mRUBiS: An Exemplar for Model-Based Architectural Self-Healing and Self-Optimization

Thomas Vogel
Humboldt-Universität zu Berlin, Germany
thomas.vogel@cs.hu-berlin.de

ABSTRACT

Self-adaptive software systems are often structured into an adaptation engine that manages an adaptable software by operating on a runtime model that represents the architecture of the software (model-based architectural self-adaptation). Despite the popularity of such approaches, existing exemplars provide application programming interfaces but no runtime model to develop adaptation engines. Consequently, there does not exist any exemplar that supports developing, evaluating, and comparing model-based self-adaptation off the shelf. Therefore, we present mRUBiS, an extensible exemplar for model-based architectural self-healing and self-optimization. mRUBiS simulates the adaptable software and therefore provides and maintains an architectural runtime model of the software, which can be directly used by adaptation engines to realize and perform self-adaptation. Particularly, mRUBiS supports injecting issues into the model, which should be handled by self-adaptation, and validating the model to assess the self-adaptation. Finally, mRUBiS allows developers to explore variants of adaptation engines (e.g., event-driven self-adaptation) and to evaluate the effectiveness, efficiency, and scalability of the engines.

CCS CONCEPTS

• **Software and its engineering → Development frameworks and environments**; **Software development techniques**;

KEYWORDS

Self-adaptation, architecture, runtime models, simulator

ACM Reference Format:
Thomas Vogel. 2018. mRUBiS: An Exemplar for Model-Based Architectural Self-Healing and Self-Optimization. In *SEAMS '18: 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3194133.3194161

1 INTRODUCTION

Self-adaptive software is a closed-loop system that implements a feedback loop to autonomously adapt to changes at runtime [11]. Engineering such systems is challenging as it involves domain and adaptation concerns. Thus, a popular engineering approach is to split the system in two parts: an *adaptation engine* that implements the feedback loop and an *adaptable software* that realizes the domain logic while the engine manages the software [30, 39]. This split promotes separation of concerns but requires that the engine has a representation of the software based on which it can perform adaptation. Such a representation is a *runtime model* that is causally connected to the adaptable software, that is, changes of the software are synchronized to the model and vice versa [6]. According to the survey by Salehie and Tahvildari [30], most self-adaptive systems in research follow such a *model-based self-adaptation* approach. In this context, many researchers argue that the *software architecture* is the appropriate abstraction level for self-adaptation and thus for the runtime model [1, 7, 8, 10, 12–15, 22, 24, 26–28, 34, 36, 38]. Consequently, our focus is on *model-based architectural self-adaptation*.

To advance software engineering research, benchmarks and exemplars have been identified as useful [32]. Researchers have proposed such artifacts to develop and evaluate self-adaptation solutions, which can be categorized in two groups. (1) Artifacts that support developing and evaluating self-adaptive software and that range from requirements for the adaptive internet of things [5], component frameworks for smart cyber-physical systems [21, 23], a platform for cloud applications [3], a tool for runtime monitoring and verification [2], and a benchmark environment for self-adaptive applications in Hadoop clusters [41]. (2) Several exemplars that provide a real or simulated adaptable software, on top of which adaptation engines should be developed, and that enable evaluation and comparison of adaptation engines [9, 16, 19, 25, 31, 37, 40].

Despite the popularity of model-based architectural self-adaptation, none of these artifacts particularly addresses this kind of self-adaptation by providing an architectural runtime model of the adaptable software. Artifacts of the first group typically do not provide a runtime model. Exceptions are Hogna [3] and Lotus [2] that use a performance model for performance analysis respectively a labeled transition system model for verification. However, these models cannot be used for generic, architectural self-adaptation. The exemplars of the second group provide application programming interfaces (APIs) to the adaptable software instead of a runtime model. Adaptation engines developed for these exemplars use these APIs to manage the software. Consequently, using these exemplars for model-based architectural self-adaptation requires from developers to implement a runtime model and a causal connection between the model and the APIs. This is challenging since developers have to assure the synchronization and fidelity of the runtime model with the running software [4, 6]. Therefore, we conclude that there does not exist any exemplar that supports developing, evaluating, and comparing model-based architectural self-adaptation off the shelf.

In this paper, we present *mRUBiS*[1], an extensible exemplar for model-based architectural self-adaptation. It simulates the mRUBiS

[1]The exemplar is available at https://github.com/thomas-vogel/mRUBiS.

SEAMS '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5715-9/18/05.
https://doi.org/10.1145/3194133.3194161

system as the adaptable software and provides as well as maintains an *architectural runtime model* of the system. This model serves as the interface for adaptation engines to realize and perform architectural adaptation of mRUBiS. Thus, model-based architectural self-adaptation is supported off the shelf. Developers are relieved from implementing a runtime model and a causal connection to the adaptable software as well as setting up a corresponding runtime infrastructure. Instead, they can focus on designing, implementing, and evaluating the adaptation logic on top of the provided runtime model. With respect to developing self-adaptive software, such an approach supports the early testing of feedback loops [18].

The simulation performed by the exemplar consists of a predefined number of iterations over the following three steps: (1) According to a *scenario*, the simulator injects *issues* into the runtime model and thus to the mRUBiS architecture, which should be handled by self-adaptation. (2) The adaptation engine developed by the user of the exemplar is triggered to analyze and adapt the mRUBiS architecture described in the model. The adaptation aims at resolving the injected issues and thus at satisfying the goals of mRUBiS. (3) According to a set of *validators*, the simulator validates the adaptation and runtime model to check whether issues are remaining in the mRUBiS architecture. It further evaluates the self-adaptation by computing the utility of the current architecture based on a *utility function* and by measuring the execution time of the self-adaptation (*i.e.*, of the 2nd step). This data is summarized at the simulation end.

