S/T/A: Meta-Modeling Run-Time Adaptation in Component-Based System Architectures

Nikolaus Huber[*], André van Hoorn[†], Anne Koziolek[‡], Fabian Brosig[*] and Samuel Kounev[*]
[*]Karlsruhe Institute of Technology, Karlsruhe, Germany, Email: {nikolaus.huber, fabian.brosig, kounev}@kit.edu
[†]Christian-Albrechts-University Kiel, Kiel, Germany, Email: avh@informatik.uni-kiel.de
[‡]University of Zurich, Zürich, Switzerland, Email: koziolek@ifi.uzh.ch

***Abstract*—Modern virtualized system environments usually host diverse applications of different parties and aim at utilizing 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 propose a novel modeling language (meta-model) providing means to describe system adaptation processes at the system architecture level in a generic, human-understandable and reusable way. We apply our approach to three different realistic contexts (dynamic resource allocation, software architecture optimization, and run-time adaptation planning) showing how the gap between complex manual adaptations and their autonomous execution can be closed by using a holistic model-based approach.**

I. Introduction

Today's software systems are increasingly flexible and dynamic providing means 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 systems 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 in [1] 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. 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 [2].

Self-adaptation approaches based on architectural models have been proposed before (e.g., [3], [4], [5]). 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.

In this paper, we present a novel domain-specific language, called S/T/A (Strategies/Tactics/Actions), providing means to describe run-time system adaptation in component-based system architectures based on architecture-level system QoS models. The latter describe the QoS-relevant components of the system and reflect 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 which 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.

Our approach has several important advantages: 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.

The contributions of the paper are: i) 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. ii) A general purpose meta-model (S/T/A) for describing the logical aspects of system adaptation processes in a generic way. The meta-model is capable of modeling (self-)adaptation described 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. iii) An example for adapting dynamic systems using our models as input. It applies meta-modeling techniques end-to-end, i.e., from the system architecture up to the high-level system adaptation plans. iv) 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.

The rest of the paper is structured as follows: In Section II, we present our modeling approach and the proposed adaptation language which is illustrated with examples. In Section III, we evaluate our approach in three different representative application scenarios. Section IV gives an overview of related work and finally Section V concludes the paper and outlines our planned future work.

II. Modeling System Adaptation

In the context of dynamic system adaptation, we distinguish between *technical view* and *logical view*. The *technical view* describes in detail the exact architecture of the system, what parts of the system can be adapted at run-time. We capture such information in architecture-level system QoS model. The *logical view* describes the adaptation process which keeps the system in a desired state. This is modeled by the S/T/A adaptation language. Figure 1 depicts the connection between the technical and logical view of the system.

A. 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 sub-models, namely system architecture sub-model and 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 QoS. Examples of suitable architecture-level QoS models are the Palladio Component Model (PCM) [6] or the Descartes Meta-Model (DMM) [7] considered in this paper, or other component-based performance models as surveyed in [8]. 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 the 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 [9] for PCM or the Adaptation Points Meta-Model which is an integral part of DMM. 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 [4], [3]. 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 in [4]). 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 real system, respectively.

B. Example: Dynamic Resource Allocation

In [10], we presented an algorithm for dynamic resource allocation in virtualized environments. This algorithm is currently implemented in Java. It is highly system-specific, unintuitive and 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 Sec. II-C and as one of the evaluation scenarios in Sec. III.

The algorithm uses an architecture-level system QoS model we already developed and successfully applied 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

