Performance Analysis of Self-Adaptive Systems for Requirements Validation at Design-Time

Matthias Becker
Software Engineering Group,
Heinz Nixdorf Institute,
University of Paderborn
Zukunftsmeile 1
33102 Paderborn, Germany
matthias.becker@upb.de

Markus Luckey
Department of Computer Science,
University of Paderborn
Zukunftsmeile 1
33102 Paderborn, Germany
markus.luckey@upb.de

Steffen Becker
Software Engineering Group,
Heinz Nixdorf Institute,
University of Paderborn
Zukunftsmeile 1
33102 Paderborn, Germany
steffen.becker@upb.de

ABSTRACT

Self-adaptation allows continuously running software systems to operate in changing and uncertain contexts while meeting their requirements in a broad range of contexts, e.g., from low to high load situations. As a consequence, requirements for self-adaptive systems are more complex than requirements for static systems as they have to explicitly address properties of the self-adaptation layer. While approaches exist in the literature to capture this new type of requirements formally, their achievement cannot be analyzed in early design phases yet. In this paper, we apply RELAX to formally specify non-functional requirements for self-adaptive systems. We then apply our model-based SimuLizar approach for a semi-automatic analysis to test whether the self-adaptation layer ensures that these non-functional requirements are met. We evaluate our approach on the design of a proof-of-concept load balancer system. As this evaluation demonstrates, we can iteratively improve our system design by improving unsatisfactory self-adaption rules.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; D.2.11 [**Software Engineering**]: Software Architectures—*Languages*

Keywords

self-adaptive, performance analysis, performance engineering, requirements

1. INTRODUCTION

To achieve cost savings, IT companies recently started migrating their systems to the cloud. The cloud provides an infrastructure to run IT systems with an on-demand price model. Hence, to achieve greatest possible cost savings, an efficient use of cloud resources is necessary, and thus, the reduction of cloud resource consumption as far as possible. That is, the cloud resource consumption has to be adapted to varying contextual conditions, such as load. Under high load conditions more cloud resources are consumed while in low load conditions cloud resources should be released. *Self-adaptation* is an approach to achieve the aforementioned adaptation to the context in a timely manner. For this, a self-adaptive cloud system monitors its context, e.g. the load, and performs adaptations, e.g. the release of cloud resources.

In contrast to non-adaptive systems, self-adaptive systems are designed to operate in a range of different contextual conditions. For instance, software engineers have to consider not only the maximum load (worst case), but the complete *range* of the possible load in order to determine the required resources and adapt accordingly. The transition from a single fixed context to a range of contexts rises a challenge for software engineering. Current software engineering approaches aim to explicitly consider self-adaptation as a separate concern [8, 13, 19]. For instance, in the requirements engineering phase, the context range can be considered explicitly with the RELAX requirement language [23]. RELAX allows softening requirements to enable the system to gradually fulfill these requirements, e.g., *as good as possible*, within the range of possible contexts. However, approaches for analyzing self-adaptive designs concerning the gradual fulfillment of requirements are still missing.

Classical performance engineering approaches [9, 5, 1] are suitable for analyzing static systems with fixed contexts. However, these approaches do not yet consider the contextual range typical for self-adaptive systems. Neither do classical performance engineering approaches consider a gradual requirement fulfillment. Only few self-adaptive system analysis approaches exist that explicitly consider RELAX requirements analysis [14].

Our goal is to provide performance engineering approach suitable for analyzing self-adaptive systems, explicitly taking gradual fulfillment of performance requirements into account. In this paper, we provide a formalization for self-adaptive systems and for RELAX-based analysis. We extended our SimuLizar approach [3] to enable a semi-automatic analysis of whether RELAX requirements are met. For this, we simulate self-adaptive systems and analyze the gradual fulfillment of requirements during this simulation.

We evaluated our extended SimuLizar approach on the design of a proof-of-concept self-adaptive load balancer system. The evaluation shows that SimuLizar provides feedback for performance engineers in order to iteratively improve the design of the self-adaptation layer.

The remainder of this paper is organized as follows. In Section 2 we outline a specification for a small self-adaptive load balancer which we use as a motivating example throughout the following sections. We summarize the basics self-adaptive systems as well as RELAX as a language for specifying requirements for these system class in Section 3. Our SimuLizar approach for semi-automated self-adaptive system analysis is detailed in Section 4. We evaluate whether SimuLizar provides sufficient feedback for software engineers to iteratively improve self-adaptive system designs in Section 5. In Section 6 we discuss related work in the area of self-adaptive system analysis. Finally, we conclude our work and discuss future work in Section 7.

2. MOTIVATING EXAMPLE

We provide high-level requirements for a small load balancer example system that we will use in the remainder of this paper. The load balancer example serves as a running example of a self-adaptive system. Our focus in this paper is on performance-related requirements. Other non-functional and functional requirements are out of scope and will not be discussed in detail.

2.1 Requirements

We want to design a load balancer system, that distributes load across an own server and a remote rented server. The load balancer accepts client requests and delegates them to one of the servers. Responses are sent directly from either the own server or remote rented servers to the clients.

The following requirements (R1-R3) shall be fulfilled by the load balancer system:

R1 The system shall keep response times for user requests low. The mean response time for a user request shall not be greater than 3 seconds.

R2 The system shall automatically detect if additional infrastructure resources are needed and automatically rent these resources in order to maintain a mean response time of 3 seconds.

R3 The system shall keep the rental fee at minimum.

For the sake of simplicity, we assume that both servers have identical resources. Hence, a request causes constant load of 0.3 seconds uncontended CPU time on the server to which it is delegated to. We assume that we have to pay an additional load-dependent fee for utilizing the external server. That could be the case for a cloud provider for example. Commonly, cloud providers do not charge depending on caused load yet, but our industry partners predict that this will be an option in the near future.

2.2 Initial Design

A system, like the load balancer described above, can be implemented in many different ways. It is the task of a software engineer to find a design, such that all requirements are met. Whether requirements can be met also depends on the environmental context of the system. For example, a system may be able to maintain a response time of 3 seconds or less when it only has to serve 5 user requests per second. If the same system has to serve 500 user requests per second it may not be able to maintain the same response time. As a consequence, software engineers design self-adaptive systems by taking the uncertainty of the environment context into account. A self-adaptive system is able to operate within a range of environmental contexts and still meets the requirements.

