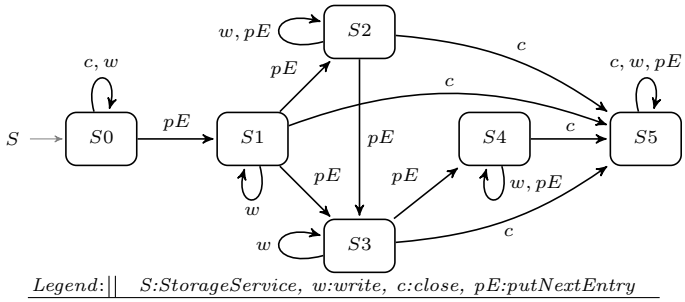
State	$close()$	$putNextEntry(e_1)$	$putNextEntry(e_2)$	$write(0)$
$S0$	$\rightsquigarrow ZipException_4$	—	—	$\rightsquigarrow ZipException_3$
$S1$	—	$\rightsquigarrow ZipException_1$	—	—
$S2$	—	—	$\rightsquigarrow ZipException_2$	—
$S3$	—	$\rightsquigarrow ZipException_1$	—	$\rightsquigarrow ZipException_3$
$S4$	—	—	$\rightsquigarrow ZipException_2$	$\rightsquigarrow ZipException_3$
$S5$	—	$\rightsquigarrow ZipException_1$	$\rightsquigarrow ZipException_2$	—
$S6$	—	$\rightsquigarrow ZipException_1$	$\rightsquigarrow ZipException_2$	$\rightsquigarrow ZipException_3$
$S7$	—	$\rightsquigarrow IOException_1$	$\rightsquigarrow IOException_1$	$\rightsquigarrow IOException_1$

$ZipException_1.getMessage() =$ *"duplicate entry: e1"*
$ZipException_2.getMessage() =$ *"duplicate entry: e2"*
$ZipException_3.getMessage() =$ *"no current ZIP entry"*
$ZipException_4.getMessage() =$ *"ZIP file must have at least one entry"*
$IOException_1.getMessage() =$ *"Stream Closed"*

Fig. 1. A BEM of the STORAGESERVICE component

In practice, they model the possibility that by performing an operation the set of operations enabled on the object may change. It is worth observing (but we do not show it here) that this abstraction may introduce nondeterminism in the automaton. Figure 2 represents the PBM derived by performing the abstraction described above to the BEM in Figure 1.

The main contribution of the proposed approach is the integration of PBMs and BEMs. Because the PBM is derived from the BEM through an abstraction process, its completeness and accuracy actually depends on the significance of the observations that produced the BEM during the setup phase. The setup phase is deeply rooted in the *small scope hypothesis*. In its original formulation [9], this hypothesis states that *most bugs have small counterexamples*, and that an exhaustive analysis of the component behavior within a limited scope is able to show most bugs. In our case, we cast it as follows: most of the significant behaviors of a component are present within a small scope. In our case, the term "significant behavior" refers to the abstracted version provided by a PBM. Thus, we expect that at setup time we can synthesize a likely complete PBM,



Legend: *S:StorageService, w:write, c:close, pE:putNextEntry*

State	*close*	*putNextEntry*	*write*
	Observer Abstraction		
$S0$	$\rightsquigarrow$ *ZipException*	—	$\rightsquigarrow$ *ZipException*
$S1$	—	[−, $\rightsquigarrow$ *ZipException*]	—
$S2$	—	[−, $\rightsquigarrow$ *ZipException*]	$\rightsquigarrow$ *ZipException*
$S3$	—	$\rightsquigarrow$ *ZipException*	—
$S4$	—	$\rightsquigarrow$ *ZipException*	$\rightsquigarrow$ *ZipException*
$S5$	—	$\rightsquigarrow$ *IOException*	$\rightsquigarrow$ *IOException*
	Modifier Behavior Abstraction		
$S0$	*Invariant*	*Variant*	*Invariant*
$S1$	*Variant*	*Variant*	*Invariant*
$S2$	*Variant*	*Invariant*	*Invariant*
$S3$	*Variant*	*Variant*	*Invariant*
$S4$	*Variant*	*Invariant*	*Invariant*
$S5$	*Invariant*	*Invariant*	*Invariant*

The notation: [−, $\rightsquigarrow$ ZipException] means that for some parameter the method returns correctly (—), and for some other parameter throws ZipException

Fig. 2. A PBM of the STORAGESERVICE component

which describes the protocol of all the possible interactions of clients with the component, while at runtime we can use the PBM to find component changes.

The two different models can be used together at run time. The behavior of a component is monitored and checked with respect to the PBM. When violations are detected, a deeper analysis exploiting the more precise information contained in the BEM can be performed in order to discover whether the observation that is not described by the PBM is a new kind of behavior that was not observed before, and thus requires a change of both the BEM and the PBM to accommodate it, or instead it detects a component change that is inconsistent with the models. The BEM synthesizes the observations used to generate the PBM, and thus it can be used to distinguish between likely changes of the analyzed component from new observations that instead just enrich the PBM. In the following sections, we will discuss these aspects: the setup time construction of BEMs and PBMs, and the runtime use of both models to detect likely component changes.

It should be noted that the PBM is not a full specification of the component, thus it cannot be used to express complex functional behaviors, in particular the ones that are not expressible with a finite state machine, like complex container behaviors. Instead, the PBM models the protocol that clients can use to interact with the component, that is, the legal sequences of operations. This limitation is also the enabling factor for runtime detection of changes: violations can be

checked and detected relatively easily and the model can be promptly updated when needed. Instead, a full fledged specification that supports infinite state behaviors, like the ones of containers, is definitely harder to synthesize, check and update at runtime.

3 Setup Phase: Model Inference

As we illustrated previously, the approach we propose prescribes two phases. The setup phase is performed on the component in a trial stage. The other phase corresponds to runtime. In the former, the component is analyzed through dynamic analysis (a set of test cases) to infer a BEM for the component. A PBM abstraction is generated next to generalize the observed behaviors. In the latter phase, the two models are used at run time to detect component changes. In this section, we describe the first phase, with particular focus on the generation of models, so that designers can get a formal description of a component whose behavior must be validated.

3.1 Generation of the Initial Behavioral Equivalence Model

To generate a BEM during the setup phase, we adapt the algorithm and the tool described in [7], which extracts BEMs through dynamic analysis. The model is generated by incrementally and exhaustively exploring a finite subset of the component behavior, that is, by exploring a small scope. As illustrated previously, the scope is determined by a set of actual parameters for each component operation and a maximum number of states for the model. The exploration is performed by building a set of traces using the values in the instance pool and abstracting them to behavioral equivalence states. The exploration is incremental; that is, if a trace t is analyzed, then all its subtraces have been analyzed in the past. To build the BEM, the approach first uses observer return values: for a trace t and every possible observer o, with fixed actual parameters, we execute $t.o()$ and we build a state of the BEM labeled with observed return values. Unfortunately, such an abstraction does not always induce behavioral equivalence: for example, it could be that for some operation m, then two traces t_1 and t_2 such that for every observer the return values are equal, then there could be that $t_1.m()$ and $t_2.m()$ are not behaviorally equivalent. Thus, state abstraction is enriched with the information given by m as a discriminating operation. For space reasons, we cannot include the specific details of the algorithm, but the interest reader can refer to [7, 11]. This approach guarantees the discovery of all the behaviors presented in the class with the given scope. The way BEMs are generated implies a strong correlation between the quality of the model and the completeness of the instance pools used to build it. The more the instances are significant, the higher the coverage of the actual behavior of the class is.

Given the importance of the objects used to perform the BEM generation phase, we want to exploit as much as possible all the knowledge available to analyze the class behavior with the most significant set of instances. The original

Spy tool relied entirely on instances provided by the user interested in obtaining the Bem of a component. While the assumption that the user is able to provide some significant instances is fair, it may be hard to achieve since it requires a lot of effort and a deep knowledge of the behaviors of the component. Fortunately, in practice the vast majority of the classes comes with a test suite containing exactly the operation calls with some significant instances as parameters.

The extraction of the significant instances is performed by collecting from the test suite all the objects passed as arguments in an operation call. Each value is then stored in an instance pool that collects all the values for each parameter. The values of the instances are stored directly in the instance pools, ready to be used in the exhaustive search. Instances collected from the test suite are very useful, but it happens that they may be redundant. To avoid the generation of models with a lot of states that do not unveil new behaviors, we should filter out the instances collected in order to keep a minimal subset able to exercise all the possible behaviors of the component without having to deal with a huge model. At this stage of the development, the tool is able to extract instances from a test suite but does not select the minimal subset of instances. This task is left to the user who has to find the best trade-off between the number of instances used for the analysis and the completeness of the Bem generated.

3.2 Synthesis of the Protocol Behavior Model

Once the Bem is generated we can go further with the analysis and generate the corresponding Pbm. Generation is quite straightforward since the Bem already includes all the needed information about the outcome of each operation in each state of the model. Pbm inference algorithm consists of the following steps: i) generalization of the Bem states through the Pbm abstraction function; ii) introduction of each Bem transition in the Pbm. The generalization of the information contained in a Bem state is performed by applying to each state of the Bem the Pbm abstraction function we discussed earlier. Then for each transition of the Bem we add a transition to the Pbm starting from the representative of the equivalence class of the starting node in the Bem and ending in the representative of the destination node. Because parameters are ignored in the abstraction, the transformation is likely to produce a non-deterministic automaton: it may happen that, given a Pbm state, the invocation of the same operation with different values of the parameters produces different outcomes that turns into different destination states.

