Efficient Utility-Driven Self-Healing Employing Adaptation Rules for Large Dynamic Architectures

Sona Ghahremani, Holger Giese and Thomas Vogel
Hasso Plattner Institute, University of Potsdam
Email: {sona.ghahremani|holger.giese|thomas.vogel}@hpi.uni-potsdam.de

***Abstract*—Self-adaptation can be realized in various ways. Rule-based approaches prescribe the adaptation to be executed if the system or environment satisfy certain conditions and result in scalable solutions, however, with often only satisfying adaptation decisions. In contrast, utility-driven approaches determine optimal adaptation decisions by using an often costly optimization step, which typically does not scale well for larger problems. We propose a rule-based and utility-driven approach that achieves the beneficial properties of each of these directions such that the adaptation decisions are optimal while the computation remains scalable since an expensive optimization step can be avoided. The approach can be used for the architecture-based self-healing of large software systems. We define the utility for large dynamic architectures of such systems based on patterns capturing issues the self-healing must address and we use pattern-based adaptation rules to resolve the issues. Defining the utility as well as the adaptation rules pattern-based allows us to compute the impact of each rule application on the overall utility and to realize an incremental and efficient utility-driven self-healing. We demonstrate the efficiency and optimality of our scheme in comparative experiments with a static rule-based scheme as a baseline and a utility-driven approach using a constraint solver.**

Keywords: self-healing, adaptation rules, architecture-based adaptation, utility.

I. Introduction

There are various ways how self-adaptation following the MAPE-K feedback loop [17] and in particular the analyzing and planning phases of the feedback loop can be realized.

On the one hand, *rule-based* approaches [8,18] combine the analyzing and planning phases. Adaptation is executed for specific events and under specific conditions by adaptation rules. In such approaches, events trigger the rules that subsequently check their additional conditions. If the conditions are fulfilled, the actions of the rule can be applied and result in the envisioned changes. The main strengths of rule-based approaches are i) the readability and elegance of individual rules, and ii) the efficiency with which the rules can be processed. The limited expressiveness of the adaptation rules is a drawback. At runtime, the applicable rules are identified (matched) and executed to adapt the system configuration [9].

On the other hand, *utility-driven* approaches [6,19] often determine optimal adaptation decisions by using optimization techniques in the planning phase that are guided by a utility function. A utility function determines how valuable each possible system configuration is and the optimization techniques then aim for finding the optimal one. However, the optimization techniques usually prevent that the approaches scale well for large configuration spaces at runtime. To achieve runtime efficiency, linear utility functions are often used since optimizing complex utility functions, as in constraint solver-based approaches, results in non-scalable solutions [9].

Therefore, we propose in this paper a combined rule-based and utility-driven approach that guarantees optimal adaptation decisions and that is scalable. Thus, the combined approach achieves the individual benefits of both approaches while it avoids the corresponding drawbacks with respect to the optimality of adaptation decisions and scalability. Particularly, we target the architecture-based self-healing of large software systems and exploit some restrictions usually present for this class of problems to achieve the guarantees for optimality. We use our former work to define the utility function in a pattern-based way for large dynamic architectures [14] and further also define the adaptation rules in a pattern-based way. This joint use of patterns allows us to combine the utility and the rules and therefore to predict the impact of each rule application on the overall utility. Based on these predictions for the rules and the knowledge about the costs of applying each adaptation rule, we can determine and execute at runtime the optimal sequence of rule applications.

We demonstrate these benefits of our approach by comparing it with two alternatives solutions in simulations of mRUBiS [30]. We show that our approach is only slightly slower but reaches a higher utility over time (reward) than a static, rule-based solution. Then, we demonstrate that our approach always makes optimal adaptation decisions similar to an alternative solution using a constraint solver. However, our approach requires considerably less time than the solver, especially for large architectures. Being incremental makes our approach more scalable since it faces less overhead. As argued by Ghezzi [15], incremental solutions are highly desirable for self-adaptive software. In our earlier work [31]–[33], we presented an incremental scheme for the monitoring and execution phases of the feedback loop operating on architectural runtime models. The results of this paper complement the earlier results by enabling the incremental analysis and planning for architectural runtime models based on adaptation rules and utility functions. Therefore, we focus in this paper on the analysis and planning phases of the feedback loop.

The rest of the paper is structured as follows: We introduce architectural self-adaptation with runtime models and the pattern-based definition of the utility in Section II. Then, we discuss our approach considering its general scheme and its

2474-0756/17 $31.00 © 2017 IEEE
DOI 10.1109/ICAC.2017.35

application in a feedback loop in Sections III and IV. We analyze, discuss, and demonstrate the benefits of our approach in Sections V and VI. Finally, we discuss related work in Section VII and provide a conclusion with an outlook on future work in Section VIII.

II. Prerequisites

A. Architectural Self-Adaptation and Runtime Models

To realize self-adaptation, a software system is equipped with a *MAPE-K* feedback loop that monitors and analyzes the system and if needed, plans and executes an adaptation to the system, which is all based on knowledge [17]. In this context, many approaches consider the *software architecture* as an appropriate abstraction level (e.g., [13,23,32]) since self-adaptation can be *generally* achieved by adding and removing components as well as connectors among components [22]. For this purpose, the feedback loop maintains a *runtime model* [2], as part of its knowledge, which represents the architecture of the system under adaptation. The model and the system are *causally connected*, that is, any relevant change of the system is reflected in the model, and vice versa [2]. Thus, the analysis and planning phases can operate on the model. Technically, a runtime model allows us to employ model-driven engineering (MDE) techniques [11]. In our earlier work [31]–[33], we presented an incremental scheme for the monitor and execute phases that employs MDE techniques and architectural runtime models and that is the basis for this paper.

As the running example, we use *mRUBiS* [30], a modular variant of RUBiS. mRUBiS is an online marketplace that hosts an arbitrary number of shops. Each shop consists of 18 components, can be configured differently, and runs isolated from the other shops. We are particularly interested into self-healing to automatically repair runtime failures by architectural self-adaptation. This allows us to consider *general* repair rules that adapt the architectural configuration of mRUBiS. Therefore, we equip mRUBiS with a MAPE-K feedback loop that uses an architectural runtime model of mRUBiS. Specifically, the model represents the runtime architecture of mRUBiS according to the deployment of mRUBiS in an EJB application server. This model conforms to the metamodel shown in Fig. 1.

The metamodel captures the mRUBiS Architecture with a set of ComponentTypes that require and provide InterfaceTypes.