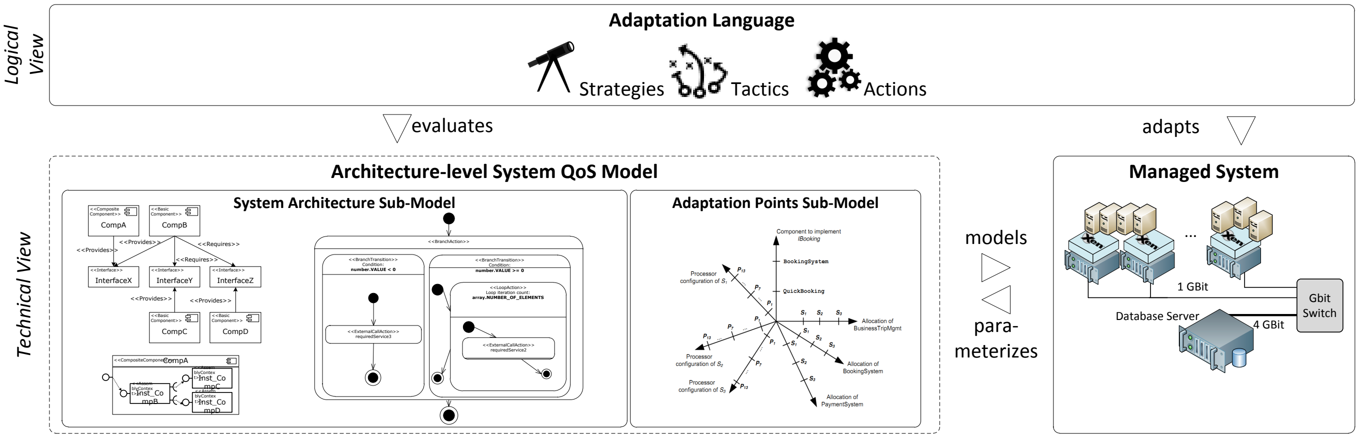

Fig. 1. Interaction of the system, the system models and the S/T/A adaptation language.

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 are 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 [10].

C. 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. 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.

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 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: i) atomic, i.e., either the whole tactic with all its contained actions is executed or the tactic must be rolled back, ii) consistent, i.e., the model's and system's state must be consistent after applying a tactic, and iii) 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 run-time. 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 jump back to the state before starting to apply the tactic in case an error is detected saving costly executions of roll-back operations on the system.

The distinction of these three abstraction levels can be found in other approaches too, e.g., in [11] or [5], however, with limited expressiveness (cf. Section IV). In contrast to 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 Figure 2.

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 3 shows the four actions we modeled in our dynamic resource allocation algorithm. The actions addVCPU and addAppServer 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. These actions do not implement the logic of the operation, they are simply references to adaptation points defined in the respective architecture-level system QoS model, i.e., DMM instance in this case. On the model level, actions are implemented by the framework using the models. On the system level, the virtualization layer executes the respective operations on the real system.

2) Tactic: A Tactic specifies a 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 abstract 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 Figure 3, 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 to scale the system up or down, 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 of the tactic chooses which resource type to add. This is an example for separating technical from logical details. The adaptation plan in the Loop action implements two actions, addVCPU and addAppServer. Which action is executed depends on the current system state. If there is no possibility to add a vCPU (determined by an OCL expression AllServersAtMaxCap), an application server is added.

The adaptation plan of the tactic removeResources either removes an application server VM if there is a server running at minimum capacity (determined using an OCL expression ServerAtMinCapExists) or removes a vCPU from an application server VM. The undoPreviousAction tactic can be used in cases where the previous adaptation step must be undone.

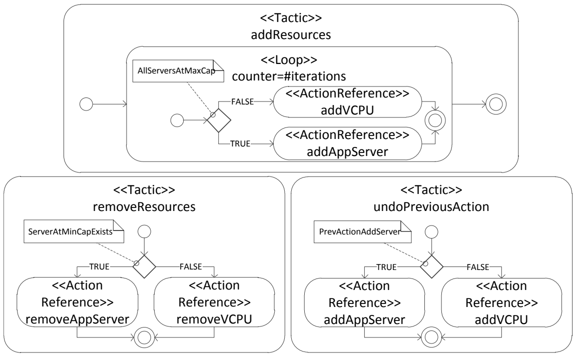

Fig. 3. Actions and tactics for the running example.

3) Strategy: The purpose of a Strategy is to achieve a high-level Objective. The objective of a strategy is part of the OverallGoal specified, e.g., by the system administrator. For example, an objective can be described with goal policies or utility function policies. One could also use any other type of description that can be automatically checked by analyzing the model or monitoring data from the real system (e.g., based on MAMBA [12]). Note that it is explicitly allowed to have multiple alternative strategies with the same objective because strategies might differ in their implementation.

