Improving Self-Adaptation Planning through Software Architecture-based Stochastic Modeling

João M. Franco*, Francisco Correia, Raul Barbosa, Mário Zenha-Rela

Center for Informatics and Systems of University of Coimbra, Coimbra, Portugal

Bradley Schmerl, David Garlan

Carnegie Mellon University, Pittsburgh, USA

Abstract

The ever-growing complexity of software systems makes it increasingly challenging to foresee at design time all interactions between a system and its environment.

Most self-adaptive systems trigger adaptations through operators that are statically configured for specific environment and system conditions. However, in the occurrence of uncertain conditions, self-adaptive decisions may not be effective and might lead to a disruption of the desired non-functional attributes.

To address this, we propose an approach that improves the planning stage by predicting the outcome of each strategy. In detail, we automatically derive a stochastic model from a formal architecture description of the managed system with the changes imposed by each strategy. Such information is used to optimize the self-adaptation decisions to fulfill the desired quality goals.

To assess the effectiveness of our approach we apply it to a cloud-based news system and predicted the reliability for each possible adaptation strategy. The results obtained from our approach are compared to a representative static planning algorithm as well as to an oracle that always makes the ideal decision. Experiments show that our method improves both availability and cost when

*Corresponding author

Email addresses: jmfranco@dei.uc.pt (João M. Franco), fcorreia@dei.uc.pt (Francisco Correia), rbarbosa@dei.uc.pt (Raul Barbosa)

compared to the static planning algorithm, while being close to the oracle.

Our approach may therefore be used to optimize self-adaptation planning.
Keywords: Self-adaptive systems, Autonomic computing, ADL, Reliability Prediction, Quality Goals, Znn.com, Impact prediction

---

1. Introduction

Self-adaptive systems modify their structure or behavior during runtime in order to meet specified goals (*e.g.*, availability, performance). These systems are monitored to obtain runtime properties which are analyzed to identify conditions where the system may be deviating from the desired quality goals. In these situations, adaptation courses are planned and executed to get the system into the right track. However, in view of the critical nature of the system, adaptations should be planned with care and take account of every possible scenario. For example, self-driving cars have strict safety requirements and whenever an adaptation is planned, it must ensure that no human lives (passengers or pedestrians) are put at risk.

A self-adaptive system determines an adaptation based on different approaches. Current adaptation approaches vary from simple algorithms that are condition-action based to other, more complex approaches that involve Markov Decision Processes (MDPs) or utility-theory [1]. Generally, these decision-making algorithms act according to a predefined set of operators to choose one adaptation over another. The problem is that these operators are considered to be static [2, 3], being defined by humans and only effective in specific domains or expected contexts in which they have been configured [4]. In systems with a high number of runtime and environment variables, the possible adaptation scenarios and consequences may rise exponentially, and become almost unfeasible for humans to reflect on in every possible combination. As a result, under unexpected conditions a system may fail to select the best strategy which may lead to a degradation of the provided services.

To overcome the limitation that occurs with static operators responsible

for triggering adaptations, we propose a method that automatically predicts whether an adaptation fulfills the desired non-functional goals, even in unexpected conditions. This method automatically predicts the reliability for each adaptation strategy and updates the static operators. As a result, our approach determines a strategy that best drives the system to attain the desired quality goals.

The main contribution of this paper is an approach that seeks to improve the planning phase of self-adaptive systems, by automatically anticipating the reliability of each adaptation on non-functional properties. In addition, we also propose a notation to formally translate a software architecture to a mathematical model, promoting researchers to apply our approach, automate their methods or extend it to other quality attributes. This formal translation poses as a novel approach and a contribution for future research.

To show the effectiveness of our approach we applied it to a cloud-based infrastructure system with three adaptation goals: availability, cost and resources utilization. Specifically, we anticipate the impact of each eligible adaptation by generating stochastic models at runtime. These models are derived from the software architecture description which allow to describe the behavior, structure and actual properties of the running system.

In our experiments, we simulated failures and load peaks in order to compare the results from our approach with traditional self-adaptive methods that use static adaptation operators. The results show that our approach is able to correctly handle unexpected conditions, in which traditional approaches fail to select the best adaptation strategy.

This paper is organized as follows. Section 2 presents the related work and Section 3 details the method adopted in this study. Section 4 describes the hypotheses which our study aims to confirm, Section 5 introduces the case-study used for evaluating our approach and its results can be depicted in Section 6. The contributions are presented in Section 7 and a discussion about the results and method is addressed in Section 8. Section 9 summarizes the practical implications before Section 10 concludes.

2. Related work

Self-adaptive systems are able to adjust their behavior in response to their perception of the environment and the system itself [5]. These systems are usually implemented through the MAPE-K approach defined by the International Business Machines (IBM) Corporation in 2004 [6]. The MAPE-K is a short name for Monitor, Analysis, Plan and Execute tasks, all with a shared Knowledge-base. To be more specific, a self-adaptive system *monitors* the environment and the system itself to *analyze* whether an adaptation is required or not to achieve the desired goals. In case of being necessary, a course of action is *planned* and *executed* in the target system to change the current behavior and achieve the desired quality goals. Communication between different adaptation phases is conducted through a shared *knowledge*-base that abstracts the system, containing data, models, decisions and behavior, enabling separation of adaptation responsibilities and allowing their coordination.

Self-adaptive systems have been an interesting focus of research study due to their ability to adapt and modify the behavior leading to a multitude of practical applications, like self-driving cars, self-maintainable software and self-ruling systems. Salehie *et al.* [4] present a survey article about the landscape of research, taxonomies, gaps, and future challenges in self-adaptive systems. They consider the adaptation process as a concept that deserves attention in future research challenges. One of the most important challenges they highlighted is the assurance that an adaptation is going to have a predictable impact on the functional and non-functional aspects of the system. To this end, we divide the approaches that relate to ours into three different groups: approaches that propose changes to the phases of the MAPE-K (Monitor, Analysis, Plan, Execute and Knowledge) loop; present new decision-making algorithms; and perform quantitative prediction of non-functional attributes or verification of correctness.

2.1. Changes to the MAPE-K loop

Fredericks *et al.* [2] present an exploratory paper introducing a change in the typical control-adaptation loop. They argue that the traditional feedback-loop

based on the MAPE-K [6], should be supplemented with testing strategies at runtime, becoming MAPE-T. Their goal is that through such testing at runtime, the self-adaptive system would be able to verify the satisfaction of its requirements, even when facing unexpected or unanticipated system or environment conditions. Being an exploratory paper, it lacks the application of the theory into a practical example.

King *et al.* [7] suggest a self-testing framework for autonomic computing. This framework must ensure that prior to each adaptation: (1) the system has to perform regression tests to ensure that new errors have not been introduced; and (2) validate the actual behavior of newly added or adapted components prior to their use in the system. This study differs from ours since King's work focuses on assuring the correct functional behavior before and after each adaptation. Our study aims on predicting the impact of the non-functional goals for each strategy and informing the planning phase to make appropriate decisions.

