# Mechanisms for Leveraging Models at Runtime in Self-adaptive Software

Amel Bennaceur[1], Robert France[2], Giordano Tamburrelli[3], Thomas Vogel[4], Pieter J. Mosterman[5], Walter Cazzola[6], Fabio M. Costa[7], Alfonso Pierantonio[8], Matthias Tichy[9], Mehmet Akşit[10], Pär Emmanuelson[11], Huang Gang[12], Nikolaos Georgantas[1], and David Redlich[13]

[1] Inria, France
[2] Colorado State University, US
[3] Università della Svizzera Italiana, Switzerland
[4] Hasso Plattner Institute at the University of Potsdam, Germany
[5] MathWorks, US
[6] Università degli Studi di Milano, Italy
[7] Universidade Federal de Goiás, Brazil
[8] Univ. degli Studi di L'Aquila, Italy
[9] Chalmers, University of Gothenburg, Sweden
[10] University of Twente, Netherlands
[11] Ericsson AB, Sweden
[12] Peking University, China
[13] Lancaster University, UK

**Abstract.** Modern software systems are often required to adapt their behavior at runtime in order to maintain or enhance their utility in dynamic environments. Models at runtime research aims to provide suitable abstractions, techniques, and tools to manage the complexity of adapting software systems at runtime. In this chapter, we discuss challenges associated with developing mechanisms that leverage models at runtime to support runtime software adaptation. Specifically, we discuss challenges associated with developing effective mechanisms for supervising running systems, reasoning about and planning adaptations, maintaining consistency among multiple runtime models, and maintaining fidelity of runtime models with respect to the running system and its environment. We discuss related problems and state-of-the-art mechanisms, and identify open research challenges.

## 1 Introduction

Many modern distributed and open software-based systems are required to adapt their behavior at runtime in order to maintain or enhance their utility [19, 58]. *Models at runtime* (M@RT) research focuses on how models describing different aspects of a software system and its environment (e.g., requirements, design, runtime configuration) can be used to manage the complexity of effectively adapting software systems at runtime [11, 40]. This chapter is a distillation of discus-

N. Bencomo et al. (Eds.): Models@run.time, LNCS 8378, pp. 19–46, 2014.

sions held in a working group at the *Dagstuhl Seminar on Models@run.time*[1]. The working group discussions focused on challenges associated with developing M@RT mechanisms to support runtime software adaptation. Specifically, we discussed challenges associated with developing mechanisms for (1) creating runtime models, and updating them in response to changes in the system and its environment, (2) reasoning about changes in the system, its requirements, or the environment to select or produce appropriate adaptation strategies, (3) analyzing and maintaining multiple runtime models, which represent different aspects of the running system or its environment, and (4) establishing and maintaining fidelity of the runtime models with respect to the running system, its requirements, and its environment.

It is important to notice that M@RT can support a plethora of tasks other than software adaptation, such as for example software auditing and monitoring. However, software adaptation is by far the most challenging application of M@RT mechanisms and thus represents the focus of our discussions. Analogously, it also important to mention that M@RT is not the only way to implement self-adaptive systems even if it represents a common approach.

We developed a *conceptual M@RT reference model* to provide a framework for our discussions. The reference model is based on what we considered to be core M@RT concepts and terminology, and it was used to situate the mechanisms we discussed. For each mechanism we identified challenges associated with its development and use in the context of the reference model. In addition, we reviewed the state of the art and formulated open research challenges for the mechanisms. The discussions raised a number of challenging research questions, for example, *What are the key abstractions needed to support effective M@RT adaptation*? and *How can these abstractions be used to create appropriate adaptation mechanisms in open settings, e.g., in Internet of Things and Cyber-Physical Systems*?

In contrast to other work that discusses the state of the art and research challenges for self-adaptive software systems [19, 58, 75, 79], our discussions focused on adaptive systems based on M@RT.

The chapter is structured as follows. Section 2 presents the terminology and the conceptual reference model we used to frame our discussions. Section 3 discusses the challenges associated with developing appropriate M@RT mechanisms. Section 4 reviews the state of the art and discuss open research challenges. Finally, Section 5 concludes with an overview of the major contributions of this chapter.

## 2 Terminology and Reference Model for M@RT

In this section, we define the terminology underlying the conceptual reference model for M@RT that will be presented afterwards and used to frame our discussions in the rest of the paper. The terminology and the conceptual reference model presented here are generic so that they can be applied to a wide variety

[1] Dagstuhl Seminar 11481: `http://www.dagstuhl.de/11481`

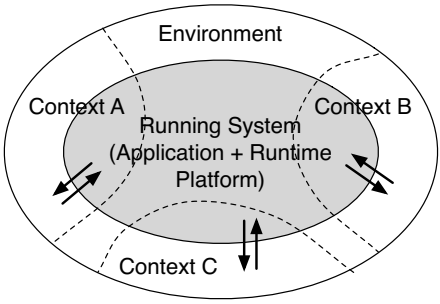


**Fig. 1.** A Terminological Framework for M@RT

of adaptive software systems driven by M@RT, including open, distributed, and embedded systems (e.g., cyber-physical systems) and cloud-based systems.

### 2.1 Terminology

One of the key questions we attempted to answer was the following: *What constitutes a M@RT system?*, i.e., *What are the major parts of an adaptive software system in which adaptation is driven by models?* The terminological framework we converged on during the discussion is shown in Figure 1. The *Running System* shown in the framework represents the executing software system. The *Environment* represents the external elements that the Running System interacts with to fulfill its requirements. The Environment corresponds to the concept of *World* that interacts with the *Machine* (i.e., the Running System) in the seminal work by Jackson and Zave [51, 101].

The usage and operation of a M@RT system can be influenced by one or more *Contexts*, that is, a context can determine how a M@RT system adapts itself to changes in its environment. For example, the types of adaptations that software on a mobile device can undergo may vary based on the device's location and time at which the change occurred; both time and location define the context. Context elements may include elements from the Running System, including hardware resources and network elements, and elements from the environment, for example, the location and time. Moreover, it is important to note that different contexts may also overlap.

A *Running System* consists of two major parts:

**The Application:** This part of the system is concerned with delivering the desired functionality and with adapting how the desired functionality is delivered to users or other systems.

**The Runtime Platform:** This part of the system provides the infrastructure on top of which the *Application* runs. For example, it can consist of middleware, a language runtime environment, an operating system, a virtualization system, and hardware resources.

The *Application* may be further broken down into the following parts:

**Core:** This is the "default" functionality used to meet users' functional requirements. It is the functionality that executes when the system is first started.

**Supervision:** This component is concerned with supervising the behavior of the Running System and monitoring the Environment. It triggers appropriate adaptations by calling functionality in the Adaptation part of the Application (see below). The monitoring performed by this component is model-driven, that is, monitoring involves providing and maintaining runtime models of the running system and its environment. Note that this component is responsible for the monitoring aspect of the MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge) model [50] proposed for autonomic control systems.

**Adaptation:** This component is in charge of reasoning, planning, and enforcing adaptations on parts of the Running System. The adaptation functionality is triggered by the Supervision component. In a M@RT system, the adaptation functionality is driven by models. Note that this component is responsible for the analysis, planning, and execution aspects of the MAPE-K model.

It is important to understand what can and cannot be adapted by a M@RT system. Therefore, the concepts in the terminological framework are classified as *adaptable*, *non-adaptable*, or *semi-adaptable*. Adaptable entities are those whose behaviors can be modified at runtime by the system. The Core part of an Application is an adaptable entity because it has been designed for adaptability by software developers. The Supervision and Adaptation parts can conceivably be designed for adaptation, for example, it may be possible to use what the system "learned" from past applications of adaptation rules to improve the adaptation mechanisms. On the other hand, the environment is typically a non-adaptable entity since it consists of entities external to the system (e.g., users) that cannot be controlled by the system[2] Some elements in the Runtime Platform may be semi-adaptable, that is, it may be possible to partially modify or configure them at runtime (e.g., by tuning certain parameters of the operating system, or setting the configuration of hardware devices).

The conceptual reference model for M@RT presented in the following subsection is based on the above terminological framework.