We assume that a software designer explicitly decides to design our load balancer system as a self-adaptive system. He wants the load balancer to self-adapt its load balancing strategy depending on its environmental context, i.e., the current load. To enable the self-adaptation the system has to be aware of its context, i.e., it has to monitor the necessary context. This is one of the first design decisions the software engineer has to take. In our example, the requirements R1 and R2 make statements about the mean response time. Hence, the system has to monitor the response time. However, it might not always be possible to monitor certain environment contexts directly. In case our example load balancer cannot maintain the mean response time of 3 seconds or less, it can rent additional external servers and route some user requests to these external servers. Here, the software engineer has to make three further decisions. First, the software engineer has to decide which load balancing strategy should be implemented. In our example, he decides to balance load using a random distribution to one of the servers. Second, the software engineer has to decide how the system can detect when to rent external servers. It is required that the mean response time needs to be 3 seconds or less (R1) and only if the own server is not able to maintain this response time additional resources can be rented (R2). A follow-up decision that has to be made is the time frame over which the system should calculate the mean response time, i.e., the size of response time measurement batches. A smaller batch size enables to react faster on high mean response times, but also increases the risk to rent external resources when they are actually not needed. Hence, the batch size is a crucial design decision that may have great effect on the system's self-adaptation. In Section 5, we show how we leverage our approach to optimize this batch size. Third, the software engineer has to decide how many user requests should be delegated to the external server. Since the rental fee for the external server is load-dependent, the load balancer should only delegate as many load to the external server as required to maintain the 3 second mean response time (R3). The software engineer could also decide to delegate at maximum 50% of the load to the external server in order to limit costs.

3. FOUNDATIONS

Before we introduce our SimuLizar approach, we briefly summarize the basics of specifying requirements for self-adaptive systems with RELAX [23] and the state-of-the-art in self-adaptive system engineering. With SimuLizar we can model and simulate self-adaptive systems. Making use of the RELAX formalization, we can reason about whether requirements are met during this simulation.

3.1 Requirements with RELAX

RELAX is a requirements language that explicitly takes uncertainty in environmental context of a software system into account. In contrast to classical requirements, in RELAX a requirement engineer can distinguish between requirements that a system has to meet at all time (invariants) and requirements that only need to be met in a best-effort fashion. The latter requirements are called RELAX-ed requirements.

A RELAX requirement consists of two parts. The first part is a RELAX expression. In the second optional part uncertainty factors in the environmental context can be specified in a declarative fashion. We formulate R3 from our load balancer example as a RELAX requirement as shown below. The RELAX keywords are denoted in capital letters.

R3: The system SHALL keep the rental fee AS CLOSE AS POSSIBLE to 0.

ENV: rental fee

MON: number of requests delegated to external server

REL: the more requests are delegated to the external server, the higher the rental fee

In the original requirement R3, we required the system to keep the renting fees at minimum. This may be in contradiction to R2 where we require the system to maintain a mean response time of 3 seconds or less by renting external resources. Relaxing R3, as shown above, enables the system to potentially fulfill both requirements R2 and R3 at once. Furthermore, the RELAX requirement describes uncertainty in the environmental context using the ENV keyword. In the RELAX requirement the rental fee may be unknown by the system. However, the MON and REL statements describe how this context can be monitored indirectly. A MON statement describes something that the system can monitor. The REL statement describes the relation between the unknown context and what can be monitored by the system. In this example, we can monitor the number of requests delegated to the external server. With this number we can estimate the renting fees.

The semantics of RELAX requirements are defined in terms of fuzzy branching temporal logic (FBTL) [17]. FBTL is a temporal logic including concepts for uncertainty in time and and logic using fuzzy logic concepts. That is, with FBTL one can describe fuzzy states and fuzzy events. In fuzzy logic, variables have a gradual truth value ranging from 0 to 1 in contrast to classical logic where truth values can only be either false (0) or true (1). The basis for fuzzy logic are fuzzy sets which are sets whose elements have gradual membership. That is, a fuzzy set A is a pair (A, m) where A is a set and $m : A \to [0, 1]$ is the membership function for A. A value x is member of the fuzzy set A, i.e., $x \in A$, if its membership value is greater than 0. That is, $x \in A \Leftrightarrow m(x) > 0$. Values higher than 0 denote the gradual membership in A of x.

For a RELAX requirement this means that it can be gradually satisfied, i.e., it can range from "not met" to "fully met". In the example above, the requirement would be fully met if there are no rental fees at all. This would be the case if the system does not delegate requests to the external server at all. However, the fulfillment of the requirement may be relaxed at run-time, e.g., in case the system cannot maintain a mean response time of 3 seconds. In that case the system could decide to not fully meet this requirement in favor of requirement R1.

RELAX requirements can be formalized as FBTL formulae using the common operators A for all, E for exist, G for general, and F for finally. Additionally, the temporal logic χ, which denotes the truth value of a formula after a time duration, can be used. We can formalize the RELAX-ed requirement R3 from above as

$$AG(\Delta(\Phi_{RF}, t) \in S) \tag{1}$$

where $\phi_{RF} \in R$ denotes the requirement "low rental fees" and $t \in T$ is a point in time. The function Δ quantifies the rental fees Φ_{RF} at a point in time t as a natural number, i.e., $\Delta : R \times T \to \mathbb{N}$. For example, $\Delta(\Phi_{RF}, 10) = 30$ denotes that the rental fees at time 10 are 30. S is the fuzzy set, with membership function m, representing all (system) states that meet the requirement ϕ_{RF}. That is, if $\Delta(\Phi_{RF}, t) \in S$, i.e., it is a member of the fuzzy set S and the requirement ϕ_{RF} is fulfilled. Hence, the membership function m defines the range where a requirement is met. Intuitively, the FBTL formula means that the rental fees shall always (AG) be low. Where the range of "low" is defined via the membership function m.

Note, that we deviate from the original FTBL formalization of RELAX. In the original formalization a requirement has to be met eventually, i.e., $\mathbf{AF}(\Delta(\Phi, t) \in S)$. We define that a requirement has to be met in every state of a system, i.e., $\mathbf{AG}(\Delta(\Phi, t) \in S)$.

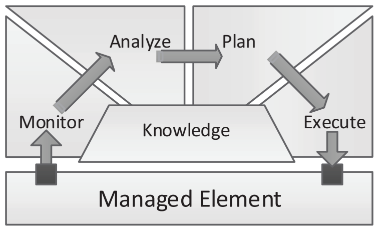


Figure 1: MAPE-K feedback loop [18].

3.2 Self-Adaptive System Modeling

Self-adaptive systems are able to adapt their structure, behavior, or allocation in order to react to a changing environmental context. On a conceptual level, these systems often follow the MAPE-K feedback-loop [18] approach, as illustrated in Figure 1.

The MAPE-K feedback loop consists of four steps. First, the managed element, the self-adaptive system and its context, is *monitored* via defined sensors, e.g., for the mean response time. Second, the monitored data is *analyzed*, e.g., whether the predefined threshold of 3 seconds is exceeded. Third, if the analysis reveals that a self-adaptation is required, it is *planned*, e.g., a predefined self-adaptation rule like the load balancing rule is selected. Finally, the self-adaptation rule is *executed* via effectors of the managed element. In all steps, a *knowledge base* containing system information, i.e., a runtime model, can be accessed. For example, monitoring data can be stored in the knowledge base.

