Adaptation Impact and Environment Models for Architecture-Based Self-Adaptive Systems

Javier Cámara[a], Antónia Lopes[b], David Garlan[a], Bradley Schmerl[a]

[a]*Institute for Software Research, Carnegie Mellon University, Pittsburgh, PA 15213, USA*
[b]*Department of Informatics, Faculty of Sciences, University of Lisbon, Portugal*

Abstract

Self-adaptive systems have the ability to adapt their behavior to dynamic operating conditions. In reaction to changes in the environment, these systems determine the appropriate corrective actions based in part on information about which action will have the best impact on the system. Existing models used to describe the impact of adaptations are either unable to capture the underlying uncertainty and variability of such dynamic environments, or are not compositional and described at a level of abstraction too low to scale in terms of specification effort required for non-trivial systems. In this paper, we address these shortcomings by describing an approach to the specification of impact models based on architectural system descriptions, which at the same time allows us to represent both variability and uncertainty in the outcome of adaptations, hence improving the selection of the best corrective action. The core of our approach is a language equipped with a formal semantics defined in terms of Discrete Time Markov Chains that enables us to describe both the impact of adaptation tactics, as well as the assumptions about the environment. To validate our approach, we show how employing our language can improve the accuracy of predictions used for decision-making in the Rainbow framework for architecture-based self-adaptation.

1. Introduction

Self-adaptive systems have the ability to autonomously change their behavior in response to changes in their operating conditions, thus preserving the capability of meeting certain requirements. For instance, to provide timely response to service requests, a news website with self-adaptive capabilities can react to high response latencies by activating more servers, or reducing the fidelity of the contents being served [11, 22].

Deciding which adaptations should be carried out in response to changes in the execution environment requires that systems embody knowledge about themselves. Knowledge about the impact of adaptation choices on a system's properties is particularly important when the decision process involves comparing alternative adaptations at runtime, as is often the case [9, 18, 24, 27].

The effectiveness of the enacted changes, which affects the system's ability to meet its requirements, strongly depends on the accuracy of the analytical models that are used for decision making. Exact models, if attainable at all, tend to be quite complex and costly to obtain. As argued in [15], an alternative is to attend to the uncertainty underlying the knowledge models in the decision process. However, existing models used to describe the impact of adaptations are either unable to capture the underlying uncertainty and variability of such dynamic execution environments, or are not compositional and described at a level of abstraction too low to scale in terms of specification effort required for non-trivial systems.

In this paper, we address the specification of probabilistic models for architecture-based self-adaptive systems. The core of our approach is a declarative specification language for expressing complex probabilistic constraints over state transitions that is equipped with a formal semantics defined in terms of Discrete Time Markov Chains. This language provides the means for expressing architectural style-specific adaptation impact and environment models in a flexible and compact way. These models can be reused across different architectural configurations that adhere to the same style.

We illustrate how the proposed models can be used in the context of the Rainbow framework [18] for architecture-based self-adaptation, where adaptation is achieved through the execution of an adaptation strategy selected at runtime. First, we present a technique for predicting the expected impact of an adaptation strategy and show that this can be used to define a strategy selector that seeks to maximize the expected utility. Then, based on the decoupling of adaptation impact from environment assumptions, we present a technique for predicting the guaranteed (i.e., worst-case) impact of a strategy and show this can be used to define a risk-averse strategy selector. This technique relies on a fine-grained semantics of strategies defined in terms of two-player stochastic games and is formulated as a process of stochastic game analysis.

We also present experimental results that quantify the benefits of using probabilistic impact models instead of constant impact vectors [9] in the context of Znn.com [10], a case study extensively used in the area of self-adaptive systems.

This paper revises and extends the work presented in [6] by addressing the

explicit modeling of assumptions about the environment in specifications, independent from those about adaptation impact. Based on a notion of environment model, we provide a turn-based game semantics of adaptation strategies and show how probabilistic model checking can be used to develop a risk-averse strategy selector. This strategy selection scheme goes beyond the original one employed by Rainbow, which is based on maximizing a probabilistic notion of aggregate utility [9]. This extended version of the work also includes additional details in the formalization of the semantics of our impact model language. We also report on the new experiments that we carried out to analyze the benefits of the proposed models.

The rest of the paper is organized as follows. Section 2 presents some background on architecture-based self-adaptation and discusses related work. Section 3 provides a formal account of the concepts required to define impact models. Section 4 presents the syntax and semantics of a new specification language of probabilistic impact models and Section 5 shows how our impact models can be used in the context of Rainbow for adaptation strategy selection. Section 6 shows how adaptation impact can be decoupled from environment assumptions through the definition of environment models and, based on this separation, shows how a risk-averse selector can be defined. Next, experimental results that quantify the benefits of using probabilistic impact models instead of impact vectors are presented in Section 7. Finally, Section 8 presents some conclusions and outlines future work.

2. Background and Related Work

In this section we provide some background on architecture-based self-adaptation, introduce the running example used in the rest of the paper and discuss related work. The ultimate goal is to motivate the need for new, more-expressive impact models, tailored to architecture-based self-adaptive systems.

2.1. Architecture-based adaptation

Architecture-based self-adaptation focuses on using architectural models at run-time as the central abstraction for observation, reflection and adaptation of self-adaptive systems. At run time, the system is monitored through a variety of probes. The observations provided by these probes are reflected in the architectural model of the system, which is used to determine when problems exist and to decide which changes in the architecture of the system should be carried out. The decision making is based on the knowledge the system has about its current state,

the state of the environment, and the impact of different adaptations on the system and its environment.

Through the example of Znn [10], a case study which has been extensively used to assess research in self-adaptive systems, we illustrate the key ideas of architectural-based adaptation — namely the use of the architectural style of the system as a basis for the adaptation, and explain the problem we address in this paper.

Znn case study. Znn.com reproduces the typical infrastructure for a news website. The system has a three-tier architecture consisting of a set of servers that provide content from backend databases to clients via front-end presentation logic. It uses a load balancer to balance requests across a pool of replicated servers, the number of which can be adjusted according to service demand. A set of clients makes stateless requests, and the servers deliver the requested contents.

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 budget. From time to time, due to highly popular events, Znn.com experiences spikes in the number of requests, which it cannot serve adequately, even at maximum pool size. To prevent loss of customers, the system can provide minimal textual content during such peak times, to avoid not providing service to some of its customers.

More concretely, there are two main quality objectives for the self-adaptation of the system: (i) performance, which depends on request response time, server load, and network bandwidth, and (ii) cost, which is associated with the power consumption of active servers.

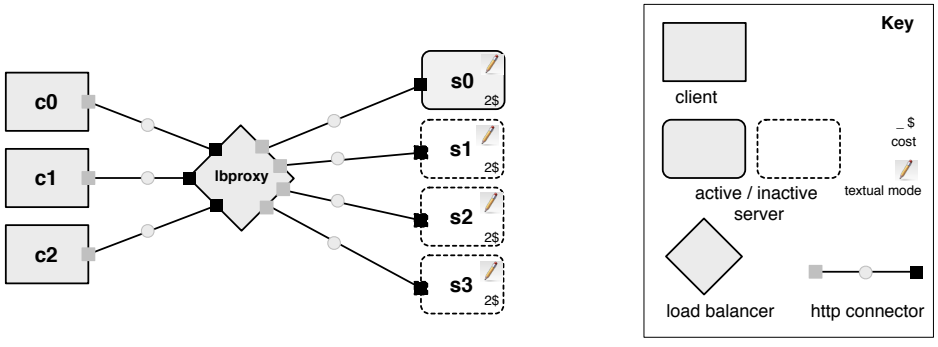

Figure 1: An architectural configuration for Znn.com

For an architecture-based self-adaptive realization of Znn.com we consider architectural models like the one presented in Fig. 1, consisting of several clients and replicated servers connected to one load balancer through http connectors. The architectural models additionally reflect (1) the cost of each server per unit time (if active), (2) whether the server is configured to serve web pages in textual or multimedia mode (if active), and (3) the observations provided by probes that monitor the health and load of each server, request response time and network bandwidth. The configuration presented in Fig. 1 has three clients and four servers; only one server is active, all have the same cost, and all are configured to serve pages in textual mode (for the sake of readability, the values of the other properties are not represented in the figure).

The characteristics of the architectural models described above are part of Znn architectural style, which additionally defines that the system is able to (1) activate an inactive server (incrementing in this case its server pool size), and (2) switch all active servers to textual or multimedia mode. The execution of these adaptation actions is expected to have certain costs and benefits. For instance, we expect that switching all active servers to textual model will contribute to decreasing the response time at the expense of the quality of the served content. In contrast, activating one server (which is known not to always succeed) will increase the operational cost. When response time becomes too high, in order to decide how to best adapt the system, it is essential to have realistic analytical models capturing how the system and the environment will respond to the execution of these two actions, at least in terms of response time, number of active servers and operational cost.

In order to have the ability to represent adaptation impact in a realistic way, it is important to support the representation of: (i) uncertainty in the outcome of adaptation actions (e.g., the activation of a server can fail with some given probability), (ii) context variability (e.g., the impact on response time of activating a single server will progressively reduce with a growing number of active servers), and (iii) assumptions about the evolution of the environment during the execution of adaptations (e.g., probability of a server crash). These three dimensions are also important because partial observability arises commonly in self-adaptive systems due to costs associated with monitoring certain parts of the system (e.g., in terms of performance), or because some parts of the system are not under the control of the adaptive system (e.g., managed by other organization).

Adaptation actions are defined for a family of systems, all of which are described according to a given architectural style. Hence, it is important to support the modeling of the impact of adaptation actions on system and environment state

at the same level of abstraction and in a compositional way.

2.2. Related Work

Environment domain models[1] are a key element used by adaptive systems to determine their behavior [4, 28]. These models capture the knowledge that the system has about itself and its environment by describing how system and environment respond to adaptation actions. Approaches to self-adaptation can be divided into two categories, depending on the way in which environment domain models are built.

A first category takes a systematic approach to modeling the impact of individual adaptation actions, which can be composed to reason about system behavior under adaptation. An example is the approach presented in [9], developed around Stitch, a language that enables the specification of adaptation strategies composed of individual adaptation actions. The impact of these actions is specified in terms of constant impact vectors which describe how the execution of adaptation actions affects system quality attributes. The same type of impact model is used in several approaches for optimizing service compositions, such as the approach presented in [24]. Adaptation actions in this approach target service composition instances and the optimal criteria rely upon impact models that are defined per adaptation action and system property as constant functions. Slightly more expressive impact models are considered in the approach presented [27], which targets component-based systems where impact models are defined per adaptation action and key performance indicators (KPIs), as functions over a given set of KPIs. These approaches address the specification of environment domain models in a compositional way and at a very high level of abstraction, thus facilitating specification and promoting reuse. However, they severely limit the ability to represent environment domain knowledge in a realistic way, since they are unable to model uncertainty and provide limited support to capture variability.

The second category consists of approaches that consider the behavior of the system and its environment modeled in a monolithic way in terms of more expressive models defined at a lower level of abstraction [4, 5, 16]. For example, in the approach presented in [5], DTMCs are used to model, for each system configuration, the probabilistic behaviour of the system and its environment if that configuration is used. These models are expressive enough to model variability and the uncertainty underlying adaptation outcomes.

[1]Note that when we employ the term *environment domain model*, we refer to a description of both the external environment, as well as of the system that we are attempting to control.

Beyond the scope of self-adaptive systems, we can find other languages like MODEST [20] or ARCADE [3], which share some similarities with the aforementioned approaches included in this second category, since they are expressive enough to capture probabilistic behavior and non-determinism. An advantage of these languages with respect to other formalisms is that they raise the level of abstraction with respect to low-level specification. MODEST [20] does so by employing a concise notation that inherits some characteristics from process algebras. Moreover, MODEST enables compositional analysis and its semantics is defined in terms of stochastic timed automata (STA), which subsume DTMCs. Although MODEST abstracts away from the complexity of low-level specification of probabilistic behavior, it lacks the abstractions required to capture relationships between behavior and architecture. ARCADE [3] is another example of a probabilistic language which forms part of a framework intended for dependability evaluation, and whose semantics are defined in terms of interactive Markov chains. While ARCADE can be used to represent dependability-related attributes of system components, its models cannot capture other aspects of architectural descriptions (e.g., component relationships or constraints), and hence cannot capture the impact of system-wide changes.

The main drawback in this second category of models is that, independently of their level of abstraction, they are being defined at the level of system configurations, and hence are system specific. For instance, in the case of Znn.com, a DTMC that models the effect of activating one server in a system that can use up to 4 servers is completely different from another that models the same on a system that can use up to 10 servers, although the effect of activating one server does not depend on the maximum number of servers.

As the number of components and configuration options increases, it becomes impractical or even impossible to understand the possible interactions between the options available for all components and, hence, these models are both difficult and cumbersome to define. The specification of a DTMC tends to be a non-trivial task, even using description languages such as the one built into the probabilistic model checker PRISM [23].

The approach described in this paper aims at striking a balance between the ease of specification and reusability found in compositional approaches, and the expressive power of monolithic approaches that use probabilistic models. We present a language for the specification of impact models, which is: (i) more intuitive than describing DTMCs in other probabilistic approaches, since it is based on architectural descriptions and therefore raises the level of abstraction, (ii) able to capture both variability and probabilistic outcomes of adaptation actions, and

(iii) scalable in terms of specification effort, since developers can focus on smaller units of conceptualization (architectural properties and adaptation actions) and reason about them individually.

3. Modeling Adaptation

In this section, we provide a formal account of the concepts required to define impact models, namely architectural style and system state.

We address the modeling of impact in the context of architecture-based approaches to self-adaptation, that take the architectural style of the managed system as a basis for the system adaptation. As discussed in the previous section, the aim is to support the specification of impact models for families of systems that share the same architectural style.

3.1. Architectural Style

When the architectural style of the managed system is taken as a basis for the system adaptation, it has to define not only the class of models to which the managed system architecture belongs, but also to determine the operators representing available configuration changes on systems in that style. Moreover, the style prescribes what aspects of a system and its execution context need to be monitored [12].

An architectural style defines a vocabulary of component and connector types that can be used in instances of that style and the properties of each of these types. In the context of self-adaptive systems, it is essential to distinguish between managed and monitored properties. *Managed properties* correspond to properties that are directly under system control. Their values can be defined at startup and changed subsequently by the control layer to regulate the system. *Monitored properties* correspond to properties of the managed system or its execution context that need to be monitored and made available to the control layer. While the properties of the execution context are not under direct system control (e.g., available bandwidth), monitored properties also include those that the system aims to control (e.g., response time).

As an example, we consider the architectural style of Znn.com. It has one connector type — HttpConnT, and three component types — ClientT, ServerT and LoadBalancerT. ServerT has two managed properties — isTextualMode:bool and cost:int. The former defines whether web pages are served by a given server in textual or multimedia mode, and the latter reflects the cost of an active server

per unit time. Since these properties are defined as being under the system's control, the cost of each server included in the system must be defined at deployment time, as well as whether it will initially start serving pages in textual or multimedia mode. Additionally, ServerT has two monitored properties — load:double and isActive:bool. The latter property is defined as monitored, since even if its activation can be controlled, a server may crash at any time.

