Autom Softw Eng (2018) 25:595–626
https://doi.org/10.1007/s10515-018-0234-9

MOSAICO: offline synthesis of adaptation strategy repertoires with flexible trade-offs

Javier Cámara[1] · **Bradley Schmerl**[1] · **Gabriel A. Moreno**[2] · **David Garlan**[1]

Received: 27 March 2017 / Accepted: 25 April 2018 / Published online: 7 May 2018

Abstract Self-adaptation improves the resilience of software-intensive systems, enabling them to adapt their structure and behavior to run-time changes (e.g., in workload and resource availability). Many of these approaches reason about the best way of adapting by synthesizing adaptation plans *online* via planning or model checking tools. This method enables the exploration of a rich solution space, but optimal solutions and other guarantees (e.g., constraint satisfaction) are computationally costly, resulting in long planning times during which changes may invalidate plans. An alternative to online planning involves selecting at run time the adaptation best suited to the current system and environment conditions from among a predefined repertoire of adaptation strategies that capture repair and optimization tasks. This method does not incur run-time overhead but requires additional effort from engineers, who have to specify strategies and lack support to systematically assess their quality. In this article, we present MOSAICO, an approach for *offline* synthesis of adaptation strategy repertoires that makes a novel use of discrete abstractions of the state space to flexibly adapt extra-functional behavior in a scalable manner. The approach supports making trade-offs: (i) among multiple extra-functional concerns, and (ii) between computation time

✉ Javier Cámara
jcmoreno@cs.cmu.edu

Bradley Schmerl
schmerl@cs.cmu.edu

Gabriel A. Moreno
gmoreno@sei.cmu.edu

David Garlan
garlan@cs.cmu.edu

[1] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA

[2] Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, USA

and adaptation quality (varying abstraction resolution). Our results show a remarkable improvement on system qualities in contrast to manually-specified repertoires. More interestingly, moderate increments in abstraction resolution can lead to pronounced quality improvements, whereas high resolutions yield only negligible improvement over medium resolutions.

Keywords Self-Adaptation · Resilience · Synthesis · Model checking

1 Introduction

The last two decades have seen a continuous growth in the complexity of software-intensive systems, which are increasingly relied on for a wide variety of tasks in different application domains, such as energy, communications, and security. The critical nature of these applications has led to a central concern for their *resilience* in the presence of environmental changes, faults, and attacks (Laprie 2008).

Autonomic and self-adaptive systems (Cheng et al. 2009; Huebscher and McCann 2008) are generally considered to be efficient approaches for engineering *resilient* software systems in a cost-effective manner. Such systems are characterized by the separation of the adaptation concern into a control layer that endows the system with the ability to modify its structure and behavior in response to run-time changes (Kephart and Chess 2003; Kramer and Magee 2007).

Many self-adaptation approaches have shown the effectiveness of employing architectural descriptions for reasoning about the best way of adapting the system by synthesizing adaptation plans *online* via planning or model checking tools (da Silva and de Lemos 2011; Mukhija et al. 2007; Sykes et al. 2010; Tajalli et al. 2010). These enable the exploration a rich solution space and can yield in some cases plans that meet formal guarantees, such as optimality with respect to quality objectives (Sykes et al. 2010), or constraint satisfaction (Tajalli et al. 2010).

However, such guarantees come at a high cost due to the computational overhead of the synthesis process, resulting in planning times that often go beyond the desirable duration of the adaptation cycle (Cámara et al. 2015; Sykes et al. 2010). When such situations occur, the resulting plan commonly becomes invalid due to changes in the system and its environment that occur during planning time. Moreover, synthesis does not guarantee a priori the existence of a plan that will satisfy adaptation goals, given a set of assumptions about the environment, potentially leading to situations in which the planning activity does not yield *any* solutions at run time.

To improve on this situation, some recent approaches to self-adaptation draw from the areas of discrete event planning (Schoppers 1987) and supervisory control (Piterman et al. 2006) for synthesizing *offline* adaptation behavior that can be reused at run time to gracefully degrade when environment assumptions are broken (D'Ippolito et al. 2014), or recover from run-time failures (Carzaniga et al. 2013). These approaches mitigate run-time overhead, but tend to focus exclusively on functional behavior.

Offline approaches that go beyond functional behavior are often grounded in control theory and target tunable variables to produce control strategies for adaptive systems with formal guarantees (Filieri et al. 2014, 2015; Klein et al. 2014). Although some

of them can handle the satisfaction of multiple objectives [e.g., Filieri et al. (2015)], they cannot explicitly consider trade-offs among them nor between computation time and quality of the solution obtained, to the best of our knowledge.

Aside from synthesis, a class of alternative approaches to this problem involves selecting at run time an adaptation from a predefined repertoire of strategies that captures repair and optimization tasks (Garlan et al. 2004; Huber et al. 2014; Nou et al. 2009). Approaches that select at run time the strategy that is best suited to the current state of the system and its environment incur in a negligible run-time overhead compared to online synthesis, and are better equipped than existing approaches based on offline synthesis to reason about trade-offs among multiple extra-functional properties, such as performance, cost, or security (Cheng et al. 2009; Huber et al. 2014; Schmerl et al. 2014). Unfortunately, this type of approach demands additional effort from software engineers, who have to write the specification of adaptation strategies (Cámara et al. 2013; Cheng 2008). Moreover, engineers lack support to systematically assess in a scalable manner the quality of the strategy repertoires produced. Hence, the development of a strategy repertoire involves iterative testing and debugging, during which developers have to manually assess aspects such as coverage of the state space (i.e., if there is always an applicable adaptation strategy for all relevant situations that can be given), which strategies get selected under what conditions, or what the impact of those selections on quality objectives is.

In this article, we introduce Multi-Objective Synthesis of AdaptatIon COllections (MOSAICO), an approach to *offline* synthesis of adaptation strategy repertoires. Our approach is inspired by a class of approaches that employ discrete abstractions of continuous system dynamics to leverage controller synthesis techniques in the area of supervisory control of discrete-event systems (Ramadge and Wonham 1987). In a nutshell, our technique consists in discretizing the system/environment state space and synthesizing optimal adaptation plans for the different points of the discrete abstraction via probabilistic model checking (Kwiatkowska et al. 2007). Our approach combines the best of both worlds by eliminating the run-time overhead of online synthesis while retaining the ability to provide *nearly-optimal* solutions, and reducing the specification effort required from engineers, compared to predefined strategy specification and selection.

To the best of our knowledge, this is the first synthesis approach that can handle trade-offs between computation time and quality of the solution, multiple extra-functional concerns, and provide feedback about the quality of the repertoires generated across different regions of the state space.

We build on work for online synthesis of self-adaptation under uncertainty described in Cámara et al. (2015), which is intended for adaptation in a single point of the state space and does not provide any estimations about the quality or feasibility of adaptation prior to deployment. In contrast, the present approach is intended to synthesize adaptation collections that cover an entire region of the state space, providing feedback about issues such as coverage, constraint violations, or quality of the adaptations. Moreover, our approach overcomes the scalability limitation of Cámara et al. (2015) by making novel use of discrete abstractions of the state space. This enables us to: (i) trade-off solution quality for computation time, (ii) parallelize the synthesis process by distributing different regions of the abstraction across different computation nodes,

and (iii) transfer the synthesis process from run time to development time, if needed. Similarly to Cámara et al. (2015), our approach fully supports probabilistic reasoning. However, this article focuses on the novel use of discrete abstractions and the way in which they enable time/quality trade-offs, and hence we do not to illustrate the probabilistic aspects of the approach, due to space and clarity concerns [cf. Cámara et al. (2015)].

We validate our approach using the Rainbow framework for self-adaptation (Garlan et al. 2004) and a Denial-of-Service (DoS) attack scenario in Znn.com, a custom-built web system that mimicks a news site with multimedia news articles (Cheng et al. 2009; Schmerl et al. 2014). Our results show: (i) consistency between synthesized repertoires and design specifications (e.g., adaptations are appropriate to the conditions under which they are chosen), (ii) flexibility in trading off computation time and quality of adaptation, and (iii) substantial improvement in quality by comparison with manually-specified repertoires.

In the remainder of this article, Sect. 2 introduces a motivating scenario. Section 3 gives an overview of our approach. Next, Sect. 4 describes the main concepts in Rainbow/Stitch adaptation models. Section 5 gives some basic background on probabilistic model checking. Sections 6 and 7 describe in detail our approach and its evaluation, respectively. Section 8 discusses limitations and threats to validity, and Sect. 9 contrasts our approach with related work. Section 10 presents some conclusions and future work.

2 Motivating scenario

On-line N-tiered systems is one of the many application domains required to be self-adaptive. In fact, there are a number, such as RUBiS (http://rubis.ow2.org), RUBBoS (http://jmob.ow2.org/rubbos.html), Znn.com (Cheng et al. 2009), or C-MART (http://theone.ece.cmu.edu/cmart) that are often employed as benchmarks for cloud-based solutions and self-adaptation approaches (Klein et al. 2014; Turner et al. 2013). These applications are considered to portray representative scenarios for self-adaptation, since they have in common: (i) a set of comparable run-time actuation points to cope with dynamic workloads that typically involve changing the pool of available resources (e.g., spinning up new VMs, enlisting replicated servers) or adjusting the fidelity (or service level, e.g., by reducing the level of computation, downgrading the quality of video streams), and (ii) a set of comparable extra-functional concerns, such as performance, cost, or security, that must be balanced for proper system operation.

Znn.com Cheng et al. (2009) embodies a typical infrastructure for a news website. It has a three-tier architecture consisting of a set of servers that provide contents from back-end databases to clients via front-end presentation logic (Fig. 1). The system uses a load balancer to balance requests across a pool of replicated servers, the size of which can be adjusted according to service demand. A set of clients makes stateless requests, and the servers deliver the requested contents.

From time to time, Znn.com can experience spikes in requests that it cannot serve adequately, even at maximum pool size. These spikes can result from legitimate client

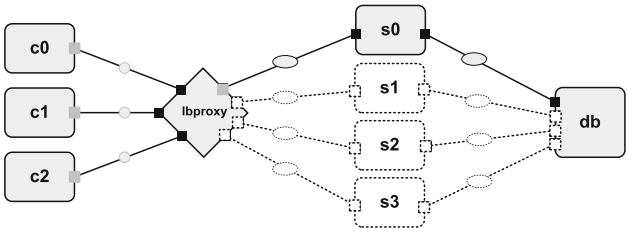


Fig. 1 Znn.com system architecture

traffic caused by a popular event [e.g., *slashdot effect* (https://en.wikipedia.org/wiki/Slashdot_effect)], or by DoS attacks in which malicious clients try to overwhelm system capacity to render system services unavailable.

System objectives System users desire service without any disruption, whereas the organization wants to minimize the cost of operating the infrastructure (including not incurring additional operating costs derived from DoS attacks). For users, service disruption can be mapped to run-time conditions such as (i) high response time for legitimate clients, and (ii) user annoyance, often related to disruptive side effects of defensive tactics [c.f., Beheshti and Liatsis (2015)]. For the organization, we map cost to the resources operated in the infrastructure at run time (e.g., number of active servers). In addition to keeping costs below budget, the organization wants to minimize the fraction of the cost that corresponds to resources consumed by malicious clients. Hence, we identify minimizing the presence of malicious clients as an additional objective.

