Computing
DOI 10.1007/s00607-013-0311-7

Architecture-based resilience evaluation for self-adaptive systems

Javier Cámara · Rogério de Lemos · Marco Vieira · Raquel Almeida · Rafael Ventura

Received: 22 May 2012 / Accepted: 10 February 2013

Abstract One of the major challenges related to self-adaptive software systems is the provision of assurances that the system is resilient against changes that may occur either in the system or its environment. These assurances should be based on complementary sources of evidence that collectively justify that the system is able to attain the specified levels of resilience. The contribution of this paper is the definition and development of an architecture-based approach that evaluates by comparison the adaptation mechanisms of a self-adaptive software system. The proposed approach relies on the identification of representative environmental and system changeloads (i.e., sequences of changes) used in the run-time stimulation of the system. The system response obtained from this stimulation is collected and aggregated into a probabilistic model that is employed in the evaluation of system resilience. Our approach is intended to be used before deployment, since the process often involves putting the system through adverse conditions which are not adequate when the system is in production. The feasibility and effectiveness of the proposed approach is demonstrated in the context of Rainbow, an architecture-based platform for self- adaptation, and Znn.com, a case study that reproduces the typical infrastructure for a news website.

J. Cámara (✉) · M. Vieira · R. Almeida · R. Ventura
University of Coimbra, Coimbra, Portugal
e-mail: jcmoreno@dei.uc.pt

M. Vieira
e-mail: mvieira@dei.uc.pt

R. Almeida
e-mail: rrute@dei.uc.pt

R. Ventura
e-mail: ventura@dei.uc.pt

R. de Lemos
University of Kent, Canterbury, UK
e-mail: r.delemos@kent.ac.uk

Keywords Architecture · Testing · Stimulation · Models · Self-adaptation · Resilience

Mathematics Subject Classification 68N01 · 68N99

1 Introduction

Modern software systems are increasingly complex, pervasive, and tend to operate in unpredictable environments. Approaches known as autonomic, self-adaptive or self-healing computing have been promoted as a means for developing dependable systems in a cost-effective manner. These approaches not only support fault recovery, but are also able to deal with more subtle conditions related to the operational state of the system or its environment (e.g., gradual degradation in performance, etc.). One way of handling the complexity of these systems is to use architectural models that provide a high-level view of the system [14,23].

Unfortunately, in addition to the uncertainty associated with run-time changes, the ability of handling changes during run-time, and the provision of flexibility, comes at the price of reducing the predictability of adaptation mechanisms with respect to the dependability requirements of the system. How can one assess if a system will maintain its dependability at run-time in spite of changes that may occur in the system or its environment? Is there a way to determine if a particular alternative for adaptation is better than another one, or that a particular repair will not make the situation worse? Although major advances have been made, existing approaches in self-adaptation do not systematically address the need to determine if a self-adaptive system can deliver a service that *can justifiably be trusted when facing changes* [11] (i.e., that it will be *resilient* [21]). This lack of assurances is an important issue that hampers widespread adoption of self-adaptive systems, which are often regarded as not being dependable by the industry.

To tackle this problem, we propose an architecture-based approach to evaluate resilience in self-adaptive systems. Specifically, we advocate the complementary use of architectural models and run-time stimulation (including testing) to assess the resilience of self-adaptive systems. The number of potential changes that the system and its environment can go through at run-time is virtually unbounded. Hence, the need to focus on identifying only the relevant stimuli (being those that trigger adaptation mechanisms of particular interest) that have the best potential to unveil system or environmental changes for evaluating how the system responds to them.

To achieve this goal, our technique consists of two stages:

- *Identification of changeload*, where architectural models are exploited to identify and select the most relevant (sequences of) changes (i.e., the changeload) that have the best potential to unveil system faults during run-time stimulation. Since it is unfeasible to consider all possible changes that can affect the system, we follow a risk-based approach that uses models to identify the most relevant stimulus (such as, variations in the conditions of the operational environment and system faults) in terms of type of change, probability of occurrence, and potential impact on the system.
- *Run-time stimulation*, where the system and its environment are stimulated during execution according to the changeload, and information about the system's

response is collected and aggregated into a probabilistic model of the system's behavior, used to evaluate by comparison alternative adaptation mechanisms.

Architectural models play a prominent role in our approach to unveil relevant changeloads and drive stimulations, enabling us to: (i) identify environmental stimulus to steer the system towards invariant violations that trigger adaptation; (ii) obtain a profile of the system that is used to identify the most representative stimulus for the system, once it is undergoing adaptation; (iii) obtain probabilistic models of the system's behavior during adaptation. Although probabilistic behavioral models are obtained during run-time stimulation, they are built over the set of properties contained in the architectural model, and describe the evolution of the values of such properties obtained from monitored variables in the system at run-time. The use of probabilistic behavioral models enables the quantification of the probability of satisfaction of system properties (expressed in *Probabilistic Computation-Tree Logic* (PCTL) [4]) when the system is subject to a particular stimulus.

The novelty of the proposed approach is the definition and development of an integrated framework for evaluating, by comparison, the resilience of adaptation mechanisms for self-adaptive systems. The architectural-based approach relies on the identification of representative changeloads for the system and its environment to be used for stimulating the system during run-time. Our approach permits a cost-effective comparison of alternative adaptations, and is well-suited to be used as guidance for system evolution, by helping the developer to assess improvements (or the lack thereof) of new versions of adaptation mechanisms against resilience metrics. We illustrate and validate our proposal with the aid of Znn.com [10], a case study implemented using Rainbow [14], a platform for architecture-based self-adaptation.

The rest of the paper is organized as follows. Sect. 2 describes the Znn.com case study that will be used throughout the paper. Section 3 presents some background information related to the proposed approach. Section 4, details an architecture-based framework for evaluating resilience of self-adaptive systems, which is based on probabilistic model checking. Section 5 describes the experiments carried out for demonstrating the effectiveness of the approach. Section 6 describes some related work. Finally, Sect. 7 concludes the paper, indicating future research directions.

2 Case study

To illustrate our approach for evaluating adaptation mechanisms by comparison, we use the Znn.com case study [10], which is able to reproduce the typical infrastructure for a news website. It has a three-tier architecture consisting of a set of servers that provide contents from backend databases to clients via frontend presentation logic. Architecturally, it is a web-based client-server system that satisfies an N-tier style, as illustrated in 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 client processes makes stateless requests, and the servers deliver the requested contents (i.e., text, images and videos).

The main objective for Znn.com is to provide content to customers within a reasonable response time, while keeping the cost of the server pool within a certain operating

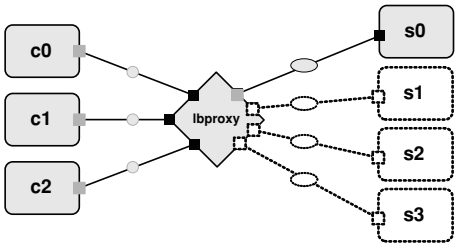

Fig. 1 Znn.com system architecture

budget. It is considered that from time to time, due to highly popular events, Znn.com experiences spikes in request s that it cannot serve adequately, even at maximum pool size. To prevent losing customers, the system can provide minimal textual contents during such peak times, instead of not providing service to some of its customers. Concretely, we identify two main quality objectives for the self-adaptation of the system: (i) maximizing performance, which depends on request response time, server load, and network bandwidth; and (ii) minimizing cost, associated to the number of active servers.

Znn.com is implemented using Rainbow [9,14] (introduced in Sect. 3), a platform that supports architecture-based self-adaptation. Rainbow is capable of analyzing trade-offs among the different objectives, and execute different adaptations according to the particular run-time conditions of the system. For instance, in Znn.com, when response time becomes too high, the system should increment server pool size if it is within budget to improve its performance; otherwise, servers should be switched to textual mode (start serving minimal text content) if cost is near budget limit.

3 Background

Over the past few years, the run-time management of increasingly complex software-intensive systems has become one central concern in Software Engineering [8]. Concretely, one of the major issues in this area is related to achieving conformance to functional and non-functional requirements in a dependable and cost-effective manner while changes may affect the system, its environment, or the system goals.

One of the proposals addressing this concern was IBM's Autonomic Computing initiative [19] by introducing a self-adaptive layer to manage the target system, implementing what is known as the MAPE-K control loop to Monitor relevant variables in the system, Analyze whether adaptation is required, Plan the best course of action, and Execute adaptation, with the addition of a shared Knowledge base, acting as a cornerstone of the process.

There are other approaches that also rely on a closed-loop control approach to self-adaptation but exploit architectural models for reasoning about the target system under management [14,23]. In this paper, we focus on Rainbow [14], an architecture-based platform for self-adaptation, which has the following distinct features: an explicit architecture model of the target system, a collection of adaptation strategies, and utility preferences to guide adaptation. Rainbow leverages the notion of *architectural style* [1] to exploit commonalities between systems, providing reusable infrastructures

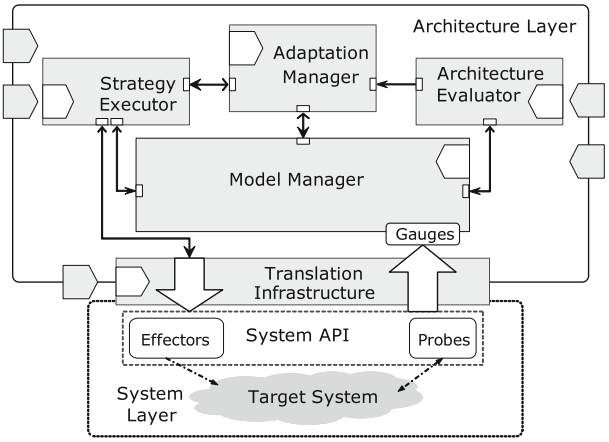

Fig. 2 The Rainbow framework

with explicit customization points that can be applied to a wide range of systems. The goal is to provide a base of reusable infrastructure through customization, which aims to reduce the cost of development.

The framework defined by Rainbow includes mechanisms for (Fig. 2): monitoring a target system and its environment (using the observations for updating the architectural model of the target system), detecting opportunities for improving the system's quality of services (QoS), deciding the best course of adaptation based on the state of the system.

Building upon the elements of the architectural style, Rainbow provides a language called *Stitch* [9] to represent human adaptation knowledge using three high-level concepts:

- *Operator* is the most primitive unit of execution. Represents a basic configuration command provided by the system (corresponding to a system-level effector), and are determined by the architectural style of the system.
- *Tactic* is an abstraction that groups operators to form a single step of adaptation. Tactics are used as primitive actions, and have an associated cost/benefit impact on the different quality dimensions. Using the operators in the architectural style of Znn.com, we can specify pairs of tactics with opposing effects for enlisting/discharging servers, and raising/lowering server content fidelity.
- *Strategy* encapsulates an adaptation process, where each step is the conditional execution of a tactic. Strategies are characterized in Stitch as a tree of condition-action-delay decision nodes, where delays correspond to a time-window for observing tactic effects. System feedback (through the dynamically-updated architectural model of the system) is used to determine the next action (i.e., tactic) at every step during strategy execution.

