

# Introducing Adaptivity to Achieve Longevity for Software

Mahdi Derakhshanmanesh<sup>1</sup>, Jürgen Ebert<sup>1</sup>, Mehdi Amoui<sup>2</sup>, Ladan Tahvildari<sup>2</sup>

<sup>1</sup>University of Koblenz-Landau, Germany, Institute for Software Technology

<sup>2</sup>University of Waterloo, Canada, Software Technologies Applied Research

Email: {manesh|ebert}@uni-koblenz.de, {mamouika|ltahvild}@uwaterloo.ca

**Abstract:** *Long living software systems* (LLSSs) must provide the flexibility to react to changes in their operating environment as well as to changes in the user's requirements, even during operation. *Self-adaptive software systems* (SASSs) face adaptivity at runtime within predefined bounds. Yet, not all types of necessary variations can be anticipated and unforeseen changes to software may happen. Thus, systems that are meant to live in such an open-ended world must provide self-adaptivity (*micro adaptation*), but there is an additional need for adaptability of the system so that it can be adjusted externally (*macro adaptation*). This paper gives an overview of the *graph-based runtime adaptation framework* (GRAF) and sketches how it targets both types of adaptation.

## 1 Introduction

As the lifetime of software tends to get even longer in the future, such *long living software systems* (LLSSs) must be adaptable to changing requirements. These requirements may stem from varying user needs or from changes in the software's operating environment. Ideally, an LLSS is capable of reacting to such changes during operation, without showing any noticeable down-time in service to the system's users. But, in practice there have to be adaptive maintenance actions as well.

A *self-adaptive software system* (SASS) is designed to face *foreseen* changes in its operating environment. On such a change, an SASS is still capable of fulfilling its set of requirements. More generally, an SASS is a *reflective* system that is able to modify its own behavior in response to changes in its operating environment [OGT<sup>+</sup>99]. It may also be called *autonomic* and/or *self-managing* system [ST09].

Although SASSs today can already deal with changes in their environment, they are not well-suited to tackle an *open-ended world*, which is especially characterized by *unforeseen* changes [SEGL10]. To handle these changes, classical approaches for software evolution have to be applied.

In this paper, we present the *graph-based runtime adaptation framework* (GRAF) [DAET], which supports the introduction of variability to software and goes beyond the goals of achieving runtime adaptivity for a predefined problem space only.

By explicitly making self-adaptation of the SASS at runtime (*micro adaptation*) as well as maintenance tasks performed by an external subject (*macro adaptation*) two comple-

mentary parts of an integrated process, the introduction of runtime adaptivity to software can help with incorporating the required flexibility for longevity of software. From that perspective, we see adaptivity as the main property of software to be adjustable to any changes in the environment, no matter if adaptation is actually performed online (on its own or with external help) or offline.

In our CSMR 2011 short paper [ADET11], we propose a flexible, model-centric architecture for SASSs that supports a process for migrating non-adaptive software towards self-adaptive software. In a second paper [DAET], we present the details of GRAF and its abilities in the area of micro adaptation together with a comprehensive case study.

This paper complements the preceding publications by focusing on the use of GRAF for achieving longevity of software. It is organized as follows. In Section 2, we describe the GRAF framework, which implements a flexible, model-centric architecture for SASSs, as proposed in [ADET11]. Then, we introduce the core ideas behind *micro* and *macro* adaptation in Section 3 and give an overview on the implementation work and the case studies done up to now in Section 4. We briefly cover related work in Section 5. Finally, we conclude this paper in Section 6 and give an outlook on possible areas of future work.

## 2 Architecture of GRAF

In the context of migrating existing, non-adaptive software towards an SASS, Amoui et al. present a model-centric architecture that is designed around an external adaptation framework in [ADET11].