For the mRUBiS architecture, we provide scenarios, issues, validators, and utility functions for a self-healing and a self-optimization case study. Furthermore, each of these elements can be replaced or extended by developers to address other case studies. Even mRUBiS as the adaptable software described by the runtime model can be replaced or extended. The *CompArch* language to express the runtime model is generic and supports modeling arbitrary component-based architectures and properties. This guarantees the extensibility of the exemplar although prescribing the language for the model.

However, the exemplar does not restrict the adaptation engines developed on top of it. In contrast, it even encourages developers to explore variants of engines, for instance, by using the provided change events to drive the self-adaptation. Developers can use their favorite technologies to implement the engines such as code (Java) or model-based rules (*e.g.*, expressed with OCL and Story Diagrams) that operate on the runtime model. Finally, the exemplar allows developers to evaluate the effectiveness (in terms of the utility of the adaptable software), efficiency (in terms of execution time), and scalability of self-adaptation by scaling the size of the architectural model and the number of injected issues per simulation round.

The paper is structured as follows: Sec. 2 presents the background. We discuss the generic, mRUBiS-specific, and implementation aspects of the exemplar in Sec. 3, 4, and 5 before concluding in Sec. 6.

2 BACKGROUND: MODELS AT RUNTIME

In this section, we conceptually outline the use of a runtime model by a self-adaptive system (*cf.* Figure 1). Although the mRUBiS exemplar does not impose a certain structure on the adaptation engine, we use the MAPE structure [20] for the illustration.

As shown in Figure 1, the self-adaptive system is split into an adaptation engine and adaptable software. The engine realizes the

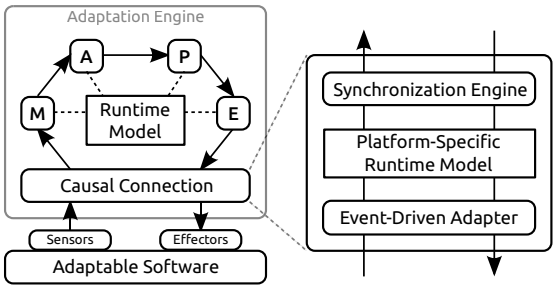


Figure 1: Self-adaptive system with a runtime model.

four MAPE steps that operate on the runtime model to perform adaptation. Such an approach requires a *causal connection* that propagates changes of the adaptable software to the runtime model and changes of the model prescribing an adaptation to the software (*cf.* [4, 6]). Thus, the monitoring and execution steps use the causal connection to synchronize the model and the adaptable software.

However, implementing a causal connection is challenging as developers have to address the abstraction gap between the software and the architectural runtime model, assure the fidelity of the model with the running software, and achieve a runtime-efficient synchronization [4, 6]. These aspects make the implementation of a causal connection a costly task considering, for instance, the complexity of the Rainbow solution that uses an architectural model of the Znn exemplar for self-adaptation [9, 14]. With a similar complexity, we realized the causal connection in earlier work [34, 36] (*cf.* Figure 1). Using the sensor and effector APIs, an event-driven adapter incrementally maintains a platform-specific runtime model that is at the same abstraction level as the APIs. A synchronization engine then propagates changes between the platform-specific and architectural runtime model taking the abstraction gap into account.

Based on these observations, we may conclude that developing self-adaptive systems with runtime models is challenging as it causes accidental complexity of implementing a causal connection. In contrast, developers should rather focus on the essence, which is developing the adaptation logic and particularly the analysis and planning mechanisms. Therefore, with the mRUBiS exemplar we propose simulating the adaptable software and causal connection so that developers experimenting with adaptation engines are relieved from implementing a runtime model and causal connection.

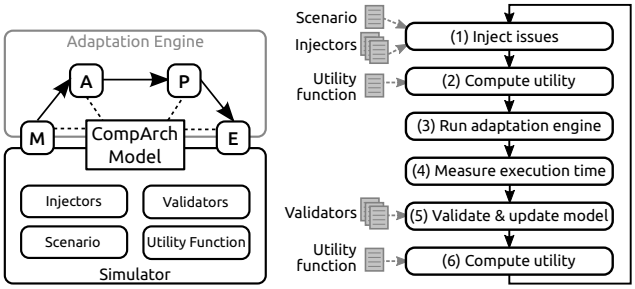
3 GENERIC SIMULATION FRAMEWORK

We now present the generic simulation framework that underlies the exemplar and that is independent of any adaptable software.

3.1 Overview

The structure of the generic simulation framework is shown in Figure 2a. As previously motivated, the simulator emulates the adaptable software and causal connection by providing an architectural runtime model, the *CompArch Model* (*cf.* Figures 1 and 2a). This model is expressed in the generic *CompArch* language (*cf.* Section 3.2). Thus, the simulator takes over the part of the monitoring and execution steps of a feedback loop that would otherwise realize the causal connection. The adaptation engine developed by users of the exemplar only relies on the CompArch model without having to

use any sensor or effector APIs. Thus, the MAPE steps directly use and operate on the CompArch model to perform self-adaptation.



(a) Structure. (b) Simulation steps.

Figure 2: Generic simulation framework.