Let us consider a sample strategy to reduce response time:

```
1  import model "ZnnSys.acme" {ZnnSys as M, ZnnFam as T};
2  import model "ZnnEnv.acme" {ZnnEnv as E};
3  importlib "znn.tactics";
4
5  define boolean styleApplies=Model.hasType(M, "ClientT") && Model.hasType(M, "ServerT");
6  define boolean cViolation=exists c:T.ClientT in
7              M.components|c.expRspTime > M.MAX_RSPTIME;
8
9  strategy ReduceResponseTime [styleApplies && cViolation]{
10     define boolean hiLatency =
11         exists k :T.HttpConnT in M.connectors| k.latency > M.MAX_LATENCY;
12     define boolean hiLoad =
13         exists s :T.ServerT in M.components | s.load > M.MAX_UTIL;
14
15     t1:(#[Pr{t1}] hiLatency)->switchToTextualMode()@[1000/*ms*/]{
16         t1a: (success)->done;
17     }
18     t2:(#[Pr{t2}] hiLoad)->enlistServer(1)@[2000/*ms*/]{
19         t2a:(!hiLoad)->done;
20         t2b:(!success)->do[1]t1;
21     }
22     t3:(default)->fail;
23 }
```

Strategy ReduceResponseTime first specifies its applicability condition, which is used in strategy selection to determine whether the strategy should be considered. In this example, this condition is defined using predicates styleApplies (line 5), which checks whether the model defines the architectural types used in the strategy, and the predicate cViolation (lines 6–7), which checks for the condition that some client is experiencing above-normal response time. Moreover, the example strategy defines two Boolean auxiliary functions, hiLatency (lines 10–11) and hiLoad (lines 14–15). In the body of the strategy, node t1 (lines 12–13), executes tactic switchToTextualMode if hiLatency evaluates to true. To account for the delay in observing the outcome of tactic execution in the system, t1 specifies a delay window of 1,000 ms (end of line 15). During the tactic execution, the child node t1a is evaluated either when the intended effect of the tactic switchToTextualMode is observed (this intended tactic effect is defined in a separate tactic script, and in the case of switchToTextualMode consists in reducing response time below threshold M.MAX_RSPTIME), or if the intended tactic effect is not observed, as soon as the delay window expires. Keyword success is used to evaluate whether the intended effect for the tactic (in this case, the one in t1) has been achieved, finishing the execution of the strategy (done) if the tactic has been successful. Finally, node t2 (line 18) contains a tactic enlistServer which activates a new server. If load is reduced (node t2a , line 19) execution finishes, otherwise (node t2b) the strategy executor repeats the execution of node t1 one time - t1[1] , line 20).

4 Architecture-based framework for evaluating resilience

In this section, we start by providing an overview of our approach and framework for resilience evaluation, and then formally describe the models on which the proposed framework relies upon.

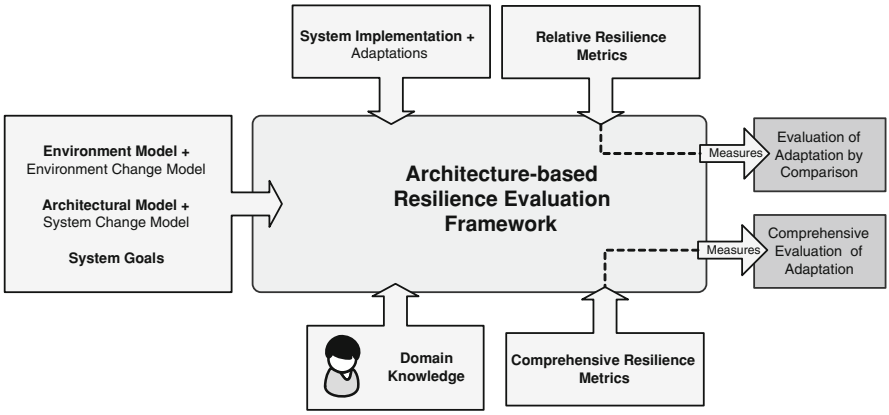

Fig. 3 Use scenarios for the architecture-based resilience evaluation framework

Figure 3 illustrates the possible use scenarios of our framework for architecture-based resilience evaluation: (i) Evaluation of Adaptation by Comparison, which makes use of relative resilience metrics in order to compare how different adaptive solutions respond to a particular set of (system or environmental) conditions; and (ii) Comprehensive Evaluation of Adaptation, which exploits a comprehensive set of resilience metrics for the independent evaluation of an adaptive system. However, the current paper focuses in evaluation by comparison, dealing only with the evaluation of resilience in alternative adaptive solutions, relative to a representative changeload particular to the specific kind of situations that adaptation tackles.

Our approach for evaluating resilience is intended to be used by the developer of the system just before its deployment, since the process often involves putting the system through adverse conditions which are not adequate when the system is already in production. Moreover, it is worth observing that although the approach can be applied to systems that include third-party components as long as the developer has control over their deployment, when the system includes third-party components deployed at remote locations (e.g., Web services), the approach is limited in the kind of changes that can be injected (e.g., simulating a remote service being down).

Our architecture-based framework for evaluating the resilience of self-adaptive systems consists of two main stages, as shown in Fig. 4 [1]:

1. *Changeload identification* consists in identifying the sequence of changes (i.e., the changeload) relevant for the run-time stimulation of the system and its environment. This stage is further partitioned into:
   (a) *Environment stimulation*, whose objective is to identify the environmental changeload required to drive the environment towards conditions that trigger system adaptations[2].

[1] Steps decorated with a gear in the figure are fully automatic, whereas steps decorated with the designer are entirely manual. The remaining steps are partially automated, requiring input from the user.

[2] In the remainder of this paper, we make a distinction between the general term *adaptation* or *adaptation alternative*, and *adaptation strategy* in the concrete context of Rainbow.

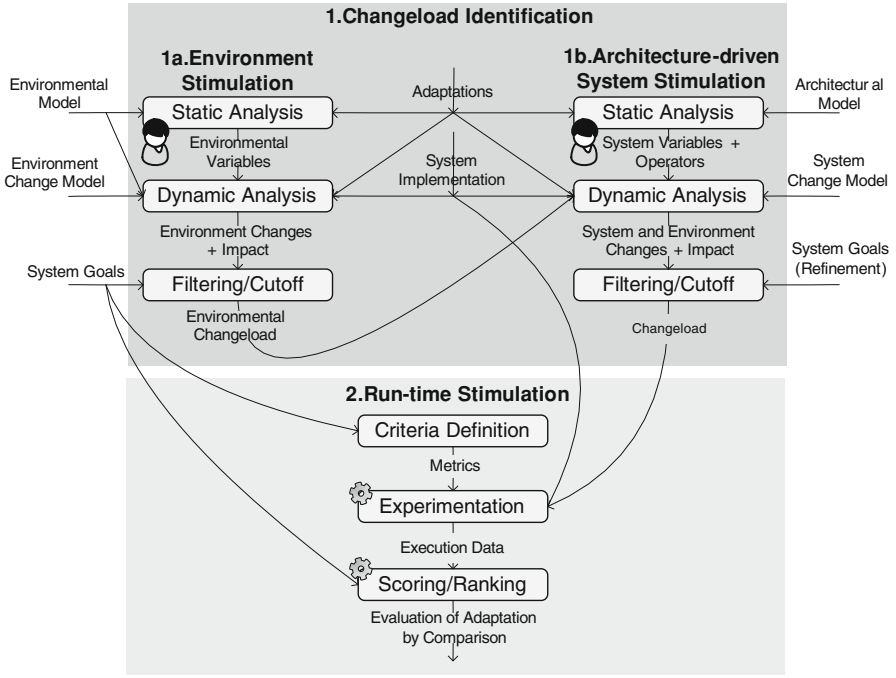

Fig. 4 Overview of the framework

(b) *Architecture-driven system stimulation*, that deals with identifying the system changeload to be used in assessing the resilience of adaptation mechanisms. Each of the aforementioned stages is further divided into (i) Static Analysis, which identifies variables and operations (based in operators of the architectural style in the case of architecture-driven system stimulation) that are referred in adaptations, architecture, and environment models; (ii) Dynamic Analysis, in which the elements identified during static analysis are refined into a set of changes that are later applied to the running system to assess their impact; and (iii) Filtering/Cutoff, of changes ranked by impact and filtered by relevance.

Each of the aforementioned stages is further divided into (i) Static Analysis, which identifies variables and operations (based in operators of the architectural style in the case of architecture-driven system stimulation) that are referred in adaptations, architecture, and environment models; (ii) Dynamic Analysis, in which the elements identified during static analysis are refined into a set of changes that are later applied to the running system to assess their impact; and (iii) Filtering/Cutoff, of changes ranked by impact and filtered by relevance.

2. *Run-time stimulation* of the running system and its environment, according to the changeload identified at the previous stage to determine the best of several alternatives when tackling a particular situation that requires adaptation. This stage is divided into: (i) Criteria Definition, where resilience metrics are defined according to the system goals, (ii) Experimentation in which the system and its environment

are stimulated according to the changeload and information regarding the system's execution for each adaptation alternatives is collected, according to the metrics, as sets of execution traces, and (iii) Scoring/Ranking, where each set of traces of a particular adaptation alternative is transformed into a probabilistic response model of the system, which is used as input to a probabilistic model-checker in addition to the resilience properties obtained from system goals.

4.1 Framework models

This section introduces the different kinds of model that we use in our approach, namely: (i) architecture and environment models that describe the system's structure and environment operational conditions; (ii) behavioral models that enable the evaluation of the system's probabilistic response to changes; and (iii) changeload model that systematically describe the changes that the system will be subjected to for evaluation purposes.

4.1.1 Architecture and environment models

We start this section by formally defining our system architectural model and environment model.

Definition 1 (*Architecture model*) An architecture model is a tuple $\mathcal{A} = (\mathcal{T}, \mathcal{G}, \Gamma_{sys})$, where:

- $\mathcal{T} = \mathcal{T}_{comp} \cup \mathcal{T}_{conn}$ is a set of architectural types, where $\mathcal{T}_{comp}$ and $\mathcal{T}_{conn}$ are the sets of component and connector types, respectively,
- $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ is a graph describing a system configuration, where:
  - $\mathcal{N}$ is a set of nodes, where a typing Λ assigns an architectural type $\Lambda(\mathfrak{n}) \in \mathcal{T}$ to every $\mathfrak{n} \in \mathcal{N}$.
  - $\mathcal{E}$ is a set of edges, where each one of them consists of an unordered pair of nodes $(\mathfrak{n}, \mathfrak{n}')$, such that $\Lambda(\mathfrak{n}) \in \mathcal{T}_{comp}$ and $\Lambda(\mathfrak{n}') \in \mathcal{T}_{conn}$,
- Γ_{sys} is a function that assigns a set of properties $\Gamma_{sys}(\mathfrak{tn})$ to each architectural type $\mathfrak{tn} \in \mathcal{T}$.

The component-and-connector style of architecture descriptions used in this work is inspired by the ACME ADL [15]. Moreover, we assume that our architectural models meet a set of constraints imposed by the architectural style used [3], that we leave implicit in our definition for the sake of clarity.

Definition 2 (*Environment model*) An environment model associated with an architecture model $\mathcal{A} = (\mathcal{T}, \mathcal{G}, \Gamma_{sys})$, is a function Γ_{env} that assigns a set of environment properties $\Gamma_{env}(\mathfrak{tn})$ to each architectural type $\mathfrak{tn} \in \mathcal{T}$.

Example 1 Figure 5 displays an architecture model of an instance of Znn.com, where the set of architectural types is $\mathcal{T} = \{\mathsf{ClientT}, \mathsf{HttpConnT}, \mathsf{ProxyT}, \mathsf{ServerT}\}$.

[3] Architectural styles are patterns of system organization that enable the exploitation of commonalities across systems [24].

