SOCA (2014) 8:73–89
DOI 10.1007/s11761-013-0144-4

SPECIAL ISSUE PAPER

Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments

Nikolaus Huber · André van Hoorn · Anne Koziolek · Fabian Brosig · Samuel Kounev

Received: 14 December 2012 / Revised: 2 August 2013 / Accepted: 7 September 2013 / Published online: 22 September 2013

Abstract Today, software systems are more and more executed in dynamic, virtualized environments. These environments host diverse applications of different parties, sharing the underlying resources. The goal of this resource sharing is to utilize resources efficiently while ensuring that quality-of-service requirements are continuously satisfied. In such scenarios, complex adaptations to changes in the system environment are still largely performed manually by humans. Over the past decade, autonomic self-adaptation techniques aiming to minimize human intervention have become increasingly popular. However, given that adaptation processes are usually highly system-specific, it is a challenge to abstract from system details, enabling the reuse of adaptation strategies. In this paper, we present S/T/A, a modeling language to describe system adaptation processes at the system architecture level in a generic, human-understandable and reusable way. We apply our approach to multiple different realistic contexts (dynamic resource allocation, run-time adaptation planning, etc.). The results show how a holistic model-based approach can close the gap between complex manual adaptations and their autonomous execution.

This work was partly funded by the German Research Foundation (DFG) under Grant No. KO 34456-1.

N. Huber (✉) · F. Brosig · S. Kounev
Institute for Programme Structures and Data Organisation, Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
e-mail: nikolaus.huber@kit.edu

A. van Hoorn
Institute of Software Technology, University of Stuttgart, 70569 Stuttgart, Germany

A. Koziolek
Department of Informatics, University of Zurich, 8050 Zurich, Switzerland

Keywords Adaptation · Language · Run-time · Meta-model · Model-based

1 Introduction

Today's software systems are increasingly flexible and dynamic provisioning aims to react quickly on changes in the environment and to adapt the system configuration accordingly, in order to maintain the required quality-of-service (QoS). For example, the main reason for the increasing adoption of cloud computing is that it promises significant cost savings, by providing access to data center resources on demand over the network, in an elastic and cost-efficient manner. Industry's most common approach for automatic run-time adaptation of dynamic systems, such as in Amazon EC2 or Windows Azure, is using rule-based adaptation mechanisms. More complex adaptations like resource-efficient server consolidation are still largely performed manually. However, the increasing complexity of system adaptations and the rising frequency at which they are required render human intervention prohibitive and increase the need for automatic and autonomous approaches.

With the growth of autonomic computing and self-adaptive system engineering, many novel approaches address the challenge of building autonomic and self-adaptive systems with considerable success. However, such systems nowadays do not separate the software design and implementation from the system adaptation logic; i.e., they are typically based on highly system-specific adaptation techniques hard-coded in the system's implementation.

Hence, many researchers agree that a remaining major challenge in engineering self-adaptive systems is the development of novel modeling formalisms, allowing to describe and perform self-adaptation and reconfiguration in a generic,

human-understandable and reusable way [8]. To reduce the amount of human intervention required in run-time system adaptations, detailed models abstracting the system architecture, its execution environment, and its configuration space, as well as models describing the implemented adaptation processes are needed [4].

Models at run-time are a promising approach for run-time system adaptation [31], and self-adaptation approaches based on architectural models have been proposed before, e.g., [12,6,25]. However, such approaches concentrate on modeling the system's software architecture, and in general, knowledge about adapting the architecture is still captured within system-specific adaptation processes and not as part of the software architecture models. For example, Cheng et al. propose a language [5] with a programming language-like notation to specify self-adaptation processes [12]. However, the knowledge about the adaptation process is defined in a strictly deterministic and application-specific manner limiting flexibility.

In this paper, we present a domain-specific language called S/T/A (strategies/tactics/actions) to describe run-time system adaptation in component-based system architectures based on architecture-level system QoS models. The latter describes the QoS-relevant components of the system and reflects the QoS properties of the system architecture that must be taken into account when adapting the system at run-time. Architecture-level system QoS models include as part of them an adaptation space model that defines the space of valid system configurations, i.e., the boundaries within which run-time adaptations may be performed. We use the S/T/A meta-model presented in this paper on top of architecture-level system QoS models to describe system adaptation processes at the architecture level, in a generic, human-understandable, and reusable way. More specifically, we present how strategies and tactics can be used to guide the system adaptation process according to predefined objectives.

Our approach has several important benefits: First, it distinguishes high-level adaptation objectives (strategies) from low-level implementation details (adaptation tactics and actions), explicitly separating platform adaptation operations from system adaptation plans. We argue that separating these concerns has the benefit that system designers can describe their knowledge about system adaptation operations independently of how these operations are used in specific adaptation scenarios. Similarly, the knowledge of system administrators about how to adapt the system is captured in an intuitive and easy-to-use S/T/A model instance, as opposed to a system-specific adaptation language or process hard-coded in the system implementation. Second, given the fact that the knowledge about system adaptations is described using a meta-model with explicitly defined semantics, this knowledge is machine-processable and can thus be easily maintained and reused in common adaptation processes in dynamic systems like cloud environments. Furthermore, S/T/A helps to formalize adaptation processes, making it possible to analyze the adaptation scenarios for, e.g., QoS fulfillment or whether the adaptation actions can be executed at all.

The contributions of this paper are as follows: (1) An approach for separating system adaptation processes into technical and logical aspects by using architecture-level system QoS models to abstract technical aspects and our adaptation language to abstract logical aspects, (2) a general purpose meta-model (S/T/A) for describing the logical aspects of system adaptation processes in a generic way. The meta-model can be used to model (self-) adaptation either as simple workflows based on conditional expressions or as complex heuristics considering uncertainty. It provides a set of intuitive and easy to use concepts that can be employed by system architects and software developers to describe adaptation processes as part of system architecture models. (3) A basic concept for evaluating the effect of applied tactics based on collected QoS metrics. This concept supports an arbitrary amount of indeterminism to direct the adaptation process out of a local optimum. (4) An example prototype that interprets S/T/A model instances to adapt dynamic systems. It applies meta-modeling techniques end-to-end, i.e., from the system architecture up to the high-level system adaptation plans. (5) An evaluation of our approach in three representative scenarios each using a different type of architecture-level system QoS model. The evaluation shows how our adaptation language can be used for dynamic resource allocation, software architecture optimization and adaptation planning, demonstrating its general applicability, flexibility and usability at run-time. This paper extends our previous work [15] by (1) introducing a QoS data repository model to store measurement data obtained either as a result of model analysis or through monitoring of the system, (2) additional formal specifications of the semantics of the adaptation and a new weighting function to guide the adaptation based on a QoS data repository, (3) a description of the implementation of our prototypical framework to interpret S/T/A model instances and (4) an additional evaluation scenario targeting the weighting function concepts.

The rest of the paper is structured as follows: In Sect. 2, we present our modeling approach and the proposed adaptation language, illustrated with examples. In Sect. 3, we evaluate our approach in different representative application scenarios. Section 4 gives an overview of related work, and finally, Sect. 5 concludes the paper and outlines our planned future work.

2 Modeling system adaptation

In the context of dynamic system adaptation, we distinguish between a *technical view* and a *logical view* to separate the

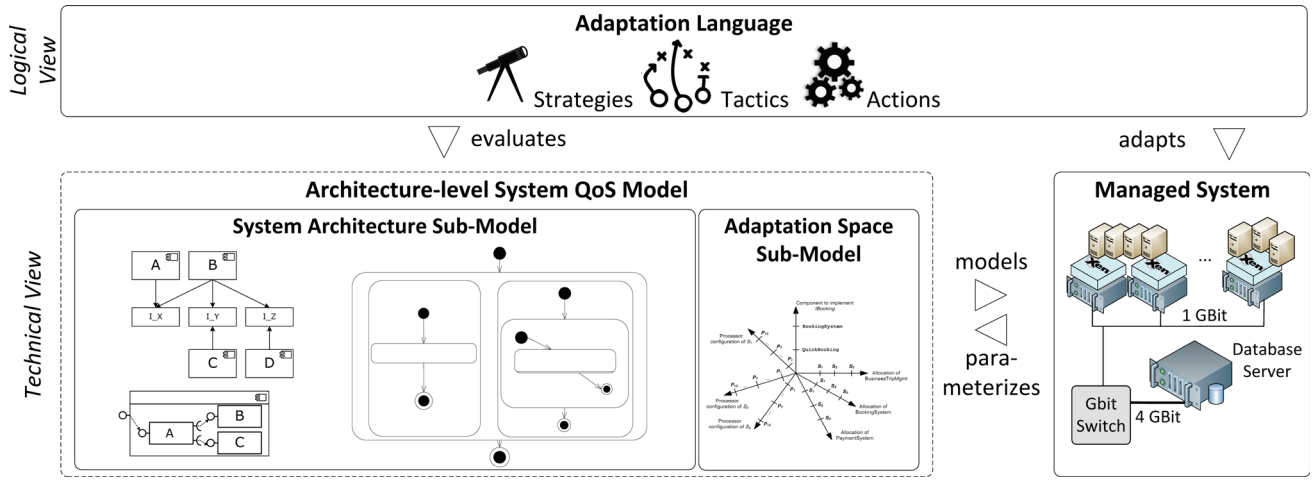

Fig. 1 Interaction of the system, the system models and the S/T/A adaptation language

knowledge of the system architecture from its adaptation processes. To further illustrate this idea, we distinguish two roles with different responsibilities, the *system designer* and the *system administrator*.