To enable the simulation, the simulator must be configured with *injectors*, a *scenario*, *validators*, and a *utility function*. For each of them, we defined interfaces that developers have to implement. The simulator orchestrates these artifacts provided by developers to perform the simulation as shown in Figure 2b.

A simulation with the exemplar consists of several rounds while each round corresponds to executing one iteration of the six steps shown in Figure 2b. The number of rounds for a simulation is defined by the *scenario*. The six simulation steps refine the three steps discussed in the introduction and are as follows:

(1) Inject issues. The simulator injects issues into the CompArch model and thus, to the architecture of the adaptable software. The *scenario* defines which issues are injected to which elements of the architecture in each simulation round. For each issue type, an *injector* is required that provides the behavior of actually injecting an issue to the element of the architecture defined by the scenario.

(2) Compute utility. Using the provided *utility function*, the simulator computes the current utility of the adaptable software based on the architecture described by the CompArch model. Computing the utility after injecting the issues measures the drop in the utility caused by these issues. We use the developing of the utility as one criterion to evaluate adaptation engines.

(3) Run adaptation engine. The adaptation engine developed by the user of the exemplar is executed to perform self-adaptation that aims at resolving the injected issues. For this purpose, the engine analyzes the architecture described by the CompArch model to identify issues, plans an adaptation of the architecture to resolve these issues, and finally executes the planned adaptation by adjusting the architecture described by the CompArch model.

(4) Measure execution time. The simulator measures the execution time of the adaptation engine when performing self-adaptation (see previous step). We use the runtime efficiency of a self-adaptation as one criterion to evaluate adaptation engines.

(5) Validate and update model. The simulator validates the model to check whether issues are remaining in the architecture. As the simulator emulates the adaptable software, it may additionally update the model as a reaction to a valid or invalid self-adaptation (*e.g.*, an adapted architecture might result in changing performance characteristics that should be reflected in the model). Such checks and updates are realized by *validators* that developers provide. Moreover, the simulator itself provides generic validators that check architectural constraints (*e.g.*, are all required interfaces of components connected to other components?). While the results of such checks give feedback to the developer about the effectiveness of the self-adaptation solution she implemented, the updates of the model enable a simulation over several rounds of self-adaptation.

(6) Compute utility. Similarly to step (2), the utility of the adaptable software is computed, but this time after the self-adaptation (3rd step) and when its effects are reflected in the model (5th step). Thus, a potential increase of the utility achieved by self-adaptation is measured and used for evaluating the effectiveness of self-adaptation.

All steps except of (3) are performed by the simulator. In each simulation round, the simulator further logs information about the simulation such as the injected issues, computed utility, measured execution time, and the results of the validators. At the very end of the simulation, the simulator stores the raw data of the developing of the utility and execution time values to files and creates corresponding charts. Developers can use the raw data to evaluate in-depth the results and the charts to visually evaluate the effectiveness and efficiency of their self-adaptation solutions.

3.2 The CompArch Modeling Language

To express the architectural runtime model of an adaptable software, we developed a generic yet simple modeling language called *CompArch* (short for component architecture). It supports modeling an individual or a multi-tenant system that comprises many individual systems (e.g., a marketplace with individual shops). In this section, we present the CompArch metamodel, notation, and editor.

3.2.1 Metamodel. The CompArch metamodel depicted in Figure 3 defines the concepts of the language. Its elements are structured into five groups (see coloring of the elements):

(1) General elements (gray elements). The Architecture is the root node of the model and represents the modeled system. The ArchitecturalElement is the super class defining an identifier (uid) and criticality (*i.e.*, how critical the element is for the operation of the system) for all type-, deployment-, and runtime-level elements.

(2) Type level (blue elements). This level covers ComponentTypes with their required and provided InterfaceTypes. An InterfaceType has a fully qualified name (fqName) and contains MethodSpecifications, each defined by a signature. A ParameterType is a configuration option for a ComponentType specified by its name, primitive data type, and defaultValue. A ComponentType has a reliability and is instantiated to a Component by the instantiate() operation.

(3) Deployment level (green elements). This level defines the architecture of a deployed system that consists of Components with their Required- and ProvidedInterfaces. Each component is in a certain life cycle state as defined by the enumeration ComponentState: UNDEPLOYED, DEPLOYED (but stopped), STARTED, and UNKNOWN (*e.g.*, crashed). It is configured by concrete values for its Parameters according to the ParameterTypes and by Connectors that wire their required interfaces to provided interfaces of other components. Finally, components are associated to Tenants. Each Tenant groups the components that comprise the tenant's sub-system in a multi-tenant system. If the modeled system is only an individual system, we consider it as a multi-tenant system with one tenant.

(4) Runtime level (red elements). This level covers runtime concepts that are monitored in the running system. An Exception