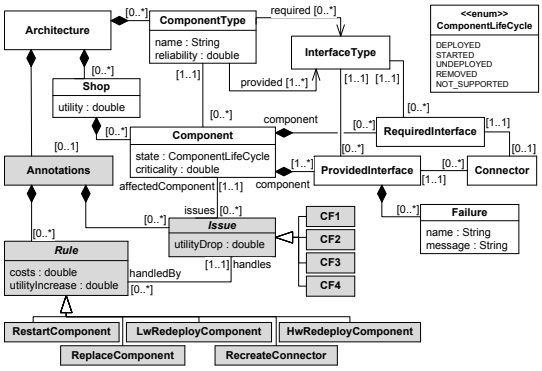

Fig. 1: Simplified Metamodel of the Runtime Model.

For each Shop, the same component types are instantiated to Components with their Provided- and RequiredInterfaces. A Connector links a required and a provided interface if both are of the same InterfaceType. These elements allow us to describe the runtime architecture of mRUBiS. The other elements are relevant for self-adaptation and outlined later.

Using (meta)models and MDE techniques, we realize the analysis rules with model queries and the adaptation rules with in-place model transformations. For self-healing, the analysis rules query the runtime model to identify failures (issues) in mRUBiS while the adaptation rules determine how to modify the runtime model, that is, how to change the architecture to repair these failures. To specify a model query, we use a pattern P of a set of patterns $\mathcal{P}$ describing a structural fragment of the architecture G. Since the architecture is represented by the runtime model, we also use G to refer to the model. An occurrence of a pattern P in the model G corresponds to a match m of P in G (we write $G \models_m P$). For instance, a match identifies a failure in the architecture. An adaptation rule r in the rule set $\mathfrak{R}$ uses such patterns or already identified matches in the model to localize where adaptation is needed and then to change the model in-place to repair the failure.

B. Pattern-Based Architectural Utility

A *utility* function U is an objective policy that expresses how well each configuration of the system in its domain satisfies the functional and non-functional *goals* of the system. For this purpose, U assigns a real-value scalar desirability belonging to $[-\infty, +\infty]$ to any possible system configuration G. Such scalar values allow us to compare different configurations and to select the one with the highest utility as the best adaptation decision. Furthermore, the accumulated utility over time described by the *reward* supports comparisons over time.

Defining a valid utility function is of high importance since in an optimization problem to find the best configuration, it is always the utility function that is maximized not the real utility of the system. There has been extensive research on *utility-driven* decision-making policies and elicitation of user preferences (e.g., [25]). Considering architectural configurations, a typical approach is to predict the impact of each variant of an architectural property on the overall goals. A normalized linear utility function uses these impact values to compute their weighted sum over all properties given a concrete architecture with concrete variants for each property. The weights represent the preferences of the user/developer and the result is the utility of the given architecture [10]. Such an approach can be used for planning a self-adaptation, that is, to identify the target architecture, to which the system should be adapted to.

Moreover, defining utility functions for architectural configurations is challenging, particularly when considering large, dynamic architectures [5]. In the following, we outline our proposal to define utility functions for large, dynamic architectures based on patterns [14]. Due to the employed patterns, our utility functions can cope with dynamic architectural changes in contrast to statically defined utility functions.

We know that for a utility function for architectural runtime models must hold that (i) the optimal architectural configuration where all the system goals are optimally fulfilled must gain the maximum utility and that (ii) if any constraint or goal is violated, this must lead to a decrease of utility.

According to (i), we include the impact of *present architectural fragments* in the utility. We define such fragments by *positive architectural utility patterns* $\mathcal{P}^+ = \{P_1^+, \ldots, P_k^+\}$ and capture their impact on the utility by utility sub-functions U_i. This impact may vary for each individual occurrence of such a fragment depending on the specific context such as the criticality of the concrete components that are present.

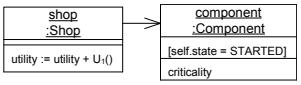

Fig. 2: Positive Architectural Utility Pattern P_1^+.

Fig. 2 shows the positive pattern P_1^+ and the related utility sub-function U_1. The pattern prescribes a started component that is associated to a shop and therefore contributes to the shop's functionality. Matching this pattern for one component in the runtime model, the utility of the associated shop is increased by U_1. We define U_1 := criticality of the component × reliability of the corresponding component type × connectivity of the component. Matching all components of a shop, the utility of the shop is the sum of the corresponding sub-utilities U_1 for all of these components. Finally, the pattern is applied to all shops of mRUBiS to obtain the utilities of each shop.

Concerning the parts of U_1, each component has a criticality (see corresponding attribute) denoting its importance for a shop. For instance, the Authentication component is more critical than the Reputation component since the former is necessarily required by a shop to close a deal while the latter is not. Additionally, each component type has a reliability. For a certain functionality, alternative component types with different reliabilities exist (e.g., local vs. various third-party authentication services). Thus, selecting the most reliable alternative for a functionality results in a higher utility increase. Finally, the connectivity of a present component in terms of the number of associated Connectors indicates the importance and thus influences the utility increase of the component.

According to (ii), we further include the negative impact of undesirable situations defined by *negative architectural utility patterns* $\mathcal{P}^- = \{P_{k+1}^-, \ldots, P_n^-\}$ in the utility. Such patterns negatively affect the architecture such that they decrease the overall utility according to their utility sub-functions U_i. Examples of such negative patterns are occurrences of runtime failures. As before, the impact may vary for each individual occurrence of a negative pattern depending on the specific context such as the criticality of the specific components that cause the failures.

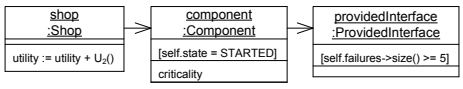

Fig. 3: Negative Architectural Utility Pattern P_2^-.

Fig. 3 shows the negative architectural utility pattern P_2^- for mRUBiS, which prescribes the case when more than four failures (exceptions) occurred in a started component. Each occurrence of such a negative pattern decreases the utility of the associated shop by U_2. Note that U_2 is typically negative.

Consequently, the positive patterns capture the possible utility gained by the current architectural configuration while the negative patterns capture whether this potential is currently realized. If it is not realized, negative patterns occur in the architecture and correspondingly decrease the utility. In the most extreme case, the whole utility gained by occurrences of positive patterns can be subtracted again when evaluating the negative patterns. In mRUBiS, adding a new shop always leads to an increase in the utility and is considered a positive pattern while occurrences of failures in components of a shop correspond to a negative pattern and thus reduce the utility.

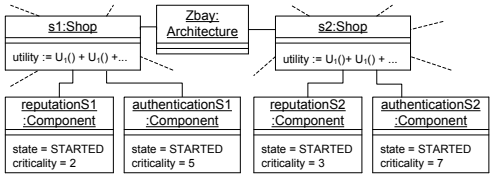

Fig. 4: Excerpt of the Architectural Runtime Model.