System designers design and implement the system; i.e., they have knowledge about the *technical* details of the system. They can describe the exact architecture of the system, what parts of the system can be adapted at run-time and by which specific actions. For example, in a virtualized environment, the system designer can specify which virtual machines (VMs) can be migrated or how many application server VMs can be added/removed from an application server cluster. We capture such information in architecture-level system QoS models which we will explain in more detail in Sect. 2.1.

System administrators are responsible to keep the system in the desired state; i.e., they design and implement the adaptation process that adapts the system when case problems occur. This role must not necessarily be a person, and it can be any instance responsible for controlling the system adaptation process, e.g., a rule-based engine. However, common for this role or instance is that it has the knowledge about the *logical* aspects of system adaptation processes. Such reconfiguration logic can be expressed in the form of simple rules or by complex heuristics or optimization algorithms usually being system-specific. By separating technical from logical aspects, the S/T/A reconfiguration language we present in Sect. 2.3 can abstract such information and is independent of system-specific details.

Figure 1 depicts the connection between the technical and logical view of the system and the models that capture the respective aspects. Section 2.1 explains the basic concepts of the technical view and the architecture-level system QoS model, which we use as foundation. Section 2.2 introduces an example we use throughout this paper to illustrate the concepts of the adaptation language meta-model presented in Sect. 2.3. Finally, in Sect. 2.4, we present a prototypical implementation of a framework that interprets S/T/A model instances and uses meta-modeling techniques to apply system adaptation.

2.1 Modeling system architecture and adaptation space

One important benefit of this approach and a major difference to others is the explicit separation of the architecture-level system QoS model into two separate sub-models, namely a system architecture sub-model and an adaptation space sub-model.

The *system architecture sub-model* reflects the system from the architectural point of view. Within adaptation processes, it can be used to analyze and evaluate QoS metrics for different configurations of the system; i.e., the model is typically used to predict the impact of possible system adaptations on the system's QoS properties. Examples of suitable architecture-level QoS models are the Palladio Component Model (PCM) [2] and the Descartes Meta-Model (DMM) [3,14,17] considered in this paper, or other component-based performance models as surveyed in [19]. These models have in common that they contain a detailed description of the system architecture in a component-oriented fashion, parameterized to explicitly capture the influences of the component execution context, e.g., the workload and the hardware environment.

The *adaptation space sub-model* describes the degrees of freedom of the system architecture in the context of the system architecture sub-model, i.e., the points where the system architecture can be adapted. Thereby, this sub-model reflects the boundaries of the system's configuration space; i.e., it defines the possible valid states of the system architecture.

The adaptation points at the model level correspond to adaptation operations executable on the real system at run-time, e.g., adding virtual CPUs to VMs, migrating VMs or software components, or load-balancing requests.

Examples of such sub-models are the Degree-of-Freedom Meta-Model [18] for PCM or the Adaptation Points Meta-Model as an integral part of DMM [14]. Having explicit adaptation space sub-models is essential to decouple the knowledge of the logical aspects of the system adaptation from technical aspects. One can specify adaptation options based on their knowledge of the system architecture and the adaptation actions they have implemented in a manner independent of the high-level adaptation processes and adaptation plans. Furthermore, by using an explicit adaptation space sub-model, adaptation is forced to stay within the boundaries specified by the model. The use of explicit adaptation space sub-models is an important distinction of our approach from other (self-)adaptive approaches based on architecture models [12,25]. Such approaches typically integrate the knowledge about the adaptation options and hence, the possible system states, in the operations and tactics, i.e., at the logical level.

Finally, it is important to mention that architecture-level system QoS models are capable of reflecting much more details of the data center environment and software architecture than classical system architecture models (e.g., as used by Garlan et al. [12]). The main resulting benefit is that we have more information about the system, thus being able to make better adaptation decisions and having more flexibility for adapting the model and the real system, respectively.

2.2 Running example: model-based dynamic resource allocation

In our previous work [13], we presented an algorithm for dynamic resource allocation in virtualized environments. This algorithm is implemented in Java. It is highly system-specific and therefore difficult to maintain and reuse. We use this algorithm throughout this paper as a running example to illustrate the concepts of our adaptation language in Sect. 2.3 and for the evaluation scenarios in Sect. 3.

The algorithm uses an architecture-level system QoS model we already developed and successfully applied in [13] for finding a system configuration that maintains given Service-Level Agreements (SLA) while using as little resources as possible. The algorithm consists of two phases, a PUSH and a PULL phase. The PUSH phase is triggered by SLA violations observed in the real system. The PULL phase is either triggered after the PUSH phase or by a scheduled resource optimization event. In the first step of the PUSH phase, the algorithm uses the system architecture sub-model to estimate how much additional resources are needed to maintain SLAs, based on the current system state. Then, it increases the resources in the model up to this estimation. These steps are repeated until the predicted QoS fulfills the SLAs. Resources can be increased by either adding a virtual CPU (vCPU) to a virtual machine (VM)—in case additional cores are available—or by starting additional VMs running application servers and adding them to the application server cluster. In the PULL phase, the algorithm uses the system architecture sub-model to estimate how much resources can be released without breaking SLAs. The amount of allocated resources is reduced stepwise by removing vCPUs from VMs and removing whole VMs from the application server cluster when their number of allocated vCPUs reaches zero. At each step, the system architecture sub-model is used to predict the effect of the system adaptation. If an SLA violation is predicted, the previous adaptation step is undone and the algorithm terminates. After the algorithm terminates successfully, the operations executed on the model are replayed on the real system. A more detailed description of this algorithm and its execution environment is given in [13].

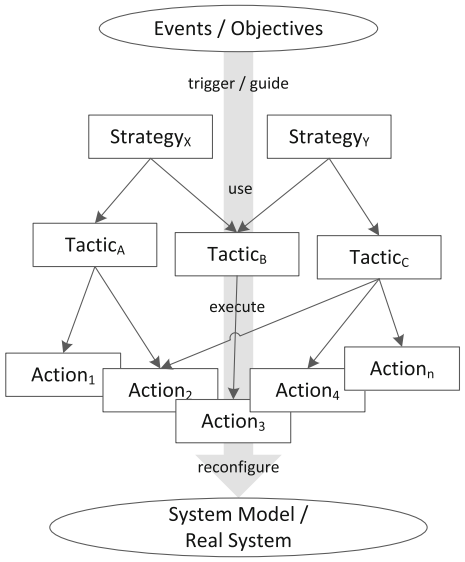

Fig. 2 Hierarchy of adaptation steps

2.3 S/T/A meta-model

Our S/T/A adaptation language consists of three major interacting concepts: *Strategy*, *Tactic* and *Action*. Each concept resides on a different level of abstraction of the adaptation steps as depicted in Fig. 2. At the top level are the strategies where each strategy aims to achieve a given high-level objective. A strategy uses one or more tactics to achieve its objective. Tactics execute actions, which implement the actual adaptation operations on the system model or on the real system, respectively. To evaluate the impact of applied tactics, we use a QoS data repository to collect model analysis results as well as monitoring data.

In this work, we use the terms strategy, tactic and action as follows. A strategy captures the *logical* aspect of planning an adaptation. A strategy defines the objective that needs to be accomplished and conveys an idea for achieving it. A strategy

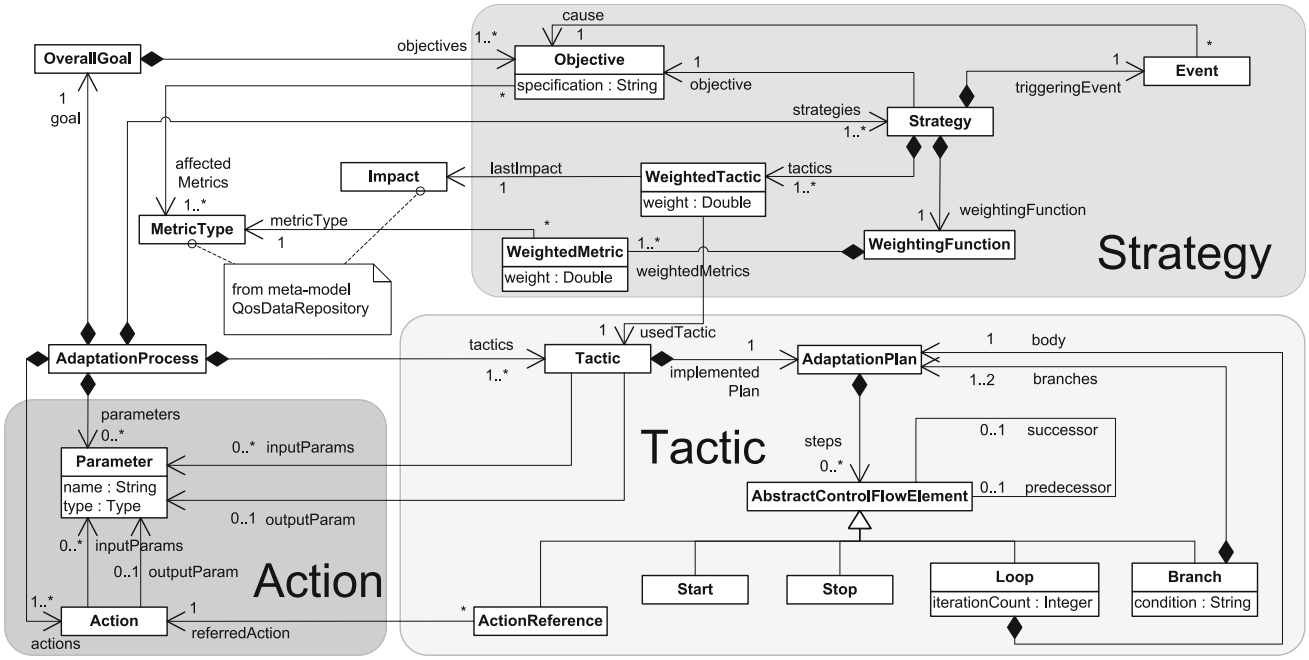

Fig. 3 Adaptation language meta-model

can be a complex, multi-layered plan for accomplishing the objective. However, which step is taken next will depend on the current state of the system. Thus, in the beginning, the sequence of applied tactics is unknown, allowing for flexibility to react in unforeseen situations. For example, a defensive strategy in the PUSH phase of our running example could be "add as few resources as possible stepwise until response time violations are resolved", whereas an aggressive strategy would be "add a large amount of resources in one step so that response time violations are eliminated, ignoring resource efficiency".