4 Runtime Phase: Monitoring and Change Detection

Bems and Pbms are used to perform runtime verification that the component, which may evolve independently, behaves accordingly to its models. To do so, we monitor the execution of an application and detect changes in the behavior of its components. Being able to detect changes is crucial when the application in

which the component is embedded has to be self-adaptive and react to component misbehaviors in an appropriate way. More precisely, in this section we show that the data collected at runtime can be used on the one hand to enrich the model with previously unobserved behaviors and on the other hand to highlight behavioral differences unveiling changes in the component under analysis.

4.1 Monitoring

A monitor is introduced into the running system to allow the comparison of the actual behavior of the component under analysis and the ones encoded by the models. Each time an instance of the scrutinized class is created, it is associated to it a monitoring process in charge of recording the observed execution trace and analyzing it to discover violations with respect to the protocol described by the model. Violation detection is performed by comparing the actual behavior with the one encoded in the model. Therefore it has to gather enough information to determine the state in which the component is. The system reports a violation when it finds an exceptional outcome for an operation that, according to the model, should always terminate normally or, on the opposite, when an operation that the model describes as exceptional does not throw anything.

In order to maintain the lowest overhead possible on the system under analysis, the violation detector relies, when possible, exclusively on the observed trace. When the PBM has only deterministic transitions this process is straightforward and violations can be detected directly from the execution trace. Unfortunately, almost all components with a complex behavior are non-deterministic so there is the need of a deeper inspection by executing operations that could provide more information and thus reveal the state in which the component is. The solution proposed in this paper is an enhanced monitoring phase, not relying exclusively on what it is observable from the current execution but also able to perform some more queries to the object under analysis. For any state having non-deterministic outgoing transitions, we can determine which are the operations that make it possible to know which one has been taken. These discriminators are the operations on the destination states having different behaviors. Nondeterminism can therefore be solved by invoking the discriminating operations on the object under analysis. With these additional operations it is easy to find the compatible state among the different nondeterministic possibilities. Discriminating operations have to be invoked on an instance of the object exactly in the same state of the actual component and should be tested with both the original instance pool and the instances observed in the trace for the operation. The original instance pool alone would not be so effective as it would make it impossible to find behaviors related to the parameter re-use.

Clearly it is not possible to call additional operations on the actual component under analysis. It would lead to interferences with the service provided by the system. Modifiers have undesirable side effects but also the invocation of a *pure* operation could introduce delays and reduce the quality of service. In order be able to carry on the analysis without disrupting the offered service, we need to assume that we can create a *clone* of the component behaving exactly in the

same way the actual instance does. Moreover, the operations performed on such clone do not have to change the actual component environment. These assumptions reduce the number of components our methodology can deal with, but we are still able to monitor and analyze a vast class of commonly used elements.

Therefore the monitoring architecture requires: i) to instrument the application using the external services; ii) to have the possibility to call operations on a sandboxed instance of the service; iii) to be able to replay execution traces in order to put the sandboxed instance in a defined state. With such an infrastructure, the verification module can detect changes in the behavior of external services without interfering with the actual execution of the system.

4.2 Response to Violations

During the monitoring phase it may happen that an observation on the actual execution conflicts with what it is described by the model. There are two possible causes for the violation observed: the model could be incomplete, and therefore needs to be updated, or the behavior of component has changed. The analysis phase has to be able to deal and react properly to both these situations.

It is possible to discover whether the violations is due to the incompleteness of the PBM or to a change in the behavior by replaying on the *clone* some significant executions encoded in the BEM. If all of them produce again the previously observed results, then the model needs to be completed and that violations simply indicate behaviors never explored before. Otherwise the violation signals a misbehavior of the component that should trigger a reaction aimed at bringing back the system in a safe state.

In order to keep the approach feasible, we cannot just test that everything described by the BEM is still valid. We should rather focus on the part of the model more closely related to the observed violation. The first step in the selection of the relevant execution traces is the identification of the BEM states corresponding to the state of the PBM in which the violation occurred. The initial part of the test cases can then be generated by looking for the shortest execution traces able to reach the selected BEM states. The traces obtained in that way have then to be completed with the operation that unveiled the violation. For any BEM state the operation has to be called with all the parameters present in the instance pool used to generate the model.

Model updates have to be performed when the monitoring tool discovers a violation but there is no evidence of behavioral change. Models are updated because the behavior of the observed execution does not contrast with what has been observed in the past. Model updates are first applied to the BEM and then to the corresponding PBM.

Updating the BEM means enriching the scope it covers with the trace unveiling the new behavior. Keeping all the information in a single BEM would make its dimension increase, so we decided to rely on a set of BEMs, each one describing a behavior of the component on the particular scope showing it. Doing that,

we can easily keep track of all the relevant execution exposing the different behaviors. Although doing that we may miss some behavior due to the interaction of the behaviors described by different BEMs, this is not an issue: the model will describe them as soon as they appear at run time. From the set of BEMs it is easy to get the corresponding PBM. It is quite straightforward to adapt the inference algorithm described in section 3 to deal with the information contained on more than a behavioral model: the algorithm has to be applied to each BEM and the data gathered have to be added to the same PBM so that it contains information about all the observed behaviors regardless of the BEM it comes form. To produce correct abstractions for the new PBM, all the BEMs must have a coherent set of observers. To ensure that, we must update the scope for the observer roles in the already existing BEMs to have them take into account all the significant values of the parameters discovered at run time.

A violation requiring to update the models of STORAGESERVICE reported in figures 1 and 2 happens when we try to write an empty string of data when no entry is available. In such situation, the expected *ZipException* is not thrown because the *write* operation does not have to write anything and the component does not check for the availability of an entry. Therefore, we need to add a BEM containing the observed trace. Since the violating trace contained a previously unseen instance, we also have to update the existing BEM to have it consider the empty string as a parameter for the *write* operation. For space limitations the updated models and other examples are only available online at [2].

Change detection takes place when there is at least one test case behaving differently than what the PBM prescribes. Since the model encodes the behaviors observed in the past, any violation can be considered as an evidence of a change: at least in the case highlighted by the failing test, the same execution trace presented a different behavior than the one assumed by the model. The system has then to react to the behavioral changes detected. We identified two possible scenarios in order to be able to guarantee the maximal safety though trying to limit the number of service interruptions. The safer scenario presents a change that just turns one or more operation call with an exceptional outcome into invocations that terminates normally. Another possible and more critical situation affects more deeply the enabledness of the different operations and so requires a stricter reaction to ensure the safety of the system.

In the first case, the change has to be notified but it does not require to stop the execution of the application. The detected change is probably just an addition of new functionalities or interaction patterns that previously were not present or were disabled. However, for safety reason it is better to leave to the user the final decision about how to react to this kind of behavioral changes. More serious problems may arise form behavioral changes that turns the outcome of an operation from normal to exceptional. Such a change makes it impossible to *substitute* the new component to the one the system is expecting to deal with. At some point there may be an invocation to the operation that changes its behavior and it is going to always produce a failure due to the exception

thrown. For this reason, when changes like this occur, the only safe solution is to stop the execution of the system requiring the intervention of a supervisor able to decide how to fix the problem.

Change detection can be demonstrated using again the models reported in Figure 1 and 2 to monitor the behavior of a StorageService. For a very simple example we can assume that the component stops working and changes its behavior to always throw an exception every time *putNextEntry* is invoked. In this scenario, any execution of *putNextEntry* now violates the Pbm. We are interested to check if the violation is specific to the trace observed or it is a component change; to check this, we derive the simple test case *StorageService().putNextEntry(e_1)* from the Bem. Since this test case violates the Bem, it highlights the change of the behavior of the component.
A more comprehensive evaluation of the effectiveness of the change detection methodology has been performed injecting faults into the component under analysis and is available online at [2].

It is important to remark that this methodology is able to identify changes only when there is at least one failing test case in the ones that can be derived from the Bem. Since the model does not contain information about every possible execution it is possible that an actual change is detected as a case in which there is the need to update the Bem and the Pbm because they does not contain anything about that particular case. However, since the updates to the model have to be reviewed by the designer of the system the procedure prescribed by our methodology can be considered effective.

5 Related Work

The protocol models discussed in this paper describe the behavior of a software component accordingly to whether its operations are enabled or not. The underlying idea has been introduced with the concept of Typestate in [13]. A similar abstraction has also been used in [5], which presents a technique to build an enabledness model from contracts and static analysis.
Tautoko [4] generates similar models starting from an existing test suite. Our tool does not infer the model directly from the test execution traces. It rather exploits the test suite to gather some domain knowledge to use with the Spy methodology.
Monitoring of both functional and non-functional properties of service-based systems are described in [3]. Our technique is based on Pbms and Bems, therefore we are able to model and monitor very precisely functional properties of a software component.
Invite [12] developed the idea of *runtime testing*, pointing out the requirements the running system has to satisfy in order to make it possible. In this work we also introduced a technique to encode test cases in Bems and to select the ones can highlight a behavioral change.
Tracer [10] builds runtime models from execution traces enabling richer and more detailed analysis at an higher abstraction level. In [8] models are used to

monitor system executions and to detect deviation from the desired behavior of consumer electronic products. Our approach combines these two aspects providing a methodology able to both detect violations and build models according to the information gathered at run time.