The execution of a strategy is triggered by a specific Event that occurs during system operation, e.g., when an SLA is violated or a adaptation to increase efficiency is scheduled. Such an event triggers the execution of the respective 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 to avoid concurrency effects, e.g., two strategies operating at the same time but with a conflicting objective. However, we do not exclude situations where there are conflicting objectives. Such

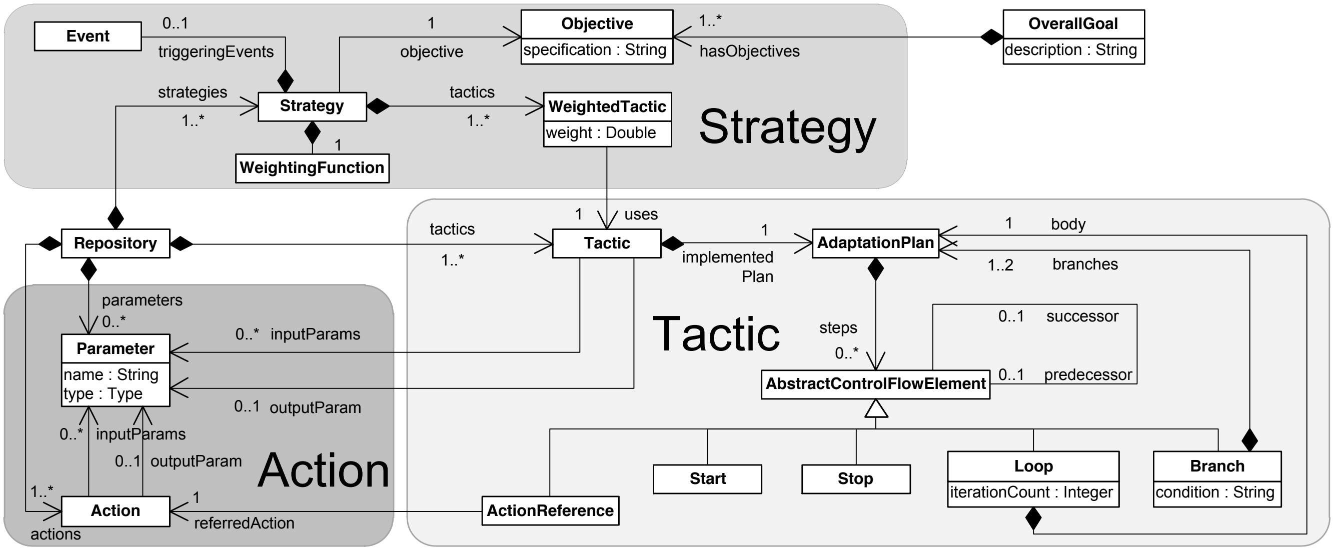

Fig. 2. Adaptation language meta-model.

cases can be handled using strategies with objectives modeled as utility function policies [11]. 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 WeightedTactics. A WeightedTactic is a decorator for a Tactic. Weights are assigned according to the strategy's WeightingFunction. The strategy uses the weight to determine which tactic to apply next. The weight depends on the current state of the system, i.e., the weights can change if the state of the system changes. Hence, the strategy might choose a different tactic in the next adaptation step. Formally, assume

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

Then one can think of a mapping $WT \in [T \times S \rightarrow [0,1] \subseteq \mathbb{R}]$ that assigns a weight to the given tactic $t \in T$ in the system state $s \in S$. It is not part of this work and it is intentionally left open to actually specify this mapping in more detail. The idea is to use existing and well-established optimization algorithms or meta-heuristics to determine the weights depending on the current state of the system possibly also considering its previous states stored in a trace. 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 4 depicts the two strategies of the algorithm in our running example, PUSH and PULL, with the objective to improve response times to maintain SLAs, and to ensure efficient resource usage, respectively. 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 system's QoS properties. 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. Determining the weight for the PUSH tactic in this scenario is straightforward as it is always 1.0 because no other tactics are defined as part of the strategy.

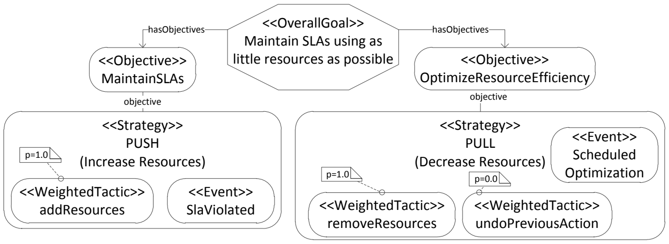

Fig. 4. Strategies using tactics and assigning weights.

The PULL strategy is triggered with the objective to OptimizeResourceEfficiency, either based on a predefined schedule or directly after the PUSH phase. The PULL strategy has a tactic to reduce the amount of used resources (removeResources). 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 their weights. An example scenario of how to use weights is described in the next section.

III. Evaluation

We evaluate our presented adaptation language in three 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 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. The second scenario demonstrates that the S/T/A adaptation language can produce useful solutions in reasonable amount of time by evaluating its usability in a framework for multi-objective software architecture optimization. Finally, the third scenario gives an example for decoupling the adaptation process by using S/T/A model instances as adaptation plans.