In short, we identify four quality objectives for Znn.com: legitimate client response time (R), user annoyance (A), cost (C), and client maliciousness (M).[1]

Adaptation mechanisms When response time becomes too high due to increases in requests, the system can employ two general approaches for dealing with the situation: absorb the excess of traffic or suppress it. While the former approach is better suited to situations in which legitimate user traffic has increased due to a popular event, the latter is indicated for dealing with DoS attacks.

Znn.com can *absorb* the excess traffic by employing the tactics: (i) *add capacity*, which commissions new replicated web servers to share the load; and (ii) *reduce service*, which reduces the level of service by serving text-only pages instead of multimedia content. These tactics are good at improving the performance of the system while causing a negligible impact on annoyance to legitimate users (when reducing service level). However, employing these tactics comes at a price, since they do not deal with reducing the cost derived from resources consumed by malicious clients, and they can even result in an increase in the cost of operating the system (when adding capacity).

[1] Other instantiations of scenarios based on Znn consider fidelity/service level as a primary objective. However, we assume that fidelity is dominated by performance and user annoyance in overall user experience. Hence, fidelity adjustment is considered only as a mechanism to affect performance in our scenario. A more detailed rationale for the extra-functional requirements of this scenario can be found in Schmerl et al. (2014).

Alternatively, Znn.com can *eliminate* the excess traffic by enacting the tactics: (i) *blackhole*, which adds the IP addresses of clients that are deemed to be attacking the system to a blacklist that blocks their requests; (ii) *throttle*, which limits the rate of requests accepted from potentially malicious clients; (iii) *enable captcha*, which forwards requests to a captcha processor which acts as a Turing test and verifies that the requester is human; and (iv) *reauthenticate* subscribing clients, which works in a similar way to captcha, but is more severe since anonymous clients who do not have any login credentials are no longer able to use the system.

Scenarios similar to the one presented here demand solutions that are able to systematically reason about the effect of adaptations on the different dimensions of concern and their trade-offs. For instance, eliminating excess traffic from potentially malicious clients has to be applied with caution, since service disruption to legitimate clients derived from the application of defensive tactics can increase user annoyance. However, rich collections of tactics and dimensions of concern can result in large solution spaces that are difficult to exhaustively reason about without proper tool support at development time, and costly in terms of computational overhead at run time for synthesis-based solutions.

In the next section, we present an overview of our proposal to improve on this situation.

3 Approach overview

Developing adaptation strategy repertoires is not an easy task for engineers, since in general it is not clear which tactic combinations are best suited to every relevant situation that can be given in a self-adaptive system. Moreover, exhaustive testing and debugging of a strategy repertoire to assess system qualities and constraint satisfaction in systems that typically exhibit large state spaces is unfeasible.

To provide software engineers with tools to help them develop strategy repetoires, we need techniques that: (i) enable the analysis of the impact of adaptations on system qualities; and (ii) facilitate the specification process by exhaustively examining which tactic combinations are better suited to different parts of the system/environment state space.

Our approach combines an architecture model of the system, tactics for manipulating the system at run time and estimates of their run-time impact on different quality attribute dimensions, as well as utility models that allow us to systematically reason about the quality of adaptations. The core of our approach consists of discretizing the system and environment state space, and synthesizing *offline* optimal adaptation plans for the different points of the discrete abstraction using probabilistic model checking (Kwiatkowska et al. 2007) (Fig. 2, top). The key idea is employing a parametric model of the system that captures properties of interest (e.g., structural, qualities) that can be used to reason about the best way to adapt. This parametric model is specified[2] based on a set of inputs (Fig. 2, top), and can be instantiated for different system/en-

[2] This specification is semi-automated. Specifically, the most repetitive and error-prone parts of the specification of the formal model, like the encoding of the utility profile and tactic impact models are automated.

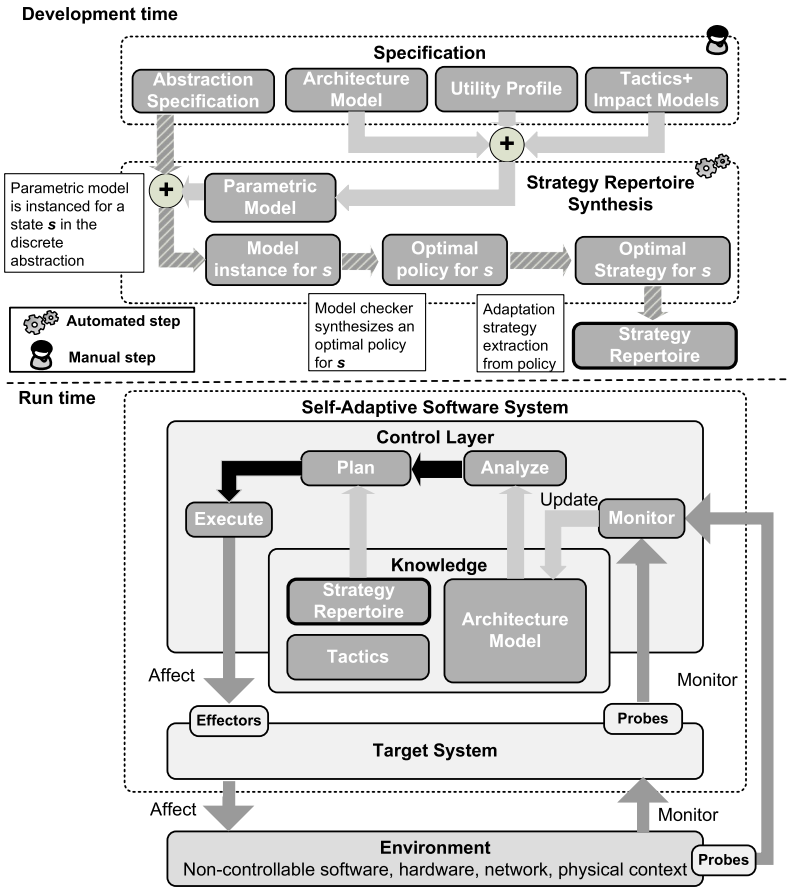


Fig. 2 Approach overview

vironment state values. Moreover, the decision of which adaptation tactic to execute is underspecified in the model as nondeterministic behavior, in such a way that we can employ a model checker to synthesize a policy (by resolving the nondeterminism) from which we can extract the sequence of tactics to execute for an optimal adaptation (with respect to the utility profile given) in different points of the state space.

In line with other architecture-based self-adaptation approaches (Sykes et al. 2010; Tajalli et al. 2010), our work adopts an architecture-centric knowledge model. Specifically, we use Stitch (Cheng and Garlan 2012), the language employed by the Rainbow framework (Garlan et al. 2004) for self-adaptation in which architecture models capture relevant properties of a system and its environment.

Our approach also fits within the general class of *sense-plan-act* architectures as embodied in the MAPE-K model (Kephart and Chess 2003), in which the different MAPE activities are instanced at run time as follows (Fig. 2, bottom): (i) during *monitoring*, observations of the system are collected, aggregated as required, and employed to update the information in the architecture model; next (ii) if the *analysis* activity

detects constraint violations in the architecture model (e.g., clients are experiencing high response time), it triggers (iii) the *planning* activity, which determines which adaptation from the repertoire (if any) should be carried out (details about the selection process are given in Sect. 4.1); finally (iii) the adaptation strategy selected in the previous phase is *executed* step-by-step, as sequences of tactics that map to system-level effectors.

In the following, we detail the knowledge model that we use in Sect. 4. Next, we introduce some basic background on probabilistic model checking in Sect. 5, and explain the technical details of our approach in Sect. 6.

4 Adaptation model

In this section, we overview Stitch adaptation models (Cheng and Garlan 2012) and illustrate some of their constructs using Znn.com.

Stitch assumes a model of adaptation that represents adaptation knowledge employing the following high-level concepts: (i) tactics, or primitive actions that correspond to a single step of adaptation; (ii) strategies, which encapsulate an adaptation process, where each step is the conditional execution of a tactic; and (iii) utility profile, which drives the selection of strategies at runtime based on a set of utility functions and preferences.[3]

Tactic A tactic is a primitive action that corresponds to a single step of adaptation. Tactics require three parts to be specified: (1) the *condition*, which specifies when a tactic is applicable; (2) the *action*, which defines the script for making changes to the system; and (3) the *effect*, which specifies the intended effect of the tactic.

Listing 1 shows an example tactic for activating a set of servers in Znn.com. Line 3 specifies the applicability condition, which says that the tactic may be executed if (i) there is a client experiencing a response above the maximum acceptable threshold (predicate cHiRespTime defined in line 1), and (ii) there are enough servers available to activate. Lines 4–7 specify the action, which is to select a set of servers among those currently inactive (line 6), and enable them (line 7). Line 9 states that the intended effect of the tactic is achieved only if all clients experience a response time below the maximum acceptable threshold.

Tactics have an associated cost/benefit impact on the different dimensions of concern in the system. Table 1 shows the impact on different properties of the tactics employed in Znn.com, as well as an indication of how the tactic affects the utility for every particular dimension of concern (the number of upward or downward arrows is directly proportional to the magnitude of utility increments and decrements, respectively).[4] While all tactics reduce the response time experienced by legitimate clients,

[3] The sources and process required to obtain the information needed to generate the artifacts required by a Stitch adaptation model are discussed in Cheng (2008).

[4] We consider fixed impacts for illustration purposes, although Stitch also supports the specification probabilistic/context-sensitive impact models (Cámara et al. 2014). Note that, to obtain the impact on the different quality dimensions of tactics in practice, the approach relies on expert knowledge, although nothing prevents the use of machine learning techniques to obtain that information [c.f. Didona and Romano (2015)].

Table 1 Tactic cost/benefit on qualities and impact on utility dimensions

Tactic	Response time (R) Δ Avg. resp. time (ms)	ΔU_R	Malicious clients (M) Δ Malicious clients (%)	ΔU_M	Cost (C) Δ Operating cost (usd/h)	ΔU_C	User annoyance (A) Δ User annoyance (%)	ΔU_A
enlistServers	− 1000	↑↑↑	0	=	+1.0	↓↓↓	0	=
lowerFidelity	− 500	↑↑	0	=	−0.1	↑	0	=
addCaptcha	− 250	↑	−90	↑↑↑	+0.5	↓↓	+50	↓↓
forceReauthentication	− 250	↑	−70	↑↑	0	=	+50	↓↓
blackholeAttacker	− 1000	↑↑↑	−100	↑↑↑	0	=	+50	↓↓
throttleSuspicious	− 500	↑↑	0	=	0	=	+25	↓

```
1
2  define boolean cHiRespTime = exists c:T.ClientT in M.components |
       c.experRespTime>M.MAX_RESPTIME;
3  tactic enlistServers (int n) {
4      condition { cHiRespTime && set.Size(s : T.ServerT in M.components | !s.isArchEnabled)>=n;}
5      action {
6          set servers = Set.randomSubset(Model.findServices(T.ServerT), n);
7          for (T.ServerT freeSvr : servers) { M.enableServer (freeSvr, true); }
8      }
9      effect { !cHiRespTime && ;set.Size(newServers())==n; }
10 }
```

Listing 1 Tactic for activating a server in Znn.com.

some of them (e.g., enlistServers and blackholeAttacker) cause a more drastic reduction, resulting in higher utility gains in that particular dimension. Regarding the presence of malicious clients, tactic blackholeAttacker is the most effective, whereas other tactics (e.g., enlistServers) do not have any impact. With respect to cost, tactic enlistServers increases the operating cost and reduces utility in this dimension, since it employs additional resources to absorb incoming traffic. Finally, all security-related tactics impact negatively on user annoyance, since there is a risk that incorrect detection of malicious clients will lead to annoying a fraction of legitimate clients (e.g., when they are blackholed or throttled).