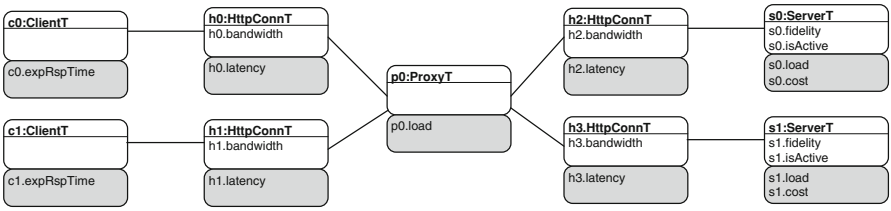

Fig. 5 Architecture model for an instance of Znn.com

Properties for each of each of the architectural types include $\Gamma_{sys}(\mathsf{HttpConnT}) = \{\mathsf{bandwidth}\}$, and $\Gamma_{sys}(\mathsf{ServerT}) = \{\mathsf{fidelity}, \mathsf{isActive}\}$. Environment properties (displayed inside the gray boxes) include $\Gamma_{env}(\mathsf{ClientT}) = \{\mathsf{expRspTime}\}$, $\Gamma_{env}(\mathsf{ProxyT}) = \{\mathsf{load}\}$, $\Gamma_{env}(\mathsf{HttpConnT}) = \{\mathsf{latency}\}$ and $\Gamma_{env}(\mathsf{ServerT}) = \{\mathsf{load}, \mathsf{cost}\}$.

4.1.2 Probabilistic models

In this section, we introduce the kind of probabilistic models that we use to reason about the behavioral response of the system, either in the presence of changes in its environment, or as it undergoes adaptation. In particular, we employ Discrete-Time Markov Chains (DTMCs) [20] as the basis to represent probabilistic behavior. Moreover, we introduce the kind of properties that we check, as well summarize as Probabilistic Computation Tree Logic (PCTL) [4], the language used for expressing them.

System behavior is directly observed through the monitoring of a set of properties on the architecture model during system execution. Concretely, in architecture-based self-adaptive platforms, relevant system-level variables that describe the system's behavior are mapped to component and connector properties in the architecture model. In particular, the behavior of the system at run-time can be characterized through a collection of n real-valued random variables $X = \{x_1, \ldots, x_n\}$, that can be mapped to $\bigcup_{\mathfrak{ty} \in \mathcal{T}} \Gamma_{sys}(\mathfrak{ty})$, given an architecture model $\mathcal{A} = (\mathcal{T}, \mathcal{G}, \Gamma_{sys})$.

The process of sampling in time and space these variables results in their natural quantization and time-discretization, in such a way that $\forall x_{i \in \{1,\ldots,n\}}$:

- $[\alpha_i, \beta_i]$ is the range of x_i, with $\alpha_i, \beta_i \in \mathbb{R}$.
- $\eta_i \in \mathbb{R}^+$ is a quantization parameter associated to x_i, which takes its values in the set:

$$[\mathbb{R}]_{x_i} = \{r \in \mathbb{R} \mid r = k\eta_i,\ k \in \mathbb{Z},\ \alpha_i \leq r \leq \beta_i\}.$$

Hence, given an observed value of x_i at time point t (denoted as $x_i(t)$), the corresponding quantized value is obtained as:

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

Considering all variables in X, we can define a metric space $([\mathbb{R}^n]_X, \delta)$ where:

- $[\mathbb{R}^n]_X = [\mathbb{R}]_{x_1} \times \ldots \times [\mathbb{R}]_{x_n}$ characterizes the state-space of the system.
- $\delta : [\mathbb{R}^n]_X \times [\mathbb{R}^n]_X \to \mathbb{N}$ is a metric such that, for any two states s and s':

$$\delta(s, s') = \max_{i=1}^{n} \frac{|s[i] - s'[i]|}{\eta_i},$$

where $s[i]$ indicates the (quantized) value of variable x_i in an arbitrary state s.

The metric space defined above is used as the basis to describe the probabilistic behavior of the system in DTMCs. Moreover, we assume a time discretization parameter $\tau \in \mathbb{R}^+$ associated to the sampling period established for the observation of the variables that determines the transition time in our DTMC-based models.

Discrete-time Markov chains In order to model the behavior of our system, we employ Discrete-Time Markov Chains, defined as state-transition systems augmented with probabilities. States represent possible configurations of the system. Transitions among states occur at discrete time and have an associated probability. DTMCs are discrete stochastic processes where the probability distribution of future states depend only upon the current state. Let AP be a fixed, finite set of atomic propositions used to label states with properties of interest.

Definition 3 (*DTMC*) A (labelled) Discrete-Time Markov Chain (DTMC), defined over a metric space $([\mathbb{R}^n]_X, \delta)$ is a tuple (S, S_0, P, L) where:

- $S \subseteq [\mathbb{R}^n]_X$ is a finite set of states;
- $S_0 \subseteq S$ is a set of initial states;
- $P : S \times S \to [0, 1]$ is a transition probability matrix s.t. $\forall s \in S,\ \sum_{s' \in S} P(s, s') = 1$;
- $L : S \to 2^{AP}$ is a labelling function which assigns to each state $s \in S$ the set $L(s)$ of atomic propositions that are true in the state.

Each state in S is an ordered n-tuple containing the quantized values of each of the system variables considered in the model. Moreover, each element $P(s, s')$ in the transition probability matrix represents the probability that the next state of the process will be s', given that the current state is s. For the sake of clarity, in the remainder of this paper we indicate the existence of a transition between two arbitrary states s and s' in a DTMC (denoted as $s \longrightarrow s'$) iff $P(s, s') \neq 0$.

Moreover, in order to assess the impact caused by changes in the system, we extend DTMCs with reward (or cost) functions that enable the quantification of variations in system variables throughout different execution paths.

Definition 4 (*DMRM*) A Discrete-time Markov Reward Model (DMRM) is a couple $(D, \langle \rho_1, \ldots, \rho_n \rangle)$, with DTMC $D = (S, S_0, P, L)$ and $\rho_{i \in \{1,\ldots,n\}} : S \to \mathbb{R}_0^+$ assignment functions that associate a reward to any state in S, where $\rho_i(s)$ is a reward obtained on leaving state s. Each of the reward assignment functions ρ_i corresponds to variable x_i in the state-space $[\mathbb{R}^n]_X$, over which DTMC D is built.

Resilience properties To express resilience properties about the system, we use PCTL [4] [4], which is a logic language inspired by CTL [4]. With the aid of PCTL, a designer can express properties about the system that are typically domain-dependent. Furthermore, to ease the formulation of probabilistic properties, we make use of property specification patterns [12] that describe generalized recurring properties in probabilistic temporal logics. In our case, we are interested in how the system responds to changes, so we restrict ourselves to properties that can be instanced by using probabilistic response patterns [17], adapted to PCTL syntax (see Table 1). These patterns include a premise Φ_1 that represents in our case a change in the system or its environment, and a subformula enclosed by the probabilisic operator $\mathcal{P}_{\bowtie p}(.)$ that represents the response to that change that we are expecting from the system (with a probability bound p and a time bound t).

Example 2 To illustrate the use of PCTL for the specification of properties, let us consider the architecture model of Znn.com introduced in Example 1, where we are interested in assessing how the system reacts to request response time going above a particular threshold. Let `expRspTime` be the variable associated with experienced request response time (defined as the maximum of all client experienced response times), and `totalCost` be the variable associated with operating cost (defined as the sum of all server operation costs). We can then define the following predicates:

$$\begin{aligned}
\texttt{cViolation} &= \texttt{expRspTime} > \texttt{MAX_RSPTIME}, \text{ with}\\
\texttt{expRspTime} &= \max_{\mathrm{n}\in\mathcal{N}:\Lambda(\mathrm{n})=\mathsf{ClientT}} \mathrm{n}.\mathsf{expRspTime}, \text{ and}\\
\texttt{hiCost} &= \texttt{totalCost} \geq \texttt{THRESHOLD_COST},\\
&\quad \text{with } \texttt{totalCost} = \sum_{\mathrm{n}\in\mathcal{N}:\Lambda(\mathrm{n})=\mathsf{ServerT}} \mathrm{n}.\mathsf{cost},
\end{aligned}$$

where `MAX_RSPTIME` is a threshold that establishes the maximum acceptable response time, and `THRESHOLD_COST` determines the maximum operating budget expected for the system. Based on these predicates, we may instantiate the following PCTL property, making use of the probabilistic constrained response pattern included in Table 1:

$$\mathcal{P}_{\geq 1}[G(\texttt{cViolation} \Rightarrow \mathcal{P}_{\geq 0.85}(\neg\texttt{hiCost}\ U^{\leq 120}\neg\texttt{cViolation}))]$$

This property reads as: "When response time goes above threshold `MAX_RSPTIME`, the probability of lowering response time below `MAX_RSPTIME` without exceeding the expected budget `THRESHOLD_COST` in 120 s is >0.85".

Moreover, to quantify the impact of changes on the different quality dimensions of the system, we also make use of Probabilistic Reward CTL (PRCTL) [3], which extends PCTL with reward-specific operators aimed at the specification of performability measures over DTMC models. Given an expected reward $r \in \mathbb{R}_0^+$, and a time bound $t \in \mathbb{N}$, the formula $\mathcal{Y}^t_{\bowtie r}(\Phi)$ asserts that the *instantaneous reward* $Ir(\Phi, t)$ in Φ-states (i.e., the states that satisfy the PCTL state formula Φ) after t transitions from

[4] A summary of PCTL is provided in Appendix A.

Table 1 Probabilistic response specification patterns

PCTL formulation	Description
$\mathcal{P}_{\geq 1}[G(\Phi_1 \Rightarrow \mathcal{P}_{\bowtie p}(F^{\leq t}\Phi_2))]$	**Probabilistic Response.** After state formula Φ_1 holds, state formula Φ_2 must become $true$ within time bound t, with a probability bound $\bowtie p$.
$\mathcal{P}_{\geq 1}[G(\Phi_1 \Rightarrow \mathcal{P}_{\bowtie p}(\neg\Phi_2 U^{\leq t}\Phi_3))]$	**Probabilistic Constrained Response.** After state formula Φ_1 holds, state formula Φ_3 must become $true$, without Φ_2 ever holding, within time bound t, with a probability bound $\bowtie p$.

the origin state, meets bound r. The intuition behind $Ir(\Phi, t)$ is that of a weighed average of the individual rewards in Φ-states that can be found exactly after t transitions from the origin state s, where each individual reward is reduced by the probabilities found along the path. Additional operators of PRCTL and details on how $Ir(\Phi, t)$ is computed can be found in [3]. For multiple rewards, we write $\mathcal{Y}^{t,i}_{\bowtie r}(\Phi)$ (respectively $Ir^i(\Phi, t)$) to indicate that a formula refers to the i-th reward in the corresponding DMRM (i.e., reward assignment function ρ_i). Computation of instantaneous rewards is illustrated by Example 9, further in this section.

Operational profiles Within a self-adaptive system, we may distinguish between a *conventional operational profile* in which the system is operating without experiencing any anomalies, and *non-conventional operational profiles* associated with changes in the system or its environment that induce anomalies in the system (typically triggering *adaptations*).

Definition 5 (*Conventional operational profile*) The conventional operational profile C of a system is the region of the state space $S = [\mathbb{R}^n]_X$ where no anomalies hold:

$$C = \{s \in S \mid \forall \alpha \in A,\ s \not\models \alpha\}$$

where A is the set of possible anomalies that the system can experience expressed as PCTL state formulae.

Definition 6 (*Non-conventional operational profile*) A Non-conventional operational profile N associated with an anomaly α, is the region of the state space $S = [\mathbb{R}^n]_X$ where the anomaly holds:

$$N = \{s \in S \mid s \models \alpha\}$$

where $\alpha \in A$ is a PCTL state formula built over the set of atomic propositions AP.

Example 3 Let us consider the predicates introduced in Example 2 as the set of possible anomalies in the system $A = \{\texttt{cViolation}, \texttt{hiCost}\}$. We may then define the conventional operational profile of the system as:

$$C = \{s \in S \mid s \models \neg(\texttt{cViolation} \vee \texttt{hiCost})\}$$

That is, the set of states in which the system is operating on budget and according to an acceptable request response time. Furthermore, associated with each anomaly, we identify a non-conventional operational profile:

$$N_{cViolation} = \{s \in S \mid s \models \texttt{cViolation}\}$$
$$N_{hiCost} = \{s \in S \mid s \models \texttt{hiCost}\}$$

Appendix B describes in detail how information observed from the system is used to obtain models of non-conventional operational profiles and impact of anomalies in the system.