6 Conclusions and Future Work

Behavior models can be useful throughout all the lifecycle of a software component. Traditionally, such models are used at design time to support system designers in they choices. However, they can also play a significant role after the application is deployed by monitoring its execution and checking system properties. That is particularly useful in the context of systems in which verification must extend to run time, because of unexpected changes that may occur during operation.

This work focuses on the runtime aspects, extending the original scope of behavior models to running systems. The models and methodology proposed can maintain an updated representation of the behavior of the component considering observations made during the actual execution of a running system. Our approach is also able to detect and notify the system designer behavioral changes in the monitored components. Preliminary experiments show that our approach is effective and can deal with non-trivial components. Further research is going to enhance the models removing current limitations and thus making it possible to monitor an even broader class of software components.

References

1. Oracle, java se 6.0 doc., 2011. `http://download.oracle.com/javase/6/docs/index.html`.
2. Spy at runtime. http://home.dei.polimi.it/mocci/spy/runtime/, 2011.
3. L. Baresi and S. Guinea. Self-supervising bpel processes. *IEEE TSE*, 2011.
4. V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *ISSTA '10*, Trento, Italy, 2010.
5. G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Automated abstractions for contract validation. *IEEE TSE*, 2010.
6. E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *ASE*, 2008.
7. C. Ghezzi, A. Mocci, and M. Monga. Synthesizing intensional behavior models by graph transformation. In *ICSE '09*, Vancouver, Canada, 2009.
8. J. Hooman and T. Hendriks. Model-based run-time error detection. In *Models@run.time '07*, Nashville, USA, 2007.
9. D. Jackson. *Software Abstractions:Logic,Language,and Analysis*. MIT Press, '06.
10. S. Maoz. Using model-based traces as runtime models. *Computer*, 2009.
11. A. Mocci. *Behavioral Modeling, Inference and Validation for Stateful Component Specifications*. Ph.D. thesis, Politecnico di Milano, Milano, Italy, 2010.
12. C. Murphy, G. Kaiser, I. Vo, and M. Chu. Quality assurance of software applications using the in vivo testing approach. In *ICST '09*, Denver, Colorado, 2009.
13. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, 1986.

Using Model-to-Text Transformation for Dynamic Web-based Model Navigation

Dimitrios S. Kolovos, Louis M. Rose, and James R. Williams

Department of Computer Science, University of York,
Deramore Lane, York, YO10 5GH, UK.
{dkolovos,louis,jw}@cs.york.ac.uk

Abstract. One of the main objectives of modelling is to enable collaborative decision making and communication among the stakeholders of the system. It is essential that both technical and non-technical stakeholders can access and comment on the models of the system at any time. In this paper we propose a model-to-text transformation approach for producing dynamic, web-based views of models – captured atop different modelling technologies and conforming to arbitrary metamodels – so that stakeholders can be provided with web-based, on-demand and up-to-date access to the models of the system using only their web browser. We demonstrate the practicality of this approach through case studies and identify a number of open challenges in the field of web-based model management.

1 Introduction

One of the main objectives of modelling is to enable collaborative decision making and communication among the stakeholders of the system – particularly so in the early stages of the software development lifecycle. To this end, stakeholders must be able to access – possibly different parts of – the *models* constructed by the designers of the system. In principle, the simplest way to achieve this is to establish a centralised repository where designers share the models that they construct with other stakeholders, so that the latter can navigate the models using the same modelling tools that the designers used to create them. However, experience obtained from interacting with our industrial partners – some of which was summarised in an earlier paper [1] – suggests that this is not always feasible or desirable, for a number of reasons:

- **Cost:** Purchasing licenses of expensive modelling tools only to view and provide feedback on the models of the system may be impractical or too expensive.
- **Time:** In some industrial environments, installing new software requires a formal approval process which can take significant time to complete.
- **Complexity/Training:** Modelling tools are typically complex because they accommodate the needs of software designers. As such, training is typically required for non-expert users, even to support them in relatively straightforward tasks.

– **Access control:** It may be desirable that some stakeholders are only granted access to some parts of the models.

We have encountered the above issues in the context of ongoing work with one of our major industrial collaborators. To address such issues in practice, modellers typically construct word processor *design documents* which are then disseminated to the stakeholders, who therefore are not required to purchase, install and learn to use any new software. However, this approach has known shortcomings. Assembling and synchronising such documents can be labour-intensive and error-prone, particularly if they are not supported natively by the modelling tool. Moreover, for large models, designers need to create correspondingly large documents, which ultimately become difficult to navigate and read. Finally, and as a consequence of the other shortcomings, such design documents quickly become out-of-date and do not reflect the current version of the models.

To eliminate the overhead of creating and distributing snapshots of the current versions of models in the form of design documents, we propose a model-to-text transformation approach for producing dynamic web-based views of models – captured atop a range of different modelling technologies – so that stakeholders can be provided with web-based, on-demand and up-to-date access to the models of the system from their web browser and without needing to purchase or install any additional tooling.

The remainder of the paper is organised as follows. Section 2 introduces the proposed transformation-based approach and discusses the details of the technical solution we have developed to realise it. Section 3 provides two case studies that demonstrate using the proposed approach to implement web application for navigating and animating state machine models, and for browsing Ecore metamodels in a Javadoc-like fashion. Section 4 discusses related work and Section 5 concludes the paper and provides interesting directions for further work on the subject.

2 Dynamic Web-Based Model Navigation

To overcome the shortcomings identified above, we propose an approach that allows stakeholders to have direct, on-demand and up-to-date access to the (parts of the) models in which they are interested. Additionally, the approach proposed in this section – like the design document approach – does not require stakeholders to purchase, install, or use any additional software other than their web browser.

To enable web-based access to models that have been defined atop a range of modelling technologies, in this work we have integrated the Epsilon Generation Language [2], which is a template-based model-to-text transformation language, with a Java-based servlet container and web-server (Tomcat). Like most model-to-text transformation languages, EGL was originally designed to support batch code generation; in this work, by integrating EGL with Tomcat, we can use EGL templates as server-side scripts to generate HTML content from models on demand, as displayed in Figure 1.

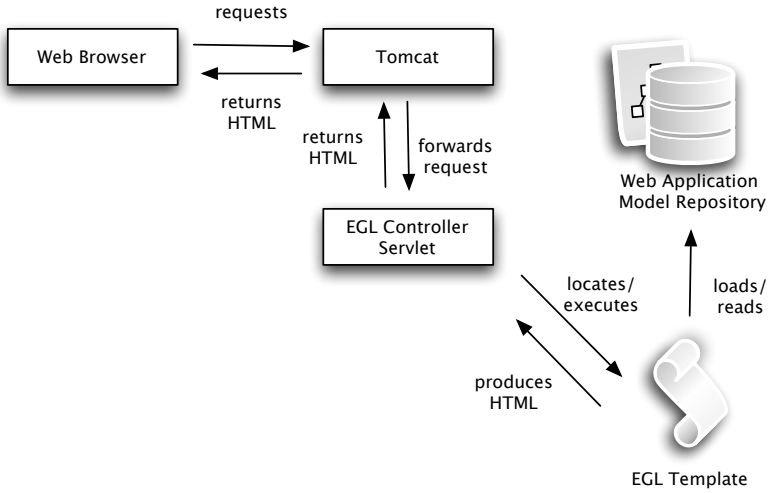


Fig. 1: EGL as an on-demand server-side scripting language in Tomcat

In the following sections we provide a brief overview of EGL and its underlying infrastructure and discuss the process and challenges involved in using it as a server-side scripting language. Before detailing the technical aspects of our work, we stress that, although in this work we use EGL and Tomcat as supporting technologies for our implementation, the proposed approach is not bound to a specific model-to-text transformation language or web-server. In principle, it can be implemented using any other model-to-text transformation language, such as Xpand[3], MOFScript[4] and Acceleo[1]. We selected EGL and Tomcat due to our technical expertise with them.

2.1 The Epsilon Generation Language

EGL is a template-based model-to-text transformation language implemented atop the Epsilon model management platform [5]. Epsilon provides a layered architecture that enables the construction of interoperable task-specific model management languages for tasks such as model transformation, validation, comparison, merging and refactoring. To enable model management languages built atop it to manage models captured using different modelling technologies, Epsilon provides an abstraction layer called EMC (Epsilon Model Connectivity) which specifies an API against which *drivers* for different modelling technologies are implemented. To date, EMC *drivers* for technologies such as EMF, MDR, Z (through CZT [6]), and plain XML have been implemented – and as such, all

[1] www.eclipse.org/acceleo

model management languages in Epsilon can manage models captured with all of these technologies. A more detailed discussion on EMC is available in [7].

As EGL is implemented atop Epsilon, it is interoperable with all of the modelling technologies listed above. Therefore, although in this work we demonstrate using EGL to build web applications around EMF-based models, any of the supported modelling technologies can also be used.

2.2 Integration with Apache Tomcat

Apache Tomcat is a widely used web server and Java servlet container. Tomcat comes with built-in support for the Java Server Pages (JSP) server-side scripting language for producing dynamic web pages, but also provides a flexible architecture which allows developers to extend it with support for additional server-side languages. To integrate EGL with Tomcat, we added a new *servlet mapping* that instructs Tomcat to redirect all requests for URLs that end with *.egl* to a dedicated controller servlet that is responsible for processing these requests.