Strategy A strategy encapsulates an adaptation process, where each step is the conditional execution of a tactic. Strategies are characterized in Stitch as trees of condition-action-delay decision nodes, where delays correspond to a time window for observing tactic effects. System feedback (via the dynamically-updated system architecture model) is used to determine the next tactic at every execution step.

```
1  strategy Outgun [cHiRespTime] {
2      t0: (cHiRespTime) –> enlistServers(1) @[30000 /*ms*/]{
3          t1: (!cHiRespTime) –> done;
4          t2: (cHiRespTime) –> lowerFidelity() @[5000 /*ms*/]{
5              t2a: (!cHiRespTime) –> done;
6              t2b: (cHiRespTime) –> fail;} } }
```

Listing 2 Strategy for absorbing excess traffic

Listing 2 shows the Stitch code for a simple adaptation strategy in Znn.com that deals with degraded performance by activating additional servers and reducing the fidelity of the contents served: line 1 specifies the applicability condition that needs to be satisfied for the strategy to be eligible for execution (in this case, predicate cHiRespTime indicates that there are clients experiencing a response time above the acceptable threshold). In the body of the strategy, node t0 (line 2) executes tactic enlistServers if the guard cHiRespTime evaluates to true. To account for the delay in observing the outcome of tactic execution in the system (settling time), t0 specifies a time window of 30 s, after which, if the tactic succeeded in driving down response time, strategy execution finishes successfully through node t1 (keyword done, line 3). Otherwise, if an acceptable reduction in response time is not observed after the delay

window expires (cHiRespTime, line 4), the strategy tries to reduce response time by executing lowerFidelity and waiting 5 s to observe its effect, exiting through node t2a if the tactic succeeds. If success of tactic lowerFidelity is not observed, the strategy exits with an error status via node t2b (keyword fail, line 6).

In the example above, we can observe that Stitch strategies assume a time window that accounts for the delay in observing the outcome of tactic execution in the system (settling time). Hence the reachability of a stable state and the fact that the environment might evolve while the tactic is executing is implicit in this time window. Stability is dealt with at a lower level in Rainbow [e.g., in gauges that collect and process information from probes, c.f. Garlan et al. (2004)]. With respect to evolution of the environment, the simplifying assumption is that the latency of the tactics (i.e., the time that it takes to observe its effects) is small, and therefore environment changes in the short term can be abstracted away. Although the practical implications of this assumption depend to a large extent on the dynamics of the target system at hand, Stitch models have been evaluated on multiple systems without showing any practical limitations derived from this assumption (Cámara et al. 2016; Cheng et al. 2009). Moreover, recent works have explored adaptation mechanisms to deal with scenarios in which such assumptions do not hold (Moreno et al. 2015, 2016).

Utility profile In Stitch, run-time strategy selection is driven by utility functions and preferences sensitive to the context of use and able to capture trade-offs among multiple potentially conflicting objectives. Qualities of concern are characterized as utility functions that map architectural properties encoding system qualities to utility values.

It is worth noting that alternative formalization styles for specifying objectives of self-adaptive systems, [and in particular, instantiated for Znn.com, e.g., using RELAX Whittle et al. (2010)], are available in the literature. However, in this case we choose our particular specification style and set of requirements because: (i) encoding requirements as sets of utility functions and weights is a natural fit that aligns well with how other approaches do run-time decision-making in self-adaptive systems (e.g., Rainbow, Descartes Adaptation Framework), and (ii) other existing instances formalizing the requirements for Znn.com do not contemplate the additional security requirements that we consider in our adaptation scenario.

We consider utility functions defined by a set of value pairs (with intermediate points linearly interpolated). Table 2 summarizes the utility functions for Znn.com. Function U_R maps low response times (up to 100 ms) with maximum utility, whereas values above 2000 ms are highly penalized (utility below 0.25), and response times above 4000 ms provide no utility. Function U_M maps lower percentages of malicious clients to higher utilities, whereas values above 70% yield no utility. Function U_C maps lower operation costs to higher utilities, whereas U_A maps higher levels of user annoyance to lower utility values. It is worth noticing that in this case, utility and mapped property values across all quality dimensions are inversely proportional, although this is not necessarily true in general.

Utility preferences capture business preferences over the quality dimensions, assigning a specific weight to each one of them. We consider three scenarios in Znn.com, whose preferences are summarized in Table 3.

Table 2 Utility functions for Znn.com

U_R	U_M	U_C	U_A
0 : 1.00	0 : 1.00	0 : 1.00	0 : 1.00
100 : 1.00	5 : 1.00	1 : 0.90	100 : 0.00
200 : 0.99	20 : 0.80	2 : 0.30	
500 : 0.90	50 : 0.40	3 : 0.10	
1000 : 0.75	70 : 0.00		
1500 : 0.50			
2000 : 0.25			
4000 : 0.00			

Table 3 Utility preferences for Znn.com scenarios

Scenario/priority	w_{U_R}	w_{U_M}	w_{U_C}	w_{U_A}
1. Minimizing malicious clients	0.15	0.6	0.1	0.15
2. Optimizing good client experience	0.3	0.3	0.1	0.3
3. Keeping cost within budget	0.2	0.2	0.4	0.2

4.1 Adaptation strategy selection

A situation demanding adaptation can generally be addressed in different ways by executing alternative strategies that may be applicable under the same run-time conditions (e.g., excess traffic can be absorbed or eliminated). Different strategies impact run-time quality attributes in various ways, thus there is a need to choose the best strategy with respect to the system's objectives. Hence, strategy selection is driven by utility functions and preferences, against which we evaluate all applicable strategies, obtaining an aggregate expected utility value for each strategy. The strategy selected is the one that maximizes its expected utility value.

The expected utility value of a strategy is obtained by: (i) computing the aggregate impact of the strategy on the system's state based on its constituent tactics [cf. Schmerl et al. (2014)], (ii) merging aggregated strategy impact with current system state to obtain the expected state after strategy execution, (iii) mapping expected state to utilities using the utility functions defined for the different concerns, and (iv) combining all utilities using utility preferences.

As an example of how the utility of a strategy is calculated, let us assume that the adaptation cycle is triggered in system state [1500, 90, 2, 0], indicating response time, percentage of malicious clients, operating cost, and user annoyance level, respectively. We focus on the evaluation of strategy Outgun.

To obtain the aggregate impact on system state of a strategy, we need to estimate the likelihood of selecting different tactics at run time due to the uncertainty in their selection and outcome within the strategy tree. To this end, we use a stochastic model of a strategy, assigning a probability of selection to every branch in the tree.[5]

[5] By default, probabilities are divided equally among the branches, although they can be progressively adjusted according to information collected from system executions. Alternatively, probabilities can also be bootstrapped based on knowledge obtained from experts or existing similar systems, when available.

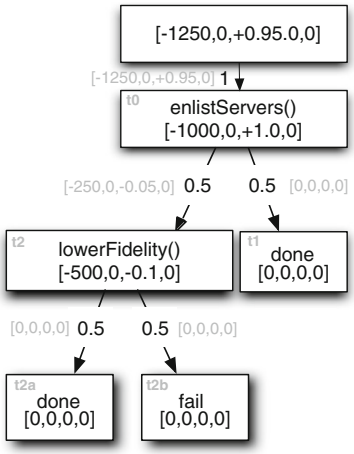


Fig. 3 Aggregate impact of strategy Outgun

Figure 3 shows how the aggregate impact on the state is computed bottom-up in the strategy tree: the aggregate impact of each node is computed by adding the aggregate impact of its children, reduced by the probability of their respective branches, with the cost-benefit attribute vector of the tactic in the node (if any).

In the example, the impact contributed by nodes t0 and t2 correspond to the cost-benefit vectors of the associated tactics, whereas leaf nodes make no changes to the system and therefore have no impact. Note that in the figure, grayed out tuples adjacent to tree branches indicate aggregate impact corresponding to the child sub-tree (including adjustments due to branch probabilities). The aggregate impact in the root node of the strategy tree results from the aggregate impacts of its children.

Once the aggregate impact of the strategy is computed, it is merged with the current system state to obtain the expected system state after strategy execution:

$$[1500, 90, 2, 0] + [-1250, 0, +0.95, 0] = [250, 90, 2.95, 0]$$

Next, we map the expected conditions to the utility space:

$$[U_R(1250), U_M(0), U_C(0.95), U_A(0)] = [0.975, 1.0, 0.11, 1.0]$$

And finally, all utilities are combined into a single utility value by making use of the utility preferences. Hence, if we assume that we are in scenario 2, the aggregate utility for strategy Outgun would be:

$$0.975^*0.3 + 1.0^*0.3 + 0.11^*0.1 + 1.0^*0.3 = 0.9035$$

Utility scores are computed similarly for all strategies. The strategy with the maximum score is selected for execution. In this case, strategies Eliminate and Outgun score 0.81 and 0.90 respectively, thus Outgun would be selected.

5 Probabilistic model checking

Probabilistic model checking (Kwiatkowska et al. 2007) is a set of techniques that enable quantitative analysis, and policy synthesis in systems that exhibit probabilistic behavior[6] guaranteed to achieve optimal expected probabilities and rewards (Kwiatkowska and Parker 2013), which are easily mapped to maximizing utility.

Our approach bases on MDPs, which describe how the state of a system can evolve in discrete time steps. In each state $s \in S$, the set of enabled actions is denoted by $A(s)$ (we assume that $A(s) \neq \emptyset$ for all states). Moreover, the choice of which action to take in every state s is assumed to be nondeterministic to let the model checker make the optimal choice. Once an action $a \in A(s)$ is selected, the successor state is chosen according to probability distribution $\Delta(s, a)$.

Definition 1 (*Markov decision process*) A Markov decision process is a tuple $\mathcal{M} = \langle S, s_I, A, \Delta, r \rangle$, where $S \neq \emptyset$ is a finite set of states; $s_I \in S$ is an initial state; $A \neq \emptyset$ is a finite set of actions; $\Delta : S \times A \to \mathcal{D}(S)$ is a (partial) probabilistic transition function; and reward structure $r : S \to \mathbb{Q}_{\geq 0}$ maps states to rewards. $\mathcal{D}(X)$ denotes the set of discrete probability distributions over finite set X.

We can reason about the behavior of MDPs using policies. A policy resolves the nondeterministic choices of an MDP, selecting which action to take in every state.