4.1.3 Changeload model

This section describes the proposed model for the changeload, presenting the definitions adopted for the fundamental concepts that form the basis of its structure.

Definition 7 (*Change type*) A change type, given a set of architectural types $\mathcal{T}$, is defined as a tuple (src, A, B) that characterizes a change, where:

- $src \in \mathcal{T}$ identifies the *source* of the change[5],
- $A = \langle a_1, \ldots, a_k \rangle$ is a vector of *attributes* that hold information about the specific properties (variables) associated with the change type,
- $B = \langle b_1, \ldots, b_k \rangle$ describes the dynamics of the attributes in A (how they evolve over time, e.g., through a polynomial, exponential, or step function).

Example 4 Referring to Fig. 5, consider the change *"Increase the bandwidth of connector* HttpConnT *by* 30 %". A possible definition of a change type would be:

$$\text{increase\_bandwith\_type} = (\text{HttpConnT}, \langle \text{bandwidth} \rangle, \langle \text{step\_function} \rangle)$$

Definition 8 (*System change model*) a *system change model* $\mathcal{CM}_{sys}$ is a set of change types applicable to the system properties (Γ_{sys}) of a family of systems that share some degree of commonality (e.g., common subset of architectural types).

Definition 9 (*Environment change model*) an *environment change model* $\mathcal{CM}_{env}$ is a set of change types applicable to the environment properties (Γ_{env}) of a system family with some degree of commonality (e.g, common subset of architectural types).

Definition 10 (*Change*) Given a set of change types CT defined for a set of architecture types $\mathcal{T}$, and an architecture model $\mathcal{A} = (\mathcal{T}, \mathcal{G}, \Gamma_{sys})$, a *change* is a tuple $(ct, srcinst, V_A, V_B, ti, d)$ that corresponds to an instantiation of a change type, where:

- $ct = (src, A, B) \in CT$ determines the change type to be instanced as a change,

[5] Actually, changes can also occur in system goals. However, in the scope of this work, we assume fixed goals, therefore change sources are always associated with (system or environment) properties of an architectural type.

- $srcinst \in \mathcal{N}$, such that $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where $\Lambda(srcinst) = src$, is the instance of the source of change (i.e., where it actually occurs),
- $V_A = \langle v_{A1}, \ldots, v_{Ak} \rangle$ is a vector of attribute values instantiating the attributes in A,
- $V_B = \langle v_{B1}, \ldots, v_{Bk} \rangle$ is a vector of behavior instances of the elements in B (i.e., a behavior instance $v_{Bi},\ i \in \{1, \ldots, k\}$ is a function of type $b_i \in B$, describing the evolution over time of the attribute $v_{Ai} \in V_A$),
- $ti \in \mathbb{R}_0^+$ determines the *time instant* in which the change instance is triggered,
- $d \in \mathbb{R}_0^+$ is the *duration* associated with the change.

It is worth observing that changes can be classified under three different categories according to their duration: (i) Permanent (duration ∞); (ii) Instantaneous (duration 0); and otherwise (iii) Transient (e.g., a temporary peak in workload).

Example 5 If we consider the change type described in Example 4, a possible instantiation of that change type could be the following instantaneous change:

$$(\mathsf{increase\_bandwidth\_type}, \mathsf{h0}, \langle 30 \rangle, \langle \theta(t) \rangle, 10, 0), \text{ with:}$$
$$\theta(t) = \begin{cases} current\_bandwidth & \text{if } t < ti \\ current\_bandwidth \times 1.3 & \text{if } t \geq ti. \end{cases}$$

The systematic identification and classification of change types is fundamental to support the definition of change scenarios, discussed in the next paragraphs.

One of the main base concepts is that of a *scenario*. A scenario is a postulated sequence of events that captures the state of the system and its environment, system goals, and changes affecting all the aforementioned elements. It is defined in terms of state (system and environment), changes applied to that state, and system goals.

Definition 11 (*State condition*) a *state condition* is a tuple (A, B, V_A, V_B) that corresponds to the description of the evolution of a set of (either system or environment) properties over time:

- $A = \langle a_1, \ldots, a_k \rangle$ is a vector of *attributes* that enumerates the specific properties of interest (variables) for the (environment or system) state,
- $B = \langle b_1, \ldots, b_k \rangle$ describes the dynamics of the attributes in A (how they evolve over time, e.g., through a polynomial, exponential, or step function).
- $V_A = \langle v_{A1}, \ldots, v_{Ak} \rangle$ is a vector of attribute values instantiating the attributes in A,
- $V_B = \langle v_{B1}, \ldots, v_{Bk} \rangle$ is a vector of behavior instances of the elements in B (i.e., a behavior instance $v_{Bi},\ i \in \{1, \ldots, k\}$ is a function of type $b_i \in B$, describing the evolution over time of the attribute $v_{Ai} \in V_A$),

Definition 12 (*Scenario*) A scenario is a tuple (wl, oc, G, C), where:

- wl is a state condition that represents the workload (amount and type of work assigned to the system),
- oc is a state condition that represents the operational conditions of the system (including software and hardware resources needed for the system to perform its service),

- G is a set of system goals, expressed as PCTL formulas,
- C is a set of changes applied to the state determined by the workload and operational conditions.

In the definition above, goals represent the quality attributes that the system should fulfill at run-time (e.g., maximum response time, minimum throughput, minimum availability, etc.), including a prioritization among potentially conflicting goals.

Scenarios can be classified into two groups: *base scenarios* and *change scenarios*. A base scenario is defined in terms of typical conditions during the execution of the system, which includes: a typical (stable) state of the system and its environment, and a set of fixed goals (for the sake of this paper, we assume goals are fixed).

Definition 13 (*Base scenario*) A base scenario is a tuple $(wl_t, oc_t, G_f, \emptyset)$, where:

- wl_t is a state condition that represents a typical workload of the system,
- oc_t is a state condition that represents the typical operational conditions of the system,
- G_f is a set of fixed system goals, expressed as PCTL formulas.

The workload of a base scenario should be representative of the typical amount and type of work assigned to (or expected from) the system in a specified time period. Typical operation conditions comprise the typical setup of systems in the domain, as well as representative characterization of the system's environment, and the hardware and software resources typically used. Hence, a base scenario reflects the operational characteristics of systems in the domain while running a typical workload and operating in the absence of changes, setting the baseline for comparison with situations when the system is faced with changes that may drive it into an adaptation process. It should be noted that "typical" does only imply a stable state of the system with no abnormal conditions, not that the workload or operation conditions cannot be dynamic.

Change scenarios are derived from a base scenario, but include a representative sequence of changes that may affect the system and its ability to achieve and maintain the fixed goals specified in the base scenario.

Definition 14 (*Change scenario*) A change scenario is a tuple (wl_t, oc_t, G_f, C)

- wl_t is a state condition that represents a typical workload of the system,
- oc_t is a state condition that represents the typical operational conditions of the system,
- G_f is a set of fixed system goals, expressed as PCTL formulas.
- $C \neq \emptyset$ is a set of changes (as defined in Definition 10) applied to the state determined by the workload and operational conditions.

A change scenario is then defined by a typical condition of the system followed by a non-empty set of changes.

Definition 15 (*Changeload*) A changeload is a set of change scenarios.

In the remainder of this paper, we refer to a changeload as *environmental changeload* if its set of scenarios only includes changes in the environment.

4.2 Identification of the changeload

This section details how our approach identifies the changeload to: (i) steer the system towards adaptation (environment), and (ii) evaluate adaptation alternatives (system).

4.2.1 Environment stimulation

The first step to generating a representative changeload that can be used to compare alternative adaptations is determining how environmental changes can trigger adaptations. This task is further partitioned into three steps:

Static analysis To determine how adaptations are triggered, we need to focus on the anomalies associated to non-conventional operational profiles, which act as applicability condition of adaptations. In this paper, we rely on these applicability conditions for identifying the environmental changeload. Hence, let us consider an anomaly α associated with a non-conventional operational profile N_α, as well as an environment model Γ_{env} for our system. The procedure to extract a set of relevant environment variables concerning adaptation triggering consists of: (i) Obtaining the set of variables V that are contained in the predicates included in α, and (ii) For each variable $v \in V$, determining the environment properties in Γ_{env} that can modify v in a way that favors the transition of the system from its conventional operational profile to N_α.

Example 6 In the non-conventional operational profile $N_{cViolation}$ introduced in Example 3, the associated anomaly `cViolation` is defined as `expRspTime > MAX_RSPTIME`. Hence, we need to determine the set of properties in the environment model that have the potential to increase `expRspTime` above the threshold `MAX_RSPTIME`. In particular, looking at the environment model in Example 1, we can observe that both an increase in server load or the latency in the connections are prone to increase expected response time, so in this case we identify the set of variables {HttpConnT.latency, ServerT.load} as relevant to trigger adaptations whose applicability condition is `cViolation`.

Dynamic analysis Once the set of variables relevant to the environmental changeload has been identified, we need to refine them into an appropriate set of representative changes in the environment, intended to achieve the conditions to trigger adaptation. To this end, we follow the following steps:

1. Identify the applicable change types in the environmental change model using the environment properties identified during static analysis.
2. Instantiate applicable change types identified in the previous step.

Example 7 Considering the case of Znn.com as a family of systems that can be instantiated in different configurations (e.g., different number of servers, proxies, etc.), we define a common environmental change model to all of them, containing a number of change types (Definition 7), such as:

$$\text{modify\_network\_latency\_type} = (\text{HttpConnT}, \langle \text{latency\_prop} \rangle, \langle \text{linear\_function} \rangle)$$

Focusing on the configuration of the Znn.com instance in Example 1, we can observe how the properties identified during static analysis can be used to instantiate a set of

changes based on the change type modify_network_latency_type :

$c_1 = (\text{modify\_network\_latency\_type}, \text{h0}, \langle \text{h0.latency} \rangle, \langle \Theta(t) = 2.5 * t \rangle, 15, 20)$
$c_2 = (\text{modify\_network\_latency\_type}, \text{h3}, \langle \text{h3.latency} \rangle, \langle \Theta(t) = 1.2 * t \rangle, 15, 10)$

The first of the changes above, for instance, increases the latency of HTTPConnT connector h0 (connecting client c0 and proxy p0), beginning 30 seconds after the start of the execution of the scenario, at a rate of $2.5ms/s^2$ for 20 seconds.

3. Run each of the identified candidate changes individually on the system under typical workload and operational conditions, gathering data (i.e., in the form of a trace) about the variables included in the set V identified in static analysis, during a particular time frame $[0, t]$. Each change is executed a number of times under similar conditions to obtain a set of traces statistically representative of the behavior of the system variables while undergoing the change. For this step, we assume the availability of effectors that can modify the values of environment variables during execution (i.e., analogously to frameworks such as Rainbow, that use effectors through *operators* to steer the system according to a set of objectives [14]).
4. Build an impact model (as described in Sect. 4.1.2) for each environmental change, using as input the set of traces for each change identified.

Filtering/cutoff Based on the impact models obtained during dynamic analysis, the purpose of filtering is to identify those changes that are going to be part of the environment changeload, by discarding those that are not relevant. For this purpose, the definition of a metric to measure the relevance of a change is required. Concretely, since the objective of the environment changes is to trigger adaptation (i.e., making the system transit from its conventional operational into a non-conventional one), our criterion consists in minimizing the estimated distance from the state of the system after the application of a particular change and the non-conventional operational profile associated to the set of adaptations that we want to evaluate. Hence, the distance from a system state s to a non-conventional operational profile N_α is defined as:

$$dst(s, N_\alpha) = \begin{cases} \min\limits_{s_N \in N_\alpha} \delta(s, s_N) & \text{if } \exists\, \pi = s \longrightarrow \ldots \longrightarrow s_N \\ +\infty & \text{otherwise.} \end{cases}$$

where $\delta : [\mathbb{R}^n]_X \times [\mathbb{R}^n]_X \to \mathbb{N}$ is the metric defined in Sect. 4.1.2 to estimate the distance between any two states of the system [6].