### 2.2 A Conceptual Reference Model for M@RT

The conceptual reference model we propose is structured into four levels ($M0$, $M1$, $M2$, and $M3$) as illustrated in Figure 2. The $M0$ level consists of the Running System that observes and interacts with the Environment. A detailed view of this level is depicted in Figure 3. The view refines the Running System and its relationship to the Environment. The Supervision and Adaptation components provide the means to effect adaptations on the Core functionality and on the

[2] Notice that, in some particular domains, the environment may be partially controllable as for cyber-physical systems.

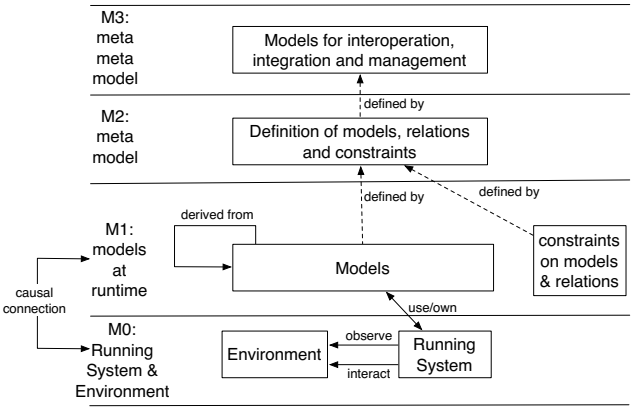


**Fig. 2.** A Conceptual Reference Model for M@RT

Runtime Platform, based on observations made on the Environment, the Core, and the Runtime Platform. The Supervision component triggers the Adaptation component to reason and plan an adaptation based on observations of the Environment, Core, or Runtime Platform. An adaptation, performed by the Adaptation component, adjusts the Core or the Runtime Platform. The Adaptation component may request more detailed information from the Supervision component that triggered its behavior. The Supervision component, on receiving such a request, monitors the Environment, Core, or Runtime Platform at a more fine-grained level in order to provide this information. The Core functionality interacts with the Environment (as is typical of software applications), and with the Runtime Platform (e.g., to use middleware services). The Runtime

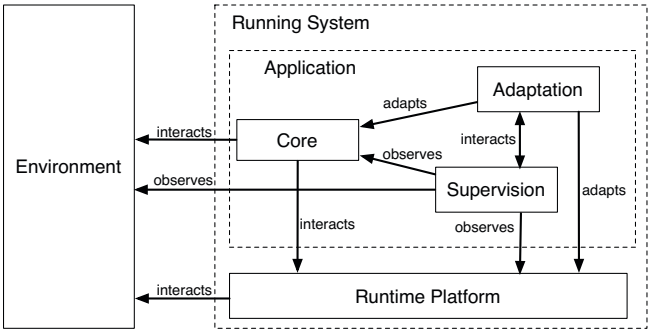


**Fig. 3.** *M*0 Level

Platform also interacts with the Environment (e.g., to establish communication with external devices or services).

While level $M0$ includes adaptation mechanisms, it does not restrict the forms of information used to drive the adaptation and thus, it is applicable to many types of adaptive systems, including those that do not use models. Level $M1$ and above specialize the form of adaptive systems to M@RT systems because they make models the primary drivers of the adaptation.

Level $M1$ includes the runtime models, relations between these models, and constraints on these models and relations. The models are used to drive the adaptation mechanisms at level $M0$. This level may have a variety of diverse models, for example, Queuing Networks, Simulink models, and UML Diagrams. The models may be derived from other models or may be composed of other models defined in the level.

$M1$ models are causally connected with (1) events and phenomena occurring in $M0$, specifically, those observed and handled by the Supervision component, and (2) change actions enacted by the Adaptation component. The Supervision component uses the $M1$ models in its event-handling processes. The processing of these events updates $M1$ models such that the models properly reflect the Running System and Environment. Event processing can lead to the invocation of adaptation functionality in the Adaptation component. Adaptations are performed by changing the models and propagating the changes to the Running System through causal connections between the models and the Running System.

Conceptually, this part of the reference model describes a *feedback control loop* between the models in level $M1$ and the Running System at $M0$, and it is based on the autonomic control loop discussed in [13, 54]. At runtime, the Running System provides data (feedback) used to attain a desired level of fidelity between the models and the system and between the models and the environment. Adaptations produced by the adaptation reasoning and planning mechanisms in the Supervision and Adaptation components are performed on the models and changes in the models are propagated to the Running System.

It is important to note that $M1$ may consist of several models representing different aspect of the Running System and Environment. These models may overlap and, as a consequence, may be in conflict in terms of actions to be triggered in the adaptation step. Inter-model conflicts and dependencies within one level are discussed later in this chapter.

The languages used to create $M1$ models are defined by metamodels. Such metamodels are located in the $M2$ level. Examples of languages defined by metamodels are UML, SysML, SPEM, BPMN, or ADLs. Likewise, languages for specifying constraints on $M1$ models are part of the $M2$ level, for instance, OCL [72] can be used as a language defined at the $M2$ level to describe constraints for $M1$ models created with UML. In addition, the types of relationships that can be defined among the different $M1$ models are defined at the $M2$ level.

The $M2$ level is relevant since it determines the languages that are used to create $M1$ models and thus, it determines the syntax and semantics of these models. Proper syntax and semantic definitions are required for automated processing of

models. For instance, model transformation techniques transform a source model to a target model at the $M1$ level and such a transformation is typically specified by referring to the abstract syntax (i.e., the metamodel) of the source and target $M1$ models [82]. Specifying a transformation using metamodels makes use of the syntax and semantic definitions and it supports the application and reuse of the transformation for all possible models that are instances of these metamodels. Such reuse eases the engineering of mechanisms. In contrast, without an $M2$ level, each individual model in the $M1$ would require a specific transformation.

Finally, the top level in the conceptual reference architecture is $M3$ which is the meta-metamodeling level. This level defines models for interoperation, integration, and management of the modeling stack and, thus, it is used to define the metamodels contained in $M2$. An example for a meta-metamodel model at the $M3$ level is the *Meta Object Facility* (MOF) [71] that is used to define the UML and other languages such as SPEM [69]. In this case, having a common meta-metamodel eases the integration of the UML and SPEM languages at the $M2$ level, which in turn, enables interoperability between UML and SPEM model processing activities.

## 3 M@RT Engineering Challenges

In this section we present the challenges associated with engineering adaptive systems that follow the M@RT reference model described in the previous section. Specifically, we consider (1) the development and evolution of runtime models for supervision, (2) the reasoning and planning of adaptation based on runtime models, (3) the maintenance of multiple and different runtime models, and (4) the maintenance of fidelity of runtime models with respect to the running system and its environment. Mechanisms that realize the reference model can be used to tackle these challenges.

### 3.1 Developing and Updating Runtime Models for Supervision

Supervision is concerned with observing the running system and its environment in order to trigger the necessary adaptation. These observations may relate to functional and non-functional concerns, which should be explicitly captured in runtime models. Realizing the conceptual M@RT reference model requires one to tackle issues related to how the runtime models at levels $M1$, $M2$, and $M3$ (cf. Figure 2) are created and updated at runtime. For $M3$, a meta-metamodel can be developed or an existing one such as the *Meta Object Facility* (MOF) [71], can be used. This meta-metamodel is used to define $M2$ metamodels. The $M2$ metamodels define the languages used to express $M1$ models.

Runtime models describe entities in a *running* software system and in its environment. Unlike development models, they capture dynamic runtime behavior. For this reason, meta-metamodels and metamodels that capture runtime aspects are required at the $M3$ and $M2$ levels, in addition to the meta-metamodels and metamodels used at development. Runtime and development meta-metamodels

and metamodels must be integrated in a M@RT system, and thus it is important to seamlessly connect model-driven development processes with the processes for creating and evolving runtime models.

The state of a runtime model should correspond as closely as possible/needed with the current state of the running system and its environment. Timely information on a running system and its environment provided by sensors can be used by a Supervision component (cf. Section 2) to update the runtime models. This requires instrumentation mechanisms that allow the Supervision component to connect runtime models with running systems and their environments. These mechanisms causally connect levels $M1$ to $M0$. It is a challenge to maintain this causal connection such that the models and the running system with its environment do not drift.