2.2. Decision-making approaches

In recent years several studies focused on proposing new methods to support decision-making algorithms. These studies had one goal in common: deal with the uncertainty of adaptive decisions to assure a correct adaptation while guaranteeing the achievement of the desire quality goals. Traditional self-adaptive approaches rely on constant weights or impact vectors to select the best strategy according to environment and system conditions. However, these constants are defined in design time and may become dated during runtime due to the varying and dynamic environment conditions. Thus, it leads to an uncertainty in the non-functional aspects of the system.

To address this uncertainty, Bencomo *et al.* [8] apply Bayesian Networks, specifically Dynamic Decision Networks (DDNs), to assess the consequences of different design alternatives and choose one that satisfies the functional requirements of the system. With the same purpose, Ghezzi *et al.* [9] propose the generation of a Markov Decision Process (MDP) with rewards from an abstract model of the system comprising a set of functionalities and their alternatives.

This MDP calculates which is the best path of execution regarding the system quality goals. In addition, their work handles unexpected faults by redirecting the execution for alternative paths when catching a runtime exception.

Sykes *et al.* [10] uses a probabilistic rule learning technique to generate new adaptation plans. These plans are ranked with a probability of achieving a particular quality goal under specific system and environment conditions. They applied their method to a case-study based on a production cell where a robot arm is controlled and various processing operations were performed. The results show that the algorithm was able to produce alternative plans and avoid the use of faulty components.

The work described in [11, 12] represents recent work on using probabilistic models in self-adaptation in the context of Rainbow. That work primarily focuses on modeling the adaptation strategies and their impacts with respect to quality goals. Although the architectural model is woven in with these models, the architecture itself is not probabilistic. In contrast, the work described here focuses on the probabilistic behavior of the architecture itself and uses that to predict the quality impact of strategies. Some basic probabilistic modeling of the architectural model is done in [13] for the purposes of synthesizing strategies, but the probabilistic aspects are encoded as simple parameters, and so are not full probabilistic models of the architecture.

Our work relates to the aforementioned ones by also proposing a method to tackle the uncertainty of quality attributes in self-adaptive systems. However, we do not propose a new decision-making algorithm, but rather we promote the quantitative prediction of non-functional attributes at runtime. Our approach uses this prediction to override constant weights and impact vectors defined at design time to avoid the use of static configurations. The novelty of our work relies in predicting the quality outcome for every possible adaptation strategy in order to assure the achievement of the desired quality goals.

2.3. Quantitative verification or prediction of quality goals

Quantitative verification is a technique to calculate the likelihood of the occurrence of certain events during the execution of the system. The benefits of having this verification at runtime to support software adaptation are discussed by Calinescu *et al.* [14]. They state that by employing modeling techniques at runtime (*e.g.*, predict requirements violation, plan recovery from such violations and verify correctness in the adaptation steps employed in recovery) we may obtain more dependable self-adaptive systems.

The work of Gallotti *et al.* [15] proposes an approach to generate stochastic models to assess reliability and performance from Activity Diagrams described in Unified Modeling Language (UML). In short, their approach takes as input a formal representation of service composition drawn as a UML Activity Diagram along with a specification of quality properties, such as response time or failure rate. The approach interprets the draw and creates an intermediate representation before generating a stochastic model to be solved by Prism [16], a model checking tool. The interpretation of the draw cannot be standardized for every UML Activity Diagram tool, since their representations differ in small details. This work differs from ours, since we focus on Architectural Description Languages (ADLs) than UML and we also propose a formal notation to standardize the translation from an architectural model complying with the ISO/IEC/IEEE 42010 Standard [17] to a stochastic model. In addition, we perform reliability prediction at runtime and show its effectiveness by conducting a performance assessment, while Gallotti *et al.* address these as future work.

Cámara *et al.* [18] propose an approach that models the behavior of a self-adaptive system with regard to trustworthy service delivery. In more detail, the authors model the adaptation behavior of the system to obtain levels of confidence regarding the resilience of each adaptation. The effectiveness of their approach is outlined through an experimentation similar to ours, using Rainbow and Znn.com as self-adaptive solution, respectively. The results show that both outcomes from the proposed modeling approach and from the running system are close validating their work. The work of Cámara *et al.* [18] differs from

ours in the aspect that we automatically generate and solve stochastic models at runtime to support the adaptation manager by deciding which is the best strategy to attain the desired quality goals.

Zheng *et al.* [19] applied Kalman Filters to model and track performance that can be used to evaluate end-to-end response times, utilization of resources and also to estimate performance parameters. Their work has been applied to autonomic computing to empower decision-making capabilities. The approach was tested in a scenario of a cluster of servers and the results show that it efficiently maintain service level and avoid system overload. In more detail, it takes into consideration disturbance changes such as the number of users, software aging or requests with modified resource demands.

Filieri *et al.* [20] explore models and system adaptations to meet a particular target reliability through a control theoretical approach. In more detail, they keep alive a model of the application at runtime which expresses reliability concerns through a DTMC. This model is continuously updated at runtime and, in a control-theory viewpoint, is viewed as the input variables to the controlled system. They consider self-adaptation at the model level, where possible variant behaviors are evaluated and the selected changes are then transferred into the running implementation. Their approach bypasses the decision-making process of the self-adaptive system and rely upon only one adaptation goal: Reliability.

Any of the aforementioned studies that address quantitative prediction or verification at runtime are likely applicable to a self-adaptive system to enrich its decision-making process. This enrichment is made by replacing constant adaptation operators by predictions on the quality outcome or by assuring correctness for each adaptation. We argue that the resulting system will be able to make informed decisions about the impact on the quality dimensions in unexpected and unanticipated situations. Other studies address uncertainty by keeping a model alive which is constantly updated with runtime properties. However, the construction of the model is the responsibility of the designer or the engineer, increasing development effort, time to deliver and cost. This is where our work augments the current research field by automatically generating

probabilistic models at runtime reducing effort and modeling time, at the same time that accounts with structural changes in the architecture such as changing architectural styles or the addition of new components.

The work that closely relates to ours is QoSMOS (Quality of Service Management and Optimization of Service-based systems) proposed by Calinescu *et al.* [21]. QoSMOS assures that QoS is delivered by adaptive systems in an equally adaptive and predictable way. QoSMOS support self-adaptation of service-based systems by choosing an optimal strategy through prediction of QoS at runtime. In short, they combine several existing techniques, such as the formal specification of QoS by using temporal logic, generation of stochastic models to evaluate reliability and performance, Bayesian-based parameter adaptation by exploiting KAMI [22] and a tool to support the planning and execution phases of the system adaptation. Our work closely relates to QoSMOS since both works assure quality of the system adaptation, perform quality prediction at runtime and evaluate the approach scalability and performance. Regarding the difference of both works, QoSMOS takes as input BPEL (Business Process Execution Language) models of service orchestration focusing on only service based systems, while our work uses Architectural Description Languages (ADLs). ADLs allow to specify a wider range of systems, reveal the topology or structure of the whole system and define architectural styles. In addition, we propose a formal notation to translate from the ADL to the stochastic model providing a generic solution that can be applied to other quality attributes or ADLs. Regarding QoSMOS scalability and evaluation efficiency, it cannot be applied to large scenarios due to the exhaustive quantitative model checking in the Analysis phase. In more detail, QoSMOS evaluates six different PCTL rules (4 for reliability and 2 for performance) while our approach assesses one rule that expresses the reliability of the system. This limits QoSMOS application to only systems in which time efficiency is not a problem, while our approach can be applied to those which have strict time requirements. The evaluation of QoSMOS is obtained through a theoretical case-study of a TeleAssistance scenario while our approach has been implemented and running on an actual case-study of a news infrastructure

system. This case-study allows to validate our approach with an application example, as well as to compare results from traditional approaches with ours.