Formally, architectural signatures are defined as follows.

Definition 1 (Architectural Signature). *An architectural signature Σ consists of a tuple of the form $\langle CompT, ConnT, \Pi^o, \Pi^m \rangle$,where:*

- *CompT and ConnT are two disjoint sets (the sets of, respectively, component and connector types), and*
- *Π^o and Π^m are functions that assign mutually disjoint sets of symbols typed by datatypes in a fixed set $\mathcal{D}$ to architectural types $\kappa \in CompT \cup ConnT$.*

Note that $\Pi^o(\kappa)$ and $\Pi^m(\kappa)$ represent, respectively, the managed and monitored properties of the type κ. Moreover, we abbreviate $\Pi^o(\kappa) \cup \Pi^m(\kappa)$ by $\Pi(\kappa)$ and, for $p \in \Pi(\kappa)$, will use $dtype(\kappa.p)$ to denote its datatype.

We write $x : d$ to denote an element x typed by the datatype d. Since property names are not necessarily unique, we use $\kappa.p : d$ to refer to the property $p : d$ in $\Pi(\kappa)$; sometimes the type is not relevant and we write simply $\kappa.p$.

In the signature of Znn.com, according to this definition, we have $\Pi^o(\mathsf{ServerT}) = \{\mathsf{load:double}, \mathsf{isActive:bool}\}$ and $\Pi^m(\mathsf{ServerT}) = \{\mathsf{isTextualMode:bool}, \mathsf{cost:int}\}$.

The architectural configurations of systems with architectural signature Σ, hereafter called Σ-system states, are captured in terms of graphs of components and connectors with state. The state of architectural elements consist of the values taken by their monitored and managed properties.

We formally define Σ-system states assuming there is a fixed universe $\mathcal{A}_\Sigma$ of architectural elements (components and connectors) for Σ, i.e., a finite set whose elements are typed by elements in $CompT \cup ConnT$. We use $type(c)$ to denote c's type.

Definition 2 (Σ-System State). *A* Σ-system state s *consists of:*

- *a simple graph $\mathcal{G}$ with nodes in $\mathcal{A}_\Sigma$,* [2]
- *a function that assigns a value $[\![c.p]\!]^s$ in the domain of $dtype(\kappa.p)$, to every pair $c.p$ such that c is a node of $\mathcal{G}$, $\kappa = type(c)$ and $p \in \Pi(\kappa)$.*

We denote by $\mathcal{S}_\Sigma$ (or simply $\mathcal{S}$ when Σ is clear from the context) the set of all Σ-system states.

An architectural style also defines the ways one can change systems that are instances of that style. For instance, the Znn architectural style defines that the property isTextualMode of servers can be modified through setLowFidelity and setHighFidelity operators, that set isTextualMode to true and false, respectively.

These operators can range from primitive operations, such as changing the value of a property of a given connector type, to higher-level operations that exploit restrictions of that style. However, in practice, most approaches to self-adaptation consider only primitive operations. Hence, we focus on architectural styles that define the set of operators provided by the target system for changing the values of the managed properties of its components and connectors. Notice that, in these architectural styles, only component $[\![\ ]\!]^s$ of a system state can change at run time. Hence, the structure of the system, defined by the graph, does not change in the class of system considered in this work.

3.2. Adaptation actions

The adaptation of the managed system is achieved through the execution of adaptation actions defined at design-time. Adaptation actions define actions specified as applications of one or more operators, with a condition of applicability. In the Znn example we could for instance define an adaptation action called switchToTextualMode, applicable only if there is at least one active service not serving pages in textual mode, prescribing the application of operator setLowFidelity to all servers in these conditions. A different adaptation action for Znn is enlistServer, which is applicable when there is at least one inactive server; it prescribes the application of startServer to one inactive server.

Applicability conditions of adaptation actions are formulas of a constraint language that are evaluated over system states, typically depending not only on the values of the properties of the system's components and connectors, but also on

[2] We consider graphs embodied in system states as a natural way of capturing architecture configurations, which are graph-based in standard architecture descriptions.

system structure. For illustration purposes, we consider a constraint language inspired by that of Acme [19] [3] in which, for instance,

```
(exists s:ServerT | exists k:HttpConnT | attached(k,s) and s.isActive)
```

holds in a system state iff the state includes at least one active server attached to one http connector.

4. A Language for Modelling Impact

Deciding how to best adapt the system when a certain anomaly is detected involves analyzing models describing the effects, in terms of costs and benefits, of the available adaptation actions defined for the system. These models capture the causal relationship between an adaptation action's execution and its impact on the different system and environment properties. Because 100% accurate models are in general not attainable, it is important to have a means by which to address the underlying uncertainty.

In this section, we describe an expressive language to model adaptation action execution, which is able to capture: (i) the context that might influence the outcome of an adaptation action's execution, and (ii) the intrinsic uncertainty that pervades self-adaptive systems. Specifically, this language enables the modeling of the expected impact of each adaptation action on the different system and environment properties. These models are based on DTMCs [25], and enable us to express alternative possible outcomes of the execution of the same adaptation action with some given probability.

In the following, we first present the syntax of our impact model language in Section 4.1, illustrating it with different examples. Then, we provide a formal description of its semantics in Section 4.2.

4.1. Impact Model Abstract Syntax

The impact model of adaptation actions is defined in terms of probabilistic expressions in a language that allows one to express probabilistic constraints over state transitions (regarded as pairs of before and after system states), incorporating some elements of the PRISM language [23].

[3] Acme is in turn derived from OCL [26], with the addition of functions that relate to architectural structure.

Since the language targets systems whose structure does not change at runtime, it is built on top of a language $\mathcal{E}$ of state expressions describing sets of components and connectors in system states. The syntax and semantics of $\mathcal{E}$ is abstracted away, so that the impact model language is defined in a more abstract way. For illustration purposes, we consider a language $\mathcal{E}$ in line with the one that we use to express constraints, in which, for instance, (s: ServerT | s.isActive) describes the set of active servers in a system state.

In the same way, to handle data, we assume that the fixed set of datatypes $\mathcal{D}$ is equipped with the relevant operations (e.g., + over double). Let X be a set of symbols typed by elements of $\mathcal{D}$ representing data variables. We denote by $\mathcal{T}$ the term language used to describe data values and by $\mathcal{T}_d(\Sigma, X)$ the set of terms built over variables in X denoting values of datatype d. Similarly, we consider sets $\mathcal{X}$ of symbols typed by architectural types in an architectural signature Σ representing type variables and use $\mathcal{E}_\kappa(\Sigma, \mathcal{X})$ to denote the set of expressions defined over the variables in $\mathcal{X}$ denoting sets of architectural elements of type κ.

Definition 3 (Probabilistic Expressions). *Let $\mathcal{X}$ be a set of variables typed by architectural types in an architectural signature Σ. The set $\mathcal{P}(\Sigma, \mathcal{X})$, of* probabilistic expressions *with free variables in $\mathcal{X}$, is defined by the following grammar:*

$$
\begin{array}{lr}
\alpha ::= x.p' = t & \textit{with } x{\in}\mathcal{X},\ p{\in}\Pi(\kappa),\ t{\in}\mathcal{T}_d(\Sigma, x_\Pi), \\
 & \textit{where } \kappa = type(x) \textit{ and } d = dtype(p) \\
\mid \textbf{forall}\ x : \epsilon \mid \alpha_1 & \\
\mid \textbf{foreach}\ x : \epsilon \mid \alpha_1 & \\
\mid \textbf{foreach}\ x : \epsilon\ \textbf{minus}\ D \mid \alpha_1 & \textit{with } x \notin \mathcal{X},\ \epsilon{\in}\mathcal{E}_\kappa(\Sigma, \mathcal{X}), \\
 & \alpha_1{\in}\mathcal{P}(\Sigma, \mathcal{X}{\cup}\{x : \kappa\}) \textit{ and } D{\subseteq}\mathcal{X}_\kappa \\
\mid \{\alpha_1 \& \ldots \& \alpha_n\} & \\
\mid \{[p_1]\alpha_1 + [p_2]\alpha_2 + \cdots + [p_n]\alpha_n\} & \textit{with, for } 1 \leq i \leq n,\ \alpha_i{\in}\mathcal{P}(\Sigma, \mathcal{X}), \\
 & 0 \leq p_i \leq 1 \textit{ and } \Sigma_{i=1}^{n} p_i = 1
\end{array}
$$

where $x_\Pi{=}\{\ x.p{:}d \mid p \in \Pi(type(x)) \wedge d = dtype(p)\}$ *and* $\mathcal{X}_\kappa{=}\{x{\in}\mathcal{X} : type(x) = \kappa\}$. $\mathcal{P}(\Sigma)$ *is the set of probabilistic expressions without free variables, i.e.,* $\mathcal{P}(\Sigma, \emptyset)$.

The atomic expression $x.p' = t$ defines the value of the property p in the next state (after the execution of the adaptation action), for every component or connector denoted by x. This value can be defined in terms of the values of the properties of the same element as well as other architectural elements in the system, but the free variables of t are limited to data variables representing properties of x (which is exactly what x_Π represents). For instance, assuming that s is a variable of type ServerT, we can write s.isActive' = !s.isActive to express that every server

denoted by s has its isActive property toggled. An example of a more sophisticated atomic expression, defined in terms of the values of properties of other architectural elements in the system, is x.p' = sum(s.q | s:ServerT)/count(s:ServerT). The expression states that, for all elements denoted by x, the value of property p in the next state is the average of the values of q for all servers.

The operator **forall** is used to impose the same constraints over a set of architectural elements of the same type, denoted by a given expression in $\mathcal{E}$. The operator **foreach** is used to define a number of alternative outcomes, all with the same probability. For instance, **foreach** x:ServerT | x.isActive' = true states that all servers have the same probability of having their isActive property set to true. Adding **minus** D to the expression reduces the target to elements not included in the denotation of variables in D. For instance,

foreach x:E | **foreach** y:E **minus** x | {x.isActive' = true & y.isActive' = true}

where E is (s:ServerT|!s.isActive), expresses that exactly two servers are activated and that all pairs of distinct inactive servers have the same probability of being activated.

A fixed number of constraints over the next state are expressed through conjunction ($\&$). Probabilities that sum to one are assigned to a fixed number of expressions defining constraints over alternative outcomes of the adaptation action execution. Assigning a probability to an expression with $[p]\alpha$ has the effect of world closure: all properties of components and connectors not constrained by α are considered to keep the same value in the next state.

To capture that an adaptation action may have different impacts under different conditions, impact models are defined as sets of guarded probabilistic expressions with mutually exclusive guards (i.e., at most one guard holds in any system state). As before, we abstract from the language used for expressing the guard conditions and assume a fixed language $\mathcal{C}$ of constraints over system states.

Definition 4 (Impact Model). *An impact model $\mathcal{I}$ of an adaptation action is a finite set of pairs $\langle \phi, \alpha \rangle$ where ϕ is a constraint in $\mathcal{C}(\Sigma)$ and α is a probabilistic expression in $\mathcal{P}(\Sigma)$ such that all ϕ are mutually exclusive.*

An example of a simple impact model is presented below for the adaptation action switchToTextualMode. For the sake of clarity, we present all examples making use of a concrete syntax that supports the definition of abbreviations and in which guarded expressions are represented as $\phi \rightarrow \alpha$.

```
1 define S=(s:ServerT | !s.isTextualMode and s.isActive)
2 define k=size(S)
```

```
3  define f(x)=x∗(1−k/(2∗(k+1)))
4  define g(x)=x∗(1−k/(k+1))
5  impactmodel switchToTextualMode
6   k>0 → { [0.8] { forall s:S | s.isTextualMode'=true & forall c:ClientT | c.expRspTime'=f(c.expRspTime) }
7           + [0.2] { forall s:S | s.isTextualMode'=true & forall c:ClientT | c.expRspTime'=g(c.expRspTime) }}
```

Listing 1: Impact model for adaptation action switchToTextualMode.

The model in Listing 1 expresses the impact of the adaptation action over manipulated properties. As mentioned before, this adaptation action prescribes the application of setLowFidelity to all active servers not serving pages in textual mode, where the fact that operator setLowFidelity sets the property isTextualMode to true is represented by s.isTextualMode'=true. Moreover, the model predicts that switchToTextualMode can impact the response time of *all clients* in two ways, both decreasing its value, considering the number of servers that were changed to low fidelity. The more severe reduction of the response time is defined to be the least likely, with probability 0.2. According to this (simplistic) model, the execution of this adaptation action is not expected to affect the remaining properties of servers, clients, or http connectors.

Alternatively, we could specify that switchToTextualMode can impact the response time of *each client* in two ways as follows:

```
1  impactmodel switchToTextualMode
2   k>0 → forall c:ClientT | { [0.8] { forall s:S | s.isTextualMode'=true & c.expRspTime'=f(c.expRspTime) }
3                            + [0.2] { forall s:S | s.isTextualMode'=true & c.expRspTime'=g(c.expRspTime) }}
```

Listing 2: Alternative impact model for switchToTextualMode.

While we have considered that the property isTextualMode of servers is subject only to system control, isActive was defined as a monitored property and it was considered that the activation of a server, through the execution of operator startServer, may fail. An impact model for enlistServer that captures this aspect is presented below.

```
1  define m=size(s:ServerT | s.isActive)
2  define S= (s:ServerT | !s.isActive)
3  define f(x)=x∗(1−((1/log(100∗m,2))∗(m/(2∗m+1))))
4  define g(x)=x∗(1−1/log(100∗m,2))
5  impactmodel enlistServer
6  m>0 → { [0.95] { foreach s:S | s.isActive'=true &
7                              { [0.7] forall c:ClientT | c.expRspTime'=f(c.expRspTime)
8                              + [0.3] forall c:ClientT | c.expRspTime'=g(c.expRspTime) }}
9          + [0.05] { forall c:ClientT | c.expRspTime'=c.expRspTime & forall s:ServerT | s.isActive'=s.isActive }}
```

Listing 3: Impact model for adaptation action enlistServer.

This impact model states that starting a server is expected to fail with probability 0.05 and predicts that the adaptation action may impact client response time in two ways, both considering the number of servers that were already active.

Moreover, isActive is also defined as a monitored property, since a server could become spontaneously inactive (e.g., due to a server crash). The impact model above does not define any impact of the adaptation action over the property isActive of already-active servers: hence the probability of an active server crashing while executing enlistServer is considered to be so small that it can be neglected. Alternatively, we can define the probability of each relevant crash scenario (e.g., for one server, two servers, etc). For instance, the impact model presented below defines that the probability of exactly one active server crashing while executing enlistServer is 0.001.

```
1 define T=(s:ServerT | s.isActive)
2 impactmodel enlistServer
3 m>0 → { [0.999] { foreach s:S | s.isActive'=true &
4                 { [0.7] forall c:ClientT | c.expRspTime'=f(c.expRspTime) +
5                   [0.3] forall c:ClientT | c.expRspTime'=g(c.expRspTime) } }
6         + [0.001] { foreach s:S | s.isActive'=true & foreach t:T | t.isActive'=false &
7                     forall c:ClientT | c.expRspTime'=c.expRspTime } }
```

Listing 4: Alternative impact model for adaptation action enlistServer.