The *graph-based runtime adaptation framework* (GRAF) acts as an *external controller* of the adaptable software. GRAF is not just an adaptation manager. It encapsulates a *runtime model* that is specific to the adaptable software. The framework achieves adaptivity of the adaptable software's behavior by *querying*, *transforming* and *interpreting* parts of the runtime model. The main layers and components are illustrated in Figure 1.

Subsequently, we discuss the structure of the adaptable software and GRAF as well as the connecting interfaces. The responsibilities and tasks of each framework layer are introduced and we explain how each of them contributes to the whole architecture of an SASS that can be extended in reaction to unforeseen changes.

### 2.1 Adaptable Software

In the context of GRAF, we refer to the software to be managed by the framework as the *adaptable software*. It is set up in a way so that it can be controlled by GRAF, which plays the role of the external adaptation management unit. Adaptable software can be built by migrating an existing, non-adaptable software system towards adaptability [ADET11]. Alternatively, it can be developed from scratch.

The adaptable software is *causally connected* to the runtime model which is encapsulated

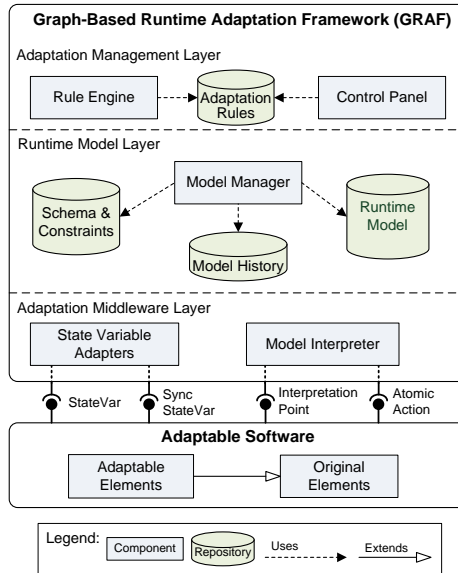


Figure 1: A SASS that is using GRAF to achieve adaptivity.

and managed by GRAF. This causal connection is established by an intermediate layer, the *adaptation middleware*, which communicates with the adaptable software using predefined interfaces.

When applying GRAF in a migration context, *original elements* are those software elements (e.g., classes or methods) that are already existing within the software application to be migrated towards a SASS. In the context of creating adaptable software from scratch, these elements can be thought of as helper elements, such as classes provided by imported libraries.

In a migration towards self-adaptive software, *adaptable elements* are derived from original elements by applying refactoring steps and by exposing them via predefined interfaces to GRAF. They are the building blocks that support a certain degree of variability by either providing data about changes, or, by offering actions to be used for adjusting the system.

Every adaptable software element needs to provide some of the interfaces for adaptation provided by GRAF as described in Table 1.

## 2.2 Graph-Based Runtime Adaptation Framework

The GRAF framework itself consists of three layers, namely the *adaptation middleware layer*, which establishes the connection to the adaptable software, the *runtime model layer*, which encapsulates and handles the runtime model and the *adaptation management layer*, which controls the adaptation actions at runtime either triggered by events stemming from

Table 1: Interfaces to be provided by any adaptable software for interaction with GRAF.

Interface	Description
<code>StateVar</code>	exposes variables that hold information about the operating environment. Changes in their values are propagated to the model manager, which decides if the value is (i) stored in the runtime model ( <i>Reification</i> ) or (ii) discarded.
<code>SyncStateVar</code>	exposes variables similar to <code>StateVar</code> , but their representation in the runtime model can be also changed from outside. The new value is then propagated back to the managed adaptable software.
<code>InterpretationPoint</code>	exposes handles to positions in the adaptable software's control flow ( <i>interpretation points</i> ) at which an associated part of the behavioral model shall be executed (by interpretation).
<code>AtomicAction</code>	exposes methods that can be used as <i>atomic actions</i> in the behavioral model. They are used by the interpreter to execute individual actions that are composed inside of the behavioral model.

the model layer or by external actions. These layers are described in more detail in the following.