Definition 2 (*Policy*) A policy of an MDP is a function $\sigma : (SA)^*S \to \mathcal{D}(A)$ *s.t.*, for each path $\pi \cdot s$, it selects a probability distribution $\sigma(\pi \cdot s)$ over $A(s)$.

In this article, we use policies that are *memoryless* (i.e., based solely on information about the current state) and *deterministic* ($\sigma(s)$ is a Kronecker function s.t. $\sigma(s)(a) = 1$ if action a is selected, and 0 otherwise).

We can check for the existence of a policy able to guarantee that an expected reward measure meets some threshold, or optimize it, based on quantitative properties expressed in a subset of probabilistic reward computation-tree logic (PRCTL) (Andova et al. 2003). Concretely, the reward maximization operator $\mathsf{R}^{\mathsf{r}}_{\mathsf{max}=?}[\mathsf{C}]$ enables the quantification of the maximum accrued reward r along paths in a model.

The following section illustrates how these properties are used in adaptation synthesis for a given utility profile.

6 Strategy repertoire synthesis

We now present the technique that we employ to synthesize adaptation strategy repertoires. This technique is inspired by a class of approaches that employ discrete abstractions of continuous system dynamics to leverage controller synthesis techniques in the area of supervisory control of discrete-event systems (Ramadge and Wonham 1987). The approach employs probabilistic model checking of MDPs to synthesize

[6] Policies are also commonly referred to as *strategies* or *adversaries*. In this article, we employ the term *policy* consistently to avoid confusion with the term *adaptation strategy*.

optimal policies that maximize the utility for every state of a discrete abstraction of the system/environment state space. The key idea is employing a parametric model of the probabilistic behavior of the system that can be instantiated for different system/environment state values. This model contains the specification that encodes: (i) the properties of the system and the environment that are necessary to compute the value of the utility functions and check the satisfaction of the constraints imposed by the designer; and (ii) the set of tactics available in the adaptation model. Moreover, the decision of which adaptation tactic to execute is underspecified in the model as nondeterministic choices, in such a way that we can employ a model checker to synthesize a policy (by resolving the nondeterminism) from which we can extract the sequence of tactics to execute for an optimal adaptation (with respect to the utility profile and constraints given) in different points of the state space.

In the remainder of this section, we first describe the class of discrete abstraction of the state space that we employ in our approach. Then, we illustrate the encoding of the parametric model and give an overview of policy synthesis. Finally, we provide a high-level description of the synthesis and evaluation algorithms for adaptation strategy repertoires.

6.1 State space abstraction

The system/environment state space is given by a set of relevant properties characterized by a collection of n random variables $X = \{x_1, \ldots, x_n\}$ that can be monitored at run time (e.g., response time, cost). Variables taking value in non-finite domains (i.e., assume real-valued) are discretized via quantization by (i) selecting a range for the possible values of every variable, i.e., $\forall x_{i \in \{1,\ldots,n\}}$: (i) $[\alpha_i, \beta_i]$; and (ii) selecting a discretization parameter $\eta_i \in \mathbb{R}^+$ that controls the"resolution" of the variable, i.e., the granularity with which we can approximate values of the variable in the continuous spectrum.

Then, every variable x_i takes values in a discrete set of values $[\mathbb{R}]_{x_i} = \{r : \mathbb{R} \mid r = k\eta_i, k \in \mathbb{Z}, \alpha_i \leq r \leq \beta_i\}$, where $\alpha_i, \beta_i \in \mathbb{R}$ are multiples of η_i.

The variables in X define a discrete state space $[\mathcal{S}^n]_X = [\mathcal{S}]_{x_1} \times \cdots \times [\mathcal{S}]_{x_n}$, where $[\mathcal{S}]_{x_i} = [\mathbb{R}]_{x_i}$ for a real-valued variable x_i, and $[\mathcal{S}]_{x_i} = \mathcal{D}_i$ if x_i takes values in a finite domain $\mathcal{D}_i$.

At run time, given an observed value v_i of a real-valued variable x_i, the corresponding value in the discrete set $[\mathbb{R}]_{x_i}$ is obtained as:

$$d(v_i) = \arg \min_{r \in [\mathbb{R}]_{x_i}} (|v_i - r|).$$

We extend function $d : \mathbb{R} \to [\mathbb{R}]_{x_i}$ to determine the discrete system/environment state of a collection of state variable values $s = (v_1, \ldots, v_n)$ as the tuple $D(s) = (v_{d1}, \ldots, v_{dn})$, where $v_{di} = d(v_i)$ if x_i is real-valued, and $v_{di} = v_i$ otherwise. Note that when the observed value of a state falls outside of the range of variables that define the discrete state space $[\mathcal{S}^n]_X$, function D defaults to the discrete state that minimizes the distance to the observed state value.

Example 1 Let us consider that the set of variables in Znn.com $X = \{\mathsf{rt}, \mathsf{mc}, \mathsf{c}, \mathsf{ua}\}$, where the variables encode the system qualities for response time, malicious clients, cost, and user annoyance, respectively.

Consider, for instance, the property rt for response time, defined over real numbers $\mathbb{R}$. We can define the operational range of response times for the system as $[\alpha_{\mathsf{rt}}, \beta_{\mathsf{rt}}] = [0, 4000]$ ms, coinciding with the ends of the spectrum for the utility function we are considering for performance ($\mathsf{U_R}$, Table 2). Moreover, to discretize the possible values that rt can take, we define a quantization parameter $\eta_{\mathsf{rt}} = 100$ ms that controls the "resolution" of the variable. Then, we take just the multiples of η_{rt} in the given range to compute the set of discrete values of rt, i.e., $[\mathbb{R}]_{\mathsf{rt}}$, which in this case is the finite set $\{100 \cdot k \mid k \in 0 \ldots 40\}$. Notice that for other properties also taking values in $\mathbb{R}$, different value ranges and quantization parameters are required. Finally, the resulting discrete state space for our example would be given by $[\mathbb{R}]_{\mathsf{rt}} \times [\mathbb{R}]_{\mathsf{mc}} \times [\mathbb{R}]_{\mathsf{c}} \times [\mathbb{R}]_{\mathsf{ua}}$.

Next, we describe the parametric model whose initial conditions are instanced for different abstraction states.

6.2 Parametric model

To enable synthesis of optimal policies, aspects of the system and Stitch adaptation models are encoded in PRISM (Kwiatkowska et al. 2011) models parameterized by the system and environment state. Synthesizing a policy is a matter of instantiating the model with values for the system and environment state and model checking. To generate the best decision for the entire state space, we generate an instance of the model for each discrete state and model check it.

6.2.1 Formal parametric model

Our parametric model for an arbitrary system is formally specified as an extended MDP $\mathcal{M} = \langle S, s_I, A, \Delta, AP, \chi, r \rangle$, where:

- $S \subseteq [\mathcal{S}^n]_X$ is the set of states.
- $s_I \in S$ is an initial state.
- A is the set of actions available to the system.
- AP is a subset of all the predicates that can be built over the state variables (e.g., $\mathsf{underAttack}$). AP always includes two extra predicates:
  - $goal$, which is satisfied in states where the goal of adaptation is achieved (c.f. Listing 3, line 7).
  - end, which labels explicitly all absorbing states of the model (for a state $s \in S$, we say that $s \models end$ iff $\forall s' \in S, a \in A \bullet \Delta(s, a)(s') \neq 0 \Leftrightarrow s = s'$). In practice, we can encode this predicate in the model as $end \equiv \bigwedge_{a \in A} C_a \vee goal$, where C_a is the conjunction of architectural constraints that limit the applicability of tactic a . Hence, this predicate states that either the goal has been achieved, or no action can be executed.
- $\chi : S \to 2^{AP}$ is a labeling function.
- r is a reward structure labeling goal states with their associated utility, computed based on the preferences defined in the utility profile. Specifically, the reward of

an arbitrary state s is defined as $r(s) = \sum_{i=1}^{q} w_i \cdot u_i(v_i^s)$ if $s \models goal$ ($r(s) = 0$ if $s \not\models goal$), where u_i and $w_i \in [0, 1]$ are the utility function and weight for quality dimension $i \in \{1, \ldots, q\}$, respectively. v_i^s is the value that the state variable associated with the architectural property representing quality attribute i (e.g., response time, percentage of malicious clients) takes in state s.

We illustrate the pattern used to encode our parametric model for an arbitrary system using a high-level specification[7] of the behavior followed by the adaptation layer of the self-adaptive system. We model the system as a set of commands that encode the set of actions available in A, which corresponds to the set of available tactics in the adaptation model. More specifically, each action $a \in A$ is encoded in a command:

$$[a]C_a \wedge \neg goal -> p_a^1 : post_a^1 + \cdots + p_a^n : post_a^n$$

Where the guard includes: (i) the conjunction of architectural constraints that limit the applicability of tactic a (abstracted by C_a, e.g., a new server cannot be activated in Znn.com's pool if all of them are already active, Listing 3, line 1), and (ii) a predicate $\neg goal$ to avoid expanding the state space beyond the satisfaction of the adaptation goal.

The execution of a command can result in a number of alternative updates on state variables (along with their assigned probabilities p_a^i) that correspond to the different probabilistic outcomes of a given tactic execution instance, denoted by $post_a^i$ (e.g., the activation of a specific server can result in a success with probability p, and fail with $1 - p$).

Finally, an additional command in the controller introduces absorbing states that succeed every state satisfying *end* (in the encoding below, $true$ is short-hand for no modification on any state variable):

$$[\,]\ end -> true$$

In the remainder of this section, we show how these models are encoded in practice using the PRISM probabilistic model checker, using our running example to illustrate the process.

6.2.2 Parametric model specification

Listings 3 and 4 give examples of the encodings for the Znn.com example. The parametric model consists of the following parts:

[7] We illustrate our approach to modeling the parametric model using the syntax of the PRISM language (Kwiatkowska et al. 2011) for Markov Decision Processes (MDPs), which are encoded as commands:

$$[action]guard -> p_1 : u_1 + \cdots + p_n : u_n$$

Where $guard$ is a predicate over the model variables. Each update u_i describes a transition that the process can make (by executing $action$) if the guard is true. An update is specified by giving the new values of the variables, and has an assigned probability $p_i \in [0, 1]$. Multiple commands with overlapping guards (and probably, including a single update of unspecified probability) introduce local nondeterminism.

System The system is encoded as a collection of variables, representing the quality properties of interest and simplified aspects of system structure. In Listing 3, lines 10–14 show the variables of interest for our running example. The qualities related to system objectives are represented in variables rt, mc, and ua, for response time, malicious clients, and user annoyance. Structural properties like number of active servers and level of fidelity are captured in as and fi, respectively. Note that all these variables are bounded for model checking purposes (constants MAX_*, MIN_*, and INIT_* represent the upper bound, lower bound, and initialization value of variable *).