The expected state of the system t_x time units after starting the execution of a change c in state s can be obtained as $exp\_s'(s, \mathcal{I}_t^c, t_x) = (s[1] + Ir^1(true, t_x), \ldots, s[n] + Ir^n(true, t_x))$, where the impact model of the change $\mathcal{I}_t^c$ is used to compute instantaneous rewards. We choose to keep the minimum of the distances for the time frame $[0, t]$ and normalize it for comparison with other changes:

[6] Although the distance to a non-conventional operational profile N_α from state s requires the existence of a trajectory π from s to a state in N_α, we pessimistically assume that non-conventional operational profiles are always reachable from any state in the conventional operational profile of the system. Therefore in practice, this distance is always **estimated** as $dst(s, N_\alpha) = \min\limits_{s_N \in N_\alpha} \delta(s, s_N)$.

$$minDstN(s, \mathcal{I}_t^c, N_\alpha) = \min_{t_x \in \{0,\ldots,t\}} \frac{dst(exp\_s'(s, \mathcal{I}_t^c, t_x), N_\alpha)}{dst(s, N_\alpha)}.$$

Finally, we set a cutoff threshold $thr_{env} \in [0, 1]$ for the relevance of the changes to be kept, discarding changes that score a $minDstN$ above the defined threshold. As a rule of thumb, changes with scores closer to 1 should be discarded in favor of those approaching 0, which indicate a higher relevance (since they tend to minimize the distance to the non-conventional operational profile N_α).

Example 8 Consider the changes c_1 and c_2 introduced in Example 7 with their respective impact models ($\mathcal{I}_{30}^{c_1}$ and $\mathcal{I}_{30}^{c_2}$) built for the time frame [0, 30], and a state s in which changes are applied. The distance to the non-conventional operational profile $N_{cViolation}$ from state s is $dst(s, N_{cViolation}) = 50$, and scores for c_1 and c_2 are $minDst(s, \mathcal{I}_{30}^{c_1}, N_{cViolation}) = 0.4$ and $minDst(s, \mathcal{I}_{30}^{c_1}, N_{cViolation}) = 0.7$, respectively. If we set a cutoff threshold $thr_{env} = 0.5$, in this case we only keep change c_1 for the environmental changeload.

4.2.2 Architecture-driven system stimulation

The identification of the system changeload follows a risk-based approach [25] that considers the probability and impact of system changes, and consists of three steps:

Static analysis Based on the architecture model of the target system, we first identify a set of system variables (from system properties in the model Γ_{sys}) and operations (from operators in the architectural style) that are used in the adaptation. Using this information, we select the potential sources of change (associated with system properties of an architectural type) and, consequently, the relevant change types.

Taking the change types, we then identify specific system changes that may impact the system goals. This is essentially a manual process that makes use of field data (when available) and expert knowledge, and basically consists of finding, for each change type, tangible changes that may affect the system during adaptation. Furthermore, instantiating a change requires the specification of concrete attribute values (that depend on each change) and of the trigger instant, as well as the duration (if applicable) of each change. For the trigger instant, we propose the use of time-based activation with a specific distribution over time.

A key aspect in this phase is the definition of the probability of occurrence of each change. When representative and statistically relevant field data is available, attributes may be characterized using a numeric scale (e.g., percentage). However, field data on the probability of system changes is seldom available. This way, our proposal is to apply expert knowledge and characterize the probability of changes qualitatively.

Dynamic analysis The goal of this step is to understand the impact of each system change in the target system. This makes use of the environmental changeload identified during Environment Stimulation to steer the system towards conditions that trigger adaptation, and the set of system changes identified before, which are injected in the system while undergoing adaptation. Thus, each of the identified system changes is run individually on the system under typical workload and operational conditions,

Table 2 Exposure matrix for changes

	Probability				
		Very high	High	Low	Very low
Impact	Catastrophic	Mandatory	Very high	High	Medium
	Critical	Very high	High	Medium	Low
	Marginal	High	Medium	Low	Very low
	Negligible	Medium	Low	Very low	Negligible

gathering data about the variables included in the set V identified in Sect. 4.2.1 for environment stimulation, during a particular time frame $[0, t]$. As in for the Environment Stimulation, each change is executed a number of times under similar conditions to obtain a set of traces statistically representative of the behavior of the system variables while undergoing the change. This information is then used to build an impact model for each system change.

Filtering/cutoff The goal of this step is deciding which system changes should be included in the changeload. This is needed as the number of potential system changes is really large (especially for complex systems) and, thus may become impracticable. Moreover, it is expected that many of the changes identified present a low probability of occurrence and/or may have a low impact in the system. This way, following a risk-based approach [25], we propose the use of an exposure matrix that allows understanding how relevant each change is. The goal is prioritizing the changes, selecting a Top-N (i.e., the most representative ones), based on a cut-off level.

The idea behind the exposure matrix is to build a correlation of the probability and impact of changes. Table 2 shows an example (in this case the impact is also based on a qualitative scale, but this can be adapted to consider a quantitative scale). For each combination of attribute values a level of representativeness is assigned, namely: Mandatory (changes that must be included in the changeload), Very High, High, Medium, Low, or Very Low representativeness, and Negligible (changes that can be overlooked).

Although the exposure matrix is predefined (i.e., it is part of the framework), adaptations may be done by the user of the framework. In practice, the proposed matrix is based on current practices on risk management in software projects, namely on the Software Risk Evaluation (SRE) method [25], initially proposed by the Software Engineering Institute (SEI) in 1992, which promotes risk management in small projects in an agile way. To build the matrix we considered four values for the Impact and Probability as discussed before. Obviously, the user of the framework may adapt these categories (e.g., when more information is available to support a more detailed classification).

In practice, the combination of the different attributes allows understanding how relevant a given change is. For example, a change with Very High probability and Catastrophic impact is of extreme relevance and must inevitably be included in the changeload. Also, changes having a High probability and a Catastrophic impact or a Very High probability and a Critical impact are of very high importance. The key issue is obviously how to define a cut-off level that separates the changes that should be included in the changeload from the ones less representative. In this example, we may

choose to keep in the changeload only scenarios with High, Very High, and Mandatory representativeness. Clearly, the cut-off level influences the representativeness of the changeload and also its size. Our proposal is to, at least, include all the scenarios with Very High probability and Catastrophic impact.

As mentioned before, field data can be used to support the process of classifying the impact and probability of changes. However, in most cases that data is not available, demanding for expert judgment. Lets analyze two change types to better understand this process:

- *When activating a server, crash the server being activated* it is well known that a server crash is a rare event (i.e., probability is **Very Low**). However, a crash during server activation has a higher probability than a common server crash, as the server activation involves a complex procedure that includes starting multiple processes, ranging from the operating system to the web server. Nevertheless, the probability of such event is still **Low**. In terms of impact, a server crash is typically catastrophic in most scenarios. In this work, although we are dealing with systems that are able to adapt to unexpected events, the impact of a crash during server activation is still **Critical**, as the crashed server is the direct target of the adaptation strategy (i.e., the crash directly impacts the strategy execution). Applying the risk matrix (Table 2), we obtain a **Medium** risk exposure, thus the change is considered in our changeload (see Table 3).
- *When activating a server, crash a different server (i.e., not the one being activated)* a crash in a running server has clearly a lower probability than a crash of a server being activated, thus we consider the probability of occurrence of this change as **Very Low**. As the crashed server is not the direct target of the adaptation strategy being executed, we consider the impact to be **Marginal** (we assume that the system will run another adaptation strategy as soon as the execution of the current one ends). Applying the risk matrix (Table 2), we obtain a Very Low risk exposure, thus the change is not considered in our changeload.

4.3 Run-time system stimulation

Run-time stimulation consists of exercising the running system (and its environment) using the identified changeload to compare the resilience of the different adaptation alternatives. It consists of the following steps:

Criteria definition The criteria for comparison (i.e., the metrics) should be defined according to system goals, as the objective is to understand how effective each adaptation alternative is in terms of assuring that the system satisfies the predefined goals after adaptation. A *resilient system* is defined by Laprie as one whose service *can justifiably be trusted when facing changes* [21], that is, a system that fulfills its goals in a dependable and persisting manner in spite of changes in its environment or the system itself. Therefore, the definition of resilience incorporates the fulfillment of system goals. In other words, the metrics are strongly related to the goals targeted by the system and have to characterize it in a useful and meaningful way. In the concrete case of Znn.com, the system is built to minimize response time while keeping a low operation cost, so system goals are basically related with performance and cost, which

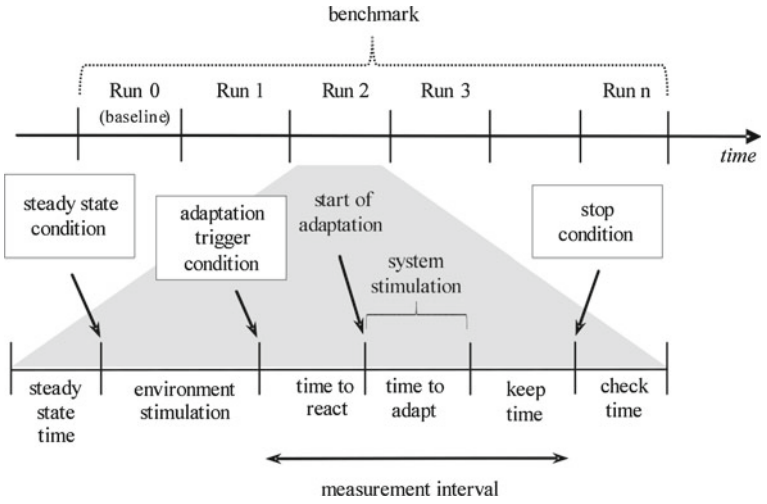

Fig. 6 Experimentation profile

are reflected on the resilience properties (formally expressed in the PCTL properties found in Table 3). Furthermore, metrics should focus on an end-to-end perspective (e.g., the point-of-view of the end-user). Finally, to allow comparison, a small set of metrics should be used, as comparison based on many criteria is difficult (or even impossible) to perform.

For example, in the context of Znn.com, we use as metrics the probability of: (i) lowering response time below `MAX_RSPTIME` after a particular time lapse; and (ii) lowering response time below `MAX_RSPTIME` without ever exceeding the expected budget `THRESHOLD_COST` during a particular time lapse. These metrics are intuitive and simple to use, and are directly related with the system goals of maintaining an acceptable response time while keeping the cost of operating the system within budget.

Experimentation The experimental profile, depicted in Fig. 6, includes a set of runs. Run 0 consists of performing environmental stimulation to trigger adaptation. No system stimulation is performed in this run, as the goal is to collect baseline information about the behavior of the adaptation alternative in the absence of system changes. This baseline will be used later as reference to understand the impact of system changes in the execution of the adaptation strategies. During Runs 1…N the system is run in such a way that environmental stimulation first triggers adaptation, and then changes are injected on top of the environmental stimulation during adaptation to measure their impact in the different adaptation alternatives. In order to assure that each run portraits a realistic scenario as much as possible, and at the same time assures that important properties such result repeatability and representativeness of results are met, the definition of the profile of the run has to follow several rules. The following points summarize those rules (see Fig. 6):

1. The system state must be explicitly restored in the beginning of each run and the effects of the system changes do not accumulate across different runs.