### 2.2.1 Adaptation Middleware Layer

The *adaptation middleware layer* is stateless and responsible for all possible ways of communication and interaction between the adaptable software and the runtime model layer. It provides a set of reusable components that are independent from the adaptable software.

**State Variable Adapters.** *State variable adapters* support the propagation of (changed) values from annotated variables in the adaptable elements to the runtime model by using the `StateVar` interface. In addition, the `SyncStateVar` interface also uses these adapters when propagating tuned variable values from their runtime model representation back into the adaptable software.

**Model Interpreter.** For adaptivity of behavior, GRAF's *model interpreter* executes parts of the behavioral model that compose *actions* available via the `AtomicAction` interface. That way, behavior for a specific point in the adaptable application's control flow (*interpretation point*) within adaptable elements is based on a model. Interpretation points are the starting points of methods that need adaptivity and, hence, are exposed to the interpreter

by the `InterpretationPoint` interface.

The model interpreter is specific to a given runtime model schema. We give an excerpt in Figure 2. Each `Action` in the behavioral model is associated with an existing, implemented method in the adaptable software. Starting at an `InitialNode` and walking along a path in the behavioral model, the interpreter resolves conditions (expressed as queries on the runtime model) at `DecisionNode` vertices. Methods of the adaptable software are invoked via the `AtomicAction` interface and using Java reflection for each `OpaqueAction`. Interpretation terminates when a `FinalNode` vertex is reached and the adaptable software continues executing.

### 2.2.2 Runtime Model Layer

The *runtime model layer* contains the runtime model that is at the core of the GRAF approach to achieve adaptivity. In addition, this stateful layer contains utility components that simplify and encapsulate necessary operations on the graph-based runtime model, such as evaluating queries or executing transformations.

**Runtime Model.** The design of GRAF's architecture is centered around a *runtime model*. Essentially, this model needs to (i) offer a view on the adaptable software's inner state as well as (ii) describe behavior that can be executed by an interpreter. Hence, the runtime model serves as a view on the adaptable software and (iii) is controllable and modifiable by the rule engine.

In GRAF, the runtime model conforms to a customizable *schema* (meta-model) that is described as UML class diagrams. GRAF automatically generates Java source code to create and manipulate models that conform to the schema as described in UML.

An excerpt of the schema that is currently used is illustrated in Figure 2. According to this schema, every runtime model contains a set of `StateVariable` vertices to store exposed data from the adaptable software and `Activity` vertices to store entry points to behavior executable by runtime interpretation.

In this schema, behavioral models are kept as variants of UML activity diagrams which are represented by their *abstract syntax graphs*. In general, also other types of behavioral models such as Petri nets, statecharts, etc. could be used.

**Schema and Constraints.** GRAF supports the validation of *constraints* on the runtime model. A set of constraints can be derived from the runtime model schema. Examples are the multiplicities for associations that restrict the allowed edges between vertices in the runtime model's internal graph representation. To express further restrictions on the modeling language, complex constraints can be defined on the runtime model schema by using a query language.

Additionally, developers of a SASS can write constraints that are specific to the domain of their adaptable software and belong to the SASS's runtime model. For instance, it is possible to express that, for a certain behavioral model (activity), specific actions (methods)

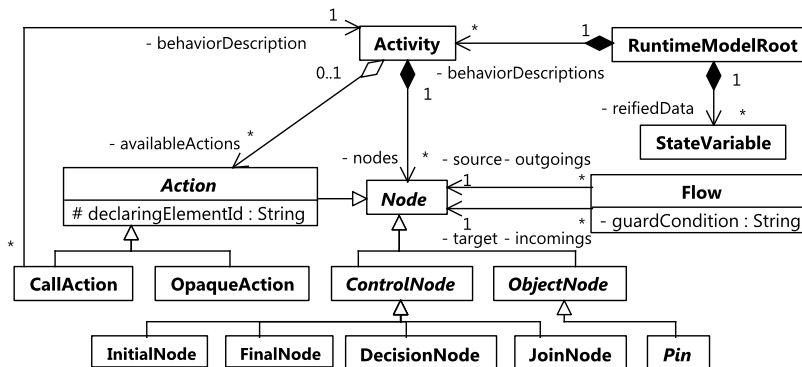


Figure 2: Excerpt of the runtime model schema.

are not allowed to be called in sequence. At present, constraint checking mechanisms have been implemented only prototypically and are not yet used in the adaptation loop.