4.2. Impact Model Semantics

The semantics of impact models is formally defined in terms of DTMCs. Since a DTMC has a discrete state space, we have to limit properties of components and connectors to take values in discrete sets and perform quantization for properties that would otherwise be continuous.

Quantization. For each property p that takes values in a datatype $d \in \mathcal{D}$ that has a non-finite domain $\mathcal{I}_d$, it is necessary that a finite set $[\mathcal{I}_d]_p$ and a quantization function $Q_p : \mathcal{I}_d \to [\mathcal{I}_d]_p$ be defined. For each property $p : d$ such that $\mathcal{I}_d$ is finite we take $[\mathcal{I}_d]_p = \mathcal{I}_d$ and Q_p as the identity function.

The quantization of the properties of component and connector types can be propagated to the level of system states, defining a finite set of states $[\mathcal{S}] = \{[s] : s \in \mathcal{S}\}$. In $[s]$, the value of a property p of a component or connector c is obtained by applying the corresponding quantization function to the value it has in s, i.e., $[\![c.p]\!]^{[s]} = Q_p([\![c.p]\!]^s)$.

The semantics of impact models is defined in terms of DTMCs over $[\mathcal{S}]$. We start by providing the semantics of the probabilistic expressions used to assemble such models.

The interpretation of a probabilistic expression α over a set of type variables $\mathcal{X}$ is defined in the context of a system state s and an *interpretation* ρ of $\mathcal{X}$ assigning, to each type variable $x{:}\kappa$, a set of elements in s of type κ. This interpretation,

denoted by $[\![\alpha]\!]^s_\rho$, consists of a set Y of properties of component and connectors in s — those which are constrained by α — and a function P defining the probability of transitions between any pair of Y-states. As an example, consider $[\![\mathsf{x.isActive'=true}]\!]^s_\rho$ where x is a type variable typed by ServerT, s is a state with servers $z_1, \ldots, z_n$ and $\rho : x \mapsto \{z_1\}$. The expression constrains only the property $isActive$ of z_1, i.e., $Y{=}\{z_1.isActive\}$ and, hence, in this case, a Y-state is just a truth value for $z_1.isActive$. Its interpretation is that the probability of a transition from any Y-state to $\{z_1.isActive{\mapsto}true\}$ is 1 and to $\{z_1.isActive{\mapsto}false\}$ is 0.

Formally, given a set Y of properties, a Y-state s is a function defining the value of each property in $y \in Y$, subject to the corresponding quantization functions. As for system states, we simply write $[\![y]\!]^s$ and use $[\mathcal{S}_Y]$ for referring to the set of all Y-states.

An important operation over probabilistic expressions is world closure through assignment of a probability. As mentioned before, when we write $[p]\alpha$, all properties of components and connectors not constrained by α are considered to retain the same value in the next state. World closure can be captured by the following notion of closure over transition probability functions:

Definition 5 (Closure). *Let $Y{\subseteq}Y'$ be two sets of properties. Given a function $P{:}[\mathcal{S}_Y] \times [\mathcal{S}_Y] \to [0,1]$, the closure of P to Y' is the function $P^{Y'}{:}[\mathcal{S}_{Y'}] \times [\mathcal{S}_{Y'}] \to [0,1]$ defined as follows:*

- *if* $Y \neq \emptyset$, $P^{Y'}(s_1, s_2) = \begin{cases} P(s_{1|Y}, s_{2|Y}) & \text{if } \forall y \in Y' \setminus Y,\ [\![y]\!]^{s_2} = [\![y]\!]^{s_1} \\ 0 & \text{otherwise} \end{cases}$
- *if* $Y = \emptyset$, $P^{Y'}(s_1, s_2) = \begin{cases} 1 & \text{if } \forall y \in Y',\ [\![y]\!]^{s_2} = [\![y]\!]^{s_1} \\ 0 & \text{otherwise} \end{cases}$

where $s_{|Y}$ is the Y-state obtained through the restriction of s to the properties in Y.

The closure of P corresponds to extending the probabilities given by P to states with more properties, considering that their values do not change.

Since, the language $\mathcal{P}$ is defined on top of a language of state expressions $\mathcal{E}$, a language of state constraints $\mathcal{C}$ and a language of data terms $\mathcal{T}$, its semantics relies on the semantics of these languages. In what follows we use:

- $[\![\epsilon]\!]^s_\rho$ to represent the set of architectural elements of type κ denoted by expression $\epsilon \in \mathcal{E}_\kappa(\Sigma, \mathcal{X})$ in state s under assignment ρ to the type variables in $\mathcal{X}$,
- $s \models \phi$ to denote that constraint $\phi \in \mathcal{C}(\Sigma)$ holds in state s,
- $[\![t]\!]^s_{\rho_d}$ to represent the data value provided by the evaluation of term $t \in \mathcal{T}_d(\Sigma, X)$ in state s under assignment ρ_d to its data variables.

Recall that the language $\mathcal{T}$ might include terms, such as

sum(s.q | s:ServerT)/count(s:ServerT)

that are state dependent and, hence, we consider that states are also required for the evaluation of terms.

Definition 6 (Interpretation of Probabilistic Expressions). *The interpretation $[\![\alpha]\!]^s_\rho$ of $\alpha \in \mathcal{P}(\Sigma, \mathcal{X})$ in a system state s and an interpretation ρ of $\mathcal{X}$ is a pair of the form $\langle Y, P{:}[\mathcal{S}_Y] \times [\mathcal{S}_Y] \to [0,1] \rangle$ defined inductively in the structure of α as follows:*

- $[\![x.p' = t]\!]^s_\rho = \langle Y, P \rangle$

  $$Y = \{c.p : c \in \rho(x)\} \text{ and } P(s_1, s_2) = \begin{cases} 1 & \text{if } \forall c \in \rho(x), [\![c.p]\!]^{s_2} = [\![t]\!]^{s_1}_{\rho_c} \\ 0 & \text{otherwise} \end{cases}$$

  $$\text{with } \rho_c = \{x.q \mapsto [\![c.q]\!]^{s_1} : q \in \Pi(type(x))\}$$

- $[\![\textbf{forall } x : \epsilon \mid \alpha]\!]^s_\rho = [\![\alpha]\!]^s_{\rho'}$ *with* $\rho' = \rho \oplus x \mapsto [\![\epsilon]\!]^s_\rho$ [4]

- $[\![\textbf{foreach } x : \epsilon \mid \alpha]\!]^s_\rho = \langle Y, P \rangle$

  Let $C = [\![\epsilon]\!]^s_\rho$. *If* $|C| = 0$, *then* $Y = \emptyset$ *and* P *is the empty function to* $[0,1]$. *Otherwise, let* $\rho_c = \rho \oplus x \mapsto \{c\}$, *for every* $c \in C$, *and* $[\![\alpha]\!]^s_{\rho_c} = \langle Y_c, P_c \rangle$.

  $$Y = \bigcup_{c \in C} Y_c \text{ and } P(s_1, s_2) = \sum_{c \in C} \tfrac{1}{|C|} \cdot P^Y_c(s_{1|Y_c}, s_{2|Y_c})$$

[4] Given an assignment ρ for $\mathcal{X}$, $\rho \oplus x \mapsto v$ represents the assignment for $\mathcal{X} \cup \{x\}$ that assigns the value v to x and assigns the value $\rho(y)$ to any other variable $y \neq x$ in $\mathcal{X}$.

- $[\![$**foreach** $x : \epsilon$ **minus** $D \mid \alpha]\!]^s_\rho = \langle Y, P\rangle$

  Let $C{=}[\![\epsilon]\!]^s_\rho \setminus \rho(D)$. *If* $|C| = 0$, *then* $Y = \emptyset$ *and* P *is the empty function to* $[0, 1]$. *Otherwise, let* $\rho_c{=}\rho \oplus x \mapsto \{c\}$, *for every* $c \in C$, *and* $[\![\alpha]\!]^s_{\rho_c} = \langle Y_c, P_c\rangle$.

  $$Y = \bigcup_{c\in C} Y_c \text{ and } P(s_1, s_2) = \sum_{c\in C} \tfrac{1}{|C|} \cdot P^Y_c(s_{1|Y_c}, s_{2|Y_c})$$

- $[\![\{\alpha_1 \& \cdots \& \alpha_n\}]\!]^s_\rho = \langle Y, P\rangle$

  Let $[\![\alpha_i]\!]^s_\rho = \langle Y_i, P_i\rangle$, *for* $i = 1, .., n$. *If the sets* $Y_1, \cdots, Y_n$ *are not mutually disjoint, then* $Y = \emptyset$ *and* P *is the empty function to* $[0, 1]$. *Otherwise,*

  $$Y = \bigcup_{i=1,..,n} Y_i \text{ and } P(s_1, s_2) = \prod_{i=1}^{n} P_i(s_{1|Y_i}, s_{2|Y_i})$$

- $[\![\{[p_1]\alpha_1 + \cdots + [p_n]\alpha_n\}]\!]^s_\rho = \langle Y, P\rangle$

  Let $[\![\alpha_i]\!]^s_\rho = \langle Y_i, P_i\rangle$, *for* $i = 1, .., n$.

  $$Y = \bigcup_{i=1,..,n} Y_i \text{ and } P(s_1, s_2) = \sum_{i=1}^{n} p_i \cdot P^Y_i(s_{1|Y_i}, s_{2|Y_i})$$

Notice that there are some state-dependent semantic restrictions over probabilistic expressions. If, in a given state s, an expression α does not meet these conditions, then α does not impose any restriction in the evolution of the system state (i.e., $Y = \emptyset$).

Proposition 1. *If* $[\![\alpha]\!]^s_\rho = \langle Y, P : [\mathcal{S}_Y] \times [\mathcal{S}_Y] \to [0, 1]\rangle$, *then* P *is a probabilistic transition function, i.e.,* $\sum_{s_2\in[\mathcal{S}_Y]} P(s_1, s_2) = 1$, *for every* $s_1 \in [\mathcal{S}_Y]$.

Proof *The proof proceeds by induction in the structure of expressions* α*:*

case $(x.p' = t)$: *In this case the result follows from the fact that the term* t *uniquely defines the values of all properties in* Y *in the next state. That is, for every* s_1 *in* $[\mathcal{S}_Y]$, *there is a single state* s *such that* $[\![y]\!]^s = [\![t]\!]^{s_1}_{\rho_c}$, *for every* $y \in Y$. *Hence, the probability of transition to* s *is* 1 *and is* 0 *for any other state different from* s, *and hence, the sum* $P(s_1, s_2)$ *for all states* s_2 *is* 1.

case (**forall** $x : \epsilon \mid \alpha$) : *In this case the result follows immediately from the induction hypothesis.*

case (**foreach** $x : \epsilon \mid \alpha$) : *Let* $C{=}[\![\epsilon]\!]^s_\rho$. *If* $|C| = 0$, *then* $Y = \emptyset$. *This implies that* $[\mathcal{S}_Y] = \emptyset$ *and, hence, the result is vacuosly true. Otherwise,* $Y = \bigcup_{c\in C} Y_c$ *and*

$$\sum_{s_2\in[\mathcal{S}_Y]} P(s_1, s_2) = \sum_{s_2\in[\mathcal{S}_Y]} \sum_{c\in C} \frac{1}{|C|} \cdot P^Y_c(s_{1|Y_c}, s_{2|Y_c})$$

$$= \sum_{s_2\in[\mathcal{S}_Y]} \sum_{c\in C} \frac{1}{|C|} \cdot P_c(s_{1|Y_c}, s_{2|Y_c}) \cdot (s_1 =_{Y\setminus Y_c} s_2)$$

$$= \sum_{c\in C} \frac{1}{|C|} \cdot \sum_{s_2\in[\mathcal{S}_Y]} P_c(s_{1|Y_c}, s_{2|Y_c}) \cdot (s_1 =_{Y\setminus Y_c} s_2)$$

$$= \sum_{c\in C} \frac{1}{|C|} \cdot \sum_{s^c_2\in[\mathcal{S}_{Y_c}]} P_c(s_{1|Y_c}, s^c_2)$$

where $(s_1{=}_{Y'}s_2)$ *denotes* 1 *if* $[\![y]\!]^{s_2}{=}[\![y]\!]^{s_1}$, *for eveinduction hypothesisry* $y \in Y'$, *and* 0 *otherwise. Using the induction hypothesis for each* $[\![\alpha]\!]^s_{\rho_c}$, *with* $c \in C$, *we reach the result.*

case (**foreach** $x : \epsilon$ **minus** $D \mid \alpha$) : *Similar to the previous case.*

case $\{\alpha_1 \& \ldots \& \alpha_n\}$: *If* $n = 1$, *then the result follows immediately from the induction hypothesis. Without loss of generality, we prove this for the case* $n = 2$. *If* Y_1, Y_2 *are not disjoint, as before, the result is vacuosly true. Otherwise, we have*

$$\sum_{s_2\in[\mathcal{S}_Y]} P(s_1, s_2) = \sum_{s_2\in[\mathcal{S}_Y]} P_1(s_{1|Y_1}, s_{2|Y_1}) \cdot P_2(s_{1|Y_2}, s_{2|Y_2})$$

$$= \sum_{s^1_2\in[\mathcal{S}_{Y_1}]} (P_1(s_{1|Y_1}, s^1_2) \cdot \sum_{s^2_2\in[\mathcal{S}_{Y_2}]} P_2(s_{1|Y_2}, s^2_2))$$

The last equality holds because, since Y_1, Y_2 *are disjoint,* $[\mathcal{S}_Y]$ *is isomorphic to* $[\mathcal{S}_{Y_1}] \times [\mathcal{S}_{Y_2}]$. *Using the induction hypothesis for each* $[\![\alpha_i]\!]^s_\rho$, *we reach the result.*

case $\{[p_1]\alpha_1 + \cdots + [p_n]\alpha_n\}$: *In this case* $Y = \bigcup_{i=1,..,n} Y_i$ *and*

$$\sum_{s_2 \in [\mathcal{S}_Y]} P(s_1, s_2) = \sum_{s_2 \in [\mathcal{S}_Y]} \sum_{i=1}^{n} p_i \cdot P_i^Y(s_{1|Y_i}, s_{2|Y_i})$$

$$= \sum_{s_2 \in [\mathcal{S}_Y]} \sum_{i=1}^{n} p_i \cdot P_i(s_{1|Y_i}, s_{2|Y_i}) \cdot (s_1 =_{Y \setminus Y_i} s_2)$$

$$= \sum_{i=1}^{n} p_i \cdot \sum_{s_2^i \in [\mathcal{S}_{Y_i}]} P_i(s_{1|Y_i}, s_2^i)$$

Using the induction hypothesis for each $[\![\alpha_i]\!]_\rho^s$ *and the fact that* $\sum_{i=1}^{n} p_i = 1$*, we reach the result.*

□

The quantization of component and connector properties may invalidate an impact model, by making pairs of constraints that were initially mutually exclusive, non mutually exclusive after quantization. Invalid models are inconsistent (i.e., they do not admit any interpretation) and, hence, we limit our attention to valid impact models.