2. The tests are conducted with the system in a steady state condition, which is achieved after a given time executing transactions (steady state time).
3. Environmental stimulation is conducted after the system achieves the steady state to reach an adaptation trigger condition in the first place, and then keeps on going throughout the execution of adaptation, keeping the conditions of the environment to make adaptation valid (e.g., high request rate in Znn.com also after adaptation is triggered). We consider the existence of a *time to react* as there may be a lag between the trigger condition and the activation of the adaptation strategy.
4. Adaptation is exercised by injecting system changes during its execution.
5. When the adaptation completes, the system must continue to run during a keep time in order to characterize the system speedup after adaptation.

Since we are interested in characterizing only the behavior of the system during adaptation, the measurement interval starts in each run only when adaptation starts, and ends when the stop condition is met.

Scoring/ranking As a final step, the measurements are used to compare and rank the adaptation alternatives, taking into account the criteria previously defined. In practice, each set of traces regarding a particular adaptation alternative is transformed into a probabilistic response model of the system, which is used as input to a probabilistic model-checker in addition to the resilience properties obtained from the system goals. As an outcome, adaptation alternatives can be evaluated by comparing them against their quantification of the resilience properties.

5 Experiments

The aim of our experiments is assessing the validity of our approach to compare alternative adaptation strategies. In particular, we evaluate the resilience of Znn.com when using alternative strategies to reduce experienced response time.

For our experimental setup, we deployed Rainbow and the corresponding implementation of Znn.com across seven different machines (Fig. 7): $\mathsf{znns0-3}$ are the four content servers running $\mathsf{Apachev2.2.16}$, znndb is a common backend database running $\mathsf{mySqlv14.14d5.1.61}$, from which the different servers extract the contents, and $\mathsf{znndist}$ is the proxy machine that runs the load balancing software ($\mathsf{distributorv0.7}$) in a round-robin configuration. Rainbow master is deployed in a separate machine ($\mathsf{znnmaster}$). All machines run $\mathsf{DebianLinuxv6.0.4}$, and have 512MB of memory. Moreover, an additional machine ($\mathsf{znnclient}$) running $\mathsf{JMeterv2.5.1}$ generates the traffic during the execution of the system. Connectors labeled in the figure as $\mathsf{h0-h4}$ are those included in the architectural model of the system. Similarly, components represented in the model include their architecture model instance name between parenthesis.

On the deployed system, we estimate the resilience of the system when using three alternative strategies[7] aimed at reducing experienced response time, namely:

1. $\mathsf{SimpleReduceResponseTime}$ ($\mathsf{s1}$) attempts to reduce response time in a simple way by enlisting a new server, and then reducing content fidelity across all servers.

[7] The code of these strategies can be downloaded from the SEAMS portal (http://seams.self-adapt.org).

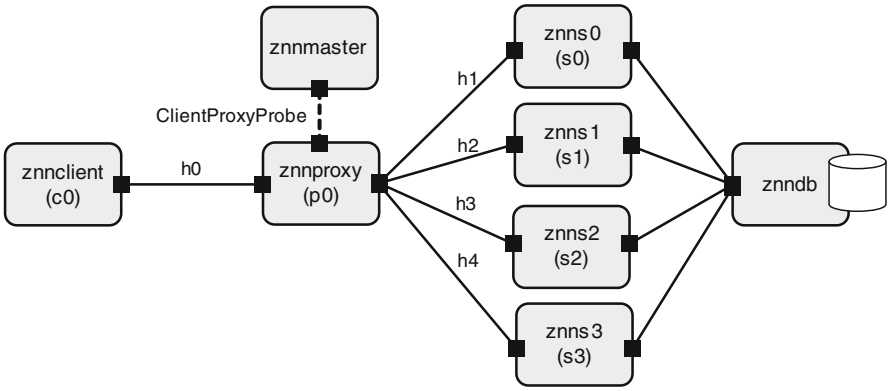

Fig. 7 Znn experimental setup

2. SmarterReduceResponseTime (s2), which defines a threshold regarding the number of clients experiencing a high response time, enlisting a new server if the number of clients is above the defined threshold. After adding the server, if the same condition persists, the strategy adds a second server. Finally, if the number of clients experiencing response time has not been reduced below threshold, the strategy reduces content fidelity across servers.
3. SophisticatedReduceResponseTime (s3) Starts by enlisting two servers, and then defines three different threshold of clients experiencing high response time (e.g., 25, 50, 75 %). If any of the thresholds is crossed, the strategy reduces quality, but proportionally to the number of clients with high response time.

In line with the system objectives of keeping an acceptable response time while keeping down the cost of running the system, we compare the alternative strategies according to the probability of: (P1) Lowering response time below `MAX_RSPTIME` after 10 s; and (P2) Lowering response time below `MAX_RSPTIME` without exceeding cost `THRESHOLD_COST` after 10s (`MAX_RSPTIME` = 1,000 ms. and `THRESHOLD_COST` =2 usd/hour).

The time sampling parameter chosen for the construction of our probabilistic models (as described in Sect. 4.1.2) is $\tau = 1s$, whereas the quantization parameters chosen for experienced response time and cost are $\eta_{\text{expRspTime}} = 50$ and $\eta_{\text{totalCost}} = 1$, respectively. Moreover, the probabilistic models are built for the following set of change scenarios during the time frame [0, 30]:

1. Only the environmental changeload is applied. The base scenario used to build our environmental changeload consists in a typical workload wl_t of constant low activity (6 requests/minute) that corresponds to the expected load for the system under common circumstances, and typical operating conditions of constant latency in the network links (80 ms. for the client-proxy link h0 - which corresponds approximately to the average network latency that can be expected for trans-atlantic requests, and a much lower 3 ms. for proxy-server links h1, h2, h3, h4 internal to the znn system):

$wl_t = (\langle \text{p0.load} \rangle, \langle \text{constant\_function} \rangle, \langle \text{6req/min} \rangle, \langle \theta_{\text{p0}}(t) = \text{6req/min} \rangle)$
$oc_t = (\ \langle \text{h0.latency, h1.latency, h2.latency, h3.latency, h4.latency} \rangle,$
$\langle \text{constant\_function, constant\_function, constant\_function,}$
$\text{constant\_function, constant\_function} \rangle,$
$\langle \text{80ms, 3ms, 3ms, 3ms, 3ms} \rangle,$
$\langle \theta_{\text{h0}}(t) = \text{80ms}, \theta_{\text{h1}}(t) = \text{3ms}, \theta_{\text{h2}}(t) = \text{3ms}, \theta_{\text{h3}}(t) = \text{3ms}, \theta_{\text{h4}}(t) = \text{3ms} \rangle)$

For the environmental changeload, the set of changes that we introduce is related to modifying the load on znnproxy. During static analysis in environmental stimulation (described in Sect. 4.2.1), both network latency and proxy load were initially identified as possible sources of changes that might influence the triggering of adaptation. However, out of these two possible sources, network latency was discarded since it never affects the triggering of our adaptation strategies. This is due to the way in which znn.com is implemented. Concretely, response time is always probed at the proxy, and does not take into account the time that the request spends out of the system traveling from and to the client (see Fig. 7):

$\text{modify\_proxy\_load\_constant\_type} = (\text{ProxyT}, \langle \text{load} \rangle, \langle \text{constant\_function} \rangle)$
$\text{modify\_proxy\_load\_linear\_type} = (\text{ProxyT}, \langle \text{load} \rangle, \langle \text{linear\_function} \rangle)$
$C = \{\ (c1 = \text{modify\_proxy\_load\_linear\_type}, \text{p0}, \langle \text{load} \rangle, \langle \theta_{\text{p0}-1}(t) = 10t + \text{load} \rangle, 60, 300),$
$(c2 = \text{modify\_proxy\_load\_constant\_type}, \text{p0}, \langle \text{load} \rangle, \langle \theta_{\text{p0}-2}(t) = \text{load} \rangle, 360, 300)\}$

Scenarios in the environmental changeload have a duration of 660 seconds, are built based on the changes in C, and conform to the pattern: (i) initial period of low activity (60s); (ii) period of sharp rise in requests ($c1$, 300s); and (iii) sustained period of peak in requests ($c2$, 300s).

2. The full changeload, including changes in the system is applied. Concretely, relevant scenarios identified include system changes during adaptation when: (i) a server is being activated; and (ii) server fidelity is being reduced. In order to identify the included scenarios, we followed the procedure described in Sect. 4.2.2. Concretely, scenarios based on non-applicable change types were initially discarded (i.e., those which are never used in adaptation strategies to reduce response time that we are comparing, such as increasing server fidelity to improve the quality of the contents being served, or disconnecting servers to reduce the operation cost of the system). Secondly, the exposure matrix illustrated in Table 2 was applied to the set of remaining candidate changes as illustrated at the end of Sect. 4.2.2.

Table 3 displays the results obtained from our experiments. The quantities appearing in the cells of the main body of the table correspond to the probability of satisfying the property (either P1 or P2) of the corresponding column in the scenario described for that row. This probability is followed by the difference with the probability of satisfying the same property with respect to the one obtained for the normal execution of the system (i.e., just using the environmental changeload - row A -). The probabilities displayed in the table have been quantified using probabilistic models synthesized from sets of 30 different executions for each scenario included. Focusing on P1, we can observe that while strategies s1 and s2 are resilient to increases in load (B.2 and B.3) or crashes in the server being activated (B.1) (behaving in some cases even better than in executions without system changes - A), s3 performs worse, presenting a drop

Table 3 Experimental results

Scenario description	Property P1: $\mathcal{P}(F^{\leq 10}\neg\text{cViolation})$			P2: $\mathcal{P}(\neg\text{hiCost}\ U^{\leq 10}\ \neg\text{cViolation})$		
	s1	s2	s3	s1	s2	s3
A. Normal execution	0.7	0.33	0.76	0.7	0.33	0.69
B. When a server is activated						
1. Crash in the server being activated	0.71 (+0.01)	0.62 (+0.29)	0.62 (−0.14)	0.71 (+0.01)	0.62 (+0.29)	0.5 (−0.19)
2. Load increase in serverbeing activated	0.7 (+0)	0.42 (+0.09)	0.28 (−0.48)	0.7 (+0)	0.28 (−0.05)	0.28 (−0.41)
3. Load increase in a different server	0.37 (−0.33)	0.5 (+0.17)	0.71 (−0.05)	0.25 (−0.45)	0.5 (+0.17)	0.71 (+0.02)
C. When fidelity is being reduced						
1. Bug in the execution of the tactic to change fidelity	0.16 (−0.54)	0 (−0.33)	0.25 (−0.51)	0 (−0.7)	0 (−0.33)	0.25 (−0.44)
2. Load increase in server where fidelity is reduced	0.33 (−0.37)	0.57 (+0.24)	0.63 (−0.13)	0.33 (−0.37)	0.57 (+0.24)	0.54 (−0.15)
3. Load increase in another server	0.42 (−0.28)	0.54 (+0.21)	0.5 (−0.26)	0.42 (−0.28)	0.54 (+0.21)	0.5 (−0.19)

in the probability of satisfaction of P1 of up to 48 % when the server being activated is heavily loaded. An interesting phenomenon is that, although initially it may seem somewhat strange that some strategies yield better probability of satisfaction of the property in the presence of changes such as server crashes, it is worth observing that the decision-making process in Rainbow regarding which tactics are applied takes place at run-time during strategy execution, so adverse system conditions can result in the selection of a different sequence of tactics with respect to the case in which adaptation is not influenced by unfavorable changes (this may result in a performance boost, since the strategy will try to compensate the adverse situation by carrying out additional actions to improve execution). An example of this kind of situation can be observed for instance in the code of strategy ReduceResponseTime in Sect. 3: while in a situation with normal load conditions, only tactic switchToTextualMode (line 17) will be selected for execution, in a situation in which load is above established levels, tactic enlistServer (line 20) may also be executed in addition to the previous one, since the condition for execution hiLoad holds in this case, yielding a better overall result.