Tactics are the essential part of strategies. They are the *technical* aspect that follows the planning. Tactics are the part that actually execute the adaptation actions. Therefore, tactics specifically refer to actions. In the strategy phase of a plan, one thinks about how to act; i.e., one decides what tactics can be employed to fulfill the strategy's objective depending on the current system state. However, in contrast to strategies, tactics specify precisely which actions to take without explicitly considering their effect which is done at the strategy level. A possible tactic of adding resources in our running example could be "if possible, add another vCPU to a VM, otherwise, request another application server". An important property of tactics is their transaction-like semantic. We define tactics as: (1) atomic, i.e., either the whole tactic with all its contained actions is executed or the tactic must be rolled back, (2) consistent, i.e., the model's and the system's state must be consistent after applying a tactic, and (3) deterministic, i.e., tactics have the same output if applied on the same system state. This transaction-like behavior is important because after applying a tactic at the model level, the effect of the performed adaptation is evaluated by analyzing the QoS model; i.e., several actions can be executed at once without having to analyze the model after each action. This can save costly model analysis time which is crucial at runtime. Furthermore, applying tactics at the model level before applying them to the real system has the advantage that we can test their effect when applied as a whole without actually changing the system. Thereby, it is always possible to go back to the model state before starting to apply the tactic in case an error is detected, thus saving costly executions of roll-back operations on the system. The presence of a model to evaluate valid system states is also the reason why we decided not to use pre- or post-conditions to enforce certain restrictions.

The idea of distinguishing three abstraction levels is a valid concept and can be found in other approaches, too [6,16,20]. However, these approaches do either not consider a full model-based approach or have limited expressiveness (cf. Sect. 4). In contrast to the existing approaches, we propose a generic meta-model explicitly defining a set of modeling abstractions to describe strategies, tactics and actions with the flexibility to model the full spectrum of self-adaptive mechanisms. In the following, we describe the concepts of the proposed meta-model as depicted in Fig. 3. The meta-model and its implementation are available for download from our Web site [26].

All model elements described in the following are contained by the root element `AdaptationProcess`, representing a specific adaptation process.

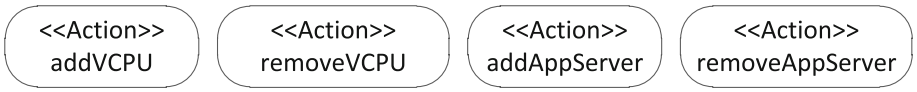

Fig. 4 Actions for the running example

2.3.1 Action

`Actions` are the atomic elements on the lowest level of the adaptation language's hierarchy. They execute an adaptation operation on the model or the real system, respectively. Actions can refer to `Parameter` entities specifying a set of input and output parameters. A parameter is specified by its `name` and `type`. Parameters can be used to customize the action, e.g., to specify the source and target of a migration action or use return values of executed actions as arguments for subsequent actions.

Example Figure 4 shows the four actions we modeled in our dynamic resource allocation algorithm. The actions `addVCPU` and `addAppServer` can be used to increase the resources used by the system, either by adding a vCPU to a VM (`addVCPU`) or by adding a new VM running an application server to the application server cluster (`addAppServer`). Similarly, `removeVCPU` and `removeAppServer` can be used to remove resources. Figure 4 depicts only the actions of our example. However, one can also specify actions for other resources, e.g., main memory (`increaseMemory` and `decreaseMemory`). The type of actions that can be modeled depends on the kind of adaptation points that are defined in the respective architecture-level system QoS model. The described actions do not implement the logic of the operation, and they are simply references to adaptation points defined in the respective architecture-level system QoS model. On the model level, actions are implemented by our prototypical framework discussed later in Sect. 2.4. On the system level, the virtualization or middleware layer executes the respective operations on the real system which could be triggered by our implementation, too.

2.3.2 Tactic

A `Tactic` specifies an `AdaptationPlan` with the purpose to adapt the system in a specific direction, e.g., to scale-up resources. The `AdaptationPlan` describes how the tactic pursues this purpose, i.e., in which order it applies actions to adapt the system. More specifically, each `AdaptationPlan` contains a set of `AbstractControlFlowElements`. The order of these control flow elements is determined by their predecessor/successor relations.

Implementations of the `AbstractControlFlowElement` are `Start` and `Stop` as well as `Loop` and `Branch`. The purpose of these control flow elements is to describe the control flow of the adaptation plan. For example, each `Branch` has an attribute `condition` which contains a condition directly influencing the control flow, e.g., by evaluating monitoring data, the system/model state or OCL expressions. Tactics can refer to `Parameter` entities to specify input or output parameters. These parameters can be evaluated to influence the control flow, e.g., by specifying iteration counts. Actions are integrated into the control flow by the `ActionReference` entity.

Example In Fig. 5, we show the three tactics specified for the running example based on the previously presented actions. These tactics are `addResources`, `removeResources` and `undoPreviousAction`. The first two tactics are used

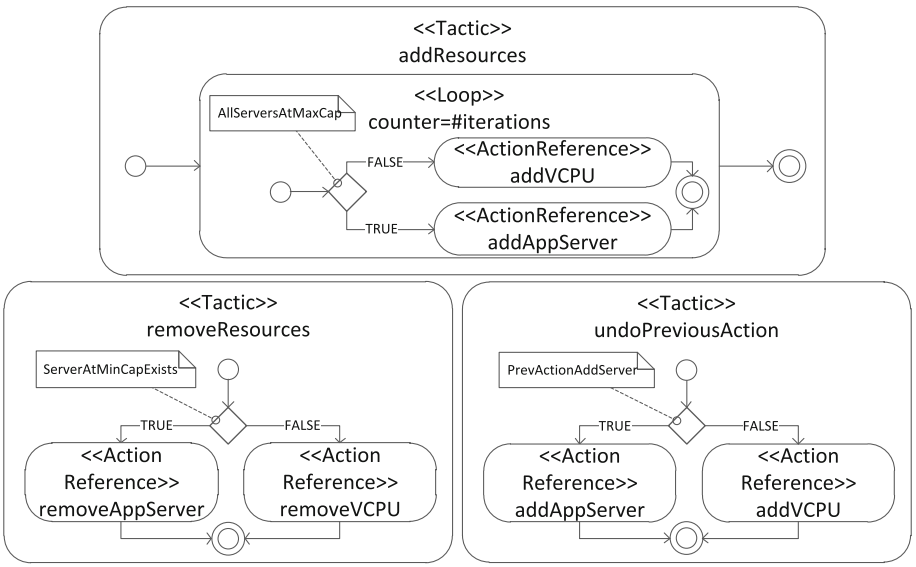

Fig. 5 Actions and tactics for the running example

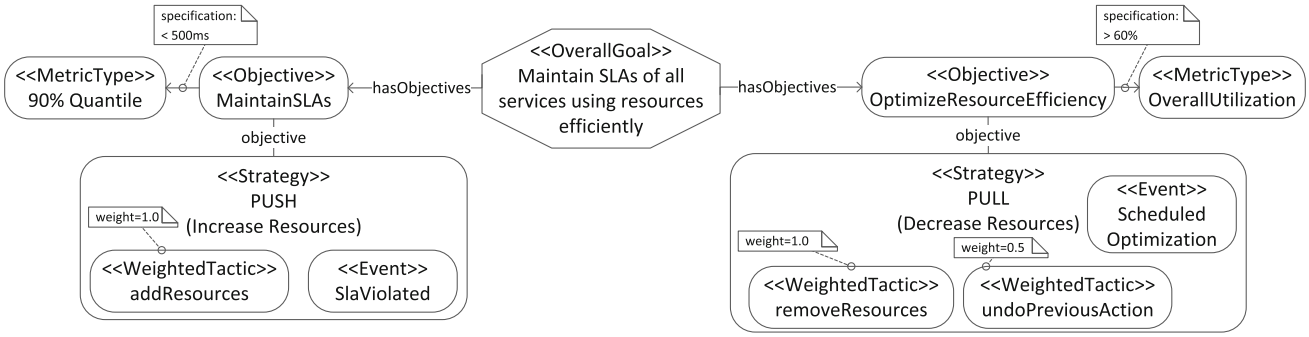

Fig. 6 Strategies using tactics and assigning weights

to scale the system up or down, and the third tactic can be applied to undo a previous action.