Tactics Available tactics are encoded as commands that model the effect of executing the different adaptation tactics as updates on the different system/environment variables. Every command includes: (i) a guard that encodes the tactic's applicability condition, e.g., there must be inactive servers in the system to apply enlistServers (lines 1, 20); and (ii) a set of updates in which variable values change based on cost/benefit attribute vectors (described in Sect. 4), which are encoded as formulas in the model (e.g., the update in response time resulting from executing tactic enlistServers is encoded in line 8, and employed on the right-hand side of the command in line 20). Note that variables representing properties not affected by the execution of a tactic do not appear in the list of updates encoded in the command (e.g., enlisting a server does not affect annoyance or the percentage of malicious clients in the system).

Together, the system and tactic encodings form a module in the PRISM model that describes the effects of executing the different tactics on the properties of interest in the system. An additional command in this module guarded by the predicate end defined in line 23 is employed to explicitly capture the utility reward of states after the execution of the adaptation, in which no further adaptation tactics are available for execution, or the objective of adaptation has been achieved. In this case, the objective is defined in line 12, where perfViolation $=$ rt $>$ 1000 and underAttack $=$ mc $>$ 50 are predicates that characterize high response time and high level of malicious clients, respectively.[8] In fact, these predicates are defined as perfViolation $\equiv$ rt $>$ MAX_RESPTIME, underAttack $\equiv$ mc $>$ MAX_MC, where rt and mc are the variables encoding the system qualities for response time and malicious clients, and MAX_RESPTIME and MAX_MC are their respective acceptable thresholds.

Utility profile Utility functions and preferences are encoded using formulas and reward structures that map different model states to a utility value. Formulas compute utility on the different dimensions of concern, and reward structures weigh them against each other by using the utility preferences of a given scenario.

Listing 4 illustrates in lines 1–5 the encoding of utility functions using a formula for linear interpolation based on the points defined for utility function U_M in the second column of Table 2. Lines 6–10 show how a reward structure can be defined to compute a single utility value for any state by using the utility preferences defined for a particular scenario (Scenario 1, Table 3). The reward structure considers only the rewards in model states that correspond to states in which no applicable tactics remain available (characterized by predicate end, defined in Listing 3, line 5).

[8] Thresholds are defined based on expert knowledge, and are analogous to the ones found in existing Stitch models for Znn Cheng and Garlan (2012).

```
formula ac_es = MAX_AS − as > 0; // Tactic applicability conditions (as = active servers)
formula ac_bh = mc > MIN_MC;
...
// Cost/Benefit Attribute vectors for tactic enlistServers
formula cb_es_rt = rt−1000 >=MIN_RT ? (rt−1000<=MAX_RT? rt−1000 : MAX_RT) : MIN_RT;
formula cb_es_as = as+1<= MAX_AS ? as+1 : as;
// Cost/Benefit Attribute vectors for tactic blackholeAttacker
formula bh_f_rt = rt−1000 >=0 ? (rt−1000<=MAX_RT? rt−1000 : MAX_RT) : 0;
formula bh_f_mc = mc−100 >=0 ? (mc−100<=MAX_MC? mc−100 : MAX_MC) : 0;
formula bh_f_ua = ua+50 >=0 ? (ua+50<=100? ua+50 : 100) : 0;
...
formula goal = !(perfViolation|underAttack);
formula end = goal | (!goal &!ac_es & ... & !ac_bh);
module arch
  rt : [MIN_RT..MAX_RT] init INIT_RT; // Response time
  as : [MIN_AS..MAX_AS] init INIT_AS; // Active servers
  fi : [MIN_FI..MAX_FI] init INIT_FI; // Fidelity
  mc : [MIN_MC..MAX_MC] init INIT_MC; // Malicious clients
  ua : [MIN_UA..MAX_UA] init INIT_UA; // Annoyance
  [enlistServers] ac_es −> (rt'=cb_es_rt) & (as'=cb_es_as);
  [blackholeAttacker] ac_bh −> (rt'=cb_bh_rt) & (mc'=cb_bh_f_mc) & (ua'=bh_f_ua);
  ...
  [] end −> true;
endmodule
```

Listing 3 Target system module specification

```
formula uM = (mc>=0 & mc <=5? 1:0)
  +(mc>5 & mc <=20? 1+(0.80−1)∗((mc−5)/(20−5)):0)
  +(mc>20 & mc <=50? 0.80+(0.40−0.80)∗((mc−20)/(50−20)):0)
  +(mc>50 & mc <=70? 0.40+(0.00−0.40)∗((mc−50)/(70−50)):0)
  +(mc>70 ? 0:0); ...
rewards "U" // Utility reward structure
  end & scenario=1 : 0.15∗uR +0.6∗uM +0.1∗uC +0.15∗uA; ...
endrewards
```

Listing 4 Utility reward structure

6.3 Policy synthesis

For an MDP model like the one described above, we can employ a model checker like PRISM to automatically synthesize a policy (see Definition 2) that satisfies one or multiple objectives captured in a temporal logic formula (Forejt et al. 2012). This capability enables our approach to optionally impose multiple designer constraints on the policies synthesized, and thereby in the strategy repertoire resulting from them.

Example 2 We want to synthesize an adaptation strategy repertoire that: (i) guarantees the system to stay in the region of the state space in which it is operating with an acceptable performance and level of malicious clients, and (ii) optimizes the solution for the utility profile encoded into the utility reward structure U described in Listing 4.

We can write a property to synthesize a policy for an MDP tailored to the objectives described above:

$$\text{multi}(R^{U}_{\max=?}[C],\ P_{\geq 1}[F!(\text{perfViolation}|\text{underAttack})])$$

The query above: (i) checks if there exists a policy that can achieve the combination of objectives expressed inside of the P operator, i.e., whether the policy can guarantee

the eventual satisfaction of an acceptable performance and level of malicious clients; and (ii) among all the possible policies that satisfy (i), if any, checks which is the one that maximizes the value of the utility reward U.

```
1 strategy lo13bl17[perfViolation | underAttack] {
2   t0: (perfViolation | underAttack) –> lowerFidelity () @[5000 /*ms*/] {
3     t1: (true) –> blackholeAttacker () @[10000 /*ms*/] {
4       t1s: (!(perfViolation | underAttack)) –> done;
5       t1f: (perfViolation | underAttack) –> TNULL;} } }
```

Listing 5 Synthesized Stitch strategy

If we run the synthesis process for the point [INIT_RT =2000, INIT_MC =70, INIT_AS =2, INIT_UA =0, INIT_FI =1] in Scenario 1, and extract the sequence of tactics from it, we obtain lowerFidelity-blackHoleAttacker. The automatic translation of this sequence into Stitch results in the strategy shown in Listing 5. A predicate encoding the constraints imposed in the PRCTL property is used in the strategy's applicability condition and in nodes t1s/t1f to indicate the success/failure status of the execution. Moreover, the sequence of tactics extracted from the policy are encoded as nested commands, where the next tactic is executed after a fixed time window used for observing the effect of the previous tactic. Time window values are manually specified by developers per tactic, based on experimental observations.

Obtaining the policy for each state in the discrete abstraction described in Sect. 6.1 requires an independent run of the model checker in which model parameters are instantiated with variable values corresponding to that state.

In the next section, we describe the algorithm that extends the synthesis of local policies to a process that encompasses the synthesis of strategy repertoires.

6.4 Strategy repertoire synthesis algorithm

We describe in this section our algorithm for synthesizing adaptation strategy repertoires. The algorithm sweeps a discrete abstraction of the state space, synthesizing policies for different points of the abstraction that: (i) are optimal with respect to a utility profile, and (ii) satisfy a (potentially empty) set of additional constraints. If the set of constraints is not satisfied for a subset of the points in the discrete abstraction, the algorithm attempts to synthesize a less constrained version of the policy by progressively relaxing the constraints imposed by the designers.

Algorithm 1 describes the synthesis process, which receives as input a discrete abstraction of the area of the state space to explore $[\mathcal{S}^n]_X$, the set of adaptation tactics T (including a specification of their impact in system state), a utility profile U, and an architecture model A that includes the properties of the system and the environment required to determine system qualities and constraint satisfaction.

Generation of the parametric model used for synthesis (cf. Sect. 6.2) is captured by *generate_model* (line 2), which abstracts the process followed to obtain the model and can range from manual specification, to full synthesis from architecture models

Algorithm 1 Adaptation Strategy Repertoire Synthesis

Input: state space abstraction $[\mathcal{S}^n]_X$, architecture model A, adaptation tactics T, utility profile U, set of constraints C.
Output: adaptation strategy repertoire R.

```
1: R ← ∅
2: model ← generate_model(A, T, U, C)
3: for all s ∈ [S^n]_X do
4:     model_s ← instantiate_model(model, s)
5:     C_s ← C; strategy_s ← [ ]
6:     while strategy_s = [ ] ∧ C_s ≠ ∅ do
7:         σ_s ← generate_policy(model_s, C_s)
8:         strategy_s ← extract_strategy(σ_s)
9:         if strategy_s = [ ] then
10:            C_s ← relax(C_s)
11:        end if
12:    end while
13:    if strategy_s ∉ R then
14:        R ← R ∪ {strategy_s}
15:    end if
16: end for
17: return R
```

and tactics. Our experience has shown that full automation of this step is possible, although human assistance is desirable for optimizing the resulting model.

The algorithm iterates over the state space abstraction $[\mathcal{S}^n]_X$, instantiating the model for every point s in the discrete set (line 4), and generating a policy for it (lines 6–12).[9] Initially, the algorithm attempts to generate a policy for the full set of constraints C, but it progressively relaxes them if no policy that satisfies the full set is found (line 10). Relaxation of constraints at run time (e.g. for graceful service degradation) is a technique commonly employed by different approaches in self-adaptation (D'Ippolito et al. 2014; Whittle et al. 2010). Our algorithm is agnostic with respect to the details of the specific constraint relaxation process. Hence, in the algorithm function $relax$ is left unspecified on purpose, since it might be implemented in different ways, ranging from modifications on acceptable thresholds in system qualities (e.g., increasing the threshold of acceptable response time by a given amount), to prioritized constraint removal (e.g., removing the constraint about malicious clients to synthesize a controller that only guarantees the response time to legitimate clients).

6.5 Strategy repertoire evaluation algorithm

At run time, the current state s of the system that requires adaptation may or may not belong to the set of states in the discrete abstraction that has been used to synthesize the repertoire of strategies. In the latter case, the strategy selected for execution in s will be the one synthesized for the closest state to s in the abstraction (i.e., $D(s)$, c.f. Sect. 6.1). Since the final result of executing the selected strategy with s and $D(s)$ as initial states might differ, we need to quantify that difference across the state space that we are studying to obtain an indication of the actual quality of the repertoire of strategies synthesized.

[9] Function $extract\_strategy$ abstracts the extraction of a sequence of actions from a policy, which is trivial (see Definition 2).

To achieve such quantification, we provide an algorithm that engineers can use to systematically assess repertoires and predict whether their effects comply with their design intentions. Repertoires are analyzed on a *maximal abstraction* of the state space $[\mathcal{S}^n]^*_X$ that we consider as ground truth, and whose resolution can be determined, e.g., based on the maximum resolution of available probes (thus guaranteeing that any state that the system can measure at run time belongs to the maximal abstraction). Algorithm 2 returns three maps built over that abstraction that indicate: (i) how strategies are selected from repertoire R across the state space according to utility profile U, using the mechanism shown in Sect. 4.1 (abstracted by $select\_strategy$), (ii) what is the impact of those selections in utility, and (iii) which constraints are violated in what parts of the space.