Definition 7 (Semantics of Impact Models). *An impact model* $\mathcal{I}$ *is valid if for every* $s \in \mathcal{S}$*, there exists at most one element* $\langle \phi, \alpha \rangle \in \mathcal{I}$ *such that* $[s] \models \phi$*. The semantics of a valid impact model* $\mathcal{I}$*, which we denote by* $[\![\mathcal{I}]\!]$*, is the DTMC* $\langle [\mathcal{S}], P : [\mathcal{S}] \times [\mathcal{S}] \to [0,1] \rangle$ *where* P *is defined as follows:*

If the graph of s_1 *and* s_2 *is not the same then* $P(s_1, s_2) = 0$*,*

else if exists $\langle \phi, \alpha \rangle \in \mathcal{I}$ *s.t.* $s_1 \models \phi$ *then* $P(s_1, s_2) = P_\alpha^{Y_{s_1}}(s_{1|Y_\alpha}, s_{2|Y_\alpha})$

else if $s_1 \neq s_2$ *then* $P(s_1, s_2) = 0$ *else* $P(s_1, s_2) = 1$

where $[\![\alpha]\!]_\rho^{s_1} = \langle Y_\alpha, P_\alpha \rangle$*,* Y_s *denotes the set of all properties of components and connectors in a system state* s*, i.e.,* $Y_s = \{c.p{:}d \mid c$ *is a node in the graph of* $s \wedge \kappa{=}type(c) \wedge p{\in}\Pi(\kappa) \wedge d{=}dtype(\kappa.p)\}$.

As an example, consider an impact model defined by $\langle \mathsf{size(E)}{>}\mathsf{0}, \alpha \rangle$ with $\alpha =$ (**foreach** x:E | x.isActive' = true) and E=(s:ServerT|!s.isActive). This impact model expresses that if there is at least one inactive server, then exactly one server is activated and all inactive servers have the same probability of being activated.

Let s, s_1, s_2 be three system states with servers z_1, z_2, z_3 that differ only in the number of active servers: (i) in s only z_3 is active, (ii) in s_1 only z_2 is inactive and (iii) in s_2 only z_1 is inactive. According to definition above, and assuming that no other system states meet the same conditions, we have for instance that $P(s, s_1) = P(s, s_2) = \frac{1}{2}$ and $P(s, s') = 0$, for every other system state s' different from s_1 and s_2.

5. Predicting Adaptation Strategy Impact

In this section, we illustrate an application of the proposed impact models by showing how they can be used in the context of Rainbow/Stitch [9] to predict the impact of adaptation strategies on quality objectives.

In the context of Rainbow, the adaptation of the managed system is achieved through the execution of an adaptation strategy selected from a portfolio of strategies specified in the language Stitch. Typically, a situation that requires adaptation can be addressed through the execution of more than one alternative adaptation strategy in the portfolio. Since different strategies impact quality attributes in different ways, there is a need to choose a strategy that will result in the best outcome with respect to achieving the system's desired quality objectives. To enable decision-making for selecting strategies, Stitch uses utility functions and preferences, which are sensitive to the context of use and able to consider trade-offs among multiple potentially conflicting objectives.[5]

As illustrated in Fig. 2, strategy selection in Rainbow/Stitch is supported by two different processes:

- *Adaptation Model Definition*, which occurs at design-time and entails the specification of the different inputs required by the run-time strategy selection process. These inputs are the specification for: (i) adaptation logic,

[5]Although selection is driven by utility in Rainbow, alternative criteria can be supported in other applications of our impact model language (e.g., maximizing the probability of satisfying a given safety or liveness property).

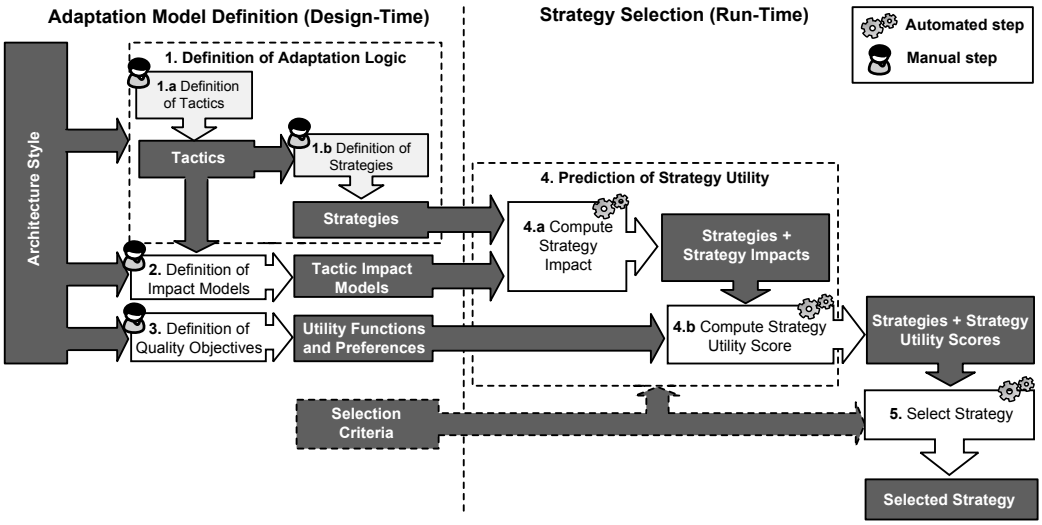

Figure 2: Strategy selection in Rainbow/Stitch.

which includes the set of adaptation tactics and strategies available, (ii) impact models for the tactics, and (iii) utility functions and preferences that embody the quality objectives of the system, and relate them to specific run-time conditions.

- *Strategy Selection*, which occurs at run-time and entails the prediction of the expected utility of every applicable strategy in three steps: (i) computing the aggregate impact of the strategy on system state, based on the impact models of its constituent tactics, (ii) computing the utility score of the strategy by mapping the predicted system state [6] to a utility value, making use of the utility functions and preferences, and (iii) selecting the best strategy according to a given set of selection criteria. In this section we consider the criterion followed by default by Rainbow which is to maximize the expected utility score. An alternative criterion based on risk-avoidance is considered in the next Section.

In the remainder of this section, we first focus on the definition of the adaptation model at design-time by introducing adaptation strategies as prescribed by Stitch

[6]The predicted state of the system after the execution of a given strategy is obtained by merging the current state of the system with the aggregate impact of the strategy computed in the previous step.

(Section 5.1), following with the specification of quality objectives as utility functions and preferences (Section 5.2). Finally, we present the process for predicting the utility of strategies at run-time (Section 5.3).

5.1. Adaptation Strategies

Strategies are built from tactics, which are Stitch's adaptation actions. Strategies have an applicability condition and a body. The body of a strategy σ is a tree T_σ whose edges $n \rightarrow m$ are labelled by a guard condition, a tactic and a success condition (with all edges leaving a node labelled by distinct tactics). Once at node n, if the guard condition is true, it means that the edge can be taken. When an edge is taken, the corresponding tactic is executed. Upon its termination the success condition is evaluated to determine if the tactic achieved what was expected and node m is reached. Guards include a special symbol $success$ capturing whether the last tactic had succeeded or not.

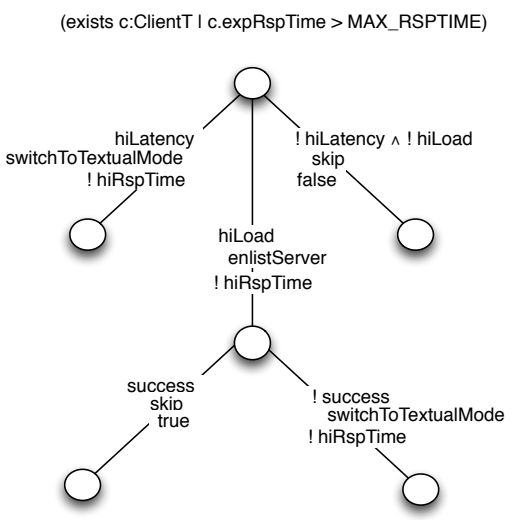

Figure 3: An adaptation strategy for Znn.

An example of a strategy for Znn is simpleReduceResponseTime presented in Fig. 3. hiLoad, hiLatency and hiRspTime are formulas expressing respectively that the system load, latency and response time is high. hiRspTime, for instance, is defined in terms of the average response time of the clients by:

$$(\text{sum}(\text{c.expRspTime}|\text{c: ClientT})/\text{count}(\text{c: ClientT}) > \text{MAX\_RSPTIME})$$

The body of the strategy defines that initially there are three alternatives, depending on the load and latency of the system. If the latency is high, then a possibility defined by the strategy is to just apply the tactic switchToTextualMode. The success condition in this case is hiRspTime that expresses that the average response time is below a given threshold. If the load is high, then a possibility is to apply the tactic enlistServer and, depending on the success of the application of this tactic, either terminate with skip or still try the application of the tactic

switchToTextualMode. If neither the latency nor the load is high, the strategy does not offer any remedy.

5.2. Defining Quality Objectives

Defining quality objectives requires the identification of the concerns for the different stakeholders. For instance, in Znn users are concerned with experiencing service without any disruptions, which can be mapped to specific run-time conditions such as response time. In contrast, the organization is interested in minimizing the cost of operating the infrastructure, which can be mapped to the cost of specific resources used at run-time (e.g., active servers). In short, we identify two quality objectives: maintaining low client response time (R), and cost (C).

Given a set of concerns, it may be impossible to achieve all of them optimally due to resource constraints or fundamental conflicts between certain quality objectives (e.g., in Znn, better performance achieved via activating additional servers results in increasing cost of operating the system). Stitch employs a systematic technique to assign different levels of importance to the various concerns based on utility theory [21].

A von Neumann-Morgenstern utility function $u_c : X_c \rightarrow \mathbb{R}$ assigns a real number to each value of a quality concern c, which can be normalized to the range $[0, 1]$. Across multiple dimensions, we can attribute a percentage weight w_c to each concern to account for its relative importance, compared to other concerns. These weights constitute the *utility preferences*. Overall utility is given by the function $U = \sum_c w_c u_c$.

Table 1 summarizes the utility functions for Znn defined by an explicit set of value pairs (where intermediate points are linearly interpolated). Function $\mathsf{U_R}$ maps low response times (up to 100ms) with maximum utility, whereas values above 2000ms are highly penalized (utility below 0.25), and response times above 4000ms provide no utility. Function $\mathsf{U_C}$ maps a increasing cost (derived from the number of active servers) to lower utility values.

Utility preferences that capture business preferences over the quality dimensions assign a specific weight $w_{\mathsf{U_R}}$ and $w_{\mathsf{U_C}}$ to each one of them in Znn, where we consider that preference is given to performance over cost.

To compute the utility of a given system state s (denoted as $Util(s)$), we first

$\mathsf{U_R}(w_{\mathsf{U_R}} = 0.6)$				$\mathsf{U_C}(w_{\mathsf{U_C}} = 0.4)$		
0 : 1.00	200 : 0.99	1000 : 0.70	2000 : 0.25	0 : 1.00	2 : 0.90	4 : 0.00
100 : 1.00	500 : 0.90	1500 : 0.50	4000 : 0.00	1 : 1.00	3 : 0.30	

Table 1: Utility functions and preferences for Znn

need to map the values of the different qualities[7] to their corresponding utility values. In a system state with 1250 ms of response time and a cost of 2 USD/hour, based on the utility functions defined in Table 1, we have [$\mathsf{U_R}$(1250), $\mathsf{U_C}$(2)] = [0.625, 0.9]. Finally, all utilities are combined into a single value, using utility preferences: 0.625 * 0.6 + 0.9 * 0.4 = 0.735.

5.3. Predicting the Utility of Strategies

The expected utility of a strategy σ in a given system state s can be formulated in terms of a tree like that presented in Fig. 4: i.e., a labelled tree with two types of nodes — normal and chance nodes, that alternate in consecutive depth levels of the tree. As with decision trees, chance nodes represent situations in which the choice between the different alternatives is external (i.e., not under the system's control), and is governed according to a given probability distribution function. Normal nodes, as decision nodes of decision trees, represent situations in which the choice between the different alternatives is internal. These nodes reflect situations of non-determinism during the execution of the strategy (that arise when more than one edge can be executed) that we assume are solved by a fair scheduler and, hence, all alternatives have the same probability of being taken. In this way, all edges $\langle n, m\rangle$ of the tree are labelled with a probability p; if n is a normal node then the edge is additionally labelled by a tactic t. For short we write, respectively, $n \overset{p}{\text{——}} m$ and $n \overset{p}{\underset{t}{\text{——}}} m$. Chance nodes are not labelled whereas every normal node n is labelled by a system state.

Formally, this type of labelled tree can be represented as a tuple $\langle N, st, E, l\rangle$ with $N{=}H{\cup}A$, where H and A are the sets of, respectively, chance and normal nodes, st is a function that labels nodes in H with system states, E is the set of edges and l labels edges with a probability and, optionally, a tactic. The tree defined by a strategy σ in a given system state s, which we denote by $T_{\mathcal{I}}(\sigma, s)$, is defined as follows.

[7]For utility calculation, we assume a representation of system state in terms of qualities. In Znn, we take the average of response time in all clients and the sum of the costs of active servers.

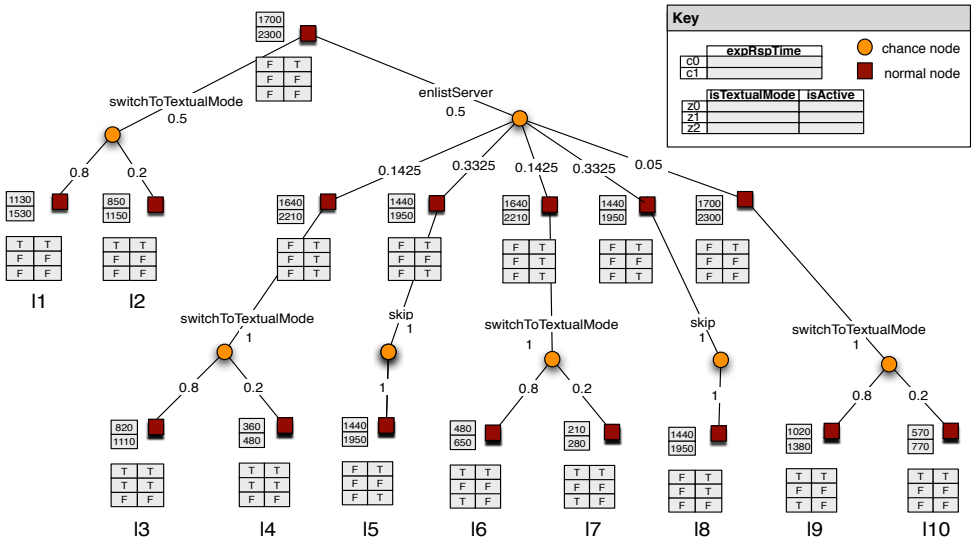

Figure 4: Tree for simpleReduceResponseTime and a system state with 2000 ms of response time, and a cost of 1 USD/hour.