### 3.2 Reasoning and Planning Adaptation Based on Runtime Models

Runtime models reflecting the running system and its environment are also utilized by the adaptation process. Reasoning about the system and its environment to identify the need for adaptation involves manipulating these models. The need to adapt can be raised through actual or predicted violations of functional or non-functional properties. If reasoning determines the need to adapt, changes are planned and analyzed using the runtime models before they are propagated to the running system. Such model-driven adaptations require automatic reasoning and planning mechanisms that work on-line and on top of runtime models.

Reasoning and planning mechanisms themselves can be adapted, as in adaptive or reconfigurable control architectures [55]. Such adaptations can be supported by explicitly describing these mechanisms in runtime models. The most popular adaptation models are rule-based or goal-based models.

### 3.3 Maintaining Multiple and Different Runtime Models

As discussed in Section 2, many different runtime models may have to be maintained in a M@RT-based adaptive system. This necessity arises because of the need to manage multiple concerns, for example, performance, reliability, and functional concerns. Each concern typically requires specific models that are able to capture the individual concern and to provide a basis for reasoning about it.

However, dealing with multiple concerns raises issues of maintaining multiple models at runtime and keeping them consistent with each other. The first issue can be handled by mechanisms that architect the runtime environment by organizing and structuring multiple runtime models in a system. In terms of the reference model, handling this issue involves refining the concepts of the Supervision and Adaptation components to realize concrete model architectures and component implementations. The second issue is concerned with defining dependency and other relationships between runtime models. Models describing a running system and its environment from different viewpoints are likely dependent on each other; they all need to provide views that are consistent with each other. Moreover, when separating concerns in different models for reasoning,

these concerns must be integrated at a certain point in time, at the latest when it comes to planning adaptations. Thus, relationships across models reify dependencies among concerns. This requires mechanisms to manage such relationships between models, especially consistency relationships among models.

### 3.4 Establishing and Maintaining Fidelity of the Runtime Models with the Running System and Its Environment

Runtime models also provide the basis for propagating changes to the running system. Thus, planned adaptations are performed on runtime models and then enacted on the running system. This requires mechanisms to map changes at the model level to changes at the system level. For this mapping mechanism, the typically significant abstraction gap between a running system and the runtime models imposes the need for refining changes on the models. For example, removing a model element that represents a component from a runtime model in order to uninstall the component might result in several system-level changes, including identifying, stopping, and uninstalling the component and to perform further clean-up activities.

Moreover, mechanisms enabling *safe* adaptations of the running system are required. This includes establishing and maintaining fidelity of runtime models with the running system and its environment. This is especially relevant in situations when adaptation fails. If the enactment of planned model changes to the running system fails, the runtime models and the running system may drift and therefore the fidelity decreases. Mechanisms that ensure fidelity, at least to a certain degree, in the face of dynamic environments and failing adaptations are needed in M@RT-based adaptive software systems.

## 4 M@RT Mechanisms: State of the Art and Research Challenges

This section discusses the state of the art and research challenges for M@RT mechanisms in adaptive software systems. The discussion is structured around the engineering challenges we identified in the previous section. For each of them, we discuss existing approaches based on a literature review and we identify open research challenges for the mechanisms. Finally, we present a research challenge that cross-cuts many of the challenges we discuss.

### 4.1 Developing and Updating Runtime Models for Supervision

#### State of the Art

There is currently no systematic way to develop runtime models and especially their metamodels. There are some initial ideas on how to manually move from design-time metamodels to runtime metamodels by following an abstract metamodeling process [57]. To increase the level of automation of such a process,

approaches aim at providing support for inferring metamodels for runtime models by statically analyzing the source code of client programs that use system management APIs [84]. However, these inferred metamodels are preliminary and have to be revised by engineers. Thus, there is a lack of means to systematically, seamlessly, and automatically generate or transform runtime metamodels/models from design-time metamodels/models. Moreover, most M@RT approaches [57, 84, 85, 94, 98] typically use a subset of MOF as a meta-metamodel, but the suitability of MOF as a runtime meta-metamodel has not been analyzed or even assessed so far. The use of MOF is motivated by relying on existing MDE frameworks, like the *Eclipse Modeling Framework Project* (EMF)[3] that provides an implementation of a subset of MOF, which is similar to *Essential MOF* (EMOF) [71].

Besides such MDE frameworks, earlier work originating from the software architecture field employs architecture description languages (ADLs) [61] to describe a running system from an architectural viewpoint. Examples are the work by Oreizy et al. [74] and Garlan et al. [41]. Both approaches connect an architectural model to a running system. Such a connection is the key to M@RT-based systems since it allows one to maintain a runtime model for a running system.

The most direct manner to achieve a causal connection between a model and a running system is to require the running system to be organized in a pre-defined form that is directly linked with the model. For example, Oreizy et al. [74] prescribe an architectural style for the running system. Concepts of this style are first class elements in the system implementation and in the runtime model, and there is a direct one-to-one mapping between the system and model elements. This eases developing the causal connection since there is no abstraction gap between the model and the system. Others, like Garlan et al. [41], take a framework perspective and specifically consider probes and executors as part of a framework. Probes and executors instrument the running system and they realize the mapping and connection between the system and its runtime model. In contrast, recent work [83, 85, 94, 98] relies on management capabilities already offered by Runtime Platforms (cf. Section 2), like the management APIs provided by a middleware or application servers. On top of such APIs, a causal connection is often manually implemented while there exists preliminary work to simplify the development by increasing the level of automation using code generation techniques [85]. Similar to developing runtime models and metamodels, there is no systematic way to develop a causal connection when engineering a system that should provide M@RT-based reflection capabilities, that is, models at higher-levels of abstraction than those known from computational reflection [11].

Another relevant stream of research that considers models of running systems is the field of reverse engineering. The goal is to extract models from an existing system to obtain a high-level view of it. Besides creating models statically from the source code, they can also be extracted by tracing a running system. One approach is to leverage features provided by reflective programming languages to extract runtime models [52], which, however, requires that the (legacy) system

[3] `http://www.eclipse.org/modeling/emf/` (last visited on July 2nd, 2012).

is implemented in such a language. Another reverse engineering approach is MoDisco [14]. This project provides a library for a number of widely-used systems to assist the development of so called *model discoverers*, which represent the implicit system state in models.

A key task for developers constructing runtime models/metamodels and causal connections is to understand what kinds of data can be obtained from the running system and how to obtain them. Some existing approaches to inferring system data and management APIs may be helpful for this task. Garofalakis et al. [42] provide an automated approach to infer the schema of XML files, and Fisher et al. [36] provide tools to extract types from the plain-text files and to automatically generate the text processing tools from the extracted types. Antkiewicz [2] provides a code analysis approach to infer how to use the system APIs provided by different frameworks. All these approaches may help in systematically developing runtime models and metamodels and, as discussed above, similar ideas for M@RT have already been proposed [84].

**Research Challenges**

*Finding the right abstractions:* A key research challenge is identifying the abstractions that models need to represent in order to support effective adaptation. Once identified, further research is needed to determine the most effective representations for the abstractions. These representations should precisely capture the information needed to describe the phenomena and should do so in a manner that allows the M@RT-based system to efficiently process the representation as it steers the behavior of the system. Finding and describing the right abstractions is key to building effective M@RT systems. Abstractions that are fine-grained may be able to deal with a variety of adaptations, but can lead to the production and manipulation of large amounts of data that are difficult to manage and costly to process. Higher-level abstractions can have representations that can be more efficiently processed, but can also ignore details that may be the actual causes of behaviors that require adaptation. Determining the right abstractions is typically a trade-off between the effectiveness of the representations and the types of adaptations that can be effectively supported.

*Creating and maintaining models at runtime:* In a M@RT-based system, the models should be faithful representations of the system and environment they abstract over. Techniques for creating faithful models and for maintaining the fidelity of the models as the system and its environment change are critical for the successful use and operation of M@RT systems. Maintaining fidelity involves monitoring (observing) runtime phenomena to be represented by models and updating the models in a timely manner when monitoring detects the phenomena.

To support effective monitoring we need to develop guidelines for determining what to monitor as well as how often and at which level of precision to monitor. These issues can dramatically impact the system performance and fidelity of the models. In addition, M@RT-based systems may also need to transform, summarize, and correlate the observations collected into pieces of information that

meaningfully correspond to abstractions supported by the models at runtime. Techniques for transforming and aggregating information in an efficient manner are therefore needed. Inefficient techniques can lead to significant drift in model fidelity or less powerful adaptation opportunities.