A. Dynamic Resource Allocation

Figure 5 shows the results of one of our experiments using the models of our running example to manage resource allocations in a virtualized environment. 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 the Descartes Meta-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.

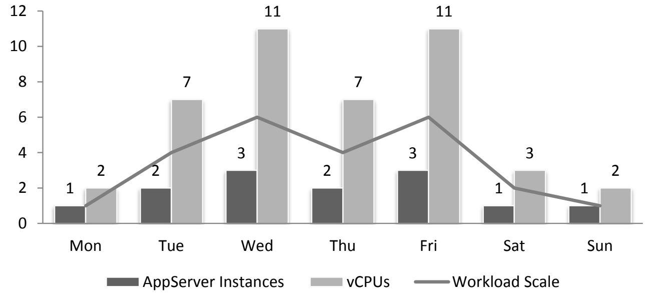

Fig. 5. Dynamic resource allocations using the modeled algorithm.

B. 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 [13]. 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 (PCM [6]).

For a given architecture-level system QoS model, PerOpteryx searches the space of possible configurations for candidates that fit given objectives. 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.

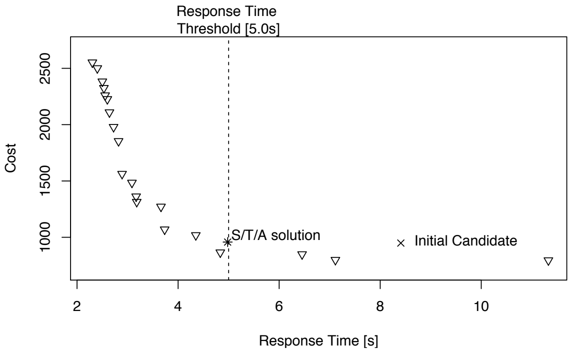

Fig. 6. Pareto optimal candidates found by PerOpteryx ($\triangledown$) and candidates found when S/T/A is applied to guide the search process ($*$).

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 in [13] to evaluate PerOpteryx. The output after 100 iterations with a population of 60 is a set of Pareto-optimal candidates marked by $\triangledown$ in Figure 6. In our scenario, we assume that the response time of the initial candidate (8.4 seconds) 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 can choose 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 final resulting candidate, i.e., system configuration, of this process is depicted by a $*$ symbol in Figure 6.

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. Thus, using S/T/A models can speed-up the process to find a close to optimal solution. 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

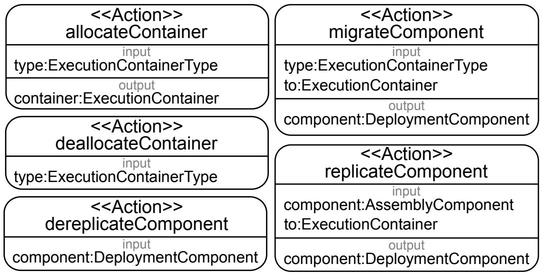

Fig. 7. Actions of the online capacity management scenario.

time.

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.

C. 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 [14], [15]. 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 approach can be used as a language to specify and execute architectural adaptation plans, in order to bring a real or simulated system from a current to a desired configuration using adaptation plans. This section presents some of our results of a lab 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 Figure 7 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 [15]. In the cloud scenario, a adaptation manager receives pre-calculated S/T/A `AdaptationPlans` and executes them by interacting with the AWS API and the allocated nodes according to the actions specified in Figure 7.

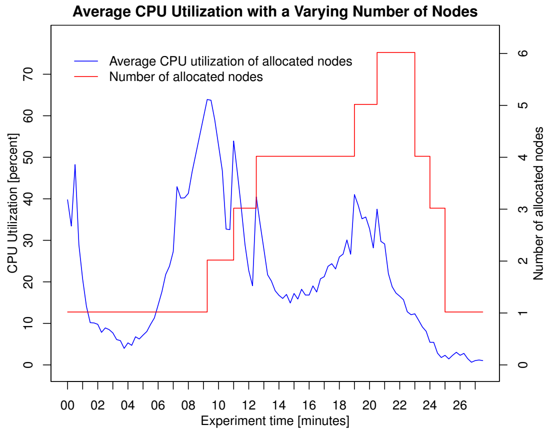

Fig. 8. Online capacity management executing S/T/A adaptation plans.