Definition 8. *[$T_{\mathcal{I}}(\sigma, s)$] Let σ be a strategy and s a system state. Given an impact model $\mathcal{I}_t$ for every tactic t used in σ, $T_{\mathcal{I}}(\sigma, s)$ is the labelled tree obtained as follows:*

1. *Start with an empty set of chance nodes and edges and with a single normal node $root(T_\sigma)$ labelled by $[s]$.*
2. *While there exists a normal node n that has not been considered before:*
   (a) *Find the edges of T_σ that start in n and can be executed in state $st(n)$:*

   $$E_n = \{n \xrightarrow{\langle \phi, t, \psi \rangle} m \text{ in } T_\sigma : st^*(n) \vDash \phi\}$$

   where, supposing that k is the parent node of n and ψ' is the success condition of the edge that leads to n in T_σ, $st^*(n)$ is the extension of $st(n)$ with the interpretation of $success$ with $true$ if ψ' holds in $st(k)$ and $false$, otherwise.
   (b) *For every $n \xrightarrow{\langle \phi, t, \psi \rangle} m \in E_n$:*
      i. *add m to the set of chance nodes and $n \xrightarrow{1/|E|}_{t} m$ to the set of edges.*
      ii. *for every state s' such that $p = P_{[\![\mathcal{I}_t]\!]}(st(n), s') > 0$, add the node $m^{s'}$ labelled by s' to the set of normal nodes and $m \xrightarrow{p} m^{s'}$ to the set of edges.*

Fig. 4 presents the tree $T_{\mathcal{I}}(\mathsf{simpleReduceResponseTime}, s)$ where s is system state with two clients (c_0, c_1) and three servers (z_0, z_1, z_2). Only server z_0 is active in s and is not working in textual mode. The cost assigned to all servers is 1 and the load assigned to z_0, z_1, z_2 is, respectively, 2, 0 and 0. The response time for c_0 is 1700 and for c_1 is 2300. The considered impact models for tactics switchToTextualMode and enlistServer were those presented, respectively, in Listings 1 and 3.[8] Then, using the utility profile, we can calculate the utility of the state associated with each leaf node. The expected utility of the strategy is given by the sum of these utilities weighted by the probability of the path that leads to that node.

n	p_n	expRspTime avg. (ms)	cost (USD/hour)	$\mathsf{U_R}(st(n))$	$\mathsf{U_C}(st(n))$	$Util(st(n))$	$p_n \cdot Util(st(n))$
l1	0.4	1330	1	0.585	1	0.751	0.3004
l2	0.1	1000	1	0.75	1	0.85	0.085
l3	0.057	965	2	0.7605	0.9	0.8163	0.046529
l4	0.01425	410	2	0.927	0.9	0.9162	0.013056
l5	0.16625	1695	2	0.4025	0.9	0.6105	0.1014496
l6	0.057	565	1	0.8805	1	0.9283	0.052913
l7	0.01425	245	2	0.9765	1	0.9859	0.014049
l8	0.16625	1695	2	0.4025	0.9	0.6105	0.101496
l9	0.02	1200	1	0.65	1	0.79	0.0158
l10	0.005	670	1	0.849	1	0.9094	0.004245
total	**1**	-	-	-	-	-	**0.734937**

Table 2: Sample calculation of aggregate utility for strategy simpleReduceResponseTime

Definition 9 (Expected Utility of a Strategy). *Given a set of impact models $\mathcal{I}$, the expected utility of a strategy σ in a system state s is given by*

$$\sum_{n \in leaves(T_{\mathcal{I}}(\sigma,s))} p_n \cdot Util(st(n))$$

where p_n is the product of the probabilities in the path leading from the root to n.

Table 5.3 illustrates the utility calculation for strategy simpleReduceResponseTime that corresponds to the tree shown in Fig. 4. Note that nodes l1 and l2 contribute half of the utility, and that the sum of all p_n assigned to leaf nodes adds to one.

At run-time, when a situation that requires adaptation is detected, the decision-making process entails the computation of the expected utility of every applicable strategy and the selection of the strategy with the highest expected utility score.

[8] For readability reasons we represented in the figure only the part of the system state that is directly manipulated or affected by the two tactics.

6. Decoupling Adaptation Impact from Environment Assumptions

In this section we present a technique for predicting strategy impact based on the separation between the adaptation impact and the assumptions over the environment. We also show how this technique can be explored in the context of strategy selection, describing a risk-avoiding strategy selector as an alternative instantiation of the strategy utility prediction and selection processes (steps 4 and 5, Fig. 2).

The technique for predicting strategy impact presented in the previous section assumes that impact models of adaptation actions capture all relevant changes in the system properties that occur during the execution of the adaptation action. This implies that the impact model of an adaptation action has to capture, on the one hand, the expected changes in the system properties that are a direct consequence of its execution and, on the other hand, the assumptions about the changes made independently by the environment. Consider, for instance, the impact model for enlistServer presented in Listing 4. It expresses that the probability of exactly one active server crashing while executing the adaptation is 0.001, which clearly corresponds to an environment assumption. With respect to the changes in the expected response time, it is not possible to tell whether these correspond to changes exclusively caused by the execution of the action, or if they also reflect some assumptions, for instance, on the evolution in the number of requests that the system receives from clients.

Since assumptions about the expected evolution in the number of requests or the probability of an active server crashing while executing an adaptation action do not typically depend on the adaptations being executed, it is convenient to have the means to express them in an independent way. The explicit modelling of assumptions about how the execution context evolves is particularly relevant when adaptations are not taken in isolation, but as an integral part of an adaptation plan (as it happens in strategies), since it allows us to obtain better predictors of strategy impact and opens the possibility to compute alternative notions of impact (e.g., worst-case scenario, compared to the average case described in Section 5).

In the remainder of this section, we first present the models that we employ to describe environment assumptions (Section 6.1), followed by a description of how stochastic multiplayer games (SMGs) can be used as a suitable formal encoding to reason about the impact of adaptations in the presence of an adversarial environment (Section 6.2). Finally, we describe how these formalisms can be employed to implement a risk-averse adaptation strategy selector in Section 6.3.

6.1. Modelling Environment Assumptions

Assumptions about the way the execution context evolves can be expressed in terms of a subset of the probabilistic expressions introduced in Section 4. Since managed properties are uniquely under the system control, we exclude the use of expressions in $\mathcal{P}(\Sigma)$ that impose constrains on the value of these properties in the next state. This can be achieved by considering that, in the first rule of Def. 3, p ranges only over monitored properties, i.e., $p \in \Pi^m(\kappa)$. We denote by $\mathcal{P}^e(\Sigma)$ the resulting subset of probabilistic expressions.

Definition 10 (Environment Model). *An environment model $\mathcal{E}$ is a finite set of pairs $\langle \phi, \alpha \rangle$ where ϕ is a constraint in $\mathcal{C}(\Sigma)$ and α is a probabilistic expression in $\mathcal{P}^e(\Sigma)$ such that all ϕ are mutually exclusive.*

Since environment models are just particular cases of impact models, Def. 7 can be used to define their semantics.

An example of a simple environment model for Znn, including both external and internal aspects of the system that are not under the direct control of the adaptive mechanism, is presented below. This model expresses that the probability of a server to become spontaneously inactive during an execution of an adaptation action is 0.001. Moreover, it also expresses some assumptions about the expected evolution in the number of requests arriving to the system. In order to keep the model simple, this is expressed directly in terms of expected response time: the probability of a small increment (inversely proportional to the number of active servers – m) in the response time of all clients is 0.25 (line 10), whereas the probability of a small decrement (directly proportional to the number of servers) is also 0.25 (line 12). The probability that response time will remain unchanged is 0.50 (line 11).

```
1  define T=(s:ServerT | s.isActive)
2  define S=(s:ServerT | !s.isActive)
3  define m = size(T)
4
5  environmentmodel
6    { { [0.999] { foreach s:ServerT | s.isActive'=s.isActive }
7      + [0.001] { foreach s:S | s.isActive'=true & foreach t:T | t.isActive'=false } }
8      &
9      { [0.25] { forall c:ClientT | c.expRspTime'=(1+0.05/m)*c.expRspTime }
10     + [0.50] { forall c:ClientT | c.expRspTime'=c.expRspTime }
11     + [0.25] { forall c:ClientT | c.expRspTime'=(1-0.05*m)*c.expRspTime } }
12   }
```

Listing 5: An environment model for Znn.

If we opt to express the impact of adaptations and the assumptions over the evolution of the environment in an independent way, both specifications can be easily merged into one impact model per tactic. We take the superposition of the environment assumptions over the impact of each tactic t to be defined by

$$[\![\mathcal{I}_t]\!] \times [\![\mathcal{E}]\!] = \langle [\mathcal{S}], P_{[\![\mathcal{I}_t]\!]} \times P_{[\![\mathcal{E}]\!]} \rangle$$

which is the DTMC obtained by taking the multiplication of the two probability matrices. This corresponds to applying the changes underlying the environment model after the changes resulting from the execution of the tactic. In this way, it is still possible to apply the method for calculating the aggregated utility of a strategy presented in the previous section and continue using expected utility as a selection criterion.

6.2. Reasoning about Strategy Impact using Stochastic Game Analysis

The separation between the adaptation impact and the assumptions over the environment opens new possibilities to reason about strategy impact. In particular, we can regard the behaviour of the system and its environment during the execution of a strategy as a two-player game and then perform stochastic game analysis for reasoning about the strategy impact.

We build our approach upon the framework for modelling and automatic verification of systems with both probabilistic and competitive behaviour presented in [7]. For this purpose, we define the semantics of a strategy as a turn-based stochastic two-player game, based on an adversary semantics of environment models that regards any state transition with non zero probability as a possible move of the environment from the source state.

Definition 11 (Adversary Semantics of Environment Models). *The adversary semantics of an environment model $\mathcal{E}$, which we denote by $(\!|\mathcal{E}|\!)$, is the set of state transitions* $\{(s_1, s_2) \in [\mathcal{S}] \times [\mathcal{S}] : P_{[\![\mathcal{E}]\!]}(s_1, s_2) > 0\}$.

This adversary semantics of environment models abstracts away the probabilities associated with state transitions, retaining only the information about the possible transitions. Each element of $(\!|\mathcal{E}|\!)$ embodies a choice of action to be taken by the environment in a specific state.

Based on this adversary semantics of the environment, we can build a two-player game that represents the behaviour of the system and its environment during the execution of the adaptation strategy. The set of states of this game is partitioned in two disjoint sets. In states of the form $sys(s)$ it is the turn of the

system, which can choose between the tactics prescribed in the adaptation strategy, the outcome of which can be probabilistic. In states of the form $env(s)$ it is the turn of the environment, which can choose between the transitions from s that, according to the environment model, were described to have a non zero probability.

Definition 12 ($SMG_{\mathcal{I},\mathcal{E}}(\sigma, s_0)$). *Let σ be a strategy and s_0 a system state. Given an impact model $\mathcal{I}_t$ for every tactic t used in σ and an environment model $\mathcal{E}$, $SMG_{\mathcal{I},\mathcal{E}}(\sigma, s)$ is the turn-based two-player game*

$$\mathcal{G} = \langle \{Sys, Env\}, Q = Q_{Sys} \cup Q_{Env}, q_0, A, \Delta, AP, \chi, r \rangle$$

where:

- $Q_{Sys} = \{sys(s, n) : s \in [\mathcal{S}]$ *and* n *is a node of* $T_\sigma\}$
- $Q_{Env} = \{env(s, n) : s \in [\mathcal{S}]$ *and* n *is a node of* $T_\sigma\}$
- $q_0 = sys(s_0, root(T_\sigma))$ *is the initial state*
- A *is the union of the set of tactics used in* σ *(denoted by* $(\!|\sigma|\!)$ *in the following) with the set* $(\!|\mathcal{E}|\!)$*. In each state* $q \in Q$*, the set of available actions is denoted by* $A(q)$*.*
- $\Delta : Q \times A \to \mathcal{D}(Q)$ *is defined as follows* [9]*:*
  - *For every tactic* $t \in (\!|\sigma|\!)$*, if* $q \in Q_{Env}$*, then* $\Delta(q, t)$ *is undefined; otherwise, let* n, s_1 *be such that* $q = sys(s_1, n)$
    - *if there exists an edge* $n \xrightarrow{\langle \phi, t, \psi \rangle} m$ *in* T_σ *and* $s_1 \vDash \phi$ *then* $\Delta(q, t)$ *is defined as follows:*

$$\Delta(sys(s_1, n), t)(q') = \begin{cases} P_{[\![\mathcal{I}_t]\!]}(s_1, s_2) & \text{if } q' = env(s_2, m) \text{ for some } s_2 \in [\mathcal{S}] \\ 0 & \text{otherwise} \end{cases}$$

    - *else* $\Delta(q, t)$ *is undefined*
  - *For every* $e \in (\!|\mathcal{E}|\!)$*, if* $q \in Q_{Sys}$*, then* $\Delta(q, e)$ *is undefined; otherwise, let* n, s_1 *be such that* $q = env(s_1, n)$

[9] $\mathcal{D}(X)$ denotes the set of discrete probability distributions over finite set X.

  - *if* e *is* (s_1, s_2) *for some* $s_2 \in [\mathcal{S}]$ *then* $\Delta(q, e)$ *is defined as follows:*

$$\Delta(env(s_1, n), e)(q') = \begin{cases} 1 & \text{if } q' = sys(s_2, n) \\ 0 & \text{otherwise} \end{cases}$$

  - *else* $\Delta(q, e)$ *is undefined*

- AP *is a set of predicates that can be built over the state variables (i.e., the monitored and managed properties of components and connectors of the system) and a predicate* $leaf$*.*

- $\chi : Q \to 2^{AP}$ *is the labeling function defined as follows: for every* $s \in [\mathcal{S}]$ *and node* n *of* T_σ*, (1)* $leaf \in \chi(sys(s, n))$ *iff* $n \in leaves(T_\sigma)$*,* $leaf \notin \chi(env(s, n))$*; (2) for every predicate* p *in* AP *over state variables,* $p \in \chi(sys(s, n))$ *iff* $p \in \chi(env(s, n))$ *iff* $s \vDash p$*.*

- $r : Q \to \mathbb{Q}_{\geq 0}$ *is a reward structure mapping each state to its utility value, i.e.,* $r(s) = Util(s)$*.*

Notice that this definition abstracts away the success conditions of tactics, which would require a much more detailed definition and are not relevant for the purpose at hand (in our game-theoretical setting they represent a constraint on the solution space). Also note that in the initial state of the game, as well as in any of the possible final states (corresponding to the leaf nodes of the tree for strategy σ), are defined to be of the form $sys(s, n)$.

Definition 13 (Path). *A path of SMG* $\mathcal{G}$ *is an (in)finite sequence* $\lambda = q_0 a_0 q_1 a_1 \ldots$ s.t. $\forall j \in \mathbb{N}\ a_j \in A(q_j) \ \wedge \ \Delta(q_j, a_j)(q_{j+1}) > 0$*.* $\Omega^+_{\mathcal{G}}$ *denotes the set of finite paths in* $\mathcal{G}$*.*

The system and environment players can follow policies for choosing actions in the game, competing to achieve their own (potentially conflicting) goals.