In scenarios C, where the fidelity of servers is being reduced, it is worth highlighting case C.1, in which there is a bug in the tactic to change fidelity (i.e., the fidelity in servers remains unchanged). Concretely, it can be observed that the impact on the probability is disastrous compared with other scenarios, such as in the case of s2 and s1 (with probabilities P1 of 0 and 0.16, respectively). This adverse effect is explained because the tactic to change fidelity is transversal to all servers, in contrast with the tactic to enlist servers, which deals with a typically reduced number of servers.

Summarizing, in scenarios in which adverse conditions are given when a server is activated, strategies s1 and s2 tend to perform better, except in cases in which the tactic to change fidelity does not work, in which s3 presents slightly better results.

Property P2, also takes into consideration the cost of operating the system, in addition to response time. In general, we can observe that including in the property does not alter significantly the values obtained with respect to P1. In particular, we can observe that s3 is slightly more costly in cases A.1 and C.2 (something that is consistent with the fact that s3 is more resilient to bugs in change fidelity, since this strategy makes a more profuse use of server activation instead of fidelity reduction, and this kind of tactic has an associated increment in cost).

6 Related work

In this section, we present related work regarding the usage of probabilistic model-checking in self-adaptive systems, and how benchmarking provides a practical way for characterizing and comparing systems according to specific characteristics.

Many methodologies support the analysis of non-functional properties, based either on modeling, or direct measurement of an existing system. While modeling is a useful approach to help in the development of a system for which there is no available implementation yet, it heavily relies on parameter estimations obtained either from domain experts, or from other similar existing systems. As an example, Calinescu and Kwiatkowska [6] introduce an autonomic architecture that uses Markov-chain

quantitative analysis to dynamically adjust the parameters of an IT system according to its environment and goals. However, their work assumes that Markov-chains describing the components/services of a system are readily available.

In the case of direct measurement, Epifani et al. [13] present a methodology and framework to keep models alive by feeding them with run-time data that updates their internal parameters. The framework focuses on reliability and performance, and uses Discrete-Time Markov Chains (DTMCs) and Queuing Networks as models to reason about non-functional properties. A complementary approach is taken by Calinescu et al. [5], who extends and combines [6] and [13] for defining a tool-supported framework for the development of adaptive service-based systems. QoS requirements are translated into probabilistic temporal logic formulae used for identifying and enforcing optimal system configurations. Our approach focuses on quantitative analysis using measurements, and not assuming the existence of Markov-chains describing components/services of the system. Moreover, while most proposals deal with estimates of the future system behavior for optimizing its operation, our approach focuses on providing levels of confidence with respect to the self-adaptive capabilities of the system. A preliminary work related to quantitative analysis using probabilistic model-checking presented an approach that considers only environment stimulation as source of change [7], leaving out changes that are internal to the system, which are dealt with in the present paper by exploiting architectural system models.

A benchmark is a standard procedure that allows characterizing and comparing systems or components according to specific characteristics (e.g., performance, dependability) [18]. Previous work on computer benchmarking can be divided in three main areas: performance benchmarking [16], dependability benchmarking [18], and security benchmarking [22]. Resilience benchmarking represents a step further, and is bound to encompass techniques from these previous efforts due to its inherent relation to performance, dependability and security. Comparing to established benchmarks, a resilience benchmark may be specified following the same basic approach, but comprising a wide-ranging changeload (which will include, but will be not limited to, faults), as well as resilience metrics [2]. In this paper, we revise and expand the definition of changeload, and apply our risk-based approach in the evaluation by comparison of the adaptation mechanisms of a self-adaptive software system.

7 Conclusions

A major challenge regarding the development, deployment and operation of self-adaptive software systems is the provision of assurances that system is able to deliver a service that can justifiably be trusted when facing changes, in other words, whether the system is resilient. The main reason for this challenge is the fact that it is difficult to foresee what changes can affect the system, its environment or goals, and when and where these changes might occur. In this paper, we have defined and developed a novel architecture-based approach for evaluating the resilience of self-adaptive software systems, which is based on the identification of changeload scenarios (i.e., representative sequence of environmental and system changes) that are used for

performing system run-time stimulation. The feasibility and effectiveness of the proposed approach was demonstrated in the context of Rainbow, a platform for architecture-based self-adaptation, and employing the Znn.com as a case study. The outcome of the experiments performed has shown the effectiveness of evaluating, by comparison, the resilience of adaptation mechanisms when employing probabilistic behavioral models for quantifying the probability of system properties being satisfied when subjected to stimuli.

As for any complex system, one cannot rely on a single source of evidence for the provision of assurances, instead one should use diverse sources. This is essentially how the proposed approach should be perceived regarding the evaluation of resilience of self-adaptive software systems, it should be considered in the context of other approaches for the provision of complementary evidence. Moreover, we also recognize that the usage of changeload scenarios may not cover the wider range of changes that may occur. That is essentially the motivation for combining techniques from resilience benchmarking with probabilistic model checking for guiding the identification and selection of representative changeloads.

As future work, there are several avenues to explore. Although the Znn.com has shown to be a very realistic case study for developing and evaluating the proposed approach, nevertheless there is the need to apply our approach to other systems, specially, industrial scale systems. This would allow to evaluate effectively the proposed approach in a context where the norm is to manage several sophisticated changeloads scenarios. The ultimate challenge regarding the proposed approach is to enhance it for enabling its deployment at run-time with minimal support from a developer. In such a setting, one needs to devise the appropriate environment for run-time stimulation while the system provides services, including the deployment of protections, so that stimulation does not affect the services delivered by the system.

Appendix A: Probabilistic computation-tree logic

To express resilience properties about the system, we use PCTL [4], which is a logic language inspired by CTL [4]. Instead of the universal and existential quantification of CTL, PCTL provides the probabilistic operator $\mathcal{P}_{\bowtie p}(.)$, where $p \in [0, 1]$ is a probability bound and $\bowtie \in \{\leq, <, \geq, >\}$. Given a time bound $t \in \mathbb{N}$, PCTL is defined by the following syntax:

$$\Phi ::= true \mid a \mid \Phi \wedge \Phi \mid \neg\Phi \mid \mathcal{P}_{\bowtie p}(\psi)$$
$$\psi ::= X\Phi \mid \Phi U \Phi \mid \Phi U^{\leq t} \Phi$$

Formulae Φ are named state formulae and can be evaluated over a Boolean domain $(true, false)$ in each state. Formulae ψ are named path formulae and describe a pattern over the set of all possible paths originating in the state where they are evaluated. In state formulae, other Boolean operators, such as disjunction ($\vee$), implication ($\Rightarrow$), etc. can be specified based on the primary Boolean operators. Moreover, we employ the abbreviations: (bounded) finally ($F^{(\leq t)}\Phi = true\ U^{(\leq t)}\Phi$) and globally ($G\Phi = \neg F \neg \Phi$). The satisfaction relation for PCTL is defined for a state s as:

$s \models true$ $\qquad$ $s \models a$ iff $a \in L(s)$
$s \models \neg\Phi$ iff $s \not\models \Phi$ $\qquad$ $s \models \Phi_1 \wedge \Phi_2$ iff $s \models \Phi_1$ *and* $s \models \Phi_2$
$s \models \mathcal{P}_{\bowtie p}(\psi)$ iff $Pr(s \models \psi) \bowtie p$

A formal definition of how to compute probability $Pr(s \models \psi)$ is presented in [4].The intuition is that its value corresponds to the fraction of paths originating in s and satisfying ψ over the entire set of paths originating in s. The satisfaction relation of a path formula with respect to a path π that originates in state $s(\pi[0] = s)$ is:

$\pi \models X\Phi$ iff $\pi[1] \models \Phi$
$\pi \models \Phi U \Psi$ iff $\exists j \geq 0.(\pi[j] \models \Psi \wedge (\forall 0 \leq k < j.\pi[k] \models \Phi))$
$\pi \models \Phi U^{\leq t} \Psi$ iff $\exists 0 \leq j \leq t.(\pi[j] \models \Psi \wedge (\forall 0 \leq k < j.\pi[k] \models \Phi))$

Appendix B: Modelling probabilistic system response

This appendix introduces the relevant concepts and procedure followed to build the two kinds of probabilistic models used in our approach, namely: (i) Operational profile model, used to quantify the probability of satisfaction of a property specified as a PCTL formula in a given operational profile, and (ii) Impact model, employed to quantify the impact of a particular change in the system or its environment on the different quality dimensions considered in the system.

For the sake of clarity, we first define a set of concepts that will help to follow the discussion throughout the rest of this section.

Definition 16 (*Boundary*) The *boundary* of an operational profile P is defined as:

$$boundary(P) = \{s_P \in P \mid \exists s \longrightarrow s_P : s \in S \backslash P\}$$

Definition 17 (*Trajectory*) A *trajectory* is a finite sequence of transitions:

$$\pi = s_1 \longrightarrow \ldots \longrightarrow s_m.$$

We assume that for all transitions $s \longrightarrow s'$, the time needed by the system to move from state s to state s' is given by $\tau \in \mathbb{R}^+$. Then, for any trajectory of the system π, we define its *duration* as $duration(\pi) = m\tau$.

A state $s' \in S$ is *reachable* from a state s in time t (denoted as $s \longrightarrow t^t s'$) if there exists a trajectory $\pi = s \longrightarrow \ldots \longrightarrow s'$, such that $duration(\pi) \leq t$.

B.1 Operational profile model

In operational profile models, initial states correspond to those where the system enters the associated operational profile, and the state space of the model includes all the states reachable from initial states within the specified time frame determined by a time bound t.

Definition 18 (*Operational profile model*) A model for an operational profile P and a collection of variables $X = \{x_1, \ldots, x_n\}$ during the time frame $[0, t]$ is a DTMC $\mathcal{M}_t^P$ built over $[\mathbb{R}^n]_X$, such that:

- Set of initial states $S_0^{\mathcal{M}} = boundary(P)$;
- Set of states $S^{\mathcal{M}} = \{s \in S \mid \exists s_0 \in S_0^{\mathcal{M}} : s_0 \longrightarrow t^t s\}$;

Algorithm 1 DTMC_synthesis

Input: Set of traces T associated with the time frame $[0, t]$, set of propositions AP.
Output: DTMC (S, S_0, P, L) associated with T.
Auxiliary variables: Transition observation matrix O.