The adaptation plan of the tactic `addResources` implements a `Loop` action executed as many times as specified in `#iterations`, which is an input parameter to this tactic. With this parameter, one can specify how many resources should be added by executing the tactic. The adaptation plan in the `Loop` action implements two actions, `addVCPU` and `addAppServer`. Which action is executed depends on the current state of the architecture-level system QoS model. If there is no possibility to add a vCPU, i.e., $\forall s \in Servers : s.capacity = maxCapacity$ (`AllServersAtMaxCap`), an application server is added. Such constraints can be expressed on the model level in OCL. The adaptation plan of the tactic is an example for separating technical from logical details because the tactic specifies that resources should be added, but there is no need to specify how to implement this.

The adaptation plan of the tactic `removeResources` either removes an application server VM if there is a server running at minimum capacity, i.e., $\exists s \in Servers : s.cap = minCapacity$ (`ServerAtMinCapExists`), or removes a vCPU from an application server VM, otherwise. The `undoPreviousAction` tactic can be used in cases where the previous adaptation step must be undone. More details about the semantics of the tactics as part of adaptation strategies will be given in Sect. 2.4.

2.3.3 Strategy

Any adaptation process has an overall goal consisting of one or multiple different `Objectives`. The purpose of a `Strategy` is to achieve an `Objective`. An `Objective` refers to one or more `MetricTypes` and specifies the target range for this metric. All objectives are collected within the `OverallGoal`. The `OverallGoal` has no explicitly defined semantics, and it serves as a human-readable description of the overall goal of the adaptation process. The specification of an objective can be either simple predicates (e.g., avg. response time of $Service_x < 250$ ms) or more complex goal policies or utility functions referring to multiple metric types (e.g., resource usage vs. utilization). Note that it is explicitly allowed to have multiple alternative strategies with the same objective because strategies might differ in their implementation. One could also use any other type of description of the objectives that can be automatically checked by analyzing the model or monitoring data from the real system (e.g., using MAMBA [11]).

The execution of a strategy is triggered by a specific `Event` that occurs during system operation, e.g., a periodic event triggering resource efficiency strategies or an event caused when an `Objective` is violated. Such events trigger the execution of their parent strategy with the target to ensure that the objective of the strategy is achieved. In our approach, events can trigger only one strategy. We assume that events occur sequentially. This avoids concurrency effects, i.e., multiple strategies operating at the same time but with conflicting objectives. However, we do not exclude situations in which different objectives must be considered. Such cases can be handled by designing strategies with objectives expressed by a utility function [16]. The respective strategy in such a situation would try to apply tactics such that the objective of the utility function is achieved.

To achieve its objective, a strategy uses a set of `WeighedTactics`. A `WeightedTactic` is a decorator for a `Tactic`. Weights are assigned according to the strategy's `WeightingFunction`, which is explained in Sect. 2.3.5. The use of weighted tactics introduces a certain amount of indeterminism at this abstraction level. Having this indeterminism at the strategy level provides flexibility to find new solutions if a tactic turns out to be inappropriate for the current system state.

Example Figure 6 depicts the two strategies of the algorithm we described in our running example. These are the `PUSH` strategy with the objective to improve response times to maintain SLAs (90 % quantile of response time <500 ms) and the `PULL` strategy with the objective to ensure efficient resource usage (OverallUtilization >60 %). To specify these objectives on the model level, the `Objectives` refer

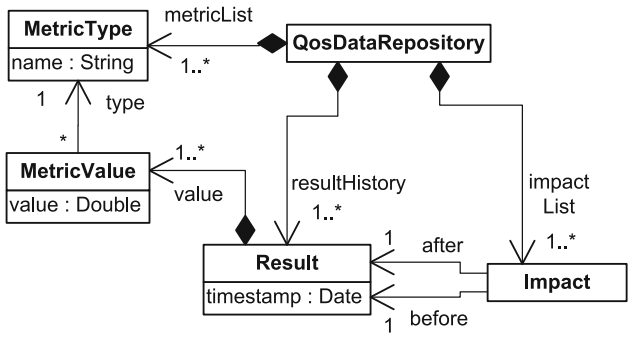

Fig. 7 The QoS data repository

to the respective `MetricType` from the QosDataRepository. The PUSH strategy uses only one tactic, namely `addResources`, and is triggered by the `SlaViolated` event. After the tactic has been successfully applied at the model level, the architecture-level system QoS model is analyzed to predict the impact on the `MetricType` referred by the objective. If the prediction results still reveal SLA violations, the strategy executes the tactic again until all SLA violations are resolved and the strategy has reached its objective.

The PULL strategy is triggered with the objective to `OptimizeResourceEfficiency`. The trigger of the related event could be either a predefined schedule or even the PUSH phase. The PULL strategy refers to the tactic `removeResources` to reduce the amount of used resources. Again, after the execution of the tactic, the model is analyzed to predict the effect of the tactic on the system performance. If no SLA violation is detected, the strategy can continue removing resources. In case an SLA violation occurs, the last adaptation must be undone, which is implemented by the `undoPreviousAction` tactic. Which of these two tactics is chosen is determined based on the weights. In our example, the initial values are 1.0 for `removeResources` and 0.5 for `undoPreviousAction`. The concepts of the weighting function will be presented with more details in Sect. 2.3.5.

2.3.4 QoS data repository

The evaluation of the effect of an executed tactic needs to quantify the impact of the actions applied by the tactic. This impact can be determined by comparing QoS metrics (either on the model or system level) before and after applying a tactic. This information is stored in a `QosDataRepository` (see Fig. 7). The repository contains a set of `MetricTypes` M that can be monitored at the model and system level, respectively. Examples for such metric types could be: m_1 as the average response time of $Service_X$, m_2 as the 90% quantile of response time of $Service_Y$, or m_3 as the average utilization of $Resource_n$.

The repository also contains a history of `Results`. A `Result` is a set of collected `MetricValues` at the `timestamp` ts. A `MetricValue` specifies the actual value $v \in V$ of a `MetricType` $m \in M$ observed at the given time stamp $ts \in TS$. More formally, the set of values V is defined as $V \in [M \times TS \to \mathbb{R}]$. An example could be $v(m_1, ts_i) = 0.5\,\text{s}$, specifying that the value v of metric type m_1 at time stamp ts_i was 500 ms.

Information also stored in the QoS data repository is the achieved `Impact` of a tactic. The impact is quantified by calculating the delta of the results `after` and `before` the application of a tactic for each `MetricValue`. More formally, the impact on the metric types $m \in M$ is specified as

$$i_t(m) = v(m, ts_i) - v(m, ts_{i-1}), \tag{1}$$

where ts_i and ts_{i-1} identify the result with the time stamp `after` and `before` applying the tactic. For example, if the metric value $v(m, ts_i)$ of metric m at time stamp ts_i was 500 ms and is 200 ms at time stamp ts_{i-1}, the impact $i_t(m)$ of tactic t on metric m would be -300 ms, i.e., an improvement of the response time metric.

Our intention was not to define a new meta-model for QoS metrics and values. Thus, we kept this meta-model very basic to adapt and reuse it in other scenarios. For example, this meta-model can serve as a decorator model for the Software Metrics Meta-Model (SMM), developed by the Architecture-Driven Modernization Task Force of the OMG [24]. Thereby, it is easier to reuse other existing tools based on SMM, e.g., the MAMBA Execution Engine or Query Language [11]. When decorating SMM, the class `MetricType` would refer to the `ObservedMeasure` of the SMM, a `MetricValue` corresponds to the `Measurement` of the SMM, and `Result` corresponds to `Observation`. In this way, it would be possible to use data that have already been collected and stored in a repository by measurement tools like Kieker [29].

2.3.5 Weighting function

A strategy uses weights to determine which tactic to apply next. A tactic's weight is calculated directly after executing the tactic, so the strategy might choose a different tactic in the next adaptation step. The weight (or value) of a tactic depends on the achieved impact, i.e., if the metrics of interest have been improved or degraded. This calculation of weights is realized by the `WeightingFunction`. Formally, a weighting function can be specified as follows. Assume

$\text{T} = \{t_1, t_2, \ldots, t_m\}$ is the set of tactics and
$\text{S} = \{s_1, s_2, \ldots, s_n\}$ is the set of all possible system states.

Then, the weighting function can be specified as a mapping $W \in [T \times S \rightarrow \mathbb{R}]$ that assigns a weight to the given tactic $t \in T$, given the system state $s \in S$. The idea is that any existing and well-established optimization algorithms or meta-heuristics (like Tabu Search or Simulated Annealing) can be used here to determine the weights depending on the current state of the system, possibly also considering its previous states stored in a trace.

In our approach, we use a basic concept to specify weighting functions, which could be extended in the future. Basically, a `WeightingFunction` consists of several `WeightedMetrics`. A `WeightedMetric` assigns a specific `weight` to the referenced `MetricType`. As explained in the previous section, `MetricTypes` could be, e.g., the response time of services or the utilization of resources. To calculate the new weight for the applied `Tactic` t, we take the achieved `Impact` of each metric $m \in M_t$ and multiply it with its assigned weight (M_t is the set of all `MetricTypes` affected by t). More formally, the new weight for a `Tactic` t is calculated as

$$wf(t) = \sum_{m \in M_t} (w_m * i_t(m)), \tag{2}$$

where w_m is the weight assigned to `MetricType` $m \in M_t$, and $i_t(m)$ is the impact of tactic t on metric m. In other words, the weights of the set of metrics affected by the applied tactic influence the weight assigned to the tactic. This way one can, e.g., prioritize the impact on the response time over the impact on the utilization.