Distribution of resources also adds to the complexity of aggregating information. For many kinds of modern systems, such as Internet of Things and Cloud Computing systems, components are often distributed across different nodes or devices. In order to maintain a global model at runtime, we need to integrate the local information from different nodes. The challenges here include when to perform the integration, what local information to retrieve and integrate, how to ensure the temporal correctness and timeliness of the global model, and how to achieve a better performance by reducing the communication between different nodes as well as the information exchanged during the communication.

### 4.2 Reasoning and Planning Adaptation Based on Runtime Models

#### State of the Art

Runtime models reflecting the running system and its environment are the basis for reasoning and planning adaptations of the system. Different techniques for reasoning and planning have been proposed and according to Fleurey and Solberg [38], they can be generally classified into two types of adaptation models.

First, rule-based approaches specify the adaptation by some form of event-condition-action (ECA) rules or policies [18, 27, 37, 41, 43, 45, 64]. An event triggers the adaptation process and conditions determine which reconfiguration action should be performed. According to Fleurey and Solberg [38], such approaches can be efficiently implemented with respect to runtime performance, and they can be simulated and verified early in the development process. However, if the number of rules grows, the approach suffers from scalability issues concerning the management and validation of the rules. The variability space of a system may be too large to enumerate all possible configurations, which is, however, required to some extent for rule-based approaches that explicitly specify the adaptation.

Therefore, the second type of adaptation models has emerged, which avoids the explicit specification of the adaptation. These search-based approaches prescribe goals that the running system should achieve, and guided by utility functions they try to find the best or at least a suitable system configuration fulfilling these goals [21, 22, 39, 76]. Other search-based mechanisms are based on model checking techniques to find plans on how to adapt the running system [90]. In general, search-based approaches solve the scalability problem of rule-based approaches, but they suffer from costly reasoning and planning processes, and weaker support for validation (cf. [38]). Since these processes have to be carried out at runtime, the runtime performance is crucial for any reasoning and planning mechanism.

Based on the different characteristics of rule-based and search-based approaches, Fleurey and Solberg [38] propose a mixture of them to balance their advantages

and disadvantages. Their general idea is to use rules to reduce the variability space of the system and environment that subsequently has to be searched for suitable configurations.

Overall, more work needs to be done to understand different reasoning and planning techniques or mechanisms and their characteristics. This is a prerequisite for selecting and tuning existing techniques or developing new techniques for a specific system. Moreover, the impact of M@RT and the benefits offered by M@RT on reasoning and planning mechanisms have to be more thoroughly investigated. Therefore, requirements for adaptation models, that is, for reasoning and planning mechanisms operating on runtime models, have been proposed in [96]. Such requirements are helpful to evaluate existing adaptation models and to systematically develop new ones.

**Research Challenges**

*Reasoning about adaptations:* Research is needed to produce efficient and effective analysis techniques for reasoning about adaptations in environments that are highly dynamic and that offer limited computational resources. The limited computational resources and time constraints make design-time formal analysis techniques too costly to apply at runtime. The identification of appropriate heuristics can dramatically improve model analysis at runtime. The language used to express the models has a direct bearing on analysis efficiency and thus should be considered when developing the metamodels to be used for runtime models. Another consideration related to model analysis concerns the exploitation of structural deltas between model changes. Techniques that allow analysis to focus only on the parts of the model that have changed can significantly reduce the time for analysis when the deltas affect small parts of the models.

*Performance and reliability analysis:* We identified the following key technologies to analyze the performance and reliability of a running system: Probabilistic model checkers, for example, PRISM [49, 56], and Queueing Network solvers, for example, MT [7, 8]. These technologies support efficient and effective model checking of complex performance and reliability models against required properties described in appropriate formalisms. Their adoption at runtime may require specific forms of optimization [15, 32, 33, 44], and thus investigating their applicability may lead to other research challenges. In the specific context of cloud computing, auto scaling technologies, which provide the means to automatically scale up or scale out a given architecture, may be used to implement automatic performance adaptation in the cloud [59].

*User-centric models:* During the Dagstuhl seminar, it was largely acknowledged that human users will inevitably be part of the process of system evolution through adaptation. To the extent that models are appropriate artifacts to communicate system requirements and functionality at a high level of abstraction, it makes sense to use them as handles for the end-user to exert some form of

control over how the system behaves at runtime. The exercise remains in terms of how to enable such high-level models to be causally connected with the system in meaningful ways and in particular how to fill the gap between the model and implementation in order to render effectively the provided control.

An example in this direction is the Communication Virtual Machine technology [25]. It enables non-expert end-users to input high-level communication models that are then interpreted to configure the desired communication sessions out of a selection of underlying communication providers. It also allows users to dynamically update the communication session by changing the model at runtime. The interpretation of such high-level, user-defined models is made possible by the adoption of a layered architecture, which contributes bridging the abstraction gap between the model and the underlying basic services in an incremental way, as well as by focusing on a specific domain, which limits the scope of choices in the interpretation process. While this approach is currently limited to the communication domain, generalizations for other domains, as well as to aspects of the middleware itself can be the subject for further research.

*Analysis and Planning based on M@RT:* Analysis and planning is concerned with reasoning about the running system and its environment and, if needed, with planning an adaptation of the system. Therefore, reasoning mechanisms are employed that operate on runtime models.

Different reasoning mechanisms have been proposed such as rule-based or search-based techniques as discussed previously. Such techniques have different effectiveness and efficiency characteristics. To systematically select or develop appropriate reasoning techniques when engineering adaptive systems requires an understanding of these characteristics. For example, the results of reasoning may differ between the techniques. A technique may provide one optimal solution at the end of the reasoning, while another technique may provide a list of all possible solutions. Considering efficiency, a technique may incrementally return solutions as soon as they are found. Moreover, techniques need not be deterministic in the sense that repeated runs of reasoning may result in different solutions for the same problem. Thus, it is important to identify and understand these characteristics when applying reasoning techniques in different application contexts. This leads to a major challenge in understanding which specific reasoning technique is best for which problems, adaptation models, or domains of adaptive systems.

In this context, influential factors, like the exponential growth of the problem size (number of environment conditions, constraints, or adaptation options), the time and resource limitations for reasoning, the accuracy or in general the quality of the resulting solution, or assurance for the resulting solution, are critical. This likely requires trade-offs between these factors, for example, between the quality of a solution and the acceptable time in which a solution has to be found.

Considering these different influential factors as well as the different reasoning techniques, it is a challenge to identify the most suitable technique and acceptable trade-offs for a certain system or problem. On the one hand, this is additionally impeded by a lack of traceability between the reasoning results and the

reasoning goals or problems. Thus, it is often difficult to understand why a certain solution has been chosen by a reasoning technique for a given problem. This is even more complicated for adaptive systems with their inherent uncertainty related to the systems' functional and non-functional goals and actual behavior as well as to the systems' operational environments. Thus, incomplete and insufficient knowledge about a system and its environment makes the development or even the selection of suitable reasoning techniques challenging. Furthermore, it impedes the software engineer's understandability and traceability of the reasoning decisions.

All these issues motivate the need for smart reasoning techniques that leverage, among others, learning techniques, incremental techniques, abstraction, problem partitioning, and decentralized reasoning to enable acceptable trade-offs considering effectiveness and efficiency of the reasoning results. Thereby, each individual system and even each situation of a running system may need different trade-offs, which requires reasoning techniques to be adaptive. Systematically engineering or employing such techniques is challenging since it requires one to grasp the influential factors for reasoning, the uncertainty in adaptive systems, and the traceability between all of the constituent parts in reasoning.

### 4.3 Maintaining Multiple and Different Runtime Models in an Adaptive System

#### State of the Art

M@RT-based systems are likely to use several runtime models for different aspects of the system and at different abstraction levels (cf. Figure 2 or [99]). This calls for mechanisms to structure and operationalize multiple runtime models and the relationships among those models. A similar problem exists in model-driven software development where a plethora of interdependent models are employed to describe the requirements, design, implementation, and deployment of a software system. The field of *Multiparadigm Modeling* has made much progress in defining, relating, transforming, and analyzing models of potentially different paradigms [65–67] based on the premise that out of a set of issues to tackle, each problem is best solved by employing the most appropriate abstractions using the most appropriate formalisms. This generally leads to a complex overall organization of a large set of models. Therefore, the concept of *megamodels*, which are models that contain other models and relationships between those models, has emerged in the model management research field [4, 9, 10, 31]. The goal is to capture the different development models and their dependencies to address traceability and consistency in the development process.