Fig. 4 shows an excerpt of the runtime model with two matches of the positive pattern P_1^+ (see Fig. 2) for each shop, that is, with two started components in each shop. For instance, the elements s1 and reputationS1 denote one match and s1 and authenticationS1 the other match of P_1^+ in shop s1. Each match increases the utility of the shop by U_1 taking the characteristics of the specific component into account (e.g., the different criticality values of reputationS1 and authenticationS1). The utility of a shop is the sum of the utility sub-functions U_1 for all components of the shop while the utility of mRUBiS is the sum of the utilities of all shops. Similarly, matches for negative patterns would decrease the utility of the shops and therefore of mRUBiS (not illustrated in Fig. 4).

Consequently, considering $M_i(G) = \{m | G \models_m P_i\}$ as the set of matches for the pattern P_i in the current architectural configuration G, the overall utility function $U(G)$ accumulates all effects due to matches of all patterns $\mathcal{P} = \{P_1, \ldots, P_n\}$[1]:

$$U(G) := \sum_{i=1}^{n} \sum_{m \in M_i(G)} U_i(G, m) \quad (1)$$

Therefore, adaptation rules can refer to such patterns. On the one hand, they should identify and repair occurrences of negative patterns in the architecture. On the other hand, they should not affect existing but rather enable new occurrences of positive patterns by repairing occurrences of negative patterns.

The definition of the pattern-based utility takes the context into account. Each pattern P_i exactly specifies a context that influences the utility sub-function U_i and thus the increase or decrease of the overall utility. For instance, the pattern

[1]If we do not have to distinguish between positive and negative patterns, we omit the superscript + and − for the patterns $P \in \mathcal{P}$.

P_1^+ in Fig. 2 specifies the criticality of the component and the associated shop as the context. Similarly, the shop and the component including criticality are the context of the pattern P_2^- in Fig. 3. Moreover, we could extend the context of both patterns, for instance, by taking the component type into account. When matching a pattern, the concrete context is dynamically identified for each match in the runtime model. Such a concrete context corresponds to a fraction of the runtime model that is navigated to obtain the required information such as the criticality of component to calculate U_i at runtime.

III. Utility-Driven Rule-Based Adaptation Scheme

We propose a utility-driven scheme to evaluate dynamic software architectures. As discussed in Section II-B, utility functions are used to map each architectural configuration of a software system to a scalar value indicating how well the configuration satisfies the goals. The need for evaluating *dynamic* architectures is motivated by architectural self-adaptation. If adaptation is required, the feedback loop has to identify a suitable or even the optimal target configuration and select accordingly the adaptation rules that move the system to this configuration. Thus, a feedback loop can use the evaluation scheme to determine the target configuration. With the proposed scheme, we are particularly interested in self-healing, that is, the automatic repairing of runtime failures by *general* rules that perform architectural adaptation.

In this context, we express *issues* (i.e., runtime failures) for an architecture as model *patterns* such that concrete issues with different impacts on the overall utility $U(G)$ relate to occurrences of these patterns in the runtime model G. Additionally, we can express an adaptation rule $r = (P, \dots)$, that is applied on the runtime model if the condition described as a model pattern P is satisfied. We denote for an adaptation rule $r = (P, \dots)$ that an occurrence as a match m for P in the runtime model G_i exists and that applying the rule results in a modified runtime model G_j by $G_i \rightarrow_{r,m} G_j$.

Our scheme can be directly mapped to a MAPE-K feedback loop. The *monitoring* phase observes change events emitted by the system to trigger the adaptation. During the *analyzing* and *planning*, our scheme requires two decisions: the target configuration of the system, and the rules and their matches that move the system to the target. Finally, the last step *executes* these rules for their matches on the running system.

These two decisions are inspired by the idea of model-predictive control [27] that first defines a target and then predicts the optimal path to reach the target. This is illustrated in Fig. 5 showing one target with three alternative paths to reach the target. Considering the self-healing, selecting an architecture where issues are repaired is equivalent to defining the target configuration. During the repair, selecting the best sequence of adaptation rules and their matches that resolve all issues is equivalent to building the path toward target.

For the target configuration G_i must hold that the utility $U(G_i)$ must be higher or equal to the utility $U(G_j)$ of all possible next configurations G_j that are the outcomes of resolving the issues in the faulty configuration G_0. To

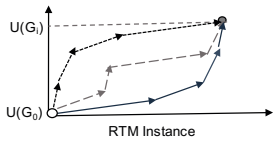

Fig. 5: Target Configuration and Different Adaptation Paths.

avoid enumerating the complete search space, our scheme computes the impact of each possible rule application for a match on the related utility sub-function and therefore on the overall utility $(U(G_i) - U(G_0))$. After defining the target G_i, the second step is to select a set of adaptation rules and their matches to actually reach G_i. Each rule application changes the runtime model G. Starting from a runtime model instance G_0 at the beginning of each MAPE-K cycle, G_0 evolves to G_i as a result of a sequence of rule applications: $G_0 \rightarrow_{r_1,m_1} G_1 \rightarrow \dots \rightarrow_{r_i,m_i} G_i$.

Based on the impact of each rule application on the related utility sub-function and thus on the overall utility, we then determine the path. To resolve an issue, multiple rules are applicable and an estimation of their impacts allows us to select a conflict-free subset of them. We assume here that for each set, we can compute $U(G_i)$ regardless of the order in which the rules are executed (see assumption (A4) later).

Our scheme guarantees (i) executing the selected set of rules and related matches eventually leads to the target configuration G_i with utility $U(G_i)$ and (ii) executing them in the right order results in the highest achievable reward (utility accumulated over time). To fulfil (i), when there are two or more alternative rules and related matches to resolve the same issue, the scheme proceeds with selecting the rule that has the highest impact on the corresponding utility sub-function. To achieve (ii), within a selected conflict-free set of rules, rules are executed in a decreasing order regarding their impact on the corresponding utility sub-functions. We claim and show that the proposed approach is optimal regarding the final utility $U(G_i)$ and the achieved reward in the meantime.

IV. Linking Utility to Adaptation Rules

The utility functions for architectural runtime models as we defined them in Section II-B in principle allow us to follow an optimization-based approach that searches the configuration space and computes the utility for each possible configuration. However, such a solution is rather wasteful if the utility would have to be computed for each configuration completely anew.

In contrast, the proposed utility-driven, rule-based scheme determines the impact of each rule application on the utility at runtime. It derives in a greedy manner from these impacts an optimal sequence of rule applications concerning the adaptation decisions. This scheme is realized by a MAPE-K feedback loop as shown in Fig. 6 and discussed in the following.