The reasons why we use this basic concepts are that it is machine-processable and that it directly relates to our specification of `Objectives`. `Objectives` refer to a `MetricType`, and the `WeightingFunction` can specify the influence of this metric on the application of tactics.

Example Imagine a situation in which we want to prioritize tactics that have a beneficial impact on important services. Therefore, assume that we have two different services, one of a gold customer ($service_g$) and of a silver customer ($service_s$). For each of these services, we observe the metric types *90% quantile of the response time*, named rt_g and rt_s. To prioritize the impact on the response time of the gold customer's service over the one of the silver customer, we set the weighted metrics to $w_{mg} = -2.0$ and $w_{ms} = -1.0$. Note that the weights are negative because improving the response time results in a negative impact. To assure that a tactic which is beneficial for the gold service gets a higher weight, we can specify a weighting function $wf(t) := w_{mg} * i_t(rt_g) + w_{ms} * i_t(rt_s)$ that assigns weights depending on the impact on the response times of the gold and silver customer, respectively.

Another example is the weighting function we specified for the `PULL` strategy to assign new weights to the two tactics `removeResources` and `undoPreviousAction`,

$$wf_{PULL}(t) := \begin{cases} 1, & \text{if } rt < \tau \\ 0, & \text{else.} \end{cases}$$

This step function assigns a weight of one to the given tactic t as long as the response time rt is below the threshold τ; i.e., the SLA is not violated. Only if an SLA is violated, the weight of the given tactic is changed to zero. As the `undoPreviousAction` has an initial weight of 0.5 and the `removeResources` an initial weight of 1.0, this assures that the tactic `removeResources` will be executed until an SLA is violated. Then, in the next adaptation step, the tactic `undoPreviousAction` will be applied, because its weight is greater than zero. A more detailed weighting function and its application are also shown in the evaluation in Sect. 3.2.

2.4 Prototypical S/T/A framework

To evaluate the usability and benefits of our adaptation language, we provide a framework with the purpose to interpret S/T/A model instance and execute them on the model level, or—if a suitable connector is implemented—on the real system. This section provides a brief overview of our framework. We use the running example to illustrate the execution of the modeled adaptation process. The prototypical implementation, a description of how to install the framework, and our example model can be found on the Web site [26].

In the *setup phase*, the framework reads the models and their initial settings into memory. It establishes a connection to the deployed system configuration and registers with the QoS data repository to retrieve information about the real system and its QoS metrics. It also starts an `AdaptationController` which waits for events to perform system adaptation. When receiving an event, the `AdaptationController` starts the system adaptation. We distinguish the *model adaptation phase* and the *system adaptation phase*.

The next phase is the model adaptation phase, and it starts when the `AdaptationController` receives an event to a certain objective and triggers the corresponding strategy. For example, assume that a `ScheduledOptimization` event occurs. The `AdaptationController` receives this event and triggers the corresponding `PULL` strategy with the objective to optimize resource efficiency (see Fig. 6). The `AdaptationController` then selects one of its referred tactics based on the current weights of the tactics (in the initial case, this is `removeResources`) and executes the tactic's adaptation plan. According to the adaptation plan of `removeResources`, the `AdaptationController`

first checks the OCL constraint `ServerAtMinCapExists`. Assume that the expression evaluates to `TRUE`, and hence, the `AdaptationController` executes the `removeAppServer` action. It looks up the action's corresponding adaptation point in the adaptation space sub-model and uses this information to adapt the model accordingly. For `removeAppServer`, this would contain a pointer to the architecture model elements that can be changed and constraints ensuring that the model is still valid after adaptation (e.g., there must be at least one server left). When all actions are executed, the execution of the tactic is finished and the control flow returns to the strategy to evaluate the impact of the tactic. For this evaluation, the `AdaptationController` may use a simulator or another analysis method that can predict the QoS metrics of the modified architecture-level system QoS model. The `AdaptationController` then evaluates the results using the weighting function to change the weights of the tactics if necessary. In our example, if the current configuration causes an SLA violation, the weighting function would assign 0.0 to the `removeResources` tactic. The next adaptation step would choose the `undoPreviousAction` tactic with weight 0.5 because the previous tactic was not successful. After evaluating the impact of the applied tactic and assigning the new weight, the `AdaptationController` checks whether the problem which caused the event has been solved. To do this, the `AdaptationController` checks whether the strategy's objective that has been referred by the event is fulfilled. If false, the phase starts over, executing the adaptation plan of the tactic that now has the highest weight until the objective is fulfilled or the application of tactics has no further impact. The output of this phase is an adapted architecture-level system QoS model instance that fulfills the modeled overall goal. Changes are, e.g., the number of servers or the amount of used resources.

In the system adaptation phase, we replay the model adaptation phase on the real system, however, without considering adaptation steps that have been discarded in the previous phase. For our experiments, we implemented a system adaptation phase that can apply changes to the virtualized environment of the actual system using the Xen API. This must be extended or replaced if other adaptation actions or different target platforms should be supported.

3 Evaluation

We evaluate our presented adaptation language in several distinct representative scenarios to demonstrate that it provides a generic and flexible formalism for modeling system adaptation based on different architecture-level system QoS models. The first and third scenario (Sects. 3.1, 3.3) demonstrate the generality and flexibility of our S/T/A adaptation language by applying it in the context of dynamic resource allocation and run-time capacity management with different architecture-level system QoS models. The second scenario (Sect. 3.2) demonstrates the applicability of a weighting function to guide the adaptation process by changing to different tactics. The third scenario (Sect. 3.3) demonstrates that the S/T/A adaptation language can produce useful solutions in a reasonable amount of time by evaluating its usability in a framework for multi-objective software architecture optimization. The fourth scenario (Sect. 3.4) gives an example for decoupling the adaptation process by using S/T/A model instances as adaptation plans. In the last Sect. 3.5, we compare the concepts of our adaptation language with other approaches and discuss the challenges of a technical comparison of these approaches.

3.1 Dynamic resource allocation

In this scenario, we evaluate our approach using the SPECjEnterprise2010[1] benchmark. We first briefly describe the experimental setup the benchmark is deployed in and discuss the applicability of our architecture-level system QoS model before we apply the dynamic resource allocation process of our running example, summarized in Sect. 2.2. The S/T/A model instance of this adaptation process can be obtained from our Web site [26].

As hardware environment for the experiments, we use six blade servers from a cluster environment. Each server is equipped with two Intel Xeon E5430 4-core CPUs running at 2.66 GHz and 32 GB of main memory. The machines are connected by a 1 GBit LAN. On top of each machine, we run Citrix XenServer 5.5 as the virtualization layer. The SPECjEnterprise2010benchmark application is deployed in a cluster of Oracle WebLogic Server (WLS) nodes. Each WLS node runs in its own XenServer VM, initially equipped with two virtual CPUs (vCPUs).

For the evaluation, we considered reconfiguration options concerning the number of WLS cluster nodes and the vCPUs the VMs are equipped with: WLS nodes are added to or removed from the WLS cluster, and vCPUs are added to or removed from a VM.

The benchmark application and the hardware environment have been modeled with PCM [2] as architecture-level system QoS model. We used SimuCom[2] to analyze the model and predict the impact of changes to the model. We then replay the changes on the real system to check whether they

[1] SPECjEnterprise2010 is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjEnterprise2010 results or findings in this publication have not been reviewed or accepted by SPEC; therefore, no comparison nor performance inference can be made against any published SPEC result. The official Web site for SPECjEnterprise2010 is located at http://www.spec.org/jEnterprise2010.

[2] http://www.palladio-simulator.org/.

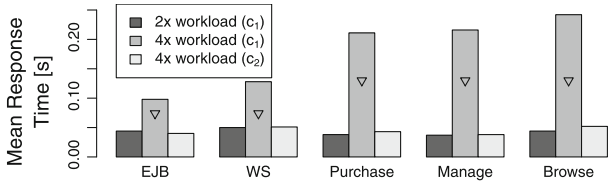

Fig. 8 The response times when changing workload from ×2 to ×4 (SLAs denoted by *inverted triangle*)

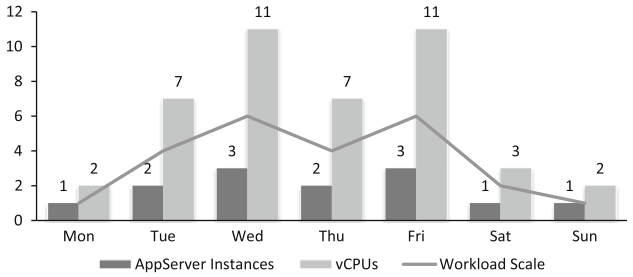

Fig. 9 Dynamic resource allocations using the modeled algorithm from the running example

have the desired impact. As an example, Fig. 8 shows the response time measurement for all five services of the benchmark for an increasing workload. The figure shows that system configuration c_1 is not able to handle the increased load (4×) without SLA violations. Hence, our adaptation process is triggered, resulting in a new system configuration c_2, where all SLAs are maintained.

Figure 9 shows the results of a series of such workload changes and the resulting changes to the model. The chart shows how the number of application servers and the allocated vCPUs change in the real system as the workload changes during system operation over a period of one week. This demonstrates how we can apply the adaptation language using an architecture-level system QoS model to dynamically allocate resources at run-time in a virtualized system. The advantage is that the hard-coded logic of the algorithm is now encoded in a generic model which is intuitive for software architects and can be easily maintained, modified or reused. Since we abstract the actual changes applied to the model and the real system as actions, the modeled dynamic resource allocation algorithm can also be reused in a different virtualized environment, e.g., based on a different virtualization platform.