Recently, such megamodel concepts have been proposed for runtime models employed in self-adaptive software [99]. In this context, megamodels are used to specify and execute adaptation processes by means of feedback loops. Besides structuring runtime models, megamodels describe the activities of a feedback loop as a flow of model operations working on runtime models. Additionally, such megamodels are kept alive at runtime to actually maintain runtime models

and to directly execute a feedback loop using a megamodel interpreter. Overall, megamodels together with an interpreter support the explicit specification and execution of feedback loops, while the flexibility provided by interpreted models also leverages the adaptation and composition of feedback loops [95, 97].

While megamodels help in structuring the interplay of runtime models, mechanisms are required that substantiate the megamodel's model operations, that is, the relationships between runtime models. Such operations are, for example, reasoning and planning mechanisms discussed previously. A particular relationship between runtime models is concerned with consistency among models describing the same running system from different viewpoints.

Consistency can be tackled by model transformation and synchronization mechanisms. Transformations are suitable for initially deriving runtime models from other models, while synchronizations support the continuous consistency by propagating changes between models. A lot of research has gone into the development of model transformation and synchronizations languages (cf. [23, 24, 63, 86]). Many such languages are based on graphs and graph transformations [29, 45, 78] that have a sound formal basis in category theory. Thus, they enable formal reasoning [5, 77] in addition to their execution. Prominent approaches are Progress [81], Story Diagrams [35], AGG [88], and Henshin [3]. A graph transformation contains a left hand side and a right hand side which are both specified as graphs. If an occurrence of the left hand side is found in the host graph, that is, in the model, it is replaced by the right hand side. Several approaches have been developed to ensure structural constraints [5, 48] which can be used to ensure consistency.

The aforementioned transformation languages mainly address the transformation of single models. Triple Graph Grammars (TGGs) [47, 80] are an approach to handle two models (with extensions to an arbitrary number of models) potentially conforming to different metamodels. TGGs specify how a subgraph in one model corresponds to a subgraph in another model. They can be used for a forward transformation from a source model to a target model, a backward transformation from a target model to a source model as well as for keeping models synchronized [46]. By construction, TGGs ensure that the specified correspondence relations exist, which can be used for consistency purposes. However, TGGs are best suited for models whose metamodels share structural similarities. Query/View/Transformation (QVT) [70] is a set of standardized languages. While QVT-Operational enables the operational specification of model transformations, QVT-Relational targets a declarative specification of relations between models similar to TGGs. Further model transformation approaches are the Atlas Transformation Language (ATL) [53], PMT [93], and the Janus Transformation Language (JTL) [20]. The latter focuses on non-bijective bidirectional model transformations. In the context of M@RT, this also enables the handling of models that do not share structural similarities.

Several approaches have been developed that deal with inconsistencies by constructing repair actions [28, 34, 68]. They address the problem of consistency preservation in the context of user induced changes. Consequently, those

approaches rely on the user to select the appropriate repair action. Thus, they are employable in the context of M@RT systems that incorporate the user in the adaptation process.

In general, model transformation and synchronization mechanisms are designed for off-line usage. They are employed in model-driven development processes but not on-line within a running M@RT system. Performing model transformations and synchronizations on-line requires light-weight and efficient mechanisms. For instance, Song et al. [83] apply a model transformation mechanism based on QVT-Relational [70] (QVT-R) on-line to support runtime software architectures. Vogel et al. [94, 98] employ on-line a model synchronization mechanism based on TGGs [45] to support self-adaptation. In particular, the efficiency of this synchronization mechanism for runtime models is shown in [98].

Overall, model transformation and synchronization mechanisms are promising for M@RT systems to maintain and keep multiple runtime models consistent to each other. However, more research is required to address scalability, efficiency, and especially assurances for such mechanisms.

**Research Challenges**

*Maintaining model consistency:* A M@RT system may require different types of models to support adaptation. In these cases, mechanisms for ensuring consistency between the models before, during, and after adaptations are needed. Short-term research in this area should focus on gaining a better understanding of what it means for models to be consistent in dynamically changing systems. This requires an understanding of the degrees of inconsistency that can be tolerated (if any) and when consistency must be established. The notion of consistency should also be applied to the cases where runtime models are organized in abstraction layers, that is, when the models are related by abstraction or refinement relationships. In these cases, it is important to understand when and how consistency is established across the abstraction layers.

*Runtime model interoperability:* The problem of model interoperability at runtime and its management present researchers with significant challenges. Any solution must include practical methods and techniques that are based on theoretical foundations. Keeping different models in a coherent and consistent state is an intrinsically difficult problem. In general, model interoperability can be pursued through *i)* consistency specification – describing not only the views but also the correspondences they have with one another, and *ii)* consistency assurance and enforcement – guaranteeing consistency before, during, and after adaptations. In essence, whenever a model describing an aspect of the Running System undergoes modifications (regardless of whether the change is performed manually or automatically), the overall consistency may be compromised. Any procedure to restore the consistency must propagate the changes and consistently adapt the other models.

Bidirectional model transformation languages seem the most adequate instrument for addressing this problem. For instance, QVT-Relational [70] (QVT-R) support the specification of correspondences as *relations* and the management of

the consistency by means of the rule *check-only* and *check-then-enforce* semantics. Unfortunately, although non-bijectivity in bidirectional model transformation is recognized to be useful and natural (see [92]) the way it is supported and implemented in current languages is not always satisfactory: even the QVT-R specification is in this respect ambivalent [87]. The main difficulty is addressing non-determinism in change propagation. This occurs when model changes that are propagated through model correspondences give rise to more than one alternative adaptation of the linked models. As typically required in current bidirectional languages (e.g., QVT-R [70], TGGs [80]), the ambiguities among transformation solutions are solved programmatically by means of choices that a designer can make when writing the transformation. In other words, these solutions require the mapping to be bijective by adding additional constraints, which have to be known before the transformation is implemented. In this way, the problem of consistency enforcement among different models is reduced to the problem of model synchronization which is inherently difficult. However, in many cases the constraints to make the mapping bijective are unknown or cannot be formalized beforehand, thus it is important to deal with non-bijectivity by managing multiple solutions.

Existing work proposes mechanisms to deal with non-bijectivity in an explicit way. For instance, PROGRES [6] is a TGG solution to create integration tools capable of dealing with non-deterministic cases, that is, when multiple rules can be applied in the current direction of a transformation. A similar approach is proposed by JTL [20], a bidirectional model transformation language specifically designed to support non-bijective transformations and change propagation. In particular, the language propagates changes occurring in a model to one or more related models according to the specified transformation regardless of the transformation direction, that is, JTL transformations can generate all possible solutions at once. Both PROGRESS and JTL have the drawback of requiring human intervention: The former requires the designer to choose the rule to be applied among the candidate rules, whereas the latter requires the modeler to choose the correct model in the solution space. Adopting these approaches requires that the knowledge necessary to resolve the non-determinism at runtime is made accessible to the transformations. For example, this knowledge can take the form of heuristics. The overall problem is worsened by the fact that model adaptations reflect or drive adaptations on the Running System (regardless of the causal dependency). This is clearly a coupled evolution case, where adaptations written in a transformation language (at the $M1$ layer) must correspond to adaptation at the $M0$ layer which can be expressed, for instance, in terms of aspect-oriented programming techniques.

### 4.4 Establishing and Maintaining Fidelity of the Runtime Models with the Running System and Its Environment

#### State of the Art

An essential aspect of M@RT is the causal connection between runtime models and the running software system. On the one hand, this includes the Supervi-

sion (cf. Figure 3) to reflect changes of the running system or environment in the model as discussed in Section 4.1. On the other hand, this includes the Adaptation, that is, that planned changes are performed on the runtime models before they are executed to the system. Both Supervision and Adaptation realize the causal connection and must ensure the fidelity of the models with the running system and its environment.