3. Approach

In this study we use quantitative prediction of non-functional attributes to assure that a particular adaptation strategy will fulfill the desired quality goals. In more detail, we applied this approach to reliability by generating mathematical models at runtime to predict the failure behavior of the system for each eligible adaptation strategy. To this end, our approach uses a software architecture in the form of an Architectural Description Language (ADL) to assess each strategy. In more detail, the ADL contains data about the topology of the system, their current metrics and how it will behave accordingly to the adaptation strategy.

This section details our approach by explaining where it sits in the self-adaptation loop, the required specifications in the architecture, how we predict reliability and how we formally translate from the ADL to the stochastic model.

3.1. MAPE-K integration

IBM [6] in 2006 introduced a standard adaptation control loop with the ability to manage itself and dynamically adapt to changes – the MAPE-K. In more detail, the MAPE-K loop can be divided into Monitoring capabilities, Analyze runtime metrics, Plan strategies and Execute the plan through a shared Knowledge-base. Figure 1 provides an overview of our approach and its integration into the different phases of the MAPE-K loop.

In concrete terms, our approach begins by collecting runtime metrics from a running system to update its architectural description. This process ensures that at each recurring analysis phase, the model of the system is updated to the current environment and system conditions. In the *Analyze* phase, our approach makes a copy of the software architecture by following each possible adaptation strategy and applies its changing operators. These operators are

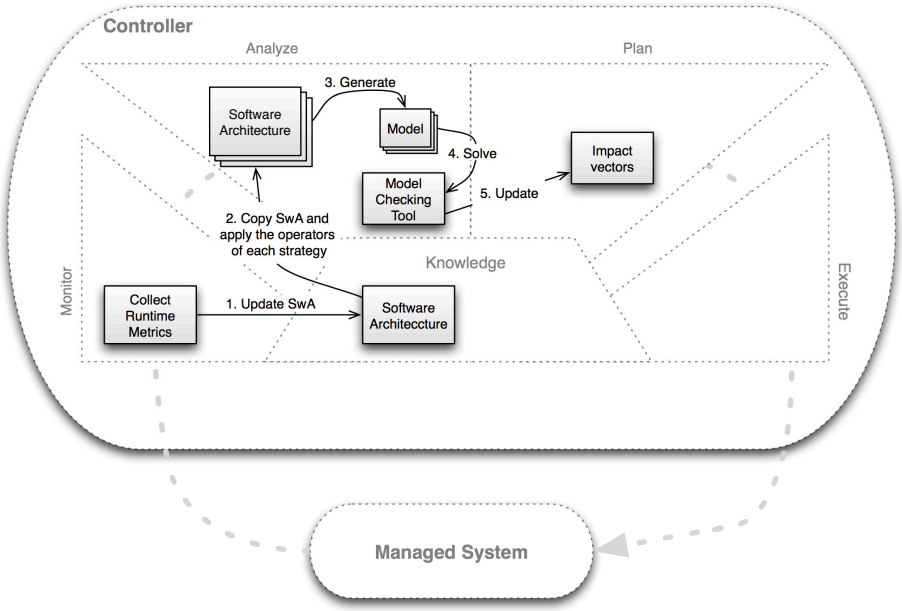


Figure 1: Approach overview

defined as the changes that each strategy would perform in the managed system if they were selected. Thus, our method generates a model that represents the system behavior for each adaptation strategy. To solve the generated model, our approach relies on a model checking tool, Prism [16], to predict the quality of the impact. In the final stage, our approach supports the planning phase by updating constant weights or impact vectors which can be used for comparative purposes and helps decide what is the best strategy to achieve the desire quality goals.

In the following subsections we formally describe the architectural model and the process to generate the stochastic model from the architecture. This formal specification is performed through the Z notation [23].

3.2. Architectural description

An architecture comprises what is essential or fundamental to a system in relation to its environment. Its description is a work product from the standpoint of architects and may encompass system constituents (*e.g.*, components, connectors), about how they are organized, their design requirements and principles regarding evolution [17].

Thus, an architectural model is a tuple $A = (C,\ Con,\ Att,\ Prop)$, where:

- $C = \{c_i\}$ is a finite set of Components. A component represents a unit of computation which can be a single operation, such as a function, a class or a set of classes that share the same interfaces or functionality, or even a complex operation as an entire system. We refer to a component as a tuple $c_i = (IP, OP, Prop, Rep)$ where:

  $IP = \{ip_j\}$ is a finite set of input ports. Each input port represents the incoming data to be processed by the component;

  $OP = \{op_k\}$ is a finite set of output ports. Each output port represents the data sent from a component after being processed;

  Prop is a set of properties annotating the component with data regarding its behavior. Each property is a tuple that holds information about the name of the property, its type and its value. For example, a component may hold a property representing its response time which is a float value representing its current or average response time.

$$
\begin{aligned}
Enum &= \{String\} \\
PropType &::= Float \mid Integer \mid String \mid Enum \\
Prop &= \{name, value : String\ ;\ type : PropType \bullet \\
&\qquad (name, type, value)\}
\end{aligned}
$$

  Rep is a representation that specifies the internal behavior of a component c_i. This internal representation is optional in each component

and when it exists describes a sub-architecture model that specifies in detail the functionality of that component and it is modeled as an architecture.

$$Rep = A' \mid \varnothing \Leftrightarrow Rep = \big(C',\ Con',\ Att',\ Prop'\big) \mid \varnothing$$

- $Con = \{con_i\}$ is a finite set of Connectors. Connectors are the architectural elements responsible for the interactions between components, distributing data among attached components. Each connector is represented by the tuple $con_i = (R, Prop)$:

  $R = \{r_i\}$ is a finite set of Roles. Each Role is responsible for coordinating the communication between the connector and a set of components, by specifying the communication protocol, assurance properties and the rules about interaction ordering or format. It is specified as a tuple $r_i = (Prop)$ where it defines its own properties. The connector role is bound as one-to-one with a component port and each connector must have at least two roles.

  Prop is a set of properties annotating the connector with data regarding its behavior and defined as the same type specified above.