Algorithm 2 Adaptation Strategy Repertoire Analysis

Input: maximal state space abstraction $[\mathcal{S}^n]^*_X$, utility profile U, set of constraints C, adaptation strategy repertoire R.
Output: strategy selection map SM, utility impact map IM, constraint violation map CM.

```
1: SM ← ∅; IM ← ∅; CM ← ∅
2: for all s ∈ [S^n]*_X do
3:     strategy_s ← select_strategy(s, U, R)
4:     SM ← SM ∪ {(s, strategy_s)}
5:     IM ← IM ∪ {(s, u'(s) − u(s))}
6:     for all c ∈ {c : C | ¬satisfies(s, strategy_s, c)} do
7:         CM ← CM ∪ {(s, c)}
8:     end for
9: end for
10: return SM, IM, CM
```

In the algorithm, functions u and u' abstract evaluation of utility for the pre- and post- states of strategy execution according to a utility reward structure like the one described in Listing 4, on a model in which the selected strategy has been fixed. Similarly, $satisfies$ abstracts the evaluation of constraint satisfaction on a model in which a strategy is executed on a given state. All these functions are based on PRCTL property checks in our implementation (e.g., in our example, u' is computed based on the PRCTL check of the formula $\mathsf{R}^{\mathsf{U}}_{\mathsf{max}=?}[\mathsf{C}]$ described in Example 2, whereas u can be checked as the instantaneous reward in the initial state $\mathsf{R}^{\mathsf{U}}_{\mathsf{max}=?}[\mathsf{I}{=}0]$).

7 Evaluation

In this section we evaluate our approach according to the following criteria:

- *Quality of adaptation choice* We assess whether adaptation choices are consistent with engineer intentions and quantify their impact with respect to achieving system goals and adequately trading-off extra-functional concerns.
- *Synthesis time/quality trade-offs* We study the influence of the size/resolution of the abstractions on synthesis time, and its trade-offs with the quality of the solution.

The novelty of our contribution resides in the offline strategy synthesis process, therefore we restrict the scope of our evaluation to it. In any case, run-time performance of the Rainbow framework has been evaluated in different settings (Cámara

et al. 2013; Cheng et al. 2009; Schmerl et al. 2014), including an industrial-scale middleware (Cámara et al. 2013) and the security scenario described in this article (Schmerl et al. 2014), in which adaptation driven by a strategy repertoire written by engineers was shown to be consistent with static analysis results based on tactic impact models. In our study, we assume proper calibration of tactic impact models, which is an orthogonal issue to the topics discussed.

We employ as setup a prototype implementation of MOSAICO running PRISM-4.3.beta-osx64 as a back-end on an Intel Core i7 2.8 GHz with 16 GB RAM.

7.1 Quality of adaptation choice

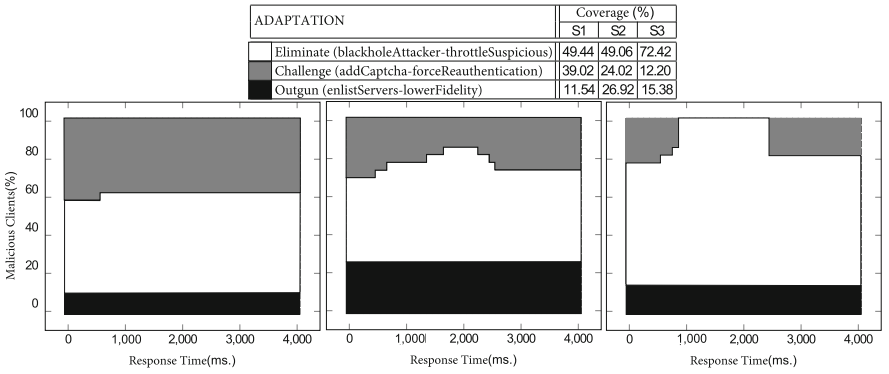
To assess the quality of adaptation choice, we statically analyze a representative region of the system/environment state space. We focus on: (i) consistency between the engineer's design intentions (as encoded in utility profiles) and the strategy repertoires synthesized, and (ii) impact of the adaptation repertoires on the achievement of system goals.

We compare strategy repertoires synthesized by our approach with a strategy repertoire manually specified by engineers presented in Schmerl et al. (2014), which includes the strategies: (i) Outgun, which employs the tactics enlistServers and lowerFidelity to absorb excess traffic, (ii) Eliminate, aimed at eliminating excess traffic employing the tactics blackholeAttacker and throttleSuspicious, and (iii) Challenge, which aims also at eliminating the excess traffic by employing the less aggressive tactics reauthenticateAll and addCaptcha.

The set of strategy specifications employed in this comparison were developed by a group of researchers (some of them experts in the area of security), and were discussed and refined over the course of several weeks of joint work meetings. These involved defining and developing mainly: (i) stitch tactic code (approx. 30% of overall effort), (ii) stitch tactic cost/benefit impact vectors (approx. 20% of overall effort), and (iii) stitch strategy code (approx. 50% of overall effort).

Note that for our setup, the development of the target system (Znn), architecture model (components, properties, etc), and most system effectors and probes were reused from previous implementations. Moreover, the set of manual steps required to develop the adaptation logic with our offline synthesis approach also required artifacts (i) and (ii) described above, but not (iii), which amounts to approximately half of the effort. Instead, our approach requires semi-automated specification of the parametric PRISM model, whose effort is counted in hours (less than one day for our case study), rather than the days that were required to implement the Stitch strategy code described as item (iii) above.

Adaptation choice Figure 4 shows strategy choice results for the different scenarios in a region of the state space projected over the dimensions corresponding to malicious clients and response time (restricted to interval [0, 4000] ms). Cost, user annoyance, and fidelity have an initially fixed value of 2 USD/h, 0% and 1(high), respectively. The discretization parameters used for response time and malicious clients are $\eta_{\mathsf{rt}} = 100$ ms and $\eta_{\mathsf{mc}} = 5$.



ADAPTATION	Coverage (%)		
	S1	S2	S3
Eliminate (blackholeAttacker-throttleSuspicious)	49.44	49.06	72.42
Challenge (addCaptcha-forceReauthentication)	39.02	24.02	12.20
Outgun (enlistServers-lowerFidelity)	11.54	26.92	15.38

ADAPTATION	Coverage (%)		
	S1	S2	S3
No Adaptation	14.05	14.05	14.05
lowerFidelity	2.32	5.23	6.39
lowerFidelity-enlistServers	1.16	3.48	0.58
enlistServers-lowerFidelity	2.90	5.23	2.32
enlistServers-lowerFidelity-blackholeAttacker	28.46	37.17	17.42
lowerFidelity-blackholeAttacker	40.65	32.52	40.65
lowerFidelity-throttleSuspicious	-	-	1.16
lowerFidelity-throttleSuspicious-blackholeAttacker		-	11.61
blackholeAttacker	4.07	0.93	-
addCaptcha-enlistServers	-	0.23	-
enlistServers-lowerFidelity-throttleSuspicious	0.58	1.16	-
lowerFidelity-throttleSuspicious-forceReauthentication-blackholeAttacker	5.81	-	5.81

Fig. 4 Adaptation choice: offline synthesis (bottom) and manual specification (top)

For strategy synthesis (Fig. 4, bottom) we employed the property shown in Example 1 in which: (i) the utility reward is maximized according to the utility profile defined for the scenario, and (ii) predicates perfViolation and underAttack impose additional constraints on the solution.

The range of adaptations obtained through strategy synthesis is richer than the one specified manually (Fig. 4 top). This is because engineers packaged tactics in strategies according to their intended target (e.g., tactics that aim at absorbing excess traffic like enlistServers are never used in combination with tactics targeting malicious clients). In contrast, synthesis systematically considers situations in which combining tactics that have different targets can provide an optimal solution in a given region of the state space (e.g., employing lowerFidelity with blackholeAttacker is an optimal combination for 32–40% of the states across all scenarios). Moreover adaptation choices are consistent

Table 4 Impact on utility for different scenarios

Scenario/priority	Avg. ΔU		
	Synthesis	Manual	Δ(%)
1. Minimizing malicious clients	0.4853	0.1972	59.37
2. Optimizing good client experience	0.4539	0.2173	52.13
3. Keeping cost within budget	0.4125	0.1448	64.89

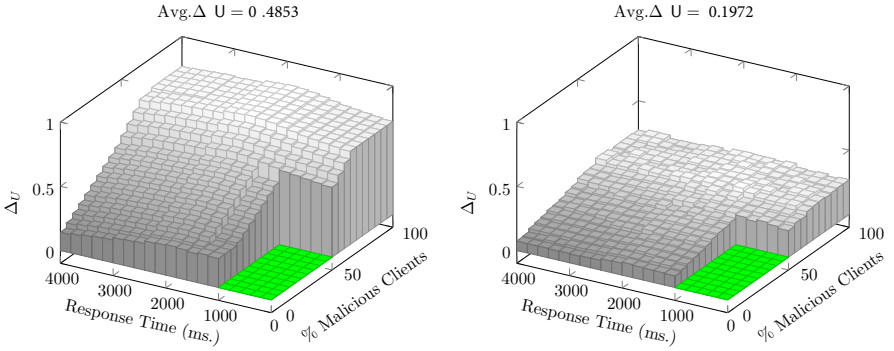


Fig. 5 Impact on utility of synthesized (left) and manually specified repertoires (right) for Scenario 1

with the utility profiles specified and the characteristics of the tactics. In Scenario 1, priority is given to eliminating malicious clients, and the combinations of tactics that are employed within the same strategy (e.g., throttleSuspicious, reauthenticateAll, and blackholeAttacker) are more aggressive than those found in Scenario 2, in which priority is given to optimizing good client experience, where at most one security-related tactic is employed in every strategy. In Scenario 3, where cost is the main concern, expensive tactics such as enlistServers are restricted to 20% of the state space, compared to 33% and 50% in scenarios 1 and 2, respectively.

Adaptation impact Analyzing the impact of the adaptation choices on utility entails calculating the difference between instantaneous utility (ΔU) before and after the execution of the selected adaptation strategies across a region of the state space (see IM, Algorithm 2).

Table 4 shows that synthesis consistently achieves a remarkable improvement (52–64%) over the manually-specified repertoire across all scenarios, independently of the differences in utility preferences and the adaptation choices made.

Figure 5 illustrates the contribution to utility in different areas of the state space: the impact of adaptations in areas that correspond to a high percentage of malicious clients significantly improve over manually-specified adaptations. In contrast, synthesis does not improve much over the manual approach in areas with low percentage of malicious clients, even in case of high response time. This is consistent with the priority of the scenario encoded in the utility profile, which weighs as more relevant eliminating malicious clients.