While the Supervision has been discussed in Section 4.1, two general kinds of mechanisms are employed to enact changes of a runtime model to the running system. First, state-based approaches compare the runtime model before the change with the runtime model after the change. Thus, changes to the model are actually performed on a copy of the model or applying changes results in a copy. Mechanisms for comparing models are provided, for example, by EMF[4]. The resulting differences are the changes that have been performed and they serve as a basis to derive a reconfiguration script to be executed to the running system. Such an approach is followed by [64]. Second, operation-based approaches monitor a model to directly obtain the operations that constitute the changes, for example, setting attribute values or relationships. For example, EMF provides a notification mechanism that emits events representing these change operations. These events serve as a basis to obtain a reconfiguration script or to map the performed operations to system-level changes [94].

In this context, the problem of refining changes performed on abstract runtime models to system-level changes is discussed in [94]. The problem is tackled by model synchronization and graph transformation techniques between abstract runtime models used for reasoning and planning adaptation and a system-level runtime model at the same abstraction level as the system implementation. Thus, the changes are refined between models before they can be directly mapped to the management capabilities provided by the running system.

Such an abstraction gap between runtime models and the running system has to be addressed for M@RT-based systems. Developing a causal connection between a model that is at the same abstraction level as the system is simpler, which is the motivation to follow this approach in [74]. However, such a model does not provide problem-oriented views at appropriate abstractions, which is the goal of M@RT [11]. Providing runtime models that abstract from platform- or implementation-specific details, and thus from the solution space, must cope with an abstraction gap. This abstraction gap created by the Supervision through discarding system-level details may complicate the Adaptation when moving from abstract runtime models down to the concrete Running System (cf. [94]).

Besides realizing the Supervision and Adaptation components by connecting runtime models to the running system, these components have to cooperate to maintain fidelity of the models and the system. If the Adaptation part fails in executing model changes to the system, the models and the system drift, which has to be recognized by the Supervision part. Then, both parts have to cope with the failure to ensure again fidelity and consistency between the model and

[4] EMF Compare Project, `http://www.eclipse.org/emf/compare/` (last visited on July 2nd, 2012).

the system. In general, the M@RT research field lacks work on assurances for the causal connection and for the co-evolution of the runtime models and the Running System over time. This is mandatory for safe adaptations and for coping with partially correct/valid runtime models in the face of uncertainty inherent to dynamically adaptive systems. Moreover, there is also a lack of work addressing the systematic engineering of causal connections, which has to be seamlessly integrated with work on engineering of the Application (cf. Section 2) and the runtime models/metamodels (cf. Section 4.1).

**Research Challenges**

*Propagating model changes to the Runtime System:* Several research issues need to be investigated for developing effective causal links between models and the running system. We obviously need to identify how to propagate changes from the model down to the system efficiently and effectively. This requires the identification of the points in the Running System where changes need to be applied, as well as constraints on when the changes can be applied. One possible approach to solve the problem of identifying the points of adaptation in the running system is to adopt a programming model that allows for changes at specific points in an execution of a program. Component-based and aspect-oriented programming models are typical examples. In addition, we need to develop mechanisms that support rollback of current operations when changes occur while the system is processing transactions.

*Maintaining model fidelity:* Middleware technologies can be used to facilitate the adaptation of applications in response to changes in the environment. In particular, reflective middleware technologies [60] use causally connected self-representations [16] to support the inspection and adaptation of the middleware system [89]. Components defined at the model level are directly mapped to specific artifacts that realize those components at the implementation level. From a software-quality perspective, this mapping is a form of traceability. In general, the term traceability can refer to any mechanism for connecting different software artifacts. In this context we specifically mean traceability from model to implementation elements, and vice-versa [91]. Maintaining the traceability link allows the model and implementation to co-evolve. Model evolution can be triggered by changes in, for example, (1) the requirements, (2) the environment, and (3) resource availability.

Keeping the model and the running system synchronized is a challenging problem, involving issues such as safety and consistency, especially when changes can be initiated in either the model or at the implementation level [17, 26, 62]. An interesting approach that has received a lot of attention over the years is generating (parts of) implementations directly from their designs using model-driven development technologies [73]. Such technologies can be used to generate (partial) implementations from detailed design models, and thus is an attractive strategy for maintaining the fidelity of models with respect to the running system they

describe. For example, given a sufficiently detailed architectural specification, including structural, interface, and complete behavioral specifications, it is possible to generate a full implementation of a component, connector, or even an entire system [100]. In theory, architectural drift and erosion can be eliminated, by generating new implementation parts from the models as the models evolve [12]. For this to be practical, the description of the detailed models must require significantly less effort than writing the implementations in a programming language. This is often not the case, primarily because the abstractions supported by the modeling languages used to describe detailed behavior are often at a level that is close to the abstractions provided by programming languages. More research is needed for developing behavioral modeling languages that are based on abstractions that allow a developer to build a model that can be efficiently transformed to code using significantly less effort than that of directly writing the implementation in a programming language.

Another approach is to generate models from running code [17, 30, 62]. The challenge here is to generate models that are based on abstractions that are at a higher level than those found in the runtime environment of the programming languages. For example, it is relatively straightforward to obtain class diagram and sequence diagram descriptions of code, but it quickly becomes clear to anyone looking at the diagrams that they simply present views of the code with very little abstraction. Generating abstractions from code is a very difficult challenge. Some progress can be made in the context of domain-specific applications where known patterns and heuristics can be used to identify potentially useful abstract concepts.

### 4.5 A Cross-Cutting Research Challenge: Developing Development Processes for M@RT Systems

Research that focuses on producing effective processes, methods, and techniques for developing M@RT-based adaptive systems is needed in the short term to support systematic development and operation of these systems. Methods, techniques, and tools should be tied together to provide an end-to-end development approach that supports evolution before and after the M@RT-based system becomes operational. This problem has also been identified in [1] for self-adaptive software systems in general.

## 5 Conclusion

This chapter presents a summary of the Dagstuhl discussions on the mechanisms used to manage runtime software adaptation. The chapter is based on a conceptual model for M@RT developed at the seminar to provide a common concept and terminology framework for the discussions.

By relying on the reference model the chapter provides an analysis of the related open problems for each M@RT mechanism identified. We analyzed and classified them into four distinct areas: (1) developing and updating runtime

models, (2) reasoning and planning for adaptation, (3) maintaining different runtime models, and (4) establishing fidelity and consistency among models and the running system.

The identified problems and their classification into such areas were also used to structure discussions on existing related work. By matching the identified problems with the existing work we formulate a set of open research challenges and goals classified in the same four areas. The identified research directions constitute an early roadmap which is the main contribution of the chapter. The roadmap's goal consists of stimulating, organizing, and driving the ongoing efforts of the research community on M@RT. Clearly, such a roadmap will be refined and extended as research that tackles the identified research goals uncovers further challenges.

## References

1. Andersson, J., Baresi, L., Bencomo, N., de Lemos, R., Gorla, A., Inverardi, P., Vogel, T.: Software Engineering Processes for Self-Adaptive Systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Self-Adaptive Systems. LNCS, vol. 7475, pp. 51–75. Springer, Heidelberg (2013)
2. Antkiewicz, M., Czarnecki, K., Stephan, M.: Engineering of framework-specific modeling languages. IEEE Transactions on Software Engineering 35(6), 795–824 (2009)
3. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010)
4. Barbero, M., Fabro, M.D.D., Bézivin, J.: Traceability and Provenance Issues in Global Model Management. In: Proc. of 3rd ECMDA Traceability Workshop (ECMDA-TW), pp. 47–55 (2007)
5. Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic invariant verification for systems with dynamic structural adaptation. In: Proceedings of the 28th International Conference on Software Engineering (ICSE), Shanghai, China. ACM (2006)
6. Becker, S.M., Herold, S., Lohmann, S., Westfechtel, B.: A graph-based algorithm for consistency maintenance in incremental and interactive integration tools. Software and System Modeling 6(3), 287–315 (2007)
7. Bertoli, M., Casale, G., Serazzi, G.: The jmt simulator for performance evaluation of non-product-form queueing networks. In: Annual Simulation Symposium, pp. 3–10. IEEE Computer Society, Norfolk (2007)
8. Bertoli, M., Casale, G., Serazzi, G.: An overview of the jmt queueing network simulator. Tech. Rep. TR 2007.2, Politecnico di Milano - DEI (2007)
9. Bézivin, J., Gérard, S., Muller, P.A., Rioux, L.: MDA components: Challenges and Opportunities. In: Proc. of the 1st Intl. Workshop on Metamodelling for MDA, pp. 23–41 (2003)
10. Bézivin, J., Jouault, F., Valduriez, P.: On the Need for Megamodels. In: Proc. of the OOPSLA/GPCE Workshop on Best Practices for Model-Driven Software Development (2004)
11. Blair, G., Bencomo, N., France, R.B.: Models@run.time: Guest Editors' Introduction. Computer 42(10), 22–27 (2009)