The state-of-the-art in self-adaptive system design is to explicitly reveal the MAPE-K control loop in order to make it analyzable. For this purpose, new modeling viewpoints are introduced [4] which are dedicated to specify sensors, monitors, and self-adaptation rules. Analyzing the MAPE-K design allows us to check whether the system under study is capable of fulfilling its requirements. Our SimuLizar approach includes a modeling approach and provides a semi-automatic analysis to check whether performance requirements are fulfilled.

4. SIMULIZAR

With SimuLizar, we are able to model self-adaptive system designs and predict their performance already at design-time. This enables us to analyze whether requirements for the self-adaptation layer are met or otherwise to iteratively improve the self-adaptation design. In this section, we first introduce a formalization for self-adaptive system models and their analysis. Second, we provide a mapping from the formalization to SimuLizar's modeling and simulation concepts. We illustrate how SimuLizar can be applied for a semi-automatic performance analysis. In Section 5, we apply SimuLizar to our motivating load balancer example to iteratively find a design that meets the requirements.

4.1 Formalization

In this section, we provide three formal definitions. First, a definition for self-adaptive system models in general. Second, a definition for a simulation of self-adaptive systems. Finally, we are using the FBTL formalization of RELAX to formally define the requirement analysis of a self-adaptive system.

We define a self-adaptive system model as follows.

Definition 1 (Self-Adaptive System Model). *A self-adaptive system model M is a tuple $(S, E, \sigma, \varepsilon, s_0)$, in which*

- *S is the set of system states, $S = \{s_0, s_1, s_2, \ldots\}$;*
- *E is the set of monitored environment states, $E = \{e_0, e_1, e_2, \ldots\}$;*
- *σ is the set of self-adaptation rules, $\Sigma = \{\sigma_0, \sigma_1, \sigma_2, \ldots\}$, where σ is a function $\sigma : S \times E \to S$.*
- *ε is a function $\varepsilon : T \to E$ that depending on the time $t \in T$ returns an environment state $e \in E$..*
- *s_0 is the initial state of the system.*

Note, that the actual configuration, i.e., wiring of components and allocation, is encoded in the system states here. The initial configuration is denoted by s_0. Self-adaptation rules are formally denoted as a function σ_i which maps a system state and the monitored environment state to another system state. That is, it changes the systems state according to the environmental context.

So far, we have formally defined a self-adaptive system. In order to provide a mapping to the RELAX formalization, we have to formally define a SimuLizar simulation and its result.

Definition 2 (Self-Adaptive System Simulation). *A self-adaptive system simulation is a function $\Delta_M(\phi, t)$ which returns a quantification for a metric ϕ at time t of the simulated model M. We denote*

- *M is a self-adaptive system model,*
- *$t \in T$ denotes a point in the simulation time T,*
- *ϕ represents a metric, e.g., response time in seconds.*
- *$\Delta_M(\phi, t)$ is quantification of ϕ at simulation time t.*

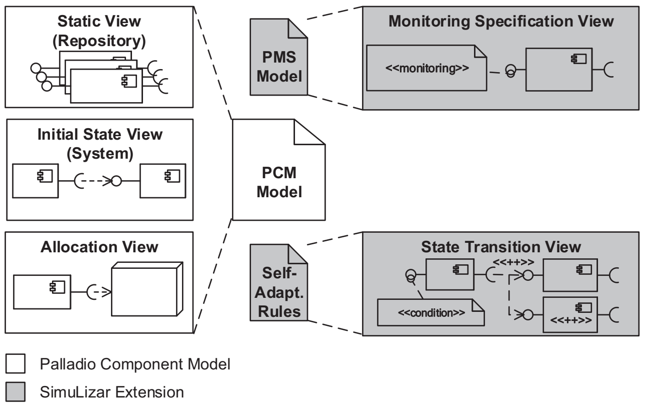


Figure 2: SimuLizar Self-Adaptive System Model.

That is, a self-adaptive system simulation of M provides us with a function $\Delta_M(\phi, t)$ which quantifies a metric ϕ, e.g. response time in seconds, at (simulation) time t. Note, that the simulation and thus $\Delta_M(\phi, t)$ is specific for a single trace of system and environment states.

If we have a RELAX requirement ϕ_R which can be quantified via a metric, we can check if ϕ_R is met at a given point in time t in a simulation. However, for most requirements we are usually not interested if a requirement is fulfilled at a single point in time, but over a certain time span $[i, j)$. Hence, we have to define how we can analyze if a requirement is met over a given time interval $[i, j)$.

Definition 3 (Requirement Analysis). *We analyze whether a self-adaptive system model M meets a (RELAX-ed) requirement $\phi_R \in R$ over a time interval $[i, j)$, i.e., $M \models_{[i,j)} \phi_R$, using the following formalization*

- *M is a self-adaptive system model,*
- *$\Delta_M(\phi_R, [i, j)) = \langle \Delta(\phi_R, t) \mid i \leq t < j \rangle$ is quantification of ϕ_R in the time interval $[i, j)$, i.e., the predicted response time of M in time $[i, j)$.*
- *$M \models_{[i,j)} \phi_R$ iff $\Delta(\phi_R, [i, j)) \in F_R$,*
- *F_R is a fuzzy set with membership function m_F. All members of F_R fullfill requirement ϕ_R.*

That means, with a requirement analysis, i.e., function $\Delta_M(\phi_R, [i, j))$, we can quantify a requirement ϕ_R of the simulated system in a time interval $[i, j)$. In order to analyze whether the requirement ϕ_R is met in the simulation, we have to define a membership function $m : F_R \to [0; 1]$ for the fuzzy set F_R.

4.2 Modeling

SimuLizar provides a modeling approach for self-adaptive systems based on ideas presented in [4] and the formalization presented above. In SimuLizar, a self-adaptive system model consists of two viewpoints with several views each. First, a system type viewpoint consisting of three views: a *static view*, a *monitoring specification view*, and an *allocation view*. Second, a runtime viewpoint including the *initial state view* and the *state transition view*.

SimuLizar's modeling approach is based on the Palladio Component Model (PCM) [6]. With PCM classical, i.e., non-adaptive, systems can be modelled and their performance analyzed. Single components of the system are specified in the *static view* (PCM repository). The initial system configuration, i.e., the initial state s_0, is specified in the *initial state view* (PCM system). In the *allocation view* the component's allocation is specified. Additionally, we introduce two new modelling artifacts with SimuLizar, as illustrated in Figure 2. First, the Palladio Measurement Specification (PMS), a domain-specific language to describe sensors for a self-adaptive system. Second, we introduce self-adaptation rules, i.e., Σ in our formalization. For now, we do not explicitly specify effectors in SimuLizar, but assume that all parts of the system potentially can be adapted.