- *Att* is an Attachment showing how components and connectors are bound together. In more detail, an Attachment is a tuple that specifies a component and its port (either input or output) that are connected to a role of a connector. As a result, one can understand how data traverses within the architecture and its elements.

$$Att = \{c_i \in C\,;\ ip_j \in IP(c_i)\,;\ op_k \in OP(c_i)\,;\ con_l \in Con\,;\ r_m \in R(con_l) \bullet \big((c_i,\ ip_j\ \cup\ op_k),\ (con_l,\ r_m)\big)\}$$

- *Prop* is a set of properties that annotates the architecture with data regarding requirements or design principles. Each property is defined as the

same *Prop* type previously specified.

This formal specification of the architecture in Z enables us to define any system in an unambiguous and rigorous way. As a result, we can automatically generate adequate stochastic models from the architectural specification. In the next section, we specify how we quantitatively predict reliability as a quality attribute from a software architecture.

3.3. Quantitative prediction of reliability

In this study we propose quantitative prediction of quality attributes at runtime to support the planning phase avoiding the use of constant weights. Considering the case-study presented in Section 5, one of the adaptation goals is to achieve high availability in the long-run. Thus, the system needs to drive adaptations to cope with failures and recover to a correct state. As such, if the system detects a failure, it triggers an adaptation as a recovery action to assure the "continuity of correct service" [24] or as "it is not doing the wrong thing" [25]. Our approach encompasses methods for quantitative prediction of reliability to guarantee that the planning phase selects the best strategy to achieve in the long-run a high availability, one of the quality goals described in Section 5.3.1.

To express the reliability behavior of a system a widely accepted method is the use of a Discrete-Time Markov Chain (DTMC) [26, 20, 27].

A discrete-time Markov Chain is a tuple $M = (S, \overline{s}, P, L)$, where:

- S is a finite, non-empty set of states;
- $\overline{s} \in S$ is the initial state;
- $P\colon S \times S \to [0, 1]$ is the transition probability matrix where $\Sigma_{s' \in S}\mathbf{P}(s, s') = 1$ for all $s \in S$;
- $L\colon S \to 2^{AP}$ is a labelling function which assigns to each state $s \in S$ the set $L(s)$ of atomic propositions that are valid in the state.

To model reliability we use an absorbing DTMC with two final states C and F, that represent the correct (s_C) and the failure (s_F) outcome, respectively.

Each state s_i represents a component of the software architecture and one of those represents the initial state $\overline{s}$. The transition probability from state s_i to s_j is represented by $P(i,j)$. To accommodate with reliability, we modified the original transition probability between states to be calculated by $P(i,j) = R_i \cdot T_{i,j}$, representing the probability that state s_i executes correctly and the control is transferred to the component represented by the state s_j. More specifically, $T_{i,j}$ represents a directed branch denoting the possible transfer of control from state s_i to s_j. R_i denotes the reliability of the state s_i. Since, we assume that every component can fail, each state s_i has a direct edge to the absorbing failure state F denoted by $P_{i,F}$. The transition probability to the F state is given by $P(i,F) = (1 - R_i)$ which represents the occurrence of an error in the execution of the component represented by the state s_i. Let the transition matrix be P where $P(i,j)$ represents the probability of transition from state s_i to state s_j in the Markov process.

System reliability is expressed through a reachability property ($true\, U\, state = s_C$) using Probabilistic Computation Tree Logic (PCTL) [28]. This PCTL rule allows to determine system reliability by computing the probability that from an initial initial state $\overline{s}$ reach the absorbing successful state s_C.

This approach requires the specification of the reliability for each state and the usage of the system to define the transitions between states (also known as usage profile or operational profile).

3.4. Translation process

An automated process to predict reliability from a software architecture requires a translation from the architectural model to the corresponding mathematical formalism. To this end, we propose a formal translation process in which the architectural model (A) is related to the generation of the Discrete-Time Markov Chain (M). This relation is specified in Z by linking each member of A to exactly one member of M: $A \rightarrow M$. Specifically we define a map-

ping from each architectural element of an Architectural Description Language (ADL) (*e.g.*, components, connectors, properties and their relations) to the according states of the Deterministic-Time Markov Chain (DTMC). This mapping aims to achieve an automated generation of a DTMC from an ADL.

Following this, we describe the translation process along with its required properties.

3.4.1. Initial state

The stochastic model requires the specification of the initial state where the control flow begins. This involves annotating the architectural model with a mandatory property called 'EntryPoint' which states the starting component $c_i \in C$ and will be mapped as the initial state in the DTMC.

$$Prop = (EntryPoint, String, Name(c_i)) \mapsto \overline{s}$$

3.4.2. Components

The translation of each component is performed in accordance with the following guidelines:

- Each component maps to a state

$$c_i \in C \mapsto s_i \in S$$

- The reliability of a component (R_c) is defined as the probability that component c_i will carry out a task successfully with no failures. A non-failed task occurs when the component processes data received by an input port and sends a response over the output port.

$$Rc(c_i) = Pr\{c_i \in C\,;\, ip_j \in IP(c_i)\,;\, op_k \in OP(c_i) :\ op_k \text{ produces an output} \mid ip_j \text{ received an input}\} \in [0,1]$$

  Rc is expressed in the architectural model through a component property, by mapping to the probability of successfully transiting (P_R) from state

s_i to s_{i+1} and also to the probability of transition to a failure absorbing state s_F:

$$Prop = (reliability, float, Rc) \mapsto P_R(s_i, s_{i+1}) = Rc \qquad \wedge$$
$$Prop = (reliability, float, Rc) \mapsto P(s_i, s_F) = 1 - Rc$$

- When specified a representation expresses the internal behavior of a component by presenting a new sub-system:

$$Rep = \big(C',\ Con',\ Att',\ Prop'\big) \mapsto s_i = (S', \overline{s}', P', L')$$

3.4.3. Connectors

Each connector $con_i \in Con$ is responsible for the communication between different components and it is translated as follows:

- The probability of transiting between two components is specified as an attachment that sends data knowing that another attachment has already received data from a component.

$$T(con_i) = Pr\{att, att' \in Att\,;\, c, c' \in C : att' \text{ communicates data to } c' \mid att \text{ receives data from } c\ \} \in [0, 1]$$

The transition probability (P_T) is expressed through the property:

$$Prop = (transitionP, float, T) \mapsto P_T(s_i, s_{i+1}) = T$$

3.4.4. Constraints

Prior to the generation of the DTMC, we validate the software architecture according to a set of constraints that it must comply to be considered correct. These constraints dictate if a correct DTMC can be generated and are following specified in the Z notation:

No dangling ports or roles. Every input or output port of a component must be attached to a connector role and the same applies conversely. The system is not valid if there are any dangling ports or roles.

$$\forall\, ip \in IP(c) \mid \exists\, c \in C; r \in R(con); con \in Con \bullet ((c, ip), (con, r)) \in Att$$

$$\wedge \quad \forall\, op \in OP(c) \mid \exists\, c \in C; r \in R(con); con \in Con \bullet ((c, op), (con, r)) \in Att$$

$$\wedge \quad \forall\, r \in R(con) \mid \exists\, con \in Con; c \in C; ip \in IP(c); op \in OP(c) \bullet ((c, ip \vee op), (con, r)) \in Att$$

Output transitions sum to 1.0. For every output port, there exists a set with at least one connection to a target component. For all the elements within that set, their transition probabilities must add up to 1.

$$\forall\, op \in OP \mid \exists\, con_i \in Con; T \in Props\,(con_i) \bullet \sum_i T_{con_i} = 1.0$$

This section describes the mapping from an architectural description to a mathematical model. We specified this mapping formally to allow an unambiguous translation from an ADL to a DTMC and support its future extension to other quality attributes.