A. Monitor

During monitoring, change events emitted by the system are observed and reflected in the runtime model, that is, the model is updated to represent the current system configuration [32,

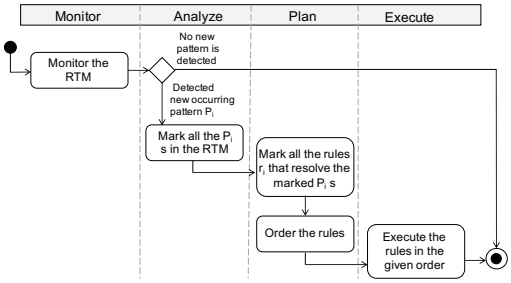

Fig. 6: Steps for the Different Phases of the MAPE-K Loop.

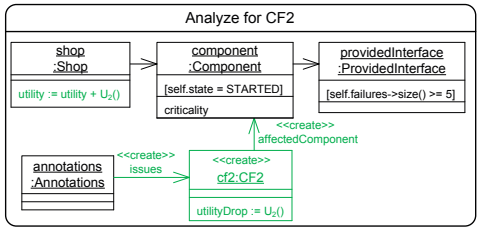

Fig. 7: Annotating a Negative Architectural Utility Pattern.

33]. In our example, we observe the life cycle state of a component (e.g., to monitor whether a component has stopped, crashed, or been removed) and Failures such as exceptions that occur when using a ProvidedInterface (cf. Fig. 1).

B. Analyze

In the analysis phase, the observed changes are analyzed to detect negative patterns (issues) in the model. This updates the known set of matches for issues. New matches are determined through applying an event-property-change mechanism and all old matches have to be checked whether they are still valid.

As a first step, we compute the utility incrementally rather than for each configuration anew. Given a former runtime model G and an updated version G', the set of new occurrences for utility patterns are $M_i^{new} = M_i(G') \backslash M_i(G)$. Similarly, $M_i^{del} = M_i(G) \backslash M_i(G')$ captures the matches for patterns that are no longer valid. We can therefore define the changes of a utility function $U(G)$ accordingly by a utility change function $U_\Delta(G', G)$ as $U(G') - U(G)$ as:

$$-\sum_{i=1}^{n} \sum_{m \in M_i^{del}} U_i(G, m) + \sum_{i=1}^{n} \sum_{m \in M_i^{new}} U_i(G', m) \qquad (2)$$

Besides computing the decrease or increase in utility, we keep track of the identified issues that need to be resolved. Considering our focus on self-healing, all the architectural utility patterns that need to be matched and resolved are the negative patterns $\mathcal{P}^- = \{P_{k+1}^-, \ldots, P_n^-\}$.

For this purpose, the analyze phase adds Annotations to the runtime model. It checks the model for occurrences of negative patterns, which are then annotated as Issues pointing to the affectedComponent (Fig. 1). As issues we consider crashes (CF1) and unplanned removals (CF3) of components, occurrences of Failures (CF2), and connector crashes (CF4).

Fig. 7 shows an analysis rule realized by a *story pattern* [7] that detects the negative pattern shown in Fig. 3. The occurrence of the negative pattern results in a drop in the utility of the shop by U_2. The story pattern creates the CF2 annotation including the utilityDrop pointing to the affected component to be used later on in the feedback loop. Here, we omit the details to avoid multiple annotations for the same match (issue).

C. Plan

Based on the annotations representing new or remaining issues in the form of matches m for negative patterns, our approach incrementally proceeds during the planning phase by 1) computing the set of all possible rule applications, 2) selecting for each issue the best rule application based on computations of the impact on utility and costs, and 3) finally ordering the best rule application for all issues to minimize the lost reward.

1) Compute All Possible Adaptation Rule Matches: The rule-based adaptation is based only on rule applications that lead to an improved utility. For this case, we will show that rules must always be linked to the negative patterns and that knowing the matches for these patterns will allow us to compute all relevant adaptation rule matches incrementally.

For any adaptation scheme that is based on the outlined utility function defined by means of patterns where considering our focus on self-healing, all the patterns that need to be matched and resolved are the negative patterns, the following observations must hold: (1) If there are no occurrences of the negative patterns, then there is no need for adaptation and no improvement of the utility is possible. (2) Any possible improvement of the utility must necessarily resolve found occurrences of the negative patterns as otherwise no improvement of the utility would be possible.

Consequently, we can safely assume that (A1) for any in-place model transformation rule $r_j = (P_j, \ldots)$ in the adaptation rule set $\mathfrak{R}$ must hold that a negative pattern P_i^- exists such that any match m_j for P_j includes a match m_i for P_i^-. Otherwise, r_j could be enabled even though no utility improvement can be achieved which would contradict observation (1). It can be the case that the initial match condition of the rule r_j might require more context and be more restricted compared to the match condition for the negative pattern, but in the presented example in Fig. 3 and 7, both conditions are exactly the same.

Furthermore, we can plausibly assume that (A2) for the rule $r_j = (P_j, \ldots)$ in the rule set $\mathfrak{R}$ and any match m_j for P_j and the included match m_i for the related negative pattern P_i^- holds that applying r_j for m_j will make the match m_i for P_i^- invalid. Otherwise, r_j would not handle the found occurrence of the negative patterns P_i^- and thus would not lead to the improvement of the utility we would expect according to observation (2). To keep our considerations simple, we consider the case where each rule covers exactly one negative pattern. Based on these assumptions, we can compute all matches for rules incrementally if for the related negative pattern P_i^- the set of new matches M_i^{new} is given.

Fig. 8 illustrates a simplified view of a planning rule for the example to repair CF2. Following the analysis phase described in Fig. 7, the rule set $\mathfrak{R}$ is checked to find all the rules that can

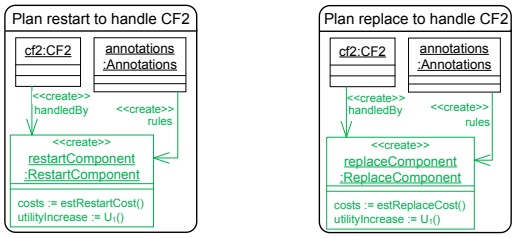

Fig. 8: Rules for Planning an Adaptation.

extend the occurred negative pattern with adaptation rules such as RestartComponent and ReplaceComponent that can handle the corresponding issue such as CF2. The cost estimation functions estRestartCost() and estReplaceCost() compute the costs of executing each rule to the system under adaptation, for instance, to restart or replace a component in the system.

In general and following the MAPE-K cycle, performing an adaptation consists of a planning and an execution part. The planning part decides which adaptation rule among all possible ones should be applied, while the execution part actually applies the selected rule to prescribe an adaptation in the runtime model that is synchronized to the running system under adaptation (cf. causal connection in Section II-A).