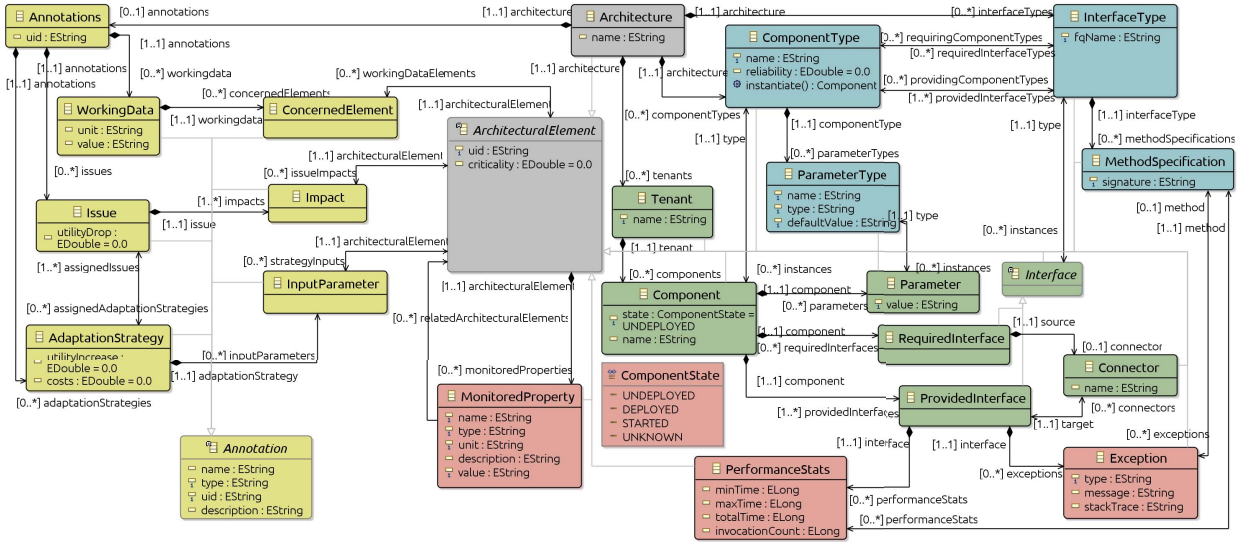


Figure 3: Complete metamodel of the CompArch modeling language.

thrown when using a ProvidedInterface is attached to this interface. It is characterized by a type, message, stackTrace, and the method that raised the exception. PerformanceStats capture the performance of using a component through a ProvidedInterface. For each method of the interface, it captures the min., max., and total execution time in *ms* and the number of invocations. Thus, the average execution time is calculated by totalTime/invocationCount. A MonitoredProperty describes an application-specific property monitored for an ArchitecturalElement. It has a name, description, data type, measurement unit, and the monitored value. Finally, the life cycle of Components is monitored and reflected by the state attribute.

(5) Annotations (yellow elements). Annotations to the architecture of the adaptable software are optionally used by adaptation engines to capture knowledge created during self-adaptation (*e.g.*, analysis results and adaptation plans). Each Annotation is characterized by a name, type, uid, and description. We consider three categories of annotations: i) Metric WorkingData with a value and measurement unit that can refer via ConcernedElements to ArchitecturalElements (*e.g.*, such data may count how often a Component violates a performance goal). ii) An Issue identified in the architecture and causing a utilityDrop of the adaptable software that has Impacts on the affected ArchitecturalElements (*e.g.*, an issue is the violation of a performance goal by a Component). iii) An AdaptationStrategy with its anticipated utilityIncrease and execution costs that has been planned to resolve an assigned Issue and that may have ArchitecturalElements as InputParameters (*e.g.*, an adaptation strategy is to replace the Component that violates the performance goal).

3.2.2 Notation and Modeling Editor. To create and visualize CompArch models, we developed a notation and a corresponding graphical modeling editor. The notation is based on the proposal by Kramer and Magee [22] and comprises four types of diagrams.

An *Overview Diagram* lists and supports creating and deleting ComponentTypes, InterfaceTypes, and Tenants that are part of the Architecture. An *Interface Type Diagram* supports specifying an InterfaceType with its MethodSpecifications. A *Component Type Diagram* supports specifying a ComponentType with its ParameterTypes and associating InterfaceTypes as required and provided. A *Tenant Architecture Diagram* supports specifying an architecture for each Tenant as shown in Figure 4, which comprises the Components with their Required- and ProvidedInterfaces, Parameters, and Connectors. This diagram furthermore visualizes the runtime-level elements and annotations (*cf.* red and yellow elements of the metamodel in Figure 3) as they refer to architectural elements.

Thus, the editor can be used to create the initial CompArch model at design time and to visualize the model at runtime when the model is enriched with runtime information. The simulator provides snapshots of the CompArch model on-demand during the simulation.

3.3 Interacting with CompArch Models

Based on CompArch (*cf.* Section 3.2), this section outlines how the simulator and the adaptation engine interact with a CompArch model during the simulation and self-adaptation (*cf.* Figure 2a).

As the simulator emulates the adaptable software, it maintains and updates the model with respect to the deployment- and runtime-level elements (*cf.* green and red elements in Figure 3). Thus, it may update the software architecture (*i.e.*, the structure of Components and Connectors, and Parameter values) and the runtime information (*i.e.*, MonitoredProperties, PerformanceStats, and Exceptions). Such updates are performed by injectors when issues are injected in the architecture (*cf.* first step in Figure 2b). For instance, to emulate a component crash, the Component is removed from the architecture, or to emulate a higher load on the software, the PerformanceStats values are increased. Moreover, such updates are performed by validators in response to a self-adaptation (*cf.* fifth step in Figure 2b). For instance, a self-adaptation reconfigures the architecture of the adaptable software, which changes the performance characteristics and therefore, the simulator updates the PerformanceStats values.