**Model Manager.** The *model manager* is responsible for all tasks that are related to manipulating the runtime model. It keeps the runtime model in sync with the inner state of the adaptable software, which is exposed via the `StateVar` and `SyncStateVar` interfaces.

In addition, the model manager acts as a controller for all other types of accesses to the runtime model and hence, it provides a set of utility services for *querying* and *transforming*. Furthermore, the model manager is responsible for the evaluation of constraints after a transformation has been applied. It can even roll back changes to the runtime model and it informs the rule engine about this failed adaptation attempt.

**Model History.** The *model history* supports the storage of temporal snapshots of the runtime model as additional information (history/prediction). The rule engine can access this data by querying the history repository via the model manager. That way, the rule engine can learn from its past actions by using the available history. The application of data mining techniques such as *frequent itemset analysis* [AS94] are possible once the repository contains a representative amount of data. Moreover, the collected data may provide valuable information for maintaining the SASS. The model history has not been implemented yet.

### 2.2.3 Adaptation Management Layer

The *adaptation management layer* acts as the controller that *runs in parallel* to the adaptable software and uses a set of adaptation rules to make changes to the runtime model

over time. This stateful layer is composed of a repository with adaptation rules and a rule engine that uses its own heuristics to plan adaptation.

**Rule Engine.** In GRAF, the *rule engine* plays the role of an adaptation manager. It gets notified by events, for instance, whenever the runtime model changes. Moreover, it can receive additional information on the current or past state of the adaptable software by using the model history. After gathering all the required information for planning, the rule engine uses the set of available adaptation rules to choose a (compound) transformation to be executed on the runtime model via the model manager.

**Adaptation Rules.** Adaptation rules are *event-condition-action* rules that describe *atomic adaptation tasks*. The three parts of an adaptation rule are as follows:

1. **Event.** The event for the application of a rule occurs whenever the runtime model changes and the rule engine needs to react. The model manager keeps track of these events.
2. **Condition.** The condition is expressed by a boolean query on the runtime model that tests whether an adaptation action may be applied or not.
3. **Action.** The action is a model transformation and encodes the actual adaptation to be performed on the runtime model.

**Control Panel.** The *control panel* allows administrators to interact with the SASS and especially with GRAF. They can (i) observe the log of adaptation steps by querying the runtime model history, (ii) modify adaptation rules and model constraints as well as (iii) override decisions and heuristics of the rule engine. Furthermore, administrators get notified whenever an adaptation attempt fails or if the SASS is totally uncontrollable by the adaptation management layer. The presented version of GRAF does not implement the control panel.

### 3 Runtime Adaptivity and Evolution

If any runtime changes to the SASS that are carried out autonomously by itself (*micro adaptation*) are not sufficient, some of its components or repositories must be maintained externally and possibly manually (*macro adaptation*). Due to the limits of SASSs in dealing with unforeseen situations as well as the increasing entropy (disorder) that may be caused by self-adaptation, phases of micro and macro adaptation need to alternate to achieve longevity. In this section, we exemplarily discuss micro and macro adaptation, assuming that the SASS has been created using GRAF.

### 3.1 Micro Adaptation

Subsequently, we give an overview of the main steps during micro adaptation, disregarding all the possible alternate flows and exceptions that might occur on purpose here. A detailed description is given in [Der10].