Definition 14 (Environment Policy). *A policy for the* Env *player in* $\mathcal{G}$ *is a function* $\rho_{Env} : (QA)^* Q_{Env} \to \mathcal{D}(\langle\!|\mathcal{E}|\!\rangle)$ *which, for each path* $\lambda \cdot q \in \Omega^+_{\mathcal{G}}$ *where* $q \in Q_{Env}$*, selects a probability distribution* $\rho_{Env}(\lambda \cdot q)$ *over* $A(q) \subseteq \langle\!|\mathcal{E}|\!\rangle$*. The set of all policies for player* Env *is denoted* P_{Env}*.*

Definition 15 (System Policy). *A policy for the* Sys *player in* $\mathcal{G}$ *is a function* $\rho_{Sys} : (QA)^* Q_{Sys} \to \mathcal{D}(\langle\!|\sigma|\!\rangle)$ *which, for each path* $\lambda \cdot q \in \Omega^+_{\mathcal{G}}$ *where* $q \in Q_{Sys}$*, selects a probability distribution* $\rho_{Sys}(\lambda \cdot q)$ *over* $A(q) \subseteq \langle\!|\sigma|\!\rangle$*. The set of all policies for player* Sys *is denoted* P_{Sys}*.*

In this paper, we always refer to policies $\rho_{i\in\{Sys,Env\}}$ that are *memoryless* (i.e., $\rho_i(\lambda \cdot q) = \rho_i(\lambda' \cdot q)$ for all paths $\lambda \cdot q$, $\lambda' \cdot q \in \Omega^+_{\mathcal{G}}$), and *deterministic* (i.e., $\rho_i(\lambda \cdot q)$ is a Dirac distribution for all $\lambda \cdot q \in \Omega^+_{\mathcal{G}}$). Memoryless, deterministic policies resolve the choices in each state $q \in Q_i$ for player i, selecting actions based solely on information about the current state in the game. These policies are guaranteed to achieve optimal expected rewards for the kind of cumulative reward structures that we use to encode utility in our models.[10]

Turn-based SMGs such as the one described in this section can be encoded in PRISM-games and analyzed via probabilistic model checking [8]. Reasoning about policies is a fundamental aspect of model checking SMGs, which enables checking for the existence of a policy that is able to optimize an objective expressed as a quantitative property in a logic called rPATL [8], which extends ATL [1], a logic extensively used to reason about the ability of a set of players to collectively achieve a particular goal. Properties written in rPATL can state that a coalition of players has a policy which can ensure that the probability of an event's occurrence or an expected reward measure meet some threshold.

In the next section, we show how rPATL and probabilistic model checking of SMGs can be employed to implement a risk-averse strategy selector.

6.3. Risk-averse Strategy Selection

To implement a risk-averse strategy selector, we make use of rPATL specifications on game-theoretical models that comply with the description given in the previous section. rPATL is a CTL-style branching-time temporal logic that incorporates the coalition operator $\langle\langle C \rangle\rangle$ of ATL, combining it with the probabilistic operator $\mathsf{P}_{\bowtie \mathsf{q}}$ and path formulae from PCTL [2]. Moreover, rPATL includes a generalization of the reward operator $\mathsf{R}^{\mathsf{r}}_{\bowtie \mathsf{x}}$ [17] to reason about goals related to rewards. Extensions of the reward operator in rPATL include $\langle\langle C \rangle\rangle \mathsf{R}^{\mathsf{r}}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \phi]$ and $\langle\langle C \rangle\rangle \mathsf{R}^{\mathsf{r}}_{\mathsf{min}=?}[\mathsf{F}^{\mathsf{c}}\ \phi]$, and enable the quantification of the maximum and minimum accrued reward r along paths that lead to states satisfying ϕ that can be guaranteed by players in coalition C, independently of the policies followed by the rest of players.

In the context of risk-averse strategy selection, we make use of rPATL specifications to analyze the maximum utility that the system can guarantee when adapting, independently of the behavior of the environment (worst-case scenario analysis). To carry out this analysis, a typical rPATL property combining the coalition

[10] See Appendix A.2 in [7] for details.

and reward maximization operators is $\langle\langle\{Sys\}\rangle\rangle \mathsf{R}^{r}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \phi]$, meaning "value of the maximum utility reward r accrued along paths leading to states satisfying state formula ϕ that the system player can guarantee, regardless of the policy followed by the environment player."

The use of such specifications leads to a notion of *guaranteed expected utility* that differs from the fully probabilistic notion of expected utility of a strategy described in Def. 9, which is unsuitable to provide an estimation of the potential bad situations that the system might incur while executing a strategy.

Definition 16 (Guaranteed Expected Utility of a Strategy). *Given an environment model $\mathcal{E}$ and a set of impact models $\mathcal{I}$, the guaranteed expected utility of strategy σ in a system state s is given by the maximum utility that Sys can guarantee while executing σ by following an optimal policy, independently of the policy followed by Env. This value, which we denote by $U^{g}_{\mathcal{I},\mathcal{E}}(\sigma, s)$, is given by the evaluation of the rPATL expression*

$$\langle\langle\{Sys\}\rangle\rangle \mathsf{R}^{ur}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ leaf]$$

over the game $SMG_{\mathcal{I},\mathcal{E}}(\sigma, s)$, where $ur : Q \to \mathbb{Q}_{\geq 0}$ is a reward structure defined as follows: $ur(q) = Util(q)$ iff $leaf \in \chi(q)$, and zero otherwise.

This alternative notion of strategy utility can be used by the strategy selector. At run time, when an anomaly is detected, this selector has to calculate the guaranteed expected utility of all the strategies that are applicable in the current state and select the strategy with the highest score. By focusing on what happens in the worst-case scenario, such a selector manifests risk aversion.

Definition 17 (Risk-averse Strategy Selector). *Given an environment model $\mathcal{E}$ and a repertoire of adaptation strategies $\mathcal{S}$, a risk-averse strategy selector is an agent that selects, in a given state s, one of the strategies $\sigma_{\uparrow}$ that maximizes the guaranteed expected utility in the repertoire:*

$$\sigma_{\uparrow} \triangleq \arg\max_{\sigma \in \mathcal{S}} U^{g}_{\mathcal{I},\mathcal{E}}(\sigma, s)$$

In the Appendix we show how to put these ideas into practice by making use of PRISM-games [8], a probabilistic model checker that supports the analysis of SMGs.

7. Experimental Results

In this section, we report on our experience quantifying the benefits of employing our impact models in Rainbow.

To carry out our study, we considered two alternative models of Znn, one using impact vectors[11] and the other using the proposed impact models. We manually encoded the two models in the language of PRISM [23] and assessed the quality of the models that we are able to specify in each case by quantifying: (i) impact of tactics on the state of the target system, and (ii) impact of strategies on system utility. In addition, both models incorporate an M/M/c queuing model [13], which is able to compute the response time of the system based on the rate of request arrivals to the system, number of active servers, and the service rate (i.e., the time that it takes to service a request, which in this case is directly proportional to the fidelity level). In our experiments, we assume that the response time computed with the queuing model is considered as the actual response time of the system, against which we compare the predictions made using either vectors or probabilistic models.

Using each of the alternative models, we explored a state space $[S] = [1, 9] \times [1, 3]$, which includes the request arrival rate in the interval $[1, 9]$ requests/s, and the number of active servers in the interval $[1, 3]$ (i.e., a valid system configuration can have up to a maximum of 4 active servers). For the sake of clarity, we fixed in our experiments the values of other variables which could have been considered as additional dimensions in our state space (network latency is 0 ms, whereas service rate is fixed at 1 ms).

In the following, we first report on the improvement of accuracy in estimating system state for a single tactic when employing probabilistic impact models (Section 7.1). Next, we describe our results concerning the improvement in the prediction of the impact of a full strategy on utility, following the strategy selection scheme for the maximization of expected utility described in Section 5 (Section 7.2). Finally, we describe our results in the context of risk-averse strategy selection, as described in Section 6 (Section 7.3).

[11] The impact of individual adaptation actions is specified in terms of constant impact vectors (called cost/benefit attribute vectors), which describe how the execution of adaptation actions affects system quality attributes [9].

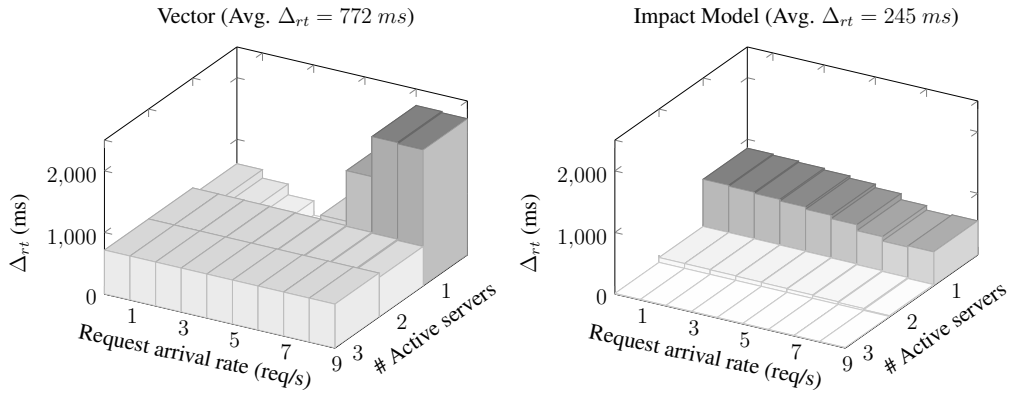

Figure 5: Deviation in response time impact prediction for tactic enlistServer: cost/benefit vector (left) and probabilistic impact model (right).

7.1. Impact of Tactics on System State

To quantify the improvement obtained using vectors with the results of employing probabilistic models, we focused on the enlistServer tactic, for which we encoded two alternative impact descriptions:

Cost-benefit Attribute Vectors. For the vector-based version of the model, we computed the average impact in response time of adding a server in all points of the explored region of the state space, making use of the M/M/c queuing model, which is the best approximation that can be obtained, given by

$$(\sum_{s \in [S]} \textit{MMc}(ar_s, as_s + 1) - \textit{MMc}(ar_s, as_s))/|[S]|$$

where $\textit{MMc}(a, b)$ returns the response time for request arrival rate a and number of active servers b. Moreover, ar_s and as_s designate the request arrival rate, and number of active servers in state s, respectively.

For our state space, this calculation yielded a reduction of response time of 714 ms. Since the cost is increased in 1 unit, and fidelity is not changed by enlistServer, the vector used in our experiments for the tactic is $[-714, +1, 0]$.

Probabilistic Impact Models. The probabilistic version of the model employed for the experiments is analogous to the one described in Listing 3.

Fig. 5 shows the deviation from actual response time impact values (computed using the M/M/c model) for tactic enlistServer. The values computed using the probabilistic impact model (right) are much more accurate, since their deviation

is far less prominent than the one presented when computing impact with vectors (average deviation Δ_{rt} in values computed using vectors is $\simeq 315\%$ with respect to the values obtained using probabilistic impact models). Moreover, while the values computed using vectors are not sensitive to context, presenting reduced deviations with respect to actual response times only in states that are close to the average (e.g., 1 server, 3-5 requests/s), values obtained with the probabilistic model better approximate actual impact, reflecting the fact that a higher number of active servers noticeably reduces the impact of the tactic on response time.

7.2. Impact of Strategies on Expected System Utility

The use of different models to express the impact of tactics on system state also affects the predictions that concern the expected utility of the system after the execution of adaptation strategies. To assess how utility prediction is affected by the constructs available to express tactic impact, we included in our PRISM model an encoding of the strategy simpleReduceResponseTime shown in Fig. 3, and computed the expected utility after its execution for each of the states included in $[S]$ for each of the alternatives.

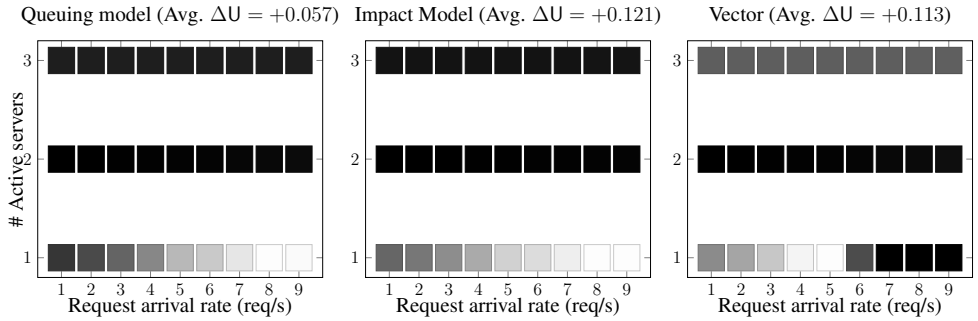

Figure 6: Utility prediction for the execution of strategy reduceResponseTime based on: queuing model, impact model, and impact vector. Lighter colors represent higher utility improvements.

Fig. 6 shows how the utility values predicted using probabilistic impact models (center) exhibit a similar pattern to the one obtained using the queuing model (left). In contrast, vectors (right) show an entirely different pattern, which only coincides with the one resulting from the queuing model in the area in which the impact of tactics is close to their average impact (i.e., when there are two active servers).

It is worth noting that the overall average difference in utility ΔU across the state space does not constitute a representative indicator of the accuracy of utility

predictions, since positive and negative utility deltas in different states can cancel each other (i.e., the average ΔU yielded by vectors is $+0.113$, which is closer to the one obtained with the queuing model of $+0.057$, even if the absolute value of the deviation in individual states in vectors is greater than in the case of impact models).

7.3. Impact of Strategies on Guaranteed Expected System Utility

In general, the actual utility obtained after executing an adaptation strategy does not coincide with the expected utility prediction (e.g., based on impact models or vectors) that the system employs for decision-making. Hence, in our experiments we need to determine the actual guaranteed expected utility of the system (i.e., based on the M/M/c queuing model) achieved when the system employs for decision-making a predicted guaranteed expected utility based either on impact models or vectors.

To assess how the actual performance of risk-averse adaptation is affected by the constructs available to express tactic impact, we created a PRISM-games model encoding a SMG for a modified version of the strategy simpleReduceResponseTime (Fig. 7).[12] This strategy includes two main branches in which tactics enlistServer and switchToTextualMode can be selected for execution in either order. Note that in general, the order in which tactics are selected results in different system configurations and outcomes of strategy execution (e.g., a server activated via tactic enlistServer after the execution of switchToTextualMode will not operate in textual mode, whereas switching to textual mode after an additional server has been activated will result in a configuration in which all servers are in textual mode).

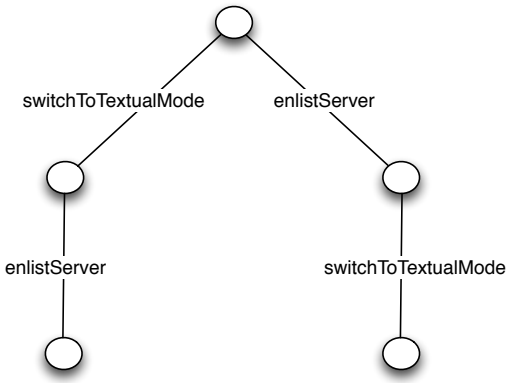

Figure 7: A simplified version of an adaptation strategy for Znn.