1: $S := \emptyset$; $S_0 := \emptyset$; $P := 0$; $L := \emptyset$;
2: $O := 0$;
3: **for** $tr \in T$ **do**
4: $current := \varnothing$;
5: $is\_first := true$;
6: **while** $tr \neq \langle\rangle$ **do**
7: $prev := current$;
8: $current := extract(tr)$;
9: **if** $is\_first \wedge current \notin S_0$ **then**
10: $S_0 := S_0 \cup \{current\}$;
11: $is\_first := false$;
12: **end if**
13: **if** $current \notin S$ **then**
14: $S := S \cup \{current\}$;
15: $L := L \cup \{(current, \{\alpha \in AP \mid current \models \alpha\})\}$;
16: **end if**
17: **if** $prev \neq \varnothing$ **then**
18: $O(prev, current) := O(prev, current) + 1$;
19: $SC := \{s' \in S \mid O(prev, s') \neq 0\}$;
20: **for** $s' \in SC$ **do**
21: $P(prev, s') := O(prev, s') / \sum_{sc \in SC} O(prev, sc)$;
22: **end for**
23: **end if**
24: **end while**
25: **end for**
26: **return** (S, S_0, P, L);

Algorithm 1 synthesizes a DTMC from a set of traces T of the form $\langle s_1, \ldots, s_m \rangle$, where each state $s_{j \in \{1,\ldots,m\}} \in [\mathbb{R}^n]_X$, and the trace length is given by $m \geq t/\tau$. Input also includes the set of atomic propositions of interest AP to label the DTMC. The algorithm starts by incrementally building the sets of states (initial and global, lines 10, 14), and their corresponding labelling (line 15). Moreover, for the second part of the algorithm where the transition probability matrix is built, we employ an auxiliary matrix O that is used to keep information about the number of times that a transition between any two states $prev$ and $current$ is observed (line 18). Hence, values in the transition probability matrix are updated according to this information by assigning to each of the successors of state $prev$ (i.e., all states $s' \in S$ s.t. $O(prev, s') \neq 0$) a value proportional to the number of times that the transition $prev \longrightarrow current$ has

been observed w.r.t. the total number of transitions observed from source state $prev$ (line 21). Function $extract$ (line 8) returns and removes the first element of a vector

$$V = \langle e_1, e_2, \ldots, e_n \rangle : extract(V) = e_1,\ with\ V = \langle e_2, \ldots, e_n \rangle.$$

A model of a specific operational profile P can be synthesized from a set of traces obtained from the observation of the running system, where initial states are in $boundary(P)$, and trace length corresponds to the chosen time frame for the model [7]. The probability of satisfaction of properties expressed as PCTL formulas instanced using probabilistic response patterns as described in Table 1 can be directly quantified against operational profile models, provided that the time bound associated to the property is smaller or equal to that of the time frame associated with the model.

B.2 Impact model

Impact models describe the evolution of the different system variables considered for adaptation over a particular time frame, starting from an initial condition Ψ (e.g., associated with an event). Concretely, every state s is labeled with a reward for each variable that consists of the difference between the value of the variable in s, and the value for the same variable in the closest initial state of the model. Hence, impact models describe the response of the system, i.e., evolution over time of the deviation of system variables (impact), caused by the occurrence of an event (e.g., a change) in the system or its environment. System variables comprised in impact models are those relevant to the satisfaction of the system's goals and can be traced back to properties in the architectural model.

Definition 19 (*Impact model*) An impact model from an initial condition Ψ (expressed as a PCTL formula) and a collection of variables $X = \{x_1, \ldots, x_n\}$, during the time frame $[0, t]$ is a DMRM $\mathcal{I}_t^{\Psi} = (D, \langle \rho_1, \ldots, \rho_n \rangle)$ such that:

- $D = (S_0^{\mathcal{I}}, S^{\mathcal{I}}, P^{\mathcal{I}}, L^{\mathcal{I}})$ is a DTMC built over the metric space $([\mathbb{R}^n]_X, \delta)$:
  - Set of initial states $S_0^{\mathcal{I}} = \{s \in S \mid s \models \Psi\}$;
  - Set of states $S^{\mathcal{I}} = \{s \in S \mid \exists s_0 \in S_0^{\mathcal{I}} : s_0 \longrightarrow t^t s\}$;
- Reward assignment function for each variable $x_{i \in \{1,\ldots,n\}}$ is $\rho_i(s) = s[i] - s_0[i]$, with $s \in S^{\mathcal{I}}$ and $s_0 \in S_0^{\mathcal{I}}$, s.t. $\forall s_0' \in S_0^{\mathcal{I}},\ \pi = s_0 \longrightarrow \ldots \longrightarrow s,\ \pi' = s_0' \longrightarrow \ldots \longrightarrow s :\ duration(\pi) \leq duration(\pi')$.

Let us remark that the reward associated with any initial state $s_0 \in S_0^{\mathcal{I}}$ is by definition 0 (i.e., $\rho_i(s_0) = 0,\ i \in \{1, \ldots, n\}$).

An impact model can be synthesized from a set of traces TR associated to an initial condition (e.g., a change). In this set, each trace is of the form $\langle s_1, \ldots, s_m \rangle$, has a length that corresponds to the time frame of the resulting impact model, the initial state satisfies the initial condition ($s_1 \models \Psi$), and each state $s_{j \in \{1,\ldots,m\}} \in [\mathbb{R}^n]_X$ contains reward assignments that correspond to the difference between the values of the variables in the first state of the trace and the (quantized) values of the variables measured at instant j (i.e., the impact of initial condition on variables at time instant j).

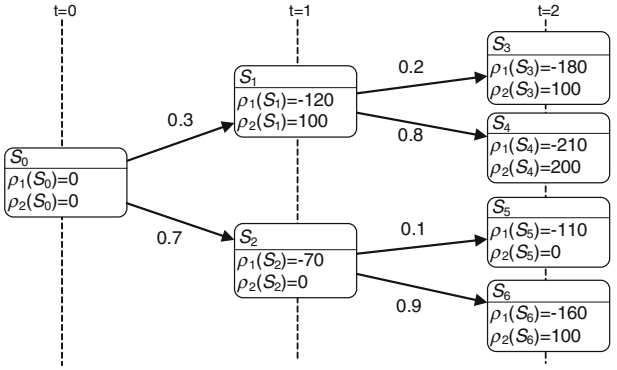

Fig. 8 Example impact model for Znn.com

In particular, let $TR_{s_j} = \{tr \in TR \mid s_j \in tr\}$ be the set of all traces in which the same arbitrary state s_j is observed. We define the impact in a trace k for variable i as:

$$impact_i^k(s_j) = quant(x_i^k(\tau j)) - s_1^k[i]; \quad \text{with } i \in \{1, \ldots, n\},\ j \in \{1, \ldots, m\}$$

where $x_i^k(\tau j)$ indicates the value of variable x_i at time instant j in trace k (always the same for all traces in TR_{s_j}), and $s_1^k[i]$ is the quantized value of variable x_i in the first state of the trace. The reward assignment for the variable x_i in a state s_j is computed as the average of all impacts for all observed occurrences of state s_j in different traces:

$$\rho_i(s_j) = \frac{\sum_{k=1}^{|TR_{s_j}|} impact_i^k(s_j)}{|TR_{s_j}|}$$

Expected impact over a set of system or environment variables, in a time instant that lies within the time frame associated with the model, can be directly checked through PRCTL formulas making use of the instantaneous reward operator $\mathcal{Y}^t_{\bowtie r}(\Phi)$ introduced previously in this section. Concretely, the instantaneous i-th reward for a time instant t_x can be quantified as $Ir^i(true, t_x)$ directly on an impact model.

Example 9 Figure 8 depicts a simple DMRM impact model for Znn.com that describes the evolution of variables `expRspTime` and `totalCost` for the time frame $[0, 2]$ (we assume that $\tau = 1$ for the sake of clarity). Every state displays the reward value assigned in that state by ρ_1 and ρ_2, that correspond to `expRspTime` and `totalCost`, respectively. Transitions are labeled with their associated probability. Focusing on response time, we can compute the expected impact in time instants 1 and 2 as:

$Ir^1(true, 1) = 0.3 * \rho_1(s_1) + 0.7 * \rho_1(s_2) = -85,$
$Ir^1(true, 2) = 0.3 * (0.2 * \rho_1(s_3) + 0.8 * \rho_1(s_4)) + 0.7 * (0.1 * \rho_1(s_5) + 0.9 * \rho_1(s_6)) = -166.1.$

Based on these values, we can check properties expressed in PRCTL such as: $\mathcal{Y}^{2,1}_{\leq -100}(true)$. This property reads as: "the expected reduction in response time after 2 seconds will be at least 100 milliseconds".

References

1. Abowd G, Allen R, Garlan D (1993) Using style to understand descriptions of software architecture. ACM Trans Softw Eng Methodol 4:319–364
2. Almeida R, Vieira M (2012) Changeloads for resilience benchmarking of self-adaptive systems: a risk-based approach. In: Proceedings of EDCC
3. Andova S, Hermanns H, Katoen J-P (2003) Discrete-time rewards model-checked. In: FORMATS of Lecture Notes in Computer Science, vol 2791, Springer, Berlin, pp 88–104
4. Baier C, Katoen J-P (2008) Principles of Model Checking. MIT Press, Cambridge
5. Calinescu R, Grunske L, Kwiatkowska MZ, Mirandola R, Tamburrelli G (2011) Dynamic QoS management and optimization in service-based systems. IEEE Trans Softw Eng 37(3):387–409
6. Calinescu R, Kwiatkowska MZ (2009) Using Quantitative Analysis to Implement Autonomic IT Systems. In: ICSE. Institute of Electrical and Electronics Engineers, MN, pp 100–110
7. Cámara J, de Lemos R (2012) Evaluation of Resilience in Self-Adaptive Systems Using Probabilistic Model-Checking. In: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2012), IEEE, pp 53–62
8. Cheng BHC, et al. (2009) Software Engineering for Self-Adaptive Systems: a Research Roadmap. In: SEfSAS of LNCS, vol 5525, Springer, Berlin, pp 1–26
9. Cheng S-W (2008) Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation. PhD thesis, Carnegie Mellon University, Pittsburgh
10. Cheng S-W, Garlan D, Schmerl BR (2009) Evaluating the Effectiveness of the Rainbow Self-Adaptive System. In: SEAMS, IEEE, Pittsburgh, pp 132–141
11. de Lemos R et al (2011) Software Engineering for Self-Adaptive Systems: a second Research Roadmap. In: de Lemos R, Giese H, Müller H, Shaw M (eds) Software Engineering for Self-Adaptive Systems, number 10431 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany
12. Dwyer MB, Avrunin GS, Corbett JC (1999) Patterns in Property Specifications for Finite-State Verification. In: ICSE, Cobleigh, pp 411–420
13. Epifani I, Ghezzi C, Mirandola R, Tamburrelli G (2009) Model Evolution by Run-Time Parameter Adaptation. In: ICSE,IEEE CS, Cobleigh, pp 111–121
14. Garlan D, Cheng S-W, Huang A-C, Schmerl BR, Steenkiste P (2004) Rainbow: architecture-based self-adaptation with reusable infrastructure. IEEE Comput 37(10):46–54
15. Garlan D, Monroe RT, Wile D (2000) Acme: architectural description of component-based systems. In: Leavens GT, Sitaraman M (eds) Foundations of Component-Based Systems, chapter 3, Cambridge University Press, Cambridge, pp 47–67
16. Gray J (1992) Benchmark Handbook: For Database and Transaction Processing Systems. Morgan Kaufmann Publishers Inc., San Francisco
17. Grunske L (2008) Specification Patterns for Probabilistic Quality Properties. In: ICSE, ACM, Hawthorn, pp 31–40
18. Kanoun K, Spainhower L (2008) Dependability Benchmarking for Computer Systems. Wiley-IEEE Computer Society Pr, Wiley
19. Kephart JO, Chess DM (2003) The vision of autonomic computing. Computer 36:41–50
20. Kwiatkowska M, Norman G, Parker D (2007) Stochastic model checking. Lecture notes in computer science. Springer, Berlin
21. Laprie J-C (2008) From Dependability to Resilience. In: DSN Fast Abstracts, IEEE CS, New York
22. Madeira H (2005) Towards a security benchmark for database management systems. In: Proceedings of the 2005 International Conference on Dependable Systems and Networks, DSN '05, IEEE Computer Society, Washington, pp 592–601
23. Oreizy P, Gorlick MM, Taylor RN, Heimbigner D, Johnson G, Medvidovic N, Quilici A, Rosenblum DS, Wolf AL (1999) An architecture-based approach to self-adaptive software. IEEE Intel Syst 14: 54–62
24. Shaw M, Garlan D (1996) Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Indiana
25. Williams R, Pandelios G, Behrens S (1999) Software Risk Evaluation (SRE) Method Description: Version 2.0. Technical report. Carnegie Mellon University, Software Engineering Institute, Pittsburgh