A PMS monitor specification consists of four characteristics: (1) a sensor location, (2) a performance metric type, (3) a time interval type, and (4) a statistical characterization. The sensor location specifies the place of the sensor within the system, for example a service call. Currently, we support the following performance metric types: waiting time, response time, utilization, arrival rate, and throughput. Each of these metrics can be measured at the specified sensor location in one of the three batch types: in periodic batches with the size B beginning at time 0.0, periodic bacthes with length B and a delay of δ from the start time, or within a single fixed batch with start time t_{start} and end time t_{stop}. Finally, for each sensor a statistical characterization can be specified to aggregate the monitored data. We support the modes *none*, *median*, *arithmetic mean*, *geometric mean*, and *harmonic mean.*

In our formalization, a self-adaptation rule is a function $\sigma : S \times E \rightarrow S$. That is, the input is the system's state and the environmental context, and the output is a (new) system state. In SimuLizar, self-adaptation rules consist of a condition (input) and a self-adaptation action (output). A condition has to reference a sensor and to provide a boolean term. If the boolean term evaluates to true, the self-adaptation action is triggered. The self-adaptation action part references elements in the PCM model. Variables of these elements can be set using the ≪*assign*≫ keyword, or new instances of model elements can be instantiated using the ≪*++*≫ keyword. We use Story Diagrams [22] to formalize the condition as well as the self-adaptation action as described above. In SimuLizar, the set of all system states S is only implicitly defined via the initial state view (s_0) and the set of all self-adaptation rules Σ.

Currently, we can only define a single, initial environment state e_0 with SimuLizar via the PCM usage model. In this model, we can define different workload scenarios. A workload scenario is either an open workload, i.e., with an unknown number of users, or a closed workload, i.e., with a known maximum number of users. The scenario defines which methods of the system are called by a user and with which rate new users arrive at the system.

Figure 3 illustrates a model of our motivating example using our SimuLizar modeling approach. We model the system according to our initial design ideas in Section 2. In the system type viewpoint, Figure 3a, an allocation view is annotated with measurement specifications, which are defined via the PMS. In this example, the component LoadBalancer is deployed on a node *lbn*. The LoadBalancer component is connected to two Server components via its required interface. The two server nodes are deployed on two different ServerNodes *own* and *ext*. The monitoring

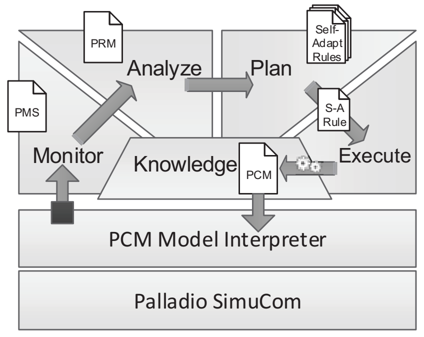


Figure 4: SimuLizar architecture.

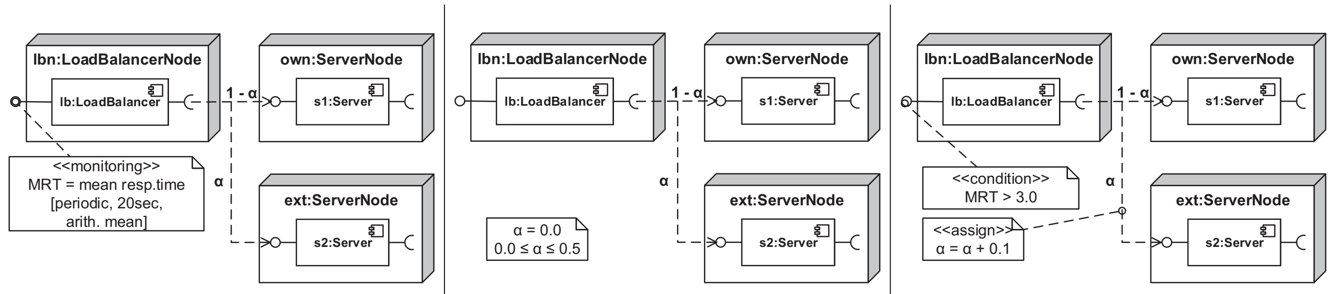
annotation connected to the provided interface of the LoadBalancer component specifies a monitor "MRT" for the mean response time (arithmetic mean) in batches with the size of $B_{\mathrm{MRT}} = 20$ *seconds*. The batch size is a design trade-off. A greater batch size reduces the monitoring overhead and prevents the system to self-adapt too often. A smaller batch size can reduce the time the system needs to detect whether the mean response time is greater than 3 seconds, i.e., R1 is violated. The initial state view, Figure 3b, illustrates the initial configuration of the system. The variable α denoting a branch probability is set to 0.0, meaning that the whole workload is initially handled by server node *own* and there are initially no rental fees (R3). In the state transition view, Figure 3c, the variable α is set to $\alpha + 0.1$ if the condition MRT > 3.0 holds. This reflects the system's reconfiguration to satisfy R2.

4.3 Simulation

Self-adaptive systems modeled with SimuLizar can be simulated with SimuLizar, i.e., in order to get response time predictions for the system under analysis. That is, a SimuLizar simulation is able to quantify performance metrics for a SimuLizar model.

The building blocks of SimuLizar's simulation implementation can be mapped to the MAPE-K feedback loop, as illustrated in Figure 4. In SimuLizar, the managed element is the modeled simulated self-adaptive system. The simulated system is monitored, the monitoring results are analyzed, reconfiguration is planned, and a reconfiguration is executed on the simulated system if required. In a SimuLizar simulation, reconfigurations are model-transformations of the system model.

A *model interpreter* traverses the system model for each single user request. It utilizes the PCM simulation engine, *SimuCom*, for simulating the user and system behavior including the simulation of resources. During the simulation, the simulated system is *monitored* and measurements are taken as specified via *PMS*. Whenever the model interpreter arrives at a monitor place, it simulates a measurement as specified in the PMS model. Once new measurements are available the runtime model of the system, the Palladio Runtime Measurement Model (*PRM*), is updated with the newly taken measurements. An update of the PRM, i.e., new measurements, triggers the *planning* phase. In the planning phase all *self-adaptation rules* are checked. If the condition of a rule holds, the corresponding self-adaptation action is *executed*, i.e., the PCM model is transformed. During the execution phase, the translated model-transformation is ap-



(a) System Type View (b) Initial State View (c) State Transition View

Figure 3: (a) System Type View, (b) Initial State View, and (c) State Transition view.

plied to the system model. Subsequently, the interpreter uses the transformed system model / reconfigured system for new users.