The model explores a state space $[S] = [1, 3]$, which includes only the number of active servers in the interval $[1, 3]$ (i.e., a valid system configuration can have

[12] A description of the game's encoding is provided in Appendix A.

up to a maximum of 4 active servers). Some of the values of other variables that could have been considered as additional dimensions in our state space have been fixed for clarity (e.g., service rate is fixed at 1 ms). In particular, it is worth noting that the arrival rate of requests does not add an extra dimension to the state-space in this case (in contrast with the results presented in Sections 7.1 and 7.2), since the value taken by this variable is controlled by the environment player, even in the initial state of the game encoding the strategy's execution.

In particular, we instantiated two variants of our SMG model in which the system player bases its decision-making on the utility prediction yielded either by fixed vectors or probabilistic impact models. Each one of the variants includes:

- The adversary semantics of an environment model that controls: (i) the rate of requests arriving at the system, and (ii) network latency.
- The behavior of the system player, including two sets of variables representing: (i) the actual state of the system (i.e., based on the queuing model – used to compute actual guaranteed expected utility, encoded in a reward structure aur), and (ii) the predicted state of the system (based on a set of probabilistic impact models ($\mathcal{I}_{\mathcal{P}}$) or vectors ($\mathcal{I}_{\mathcal{V}}$) in the respective variants of the SMG model – used to compute the predicted guaranteed expected utility employed for decision-making, encoded in a reward structure pur).

After instantiating the PRISM-games model for a stochastic game $SMG_{\mathcal{I},\mathcal{E}}(\sigma, s)$, we proceed with our analysis in two stages:

1. Compute the optimal policy ρ_{pur} [13] that the system player follows based on the information about predicted utility for decision-making. That policy is computed based on an rPATL specification that obtains the expected guaranteed utility as predicted by the system player during strategy execution in the original game $SMG_{\mathcal{I},\mathcal{E}}(\sigma, s)$:
$$U^{pg}_{\mathcal{I},\mathcal{E}}(\sigma, s) \triangleq \langle\langle\{Sys\}\rangle\rangle \mathsf{R}^{pur}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ leaf]$$
2. Quantify the actual expected guaranteed utility under the generated system policy. We do this by using PRISM-games to build a product of the existing game model and the policy synthesized in the previous step, obtaining a new game $SMG^{\rho_{pur}}_{\mathcal{I},\mathcal{E}}(\sigma, s)$ under which further properties can be verified. In our

[13] Note that the synthesis algorithm returns a single policy if there exists more than one optimal policy yielding the same reward value for the game [8].

	Impact Vectors $\mathcal{I}_{\mathcal{V}}$		Probabilistic Impact Models $\mathcal{I}_{\mathcal{P}}$	
Number of servers	**Predicted Utility** $U^{pg}_{\mathcal{I}_{\mathcal{V}},\mathcal{E}}(\sigma,s)$	**Actual Utility** $U^{ag}_{\mathcal{I}_{\mathcal{V}},\mathcal{E}}(\sigma,s)$	**Predicted Utility** $U^{pg}_{\mathcal{I}_{\mathcal{P}},\mathcal{E}}(\sigma,s)$	**Actual Utility** $U^{ag}_{\mathcal{I}_{\mathcal{P}},\mathcal{E}}(\sigma,s)$
1	0.3300	0.5127	0.7592	0.7075
2	0.8716	0.7802	0.7940	0.7802
3	0.7983	0.6700	0.6910	0.6700
Avg. $U^{ag}_{\mathcal{I},\mathcal{E}}(\sigma,s)$	-	0.6543	-	0.7192
Avg. $\Delta U^{g}_{\mathcal{I},\mathcal{E}}(\sigma,s)$	0.1341		0.0288	

Table 3: Risk-averse adaptation analysis results.

case, once we have computed a policy for the system player to maximize predicted utility, we quantify the reward for actual guaranteed utility in the new game in which the system player policy has already been fixed:

$$U^{ag}_{\mathcal{I},\mathcal{E}}(\sigma,s) \triangleq \langle\langle\{Sys\}\rangle\rangle \mathsf{R}^{aur}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ leaf]$$

Table 7.3 shows the results of our experiments concerning the impact of strategies on guaranteed expected utility. The table is divided in two sections that compare the performance of risk-averse adaptation when using impact vectors and probabilistic impact models, respectively. Each section contains the value of predicted guaranteed utility ($U^{pg}_{\mathcal{I},\mathcal{E}}(\sigma,s)$), and actual guaranteed utility computed based on the predicted utility ($U^{ag}_{\mathcal{I},\mathcal{E}}(\sigma,s)$). If we focus on the actual utility of risk-averse adaptation when using probabilistic impact models, we can observe that it improves on average about 6.5% with respect to impact vectors. We also consider the accuracy of the predictions with each one of the models, defined as the difference between predicted and actual guaranteed utilities $\Delta U^{g}_{\mathcal{I},\mathcal{E}}(\sigma,s) = |U^{pg}_{\mathcal{I},\mathcal{E}}(\sigma,s) - U^{ag}_{\mathcal{I},\mathcal{E}}(\sigma,s)|$. In this case, the accuracy of probabilistic impact models improves more than 10% over vectors. This is particularly noticeable in the configuration with just one active server, in which the predicted utility with vectors deviates about 18% from the actual utility, whereas the the maximum deviation in the case of probabilistic impact models is approximately 5%.

In our experiments, the improvement in utility achieved using probabilistic impact models is modest with respect to vectors. However, it is worth observing that this stems from the fact that both the adaptation strategy analyzed and the space of system configurations are rather limited, leading to similar tactic selections for execution in the configurations with 2 and 3 active servers (i.e., resulting in similar actual utility values after strategy execution). In this sense, it is reasonable to

assume that more variability in terms of adaptation logic and environment behavior will favor the predictions of context-sensitive probabilistic models, leading to better-informed decisions and higher utility improvements.

8. Conclusions and Future Work

In this paper we addressed the specification of impact models for self-adaptive systems and presented a declarative language that allows one to explicitly represent the uncertainty in the outcome of adaptation actions and also to capture assumptions about the evolution of the environment. The mathematical underpinnings of the language were heavily influenced by the input language of PRISM [23], but its syntax is also based on the language of structural constraints of Acme [19]. The language was shown to have the ability to express sophisticated impact models, providing expressive and compact descriptions. Although there is an upfront investment in learning the notation and specifying these impact models compared to other approaches [9, 27], the fact that we can model variability improves reusability (e.g., across systems sharing the same architectural style).

We also showed how the proposed impact models can be used in the context of Rainbow with adaptation strategies defined in the language Stitch [9], and proposed two methods for calculating the utility of a strategy, one focused on average utility and the other on the utility that is guaranteed in the worst-case scenario. The benefits of the proposed impact models and analysis techniques can be extended to other architecture-based approaches to self-adaptation that rely on impact models for adaptation decision-making such as [24] and [27].

Regarding future work, we plan on extending our declarative language to cater to architectural styles that support structural changes (i.e., dynamic changes in the structure of the configuration graph). Moreover, we plan on leveraging and furthering formal analysis of adaptation behavior by encoding impact models described in our language into existing tools, such as UPPAAL stratego [14] and new versions of PRISM-games [8]. A third research direction aims at further refining our approach to capture timing aspects in our impact models, thus enabling reasoning about time-quality trade-offs. Finally, when there is limited availability of domain expert knowledge or field data about similar existing systems, proper parameterization of the impact models (e.g., of update functions, probabilities) may be challenging. To mitigate such situations, we plan on exploring machine learning techniques in order to allow designers to write parametric impact models in which actual parameter values can be automatically inferred and periodically updated from system observations.

Acknowledgements

This work is supported in part by awards N000141310401 and N000141310171 from the Office of Naval Research, CNS-0834701 from the National Science Foundation, by the National Security Agency, the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center, and by the Portuguese Foundation for Science and Technology (FCT) via projects CMU-PT/ELE/0030/2009 and CMUP-EPB/TIC/0042/2013. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Carnegie Mellon University or its Software Engineering Institute, the Office of Naval Research, or the U.S. government. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by any of the above sponsoring parties.

References

[1] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, Sept. 2002.

[2] A. Bianco and L. d. Alfaro. Model checking of probabalistic and nondeterministic systems. In *Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 499–513. Springer-Verlag, 1995.

[3] H. Boudali, P. Crouzen, B. Haverkort, M. Kuntz, and M. Stoelinga. Architectural dependability evaluation with arcade. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 512–521, June 2008.

[4] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive software needs quantitative verification at runtime. *Commun. ACM*, 55(9):69–77, Sept. 2012.

[5] R. Calinescu and M. Kwiatkowska. Using Quantitative Analysis to Implement Autonomic IT Systems. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 100–110, Washington, DC, USA, 2009. IEEE Computer Society.

[6] J. Cámara, A. Lopes, D. Garlan, and B. Schmerl. Impact models for architecture-based self-adaptive systems. In *Proceedings of the 11th International Symposium on Formal Aspects of Component Software (FACS2014)*, volume 7795 of *LNCS*, pages 1–19, 2015.

[7] T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, and A. Simaitis. Automatic verification of competitive stochastic systems. *Formal Methods in System Design*, 43(1):61–92, 2013.

[8] T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, and A. Simaitis. PRISM-games: A model checker for stochastic multi-player games. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *LNCS*, pages 185–191, Berlin, Heidelberg, 2013. Springer-Verlag.

[9] S.-W. Cheng and D. Garlan. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software*, 85(12):2860–2875, 2012.

[10] S.-W. Cheng, D. Garlan, and B. Schmerl. Evaluating the Effectiveness of the Rainbow Self-Adaptive System. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '09, pages 132–141, Washington, DC, USA, 2009. IEEE Computer Society.

[11] S.-W. Cheng, D. Garlan, and B. R. Schmerl. Raide for engineering architecture-based self-adaptive systems. In *ICSE Companion*, pages 435–436, 2009.

[12] S.-W. Cheng, D. Garlan, B. R. Schmerl, J. a. P. Sousa, B. Spitnagel, and P. Steenkiste. Using architectural style as a basis for system self-repair. In *Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance*, WICSA 3, pages 45–59, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[13] R. Chiulli. *Quantitative Analysis: An Introduction*. Automation and production systems. Taylor & Francis, 1999.

[14] A. David, P. Jensen, K. G. Larsen, M. Mikucionis, and J. H. Taankvist. Uppaal stratego. In C. Baier and C. Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 206–211. Springer Berlin Heidelberg, 2015.

[15] N. Esfahani and S. Malek. Uncertainty in self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems 2*, pages 214–238, 2010.

[16] A. Filieri, C. Ghezzi, A. Leva, and M. Maggio. Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements. In *ASE*, pages 283–292, 2011.

[17] V. Forejt et al. Automated verification techniques for probabilistic systems. In *Proceedings of SFM'11*, volume 6659 of *LNCS*, pages 53–113. Springer, 2011.

[18] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37:46–54, October 2004.

[19] D. Garlan, R. T. Monroe, and D. Wile. Acme: an architecture description interchange language. In *CASCON*, pages 169–183, 1997.

[20] A. Hartmanns. MODEST - A unified language for quantitative models. In *Proceeding of the 2012 Forum on Specification and Design Languages, Vienna, Austria, September 18-20, 2012*, pages 44–51. IEEE, 2012.

[21] J. E. Ingersoll. *Theory of financial decision making*. Rowman & Littlefield studies in financial economics. Rowman & Littlefield publ, Savage (Md.), 1987.

[22] C. Klein, M. Maggio, K.-E. Arzen, and F. Hernandez-Rodriguez. Brownout: Building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 700–711, New York, NY, USA, 2014. ACM.

[23] M. Kwiatkowska et al. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[24] P. Leitner, W. Hummer, and S. Dustdar. Cost-based optimization of service compositions. *IEEE T. Services Computing*, 6(2):239–251, 2013.

[25] J. R. Norris. *Markov chains*. Cambridge series in statistical and probabilistic mathematics. 1998.

[26] OMG. Object Constraint Language V2.4. `http://www.omg.org/spec/OCL/2.4`, february 2014.

[27] L. Rosa, L. Rodrigues, A. Lopes, M. A. Hiltunen, and R. D. Schlichting. Self-management of adaptable component-based applications. *IEEE Trans. Software Eng.*, 39(3):403–421, 2013.

[28] D. Sykes, D. Corapi, J. Magee, J. Kramer, A. Russo, and K. Inoue. Learning revised models for planning in adaptive systems. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 63–71, Piscataway, NJ, USA, 2013. IEEE Press.

Appendix A. Znn.com SMG PRISM Encoding

Our formal model of Znn.com is implemented in PRISM-games [8], a probabilistic model-checker for modeling and analyzing SMGs. The game is played in turns by two players in control of the behavior of the environment and the system, respectively. The SMG model consists of:

Appendix A.1. Player Definition

Listing 6 illustrates the definition of the players in the stochastic game: player env is in control of all the (asynchronous) actions that the environment can take (as defined in the environment module, Listing 7). Player sys controls all transitions that belong to the SimpleReduceResponseTime module (Listing 9), including all the transitions that synchronize with the target_system module (Listing 8), which represent the execution of tactics upon the target system (explicitly listed between square brackets in Listing 6, line 2). Global variable turn in Listing 6, line 5 is used to control turns in the game and make players alternate, ensuring that for every state of the model, only one player can take action.

```
1 player env environment endplayer
2 player sys target_system, SimpleReduceResponseTime, [enlistServer], [switchToTextualMode] endplayer
3
4 const ENV_TURN=1, SYS_TURN=2;
5 global turn:[ENV_TURN..SYS_TURN] init SYS_TURN;
6
7 formula te=turn=ENV_TURN;
8 formula ts=turn=SYS_TURN;
```

Listing 6: Player definition for Znn.com's SMG.

Appendix A.2. Environment

Listing 7 shows the encoding used for a simple version of Znn.com's environment, which is able to place an arbitrary amount of request arrivals in the execution context. Lines 1 defines the constant MAX_ARRIVALS_PPERIOD that parameterizes its behavior by limiting the maximum amount of arrivals that the environment can place per time period. [14]

```
1 const MAX_ARRIVALS_PPERIOD;
2 module environment
3 arrivals_current : [0..MAX_ARRIVALS_PPERIOD] init INIT_ARRIVALS;
4 hiLatency : bool init false;
5
```

[14] Constant values not defined in the model are provided as command-line input parameters to the tool.

```
6  [] (te) −> (arrivals_current'=0) & (hiLatency'=true) & (turn'=SYS_TURN);
7  [] (te) −> (arrivals_current'=0) & (hiLatency'=false) & (turn'=SYS_TURN);
8  . . .
9  [] (te) −> (arrivals_current'=X) & (hiLatency'=true) & (turn'=SYS_TURN);
10 [] (te) −> (arrivals_current'=X) & (hiLatency'=false) & (turn'=SYS_TURN);
11 endmodule
```

Listing 7: environment module.