Synthesis Abstraction $[\mathcal{S}^n]_X$				**Scenario 1 - analyzed on** $[\mathcal{S}^n]^*_X$, $\eta_{\mathsf{rt}} = 10$, $\eta_{\mathsf{mc}} = 1$			
Resolution	η_{rt}	η_{mc}	# States	Time (s)	# Strategies	Avg. ΔU	Constr. Sat. (%)
Low	1000	50	15	38	5	0.4051	100
Med-low	500	20	54	134	7	0.4728	100
Med-high	200	10	231	584	7	0.4781	100
High	100	5	861	2130	8	0.4853	100
Synthesis Abstraction $[\mathcal{S}^n]_X$				**Scenario 2 - analyzed on** $[\mathcal{S}^n]^*_X$, $\eta_{\mathsf{rt}} = 10$, $\eta_{\mathsf{mc}} = 1$			
Resolution	η_{rt}	η_{mc}	# States	Time (s)	# Strategies	Avg. ΔU	Constr. Sat. (%)
Low	1000	50	15	36	4	0.3849	100
Med-low	500	20	54	132	5	0.4375	100
Med-high	200	10	231	565	7	0.4468	100
High	100	5	861	2104	8	0.4539	100
Synthesis Abstraction $[\mathcal{S}^n]_X$				**Scenario 3 - analyzed on** $[\mathcal{S}^n]^*_X$, $\eta_{\mathsf{rt}} = 10$, $\eta_{\mathsf{mc}} = 1$			
Resolution	η_{rt}	η_{mc}	# States	Time (s)	# Strategies	Avg. ΔU	Constr. Sat. (%)
Low	1000	50	15	37	6	0.3341	100
Med-low	500	20	54	131	7	0.3987	100
Med-high	200	10	231	566	8	0.4042	100
High	100	5	861	2107	8	0.4125	100

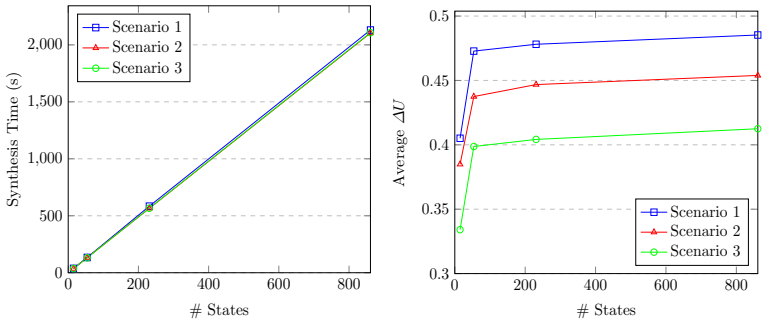


Fig. 6 Time/utility impact of synthesized strategy repertoires

7.2 Synthesis time/quality trade-offs

To assess time/quality trade-offs, we: (i) measure synthesis time for repertoires obtained from abstractions with different resolutions, and (ii) statically analyze the state space for every repertoire to measure their impact on utility.

Figure 6 shows the results for increasingly finer resolutions across adaptation scenarios. As expected, finer resolutions result in larger state spaces, longer computation time, and higher utility increments. However, even low resolution doubles the improvement in utility of the manual approach across all scenarios, with computation time under 40 s.

Interestingly, *moderate increments in resolution yield substantial improvements in utility* in the range 10–15% with computation times of approximately 2 min. In contrast, going from medium–low to higher resolutions increases time substantially and results in marginal utility improvements. This linear increment of computation time vs. logarithmic increment on utility impact on the size of the abstraction (Fig. 6, right), is explained because strategies that are optimal in large regions of the state space (and therefore have a greater impact on overall utility, e.g., lowerFidelity-blackholeAttacker in all scenarios of Fig. 4) are more likely to be discovered with coarser abstraction

resolutions than strategies which are optimal only in small regions and have a very limited impact in utility. This is shown in the table on the left of Fig. 6, in which medium-low abstraction resolutions are enough to uncover most of the strategies for the repertoires synthesized with higher resolutions (e.g., 7 out of 8 for scenarios 1 and 3). Consistent with this finding, our data shows that strategies that are optimal in very restricted areas of the state space (e.g., addCaptcha-enlistServers) can only be uncovered with high resolution. Although more experiments are needed to generalize, these observations appear to respond to the 80–20 pareto principle or *vital few, trivial many* that holds in other areas of software (Pressman 2001).

Concerning the satisfaction of the constraint in Example 1, in our study, MOSAICO always finds solutions that meet it, although we do not provide strict formal guarantees about constraint satisfaction (this issue is discussed in Sect. 8).

Finally, in our study we run policy synthesis sequentially for every point of the abstraction provided as input. However, different regions of the abstraction can be distributed across different computation nodes to parallelize the synthesis process, facilitating the scalability of the approach.

8 Threats to validity

Regarding internal validity, the main concern is related to the actual satisfaction of constraints and strategy optimality criterion, since guarantees for them in an abstraction state $D(s) \in [\mathcal{S}^n]_X$ (see Sect. 6.1) do not necessarily hold in s. However, static analysis of adaptation strategies generated using different resolution abstractions shows that in practice, utility values reasonably approximate the optimal solution obtained with very high resolutions (and hence, we employ in this article the term *nearly-optimal* to qualify the repertoires generated). In that sense, note that the values shown on Table 4 are obtained via static analysis (Algorithm 2) iterating over states in the *actual* state space (i.e., the maximal abstraction $[\mathcal{S}^n]^*_X$), rather than over the states in the abstraction $[\mathcal{S}^n]_X$ employed for synthesis. In the same line, although we do not provide formal guarantees concerning constraint satisfaction in the actual state space a priori, the fact that our proposal is chiefly intended for design time mitigates this issue. In fact, the intent of our approach is supporting engineers in achieving the best possible results with less effort during the specification process, providing them with useful feedback about the satisfaction of constraints, and giving them the opportunity to revisit the strategy repertoire to fix any potential issues if needed (e.g., by reworking tactics) before deployment. Concerning the cost of static analysis captured in Algorithm 2, it is negligible by comparison with the cost of synthesis, since PRCTL is amenable to verification via statistical model checking once a strategy has been fixed in the model.

A second issue concerns the need to recompute the strategy repertoire when tactics, utility functions, or preferences change. However, the timescale at which changes to these elements occur in practice is likely to be much larger than the time needed to automatically recompute a new repertoire.

Finally, while in this article we do not show the probabilistic aspects of the approach for space and clarity concerns, uncertainty is inherent to self-adaptive systems and justifies the use of probabilistic model checking, along with the capability of quantitative

analysis. While everything indicates that the approach can scale for fairly large system/environment state spaces, problem instances with very large tactic collections or probabilistic outcomes might threat scalability, since they increase the local MDP state space built to solve the synthesis problem in each state of the abstraction. In that sense, the fact that abstractions can be distributed across different computation nodes mitigates this issue. Moreover, results in Cámara et al. (2015) show that local MDP synthesis including probabilistic aspects scales well for architectures of 200+ components, with computation times that are not far off from those measured in the deterministic case.

Concerning external validity, our evaluation scope is limited to a specific framework (Rainbow), specification language (Stitch), and class of system. However, we believe the results generalize to other systems in the MAPE-K family that include similar concerns and planning activities to the ones described in this work. Concretely, nothing prevents using MOSAICO in other application domains in which Rainbow has been assesed (e.g., industrial middleware Cámara et al. 2013), or adaptation platforms like the run-time piece of the Descartes Modeling Language, which does run-time strategy selection in a similar fashion and has been applied to e.g., grid computing or service-oriented environments (Huber et al. 2014; Nou et al. 2009).

9 Related work

Our work draws from several areas of research, such as supervisory control (Piterman et al. 2006), planning (Giunchiglia et al. 1999; Schoppers 1987), and probabilistic model checking (Kwiatkowska et al. 2007). Although we cannot give an exhaustive list, we outline here some of the related work.

Some approaches closely related to ours employ model checking or planning tools to devise adaptation behavior via online synthesis. Tajalli et al. (2010) introduce PLASMA, an approach that uses plan-based and architecture-based mechanisms for model-driven adaptation via model checking. The approach is able to derive plans for a given goal and initial state provided by the user, although the quality of the solution is not considered. In contrast, Sykes et al. present in Sykes et al. (2010) two variants of an approach for the assembly of component configurations that uses non-functional information to guide the synthesis. One variant guarantees optimal solutions, but it is costly, since in the worst case it needs to generate the full list of candidate solutions to rank them. A second variant is a greedy algorithm that lowers computation cost by eliminating guarantees about the solution.

Other approaches employ probabilistic model checking to deal with uncertainty are more limited in the mechanisms used to adapt the system (e.g., parameter optimization, or instantiation of predefined workflows) (Calinescu et al. 2011; Calinescu and Kwiatkowska 2009; Epifani et al. 2009).

Complex adaptation behavior can also be synthesized *offline* to be reused at run time. D'Ippolito et al. (2014) present a formal adaptation framework that allows the system to fall back onto a graceful degradation of the service when environment assumptions are broken. Carzaniga et al. (2013) present an approach that synthesizes behavior to recover from run-time failures. Li et al. (2014) present an approach to synthesize

behavior that automatically hands over control to a human supervisor when the control layer cannot handle a problem. All of these approaches eliminate run-time overhead and focus on functional behavior.

Concerning offline approaches that go beyond functional behavior, Marco et al. (2013) focus on interoperability and use behavioral synthesis to generate connectors that meet both functional and performance concerns. Finally, works by Filieri et al. (2014, 2015) and Klein et al. (2014) propose methodologies that target tunable variables and are grounded in control theory to produce control strategies for adaptive systems with formal guarantees. In contrast with our approach, the proposals described in Filieri et al. (2014) and Klein et al. (2014) focus on a single adaptation mechanism, whereas the approach in Filieri et al. (2015) can handle the satisfaction of multiple objectives, but does not explicitly consider trade-offs among them. However, the works described in Filieri et al. (2014, 2015) are shown to enable the design of controllers that can trade off responsiveness and robustness.

10 Conclusions and future work

In this article we have presented MOSAICO, an approach to offline synthesis of adaptation strategy repertoires in MAPE-K systems. The approach supports formal reasoning about trade-offs among extra-functional system properties, and our results have demonstrated that the class of abstractions that we employ enables analyzability, as well as flexibility in making trade-offs between quality and computation cost. Moreover, results show that computation time increases linearly with resolution, while utility does so logarithmically, enabling us to obtain repertoires that reasonably approximate the optimum with a relatively low computation cost. The overall goal of the approach is improving quality while reducing developer costs associated with strategy specification by automatically generating strategy repertoires. Hence, although synthesis is done offline, the ability to trade quality by computation time is relevant because of its direct impact on monetary cost (e.g., when using public cloud infrastructure to distribute the synthesis of repertoires across different computation nodes). While our results are only first steps, we believe that they have relevant implications for automating future self-adaptive systems development.

Concerning future work, we aim at using multi-scale abstractions that can be progressively refined in regions in which higher resolutions are needed, enabling smarter, more efficient schemes for exploring the state space. Moreover, we intend to extend our abstractions to model time explicitly (e.g., adaptation tactic latency), and investigate applicability in other domains and self-adaptation frameworks. Finally, we will also explore combining machine learning techniques [e.g., Didona and Romano (2015)] with our approach to extend the degree of automation of the approach and obtain tactic impacts automatically, instead of having them manually specified by developers.