The SimuLizar simulation provides us predicted performance metrics of the simulated system model, i.e., function $\Delta_M(\phi_R, [i, j))$. We can use this information to quantify to which degree fuzzy RELAX-ed performance requirements are met for a given simulation trace, i.e., with respect to the system's and environment's states within time $[i, j)$. Note, for classical performance analysis in contrast to SimuLizar, we can only analyze the time interval $[0, \infty)$, i.e., the quantification function is only defined for $\Delta(\phi_R, [0, \infty))$. This means, we can only check whether a requirement is met if the system under analysis runs infinitely long. This is also called steady state analysis. However, with SimuLizar we are also able to analyze whether requirements are met in a certain time interval $[i, j)$.

5. EVALUATION

We have already evaluated the applicability of SimuLizar for performance predictions in general [3]. In this section we evaluate whether we can iteratively improve a self-adaptive system design using the extended SimuLizar approach. First, we define RELAX-ed requirements for our example load balancer system. Second, we simulate the system with the initial design presented in Section 4.2. We show the simulation results can be used to iteratively improve the system's self-adaptation design. Finally, we discuss the limitations of our approach.

5.1 Defining RELAX-ed Requirements

Table 1 shows the RELAX-ed requirements for our load balancer example from Section 2. The first column identifies the requirement. In the second column, the RELAX-ed requirements are presented using the "AS MANY AS POSSIBLE", "AS CLOSE AS POSSIBLE", and "AS SOON AS POSSIBLE" keywords. We skipped the declarative part to specify contextual uncertainty in the RELAX requirements in the table, because they are analogous to the RELAX requirement presented in Section 3. We provide the FBTL formalization in the third column of the table. The FBTL formula $\phi_{MRT \leq 3s}$ can be informally expressed as "the mean response time (MRT) is 3 seconds or less". The renting fees of the system are denoted with the FBTL formula ϕ_{RF}. Both formulae get quantified in a SimuLizar simulation, i.e., a function Δ is provided to quantify each formula.

To check whether the RELAX-ed requirements are met during a SimuLizar simulation, we additionally have to provide membership functions for F_{R1}, F_{R2}, and F_{R3}. R1 states that as many user request shall be served in 3 seconds or less as possible. We define $s_{i,j}$ as the number of requests with a response time of 3 seconds or less in the time interval $[i, j)$, $a_{a,j}$ the total number of requests in that time interval. The membership function for the fuzzy set F_{R1} that defines the grade to which R1 is fulfilled is defined as

$$m_{F_{R1}}(\Delta(\phi_{RF}, [i,j))) = \begin{cases} 1 & \text{if } \Delta(\phi_{RF}, [i,j)) \leq T_{max} \\ \frac{s_{i,j}}{a_{i,j}} & \text{else} \end{cases} \qquad (2)$$

where $T_{max} = 3$ *seconds* is the mean response time threshold and $[i, j)$ is a time interval with $i, j \in T$. The function has value 1 if the mean response time is less or equal than T_{max}, i.e., the requirement is fully met. In any other case, the function has a value $\frac{s_{i,j}}{a_{i,j}}$. That is, the function reflects the percentage of requests with a request time less or equal T_{max} of all requests in time interval $[i, j)$.

The membership function for the fuzzy duration F_{R2} is defined as

$$m_{F_{R2}}(\Delta(\phi_{RF}, [i,j))) = \begin{cases} 1 - (\frac{t - t(e_d)}{D_{max}}) & \text{if } \Delta(\phi_{RF}, [i,j)) > T_{max} \\ & \wedge t < t(e_d) + D_{max} \\ 1 & \text{else} \end{cases} \qquad (3)$$

where $T_{max} = 3$ *seconds* is the mean response time threshold and $[i, j)$ is a time interval with $i, j \in T$. t is a point in time, i.e., $t \in T$. The maximum time to re-establish a response time of 3 seconds or less is denoted as D_{max}. For our load balancer example, we assume $D_{max} = 40$ *seconds*, i.e., the load balancer system shall re-establish the desired mean response time within 40 seconds. Furthermore, $t(e_d)$ denotes the point in time, when the system detects that the mean response time is greater than 3 seconds. In case $\Delta(\Phi_{RF}, t) > T_{max}$, the value of the membership function $m_{F_{R2}}$ is 1 at time $t(e_d)$ and decreases linearly to 0 at time $t(e_d) + D_{max}$. That is, the requirement R2 is fully met when the mean response time is re-established at time $t(e_d)$ and it is not met if the mean response time is not re-established at time $t(e_d) + D_{max}$.

The membership function for the fuzzy set F_{R3} is defined as

$$m_{F_{R3}}(\Delta(\phi_{RF}, [i,j))) = \begin{cases} 1 & \text{if } \Delta(\phi_{RF}, [i,j)) = 0 \\ 0 & \text{if } \Delta(\phi_{RF}, [i,j)) \geq RF_{max} \\ 1 - \frac{\Delta(\phi_{RF}, [i,j))}{RF_{max}} & \text{else} \end{cases} \qquad (4)$$

where RF_{max} is the rental fee threshold and $[i, j)$ is a time interval with $i, j \in T$. The membership function has value 1 if $\Delta(\phi_{RF}, [i, j)) = 0$, i.e., there are no rental fees. In this

Table 1: RELAX-ed requirements

Req.	RELAX requirement	FBTL formalization
R1	The system SHALL serve AS MANY user request in under 3 seconds AS POSSIBLE.	$AG(\Delta(\phi_{MRT\leq 3s}) \in F_{R1})$
R2	The system SHALL automatically re-establish a mean response time of 3 seconds or less AS SOON AS POSSIBLE (AFTER it detects that the mean response time is greater than 3s).	$A\chi_{>e_d}(A\chi_{\geq d}(\Delta(\phi_{MRT\leq 3s}) \in F_{R2})$
R3	The system SHALL keep the rental fee AS CLOSE AS POSSIBLE to 0	$AG(\Delta(\phi_{RF}) \in F_{R3})$

case the requirement is fully met. The value decreases with increasing rental fees, i.e., higher values for $\Delta(\phi_{RF}, [i, j))$. The function has value 0 if $\Delta(\phi_{RF}, [i, j)) \geq RF_{max}$. That is, the requirement is not met if the rental fees reach RF_{max} or more.

Note, that concrete values for the response time threshold (T_{max}), the maximum time to re-establish the desired mean response time (D_{max}), and maximum accepted rental fee (RF_{max}) have to be specified by the requirements engineer. The software engineer has to take these thresholds into account for his design decisions, e.g., for determining a batch size for mean response time measurements B_{MRT}.

5.2 Simulation Results

We simulate our load balancer system, as modeled in Section 4, with SimuLizar and evaluate whether the requirements are met for a specific usage scenario. In our usage scenario, we have an open workload with an exponentially distributed interarrival time ($\lambda = 4\frac{1}{second}$) between user requests. That is, in the mean 4 user requests arrive in our system.