For our example, the planning phase addresses the identified issues by selecting for each of them the adaptation rule to be executed such that it enriches the model with Rules that handle the identified Issues. These rules are finally enacted by the execute phase. As adaptation rules, we consider restarting, redeploying, and replacing components as well as recreating connectors (see Fig. 1). For the redeployment, there exists two variants. The light-weight variant keeps the latest configuration of the component to be redeployed, while the heavy-weight variant adapts the configuration of the redeployed component.

In general, a specific issue such as CF2 can be handled by multiple rules such as restarting or redeploying the affected component. Thus, the planning must decide which rule should be applied. Likewise, if multiple issues occur at the same time, the planning must decide which issue should be handled first.

2) Select Adaptation Rule Match for Each Negative Pattern Match Based on the Impact on Utility and Estimates for Costs: To determine the best adaptation rule for each found occurrence of any negative pattern, our approach determines the impact of adaptation rules on the utility for each match.

For a single rule $r_j = (P_j, \dots)$ where P_j extends the negative pattern P_i^-, it holds that each time r_j is applied to m_j then the match m_i for a negative pattern P_i^- is removed (see A2 discussed previously). We further assume that (A3) r_j does not result in any new match or removed matches besides m_i for any negative pattern. Then, we conclude for any G, G' that results from applying rule r_j to G for match m_j ($G \rightarrow_{r_j,m_j} G'$):

$$U_\Delta^{r_j}(G, m_j) := U_\Delta(G', G) = U_i^-(G, m_i) \tag{3}$$

Thus, for the discussed form of rules for which (A1) to (A3) hold, we can locally compute their impact on the utility.

If further the assumption (A4) holds that r_j does not affect any utility sub-function for any match m_k for another negative pattern P_k^-, then applying a rule r_j for a match m_j does not affect the impact on the utility for any other rule r_k and match m_k. Thus, if (A1) to (A4) hold, we can independently and locally compute the utility impact of each match of a rule.

There can be cases that the side effect of applying a rule r_j (i.e., $G \rightarrow_{r_j,m_j} G'$) results in new matches for one or more positive patterns. In such cases, the impact on the utility by the corresponding positive utility sub-function of these matches is added to $U_\Delta^{r_j}(G, m_j)$ in equation (3). For this purpose, it must hold that all the potential positive patterns are completely within the scope of the application condition and side effect of r_j and do not match only partially (A5). Otherwise, matches for the positive patterns cannot be enabled by applying r_j. Thus, the impact can be simply considered in $U_\Delta^{r_j}(G, m_j)$ since the resulting formula for the corresponding increase of the utility can be determined at development-time. An example for such a case is replacing a local authentication component with an alternative third-party service. Each of the possible alternative services results in a different positive pattern with different utility sub-functions regarding their offered reliability.

Similarly, the execution time for adaptation rules can be estimated by defining a cost function $Cost^{r_j}(G, m_j)$ for each rule application which may depend on the match and its context in G. In our example, we have cost estimation functions for each type of rule such as estRestartCost() (see Fig. 8).

The planning phase then proceeds by first ranking the applicable rules for each pattern match found according to their utilityIncrease $U_\Delta^{r_j}(G, m_j)$ and, if this is the same, according to the costs $Cost^{r_j}(G, m_j)$. The utilityIncrease of each rule is the computed increase of the overall utility as the result of applying the rule. This attribute value is computed based on the runtime computation of criticality, connectivity, and reliability of the affectedComponent. The costs for each rule is the computed execution time computed at runtime.

3) Order the Execution for All Selected Adaptation Rule Matches: The final planning step is determining the order in which the issues should be resolved. For this purpose, all rules annotated in the model are sorted regarding their impact on the overall utility divided by the costs. This guarantees in the execution phase that the maximal utility is reestablished as fast as possible and that the lost reward is minimized.

D. Execute

Finally, we combine the beforehand outlined steps such that the utility-driven, rule-based adaptation is achieved. Thus, each detected negative pattern is handled with the most appropriate rule and the rule applications are ordered such that those with the highest impact on utility are executed first. The execute phase takes over the ordered list of adaptation rule matches from the planning phase and executes them in the given order.

Fig. 9 illustrates an adaptation rule that restarts a component to address CF2. The control flow within the rule complies with an UML activity diagram (cf. [7]). Based on the analysis and planning phases (see Fig. 7 and 8), an adaptation rule, in this case restartComponent, has been selected to handle CF2 affecting the specific component (see first node in Fig. 9). This

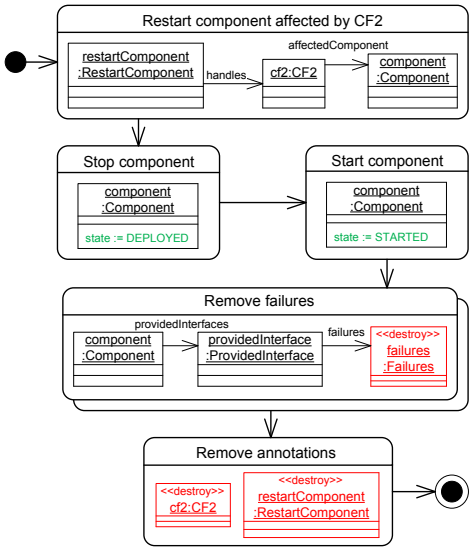

Fig. 9: Rule for Executing a Component Restart.

component is then restarted (see second and third node). After that, the runtime model is cleaned up by removing (destroying) the observed exceptions (failures) and the annotations for the executed rule (restartComponent) and the handled issue (CF2).

V. Analysis and Discussion of the Approach

We now analyze and discuss the computational effort and the optimality of the resulting utility and reward of our approach.

a) Incremental Computational Effort: If the patterns to be matched as well as the rules do only contain links for associations with fixed small upper bounds smaller than a constant, we can conclude the following: the outline adaptation scheme requires only incremental computational effort for finding new matches for each analysis rule to annotate negative architectural utility pattern, to extend such matches for an adaptation rule, or to check for old issues or rules whether they are still matched. As we have only a small finite number of rules for analysis and planning, the effort to compute all new issues and determine all related new matches for rules extending these issues for Δ changes of the architectural runtime model are both in $O(\Delta)$. The effort to check all old issues and related new matches for rules extending these issues for Δ' old unprocessed changes of the architectural runtime model are both in $O(\Delta')$. Thus, the analysis and planning activities only require an incremental computational effort as long as the set of accumulated unprocessed changes do not grow unbounded.

b) Optimality of the Adaptation: We know that as a result of executing a selected adaptation rule, a maximal increase of the utility is guaranteed. Due to assumption (A2), the rules will remove the match m and, due to assumptions (A3) and (A4), no other matches for issues are affected such that the overall increase after applying all selected rules and matches must be maximal. Furthermore, the ordering of the rules and determined matches leads to the maximal reward. Consequently, for our approach holds that the resulting utility after the complete adaptation is maximal and so is the reward for the chosen sequence of executing the adaptation rules.