Assuming application-level adaptation, as discussed in [ST09], these changes lead to changes of the *inner state* of the adaptable software (the *self*) and thus, can be sensed in its variables and fields. Changed values from the adaptable software's inner state are exposed to state variable adapters via the `StateVar` interface and finally propagated to the runtime model. This results in a notification of the rule engine in the adaptation management layer.

Following that, GRAF executes steps similar to the *MAPE-loop* as described in IBM's architectural blueprint for autonomic computing [IBM06]. When a change listener of the rule engine receives an event from the model manager, the rule engine analyzes the change and plans a (compound) transformation to be applied on the runtime model via the model manager. Afterwards, the runtime model is adjusted.

Finally, an actual behavior change of the managed adaptable software has to be achieved. Two different ways are supported in GRAF: (i) adaptation via state variable values and (ii) adaptation via interpretation of a behavioral (sub-)model.

In the first case, state variables can be chosen so that changes to them are reflected back to the adaptable software. For this mechanism to work, the corresponding variable of the adaptable application must be exposed to GRAF via the `SyncStateVar` interface. If the runtime model value of such a state variable is changed by a transformation, the adaptable software is made aware of the new value whenever the variable is *used* in the adaptable software.

In the second case, GRAF supports the modeling of behavior (using UML activity diagrams at the moment). During execution of some operation of the adaptable software which is exposed to GRAF via the `InterpretationPoint` interface, the model interpreter is invoked and finds the associated behavioral model. If this behavior is default behavior, there is no need for adaptation and the interpreter returns this information so that the adaptable element executes the existing default code block in the adaptable software. Otherwise, the model interpreter executes the associated part of the behavioral model using the `AtomicAction` interface, replacing a part of the adaptable software's default control flow.

### 3.2 Macro Adaptation

In cases where the SASS faces new challenges that it cannot tackle adequately, the system must be evolved, that is, an external subject (here, the human administrator) must be able to adjust the system. Due to the flexibility of the proposed architecture, many of these maintenance tasks can be performed at runtime. Still, there are cases where the

SASS must be stopped and restarted again when technical reasons require it, e.g., after modifications to the source code. In this section, we discuss ideas and sketch macro adaptation in the context of SASSs built around GRAF. We start top-down, from the adaptation management layer.

Ways to extend the adaptivity property of the SASS are to (i) *add* new adaptation rules to the repository, (ii) *modify* existing adaptation rules in terms of their queries and transformations or even (iii) *remove* adaptation rules. Such changes can be made via the external control panel. Ideally, the rule engine does not have to be adjusted. Changes to its analyzing and planning algorithms may be needed, though. By separating them into different modules and keeping them independent from the core, this issue can be solved.

At the runtime model layer, further adjustments are possible. Among the smaller changes, there are modifications similar to the adaptation rules (add, modify, remove) that can be performed for application-specific constraints. Likewise, changes to the runtime model may be desirable, such as refining parts of the behavioral model. More complex changes are schema-related adjustments. Developers may want to change the language for modeling the runtime model and use Petri nets or statecharts instead of UML activity diagrams. Given such a heavy change, the adaptation rules will be affected as well and the model interpreter has to be adjusted or even replaced.

As the adaptation middleware is a generic communication layer, it stays stable for most of the time. Only a change of the runtime model schema (description of syntax) will result in a re-write of the model interpreter (description of semantics). Changes to state variable adapters are only needed in cases where the concept of state variables is modeled differently by the runtime model schema.

There might be necessary changes at the level of the adaptable software as well. In cases where existing state variables do not provide data about changes sufficiently, new ones have to be introduced and exposed to GRAF. Again, existing state variables can be deleted as they may become obsolete over time. Furthermore, the existing atomic actions may not provide enough expressiveness to model the desired behavior and the pool of available, exposed actions needs to be refined.

The necessity to start a macro adaptation should be detected by the framework itself. This can be done by special rules that test some quality properties of the model or by analyzing the history information periodically, for instance.