Acknowledgements This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. [Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution (DM-0004612).

References

Andova, S., Hermanns, H., Katoen, J.: Discrete-time rewards model-checked. In: Larsen, K.G., Niebert, P. (eds.) Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS 2003, Marseille, France, 6–7 September 2003. Revised Papers, Volume 2791 of Lecture Notes in Computer Science, pp. 88–104. Springer, Berlin (2003)

Beheshti, S.M.R.S., Liatsis, P.: Captcha usability and performance, how to measure the usability level of human interactive applications quantitatively and qualitatively? In: 2015 International Conference on Developments of E-Systems Engineering (DeSE), pp. 131–136 (2015)

Bulletin Board Benchmark. http://jmob.ow2.org/rubbos.html

Calinescu, R., Kwiatkowska, M.Z.: Using quantitative analysis to implement autonomic IT systems. In: Atlee, J.M., Inverardi, P. (eds.) Proceedings of the 31st International Conference on Software Engineering, ICSE 2009, 16–24 May 2009, Vancouver, Canada, pp. 100–110. IEEE (2009)

Calinescu, R., et al.: Dynamic QoS management and optimization in service-based systems. IEEE Trans. Softw. Eng. **37**(3), 387–409 (2011)

Cámara, J., Correia, P., de Lemos, R., Garlan, D., Gomes, P., Schmerl, B. R., Ventura, R.: Evolving an adaptive industrial software system to use architecture-based self-adaptation. In: Litoiu and Mylopoulos (2013), pp. 13–22 (2013)

Cámara, J., Lopes, A., Garlan, D., Schmerl, B.R.: Impact models for architecture-based self-adaptive systems. In: Lanese, I., Madelaine, E. (eds.) Formal Aspects of Component Software—11th International Symposium, FACS 2014, Bertinoro, Italy, 10–12 September 2014, Revised Selected Papers, Volume 8997 of Lecture Notes in Computer Science, pp. 89–107. Springer, Berlin (2014)

Cámara, J., Garlan, D., Schmerl, B.R., Pandey, A.: Optimal planning for architecture-based self-adaptation via model checking of stochastic games. In: Wainwright, R.L., Corchado, J.M., Bechini, A., Hong, J. (eds.) Proceedings of the 30th Annual ACM Symposium on Applied Computing, pp. 428–435. ACM (2015)

Cámara, J., Correia, P., de Lemos, R., Garlan, D., Gomes, P., Schmerl, B.R., Ventura, R.: Incorporating architecture-based self-adaptation into an adaptive industrial software system. J. Syst. Softw. **122**, 507–523 (2016)

Carzaniga, A., Gorla, A., Mattavelli, A., Perino, N., Pezzè, M.: Automatic recovery from runtime failures. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, 18–26 May 2013, pp. 782–791. IEEE/ACM (2013)

Cheng, S.-W.: Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation. PhD thesis, CMU (2008)

Cheng, S.-W., Garlan, D.: Stitch: a language for architecture-based self-adaptation. J. Syst. Softw. **85**(12), 2860–2875 (2012)

Cheng, B.H.C. et al.: Software engineering for self-adaptive systems: a research roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems, Volume 5525 of Lecture Notes in Computer Science, pp. 1–26. Springer, Berlin (2009)

Cheng, S., Garlan, D., Schmerl, B.R.: Evaluating the effectiveness of the rainbow self-adaptive system. In: 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2009, Vancouver, BC, Canada, 18–19 May 2009, pp. 132–141. IEEE Computer Society (2009)

C-MART. http://theone.ece.cmu.edu/cmart

da Silva, C.E., de Lemos, R.: Dynamic plans for integration testing of self-adaptive software systems. In: Giese, H., Cheng, B.H.C. (eds.) 2011 ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, Waikiki, Honolulu , HI, USA, 23–24 May 2011, pp. 148–157. ACM (2011)

Didona, D., Romano, P.: Using analytical models to bootstrap machine learning performance predictors. In: 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS), pp. 405–413 (2015)

D'Ippolito, N., Braberman, V. A., Kramer, J., Magee, J., Sykes, D., Uchitel, S.: Hope for the best, prepare for the worst: multi-tier control for adaptive systems. In: Jalote et al. (2014), pp. 688–699 (2014)

Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: Atlee, J.M., Inverardi, P. (eds.) Proceedings of the 31st International Conference on Software Engineering, ICSE 2009, 16–24 May 2009, Vancouver, Canada, pp. 111–121. IEEE (2009)

Filieri, A., Hoffmann, H., Maggio, M.: Automated design of self-adaptive software with control-theoretical formal guarantees. In: Jalote et al. (2014), pp. 299–310

Filieri, A., Hoffmann, H., Maggio, M.: Automated multi-objective control for self-adaptive software design. In: Nitto et al. (2015), pp. 13–24

Forejt, V., Kwiatkowska, M.Z., Parker, D.: Pareto curves for probabilistic model checking. In: Chakraborty, S., Mukund, M. (eds.) Automated Technology for Verification and Analysis—Proceedings of the 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, 3–6 October 2012, Volume 7561 of Lecture Notes in Computer Science, pp. 317–332. Springer, Berlin (2012)

Garlan, D., Cheng, S.-W., Huang, A., Schmerl, B., Steenkiste, P.: Rainbow: architecture-based self-adaptation with reusable infrastructure. IEEE Comput. **37**(10), 46–54 (2004)

Giunchiglia, F. et al.: Planning as model checking. In: ECP, Volume 1809 of LNAI. Springer, Berlin (1999)

Huber, N., van Hoorn, A., Koziolek, A., Brosig, F., Kounev, S.: Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments. Serv. Oriented Comput. Appl. **8**(1), 73–89 (2014)

Huebscher, M.C., McCann, J.A.: A survey of autonomic computing—degrees, models, and applications. ACM Comput. Surv. **40**(3), 7 (2008)

Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Comput. **36**(1), 41–50 (2003)

Klein, C., Maggio, M., Årzén, K., Hernández-Rodriguez, F.: Brownout: building more robust cloud applications. In: Jalote et al. (2014), pp. 700–711 (2014)

Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: Briand, L.C., Wolf, A.L. (eds.) International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, 23–25 May 2007, Minneapolis, MN, USA, pp. 259–268 (2007)

Kwiatkowska, M.Z., Parker, D.: Automated verification and strategy synthesis for probabilistic systems. In: Hung, D.V., Ogawa, M. (eds.) Automated Technology for Verification and Analysis—Proceedings of the 11th International Symposium, ATVA 2013, Hanoi, Vietnam, 15–18 October 2013, Volume 8172 of Lecture Notes in Computer Science, pp. 5–22. Springer, Berlin (2013)

Kwiatkowska, M.Z., Norman, G., Parker, D.: Stochastic model checking. In: Bernardo, M., Hillston, J. (eds.) Formal Methods for Performance Evaluation, 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28–June 2 2007, Advanced Lectures, Volume 4486 of Lecture Notes in Computer Science, pp. 220–270. Springer, Berlin (2007)

Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification—Proceedings of the 23rd International Conference, CAV 2011, Snowbird, UT, USA, 14–20 July 2011, Volume 6806 of Lecture Notes in Computer Science, pp. 585–591. Springer, Berlin (2011)

Laprie, J.-C.: From Dependability to Resilience. In: DSN Fast Abstracts. IEEE CS (2008)

Li, W., Sadigh, D., Sastry, S., Seshia, S.: Synthesis for human-in-the-loop control systems. In: Abraham, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Volume 8413 of Lecture Notes in Computer Science, pp. 470–484. Springer, Berlin (2014)

Marco, A.D., Inverardi, P., Spalazzese, R.: Synthesizing self-adaptive connectors meeting functional and performance concerns. In: Litoiu and Mylopoulos (2013), pp. 133–142 (2013)

Moreno, G.A., Cámara, J., Garlan, D., Schmerl, B.R.: Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In: Nitto et al. (2015), pp. 1–12 (2015)

Moreno, G.A., Cámara, J., Garlan, D., Schmerl, B.R.: Efficient decision-making under uncertainty for proactive self-adaptation. In: Kounev, S., Giese, H., Liu, J. (eds.) 2016 IEEE International Conference on Autonomic Computing, ICAC 2016, Wuerzburg, Germany, 17–22 July 2016, pp. 147–156. IEEE Computer Society (2016)

Mukhija, A., Dingwall-Smith, A., Rosenblum, D.S.: Qos-aware service composition in dino. In: Fifth IEEE European Conference on Web Services (ECOWS 2007), 26–28 November 2007, Halle (Saale), Germany, pp. 3–12. IEEE Computer Society (2007)

Nou, R., Kounev, S., Julià, F., Torres, J.: Autonomic Qos control in enterprise grid environments using online simulation. J. Syst. Softw. **82**(3), 486–502 (2009)

Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) Verification, Model Checking, and Abstract Interpretation, Proceedings of the 7th International Conference, VMCAI 2006, Charleston, SC, USA, 8–10 January 2006, Volume 3855 of Lecture Notes in Computer Science, pp. 364–380. Springer, Berlin (2006)

Pressman, R.S.: Software Engineering: A Practitioner's Approach, 5th edn. McGraw-Hill Higher Education, New York City (2001)

Ramadge, P.J.G., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control Optim. **25**(1), 206–230 (1987)

Rice University Bidding System. http://rubis.ow2.org

Schmerl, B.R., Cámara, J., Gennari, J., Garlan, D., Casanova, P., Moreno, G.A., Glazier, T.J., Barnes, J.M.: Architecture-based self-protection: composing and reasoning about denial-of-service mitigations. In: Williams, L.A., Nicol, D.M., Singh, M.P. (eds.) Proceedings of the 2014 Symposium and Bootcamp on the Science of Security, HotSoS 2014, Raleigh, NC, USA, 08–09 April 2014, p. 2. ACM (2014)

Schoppers, M.: Universal plans for reactive robots in unpredictable environments. In: McDermott, J.P. (ed.) Proceedings of the 10th International Joint Conference on Artificial Intelligence. Milan, Italy, August 1987, pp. 1039–1046. Morgan Kaufmann (1987)

Sykes, D., Heaven, W., Magee, J., Kramer, J.: Exploiting non-functional preferences in architectural adaptation for self-managed systems. In: Shin, S.Y., Ossowski, S., Schumacher, M., Palakal, M.J., Hung, C. (eds.) Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, 22–26 March 2010, pp. 431–438. ACM (2010)

Tajalli, H., Garcia, J., Edwards, G., Medvidovic, N.: PLASMA: a plan-based layered architecture for software model-driven adaptation. In: Pecheur, C., Andrews, J., Nitto, E.D. (eds.) ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, 20–24 September 2010, pp. 467–476. ACM (2010)

Turner, A., Fox, A., Payne, J.I., Kim, H.S.: C-MART: benchmarking the cloud. IEEE Trans. Parallel Distrib. Syst. **24**(6), 1256–1266 (2013)

Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.: RELAX: a language to address uncertainty in self-adaptive systems requirement. Requir. Eng. **15**(2), 177–196 (2010)

Wikipedia. Slashdot Effect. https://en.wikipedia.org/wiki/Slashdot_effect