As illustrated in Figure 1, when Tomcat receives a request for a URL that ends with *.egl*, it forwards the request to the EGL controller servlet which in turn locates and parses the respective EGL template, and if no errors occur during parsing, it executes the template and returns the produced text to Tomcat – which finally returns it to the browser. Similarly to the majority of server-side programming languages, EGL templates have access to a number of predefined variables for accessing *request* parameters and setting/getting *session* and *application* properties. Moreover, since the expression language on which EGL builds can reflectively access Java objects, EGL templates can interoperate seamlessly with existing Java libraries, and can be used in the context of frameworks such as Apache Struts[2], which facilitates the creation of J2EE applications.

Accessing Models To minimise the overhead of loading and storing models in individual EGL templates, each web application is provided with a dedicated model repository which EGL templates can access – to load and store models – via the built-in *modelManager* variable. The application model repository caches models so that they can be readily accessed by all of the EGL templates in an application. Templates have read/write access to the models in the repository; however, the ability for multiple users to modify models in the repository concurrently depends on whether the underlying modelling technology is thread-safe or not. In the current EGL–Tomcat integration, only support for EMF models has been implemented, and as EMF is not thread-safe, all of the applications that we have constructed so far have read-only access to the underlying models.

Template Factories EGL provides several types of built-in template [2]. For example, *EglTemplate* is used for generating plaintext and *EglFileGeneratingTemplate* for generating files on disk. Templates are accessed via the built-in *TemplateFactory* variable. Extenders of EGL can also specify their own template

[2] http://struts.apache.org/

types and their own template factories for capturing and re-using other code generation logic. For instance, the example described in Section 3.1 uses a custom template type and factory to produce an image file from the text generated by a template.

The dedicated EGL servlet queries the metadata of each web application to determine which type of template factory will be used to execute the templates of that application. Users can specify the fully-qualified Java class name of the template factory for their application as a parameter to the EGL servlet definition.

Caching To facilitate scalability of the applications developed atop the approach described in this section, the EGL servlet provides two types of caching, which can be used in web-based EGL applications. Caching is achieved from EGL templates via a built-in *cache* object.

Page Caching allows repeated requests for the same URL to be served without invoking any EGL templates. The EGL controller servlet maintains a cache that maps requests (URLs) to responses (the HTML generated by invoking an EGL template). A response is cached the first time that it is requested, and subsequent requests for to same URL are served from the cache. Applications can control which requests should be cached via the built-in *cache* object. Similarly, a URL can be marked as expired using the built-in *cache* object, and the next request for that URL will be served by invoking an EGL template rather than from the cache.

Fragment Caching allows unchanging page elements – such as headers and footers – to be shared between requests for different URLs, and requires that the EGL application be decomposed into separate subtemplates. The EGL controller servlet maintains a cache that maps subtemplates to partial responses (part of the HTML generated for a request to a particular URL). As with page caching, applications can control the fragment caching strategy via the built-in *cache* object, which provides methods for caching and expiring fragments. The case study in Section 3 uses fragment caching to cache headers, footers and sidebars.

Page caching results in less server-side processing than fragment caching, but is more brittle. For example, consider the effects of changes to a model on page and fragment caches. Pages and fragments that have been affected by changes to a model must be expired (removed from the cache). In the worst case then, the page and fragment caches must be completely emptied to ensure that stakeholders can view and navigate the updated model. The page cache will become fully populated only when every page of the web application is visited. By contrast, fragments for say, shared headers and footers, are cached by a request to any page, and future requests to any other page benefit from the cached fragment.

3 Case Studies

In this section, we present two case studies that demonstrate using the proposed approach for browsing behavioural and structural models. The first case study employs the proposed approach to visualise state machines and concentrates on automated diagram generation. The second case study illustrates a web application for navigating Ecore metamodels in a Javadoc-like style, through which we demonstrate the caching mechanisms discussed in Section 2.2 and evaluate the scalability of our implementation.

3.1 Visualisation and Animation of State Machine Models

Figure 2 shows the metamodel for a Mealy machine. A Mealy machine is finite-state machine that produces an output string in response to an input string. Each transition between states matches a character from the input string and generates an output character when fired. A Mealy machine must have one initial state and the execution ends when the input string has been completely parsed.

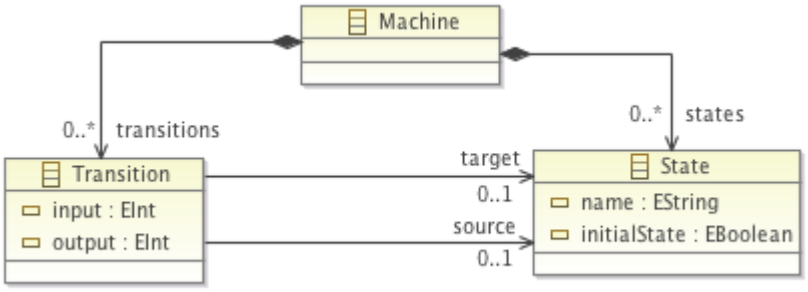


Fig. 2: The metamodel for a Mealy machine

In order to facilitate animation of the Mealy machine, asynchronous calls to two different server-side EGL scripts are required – one to generate an image of the model's current state, and another to request the output of the machine upon firing the selected transition. The user initially requests a standard HTML page, which, upon loading, makes a request to `MealyHandler.egl` to generate an image of the machine in its initial state. Clicking on a successor state will cause another request to `MealyHandler.egl` to return the image representing the machine in the new state, and will also make a request to `MealyOutput.egl` to display the results of executing that transition.

Before discussing the two templates used to animate the Mealy machine, we first briefly summarise the structure of a typical EGL template. EGL templates comprises dynamic sections, which contain executable code, and static sections, which contain text to be emitted. Consider Listing 1.2. Dynamic sections, such

```
1  [% modelManager.registerMetamodel('MealyMachine.ecore');
2   modelManager.loadModel('Example', 'My.mealymachine', 'MealyMachine');
3
4   var currentState = null;
5
6   if (request.getParameter('st').isDefined()) {
7    currentState = request.getParameter('st');
8   } else {
9    currentState = State.all.select(s|s.initialState==true).first().name;
10  }
11
12  var dotTemplate : Template := TemplateFactory.load('Mealy2Dot.egl');
13
14  dotTemplate.populate('currentState', currentState);
15  dotTemplate.generate("mealy_" + currentState + ".svg"); %]
```

Listing 1.1: Generating an SVG image of the Mealy machine

```
1  digraph G{
2   center=true;
3   nodesep=1.5;
4
5   node [ shape=circle, fontname=Tahoma, fontsize=10 ];
6    [%for (s in State.all) { %]"[%=s.name%]";[% }%]
7   "[%=currentState%]" [ color=deepskyblue, style=filled, fillcolor=lightgrey
        ];
8
9   [% for (t in Transition.all.select(t|t.source.name=currentState)) {%]
10   "[%=t.target.name%]" [color=deepskyblue, href="javascript:top.
         doTransition('[%=currentState%]', '[%=t.target.name%]')" ]
11  [%}%]
12
13  [% for (t in Transition.all) {%]
14   "[%=t.source.name%]" -> "[%=t.target.name%]" [ label="[%=t.input%]/[%=t.
         output%]" ]
15  [%}%]
16 }
```

Listing 1.2: Generating the Graphviz representation of the Mealy machine

as the one on line 9, are enclosed in `[% %]` tags; static sections, such as the one on lines 1-5, are not enclosed in `[% %]` tags. Dynamic sections can take an alternate form, using a `[%= %]` tag (such as the one on line 7) to emit a dynamically computed value (the `currentState` variable in this case).

`MealyHandler.egl` (listing 1.1) generates the image of the machine using `Mealy2dot.egl`. The call to `generate` (line 15) delegates to an image generating template factory, which invokes DOT to convert the output of `Mealy2dot.egl` to an SVG (Scalable Vector Graphics) image.

`Mealy2dot.egl` (listing 1.2) outputs a Graphviz[3] DOT language description of the machine and the template factory executes the DOT description, creating an SVG file. Animation is achieved by passing the current state name as a parameter in the URL query string to `MealyHandler.egl` (e.g. `MealyHandler.egl?st=s0`) which populates the `currentState` variable in the

[3] www.graphviz.org

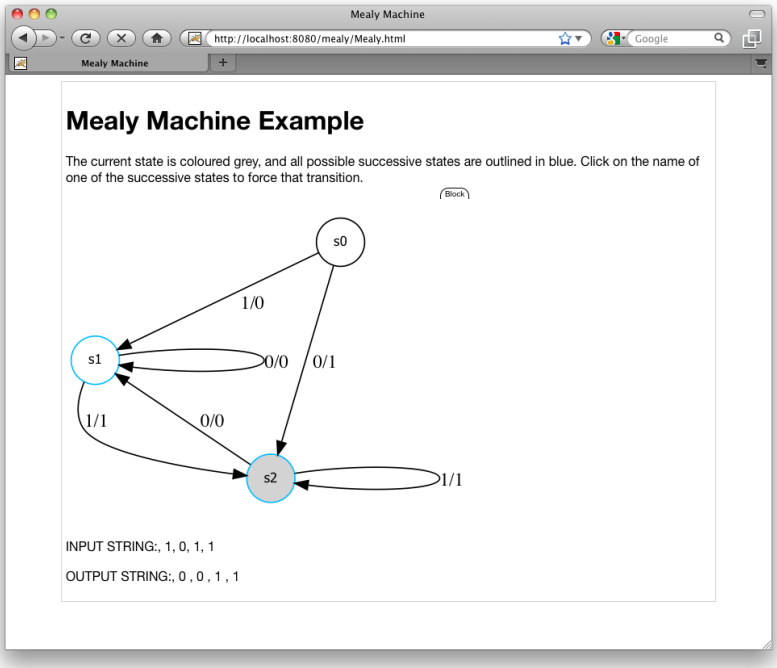


Fig. 3: Example Mealy machine animation

`Mealy2dot.egl` template (listing 1.1, line 14). `Mealy2dot.egl` highlights the current state by changing its background colour, and assigns URLs to any nodes reachable by outgoing transitions from the current state, making them clickable in the generated image. The URLs execute a JavaScript function, `doTransition(src,tgt)`, which handles the asynchronous request to generate the new image and replace the existing image with the newly generated one.