3.2 Guiding adaptation using a weighting function

The increase of resources explained in the previous section can be achieved with the PUSH strategy, implementing two different tactics. Tactic t_1 is an aggressive but more costly tactic, which adds more resources by starting new VMs, first. In contrast, tactic t_2 is more conservative, assigning additional vCPUs to the existing VMs before starting additional VMs.

Table 1 Application of the example weighting function

	Tactic t_1			Tactic t_2		
	m_{ws}	m_p	m_b	m_{ws}	m_p	m_b
Before	762	221	217	762	221	217
After	376	119	79	564	174	148
Impact	−386	−102	−138	−198	−47	−69
Weight	−2.0	−2.0	−1.0	−2.0	−2.0	−1.0
Result	1,114			559		

All metric values are in milliseconds

This section describes the implementation of a weighting function to determine which tactic to apply next (see Sect. 2.3.5). We will use simulation results of our model to demonstrate the effect of our weighting function. Assume we observe the metric types $m \in \{m_{ws}, m_p, m_b\}$, defined as the 90 % quantile of the response time of the services $S = \{WS, Purchase, Browse\}$. Services WS and *Purchase* are the services of a gold customer, and *Browse* the service of a silver customer. Therefore, we set the weights w_m for each service's metric to $\{-2.0, -2.0, -1.0\}$ (see Table 1). Note that the weights for the response time metrics are negative because the impact of a response time improvement is negative, too. This assures that the absolute value $(w_m * i_t(m))$ is positive for positive changes in the affected metric values and negative for undesired changes. The setting of the weights also guarantees that a response time improvement of the gold customer is more valuable compared to the silver customer.

Table 1 shows the measured metrics for a given system configuration *before* (two VMs using two vCPUs each) and *after* applying either the aggressive (one additional VM) or conservative (one additional vCPU) tactic. The table also shows the impact of the tactics as well as the calculated weighting function result. The result indicates that tactic t_1 achieves a higher result because it improved the response times more significantly.

Now assume that the adaptation process in the initial state and the weights assigned to (t_1, t_2) are (750, 1,000). The aggressive tactic has a lower weight because it is more costly to add new VMs. Hence, the adaptation strategy would choose to execute t_2. However, after applying t_2, the new weight of t_2 is 559. In case the SLAs are still violated and a further adaptation step is required, the PUSH strategy will apply the more aggressive strategy t_1 because it has a higher weight.

This example shows how it is possible to design strategies based on the same tactics but with different goals using different weights and weighting functions. If the system administrator's goal is to maintain SLAs at all cost, they would assign higher weights to the aggressive tactic and implement a weighting function that prioritizes the application of this tactic.

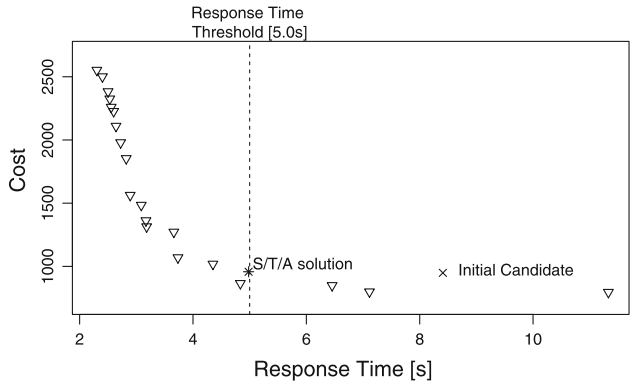

Fig. 10 Pareto optimal candidates found by PerOpteryx (*inverted triangle*) and candidates found when S/T/A is applied to guide the search process (*asterisk*)

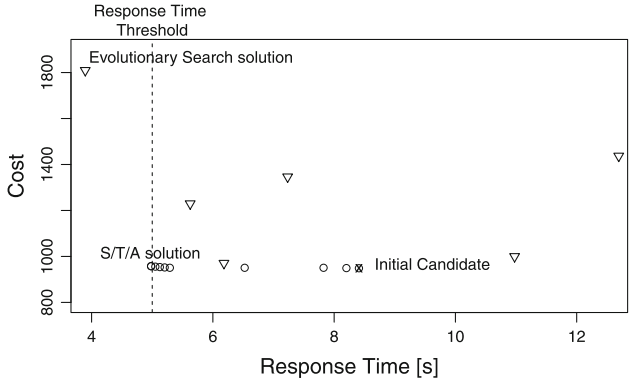

Fig. 11 Comparison of the PerOpteryx configuration search with (*open circle*) and without S/T/A (*inverted triangle*)

3.3 Simulating S/T/A adaptation

In this scenario, we integrate our approach into a framework for improving software architectures by automatically trading-off different design decisions, called PerOpteryx [21]. Thereby, we show that our approach can find a suitable system configuration within a reasonable amount of time. This scenario also shows that our adaptation language can be applied to a different architecture-level system QoS model (e.g., PCM [2]).

For a given architecture-level system QoS model, PerOpteryx searches the space of possible configurations for candidates that fit given objectives. Therefore, PerOpteryx starts from an initial system configuration modeled with PCM and generates new candidates according to the adaptation space sub-model. These new candidates are evaluated w.r.t. the given objectives, and a new iteration starts with half of the best fitting candidates.

PerOpteryx is targeted at design-time optimization; i.e., there are no strict time constraints to evaluate a huge number of candidates. Although PerOpteryx implements heuristics encapsulating domain knowledge to reduce the number of candidates that are evaluated, the PerOpteryx approach with its implemented genetic algorithms is not designed for use at run-time.

In this scenario, we implemented a strategy and a set of tactics to focus the generation of candidates (i.e., system configurations) within PerOpteryx to find a candidate that fulfills the objective of the strategy as quickly as possible. For our experiments, we use the same models and settings as in the Business Reporting System case study used to evaluate PerOpteryx [21]. Figure 10 shows the output after 100 iterations with a population of 60. It depicts the set of Pareto optimal candidates (marked by $\triangledown$) when trading-off response time (in seconds) versus cost (an abstract unit used by PerOpteryx). We use this Pareto optimal set of candidates as the baseline for the following accuracy and efficiency evaluation of our approach. In our scenario, we assume that the response time of the initial candidate (8.4 s) violates an SLA which must guarantee response times below five seconds. This triggers our implemented strategy with the objective to adapt the system such that the response time is below five seconds. The strategy chooses a tactic from the set of tactics $T = \{$`IncreaseResource`, `LoopIncreaseResource`, `BalanceLoad`$\}$. Due to space constraints, we omit a detailed depiction of the respective S/T/A model. The `IncreaseResource` tactic implements one action, increasing the CPU capacity of the server with the highest utilization by 10 % w.r.t. its initial capacity. `LoopIncreaseResource` implements the same action but within a loop action repeating the *increaseCPU* action as often as specified by the loops `counter` parameter. `BalanceLoad` migrates a software component from the server with the highest utilization to the server with the lowest.

For the initial system state $s_0 \in S$, we set the weights for the three tactics to $wt_0 = (1.0, 0.0, 0.0)$. We then use the PerOpteryx framework to execute our strategy. The strategy chooses a tactic, executes it and evaluates the resulting candidate. The evaluation reassigns weights to the tactics according to their effect. In this scenario, if `IncreaseResource` has no effect (response time increase $\leq$5 %), we assign 1.0 to `LoopIncreaseResources`. If there is still no effect, we assign 1.0 to `BalanceLoad`. In case we detect an effect, we reset the initial weights. We repeat applying a new tactic according to the weights and evaluating the results until we find a state that fulfills the objective of the strategy. The final resulting candidate, i.e., system configuration, of this process is depicted by a $*$ symbol in Fig. 10.

As we can see, the configuration found using the S/T/A model is not globally optimal. However, it fulfills the given target and was found within eight iterations. The standard evolutionary search of PerOpteryx provides the first SLA-fulfilling candidate after ten iterations (see Fig. 11). Thus,

using S/T/A models can provide a quicker process to find a close to optimal solution than a full-fledged optimization approach. Of course, this comes at the cost that the S/T/A solution might not be optimal. Furthermore, five out of the eight iterations using an S/T/A model executed the `LoopIncreaseResource` tactic which executes two nested adaptation actions. This results in saving five (out of thirteen) model evaluations that would have been necessary without S/T/A, thereby saving costly analysis time. Note that the absolute amount of saved time and the duration of the evaluation depend on the input model and the employed analysis technique.

We emphasize that the contribution of this paper is not a new optimization algorithm. Instead, our goal in this scenario is to focus system optimization such that a system configuration that fulfills a given set of objectives is found as quickly as possible. This system configuration must not necessarily be globally optimal.

However, the evaluation results demonstrate that with the knowledge about the system adaptation strategies modeled using our adaptation language, it is possible to find suitable candidates within a shorter amount of time reasonable for run-time system adaptation.

3.4 S/T/A adaptation plans

This section demonstrates how we use the S/T/A approach in the SLAstic framework for architecture-based online capacity management [28,30]. SLAstic aims to increase the resource efficiency of distributed component-based software systems employing architectural run-time adaptations. SLAstic also relies on architectural models describing a system's QoS-relevant aspects and adaptation capabilities. SLAstic's purpose is to determine required adaptations proactively, in order to calculate and execute appropriate adaptation plans. In this scenario, we show that our S/T/A language can be used to specify and execute architectural adaptation plans, in order to bring a real or simulated system from a current to a desired configuration. This section presents some of our results of a laboratory experiment, employing SLAstic to control the capacity of a software system deployed to an Eucalyptus-based IaaS cloud environment, which is compatible with the Amazon Web Services (AWS) API.