12. Blair, G.S., Blair, L., Issarny, V., Tůma, P., Zarras, A.: The role of software architecture in constraining adaptation in component-based middleware platforms. In: Coulson, G., Sventek, J. (eds.) Middleware 2000. LNCS, vol. 1795, pp. 164–184. Springer, Heidelberg (2000)
13. Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Engineering Self-Adaptive Systems through Feedback Loops. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009)
14. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: Modisco: a generic and extensible framework for model driven reverse engineering. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 173–174. ACM (2010)
15. Calinescu, R., Grunske, L., Kwiatkowska, M.Z., Mirandola, R., Tamburrelli, G.: Dynamic qos management and optimization in service-based systems. IEEE Trans. Software Eng. 37(3), 387–409 (2011)
16. Cazzola, W.: Evolution as Reflections on the Design. In: Bencomo, N., Chang, B., France, R.B., Aßmann, U. (eds.) Models@Run-Time. LNCS, vol. 8378, pp. 259–278. Springer, Heidelberg (2014)
17. Cazzola, W., Pini, S., Ghoneim, A., Saake, G.: Co-Evolving Application Code and Design Models by Exploiting Meta-Data. In: Proceedings of the 12th Annual ACM Symposium on Applied Computing (SAC 2007), Seoul, South Korea, pp. 1275–1279. ACM Press (March 2007)
18. Chauvel, F., Barais, O.: Modelling Adaptation Policies for Self-Adaptive Component Architectures. In: Proceedings of the Workshop on Model-Driven Software Adaptation (M-ADAPT 2007) at the 21st European Conference on Object-Oriented Programming (ECOOP 2007), Berlin, Germany, pp. 61–68 (2007)
19. Cheng, B.H.C., et al.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
20. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: JTL: A bidirectional and change propagating transformation language. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 183–202. Springer, Heidelberg (2011)
21. Cugola, G., Ghezzi, C., Pinto, L.S., Tamburrelli, G.: Adaptive service-oriented mobile applications: A declarative approach. In: Liu, C., Ludwig, H., Toumani, F., Yu, Q. (eds.) Service Oriented Computing. LNCS, vol. 7636, pp. 607–614. Springer, Heidelberg (2012)
22. Cugola, G., Ghezzi, C., Pinto, L.S., Tamburrelli, G.: Selfmotion: A declarative approach for adaptive service-oriented mobile applications. Journal of Systems and Software (2013), http://www.sciencedirect.com/science/article/pii/S0164121213002653
23. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal – Model-Driven Software Development 45(3), 621–645 (2006)
24. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: A cross-discipline perspective. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 260–283. Springer, Heidelberg (2009)

25. Deng, Y., Sadjadi, S.M., Clarke, P.J., Zhang, C., Hristidis, V., Rangaswami, R., Prabakar, N.: A communication virtual machine. In: 30th Annual International on Computer Software and Applications Conference, COMPSAC 2006, vol. 1, pp. 521–531. IEEE (2006)
26. D'Hondt, T., De Volder, K., Mens, K., Wuyts, R.: Co-Evolution of Object-Oriented Software Design and Implementation. In: Akşit, M. (ed.) Proceedings of the International Symposium on Software Architectures and Component Technology, Twente, The Netherlands, pp. 207–224. Kluwer (January 2000)
27. Dubus, J., Merle, P.: Applying OMG D&C specification and ECA rules for autonomous distributed component-based systems. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 242–251. Springer, Heidelberg (2007)
28. Egyed, A., Letier, E., Finkelstein, A.: Generating and evaluating choices for fixing inconsistencies in uml design models. In: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), L'Aquila, Italy, 15-19 September, pp. 99–108. IEEE (2008)
29. Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.): Nagl Festschrift. LNCS, vol. 5765. Springer, Heidelberg (2010)
30. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: ICSE, pp. 111–121. IEEE (2009)
31. Favre, J.M.: Foundations of Model (Driven) (Reverse) Engineering: Models – Episode I: Stories of The Fidus Papyrus and of The Solarus. In: Language Engineering for Model-Driven Software Development. No. 04101 in Dagstuhl Seminar Proceedings, IBFI, Schloss Dagstuhl (2005)
32. Filieri, A., Ghezzi, C., Tamburrelli, G.: Run-time efficient probabilistic model checking. In: 2011 33rd International Conference on Software Engineering (ICSE), pp. 341–350 (2011)
33. Filieri, A., Ghezzi, C., Tamburrelli, G.: A formal approach to adaptive software: continuous assurance of non-functional requirements. Formal Asp. Comput. 24(2), 163–186 (2012)
34. Finkelstein, A.C.W., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B.: Inconsistency handling in multiperspective specifications. IEEE Transactions on Software Engineering 20(8), 569 (1994)
35. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the unified modeling language and java. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000)
36. Fisher, K., Gruber, R.: Pads: a domain-specific language for processing ad hoc data. In: ACM SIGPLAN Notices, vol. 40, pp. 295–304. ACM (2005)
37. Fleurey, F., Dehlen, V., Bencomo, N., Morin, B., Jézéquel, J.-M.: Modeling and Validating Dynamic Adaptation. In: Chaudron, M.R.V. (ed.) MODELS 2008. LNCS, vol. 5421, pp. 97–108. Springer, Heidelberg (2009)
38. 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 2009. LNCS, vol. 5795, pp. 606–621. Springer, Heidelberg (2009)
39. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E.: Using Architecture Models for Runtime Adaptability. IEEE Software 23(2), 62–70 (2006)
40. France, R., Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. In: FOSE 2007: 2007 Future of Software Engineering, pp. 37–54. IEEE Computer Society, Washington, DC (2007)

41. 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)
42. Garofalakis, M., Gionis, A., Rastogi, R., Seshadri, S., Shim, K.: Xtract: A system for extracting document type descriptors from xml documents. In: ACM SIGMOD Record, vol. 29, pp. 165–176. ACM (2000)
43. Georgas, J.C., Hoek, A., Taylor, R.N.: Using Architectural Models to Manage and Visualize Runtime Adaptation. Computer 42(10), 52–60 (2009)
44. Ghezzi, C., Pinto, L.S., Spoletini, P., Tamburrelli, G.: Managing non-functional uncertainty via model-driven adaptivity. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) ICSE, pp. 33–42. IEEE/ACM (2013)
45. Giese, H., Lambers, L., Becker, B., Hildebrandt, S., Neumann, S., Vogel, T., Wätzoldt, S.: Graph Transformations for MDE, Adaptation, and Models at Runtime. In: Bernardo, M., Cortellessa, V., Pierantonio, A. (eds.) SFM 2012. LNCS, vol. 7320, pp. 137–191. Springer, Heidelberg (2012)
46. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and Systems Modeling (SoSyM) 8(1) (2009)
47. Greenyer, J., Kindler, E.: Comparing relational model transformation technologies: implementing query/view/transformation with triple graph grammars. Software and System Modeling 9(1), 21–46 (2010)
48. Heckel, R., Wagner, A.: Ensuring consistency of conditional graph rewriting - a constructive approach. Electr. Notes Theor. Comput. Sci. 2, 118–126 (1995)
49. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
50. IBM: An architectural blueprint for autonomic computing. Tech. rep., IBM (2003)
51. Jackson, M., Zave, P.: Deriving specifications from requirements: an example. In: ICSE 1995: Proceedings of the 17th International Conference on Software Engineering, pp. 15–24. ACM, New York (1995)
52. Jouault, F., Bézivin, J., Chevrel, R., Gray, J.: Experiments in Run-Time Model Extraction. In: Proceedings of 1st International Workshop on Models@run.time (2006)
53. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
54. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
55. Kokar, M.M., Baclawski, K., Eracar, Y.A.: Control Theory-Based Foundations of Self-Controlling Software. Intelligent Systems and their Applications 14(3), 37–45 (1999)
56. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In: Field, T., Harrison, P.G., Bradley, J., Harder, U. (eds.) TOOLS 2002. LNCS, vol. 2324, pp. 200–204. Springer, Heidelberg (2002)
57. Lehmann, G., Blumendorf, M., Trollmann, F., Albayrak, S.: Meta-modeling Runtime Models. In: Dingel, J., Solberg, A. (eds.) MODELS 2010. LNCS, vol. 6627, pp. 209–223. Springer, Heidelberg (2011)
58. de Lemos, R., et al.: Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Self-Adaptive Systems. LNCS, vol. 7475, pp. 1–32. Springer, Heidelberg (2013)
59. Mao, M., Li, J., Humphrey, M.: Cloud auto-scaling with deadline and budget constraints. In: 2010 11th IEEE/ACM International Conference on Grid Computing (GRID), pp. 41–48. IEEE (2010)