After the asynchronous request to generate the image has been made, `doTransition()` also makes an asynchronous request to `MealyOutput.egl` which returns a string containing the input and output strings from the transition. These are then printed underneath the image, showing the input and output history of the animation. Figure 3 shows the output of animating a Mealy machine over four transitions.

3.2 Navigation of Ecore metamodels

This section presents a web-application for on-demand navigation of Ecore metamodels in a Javadoc-like style and explores the way in which the scalability of the proposed approach is affected by the server-side processing load and the type of caching employed. The results presented in this section suggest that the

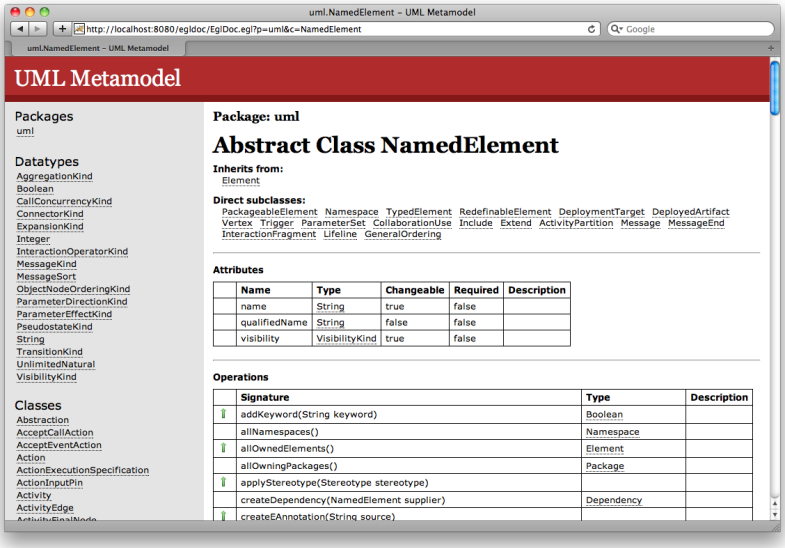


Fig. 4: EglDoc for the NamedElement class of UML 2.2 [9].

approach proposed in Section 2 is well-suited to viewing and navigating models via the web for a significant number of concurrent users.

EglDoc: Metamodel Documentation This section demonstrates the approach proposed in Section 2 by describing the way in which an existing EGL application, EglDoc [8, Ch.5], has been ported to provide web-based model navigation and view extraction. EglDoc generates HTML documentation for Ecore metamodels (and is similar to Javadoc for Java in this respect). Figure 4 shows the output produced by EglDoc for the `NamedElement` class of the UML 2.2 [9] metamodel. EglDoc can be applied to produce documentation for any Ecore metamodel.

EglDoc comprises several templates, which are used to generate the header, footer, navigation bar and content of each page. An extract of the EGL template that generates documentation for attributes is shown in Listing 1.3. EGL is a template-based model-to-text language. Listing 1.3 generates an HTML table of attributes, listing the name and type of each attribute. Lines 10-12 generate a link to other pages, using the `toUrl()` operation on lines 19-21.

The existing version of EglDoc generates one HTML file for each element of the Ecore metamodel. Porting EglDoc to interoperate with the web-based model navigation and view extraction approach proposed in Section 2 involved parameterising the existing EGL templates to facilitate the identification of metamodel elements via the URL of a request to the web server. For example, a request for

```
[% if (class.eAllAttributes.size() > 0) { %]
  <h4>Attributes</h4>
  <table cellspacing="0">
    <tr> <th>Name</th> <th>Type</th> </tr>
    [% for (attribute in class.eAllAttributes.sortBy(a|a.name)) { %]
      <td>[%=attribute.name%]</td>

      [% if (attribute.eType.isDefined()) { %]
         <td>
           <a href="[%=attribute.eType.toUrl()%]">
             [%=attribute.eType.name%]
           </a>
         </td>
      [% } else { %]
         <td> </td>
      [% } %]
  [% } %]
[% } %]
[% operation EClassifier toUrl() : String {
   return self.ePackage.name + '-' + self.name + '.html';
} %]
```

Listing 1.3: View extraction for attributes in EGLDoc

```
[%
   modelManager.registerMetamodel('Ecore.ecore');
   modelManager.loadModel('Sample', 'UML.ecore', 'http://www.eclipse.org/emf
        /2002/Ecore');

   if (request.getParameter('p').isDefined()) {
     package = EPackage.all.selectOne(p|p.name = request.getParameter('p'));
   }
%]
```

Listing 1.4: Model loading in EGLDoc

the URL `EglDoc.egl?p=uml&c=NamedElement` to the web-based EglDoc returns HTML for the `NamedClass` class of the `uml` package (in the UML 2.2 metamodel). Listing 1.4 shows the way in which the UML metamodel is loaded (lines 2-3) and the `p` parameter of the request is interpreted (lines 6-8). Note the use of the *modelManager* built-in variable for loading a model, which was discussed in 2.2. In Listing 1.4, the model to be loaded (`'UML.ecore'`) is hard-coded for clarity. In practice, the location of the model is specified as a URI, which can be configured by the user.

4 Related Work

Several mature Java-based template languages such as JSP, Velocity[4] and FreeMarker[5] are available as server-side scripting languages for Java-based web servers such as Tomcat. In principle we could have used any of these languages

[4] http://velocity.apache.org
[5] http://freemarker.sourceforge.net

in combination with the reflective API of EMF – or using code generated from the respective Ecore metamodels in order to achieve a more concise syntax. For dynamic languages such as Velocity and FreeMarker, developers could even implement EMF-specific extensions to enable a concise navigation style without needing to generate code from the Ecore metamodels. However, compared to these languages, we strongly believe that EGL is more suitable for the task as it provides first-order logic OCL-based collection navigation operations, built-in support for accessing mutliple models concurrently, and a number of existing *drivers* for interacting with a number of modelling technologies. Even more importantly, since the model connectivity framework discussed in Section 2.1 provides a uniform interface for different modelling technologies, the underlying modelling technology can be substituted later on if necessary (e.g. switch to a database-backed model serialisation format for performance reasons) without requiring changes to the EGL templates.

The Web 2.0 MetaModelbrowser[6] is a web application that can visualise Ecore metamodels and their instances using a fixed tree-based interface that closely mimics the Eclipse-based EMF reflective tree editor. In contrast to MetaModelbrowser, the approach proposed in this paper allows developers to implement custom interfaces for displaying models to end users. Also, using the Graphviz extension our approach enables developers to also embed automatically-generated diagrams to their model browsing web applications.

5 Conclusions and Further Work

In this paper we have presented an approach for re-using a model-to-text transformation language in order to provide support for web-based model navigation and view extraction. Using this approach, non-technical stakeholders can have access to the latest versions of the models of the system from their browser, without needing to purchase and install additional software. Moreover, using such an approach, access control can be enforced if necessary.

With the advent of cloud computing, we believe that web-based model management is a promising field of study with significant potential for practical real-world impact. A few of the open issues that we have identified through this work include management of very large models, concurrent modification of models, caching, and access control. In the future, we will investigate related work in areas such as the management of very large databases and web-based technologies, and adapt best-of-breed approaches to solve the respective problems in the field of web-based model management.

Acknowledgements. The work in this paper was supported by the European Commission via the MADES and INESS projects, co-funded under the 7^{th} Framework programme (grants #218575 (INESS), #248864 (MADES)). We would also like to thank Darren Clowes, Chris Holmes, Julian Johnson, Ray

[6] http://www.metamodelbrowser.org/

Dawson, and Steve Probets from BAE Systems and Loughborough University for their contributions to earlier work that led to the approach presented in this paper.

References

1. Darren Clowes, Dimitrios S. Kolovos, Chris Holmes, Louis Rose, Richard Paige, Julian Johnson, Ray Dawson, and Steve Probets. A Reflective Approach to Model Driven Web Engineering. In *Proc. 6th European Conference on Modelling Foundations and Applications (ECMFA)*, Paris, France, 2010 2010.
2. Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, Fiona A.C. Polack. The Epsilon Generation Language (EGL). In *Proc. European Conference in Model Driven Architecture (ECMDA)*, 2008.
3. Sven Efftinge. XPand Language Reference. http://www.eclipse.org/gmt/oaw/doc/4.1/r20_ xPandReference.pdf.
4. Jon Oldevik. MOFScript User Guide. http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide.pdf.
5. Eclipse Foundation. Epsilon Modeling GMT component. http://www.eclipse.org/gmt/epsilon.
6. Community Z Tools. http://czt.sourceforge.net.
7. Dimitrios S. Kolovos, Louis M. Rose, Richard F. Paige. *The Epsilon Book*. 2008. http://www.eclipse.org/gmt/epsilon/doc/book/.
8. Louis M. Rose. A text-generation language for Epsilon. Master's thesis, University of York, 2007.
9. OMG. Unified Modelling Language 2.2 Specification [online]. [Accessed 10 February 2011] Available at: http://www.omg.org/spec/UML/2.2/, 2007.