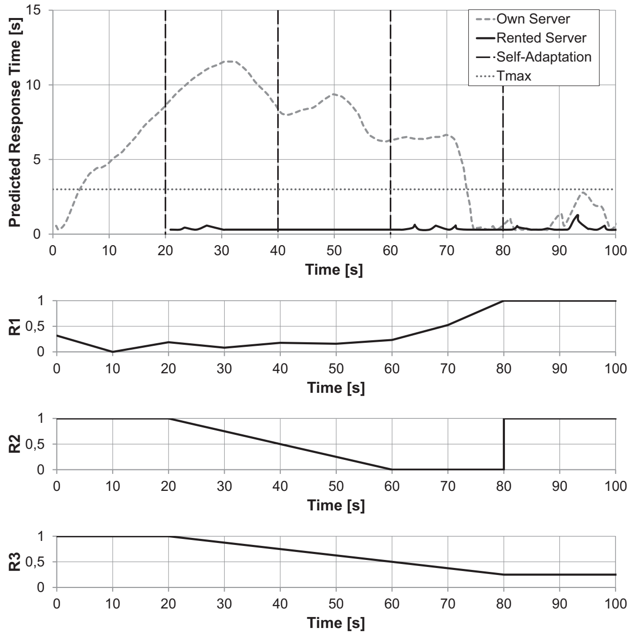
We already defined the mean response time threshold to be $T_{max} = 3$ *seconds* in Section 4. Additionally, we defined the maximum duration to re-establish the desired mean response time as $D_{max} = 40$ *seconds*. Designing the system, we had to make several design decisions. In this evaluation, we will specifically focus on the design decision regarding the batch size B_{MRT} for mean response time measurements. Our hypothesis is, that B_{MRT} has an influence on the performance of the system's self-adaptation.

5.2.1 Initial Design

Initially, we simulate the system for a batch size of 20 seconds, i.e., $B_{\text{MRT}} = 20$ *seconds*. Figure 5 shows the SimuLizar analysis result graphs.

The top most graph shows the (interpolated) time series of response times of our specified evaluation usage scenario. The time series shows that initially all requests are answered by server *own* (dashed grey curve). The simulation shows, that the response times increase steadily. Hence, server *own* is in a high load situation. Approximately after 20 seconds, the system self-adapts its load balancing strategy. Shortly after the self-adaptation the first requests are answered by server *ext* (solid black curve). The effect of the self-adaptation can be observed by the drop in the response times from application server *own* approximately at time 30. We can observe that the system triggers self-adaptation four times (dashed black vertical lines). Only after the four self-adaptations, approximately at time 75, the response times stabilize below the threshold of 3 seconds (dashed grey horizontal line).

Figure 5: Initial Design: Analysis for 20s Batches.



The three graphs below show the gradual compliance for each RELAX requirement $R1$, $R2$, and $R3$ at time 0, 10, 20, . . . , 100. The values correspond to the values of the according membership functions $m_{F_{R1}}$, $m_{F_{R2}}$, and $m_{F_{R3}}$ for the intervals $[i, i + 10)$. The values between two points in time are linearly interpolated. We can see in the graph for R1 that up to time 70 that the response time for less than 50% of the request is 3 seconds or less, i.e., the values are below 0.5. At time 10 the value is 0, i.e., R1 is not fulfilled. Interpreting the graph for R2, we see that the system detects that the mean response time is greater than 3 seconds, i.e., the values decrease linearly. After 40 seconds, at time 60, the value reaches 0. That is, R2 is not fulfilled at this time. In the graph for R3, we see that the value is linearly decreasing with every self-adaptation. This is the case, because with every self-adaptation the percentage of requests the load balancer delegates to the external server *ext* increases. We assume, the caused load and thereby the rental fees increase proportional accordingly in our evaluation. The maximum acceptable rental fee RF_{max} for our evaluation is less than 50% of the requests are delegated to server *ext*, i.e., $\alpha < 0.5$. We can see that this is always the case in the graph for R3, i.e., the value is always higher than 0. Hence, R3 is fulfilled.

Figure 6: First Design Iteration: Analysis for 5s Batches.

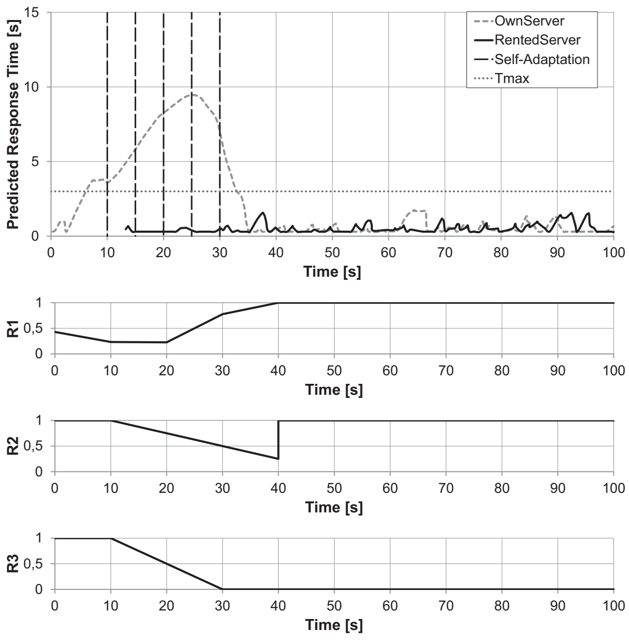
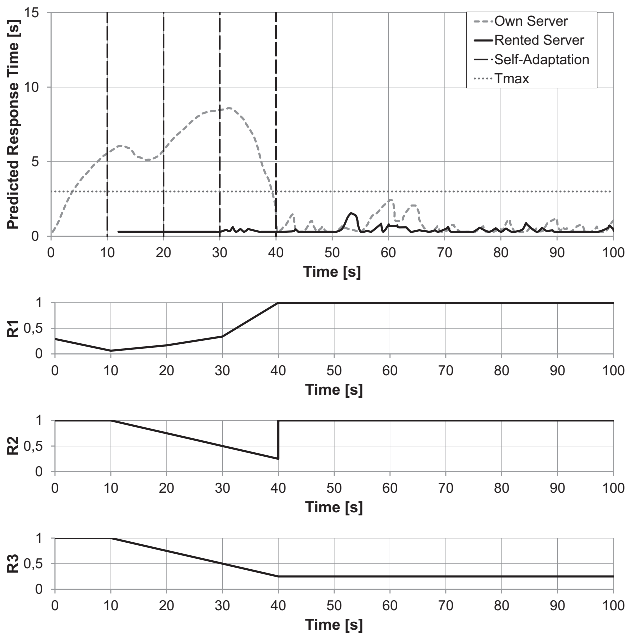


Figure 7: Analysis for 10s Batches.