Following the relationships defined by the metamodel, an adaptation engine can query and navigate and thus analyze the whole CompArch model. However, there are restrictions for adapting the

model. Focusing on architectural self-adaptation, an adaptation engine can change the composition of Components and Connectors by adding, removing, and replacing them (*cf.* structural adaptation) as well as the operational modes of Components by changing values of configuration Parameters (*cf.* parameter adaptation). Structural adaptations include changing the life cycle states of Components, for instance, to start or stop a component. Thus, the changeable elements of the model refer to the architecture (*cf.* green elements in Figure 3) while keeping the type level (*cf.* blue elements) unchanged. Moreover, the runtime-level elements (*cf.* red elements) are not changeable as they represent only observable information. Finally, the annotations (*cf.* yellow elements) are exclusively used by adaptation engines and can therefore be changed at any will.

3.4 CompArch Change Events

Besides the CompArch model, the simulator provides on-demand events that notify about changes of the model. We defined eleven types of such events notifying, for instance, about changes of the life cycle state of a Component, the addition or removal of a Component from the architecture, the re-routing of a Connector, the occurrence of an Exception, or updates of a Parameter, PerformanceStats, and MonitoredProperty. Each event has a time stamp of the change and points to the specific architectural element affected by the change.

Thus, adaptation engines can process such events to drive the self-adaptation process (*cf.* event-condition-action rules for adaptation). Such an event-driven process avoids querying the whole architecture and CompArch model to identify changes since the events directly point to the changed elements in the architecture. This typically results in a runtime-efficient self-adaptation process.

4 mRUBIS

In this section, we instantiate the generic simulation framework for the self-healing and self-optimization of mRUBiS by providing injectors, scenarios, validators, and utility functions (*cf.* Figure 2).

4.1 System Description and Goals

mRUBiS is a marketplace on which users sell or auction items. It is derived from *RUBiS*, a popular case study to evaluate control theoretic adaptation (*cf.* [29]). Since RUBiS is monolithic and just runs one shop, we added new functionalities (*e.g.*, the pipe of filters), modularized the shop to 18 components, and extended it to a marketplace that hosts arbitrarily many shops. Each of these shops consists of the 18 components and belongs to a tenant (see Figure 4). All shops share the same component types but each shop has its own, individually configured components. Thus, the architecture of each shop is isolated from the architectures of the other shops (*cf.* multi-tenancy). While the modularization enables architectural adaptations that are otherwise not possible with the monolithic RUBiS, the multi-tenancy setting enables scaling up the system and thus the architectural runtime model expressed in CompArch.

As shown in Figure 4, each shop has components to manage items (Item Management Service), users (User Management Service), auctions/purchases (Bid and Buy Service), inventory (Inventory Service), and user ratings (Reputation Service), to authenticate users (Authentication Service), and to persist and retrieve data (Persistence and Query Services) from the database. The pipe of filter

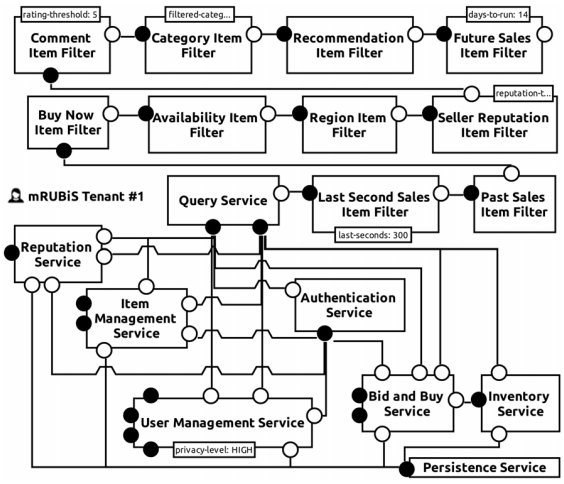


Figure 4: CompArch model of an individual shop (tenant).

components improves the quality of the results when users search for items. The pipe follows the batch sequential pipe-and-filter architectural style. It iteratively filters the list of items obtained from the Query Service by removing items that are not relevant for the specific user and search request. Hence, the pipe improves the quality while also increasing the response time of the search.

The company running mRUBiS aims for high sales volumes by achieving customer satisfaction and encouraging customers to additional purchases. Therefore, the system should be highly available and its response times should be low. To reach these goals, self-adaptation should be employed to automatically repair failures (*i.e.*, self-healing) and improve performance (*i.e.*, self-optimization).

4.2 Self-Healing mRUBiS

To achieve a high availability, the self-healing aims at repairing architectural failures that disrupt the operation of mRUBiS.

Injectors. For the mRUBiS architecture, we provide injectors for five classes of critical failures (CFs). They target Components that either crash and enter the UNKNOWN life cycle state (CF1), throw Exceptions exceeding a given threshold (CF2), are destroyed and removed from the architecture (CF3), and are continuously affected by CF1 or CF2 requiring novel repair mechanisms (CF4), and Connectors that are lost and removed from the architecture (CF5).

For these CFs, we consider adaptation strategies (AS) that restart (AS1) or redeploy (AS2) the affected Component by controlling the component's life cycle state (*e.g.*, adjusting the state from STARTED to DEPLOYED stops the component), that replaces the affected Component with a new instance of the same (AS3) or different (AS4) ComponentType, and that reestablishes a lost Connector (AS5).

Scenario. We provide a basic scenario that injects in each simulation round one CF to a random Component or Connector.

Validators. We provide generic and mRUBiS-specific validators that check for the validity of the architecture described by the CompArch model and that take the CFs into account.

Utility Function. We define the utility of mRUBiS as the sum of the utilities of all Components and the utility of a single component c as c.type.reliability × c.criticality × c.connectivity (*cf.* [17]). The connectivity refers to the number of Connectors associated to c. If a

component is affected by a CF, it does not provide any utility so that the utility of mRUBiS is decreased by the utility of this component.