The five S/T/A actions depicted in Fig. 12 correspond to the set of architectural run-time adaptation operations currently supported by the SLAstic framework: *allocate* and *deallocate* (typed) execution containers (i.e., physical or virtual servers), as well as *migrate*, *replicate* and *dereplicate*, a given software component to or, respectively, from a given execution container. Input parameters refer to types from the SLAstic meta-model. Note that these actions are the same regardless of whether SLAstic is connected to an IaaS environment or, for example, to a simulator for run-time adaptable PCM instances [30]. In the cloud scenario, an adaptation manager receives precalculated S/T/A `AdaptationPlans` and executes them by interacting with the AWS API and the allocated nodes according to the actions specified in Fig. 12.

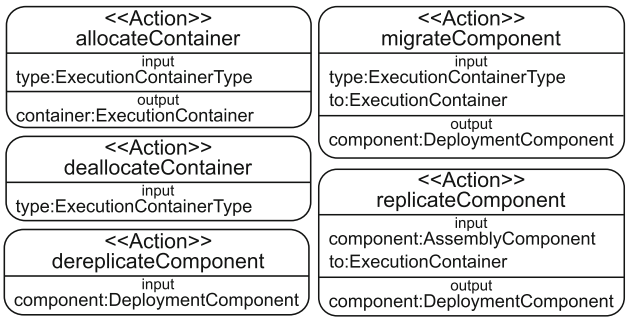

Fig. 12 Actions of the online capacity management scenario

In our evaluation, we expose a Java-based application (JPetStore 5.0) to a probabilistic workload with varying intensity based on a 24 h workload profile obtained from an industrial system. The profile was scaled to an experiment duration of 24 min plus 2 min cooldown. In this setting, we made the assumption that we have a good understanding of the correlation between application-level workload intensity—in this case, the number of requests to a software component per minute—and the CPU utilization. For each software component, we defined a rule set specifying the number of component instances to be provided at certain workload intensity levels, e.g., five instances in periods with a workload intensity of 27,000 requests per minute. Deviations between the number of component instances specified in the rule set and the number of instances actually allocated trigger the adaptation planner to create an S/T/A `AdaptationPlan` with the goal to achieve the requested architectural configuration. This plan is then sent to the adaptation manager for execution.

We executed the experiment with and without adaptation being enabled. In the latter scenario, a fix number of 6 nodes was allocated throughout the entire experiment. Figure 13 shows the measured CPU utilization and the varying number of allocated nodes with adaptation enabled. The number of allocated nodes in this experiment varies between one and six. Comparing the results of both settings, the average CPU utilization increased with adaptation enabled, while (average) response times were very similar (5 ms measured at the application's entry points).

This scenario demonstrates the generic applicability of our adaptation language. Furthermore, we show that it is possible to exchange precalculated adaptation plans between the planning and executing parties to achieve a desired system configuration in a cloud scenario.

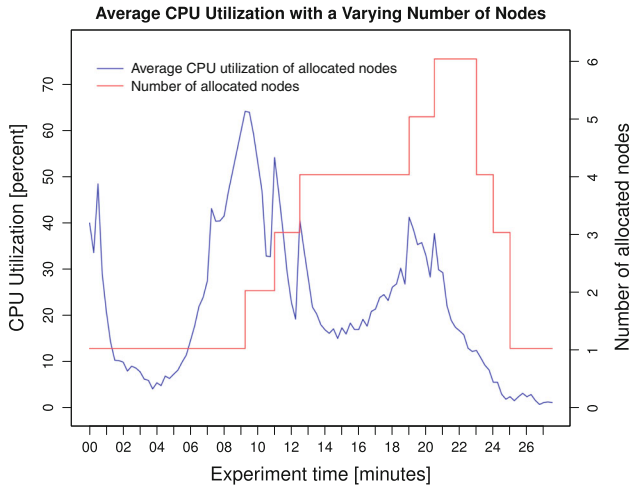

Fig. 13 Online capacity management executing S/T/A adaptation plans

3.5 Discussion and comparison

Vogel and Giese presented an extensive list of functional and nonfunctional requirements for adaptation languages and frameworks [31]. This list of requirements was used to assess two approaches, Stitch [5] and Story Diagrams [10]. We now use the same list to assess our adaptation language S/T/A and to compare it to Stitch and Story Diagrams (see Table 2). We also discuss technical aspects of S/T/A, i.e., its efficiency, accuracy, etc.

Concerning functional language requirements, S/T/A provides a way to specify the adaptation goal (LR-1), although not fully operationalized yet. S/T/A is designed to support all quality dimensions (LR-2) and provides the weights of the tactics to specify adaptation preferences (LR-3). Our language explicitly uses an architecture-level system QoS model which corresponds to the concept of a reflective model (LR-4). Adaptations are triggered by events (LR-5) and the evaluation conditions (LR-6) can be described using objectives. Evaluation results (LR-7) are not directly supported, but they can be stored in a separate QoS data repository. Similarly, adaptation options (LR-8) can be specified but are part of the adaptation space sub-model, not of our adaptation language directly. Specifying adaptation conditions (LR-9) is not possible, but one can define different strategies which can be applied under special conditions. Adaptation costs and benefits (LR-10) can be expressed by user-defined weighting functions that assign specific, cost- and benefit-considering weights to the tactics. Persisting the decision history (LR-11) is not part of S/T/A but is implemented in the prototypical framework interpreting S/T/A instances.

Regarding the nonfunctional language requirements, with its separation into strategies, transaction-like tactics and parameterized actions, S/T/A fulfills LR-13 and LR-14. However, as it is limited to the three abstraction-level strategies, tactics, and actions, it supports LR-12 only partially. Similarly, as S/T/A is specified using a meta-model and has no explicit formal definition, LR-15 is only partly fulfilled. S/T/A is partly reusable (LR-16) because it is independent of the architecture-level system QoS model, but its instances are coupled to concrete architecture-level QoS model instances like DMM. Concerning the ease of use (LR-17), S/T/A requires understanding of the concepts (e.g., control flow diagrams), but no programming skills, etc.

Looking at the framework requirements, our prototype supports consistency (FR-1), incrementality (FR-2) and reversibility (FR-3) because it works on the model level where consistency checks can be executed and changes can be executed incrementally and reversed, if necessary. It considers priorities (FR-4) by assigning weights to tactics and covers different time scales (FR-5) if appropriate strategies have been specified. It also supports a great amount of flexibility because tactics (and their weights) can be exchanged during execution.

Table 2 Language requirements (LR) and framework requirements (FR) comparison of Stitch and Story Diagrams [31], extended by S/T/A: '–' is no support, 'P' is partial support, and 'F' is full support

Req.	S/T/A	Stitch	SD
Functional language requirements			
LR-1	P	–	F
LR-2	F	F	F
LR-3	F	F	F
LR-4	F	P	F
LR-5	F	P	F
LR-6	F	F	F
LR-7	P	–	F
LR-8	P	F	F
LR-9	P	F	F
LR-10	P	F	F
LR-11	P	P	F
Nonfunctional language requirements			
LR-12	P	P	F
LR-13	F	–	P
LR-14	F	P	F
LR-15	P	–	P
LR-16	P	P	P
LR-17	P	P	F
Framework requirements			
FR-1	F	P	F
FR-2	P	–	P
FR-3	F	–	P
FR-4	F	–	F
FR-5	F	–	F
FR-6	F	–	F

When comparing the fulfillments of functional and non-functional requirements, our S/T/A approach is in between Stitch, which is more focused on system administration tasks, and the more general purpose-like Story Diagram language. This comparison has been conducted from a conceptual point of view and is based on the literature [5,10,31]. Comparing technical aspects like accuracy and efficiency of these three approaches remains a big challenge [4,8]. The main reasons are that the approaches must provide tool support and that they must be integrated into an existing self-adaptive system, ideally a benchmark. Although Story Diagrams provide graphical editors to model and validate adaptation models and Stitch provides some tool support, the effort to adapt these approaches to our scenario is huge. However, comparing the accuracy of our approach with the optimization approach PerOpteryx as a baseline (see Sect. 3.3), we can conclude that using S/T/A, it is possible to find a solution within a reasonable amount of time. The accuracy of the solution depends on the accuracy of the architecture-level system QoS model. If this model reflects the system state within a given confidence level, the solutions of S/T/A maintain the same accuracy. Regarding efficiency, the time consumed to find a solution mainly depends on the amount of required model analyses. Hence, if strategies and tactics are specified in an efficient manner, solutions are also found efficiently. Of course, one can speed up the analysis, e.g., by caching previous analysis steps or by using quicker but less accurate analysis techniques. However, optimizing our framework is part of future work.

4 Related work

In this section, we discuss the abstraction level employed by other approaches and compare our approach to related languages for defining adaptation (control flow), and other options to express the adaptation space of software architecture models.