Runtime Variability Management for Energy-Efficient Software by Contract Negotiation

Sebastian Götz, Claas Wilke, Sebastian Cech, and Uwe Aßmann

Technische Universität Dresden
Institut für Software- und Multimediatechnik
D-01062, Dresden, Germany
sebastian.goetz@acm.org,
{claas.wilke, sebastian.cech, uwe.assmann}@tu-dresden.de

Abstract. Improving the energy efficiency of software systems requires runtime adjustments and explicit knowledge about the system's variability. Component-based software has inherent variability in terms of multiple implementations for components. These implementations utilize hardware resources, which are direct energy consumers, leading to a further dimension of variability: the mapping of implementations to resources. The performance modes of hardware resources span a third dimension of variability. Hence, to realize energy-efficient software systems the central question is: which implementations should run on and utilize which resources in which performance mode to serve the user's demands? This question can only be answered at runtime, as it relies on the runtime state of the system. In this paper, we show how combined hard- and software models can be utilized at runtime to determine valid system configurations and to identify the optimal one.

1 Introduction

Component-Based Software Development (CBSD) [15] has become a major development approach for software systems. Although both functional and non-functional properties of component-based software systems have been considered, only few approaches focus on energy consumption as a non-functional requirement. Within the research projects CoolSoftware[1] and QualiTune[2] we are developing a *model-driven* CBSD approach for software systems that can be optimized w.r.t. their provided quality and energy consumption at runtime. We call this approach *Energy Auto Tuning (EAT)* [8]. Since the energy consumption of software components depends on the utilized hardware, our component model requires a modeling of both: software components having multiple implementations and hardware components (for simplicity called *resources*). Notably,

[1] http://www.cool-software.org/
[2] http://www.qualitune.org/

components do not only impose energy consumption by being executed on hardware. For example, energy consumption due to sending data using a network device strongly depends on the kind of network device utilized. While resources can vary in their energy consumption and provided qualities (e.g., CPU speed or memory size), software components can vary in their required hardware resources (e.g., different sort algorithms require different amounts of memory) and their required and provided qualities (e.g., a software component can require another component's service of a certain quality and/or can provide services in different qualities to other components).

As models comprising only few components, can already describe different component implementations and deployment locations (e.g., servers), they span a solution space that describes all possible configurations of modeled software systems on modeled hardware landscapes. The problem we address in this paper is, how to select the most appropriate configuration from this solution space specified by models w.r.t. minimum energy consumption at runtime. Thus, we first have to compute all configurations that are valid (i.e., which fulfill requirements of involved components such as dependencies, required resources and provided qualities). Second, we have to select the variant requiring the minimum energy consumption whilst still serving the user's non-functional requirements. We use contracts to model quality dependencies between components and call the determination of the optimal configuration *contract negotiation*, following the definition of Quality-of-Service-level contracts by Beugnard et al. [2] along with Meyer's *design by contract* principle [10].

The process of contract negotiation has to take place at runtime, which imposes the need to utilize our aforementioned models at runtime, too. E.g., the available hardware infrastructure is represented by a model describing each existing resource along with its properties. This model changes over time, e.g., due to failing or added resources. The same holds for our software models, which need to be kept up to date, e.g., due to added components. The runtime data represented by these models is collected by resource or energy managers. Both processes, hence, work on runtime models.

The remainder of this paper is structured as follows: We introduce the Cool Component Model (CCM) which is the basis for our CBSD architecture in Sect. 2 using a simple video server example. Furthermore, we present the Energy Contract Language (ECL), that describes provided and required qualities of software and hardware components. We describe our *contract negotiation* approach in Sect. 3. Afterwards, in Sect. 4 we present and demarcate from related work. Finally, in Sect. 5 we discuss future work and conclude this paper.

2 Background / Context

To capture EAT systems, we developed the energy-aware component model CCM and the contract language ECL. The CCM provides concepts to model hierarchical system architectures and covers both software components and hardware resources, because software consumes energy only in an indirect manner (i.e.,

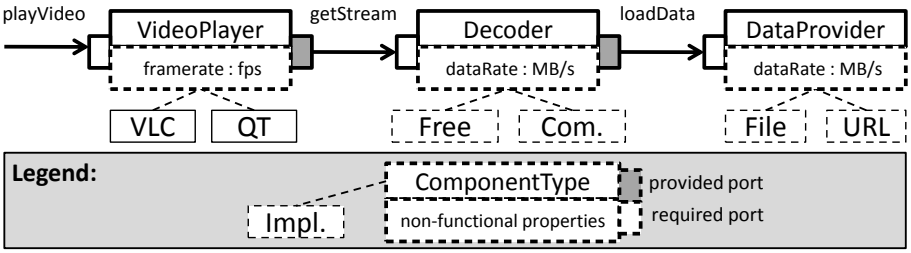


Fig. 1. VideoPlayer SW-Component Types and Implementations.

the energy is consumed by physical resources which are utilized by the software). ECL provides concepts to express dependencies between CCM components based on non-functional properties. This implies dependencies between software components as well as software and hardware components. In this section we introduce CCM and ECL, by means of a video application scenario. In Sect. 3 we use the scenario to explain our contract negotiation approach.

2.1 Capturing Software Components and Resources

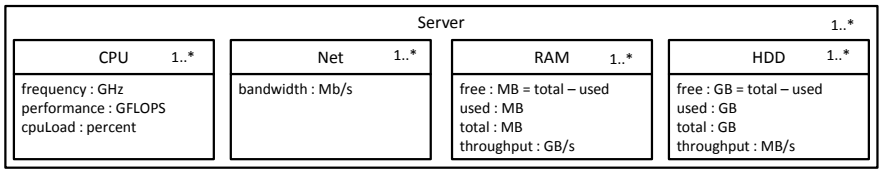
The CCM distinguishes between modeling of the system structure of hardware resources, software components and variants of both. In our project's scope, variants are concrete hardware resources as well as software component implementations. The system structure defines how a system may look like and, thus, represents type declarations for specific variants. For instance, consider the upper part of Figure 1 that shows the types of a video application. It consists of several software component types, namely a `VideoPlayer`, a `Decoder` and a `DataProvider`[3]. Each type may have one or more port types representing an interface of the component. Port types can be used to connect different components. A set of connected components describes the software part of a system.

Concrete implementations (i.e., variants) of a software component (shown in the lower part of Figure 1) have to correspond to the component's type. In our example there are two variants of the type `Player`, the *VLC* and Quicktime (*QT*) implementation. For the `Decoder` type a free (*Free*) and a commercial (*Com.*) implementation are available. Finally, the `DataProvider` is implemented as a local file reader (*File*) and a remote URL reader (*URL*).

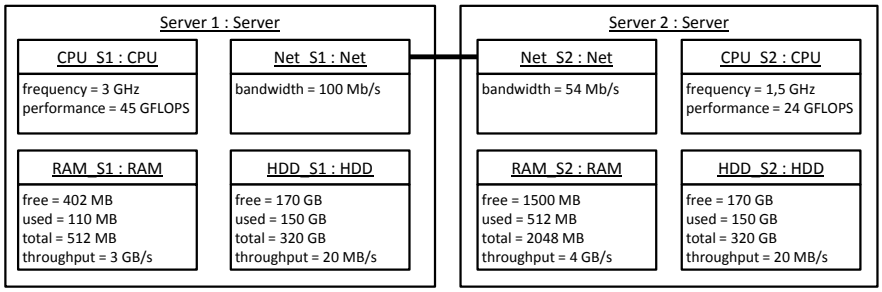
To capture types available in the hardware landscape, resource types have to be specified. Figure 2(a) defines resource types of a hardware landscape on which our video application shall be executed. The `Infrastructure` consists of one or more `Server`s, whereas each server contains one or more `CPU`s, network interfaces (`Net`), `RAM` chips and hard disks (`HDD`). For reasons of simplicity, we omit port types of resource types in the given example.

For each component type (software and hardware) non-functional properties can be defined. For instance, the software component type `Player` defines a

[3] We denote component types using typewriter and variants of them using italic font.



(a) CCM Structure Model for Hardware Landscapes.



(b) CCM Variant Model of a Hardware Landscape Comprised of 2 Servers.

Fig. 2. CCM Hardware Structure and Variant Model.

property `framerate` in frames per second (fps) whereas the resource type `HDD` defines a property `used` (disk space) in GB. Such properties play an important role for specifying ECL contracts and are the basis for contract negotiation.

Figure 2(b) shows a concrete hardware landscape of the resource type system mentioned above. It consists of two servers with specific resources according to the definitions at the type level. The servers are connected by their network devices as depicted by the solid line between *Net_S1* and *Net_S2*. Consider that properties defined at type level are available at variant level with concrete values. Furthermore, each hardware resource variant has to provide a behavior model that defines its energy consumption w.r.t. its utilization. These behavior models have to be provided as templates for each resource type and are instantiated for each concrete resource using the values determined by our resource managers at installation time (i.e., the first time the resource is registered at the runtime environment). We derive the implied energy consumption for a given system configuration (i.e., the distribution of SW components in the infrastructure) and user request by simulating these models. Since the energy consumption computation is not part of contract negotiation, these details are omitted here. The general idea is described in [7].