3.5. Automated prediction

As a result of the shortcomings highlighted by current surveys and limitations presented in Section 2, our approach addresses the quantitative prediction of reliability from a software architecture in an automated fashion. For this purpose, we formally annotate the translation procedure by making it possible to parse and analyze an ADL that complies with the ISO/IEC/IEEE 42010 Standard [17]. Furthermore, we test the effectiveness of our approach by applying it to the Acme ADL [29]. The reason for this choice rather than other ADLs is due to the fact that Acme is a general purpose language and not domain-specific like the others [30]. In addition, the development team of Acme, the Software Engineering Institute (SEI) in Carnegie Mellon University (CMU), have set up a software library, AcmeLib, that allows Acme models to be manipulated by third-party applications.

After parsing the architectural models, we generated a Discrete-Time Markov Chain (DTMC) in the Prism language [16]. The generated formalism is then resolved by the Prism Model Checker tool [31] which provides a quantitative result for the system reliability. Although other probabilistic model checking tools could be applied, we selected Prism since it is a free, open-source tool with a vast documentation and support for discrete-time Markov chains.

Although in this work we applied our approach to reliability, other non-functional attributes could also be subject of study with no significant changes in the translation process, such as performance, as shown in the study of Galloti *et al.* [15].

4. Hypotheses under test

Self-adaptive systems are deployed in highly dynamic and unpredictable environments. They are able to autonomously decide and adapt to meet functional or non-functional goals. However, these self-adaptive systems are configured to adapt according to specific system and environment conditions. Such configurations are performed manually by human operators and remain static throughout the life-cycle of its operation. This means that the system will behave exactly the same while experiencing the same conditions. Since self-adaptive systems are becoming more complex and being applied in diverse contexts, manually configuring such conditions by a human operator is error-prone and may lead the system to unexpected states. In these states and due to the static configurations, the self-adaptation process may fail to select the best strategy and decide for an adaptation that causes failures, performance issues or decreasing the quality of the system. With this in mind, our work aims to confirm the following hypotheses:

1st hypothesis Self-adaptive planning algorithms based on static configurations have a limited ability to select the best adaptation strategy upon unanticipated and untested conditions.

2nd hypothesis Applying quantitative prediction or verification methods at runtime improves the ability to reach quality goals under unanticipated conditions, while maintaining similar ability under known conditions.

3rd hypothesis Runtime modeling and prediction of quality attributes can be sufficiently efficient to guarantee the performance of a self-adaptive system.

The first hypothesis reflects the limitations of traditional self-adaptive approaches. Self-adaptive operators trigger adaptations based on human configured values that take into consideration multiple quality objectives, adaptation strategies and system and environment conditions. These operators are manually configured, resulting in an unfeasible task when taking into account every possibility, namely due to the exponential growth of combinations for each added strategy or quality objective. As a result, a self-adaptive system may select a non-optimal adaptation strategy in unanticipated or untested conditions, leading to a degradation of the desired quality goals.

The second hypothesis states that our approach solves the identified limitations of traditional self-adaptive planning algorithms. This approach consists in predicting the impact on quality goals for each adaptation strategy and supporting the planning phase through dynamic update of the human configured values.

The third hypothesis attests that runtime modeling and prediction of quality attributes should not affect the performance of the overall system or the achievement of desired adaptation goals.

5. Case-study

To test the effectiveness of our approach and confirm the hypotheses presented in Section 4, we adopted a "de facto" standard case-study from the self-adaptive community, the Znn.com [1]. This section outlines the experimental setup along with an example of our approach by generating stochastic models for each adaptation strategy.

5.1. Adopted self-adaptive system

A system is considered to be self-adaptive when it modifies its own behavior in response to changes in its operating environment [32]. Weyns *et al.*[33] presented FORMS as a formal reference model for specifying self-adaptive software systems. Their work uses the Z notation to formally specify three self-adaptive approaches that have influenced today systems: computational reflection, distributed computation and MAPE-K. Our work targets the MAPE-K approach, since it was built with a separation of the adaptive phases in mind.

On the basis of the study carried out by Villegas *et al.* [34], we chose the Rainbow self-adaptive system [35, 36, 37] which supports quality-driven goals and quantitative metrics. Moreover, Rainbow is based on the MAPE-K approach from IBM Autonomic Computing Initiative [6] and makes use of the Acme as the basis Architecture Description Language (ADL) as does our proposed approach outlined in Section 3. Rainbow is an architecture-based self-adaptive system designed by the Carnegie Mellon University and its framework is depicted in Figure 2 which shows that it consists of two main subsystems: the controller and target.

The controller monitors the target system through probes and gauges which update properties in the architectural model managed by the Model Manager. The Architecture Evaluator evaluates the model to determine if the system is operating within an acceptable range of quality goals. If the evaluation finds that the system is not operating under normal conditions, it invokes the Adaptation Manager which is responsible for selecting a more suitable adaptation strategy. Each strategy involves a bundle of simple courses of action denoted as adaptation tactics. After a strategy has been selected, the Strategy Executor is responsible for applying a sequence of actions to the target system, so that the selected strategy is instantiated in the system.

The target system is defined as the resource that will be monitored and adapted to meet the self-adaptation goals. The environment consists of the external world that interacts with the target system. It is considered to be non-controllable and at the same time, capable of influencing the runtime properties

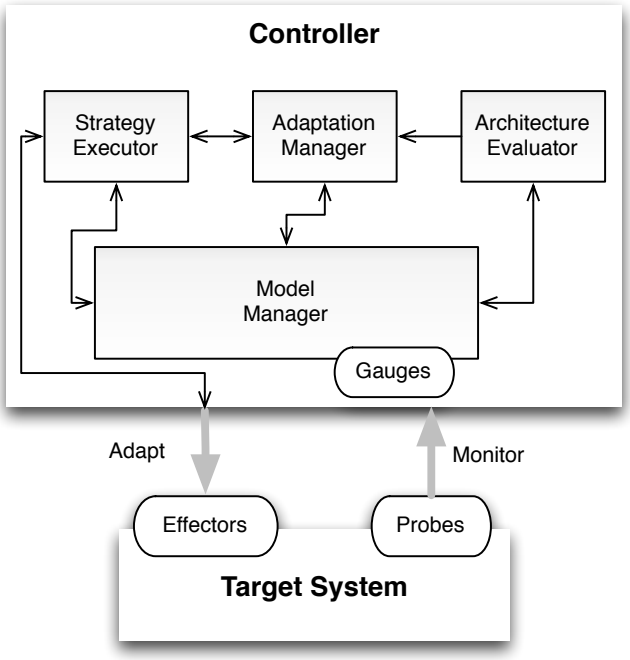


Figure 2: Rainbow framework

(*e.g.*, hardware, physical context or network).