Architectural models provide common means to abstract from the system details and analyze system properties. Such models have been used for self-adaptive software before, e.g., in [12,25], however, existing approaches do not explicitly capture the degrees of freedom of the system configuration as part of the models. The three-level abstraction of adaptation processes can be found in other approaches too, e.g., by Kephart and Walsh [16] to specify policy types for autonomic computing or especially by Cheng et al. [6], defining an ontology of tactics, strategies and operations to describe self-adaptation. However, to the best of our knowledge, none of the existing approaches separates the specification of the models at the three levels. By separating the knowledge about the adaptation process and encapsulating it in different sub-models, we can reuse this knowledge in other self-adaptive systems.

In the field of software engineering, there exist various languages with different purposes to describe adaptation. Cheng and Garlan introduce Stitch [5], a programming language-like notation for using strategies and tactics. However, strategies refer to tactics in a strictly deterministic, process-oriented fashion. Therefore, the knowledge about system adaptation specified with Stitch is still application-specific, making it difficult to adapt in situations of uncertainty. Other languages like Service Activity Schemas (SAS) [9] or the Business Process Execution Language (BPEL) [23] are very domain-specific and also describe adaptation processes with predefined control flows for service-oriented systems. Moreover, because of their focus on modeling business processes, these approaches are not able to model the full spectrum of self-adaptive mechanisms from conditional expressions to algorithms and heuristics. Yet Another Workflow Language (YAWL) [27] is based on Petri nets and has been used by da Silva and de Lemos [7] to coordinate the self-adaptation of a software system. The authors execute architectural reconfiguration using dynamic workflows to adapt to changing requirements at run-time. The authors also consider strategies and tactics corresponding to [6]. However, the focus of this approach is to automatically generating workloads for self-adaptation. Thus, it does not integrate a detailed architecture-level system QoS model as this approach to evaluate the effect of the adaptation process.

If constraining the specification of adaptation to the model level (i.e., adaptation languages for adapting model instances), well-known methods from the area of graph grammars could be used to adapt models, too. For example, Agrawal et al. [1] present an approach on how to define model transformations based on UML. Another example for a graph grammar language is previously presented Story Diagrams [10], which is also based on UML and Java. However, these approaches remain on the model level, whereas our approach should also execute real system adaptations. For a more detailed discussion of the two most related approaches (Stitch and Story Diagrams), we refer to Sect. 3.5.

For modeling the adaptation space of a software architecture, we use PCM's Degree-of-Freedom Meta-Model [18] or the Adaptation Points Meta-Model, which is an integral part of DMM [17], allowing to capture different types of adaptation changes, e.g., to add vCPUs, to add servers and to exchange software components, in a single model. In the area of automated software architecture improvement, most existing approaches use a fixed representation of the adaptation space and thus do not allow to freely model an adaptation space. Two notable exceptions are PETUT-MOO and the Generic Design Space Exploration Framework (GDSE). The PETUT-MOO approach [22] uses model transformations to describe changes in the configuration of software

architectures. However, this idea has not been followed up in later works of the authors, which focuses on architecture optimization and does not describe the adaptation space in detail.

5 Conclusions and future work

In this paper, we presented our novel S/T/A meta-model for describing system adaptation in component-based system architectures. Our approach is based on architecture-level system QoS models, on the one hand, and the S/T/A model, on the other hand, separating the knowledge about possible adaptation steps from the actual adaptation plans. This separation allows to explicitly model adaptation processes in an intuitive and at the same time machine-readable manner, enabling the reuse of plans in different autonomic or self-adaptive systems.

In an extensive evaluation, we applied our approach in multiple distinctive and representative scenarios using different architecture-level system QoS models. In this scenarios, we demonstrated the use of the proposed adaptation language to model dynamic resource allocation algorithms, online capacity planning and design-time system optimization. We showed how our S/T/A models interact with the underlying system models and how they improve system adaptation by focusing the search for suitable configurations and reducing the number of costly evaluations. Finally, we showed how S/T/A can be used as an intermediate language by adaptation planners or agents. Overall, our developed approach showed how the gap between complex system adaptations and self-adaptation at run-time can be closed.

Our future work aims into two directions. Conceptually, we want to provide a more detailed formalization of our adaptation language. Additionally, it would be valuable to conduct empirical studies on the benefits of adaptation languages in general and of our language compared to other approaches. On the technical level, we are working on graphical and textual editors to ease modeling adaptation processes with S/T/A. Finally, we intend to work on developing further heuristics and optimization algorithms specifically tailored for use in our S/T/A language.

References

1. Agrawal A, Karsai G, Shi F (2003) A UML-based graph transformation approach for implementing domain-specific model transformations. J Softw Syst Model 1–19
2. Becker S, Koziolek H, Reussner R (2009) The Palladio component model for model-driven performance prediction. J Syst Softw 82(1):3–22
3. Brosig F, Huber N, Kounev S (2012) Modeling parameter and context dependencies in online architecture-Level Performance Models. In: International symposium on component based software engineering (CBSE), ACM, pp 3–12
4. Cheng B et al. (2009) Software engineering for self-adaptive systems: a research roadmap. Softw Eng Self-Adaptive Syst 1–26
5. Cheng SW, Garlan D (2012) Stitch: a language for architecture-based self-adaptation. J Syst Softw 85(12):2860–2875
6. Cheng SW, Garlan D, Schmerl B (2006) Architecture-based self-adaptation in the presence of multiple objectives. In: International workshop software engineering for adaptive and self-managing systems (SEAMS), ACM, pp 2–8
7. da Silva C, de Lemos R (2009) Using dynamic workflows for coordinating self-adaptation of software systems. In: Software engineering for adaptive and self-managing systems (SEAMS). IEEE, pp 86–95
8. de Lemos R, Giese H, Müller H, Shaw M (2011) Software engineering for self-adaptive systems: a second research roadmap. In: Software engineering for self-adaptive systems, no. 10431 in dagstuhl seminar proceedings
9. Esfahani N, Malek S, Sousa J, Gomaa H, Menascé D (2009) A modeling language for activity-oriented composition of service-oriented software systems. In: International conference on model driven engineering languages and systems (MODELS), ACM, pp 591–605
10. Fischer T, Niere J, Torunski L, Zündorf A (2000) Story diagrams: a new graph rewrite language based on the unified modeling language and java. Theory Appl Graph Transform, pp 296–309
11. Frey S, van Hoorn A, Jung R, Hasselbring W, Kiel B (2011) MAMBA: A measurement architecture for model-based analysis. Technical report, University of Kiel, Germany
12. Garlan D, Cheng S, Huang A, Schmerl B, Steenkiste P (2004) Rainbow: architecture-based self-adaptation with reusable infrastructure. Computer 37(10):46–54
13. Huber N, Brosig F, Kounev S (2011) Model-based self-adaptive resource allocation in virtualized environments. In: International symposium on software engineering for adaptive and self-managing systems (SEAMS), ACM, pp 90–99
14. Huber N, Brosig F, Kounev S (2012a) Modeling dynamic virtualized resource landscapes. In: International conference on the quality of software architectures (QoSA), ACM, pp 81–90
15. Huber N, van Hoorn A, Koziolek A, Brosig F, Kounev S (2012b) S/T/A: meta-modeling run-time adaptation in component-based system architectures. In: International conference on e-business engineering (ICEBE), IEEE, pp 70–77
16. Kephart J, Walsh W (2004) An artificial intelligence perspective on autonomic computing policies. In: International workshop on policies for distributed systems and networks. IEEE, pp 3–12
17. Kounev S, Brosig F, Huber N (2012) Descartes meta-model (DMM). Technical report, Karlsruhe Institute of Technology (KIT), http://www.descartes-research.net/metamodel/, to be published
18. Koziolek A, Reussner R (2011) Towards a generic quality optimisation framework for component-based system models. In: International symposium on component-based software engineering (CBSE), ACM, pp 103–108
19. Koziolek H (2010) Performance evaluation of component-based software systems: a survey. Perform Eval 67(8):634–658
20. Kramer J, Magee J (2007) Self-managed systems: an architectural challenge. In: Future of software engineering (FOSE), pp 259–268
21. Martens A, Koziolek H, Becker S, Reussner RH (2010) Automatically improve software models for performance, reliability and cost using genetic algorithms. In: International conference on performance engineering (ICPE), ACM, pp 105–116
22. Maswar F, Chaudron MRV, Radovanovic I, Bondarev E (2007) Improving architectural quality properties through model transformations. In: Software engineering research and practice. CSREA Press, pp 687–693

23. OASIS (2007) WS-BPEL Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html
24. Object Management Group (2012) Structured Metrics Meta-Model (SMM). http://www.omg.org/spec/SMM/1.0/
25. Oreizy P et al (1999) An architecture-based approach to self-adaptive software. Intell Syst Appl, IEEE 14(3):54–62
26. S/T/A Framework and Meta-Model (2013) http://www.descartes-research.net/tools/
27. van der Aalst W, ter Hofstede A (2005) YAWL: yet another workflow language. Inf Syst 30(4):245–275
28. van Hoorn A (2013) Online capacity management for increased resource efficiency of component-based software systems. PhD thesis, University of Kiel, Germany, work in progress
29. van Hoorn A, Waller J, Hasselbring W (2012) Kieker: a framework for application performance monitoring and dynamic software analysis. In: International conference on performance engineering (ICPE), ACM, pp 247–248
30. von Massow R, van Hoorn A, Hasselbring W (2011) Performance simulation of runtime reconfigurable component-based software architectures. In: European conference on software architecture (ECSA). Springer, pp 43–58
31. Vogel T, Giese H (2012) Requirements and assessment of languages and frameworks for adaptation models. In: International conference on models in software engineering (MODELS). Springer, pp 167–182