60. McKinley, P.K., Cheng, B.H.C., Ramirez, A.J., Jensen, A.C.: Applying evolutionary computation to mitigate uncertainty in dynamically-adaptive, high-assurance middleware. Journal of Internet Services and Applications, 1–8 (2011)
61. Medvidovic, N., Taylor, R.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering 26(1), 70–93 (2000)
62. Mens, K., Kellens, A., Pluquet, F., Wuyts, R.: Co-evolving Code and Design Using Intensional Views - A Case Study. Journal of Computer Languages, Systems and Structures 32(2), 140–156 (2006)
63. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science 152, 125–142 (2006)
64. Morin, B., Fleurey, F., Bencomo, N., Jézéquel, J.-M., Solberg, A., Dehlen, V., Blair, G.S.: 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 2008. LNCS, vol. 5301, pp. 782–796. Springer, Heidelberg (2008)
65. Mosterman, P.J., Sztipanovits, J., Engell, S.: Computer-automated multiparadigm modeling in control systems technology. IEEE Trans. Contr. Sys. Techn. – special issue on Computer Automated Multiparadigm Modeling 12(2), 223–234 (2004)
66. Mosterman, P.J., Vangheluwe, H.: Computer automated multi-paradigm modeling in control system design. In: Proceedings of the IEEE International Symposium on Computer-Aided Control Systems Design (CACSD 2000), pp. 65–70 (2000)
67. Mosterman, P.J., Vangheluwe, H.: Guest editorial: Special issue on computer automated multi-paradigm modeling. ACM Trans. Model. Comput. Simul. 12(4), 249–255 (2002)
68. Nentwich, C., Emmerich, W., Finkelstein, A.: Consistency management with repair actions. In: Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, USA, May 3-10, pp. 455–464. IEEE Computer Society (2003)
69. Object Management Group (OMG): Software & Systems Process Engineering Metamodel specification (SPEM) Version 2.0 (2008), OMG Adopted Specification formal/2008-04-01
70. Object Management Group (OMG): MOF 2.0 QVT Final Adopted Specification v1.1 (2011), OMG Adopted Specification formal/2011-01-01
71. Object Management Group (OMG): OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1, OMG Adopted Specification formal/2011-08-07 (2011)
72. Object Management Group (OMG): OMG Object Constraint Language (OCL) Version 2.3.1, OMG Adopted Specification formal/2012-01-01 (2012)
73. OMG: Model Driven Architecture (MDA). Technical Report ORMSC/2001-07-01, OMG (July 2001)
74. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: ICSE 1998: Proceedings of the 20th International Conference on Software Engineering, pp. 177–186. IEEE Computer Society, Washington, DC (1998)
75. Oreizy, P., Medvidovic, N., Taylor, R.N.: Runtime software adaptation: framework, approaches, and styles. In: ICSE Companion 2008: Companion of the 30th International Conference on Software Engineering, pp. 899–910. ACM, New York (2008)

76. Ramirez, A.J., Cheng, B.: Evolving Models at Run Time to Address Functional and Non-Functional Adaptation Requirements. In: Bencomo, N., Blair, G., France, R., Jeanneret, C., Munoz, F. (eds.) Proceedings of the 4th International Workshop on Models@run.time. CEUR Workshop Proceedings, vol. 509, pp. 31–40 (2009)
77. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004)
78. Rozenberg, G.: Handbook of Graph Grammars and Computing by Grah Transformation, vol. 1. Foundations. World Scientific (1997)
79. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4(2), 1–42 (2009)
80. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
81. Schürr, A.: Programmed graph replacement systems. In: Handbook of graph grammars and computing by graph transformation: Foundations, vol. 1, pp. 479–546. World Scientific Publishing Co., Inc., River Edge (1997)
82. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Softw. 20(5), 42–45 (2003)
83. Song, H., Huang, G., Chauvel, F., Xiong, Y., Hu, Z., Sun, Y., Mei, H.: Supporting runtime software architecture: A bidirectional-transformation-based approach. Journal of Systems and Software 84(5), 711–723 (2011)
84. Song, H., Huang, G., Xiong, Y., Chauvel, F., Sun, Y., Mei, H.: Inferring Metamodels for Runtime System Data from the Clients of Management APIs. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part II. LNCS, vol. 6395, pp. 168–182. Springer, Heidelberg (2010)
85. Song, H., Xiong, Y., Chauvel, F., Huang, G., Hu, Z., Mei, H.: Generating Synchronization Engines between Running Systems and Their Model-Based Views. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 140–154. Springer, Heidelberg (2010)
86. Stevens, P.: A landscape of bidirectional model transformations. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) Generative and Transformational Techniques in Software Engineering II. LNCS, vol. 5235, pp. 408–424. Springer, Heidelberg (2008)
87. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. Software and Systems Modeling 8 (2009)
88. Taentzer, G.: AGG: A graph transformation environment for modeling and validation of software. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 446–453. Springer, Heidelberg (2004)
89. Taïani, F., Grace, P., Coulson, G., Blair, G.S.: Past and future of reflective middleware: towards a corpus-based impact analysis. In: ARM, pp. 41–46 (2008)
90. Tajalli, H., Garcia, J., Edwards, G., Medvidovic, N.: PLASMA: a plan-based layered architecture for software model-driven adaptation. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE 2010), pp. 467–476. ACM, New York (2010)
91. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. Wiley Publishing (2009)
92. Tratt, L.: Model transformations and tool integration. SOSYM 4(2), 112–122 (2005)
93. Tratt, L.: A change propagating model transformation language. Journal of Object Technology 7(3), 107–124 (2008)

94. Vogel, T., Giese, H.: Adaptation and Abstract Runtime Models. In: Proceedings of the 5th ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2010), pp. 39–48. ACM (2010)
95. Vogel, T., Giese, H.: A Language for Feedback Loops in Self-Adaptive Systems: Executable Runtime Megamodels. In: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2012), pp. 129–138. IEEE Computer Society (2012)
96. Vogel, T., Giese, H.: Requirements and Assessment of Languages and Frameworks for Adaptation Models. In: Kienzle, J. (ed.) MODELS 2011 Workshops. LNCS, vol. 7167, pp. 167–182. Springer, Heidelberg (2012)
97. Vogel, T., Giese, H.: Model-driven engineering of self-adaptive software with eurema. ACM Trans. Auton. Adapt. Syst. 8(4), 1–18 (2014)
98. Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B.: Incremental Model Synchronization for Efficient Run-Time Monitoring. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 124–139. Springer, Heidelberg (2010)
99. Vogel, T., Seibel, A., Giese, H.: The Role of Models and Megamodels at Runtime. In: Dingel, J., Solberg, A. (eds.) MODELS 2010. LNCS, vol. 6627, pp. 224–238. Springer, Heidelberg (2011)
100. Zarras, A.: Applying model-driven architecture to achieve distribution transparencies. Information & Software Technology (2006)
101. Zave, P., Jackson, M.: Four dark corners of requirements engineering. ACM Trans. Softw. Eng. Methodol. 6(1), 1–30 (1997)