Finally, *traceability* between different elements at all layers is important, e.g., as atomic actions are removed, they may no longer be used in the behavioral model and adaptation rules shall not create variations of behavior that make use of these missing actions.

## 4 Implementation and Case Studies

GRAF is implemented around a small set of interfaces and abstract classes in Java and each of its components can be extended or replaced with minor effort, thus, making it suitable for constructing long living, self-adaptive Java software systems.

Java *annotations* provide an easy way to add meta-data to existing source code without altering its functional properties. With the help of *aspect oriented programming* (AOP) techniques and a powerful AOP library like JBoss AOP<sup>1</sup>, these annotations are then used in pointcut expressions resulting in automatic instrumentation of the byte-code to connect the adaptable software to GRAF via its middleware.

GRAF's runtime model and its operations are based on graph technology. Models, schemas as well as queries and transformations are implemented in the technological space of *TGraphs* [ERW08]. Using the *IBM Rational Software Architect*<sup>2</sup>(RSA), the runtime model schema is developed by drawing UML class diagrams. Based on the runtime model schema, a Java API is generated using JGraLab<sup>3</sup>. The generated Java classes represent vertex- and edge-types of runtime models (TGraphs) that conform to the runtime model schema. The generated API is then used for creating, accessing, and modifying runtime models. While transformations, the action part of adaptation rules, are currently realized using the generated Java API, we use the *Graph Repository Query Language* (GReQL) for checking constraints and retrieving data from the runtime model TGraph.

We successfully applied GRAF to the open-source VoIP application OpenJSIP<sup>4</sup>, where adaptivity was introduced to achieve less timeouts and a more clear reaction of the system to the user under high system load [Der10]. Furthermore, we performed a more comprehensive case study with the game engine Jake2<sup>5</sup>, where the SASS adapts the behavior of artificial players depending on the human player's skills [DAET]. An increase in used memory (model representation, JBoss AOP wrappers) and in execution times (model interpretation) is measurable in both case studies, but this did not affect the user's experience.

## 5 Related Work

Schmid et al. motivate in [SEGL10] that as future systems will operate even longer, they must be capable of coping with changes in their environment or their requirements during operation. Although self-adaptive software can handle such challenges within given bounds, *long living software systems* need to be extendable and adaptable in an open-ended world, where changes are unforeseen. Schmid et al. propose the modularization of models for evolving adaptable systems and discuss an adaptive system with adaptivity enhancement. This paper strengthened our motivation to present GRAF in the context of longevity.

The DiVA project [MBJ<sup>+</sup>09] considers design and runtime phases of adaptation. It is based on a solid four dimensions modeling approach to SASS [FS09]. At design time, a base model and its variant architecture models are created. The models include invariants and constraints that can be used to validate adaptation rules. In comparison to our GRAF approach, DiVA also focuses on adaptation at the architectural level, whereas we target

---

<sup>1</sup><http://www.jboss.org/jbossaop>

<sup>2</sup><http://www-01.ibm.com/software/awdtools/swarchitect/websphere/>

<sup>3</sup><http://www.ohloh.net/p/jgralab>

<sup>4</sup><http://code.google.com/p/openjsip/> (Last access: 14<sup>th</sup> of January, 2011)

<sup>5</sup><http://bytonic.de/html/jake2.html> (Last access: 14<sup>th</sup> of January, 2011)

adaptation at a lower level by incorporating methods and fields as well.

Vogel and Giese [VG10] propose a model-driven approach which provides multiple architectural runtime models at different abstraction levels. They derive abstract models from the source models. Subsets of the target models focus on a specific adaptation concern to simplify autonomic managers. Similar to DiVA, this research also targets adaptivity from an architectural point of view.

The Rainbow framework [CHG<sup>+</sup>04] uses an abstract architectural model to monitor the runtime properties of an executing system. Similar to GRAF, certain components of Rainbow are reusable and the framework is able to perform model constraint evaluation. However, this framework is mainly targeting architecture-level adaptation, where adaptive behavior is achieved by composing components and connectors instead of interpreting a behavioral model.