We applied our approach discussed in Section 3 to the above presented self-adaptive system, Rainbow. They are both based on the MAPE-K and so there is a direct mapping from the components of Rainbow depicted in Figure 2 to the ones of our approach presented in Figure 1. In detail, the Probes and Gauges in Figure 2 collect runtime information which is mapped to the Monitor phase in Figure 1. The Architecture Evaluator decides whether an adaptation is required being mapped to the Analyze phase. The Adaptation Manager is responsible for planning a course of action being related to the Planning phase while the Strategy Executor maps to the Execute phase in our approach. The shared component in Rainbow, Model Manager, holds the information about the runtime architectural model and is mapped to the Knowledge phase in the MAPE-K.

We integrate our approach within Rainbow to support the Adaptation Manager. In detail, our approach does not replace the Adaptation Manager, nor the algorithm that plan the course of action. Instead, our approach is connected to Rainbow by updating the values used for planning with the reliability predictions from our approach. Regarding applying it to other self-adaptive systems, the target system just needs to be based on the MAPE-K and since, no replacement is needed, the integration becomes easier and effortless. In the following section we discuss the target-system used in our case-study.

5.2. Target system

We defined the target system as being the Znn.com, a typical infrastructure for a news website and its diagram is depicted in Figure 3.

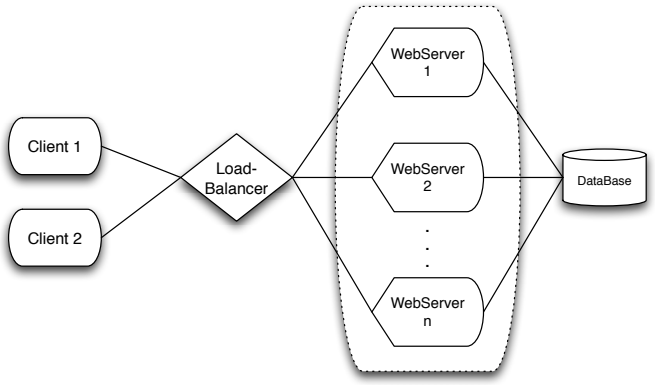


Figure 3: Znn.com diagram

It has a tiered architecture with a set of web-servers that serve content, both textual and graphical, from back-end databases to clients through a front-end presentation logic. In addition, it uses a load-balancer to reroute the requests from the client to a pool of servers. The number of active servers will depend on the selected adaptations required to fulfill the system goals. It must be stressed that Znn.com is an actual platform running in actual servers using a standard Apache distribution. Znn.com is not a simulation model.

5.3. Experimental design

The concepts used throughout this case-study are discussed in Table 1 which also serve as a quick definition reference for the reader.

Table 1: Adaptation Concepts

Tactic	Is a set of actions triggered in the target system. It can be a bash script or an executable that causes effects and changes to the target system. (*e.g.*, reboot a server, kill an application).	
Strategy	Is a pattern of adaptation that includes a condition, an action and a delay. In detail, when a condition is satisfied a tactic is triggered in the target system, which after a delay an effect should be observable.	
Quality Goals	Business qualities of concern (*e.g.*, availability, cost)	
Adaptation Metric	A runtime property that is collected for each time-frame and it is related to a long-term quality goal (*e.g.*, reliability is collected and predicted at the planning phase to obtain a higher availability).	
Adaptation Operator	Quantifies the impact of a strategy on each of the quality dimensions to determine the merits of one strategy over another. These operators include the following measures:	
	Utility Function	A function that determines the quality impact on adaptation metrics (*i.e.*, we aim to achieve 100% of reliability which results in 1.0 of utility. For lower reliability values, the utility declines, as shown by Table 2);
	Utility Preferences	Business preferences over quality goals (*i.e.*, Availability has a utility preference of 50%, as shown in Table 3);
	Impact Vectors	Cost-benefit attributes considered in strategy choice with regard to quality goals (*e.g.*, the strategy 'Enlist a Server' improves the overall availability by 10%).

Self-adaptive systems are designed with a set of operators to support decision-making and drive adaptations to meet desired quality goals. These operators are usually statically defined by a human operator and include utility functions,

preferences and impact vectors. In this section we specify the adaptation goals, metrics, strategies and operators of the proposed Znn-like system.

5.3.1. Adaptation goals

As with a typical news provider, Znn.com focuses on providing news content to its customers in a reliable way while keeping the operating costs to a minimum. In short, we identify three quality objectives for self-adaptation:

- Availability – this expresses the probability that the system is operating properly when it is requested for use. In specific terms, it is a long-run measure that takes into consideration the reliability of each time-frame and the 'repair actions' as adaptations executed in the target system. The goal of this quality attribute is to maximize its potential, even if it has to incur a higher operational cost;

- Operational Cost – this measures the number of computational resources that need to be available during the experiment. Each server in the pool of servers is deployed through a Virtual Machine (VM) and the goal is to reduce the number of VMs to a minimum. For example, the system switches off virtual machines when processing a low number of requests;

- Utilization – this defines the amount of work received by the system in terms of the maximum load that is supported by all the available servers. For example, if the current work that is being processed reaches the maximum capacity that all the servers are able to process, the system may have to adapt by enlisting a new server to increase the total load that can be handled.

5.3.2. Adaptation metrics

We monitor the target system to collect data for each ten second timeframe — the period between adaptations — about the following runtime properties:

- Active resources: the number of servers that are active and responding to requests;

- Reliability: defined as *"functioning correctly"* [38] and in this case-study it refers to the number of non-failed requests between recovery actions (*i.e.*, adaptations). Hence, we define the failure behavior as a request that takes an unreasonable time to receive a response (*i.e.*, $> 2000\ ms$) or when the returning code is not successful (*i.e.*, HTTP status code $\neq 200$). To compute reliability we need to collect at runtime the response time of each request and their HTTP response code;

- Load: indicates the number of requests that have been responded to within the time-frame over the total maximum capacity the system can hold. To compute Load we need to collect the throughput of the system and keep a record of the maximum requests that each server is able to respond.

5.3.3. Adaptation strategies

As stated by Table 1, an adaptation strategy is a set of tactics that will trigger a collection of actions. These actions aim for changing the behavior of the target system to achieve desired quality goals. A strategy includes a condition that must be satisfied to be selected and a delay to wait for the strategy to be successfully accomplished.

To trigger a strategy, the system collects the runtime metrics for a ten second time-frame and then, during the planning phase determines which strategy to choose or continues without adapting.