In our evaluation, we expose a Java-based application (JPetStore 5.0) to a probabilistic workload with varying intensity based on a 24-hour workload profile obtained from an industrial system. The profile was scaled to an experiment duration of 24 minutes plus 2 minutes 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.

The experiment was executed with and without adaptation being enabled. In the latter scenario, a fix number of 6 nodes was allocated throughout the entire experiment. Figure 8 shows the measured CPU utilization and the varying number of allocated nodes with adaptation enabled. The number of allocated nodes varies between 1 and 6. 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 pre-calculated adaptation plans between the planning and executing parties to achieve a desired system configuration in a cloud scenario.

IV. Related Work

In this section, we discuss the abstraction level employed by other approaches, 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 [3], [4], 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., in [11] to specify policy types for autonomic computing or especially in [5], 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 [16], Cheng introduces Stitch, 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) [17] or the Business Process Execution Language (BPEL) [18]) are very domain specific and also describe adaptation processes with predefined control flows. 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.

For modeling the adaptation space of a software architecture, we use PCM's Degree-of-Freedom Meta-Model [9] or the Adaptation Points Meta-Model which is an integral part of DMM [7], 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 [19] 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 adaptation space in detail.

V. 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 three distinctive and representative scenarios using different architecture-level system QoS models. Thereby, 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 focussing 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.

In our future work, we plan to extend our framework such that it can be used by third parties in further scenarios. Moreover, we are working on graphical editors to ease modeling adaptation processes with S/T/A.

References

[1] R. de Lemos, H. Giese, H. Müller, and M. Shaw, "Software Engineering for Self-Adaptive Systems: A second Research Roadmap," in *Software Engineering for Self-Adaptive Systems*, no. 10431, 2011.

[2] B. Cheng *et al.*, "Software engineering for self-adaptive systems: A research roadmap," *Software Engineering for Self-Adaptive Systems*, 2009.

[3] P. Oreizy *et al.*, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems*, 1999.

[4] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[5] S.-W. Cheng, D. Garlan, and B. Schmerl, "Architecture-based self-adaptation in the presence of multiple objectives," in *SEAMS'06*, 2006.

[6] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *Journal of Syst. and Softw.*, 2009.

[7] F. Brosig, N. Huber, and S. Kounev, "Descartes Meta-Model (DMM)," Karlsruhe Institute of Technology (KIT), Tech. Rep., 2012, to be published. [Online]. Available: www.descartes-research.net

[8] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Perf. Eval.*, vol. 67, no. 8, pp. 634–658, 2010.

[9] A. Koziolek and R. Reussner, "Towards a generic quality optimisation framework for component-based system models," in *CBSE 2011.*

[10] N. Huber, F. Brosig, and S. Kounev, "Model-based Self-Adaptive Resource Allocation in Virtualized Environments," in *SEAMS'11*, 2011.

[11] J. Kephart and W. Walsh, "An artificial intelligence perspective on autonomic computing policies," in *IEEE Int'l Workshop on Policies for Distributed Systems and Networks*, 2004.

[12] S. Frey, A. van Hoorn, R. Jung, W. Hasselbring, and B. Kiel, "MAMBA: A measurement architecture for model-based analysis," University of Kiel, Germany, Tech. Rep., 2011.

[13] A. Martens, H. Koziolek, S. Becker, and R. H. Reussner, "Automatically improve software models for performance, reliability and cost using genetic algorithms," in *WOSP/SIPEW '10*, 2010.

[14] A. van Hoorn, "Online capacity management for increased resource efficiency of component-based software systems," Ph.D. dissertation, University of Kiel, Germany, 2012, work in progress.

[15] R. von Massow, A. van Hoorn, and W. Hasselbring, "Performance simulation of runtime reconfigurable component-based software architectures," in *ECSA'11*. Springer, 2011.

[16] S.-W. Cheng, "Rainbow: cost-effective software architecture-based self-adaptation," Ph.D. dissertation, Carnegie Mellon University, 2008.

[17] N. Esfahani, S. Malek, J. Sousa, H. Gomaa, and D. Menascé, "A modeling language for activity-oriented composition of service-oriented software systems," *Model Driven Engin. Languages and Systems*, 2009.

[18] OASIS, "WS-BPEL Version 2.0," 2007.

[19] F. Maswar, M. R. V. Chaudron, I. Radovanovic, and E. Bondarev, "Improving architectural quality properties through model transformations," in *Software Engineering Research and Practice*, 2007, pp. 687–693.