In summary, the design did meet only R3 for the simulation and violated R1 and R2. The load balancer system was not able to maintain a response time of 3 seconds or less for at least some requests. Also, the system could not re-establish a mean response time of 3 seconds or less 40 seconds after it detected the violation. We conclude that the batch size B_{MRT} of 20 seconds is too large. Consequently the system's self-adaptation is triggered too late and not frequently enough.

5.2.2 First Design Iteration

In order to improve the system's self-adaptation design, we decrease the batch size to $B_{\text{MRT}} = 5$ *seconds*. We assume, that the system triggers self-adaptations earlier and more frequent. Figure 6 shows the SimuLizar analysis for the changed batch size. We can again observe a similar curve in the upper most graph. The time series shows increasing response times for server *own*. The load balancer detects at time 10 that the mean response time is higher than 3 seconds and triggers the first self-adaptation. Shortly after the first self-adaptation the first requests are served by the rented server *ext*. In total the system triggers five self-adaptations before the response times stabilize below 3 seconds.

Looking at the graph for R1, we can make two observations when comparing it to the R1 graph of the inital design. First, we can see that more requests served in 3 seconds or less is higher than in the initial design. Also, beginning at time 40 R1 is fully met, i.e., the value is 1. Furthermore, the load balancer is able to serve at least some requests within 3 seconds or less, i.e., R1 is fulfilled. The graph for R2 shows that the load balancer system detects at time 10 that the mean response time is greater than 3 seconds. The values are linearly decreasing until time 40, when the load balancer has re-established a mean response time of 3 seconds or less. Hence, R2 is also fulfilled. When we look at the graph for R3 we see the values decreasing with every self-adaptation, because the rental fees are increasing. At time 30 the value for R3 reaches 0, i.e., R3 is not fulfilled.

In summary, the first design iteration improved the overall requirement compliance. R1 and R2 are now met. However, R3 is not met using this self-adaptation design. We now assume, that the batch size B_{MRT} of 5 seconds is too small. As a consequence, the self-adaptations are triggered too frequently.

5.2.3 Second Design Iteration

Since the first iteration of our self-adaptation design was not sufficient, we slightly increase the batch size again to $B_{\text{MRT}} = 10 seconds$. The output of the SimuLizar analysis in Figure 7 shows the same characteristic curve in the time series for the response time. The load balancer detects that the mean response time threshold is exceeded at time 10. At this time the system triggers its first self-adaptation. In total, four self-adaptations are triggered, when finally approximately at time 40 the response times stabilize below 3 seconds.

When we examine the requirement compliance, we can see that less requests have a response time of 3 seconds or less compared with the previous self-adaptation design iteration. However, as the graph for R1 shows there are at least some requests with response times of 3 seconds or less. Hence, R1 is met. Looking at the R2 graph we can see, that the system detects that the mean response time threshold is exceeded at time 10. The system re-establishes a mean response time of 3 seconds or less again at time 40, i.e., in 30 seconds. R2 is also met. Finally, we see that with every self-adaptation the values in the R3 graph are decreasing. Since the self-adaptation is only triggered four times, R3 is also met.

In conclusion, we found a self-adaptation design that is able to fulfill all requirements with the last design iteration.

5.3 Limitations

SimuLizar as well as our evaluation still underlie several assumptions and limitations. First, since SimuLizar relies on simulation of single usage scenarios it cannot be considered as a sufficient validation of self-adaptation designs. Second, our evaluation is limited due to some threats to validity.

Self-adaptive behavior is usually triggered to cope with environmental changes, e.g., changing customer arrival rates. At the moment we can only simulate one specific environmental state, i.e., an arbitrary but fixed $e \in E$. However, in order to validate a self-adaptation design, one has to prove for all environmental states, that the self-adaptation is sufficient. Nevertheless, using SimuLizar we can identify unsatisfactory self-adaptation design, as we have shown in the previous subsection. Furthermore, the environment we can simulate is static, i.e., dynamically or randomly changing environmental context cannot be simulated. Thus, we currently simulate environmental states in which the condition of a self-adaptation rule holds from the start. This limits the scope of a simulation to only a set of rules which are triggered for the modeled static usage scenario.

We are currently making some assumptions in a SimuLizar simulation with respect to the performance of the self-adaptations. In the self-adaptive system models as well as in the simulation, we neglect resource consumption of self-adaptations for now. Furthermore, we do not handle exceptions that might occur during the self-adaptation.

Our evaluation is mainly limited due to the use of randomness in the example. The usage scenario we specified for our evaluation example contains an exponentially distributed arrival rate of the customers. Hence, the arrivals are random and are consequently not generalizable. The actual inter-arrival rate may vary in each simulation run.

6. RELATED WORK

In recent years, several related articles about self-adaptive system modeling and model-driven performance engineering have been published. We have surveyed them in [2].

The D-KLAPER approach [11] is a model-driven software performance engineering approach which can be applied to self-adaptive systems. D-KLAPER does not aim to provide modeling support for software design models. Instead software design models annotated with performance-relevant resource demands have to be transformed into D-KLAPER's intermediate language. The performance of models in this intermediate language can be analyzed with D-KLAPER. However, in contrast to our focus on analyzing requirements of the self-adaptation layer, D-KLAPER's focus on analyzing requirements of the business layer.

Huber et al. present a modeling-language for self-adaptive behavior in the context of software design models [12]. Their work is integrated within the project "Descartes", which envisions self-adaptive cloud systems enhanced with run-time performance analysis. The focus of the Descartes project is on run-time self-adaptation in contrast to our aims of design-time performance analysis of self-adaptation.

QoSMOS [7] is an approach to model and implement self-adaptive systems whose self-adaptation is driven by performance requirements. The system designer has to manually derive an analysis model from the architectural model. Manually derived analysis model serve as an initial input for the online performance analysis and its parameters are updated at run-time. In contrast, we provide a full tool chain for design-time performance analysis of self-adaptive systems that also able to adapt their structure.

Troya et al. present a model-driven approach for rule-based adaptive software systems in [21]. In such a system observers can be specified which observe non-functional properties of the system. These observers could also trigger adaptation rules, but the authors have not implemented this yet. The focus of this approach is more on real-time system in contrast to our focus on business information systems.

Souza et al. [20] present a framework for requirements-driven self-adaptive systems. Similar to our approach, the framework provides a quantification requirement satisfaction. Whereas our approach is to quantify the potential requirement satisfaction at design-time, the presented framework uses the requirement quantification at run-time to trigger self-adaptations.

In [10], Fleurey and Solberg propose a domain-specific modeling language and simulation tool for self-adaptive systems. With the provided modeling language self-adaptive systems can be specified in terms of variants. SimuLizar's focus is on the performance aspect of self-adaptation in contrast to the approach by Fleurey and Solberg which focuses on functional aspects.