Moreover, lines 3-5 declare the different variables that define the state of the environment: arrivals_current (line 3) corresponds to the number of request arrivals introduced by the environment in the current turn. hiLatency (line 4) abstracts in a boolean variable a high level of network latency.

Each turn of the environment consists of setting the amount of request arrivals. This is achieved through a set of commands that follow the pattern shown in Listing 7, lines 9-10: the guard in the command checks that it is the turn of the environment to move (te). If the guard is satisfied, the command: (i) sets the value of request arrivals and latency level for the current time period (represented by X in the command), and (ii) modifies the value of turn, yielding control to the system player. Note that there may be as many of these commands as possible values can be assigned to request arrivals for the current environment turn, including zero for no arrivals (lines 6-7), resulting in a natural encoding of the adversary semantics of the environment as described in Definition 11.

Appendix A.3. Target System

Module target_system (Listing 8) models the behavior of the target system in which valid configurations may include up to four servers. This module encodes the behavior of the system (including the execution of tactics upon it), and is parameterized by the constants:

- MIN_SERVERS and MAX_SERVERS, which specify the minimum and maximum number of active servers that a valid system configuration can have, respectively.
- MAX_RT and INIT_RT, which specify the system's maximum and initial response times, respectively.
- MAX_FIDELITY, MIN_FIDELITY, which specify the minimum and maximum fidelity levels of a server (in this case, we assume that MAX_FIDELITY indicates that the server is operating in multimedia mode, whereas MIN_FIDELITY indicates that the server is in textual mode).

- INIT_SX_ACTIVE and INIT_SX_F, specify whether server X is initially active and its initial fidelity level, respectively.

```
formula cert=(c1_cert+c2_cert)/2; // Predicted response time (probabilistic impact models)

module target_system
start : bool init true;

rt : [0..MAX_RT] init INIT_RT; // Actual response time (queuing model)
ert: [0..MAX_RT] init INIT_RT; // Predicted response time (fixed impact vectors)
c1_cert: [0..MAX_RT] init INIT_RT1; // Predicted response time (probabilistic impact models)
c2_cert: [0..MAX_RT] init INIT_RT2;

s1_active:[0..1] init INIT_S1_ACTIVE;
s2_active:[0..1] init INIT_S2_ACTIVE;
s3_active:[0..1] init INIT_S3_ACTIVE;
s4_active:[0..1] init INIT_S4_ACTIVE;

s1_f:[MIN_FIDELITY..MAX_FIDELITY] init INIT_S1_F;
s2_f:[MIN_FIDELITY..MAX_FIDELITY] init INIT_S2_F;
s3_f:[MIN_FIDELITY..MAX_FIDELITY] init INIT_S3_F;
s4_f:[MIN_FIDELITY..MAX_FIDELITY] init INIT_S4_F;

 [] (ts) & (start) -> (ert'=s_tt) & (c1_cert'=s_tt) & (c2_cert'=s_tt) & (rt'=s_tt) & (start'=false);

// Pass on enlistServer (applicability condition not satisfied)
 [enlistServer] (!start) & (s>=MAX_SERVERS) -> (rt'=s_tt);

// Execution of enlistServer (Probabilistic impact models variant)
 [enlistServer] (!start) & (s<MAX_SERVERS) & (PMODELS) ->
                        (0.665)*(s1_active=0?(1/inactive_s):0):(s1_active'=(s1_active=0?1:0))
                                                & (c1_cert'= s_es_f_rt1) & (c2_cert'= s_es_f_rt2) & (rt'=s_tt) +
     (0.285)*(s1_active=0?(1/inactive_s):0):(s1_active'=(s1_active=0?1:0))
                                                & (c1_cert'= s_es_g_rt1) & (c2_cert'= s_es_g_rt2) & (rt'=s_tt) +
     (0.665)*(s2_active=0?(1/inactive_s):0):(s2_active'=(s2_active=0?1:0))
                                                & (c1_cert'= s_es_f_rt1) & (c2_cert'= s_es_f_rt2) & (rt'=s_tt) +
     (0.285)*(s2_active=0?(1/inactive_s):0):(s2_active'=(s2_active=0?1:0))
                                                & (c1_cert'= s_es_g_rt1) & (c2_cert'= s_es_g_rt2) & (rt'=s_tt) +
     (0.665)*(s3_active=0?(1/inactive_s):0):(s3_active'=(s3_active=0?1:0))
                                                & (c1_cert'= s_es_f_rt1) & (c2_cert'= s_es_f_rt2) & (rt'=s_tt) +
     (0.285)*(s3_active=0?(1/inactive_s):0):(s3_active'=(s3_active=0?1:0))
                                                & (c1_cert'= s_es_g_rt1) & (c2_cert'= s_es_g_rt2) & (rt'=s_tt) +
     (0.665)*(s4_active=0?(1/inactive_s):0):(s4_active'=(s4_active=0?1:0))
                                                & (c1_cert'= s_es_f_rt1) & (c2_cert'= s_es_f_rt2) & (rt'=s_tt) +
     (0.285)*(s4_active=0?(1/inactive_s):0):(s4_active'=(s4_active=0?1:0))
                                                & (c1_cert'= s_es_g_rt1) & (c2_cert'= s_es_g_rt2) & (rt'=s_tt) +
     (0.05): (rt'=s_tt);

// Execution of enlistServer (Fixed impact vectors variant)
 [enlistServer] (!start) & (s<MAX_SERVERS) & (VECTORS) ->
     (s1_active=0?(1/inactive_s):0) : (s1_active'=(s1_active=0?1:0)) & (ert'=es_f_rt) & (rt'=s_tt) +
     (s2_active=0?(1/inactive_s):0) : (s2_active'=(s2_active=0?1:0)) & (ert'=es_f_rt) & (rt'=s_tt) +
     (s3_active=0?(1/inactive_s):0) : (s3_active'=(s3_active=0?1:0)) & (ert'=es_f_rt) & (rt'=s_tt) +
     (s4_active=0?(1/inactive_s):0) : (s4_active'=(s4_active=0?1:0)) & (ert'=es_f_rt) & (rt'=s_tt);

// Pass on switchToTextualMode (applicability condition not satisfied)
```

```
54  [switchToTextualMode] (!start) & (f<=MIN_FIDELITY) -> (rt'=s_tt);
55
56  // Execution of switchToTextualMode (Probabilistic impact models variant)
57  [switchToTextualMode] (!start) & (f>MIN_FIDELITY) & (PMODELS) ->
58      (0.8*0.8): (s1_f'=(s1_active=1 & s1_f>MIN_FIDELITY?s1_f-1:s1_f)) & (s2_f'=(s2_active=1 & s2_f>
            MIN_FIDELITY?s2_f-1:s2_f)) &
59                      (s3_f'=(s3_active=1 & s3_f>MIN_FIDELITY?s1_f-1:s3_f)) & (s4_f'=(s4_active=1 & s4_f>
                            MIN_FIDELITY?s4_f-1:s4_f))
60                &(c1_cert'=s_lf_f_rt1) & (c2_cert'=s_lf_f_rt2) & (rt'=s_tt) +
61      (0.8*0.2): (s1_f'=(s1_active=1 & s1_f>MIN_FIDELITY?s1_f-1:s1_f)) & (s2_f'=(s2_active=1 & s2_f>
            MIN_FIDELITY?s2_f-1:s2_f)) &
62                    (s3_f'=(s3_active=1 & s3_f>MIN_FIDELITY?s1_f-1:s3_f)) & (s4_f'=(s4_active=1 & s4_f>
                          MIN_FIDELITY?s4_f-1:s4_f))
63                               &(c1_cert'=s_lf_f_rt1) & (c2_cert'=s_lf_g_rt2) & (rt'=s_tt) +
64      (0.2*0.8): (s1_f'=(s1_active=1 & s1_f>MIN_FIDELITY?s1_f-1:s1_f)) & (s2_f'=(s2_active=1 & s2_f>
            MIN_FIDELITY?s2_f-1:s2_f)) &
65                                 (s3_f'=(s3_active=1 & s3_f>MIN_FIDELITY?s1_f-1:s3_f)) & (s4_f'=(s4_active=1 &
                                       s4_f>MIN_FIDELITY?s4_f-1:s4_f))
66                           &(c1_cert'=s_lf_g_rt1) & (c2_cert'=s_lf_f_rt2) & (rt'=s_tt) +
67      (0.2*0.2): (s1_f'=(s1_active=1 & s1_f>MIN_FIDELITY?s1_f-1:s1_f)) & (s2_f'=(s2_active=1 & s2_f>
            MIN_FIDELITY?s2_f-1:s2_f)) &
68                            (s3_f'=(s3_active=1 & s3_f>MIN_FIDELITY?s1_f-1:s3_f)) & (s4_f'=(s4_active=1 & s4_f
                                  >MIN_FIDELITY?s4_f-1:s4_f))
69                      &(c1_cert'=s_lf_g_rt1) & (c2_cert'=s_lf_g_rt2) & (rt'=s_tt);
70
71  // Execution of switchToTextualMode (Fixed impact vectors variant)
72  [switchToTextualMode] (!start) & (f>MIN_FIDELITY) & (VECTORS) ->
73                                 (s1_f'=(s1_active=1 & s1_f>MIN_FIDELITY?s1_f-1:s1_f)) & (s2_f'=(s2_active=1 &
                                       s2_f>MIN_FIDELITY?s2_f-1:s2_f)) &
74                                 (s3_f'=(s3_active=1 & s3_f>MIN_FIDELITY?s1_f-1:s3_f)) & (s4_f'=(s4_active=1 &
                                       s4_f>MIN_FIDELITY?s4_f-1:s4_f)) &
75                                 (ert'=lf_f_rt) & (rt'=s_tt);
76  endmodule
```

Listing 8: target_system module.

The module employs the following set of variables to represent the system state: rt (line 6) is the actual response time of the system (computed based on the request arrivals in the environment, the current level of fidelity of the contents served, and the number of active servers, according to an M/M/c queuing model). ert (line 7) is the predicted response time, according to the impact vectors. cx_cert (lines 8-9) is the predicted response time for client x according to the probabilistic impact models for tactics, whereas cert (line 1) is the overall predicted response time based on probabilistic impact models. sx_active (lines 11-14) indicates whether server x is currently active, and sx_f (lines 16-19) indicates the current fidelity level of server x.

Each tactic that can be executed upon the target system is represented by a set of commands labelled with the tactic's name. These commands are guarded by the applicability condition of the tactic (e.g., a specific server should be inactive to be enlisted), updating the different state variables of the system according to

the effect of the tactic's execution:

- enlistServer activates an inactive server, setting the value of sx_active to true, and updating the value of actual response time according to the value computed by the queuing model with the new number of active servers (encoded in formula s_tt). The command in lines (27-44) that includes in the guard the additional predicate PMODELS encodes in addition the update of predicted response time cert according to the probabilistic impact model for the tactic described in Listing 3. Alternatively, the command in lines 47-51 (guarded by predicate VECTORS) updates predicted response time ert according to the impact vectors defined for the tactics.

- switchToTextualMode lowers the fidelity of an active server to textual mode, decreasing response time (modifying variables sx_f). In this case, the commands also include a version encoding the probabilistic impact model for this tactic defined in Listing 1 (lines 57-69), and a version encoding updates of predicted response time with impact vectors (lines 72-75).

An additional set of commands guarded by the negation of tactic applicability conditions model situations in which the traversal of the strategy tree progresses without the execution of the tactic when applicability conditions are not met (lines 24 and 54 for tactics enlistServer and switchToTextualMode, respectively).

Appendix A.4. Adaptation Logic

Module SimpleReduceResponseTime (Listing 9) models the execution tree of the Stitch adaptation strategy shown in Figure 7, in which each command corresponds to the different branches in the tree including the execution of the tactic that can be executed on the target system.

```
1  module SimpleReduceResponseTime
2  node : [0..5] init 0;
3  end : bool init false;
4  leaf : bool init false;
5  yield : bool init false;
6
7  [enlistServer] (ts & !yield) & (node=0) -> (node'=1) & (leaf'=false) & (yield'=true);
8  [switchToTextualMode] (ts & !yield) & (node=0) -> (node'=2) & (leaf'=false) & (yield'=true);
9  [switchToTextualMode] (ts & !yield) & (node=1) -> (node'=3) & (leaf'=true) & (yield'=true);
10 [enlistServer] (ts & !yield) & (node=2) -> (node'=4) & (leaf'=true) & (yield'=true);
11 [] (ts & yield & !leaf)    -> (yield'=false) & (turn'=ENV_TURN);
12 [] (ts & yield & leaf & !end) -> (end'=true);
13 endmodule
```

Listing 9: SimpleReduceResponseTime module.

The module contains a set of synchronous commands, each one corresponding to one of the branches in the strategy tree that include tactics to be executed on the target system. Each one of them can synchronize with any of the commands labeled with the same action name in the target_system module (e.g., the command in Listing 9, line 7, could synchronize with the commands in Listing 8, in lines 24, 27, or 47 to enlist a server). The system player is in control of all these synchronous transitions (as defined in Listing 6, line 2). A synchronous command for the execution of a tactic in a branch can only be fired if: (i) It is the turn of the system to take action (ts), (ii) the execution of the strategy has advanced to a point in which the current branch can be taken (explicitly encoded in variable node), and (iii) it is not the end of the system's turn (explicitly encoded in variable yield).

In addition, the module also contains two asynchronous commands: the first one (line 11) yields the turn to the environment player after a branch in the strategy tree has been traversed via the execution of its corresponding command, and a leaf node has not been reached. The second one (line 12) sets the value of the end variable to true whenever a leaf node in the strategy has been reached and it is the end of the system's turn, indicating the end of the game.

Appendix A.5. Utility profile

Utility functions and preferences are encoded using formulas and reward structures that enable the quantification of instantaneous utility in states of the model that correspond to leaf nodes in the adaptation strategy tree. Specifically, formulas compute utility on the different dimensions of concern, and reward structures weigh them against each other by using the utility preferences.

```
1 const W_UR, W_UF, W_UC;
2 formula uR = (rt>=0 & rt<=100? 1:0)
3         +(rt>100&rt<=200?1+(−0.01)∗((rt−100)/(100)):0) ...
4         +(rt>2000&rt<=4000?0.25+(−0.25)∗((rt−2000)/(2000)):0)
5         +(rt>4000 ? 0:0); ...
6 rewards "rAUR" leaf : W_UR∗uR + W_UF∗uF + W_UC∗uC; endrewards
```

Listing 10: Utility functions and preferences encoding.

Listing 10 illustrates in lines 2-5 the encoding of utility functions using a formula for linear interpolation based on the points defined for utility function U_R in the first column of Table 1. The formula in the example computes the actual utility for performance, based on the value of the variable for system response time rt (analogous rewards are defined in the game for predicted utility rewards based on fixed impact vectors and probabilistic models using the values of variables ert and cert, respectively). Moreover, line 6 shows how a reward structure can be

defined to compute a utility value for any state by using utility preferences (defined in line 1 as weights W_UR, W_UF, and W_UC for performance, fidelity, and cost respectively). Labeling leaf states in the model with utility rewards in such a way effectively enables the synthesis of optimal policies leading to target system configurations that maximize guaranteed utility, as described in Section 7.3.