Notably, variant models are not defined by the developer, but generated at runtime by our Three Layer Energy Auto Tuning Runtime Environment (THEATRE) in accordance to the structural models for HW and SW. THEATRE consists of three layers: the user-, software- and resource layer. Each layer is controlled by a global manager; the Global User Manager (GUM), Global Energy Manager (GEM) and Global Resource Manager (GRM). The GEM has the central role of retrieving information from the GUM and GRM to initiate the

process of contract negotiation and, based on the result, to perform a system reconfiguration. The GUM knows about the details of user requests and associated non-functional requirements of the respective users. Finally, the GRM knows the details about the currently available hardware by monitoring it. These managers generate the respective variant models and keep them up-to-date. Distributed servers are handled by local managers for each layer. E.g., in a 2-server scenario, the first server takes the lead by hosting the global managers, whereas the second server hosts local managers only. Each server, which is part of the system, runs a Local Resource Manager, which registers the server and all its resources at the GRM and sends notifications whenever a property (i.e., the frequency of a CPU) changes. It is important to note that we include subsymbolic information into our variant models (i.e., concrete numbers) and postpone symbolization until contract negotiation. The information required to derive symbols from the values of non-functional properties is encapsulated in our contracts, which are described in the following subsection.

2.2 Specification of ECL Contracts

ECL is used to define dependencies between CCM components using contracts, which are specified for each variant. Therefore, an ECL contract represents a specific view of a variant regarding its dependencies to other types. A contract may define one or more `mode`s, whereas each `mode` defines dependencies to other components. Software components can depend on other software components as well as hardware resources, whereas hardware resources can depend on other hardware resources only. Each dependency relates to a component type and defines bounds for required values of properties at runtime. In addition to constraints expressing required properties, provided properties are specified as well.

Listing 1 shows a contract for the *VLC* video player as a concrete implementation of the `VideoPlayer` component. It defines that the player can be used in two modes: `high-` and `lowQuality`. For `highQuality` the contract specifies that a `CPU` and a `Net` device are required. The `CPU` needs to be utilized at most to 50% and needs to have a frequency of 2 GHz at least.[4] The `Net` device has to offer at least a 10 MBit/s bandwidth. Furthermore, a `Decoder` component is required. Any implementation of that software component type, which is able to provide a data rate of at least 50 KB/s can be used. Finally, the contract defines that in the `highQuality` mode a minimum framerate of 25 fps and a resolution of 1024x768 pixels is provided. To determine the hardware requirements, micro-benchmarks written by the component developer, which address the non-functional properties of interest, are used. As each software component variant is defined by one contract, there exist six contracts specifying SW/HW dependencies in total. The remaining contracts not shown in Listing 1 are similarly structured.

[4] To ease comprehensibility, we use frequency instead of a performance property measured in instructions per second.

```
1  contract VLC implements VideoPlayer {
2    mode highQuality {
3      //required resources
4      requires resource CPU {
5        max cpuLoad = 50 percent
6        min frequency = 2 GHz
7      }
8      requires resource Net {
9        min bandwidth = 10 MBit/s
10     }
11     //dependencies on other SW components
12     requires component Decoder {
13       min dataRate = 50 KB/s
14     }
15     //what is provided in turn
16     provides min frameRate 25 Frame/s
17     provides min imageWidth 1024 Pixel
18     provides min imageHeight 768 Pixel
19   }
20   mode lowQuality { ... }
21 }
```

Listing 1. Example Contract for VLC Video Player.

```
1  contract StandardRAM implements resource RAM {
2    mode low {
3      provides max free: 0.1*total MB
4    }
5    ...
6  }
```

Listing 2. Example Contract for Memory Resource.

In addition to contracts for software components, we allow to define contracts for resources, too. Such contracts suitably illustrate, that the modes of contracts are symbols, which are derived from the resources' properties. E.g., a contract for the resource `RAM` could define the modes *HIGH*, *MEDIUM* and *LOW* as indicated in Listing 2.

In summary, a system modeled with CCM and ECL is highly variable in terms of multiple implementations of component types, multiple quality modes of each implementation and, according to resource requirements of each quality mode, multiple possible mappings of implementations to hardware resources.

3 Contract Negotiation

There are three different kinds of variability captured by CCM/ECL. First, multiple implementations may exist for each software component type. Second, in

an IT infrastructure with more than one server, variability exists in the decision on how to distribute the software components on this IT infrastructure. Finally, the different quality modes specified in ECL denote the third kind of variability. To utilize this variability at runtime for increasing energy efficiency, an approach to determine the optimal system configuration in this regard is required.

This determination can be seen as a special kind of constraint solving optimization problem (CSOP). The goal is to identify the configuration, which implies the lowest energy consumption whilst still serving the user's request and demands. Thus, resource employment (cost) has to be negotiated against the gained utility, where the connection between cost and utility can be expressed as constraints. Hence, constraint programming in general can be applied to solve the CSOP. Because our particular problem is a tradeoff negotiation, expressible using linear constraints, we can use linear programming, which allows the employment of more efficient solving algorithms. To express mappings of components to resources, we need to constrain the domain of this kind of variables to be of Boolean type, i.e., only the integer values 0 and 1 are permitted. Thus, our problem can be classified as a mixed integer linear program (MILP).

An ILP consists of an objective function, a set of constraints and variables being used in both [12]. The objective function is either maximizing or minimizing. In our special case, the objective is to minimize the energy-rated resource usage. Energy-rated means, that there is a factor, which translates between resource usage and energy consumption. This normalizes the different resource usage domains, which else would not be comparable (e.g., size of RAM versus frequency of CPU). A naive approach to determine these factors is to use the standard energy consumption rate, which usually can be found in the resource specification. A more sophisticated approach takes energy-saving and performance modes of resources into account. In this case, factors can be computed using profiling approaches, like Süttner presented in [14].

Our ILP comprises four kinds of variables. Variables expressing resource usage are the first kind, e.g., $usage\#Server_1\#RAM_{S1}\#size$. This variable denotes the size used of the main memory on Server 1. The second kind of variable expresses the mapping selection. E.g., $b\#FreeDecoder\#fast\#Server_2$ denotes whether or not the `FreeDecoder` implementation in `fast` mode has to be mapped to Server 2 (the initial `b` is meant to indicate that this variable is of Boolean type). Software-related properties, like $framerate$, form the third kind of variable. Finally, the server baseload consumption forms the forth kind of variable.

The objective function of our ILP is shown in Equation 1. The goal is to minimize the sum of all resource usage variables and the server baseload consumption. The resource usage variables are normalized by a respective *factor*, which translates resource usage into power consumption and is determined by our resource managers the first time a resource registers at the system.

$$min \sum (factor_{xyz} \times usage\#server_x\#resource_y\#property_z) + \sum baseload\#server_i \quad (1)$$

The constraints of the ILP can be divided into four classes: (i) selection criterias, (ii) resource usage and server baseload, (iii) implied values for non-functional properties and (iv) user demands. The first class corresponds to the information present in the structural model, that is, which (and how many) components are required. In our example, one implementation of each software component is required. The requirement, that exactly one implementation of a component type t has to be chosen, can be expressed by constraints of the following form.

$$\sum_{x,y,z} (b\#impl_{x,t}\#mode_y\#server_z) = 1.0; \tag{2}$$

The second class of constraints describes the boundaries of resource usage variables and how they correlate to the mapping of implementations to resources. For each resource usage variable a constraint for the upper bound (3) and the lower bound (4) is introduced. The values for these boundaries are extracted from the hardware variant model.

$$usage\#Server_1\#RAM_{s1}\#size <= 512.0; \tag{3}$$

$$usage\#Server_1\#RAM_{s1}\#size >= 0.0; \tag{4}$$

The baseload of servers (determined by the local resource managers) is reflected by constraints for each mapping variable as depicted in Equation 5.

$$baseload\#server_i = b\#impl_x\#mode_y\#server_i \tag{5}$$

The impact of mapping an implementation to a resource can be extracted from ECL contracts and can be represented in constraints like exemplary depicted in Equation 6. Here, e.g., the first addend of the sum, states that the `FreeDecoder` implementation requires 512 MB of RAM to operate in `fast` mode. The same statement holds for any other server, but the example equation refers to RAM usage of Server 1 only.

$$\begin{aligned} usage\#Server_1\#RAM_{s1}\#size = \\ 512.0 * b\#FreeDecoder\#fast\#Server_1 + \\ 256.0 * b\#FreeDecoder\#slow\#Server_1 + \\ 128.0 * b\#CommercialDecoder\#slow\#Server_1 + \\ 512.0 * b\#CommercialDecoder\#fast\#Server_1 + \\ 1536.0 * b\#CommercialDecoder\#ultrafast\#Server_1 \end{aligned} \tag{6}$$

The same principle is applied for software-related non-functional properties, which are the third class of constraints as shown exemplary in Equation 7. It states, among others, that the *throughput* will be five bit per second, if the `URLReader` is mapped to Server 1 and configured to run in `URL` mode.

$$\begin{aligned} throughput = {} & 5.0 * b\#URLReader\#url\#Server_1 + \\ & 20.0 * b\#FileReader\#file\#Server_2 + \\ & 5.0 * b\#URLReader\#url\#Server_2 + \\ & 20.0 * b\#FileReader\#file\#Server_1 \end{aligned} \tag{7}$$

Finally, the user demands have to be integrated as a constraint, too. Such a user request, like playing a video with a framerate of at least 20 frames per second, can be integrated as a constraint in a straightforward way:

$$framerate >= 20.0; \tag{8}$$

The ILP as a whole is generated at runtime, whereby the required information is extracted from the runtime model of the current hardware infrastructure and software configuration (software variant model), as well as from the ECL contracts. To solve the ILP a variety of free-to-use solvers exists. For our prototype we have chosen LP_Solve 5.5[5]—one of the mature, stable solvers. LP_Solve allows to solve linear programs (LP), too.