Luckey et al. describe a UML-based modeling language named Adapt Case Modeling Language (ACML) [15] and a suitable quality assurance approach named QUality Assurance for Adaptive SYstems (QUAASY) [14, 16]. The ACML allows the specification of the self-adaptive system, its environment, and its adaptation rules. The ACML models are considered as a first operationalization of the system's requirements, e.g., described with RELAX. The language's core drivers are the separation of concerns, its analyzability, and its easy integration into existing engineering processes. The language's semantics are formally defined using graph transformation, allowing for static functional analysis within the QUAASY approach. Hidden from the user, QUAASY uses model checking techniques for show properties such as freedom of conflicts and deadlocks, termination, stability, and arbitrary application-specific safety and liveness properties. In contrast to the presented SimuLizar approach, QUAASY does not yet support the analysis of non-functional properties.

7. CONCLUSION

In this paper, we present an extension for our SimuLizar approach. The extension enables a semi-automatic analysis of performance requirements for self-adaptive systems. We provide a formalization for self-adaptive systems and their analysis concerning RELAX requirements. We show how SimuLizar can be used in conjunction with RELAX requirements. The gradual fulfillment of RELAX requirements can automatically be analyzed in a SimuLizar simulation.

Our evaluation shows that SimuLizar is applicable to identify unsatisfactory self-adaptation designs. A SimuLizar simulation enables to easily compare designs in terms of gradual requirement fulfillment. Thus, SimuLizar provides valuable feedback for software engineers and helps to iteratively improve the design of self-adaptive systems.

Future work is directed into two main directions. First, we are working towards further enhancements of the presented approach. Therefore, we plan to provide a domain-specific modeling language for specifying dynamic environments, i.e., changing usage scenarios during simulation. Further, we are currently implementing automatic generation of a performance prototype with self-adaptation capabilities from models specified with SimuLizar. Finally, we plan to enhance our tool to support developers modeling and evaluating self-adaptive systems, i.e., automatically detecting self-adaptation design flaws and providing design alternatives. The second main direction is the approach's embedding into an overall approach for the specification and analysis of both functional and non-functional properties of self-adaptive systems. Therefore, we are currently working on an integration of SimuLizar and the ACML/QUAASY approach that has been briefly presented in Section 6.

8. ACKNOWLEDGMENTS

This work is supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

9. REFERENCES

[1] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[2] M. Becker, M. Luckey, and S. Becker. Model-driven performance engineering of self-adaptive systems: A survey. In *Proc. of the 9th ACM SigSoft Int. Conf. on Quality of Software Architectures*, June 2012.

[3] M. Becker, J. Meyer, and S. Becker. SimuLizar: Design-Time Modeling and Performance Analysis of Self-Adaptive Systems. In *Proc. of the Software Engineering Conference (SE 2013)*, 2013 (to appear).

[4] S. Becker. Towards System Viewpoints to Specify Adaptation Models at Runtime. In *Proc. of the Software Engineering Conference, Young Researches Track (SE 2011)*, volume 31 of *Softwaretechnik-Trends*, 2011.

[5] S. Becker, T. Dencker, and J. Happe. Model-driven generation of performance prototypes. In S. Kounev, I. Gorton, and K. Sachs, editors, *Performance Evaluation: Metrics, Models and Benchmarks*, volume 5119 of *LNCS*, pages 79–98. Springer, 2008.

[6] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[7] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic QoS Management and Optimization in Service-Based Systems. *IEEE Trans. on Software Engineering*, 37:387–409, 2011.

[8] B. Cheng, R. d. Lemos, H. Giese, P. Inverardi, J. Magee, and et. al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In B. Cheng, R. d. Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 1–26. Springer, 2009.

[9] V. Cortellessa, A. Di Marco, and P. Inverardi. *Model-Based Software Performance Analysis.* Springer, 2011.

[10] F. Fleurey and A. Solberg. A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In *Model Driven Engineering Languages and Systems*, volume 5795 of *LNCS*, pages 606–621. Springer, 2009.

[11] V. Grassi, R. Mirandola, and E. Randazzo. Model-Driven Assessment of QoS-Aware Self-Adaptation. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 201–222. Springer, 2009.

[12] N. Huber, A. van Hoorn, A. Koziolek, F. Brosig, and S. Kounev. S/T/A: Meta-Modeling Run-Time Adaptation in Component-Based System Architectures. In *9th IEEE Int. Conf. on e-Business Engineering (ICEBE 2012)*, 2012.

[13] J. Kramer and J. Magee. Self-Managed systems: an architectural challenge. In *2007 Future of Software Engineering*, pages 259–268. IEEE Computer Society, 2007.

[14] M. Luckey, C. Gerth, C. Soltenborn, and G. Engels. QUAASY - QUality Assurance of Adaptive SYstems. In *ICAC'11*. ACM, 2011.

[15] M. Luckey, B. Nagel, C. Gerth, and G. Engels. Adapt cases: Extending use cases for adaptive systems. In *Proc. of the 6th Int. Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 30–39. ACM, 2011.

[16] M. Luckey, C. Thanos, C. Gerth, and G. Engels. Multi-staged quality assurance for self-adaptive systems. In *Proc. of 1st International Workshop on Evaluation for Self-Adaptive and Self-Organizing Systens at SASO'12 (to appear)*, 2012.

[17] S. Moon, K. Lee, and D. Lee. Fuzzy branching temporal logic. *IEEE Trans. on Systems, Man, and Cybernetics, Part B: Cybernetics*, 34(2):1045 – 1055, 2004.

[18] R. Murch. *Autonomic Computing.* IBM Press, 2004.

[19] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

[20] V. Souza, A. Lapouchnian, and J. Mylopoulos. Requirements-driven qualitative adaptation. In R. Meersman, H. Panetto, T. Dillon, S. Rinderle-Ma, P. Dadam, X. Zhou, S. Pearson, A. Ferscha, S. Bergamaschi, and I. Cruz, editors, *On the Move to Meaningful Internet Systems: OTM 2012*, volume 7565 of *LNCS*, pages 342–361. Springer, 2012.

[21] J. Troya, A. Vallecillo, F. Durán, and S. Zschaler. Model-driven performance analysis of rule-based domain specific visual models. *Information and Software Technology*, 55(1):88 – 110, 2013.

[22] M. von Detten, C. Heinzemann, M. C. Platenius, J. Rieke, D. Travkin, and S. Hildebrandt. Story Diagrams - Syntax and Semantics. Technical Report tr-ri-12-324, Software Engineering Group, Heinz Nixdorf Institute, July 2012. Ver. 0.2.

[23] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J. Bruel. RELAX: incorporating uncertainty into the specification of Self-Adaptive systems. In *Proc. of the 2009 17th IEEE Int. Requirements Engineering Conference, RE*, pages 79–88. IEEE, 2009.