c) Limitations: Rules that are not triggered by any issue or that do not resolve any issue and thus have no impact on the utility do not make any sense so that the assumptions (A1) and (A2) are justified in general. Similarly, rules that cause new issues are not helpful and therefore could also be excluded (see first part of (A3)). However, it will not always be the case that rules do not impact other issues than the ones they should handle (see second part of (A3) and (A4)), for instance, when due to resource limitations the planned sequence of rule applications can not be fully executed such that not all of the issues can be repaired. It can also be the case that rules do not completely cover the positive patterns (see (A5)). In these cases to avoid any interferences, the design of the rules could be revised based on an analysis of the rules for conflicts. In particular for (A5), this is always possible by splitting the rule into multiple ones taking a larger context into account such that the overlap with the positive patterns is always included. If this is not possible or feasible, the *receding horizon* concept from model predictive control [1] of k can be employed where only the first k steps of the plan are realized before the further steps are re-planned anew. By using a receding horizon of $k = 1$, we can ensure that the repeated planning steps take the effects of the executed rules into account.

VI. Experimental Evaluation

To evaluate our approach, we use a simulator of mRUBiS [30], a variant of the common RUBiS that is frequently used for validating self-adaptation mechanisms [24]. Having fault injection capabilities, the simulator emulates the failures in the system by reflecting them in the runtime model as it would be otherwise done by monitoring the faulty system. mRUBiS hosts different numbers of shops (1 to 1000), each containing 18 components with a different criticality and connectivity.[2] The overall utility of a shop is the sum of the sub-utilities of all the components in the shop. As described in Section II-A, we equipped mRUBiS with a MAPE-K feedback loop. The three issues CF1, CF2, and CF3 are the negative patterns that affect the system. The rule set $\mathfrak{R}$ includes the adaptation rules each representing a repair plan. Each rule has two attributes, costs and utilityIncrease (see Fig. 1). Costs refers to the execution time of the rule and utilityIncrease is the impact on the utility of the affectedComponent when applying the rule.

We validate the optimality of our scheme with analytical experiments (Section VI-A). Moreover, we investigate the scalability and performance of our approach in a comparative study with two alternative approaches (Section VI-B). Thus, the study compares three solutions:

1) Static Approach: This approach is purely rule-based and uses static priorities without any utility function. Thus, the costs and utilityIncrease of the rules are defined at design time

[2]The experiments and simulations have been conducted on a machine with OS X 10.10, Intel processor 2.6GHz core $i5$, and 8GB of memory.

so that for each CF the repair rule is selected staticly. The utilityDrop caused by each CF is also estimated at design time which leads to a fixed order in which the issues are resolved.

2) Solver-based Approach: This approach is purely utility-based and uses the IBM ILOG CPLEX constraint solver [16] for planning. Specifically, it uses the utility function described in equation (1) for the sequence of rule applications as its objective function. The tasks of assigning proper adaptation rules to each CF and ordering them are defined as optimization problems. This approach maximizes the objective function as the overall utility of the system after each decision.

3) Utility-driven Approach: Our approach computes the impact of different adaptation rules at runtime using the utility function shown in equation (1) and selects the one with largest impact on the overall utility. The order in which CFs are addressed and the proper adaptation rule to resolve the CFs are decided based on the runtime observations regarding the affectedComponent and the utility drop caused by the CFs. This approach is referred to as *u-driven* in the following.

Thus, the three approaches have different planning phases while they share the same incremental behavior—as suggested for our approach—for the other phases of MAPE-K.

A. Analytical Experiments

The conducted experiments for analytical purposes are set to separately evaluate the two main steps of our approach. In these experiments, we consider mRUBiS with 100 shops (1800 components). The experiment starts with occurrences of three failures of type CF1, CF2, and CF3 causing the utility of the system to drop. Utility drops are followed by three MAPE executions. During each MAPE cycle, we consider a receding horizon of size one resolving one CF in one MAPE execution. As the effort of the repeated planning step is negligible due to its incremental nature, a receding horizon of 1 will take the effects of the executed rules into account, even though assumptions (A3) to (A5) do not hold.

Here, we investigate that the u-driven approach makes the optimal decision during incremental rule matching for CFs by selecting the rule that results in the maximum increase in the overall utility. In contrast, the static approach fails to do so and hence is non-optimal. We also show how the order in which the adaptation rules are executed impacts the achieved reward.

When a match for an issue is detected, our approach computes the utilityIncrease and costs of all possible matches among the adaptation rules. The effect of the increase in the utility achieved by applying each rule remains in the system as long as the same component is not affected again by another issue. However, the costs of applying a rule has only a short time effect on the overall utility and defines when the expected increase of the utility can be realized. Thus, in our approach, rules with the highest utilityIncrease are prior to those with lower increase but less costs. The type of the occurred issue and the specific component that is affected by the issue determine the utilityIncrease and costs of the rules.

Fig. 10 describes a case where the static approach fails to reach the maximum utility due to non-optimal rule selection.

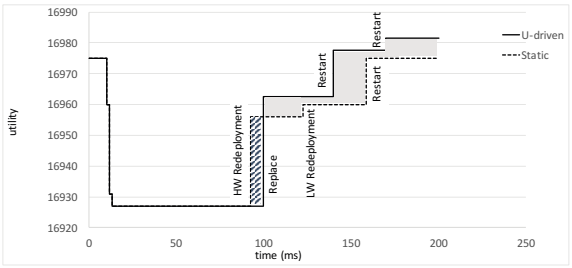

Fig. 10: Lost Reward Due to Non-optimal Decisions.

During the first MAPE cycle, both approaches select CF3 to be resolved first. The static approach performs a Heavy Weight (HW) Redeployment while the u-driven approach Replaces the affected component and reaches a higher utility. Here, the static approach selects a rule with less costs and manages to have the utility increase faster than the u-driven approach but obtaining a considerably smaller reward (the hachure region after the first increase). The impact of this non-optimal rule selection remains in the system during the whole experiment and results in a lower reward for the static approach equal to the area of gray colored regions. In the second MAPE cycle, the static approach resolves CF1 by a Light Weight (LW) Redeployment while the u-driven approach decides to resolve CF2 by a Restart of the component which has higher impact on the overall utility. In the third MAPE cycle, the static approach resolves CF2 by a Restart and reaches the same increase in utility as the u-driven approach in the second MAPE cycle, but with a delay and thus with a lower reward as the utility is lower during the whole time. The u-driven scheme repairs CF1 by a Restart in the last MAPE cycle. The static approach is slightly faster than the u-driven approach due to avoiding all the runtime computations. The gray and hachure regions represent respectively the lost and gained utility of the static compared to the u-driven approach. The additional utility gained by the static approach due to less overhead and choosing the cheaper HW Redeployment over the Replace rule does not compensate for the loss of reward due to making non-optimal decisions.