Similar to some of the presented work, our implementation of GRAF makes use of *aspect oriented programming* for binding the adaptable software to the framework. In contrast to the mentioned research projects, we focus on the *interpretation of behavioral models* for achieving adaptability at runtime, while also supporting parameter adaptation.

## 6 Conclusions and Future Work

In this paper, we presented an approach for achieving runtime adaptivity in software based on the model-centric adaptation framework GRAF. Following the proposal for a model-centric SASS in [ADET11], we implemented GRAF to support the migration of existing software towards a SASS. The construction of an SASS from scratch is supported as well.

This framework is based on a layered architecture that features (i) a *middleware layer* for managing the coupling and communication of GRAF with the adaptable software, (ii) a *runtime model layer* that reflects the current state of the adaptable software and its behavior at certain parts that need adaptivity as well as (iii) an *adaptation management layer* with a rule engine that is able to adapt the runtime model by using on given adaptation rules. A first case study is described in [Der10] and a more comprehensive one will be available in [DAET].

Based on the view of micro and macro adaptation, we plan to experiment with different scenarios to learn more about sequences of alternation between these two phases. Finally, we see additional future work in the area of designing an evolution process that combines (i) the knowledge about *traditional maintenance* with (ii) the flexibility of *runtime adaptivity* given by self-adaptive software systems. We believe, that integrating these two areas provides a promising way towards achieving longevity in software.

**Acknowledgement.** Our thanks go to the anonymous reviewers for their thorough and inspiring feedback.

## References

- [ADET11] Mehdi Amoui, Mahdi Derakhshanmanesh, Jürgen Ebert, and Ladan Tahvildari. Software Evolution Towards Model-Centric Runtime Adaptivity. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, Oldenburg, Germany, 2011. To appear.
- [AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.
- [CHG<sup>+</sup>04] Shang-wen Cheng, An-cheng Huang, David Garlan, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37:46–54, 2004.
- [DAET] Mahdi Derakhshanmanesh, Mehdi Amoui, Jürgen Ebert, and Ladan Tahvildari. GRAF: Graph-Based Runtime Adaptation Framework. Submitted elsewhere.
- [Der10] Mahdi Derakhshanmanesh. Leveraging Model-Based Techniques for Runtime Adaptivity in Software Systems. Master's thesis, University of Koblenz-Landau, Germany, 2010.
- [ERW08] Jürgen Ebert, Volker Riediger, and Andreas Winter. Graph Technology in Reverse Engineering, the TGraph Approach. In Rainer Gimnich, Uwe Kaiser, Jochen Quante, and Andreas Winter, editors, *10th Workshop Software Reengineering (WSR 2008)*, volume 126 of *GI Lecture Notes in Informatics*, pages 67–81, Bonn, 2008. GI.
- [FS09] Franck Fleurey and Arnor 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*, pages 606–621, 2009.
- [IBM06] IBM. An Architectural Blueprint for Autonomic Computing. *Autonomic Computing*, 2006. White Paper.
- [MBJ<sup>+</sup>09] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models@ Run.time to Support Dynamic Adaptation. *Computer*, 42(10):44–51, October 2009.
- [OGT<sup>+</sup>99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *Intelligent Systems and their Applications*, *IEEE*, 14(3):54–62, 1999.
- [SEGL10] Klaus Schmid, Holger Eichelberger, Ursula Goltz, and Malte Lochau. Evolving Adaptable Systems: Potential and Challenges. In *2. Workshop Design For Future 2010 - Langlebige Softwaresysteme (L2S2)*, Bad Honnef, Germany, 2010.
- [ST09] Mazeiar Salehie and Ladan Tahvildari. Self-Adaptive Software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):1–42, 2009.
- [VG10] Thomas Vogel and Holger Giese. Adaptation and abstract runtime models. In *Proceedings of the Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 39–48, 2010.