The selected adaptations of this particular case-study focus on the server pool. Hence, depending on the current state of the runtime properties of the system, the controller may select one of the following strategies:

- Enlist server – Enables a server, if there is a spare one ready to be activated;

- Discharge the slowest server – If there are at least two active servers and no failure has occurred, our approach will discharge the slowest one (*i.e.*, the one with the highest value of mean response time);

- Discharge the least reliable server – If there are at least two active servers and at a failure has occurred, the system will discharge the less reliable (*i.e.*, the server with the highest failure rate).

5.3.4. Static adaptation operators

The self-adaptive solution used in this study, Rainbow, uses utility-theory as its decision-making algorithm. This type of algorithm relies on utility functions, preferences and impact vectors to ensure that adaptations fulfill the defined quality goals. We following detail the adaptation operators:

Utility functions. The utility-theory measures the monitored system properties according to a utility function. It provides a score which reflects how properties are behaving when seeking to achieve the proposed goals. These values and functions are defined in the design phase of the self-adaptive system. An illustrative example is given in Table 2. The values that fall in intermediate points are linearly extrapolated.

From Table 2 it can be inferred that the utility is higher for reliability values close to 100% and lower values are heavily punished because of their utility function. In addition, the defined values lead to a low consumption of computational resources by designating one active web-server as the best utility outcome of the system. With regard to the experienced load, the system favors a low utilization capacity.

Utility preferences. The preferences define the relative importance of the quality dimensions and serve as an example to prioritize quality attributes. They are shown in Table 3 and it can be noticed that availability is twice as important as the cost or utilization of the system.

Table 2: Utility Functions

Reliability		Active Resources		Load	
Value (%)	Utility	Value	Utility	Value (%)	Utility
100	1.0	0	0.0	100	0.0
99	0.88	1	1.0	90	0.05
95	0.54	2	0.85	80	0.1
90	0.3	3	0.55	70	0.4
85	0.16	4	0.3	60	0.5
80	0.09			50	0.6
75	0.05			40	0.7
50	0.002			30	0.8
0	0			20	0.9
				10	0.95

Although they may not be achieved optimally, these preferences can be used to solve resource constraints or trade-offs between certain quality attributes. An example of a resource constraint is that on a server discharge, the remaining servers may not be able to process the current demand of requests.

Impact Vectors. The impact of a strategy on each of the quality dimensions is represented as a vector of cost-benefit values between the strategy and each quality dimension. Table 4 shows the adopted values in our case-study and it

Table 3: Utility preferences

	Percentage
Availability	50%
Cost	25%
Utilization	25%
Total	100%

should be noted that our *Enlist Server* strategy increases both availability and the used computational resources, at the same time that it reduces the system utilization, since it has more machines to process the same demand of requests. Conversely, both *Discharging Server* strategies will reduce costs and increase the use of the system. When deciding to discharge the least reliable server, the system will increase availability, but it will be kept the same when discharging the slowest server from the pool.

Table 4: Static impacts on the quality dimensions for each strategy

	Availability	Cost	Utilization
Enlist a Server	+10%	+1.0	-20%
Discharge the Least Reliable Server	+1%	-1.0	+20%
Discharge Slowest Server	0.0%	-1.0	+20%

Utility rate calculations. Adaptation strategies are ranked through the utility outcome of the following formula:

$$
\begin{aligned}
\mathrm{U}_{Rel} &= \text{Utility(reliability } + \text{Impact } (e.g., +10\%)) \\
\mathrm{U}_{AR} &= \text{Utility(active resources } + \text{Impact } (e.g., +1)) \\
\mathrm{U}_{Load} &= \text{Utility(load } + \text{Impact } (e.g., +20\%)) \\
\text{Utility} &= \mathrm{U}_{Rel} \times \text{Availability Preference } (50\%) \\
&+ \mathrm{U}_{AR} \times \text{Cost Preference } (25\%) \\
&+ \mathrm{U}_{Load} \times \text{Utilization Preference } (25\%)
\end{aligned}
$$

To calculate the utility for a strategy, we firstly assess the impact of each adaptation metric (U_{Rel}, U_{AR} and U_{Load}). Then, we calculate the overall utility obtained from the adaptation strategy by averaging the utilities obtained from each metric with the adaptation preferences. The controller performs this assessment to plan an adaptation and chooses the strategy that has the highest utility value.

5.4. Example

To show the effectiveness of applying quantitative verification methods at runtime, we provide a demonstrative example detailing the generation of the stochastic model and utility calculations. Figure 4 illustrates the architectural model of the Znn.com taken from a snapshot of an actual run of the system.

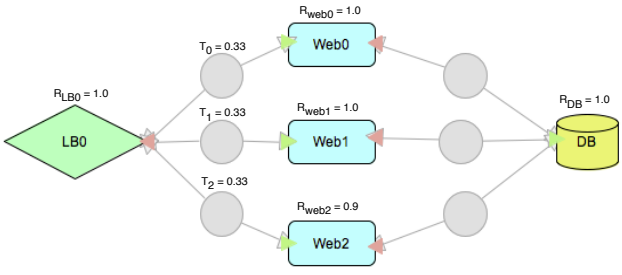


Figure 4: Demonstration Architectural Model

In this time-frame, the collected metrics from the running system are as follows:

- Reliability: 95.7% $\rightarrow$ $\mathrm{U}_{Rel} = 0.5995 \approx 0.6$;
- Active Resources: 3 active web-servers $\rightarrow$ $\mathrm{U}_{AR} = 0.55$;
- Load: 40% $\rightarrow$ $\mathrm{U}_{Load} = 0.7$.

In this time-frame, the system depicts three active web-servers one of which is failing to respond to client requests (*i.e.*, Web2). In these circumstances, the system determines whether an adaptation is required or not. To this end, the system calculates an utility value for the current system (non-adaption) and for each one of the eligible adaptation strategies: 'enlist server' and 'discharge the least reliable' (the strategy 'discharge the slowest server' is not eligible due to the occurrence of failures which does not satisfy the adaptation conditions presented in Section 5.3.3).

To cope with failure and conduct the system to achieve the desired goals, the planing phase is responsible to select an adaptation strategy or continue with the current system by non-adapting. A decision is determined by assessing the utility outcome from the eligible strategies.

To show an example of the application of our approach to the planning phase of the self-adaptive system, we following discuss the generation of the stochastic models and how static adaptation operators are updated at runtime. Then, we show a comparison between our approach and traditional self-adaptive planning algorithms based on static adaptation operators.

5.4.1. Our Approach

Our approach predicts the behavior of each strategy by applying its changing operators to the architectural model. From the running example shown in Figure 4, we discuss each one of the steps of our approach outlined in Figure 1, as follows:

1st step We use the collected runtime properties to update the Software Architecture, as shown in Figure 4.

2nd step We make a copy of the software architecture for each eligible strategy and apply its changes. As previously discussed, at this point there are two eligible strategies: 'Enlist Server' and 'Discharge the Least Reliable'. Figure 5 depicts the architecture with the changes performed by the 'Enlist Server' strategy. This strategy adds a new server and reroutes the requests from the Load-Balancer as can be observed by Figure 5.