To back our claim for optimality of the u-driven scheme, our approach executes the adaptation rules in the optimal order such that the maximum utility over time is achieved. We investigate this issue in Fig. 11. The order in which our scheme resolves the issues is such that those resulting in a higher increase in utility are prior to those with lower impacts. The static approach decides for the order at design time. This can be done considering the type of the potential issues. A reasonable order of the three issues in our example is: severe crashes (i.e., unplanned removals) of components (CF3), crashes of components that, however, might still be operating to a certain extent (CF1), and occurrences of Failures such as exceptions (CF2). This ordering fails to take into account the utility of the affectedComponent which is a function of criticality, connectivity, and reliability. Such properties can dynamically change such that they are only known at runtime and cannot be foreseen at design time. Fig. 11 illustrates a case where the

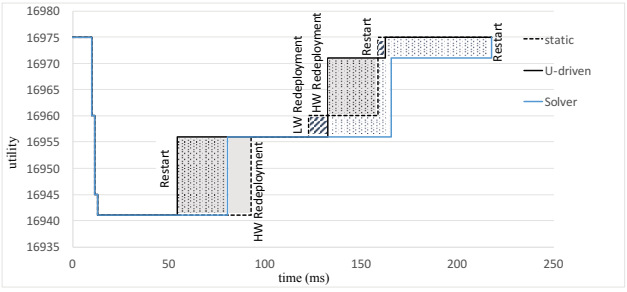

Fig. 11: Lost Reward Due to Wrong Ordering and Overhead.

static approach fails to address the issues in the right order. Despite the fact that both, the u-driven and static approaches achieve the same final utility, which is not necessarily always the case (cf. Fig. 10), the static approach loses reward equal to the gray regions and gains only a slight improvement due to the lower overhead in planning time (hachure region).

Considering Fig. 11, both, the u-driven and static approaches repair CF3 in the first MAPE cycle. The static approach applies a HW Redeployment while the u-driven approach Restarts the affected component reaching the same utility but with considerably less cost. In this MAPE cycle, the static approach selects a rule with a similar utilityIncrease to the one selected by the u-driven approach but with higher costs such that it loses utility equal to the area of the first gray region. In the second MAPE cycle, the static approach resolves CF1 by a LW Redeployment while the u-driven approach decides to resolve CF2 by a HW Redeployment since solving CF2 has a higher impact on the overall utility. In the third MAPE cycle, the static approach resolves CF2 by a Restart and reaches the same increase in utility as the u-driven approach, but loses utility over time due to the wrong execution order. The u-driven scheme saves the repair of CF1 by a Restart for the last MAPE cycle since in this case CF1 has less impact than CF2 and CF3 on the utility.

We conducted the same experiment to compare the solver-based and u-driven approaches. As depicted in Fig. 11, both approaches make similar decisions regarding both rule matching and ordering of the adaptation rules and reach the optimal configuration. However, the solver-based approach reaches it after a considerable delay due to its computational overhead for planning, which depends on the size of the architecture and number of the issues. It is visible that despite the fact that both approaches select the same rules each time with similar costs and have the same ordering of the issues, the solver approach finishes with a large delay compared to the u-driven approach.

TABLE I: Planning Time of the Approaches in (ms).

No. of Components	1 Failure			10 Failures			100 Failures			1000 Failures		
	Static	U-driven	Solver	Static	U-driven	Solver	Static	U-driven	Solver	Static	U-driven	Solver
18	0.76	0.89	5.02	10.37	14.36	56.68	-	-	-	-	-	-
180	0.68	0.89	5.01	9.71	13.58	59.07	14.22	17.7	219.54	-	-	-
1800	0.61	0.74	4.83	10.6	13.47	58.24	13.82	26.65	211.09	54.5	60.09	3216.6
18000	0.65	0.71	4.9	10.14	13.87	71.93	21.8	26.38	271.51	127.8	171.31	3611.95

B. Experiments for Performance

To compare the performance of the approaches, we tested them on mRUBiS with 4 different sizes of the architecture and with scenarios in which 1, 10, 100 and 1000 failures occur. In each scenario, the equal number of failures of all CF types are injected. The measurements are repeated at least 300 times and until the standard deviation is 5% or lower. The measurements were conducted following benchmarking guidelines [28]. All three approaches are tested with the same scenarios. Since the decision-making process takes place during the planning phase and the potential overheads due to runtime computations are reflected there, we only present the data for the planning phases of the approaches in Table I.

For the measurements, we only consider meaningful combination of architecture size and number of failures. Thus, we do not inject a large number failures to small architectures. Therefore, we do not present any data where more than 10 (100) failures occur in a system with 18 (180) components.

As the number of failures in the system increases, the planning time of all approaches increases as well. However, this growth is more drastic for the solver as the size of the optimization problem is growing. Based on the presented data, the solver-based approach requires between 295% to 5953% more time for planning than the u-driven approach [42 to 3156 (ms)]. The solver approach always reaches the same optimal configuration as our u-driven scheme, but it can have an extreme overhead in the case of a large number of issues. The additional planning time required by the u-driven approach compared to the static approach varies between 9% to 92%, [0.06 to 12.8 (ms)].

VII. Related Work

As the related work of this work, we investigate how the trade-off between long-term planning and aiming for an optimal repair or settling for a quick and efficient adaptation has been practiced in the field of self-adaptive software.

On one end of the spectrum, there are optimization-based approaches employing runtime reasoning. An objective function is computed at runtime to investigate all the potential decisions, thus encountering scalability and efficiency issues [9,20]. Employing utility functions and utility-driven decision-making schemes have been extensively investigated [12,21]. MUSIC [26] is a planning-based middleware for component-based application that plans the adaptation by exploiting the characteristics of component implementations for the software architecture. [20] applies a reinforcement learning-based approach for on-line planning. [6] solves an optimization problem to find the optimal set of features that maximizes the utility via a learning-based method. Solving an optimization algorithm for each reconfiguration at runtime causes large overheads. The outlined utility-driven approaches pursue a search-based optimization in the solution space that often do not scale well for complex systems. Such approaches manage to find the optimal configuration but there is no guarantee to reach the result within a reasonable time, for instance, when quickly needing a repair plan. In [29], we suggested to reduce

the search space to speed up repair and avoid too long delays. However, the approach proposed here computes the utility for each potential adaptation strategy in an incremental scheme taking into account the current issues that affect the achieved utility. Therefore, the approach is scalable and does not have to restrict the search space for the considered self-healing setting.