The key difference between LP and ILP is that an ILP restricts its variables to be integers instead of floating reals. In our scenario we need floating reals for the resource usage and property variables, but integers for our mapping selection variables. In consequence, the ILP presented above is a mixed integer linear program (MILP). The need for the integer restriction can be easily illustrated: if the variables $b\#VLC\#highQuality\#Server_1$ and $b\#VLC\#highQuality\#Server_2$ are allowed to be floating reals the LP's solution could be, to map 33% of the `VLC` to Server 1 and the remaining 67% to Server 2, which is obviously not possible.

The most commonly used algorithm to solve a MILP is the simplex algorithm [12]. The major drawback of simplex is its exponential runtime. But, importantly for our scenario, it is an iterative approach. That is, once an MILP has been solved, slight changes to it do not require to perform the whole computation again, but only parts of it. Thus, unless the system significantly changes, our optimization approach will benefit from this property of the algorithm. We plan to measure the energy consumption of solving our ILPs to derive a model for the prediction of the energy required to compute the optimal configuration.

The solution of the example introduced throughout the paper is that the *VLC* implementation should run in *highQuality* mode on Server 1, the *CommercialDecoder* should run in *slow* mode on Server 2 and the *URLReader* in *URL* mode on Server 1. The `Decoder` implementation is mapped to Server 2 instead of Server 1, due to the CPU performance requirements. If all implementations run on Server 1, the resulting energy consumption will be lower, but the framerate of at least 20 fps cannot be ensured. Furthermore, the solution of the ILP tells us amongst other details, that we need the CPU of Server 1 to operate at 1.5GHz and the CPU of Server 2 at 800 MHz. Thus, we could force the CPUs

[5] http://lpsolve.sourceforge.net/5.5/

to operate slower than usual by exploiting the application knowledge in terms of the ILP to save energy whilst ensuring the requested user utility.

Notably, if a new server, with a more powerful CPU, is added to the infrastructure, the ILPs solution is to map all three implementations to that new server, which requires a system reconfiguration: all implementations have to be migrated from Server 1 or 2 to Server 3. In general, changes in the infrastructure as well as to (the available) component implementations are propagated to our variant models at runtime, which are then used to generate an ILP to derive the optimal system configuration. Finally, the system has to perform a reconfiguration, which is a sequence of migration steps.

4 Related Work

Within the research project COMQUAD, a component model was developed that separated components into their specifications and implementations [6], similar to the component types and component implementations of the CCM. Additionally, the contract language CQML+ [13] was developed to describe required and provided non-functional properties of software components. The enhancement of our approach is the more detailed modeling and monitoring of resources.

During the research project SPEEDS and its successor CESAR, the HRC metamodel [16] was developed. It allows for component-based development of embedded systems, which includes capabilities to describe hard- and software components and their behavior. CESAR focuses on a multi-viewpoint, multi-level development process for embedded systems. Contracts are a central concept in HRC models which are organized in behavior, safety and real-time viewpoints. Contracts capture functional and non-functional assumptions and promises of HRC components. They are used to reason about the consistency of a given HRC model. In contrast to our approach, SPEEDS and CESAR focus on contract negotiation at development time and not at runtime. Thus, the HRC metamodel and its successor CSM support variability at development time, whereas the CCM focuses on runtime. In addition, neither SPEEDS nor CESAR explicitly consider energy consumption or EAT.

In the MADAM research project and its successor MUSIC, a component model for self-adaptive applications on mobile devices has been developed [5]. It supports modeling of non-functional properties and implementation variants. Although, energy optimization is possible in general, in contrast to our approach, MADAM/MUSIC do not focus on complete hardware landscapes.

The DIVA research project focused on the management of dynamic adaptive systems with special focus on the problem of exponential growth of potential system configurations by combining methods from aspect-oriented programming/modeling [9] and Model-Driven Software Development (MDSD) [11]. The DIVA approach allows to automatically adapt a system at runtime supporting goal-based optimization of non-functional properties as well as rule-based reconfiguration of the system [4]. The major difference to our approach is the level of abstraction regarding the values of non-functional properties. DIVA symbol-

izes the impact of implementations on non-functional properties (i.e., the free size of memory is represented by symbols like *LOW*, *MEDIUM* and *HIGH* and the impact of an implementation can only be expressed as being low, medium and so on, too). Our approach allows to consider subsymbolic information in addition (i.e., the actual value of free size of memory in MB). We encapsulate symbolization in our contracts, as was shown in Sect. 2.2. Though, reasoning on subsymbolic information is less efficient, due to the raised complexity, it allows to derive finer-grained configurations. E.g., a configuration could include not just the information which CPU to use, but the (optimal) frequency this CPU should have. Such fine-grained information allows to reduce energy consumption in addition to the coarse-grain decision of which resources to use. Current hardware is usually far from being energy-proportional [1], which is reflected by a very high baseload electricity and a narrow working area. Imagine, for example, a server consuming 100W being idle and 120W at full load. In consequence, energy savings can mostly be achieved by selecting or turning off the right resources. In such a scenario symbolic reasoning, as in DIVA, is feasible. But especially for the next generation of hardware, which is supposed to be more and more energy-proportional [3] there is a need for finer-grained energy optimizations.

5 Conclusion

In this paper we introduced our contract negotiation approach, which allows to identify the most energy-efficient mapping of software component implementations to resources serving a user's request and demands.

We showed how to formulate this optimization problem as an integer linear program (ILP) and presented a mechanism using models at runtime to generate the ILPs in accordance to the current hard- and software as well as the current users requests and demands. This allows to optimize even dynamic systems, whose hard- and software entities can supervene, disappear or change over time.

Our approach allows for runtime subsymbolic reasoning, which demarcates us from existing approaches. Notably, energy efficiency is just one quality which benefits from subsymbolic reasoning. E.g., optimizations in systems integrating the physical and virtual world (like robot swarms) require the support for floating point numbers, too.

In the future we plan to improve the approach to consider more complex relations between resource usage and energy consumption and will evaluate our approach in a real world scenario. Furthermore, we plan to investigate how our approach can be applied to systems integrating the physical and virtual world.

Acknowledgement

This research has been funded by the European Social Fund and Federal State of Saxony within the project ZESSY #080951806, by the Federal Ministry of Education and Research within the project CoolSoftware #FKZ13N10782, part of the Leading-Edge Cluster "Cool Silicon" within the scope of its Leading-Edge Cluster Competition and by the collaborative research center 912 (HAEC), funded by the DFG.

References

1. L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, 2007.
2. A. Beugnard, J.-M. Jézéquel, and N. Plouzeau. Contract aware components, 10 years after. In *Electronic proceedings in theoretical computer science*, number 37, pages 1–11, 2010.
3. S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54:67–77, May 2011.
4. F. Fleurey and A. Solberg. A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pages 606–621, Berlin, Heidelberg, 2009. Springer-Verlag.
5. K. Geihs, M. U. Khan, R. Reichle, A. Solberg, S. Hallsteinsen, and S. Merral. Modeling of component-based adaptive distributed applications. In *SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 718–722, New York, NY, USA, 2006. ACM.
6. S. Göbel, C. Pohl, S. Röttger, and S. Zschaler. The COMQUAD component model - enabling dynamic selection of implementations by weaving non-functional aspects. In *Proceedings of the 3rd international conference on aspect-oriented software development, Lancaster, UK, March 22 - 24, 2004*, volume 3, pages 74–82, New York, NY, USA, March 2004. ACM Press.
7. S. Götz, C. Wilke, M. Schmidt, and S. Cech. THEATRE resource manager interface specification. Technical Report TUD-FI10-0X, Technische Universität Dreden, Dresden, Germany, 2010.
8. S. Götz, C. Wilke, M. Schmidt, S. Cech, and U. Aßmann. Towards energy auto tuning. In *Proceedings of First Annual International Conference on Green Information Technology (GREEN IT)*, pages 122–129. GSTF, 2010.
9. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Berlin / Heidelberg, 1997.
10. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.
11. B. Morin, O. Barais, G. Nain, and J.-M. Jézéquel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 122–132, Washington, DC, USA, 2009. IEEE Computer Society.
12. G. L. Nemhauser and L. A. Wolsey. *Integer and combinatorial optimization*. Wiley-Interscience, New York, NY, USA, 1988.
13. S. Röttger and S. Zschaler. CQML+: Enhancements to CQML. In *Proceedings of the 1st International Workshop on Quality of Service in Component-Based Software Engineering*, pages 43–56, Toulouse, France, 2003. Cépaduès-Éditions.
14. P. Süttner. Abstract behavior description of CCM software components (Abstrakte Verhaltensbeschreibung von CCM Softwarekomponenten). Master's thesis, Technische Universität Dresden, Mar. 2011.
15. C. Szyperski, D. Gruntz, and S. Murer. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley and ACM Press, 1999.
16. The SPEEDS Consortium. D.2.1.5 SPEEDS L-1 Meta-Model. http://speeds.eu.com/downloads/SPEEDS_Meta-Model.pdf, May 2009.