Challenges. Using this case, developers can investigate different self-adaptation solutions for identifying and resolving CFs. We provide three example solutions that are either monolithic, decomposed into a MAPE structure, or use the change events (*cf.* Section 3.4). Moreover, we used mRUBiS to scale the size of the architecture and number of injected CFs to investigate the scalability of planning mechanisms [17]. In this context, the exemplar provides a generator for mRUBiS architectures with user-defined numbers of shops to obtain various sizes of the CompArch model for scalability analyses.

4.3 Self-Optimizing mRUBiS

To achieve low response times, the self-optimization aims at improving the performance of mRUBiS by architectural reconfiguration.

Issues. We define three performance issues (PIs) with corresponding adaptation strategies (AS) for mRUBiS that target the pipe of filters (*cf.* Section 4.1). The filters of a pipe are ordered based on their performance, that is, filters that perform well are located toward the front of the pipe where more items must be processed than toward the end of the pipe. In this context, the performance of a filter might change so that the pipe is not optimally ordered (PI1) which requires a re-ordering of filters (AS6). The average response time of the search in mRUBiS is above a threshold (PI2) so that filters should be skipped from the pipe (AS7) to reduce the response time at the cost of a decreased quality of the search results. Similarly, the average response time can be below a threshold (PI3) so that skipped filters can be added back to the pipe (AS8).

Scenario. We provide a basic scenario that injects in each simulation round one PI to the pipe of a random shop of mRUBiS.

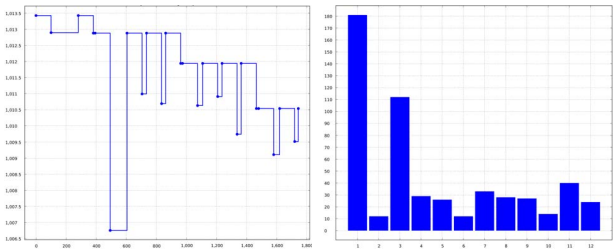
Validators. We provide generic and mRUBiS-specific validators that check for the validity of the architecture, particularly of the pipe of filters, and that take the PIs into account.

Utility Function. The initial utility of mRUBiS is computed similarly to the self-healing case. Additionally, it is reduced by 50% of the utility of a filter that is not located at its optimal position in the pipe, and in total by 20% if the performance goal is not met (*i.e.*, the average response time of the search is above the threshold).

Challenges. In addition to the challenges for the self-healing case, using this case, developers can investigate self-adaptation solutions that targets a specific architectural style (pipe-and-filter). We provide an example solution that is driven by change events and takes the architectural style into account. Finally, a challenge is to incorporate the self-healing and self-optimization and thus, multiple adaptation concerns into one self-adaptation solution.

4.4 Evaluating Self-Adaptation Solutions

Using the exemplar, self-adaptation solutions are evaluated by the simulation results. The simulator measures the utility of mRUBiS over time (as calculated by the utility function) and the execution time of a solution in each simulation round. After the simulation, the simulator generates charts of the results as shown in Figure 5 and stores the raw data to csv files. Developers can use the data and charts to evaluate the effectiveness and efficiency of their solutions. For instance, Figure 5a shows a step chart of the utility over time of mRUBiS with 50 shops, in which a drop is caused by an injected



(a) Utility of mRUBiS. (b) Execution time of a solution.

Figure 5: Generated charts of the simulation results.

issue and an increase by a self-adaptation that successfully resolved issues. Since the utility is decreasing in steps, the self-adaptation was not always successful. Figure 5b shows the execution time of a self-adaptation solution (feedback loop) for each simulation round.

Finally, by increasing the number of shops and injected issues per round, the scalability of self-adaptation solutions can be evaluated for large architectures. For this purpose, the exemplar provides a generator for mRUBiS architectures with arbitrarily many shops.

5 IMPLEMENTATION

mRUBiS in terms of the simulator, CompArch language, and modeling editor is implemented as an Eclipse plug-in on top of the Eclipse Modeling Framework (EMF) [33]. Thus, the exemplar can be used within Eclipse although it can be run decoupled from Eclipse. Using the simulator boils down to using a Java interface to interact with it. EMF provides the infrastructure for defining the CompArch language and handling CompArch models. Thus, users of the exemplar can use Java or any EMF-compatible technology such as OCL and Story Diagrams (*cf.* [17, 18]) to process a CompArch model.

mRUBiS can be easily extended due to its modular implementation architecture. It has sub-projects for the generic simulation framework (Section 3.1), the CompArch language (Section 3.2.1), the editor (Section 3.2.2) as well as for the mRUBiS-specific extensions to the simulator and for the example solutions (Section 4).

6 CONCLUSION AND FUTURE WORK

In this paper we presented mRUBiS, an extensible exemplar that supports model-based architectural self-adaptation off the shelf. It simulates the adaptable software and provides an architectural runtime model of the software. This model is the interface for adaptation engines to realize and perform architectural self-adaptation. Thus, developers are relieved from implementing a runtime model so that they can focus on developing and evaluating the model-based adaptation logic. Moreover, we presented the self-healing and self-optimization case studies for using the exemplar.

So far, mRUBiS was useful as it supports early testing [18] and evaluations [17, 35] of solutions. Moreover, we successfully used it for student projects in a course on self-adaptive software, in which the simulator and the students' solutions play against each other using simulation scenarios that were not known to the students.