On the other end of the spectrum, there are pure rule-based approaches [8]. They are recognized to be efficient and stable in predictable domains and support the early validation [9]. These approaches provide a quick recovery from a goal violation, however, they often result in sub-optimal solutions since they ignore the scenarios that are unforeseen at design time [3]. Rainbow applies utility theory in combination with a stochastic model of the possible outcomes of the reasoning process [4]. While in our approach the utility of the adaptation rules is dynamically computed at runtime, Rainbow considers the success rate of adaptation rules in the past to rank them.

The proposed approach is distinguished from the existing work as it is fast and optimal since it employs adaptation rules and does not struggle with scalability issues. Employing a utility function guarantees optimal adaptation decisions on top of the applied rule-based scheme. However, unlike the optimization-based approaches, the incremental manner of computing the utility function over the patterns makes the approach scalable for large complex systems.

VIII. Conclusion and Future Work

Achieving optimal adaptation decisions online within a reasonable time is an important challenge addressed by this work. We presented a novel approach to improve the self-healing reward by combining utility-driven and rule-based adaptation at the architectural level to achieve the benefits of each of them. The approach addresses the requirements of scalability and optimality regarding the utility computation. Our experiments demonstrate that our approach results in significantly improved reward compared to an alternative static approach while only having a negligible overhead. The comparison of our approach to a solver-based solution shows that both perform optimal adaptation decisions while our approach drastically reduces the computation efforts for planning self-adaptation.

However, the presented approach has some limitations that we plan to address in future work. This includes weakening the assumptions, supporting more complex utility functions, and studying how to support other capabilities of self-adaptive software such as self-configuration and self-optimization.

References

[1] K. Angelopoulos, A. V. Papadopoulos, V. E. Silva Souza, and J. Mylopoulos. Model predictive control for software systems with cobra. In *SEAMS*, pages 35–46. ACM, 2016.

[2] G. Blair, N. Bencomo, and R. B. France. Models@run.time. *Computer*, 42(10):22–27, 2009.

[3] S.-W. Cheng. *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation*. PhD thesis, Carnegie Mellon Univ., Pittsburgh, USA, 2008.

[4] S.-W. Cheng and D. Garlan. Stitch: A language for architecture-based self-adaptation. *J. Syst. Softw.*, 85(12), 2012.

[5] S.-W. Cheng, D. Garlan, and B. Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *SEAMS*. ACM, 2006.

[6] N. Esfahani, A. Elkhodary, and S. Malek. A learning-based framework for engineering feature-oriented self-adaptive software systems. *IEEE Transactions on Software Engineering*, 39(11):1467–1493, 2013.

[7] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In *TAGT*, volume 1764 of *LNCS*, pages 296–309. Springer, 1998.

[8] F. Fleurey, V. Dehlen, N. Bencomo, B. Morin, and J.-M. Jézéquel. Modeling and validating dynamic adaptation. In *Models in Software Engineering*, volume 5421 of *LNCS*, pages 97–108. Springer, 2009.

[9] F. Fleurey and A. Solberg. A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In *MoDELS*, LNCS 5795, pages 606–621. Springer, 2009.

[10] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, 2006.

[11] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *FOSE*, pages 37–54. IEEE, 2007.

[12] J. Franco, F. Correia, R. Barbosa, M. Zenha-Rela, B. Schmerl, and D. Garlan. Improving self-adaptation planning through software architecture-based stochastic modeling. *J. Syst. Softw.*, 115:42–60, 2016.

[13] D. Garlan, B. Schmerl, and S.-W. Cheng. Software architecture-based self-adaptation. In *Autonomic Computing and Networking*, pages 31–55. Springer, 2009.

[14] S. Ghahremani, H. Giese, and T. Vogel. Towards linking adaptation rules to the utility function for dynamic architectures. In *SASO*, pages 142–143. IEEE, 2016.

[15] C. Ghezzi. Evolution, adaptation, and the quest for incrementality. In *Large-Scale Complex IT Systems. Development, Operation and Management*, volume 7539 of *LNCS*, pages 369–379. Springer, 2012.

[16] IBM. IBM ILOG CPLEX Optimization Studio. http://www-03.ibm.com/software/products/en/ibmilogcpleoptistud.

[17] J. O. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[18] J. O. Kephart and R. Das. Achieving self-management via utility functions. *Internet Computing, IEEE*, 11(1):40–48, 2007.

[19] J. O. Kephart and W. E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *POLICY*, pages 3–12. IEEE, 2004.

[20] D. Kim and S. Park. Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. In *SEAMS*, pages 76–85. IEEE, 2009.

[21] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE*, pages 259–268. IEEE, 2007.

[22] J. Magee and J. Kramer. Dynamic structure in software architectures. In *SIGSOFT FSE*, pages 3–14. ACM, 1996.

[23] P. Oreizy, M. M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[24] T. Patikirikorala, A. Colman, J. Han, and L. Wang. A systematic survey on the design of self-adaptive software systems using control engineering approaches. In *SEAMS*, pages 33–42. IEEE, 2012.

[25] V. Poladian, J. P. Sousa, D. Garlan, and M. Shaw. Dynamic configuration of resource-aware services. In *ICSE*, pages 604–613. IEEE, 2004.

[26] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz. Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments. In *SEfSAS*, volume 5525 of *LNCS*, pages 164–182. Springer, 2009.

[27] D. E. Seborg, D. A. Mellichamp, T. F. Edgar, and F. J. Doyle. *Process Dynamics and Control*. John Wiley & Sons, 3rd edition, 2011.

[28] P. Sestoft. Microbenchmarks in java and c#, 2013.

[29] M. Tichy and H. Giese. A self-optimizing run-time architecture for configurable dependability of services. In *Architecting Dependable Systems II*, volume 3069 of *LNCS*, pages 25–51. Springer, 2004.

[30] T. Vogel. Modular Rice University Bidding System (mRUBiS), 2013. http://www.mdelab.de [Online; accessed 09-May-2016].

[31] T. Vogel and H. Giese. Adaptation and abstract runtime models. In *SEAMS*, pages 39–48. ACM, 2010.

[32] T. Vogel, S. Neumann, S. Hildebrandt, H. Giese, and B. Becker. Model-driven architectural monitoring and adaptation for autonomic systems. In *ICAC*, pages 67–68. ACM, 2009.

[33] T. Vogel, S. Neumann, S. Hildebrandt, H. Giese, and B. Becker. Incremental model synchronization for efficient run-time monitoring. In *Models in Software Engineering*, volume 6002 of *LNCS*, pages 124–139. Springer, 2010.