As future work, we plan to improve the efficiency by incrementally computing the utility, to support environment models, and to provide further case studies and metrics for evaluating solutions.

REFERENCES

[1] Konstantinos Angelopoulos, Vítor E. Silva Souza, and João Pimentel. 2013. Requirements and Architectural Approaches to Adaptive Software Systems: A Comparative Study. In *Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 23–32. https://doi.org/10.1109/SEAMS.2013.6595489

[2] Davi Monteiro Barbosa, Rômulo Gadelha de Moura Lima, Paulo Henrique Mendes Maia, and Evilásio Costa Junior. 2017. Lotus@Runtime: A Tool for Runtime Monitoring and Verification of Self-adaptive Systems. In *Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 24–30. https://doi.org/10.1109/SEAMS.2017.18

[3] Cornel Barna, Hamoun Ghanbari, Marin Litoiu, and Mark Shtern. 2015. Hogna: A Platform for Self-adaptive Applications in Cloud Environments. In *Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 83–87. https://doi.org/10.1109/SEAMS.2015.26

[4] Amel Bennaceur, Robert B. France, Giordano Tamburrelli, Thomas Vogel, Pieter J. Mosterman, Walter Cazzola, et al. 2014. Mechanisms for Leveraging Models at Runtime in Self-adaptive Software. In *Models@run.time*. LNCS, Vol. 8378. Springer, 19–46. https://doi.org/10.1007/978-3-319-08915-7_2

[5] Amel Bennaceur, Ciaran McCormick, Jesús García Galán, Charith Perera, Andrew Smith, Andrea Zisman, and Bashar Nuseibeh. 2016. Feed Me, Feed Me: An Exemplar for Engineering Adaptive Software. In *Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 89–95. https://doi.org/10.1145/2897053.2897071

[6] Gordon Blair, Nelly Bencomo, and Robert B. France. 2009. Models@run.time. *IEEE Computer* 42, 10 (2009), 22–27. https://doi.org/10.1109/MC.2009.326

[7] Victor Braberman, Nicolas D'Ippolito, Jeff Kramer, Daniel Sykes, and Sebastian Uchitel. 2017. An Extended Description of MORPH: A Reference Architecture for Configuration and Behaviour Self-Adaptation. In *Software Engineering for Self-Adaptive Systems III. Assurances*. LNCS, Vol. 9640. Springer, 377–408. https://doi.org/10.1007/978-3-319-74183-3_13

[8] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. 2004. A Survey of Self-management in Dynamic Software Architecture Specifications. In *Workshop on Self-managed Systems (WOSS)*. ACM, 28–33. https://doi.org/10.1145/1075405.1075411

[9] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. 2009. Evaluating the effectiveness of the Rainbow self-adaptive system. In *Intl. Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 132–141. https://doi.org/10.1109/SEAMS.2009.5069082

[10] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. 2002. Towards Architecture-based Self-healing Systems. In *Workshop on Self-healing Systems (WOSS)*. ACM, 21–26. https://doi.org/10.1145/582128.582133

[11] Rogério de Lemos, Holger Giese, Hausi Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, Danny Weyns, et al. 2013. Software Engineering for Self-Adaptive Systems: A second Research Roadmap. In *Software Engineering for Self-Adaptive Systems II*. LNCS, Vol. 7475. Springer, 1–32. https://doi.org/10.1007/978-3-642-35813-5_1

[12] George Edwards, Joshua Garcia, Hossein Tajalli, Daniel Popescu, Nenad Medvidovic, Sukhat Gaurav, and Brad Petrus. 2009. Architecture-driven self-adaptation and self-management in robotics systems. In *Intl. Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 142–151. https://doi.org/10.1109/SEAMS.2009.5069083

[13] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. 2006. Using Architecture Models for Runtime Adaptability. *IEEE Software* 23, 2 (2006), 62–70. https://doi.org/10.1109/MS.2006.61

[14] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. 2004. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Comp.* 37, 10 (2004), 46–54. https://doi.org/10.1109/MC.2004.175

[15] David Garlan and Bradley Schmerl. 2002. Model-based Adaptation for Self-healing Systems. In *Workshop on Self-healing Systems (WOSS)*. ACM, 27–32. https://doi.org/10.1145/582128.582134

[16] Simos Gerasimou, Radu Calinescu, Stepan Shevtsov, and Danny Weyns. 2017. UNDERSEA: An Exemplar for Engineering Self-adaptive Unmanned Underwater Vehicles. In *Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 83–89. https://doi.org/10.1109/SEAMS.2017.19

[17] Sona Ghahremani, Holger Giese, and Thomas Vogel. 2017. Efficient Utility-Driven Self-Healing Employing Adaptation Rules for Large Dynamic Architectures. In *Intl. Conference on Autonomic Computing (ICAC)*. IEEE, 59–68. https://doi.org/10.1109/ICAC.2017.35

[18] Joachim Hänsel, Thomas Vogel, and Holger Giese. 2015. A Testing Scheme for Self-Adaptive Software Systems with Architectural Runtime Models. In *Intl. Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*. IEEE, 134–139. https://doi.org/10.1109/SASOW.2015.27

[19] M. Usman Iftikhar, Gowri Sankar Ramachandran, Pablo Bollansée, Danny Weyns, and Danny Hughes. 2017. DeltaIoT: A Self-adaptive Internet of Things Exemplar. In *Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 76–82. https://doi.org/10.1109/SEAMS.2017.21

[20] Jeffrey O. Kephart and David Chess. 2003. The Vision of Autonomic Computing. *IEEE Computer* 36, 1 (2003), 41–50. https://doi.org/10.1109/MC.2003.1160055

[21] Michal Kit, Ilias Gerostathopoulos, Tomas Bures, Petr Hnetynka, and Frantisek Plasil. 2015. An Architecture Framework for Experimentations with Self-adaptive Cyber-physical Systems. In *Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 93–96. https://doi.org/10.1109/SEAMS.2015.28

[22] Jeff Kramer and Jeff Magee. 2007. Self-Managed Systems: An Architectural Challenge. In *Future of Software Engineering (FOSE)*. IEEE, 259–268. https://doi.org/10.1109/FOSE.2007.19

[23] Filip Krijt, Zbynek Jiracek, Tomas Bures, Petr Hnetynka, and Ilias Gerostathopoulos. 2017. Intelligent Ensembles: A Declarative Group Description Language and Java Framework. In *Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 116–122. https://doi.org/10.1109/SEAMS.2017.17

[24] Jeff Magee and Jeff Kramer. 1996. Dynamic Structure in Software Architectures. In *Symposium on Foundations of Software Engineering (SIGSOFT)*. ACM, 3–14. https://doi.org/10.1145/239098.239104

[25] Martina Maggio, Alessandro V. Papadopoulos, Antonio Filieri, and Henry Hoffmann. 2017. Self-adaptive Video Encoder: Comparison of Multiple Adaptation Strategies Made Simple. In *Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 123–128. https://doi.org/10.1109/SEAMS.2017.16

[26] Sara Mahdavi-Hezavehi, Vinicius H.S. Durelli, Danny Weyns, and Paris Avgeriou. 2017. A systematic literature review on methods that handle multiple quality attributes in architecture-based self-adaptive systems. *Information and Software Technology* 90 (2017), 1–26. https://doi.org/10.1016/j.infsof.2017.03.013

[27] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. 2009. Models@ Run.time to Support Dynamic Adaptation. *IEEE Computer* 42, 10 (2009), 44–51. https://doi.org/10.1109/MC.2009.327

[28] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. 1998. Architecture-based Runtime Software Evolution. In *Intl. Conference on Software Engineering (ICSE)*. IEEE, 177–186. https://doi.org/10.1109/ICSE.1998.671114

[29] Tharindu Patikirikorala, Alan Colman, Jun Han, and Liuping Wang. 2012. A Systematic Survey on the Design of Self-adaptive Software Systems Using Control Engineering Approaches. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 33–42. https://doi.org/10.1109/SEAMS.2012.6224389

[30] Mazeiar Salehie and Ladan Tahvildari. 2009. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.* 4, 2 (2009), 1–42. https://doi.org/10.1145/1516533.1516538

[31] Sanny Schmid, Ilias Gerostathopoulos, Christian Prehofer, and Tomas Bures. 2017. Self-adaptation Based on Big Data Analytics: A Model Problem and Tool. In *Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 102–108. https://doi.org/10.1109/SEAMS.2017.20

[32] Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. 2003. Using Benchmarking to Advance Research: A Challenge to Software Engineering. In *International Conference on Software Engineering (ICSE)*. IEEE, 74–83. https://doi.org/10.1109/ICSE.2003.1201189

[33] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework* (2nd ed.). Addison-Wesley.

[34] Thomas Vogel and Holger Giese. 2010. Adaptation and Abstract Runtime Models. In *Intl. Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 39–48. https://doi.org/10.1145/1808984.1808989

[35] Thomas Vogel and Holger Giese. 2014. Model-Driven Engineering of Self-Adaptive Software with EUREMA. *ACM Trans. Auton. Adapt. Syst.* 8, 4 (2014), 18:1–18:33. https://doi.org/10.1145/2555612

[36] Thomas Vogel, Stefan Neumann, Stephan Hildebrandt, Holger Giese, and Basil Becker. 2009. Model-Driven Architectural Monitoring and Adaptation for Autonomic Systems. In *Intl. Conference on Autonomic Computing and Communications (ICAC)*. ACM, 67–68. https://doi.org/10.1145/1555228.1555249

[37] Danny Weyns and Radu Calinescu. 2015. Tele Assistance: A Self-adaptive Service-based System Examplar. In *Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 88–92. https://doi.org/10.1109/SEAMS.2015.27

[38] Danny Weyns, M. Usman Iftikhar, Sam Malek, and Jesper Andersson. 2012. Claims and supporting evidence for self-adaptive systems: A literature study. In *Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 89–98. https://doi.org/10.1109/SEAMS.2012.6224395

[39] Danny Weyns, Sam Malek, and Jesper Andersson. 2012. FORMS: Unifying Reference Model for Formal Specification of Distributed Self-adaptive Systems. *ACM Trans. Auton. Adapt. Syst.* 7, 1 (2012), 8:1–8:61. https://doi.org/10.1145/2168260.2168268

[40] Jochen Wuttke, Yuriy Brun, Alessandra Gorla, and Jonathan Ramaswamy. 2012. Traffic Routing for Evaluating Self-adaptation. In *Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 27–32. https://doi.org/10.1109/SEAMS.2012.6224388

[41] Bo Zhang, Filip Křikava, Romain Rouvoy, and Lionel Seinturier. 2017. Hadoop-benchmark: Rapid Prototyping and Evaluation of Self-adaptive Behaviors in Hadoop Clusters. In *Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 175–181. https://doi.org/10.1109/SEAMS.2017.15