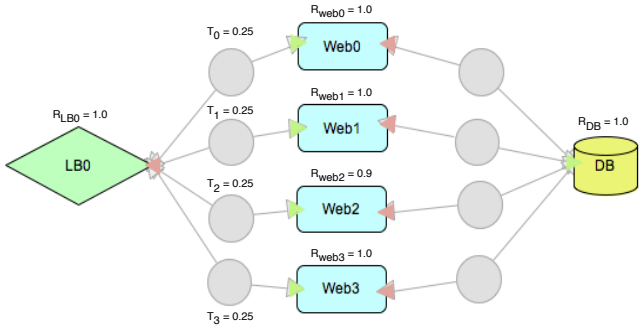


Figure 5: Architectural changes for the Enlist Server strategy

Regarding the strategy of 'Discharging the least reliable', Figure 6 depicts the resulting software architecture after applying this strategy. As one may depict, the system removed the failing server from the system and reroute the requests from the Load-Balancer to the available servers.

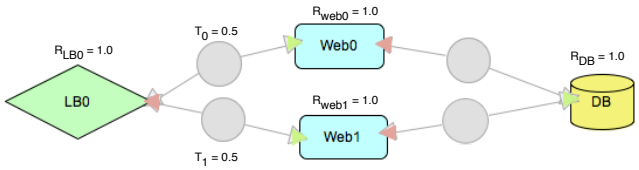


Figure 6: Architectural changes for the Discharge the Least Reliable strategy

3rd step Generates a DTMC to predict the reliability of each eligible strategy. The generation process uses the ADL of the software architecture along with the translation process, presented in Section 3.4. The resulting

DTMCs can be viewed as state-space models with an initial state $\overline{s} = s_1$ and two absorbing states: s_f and s_c representing the failed and correct behavior, respectively.

The generated DTMC from applying the strategy 'Enlist Server' can be viewed in Figure 7. In this DTMC one server was added and the requests have been rerouted to account with the modification of the server pool. The values used for generating the DTMC (discussed in Section 3) are outlined in Table 5.

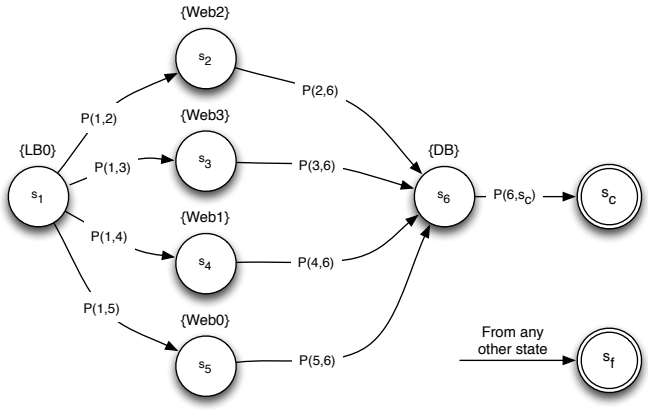


Figure 7: DTMC for the 'Enlist Server' strategy

Table 5: DTMC values for the 'Enlist Server' strategy

Reliability	Transition Probability
$R_1 = R_2 = R_3 = R_5 = R_6 = 1.0$	$T_{1,2} = T_{1,3} = T_{1,4} = T_{1,5} = 0.25$
$R_4 = 0.9$	$T_{2,6} = T_{3,6} = T_{4,6} = T_{5,6} = 1.0$

The transition probability from state s_i to state s_j is represented by $P(i, j)$ as specified in Section 3.3. To calculate this probability we employ the following formula: $P(i, j) = R_i \cdot T_{i,j}$, representing the probability that state s_i executes correctly and transfers the control to state s_j. Conversely, the transition to a failure state is determined by $P(i, f) = 1 - R_i$, represent-

ing the probability of state s_i failing to respond to a request. The Prism model generated by applying this strategy can be found in Appendix A.

Regarding 'Discharge the least Reliable' strategy, Figure 8 outlines the generated DTMC showing one less server than the current system depicted in Figure 4. The values used for the generation of the DTMC can be found in Table 6.

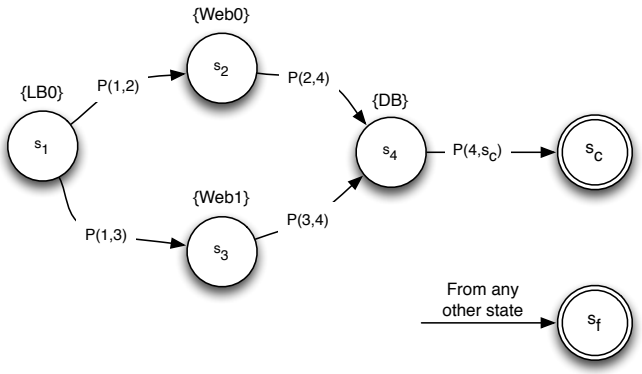


Figure 8: DTMC for the 'Discharge the Least Reliable Server' strategy

Table 6: DTMC values for the 'Discharge the Least Reliable Server' Strategy

Reliability	Transition Probability
$R_1 = R_2 = R_3 = R_4 = 1.0$	$T_{1,2} = T_{1,3} = 0.5$
	$T_{2,4} = T_{3,4} = 1.0$

The generated Prism model is outlined in Appendix B and following, we show the utility calculations for the eligible strategies.

$4^{\mathbf{th}}$ step The above outlined DTMCs can be solved through a reachability property as detailed in Section 3.3. In short, the reachability property determines the probability that from a given initial state reach the state s_c, representing the correct state. To automatically solve a DTMC we use the Prism Model Checker tool with the generated models shown in

Appendix A and Appendix B. Table 7 shows the results after solving each DTMC.

Table 7: Reliability Prediction for each DTMC

Strategy	Reliability Prediction
Enlist Server Strategy	97.5%
Discharge the Least Reliable Server	100%

5th step The last step of our approach consists on dynamically update the static adaptation operator values. In short, we use the above reliability predictions to rewrite the utility rate calculation formula presented in Section 5.3.4. The new formula avoids the use of impact vectors for reliability by defining its utility as: $U_{Rel} =$ Utility(predicted reliability).

Following we present the utility rates for each of the eligible adaptation strategies:

Enlist Server

$$
\begin{aligned}
U_{Rel} &= \text{Utility(predicted reliability)} = \text{Utility}(0.975) = 0.75 \\
U_{AR} &= \text{Utility(active resources } + \text{Impact)} = \text{Utility}(3+1) = 0.3 \\
U_{Load} &= \text{Utility(load } + \text{Impact)} = \text{Utility}(40\% - 20\%) = 0.9 \\
\text{Utility} &= U_{Rel} \times \text{Availability Preference (50\%)} \\
&\quad + U_{AR} \times \text{Cost Preference (25\%)} \\
&\quad + U_{Load} \times \text{Utilization Preference (25\%)} \\
&= 0.75 \times 50\% + 0.3 \times 25\% + 0.9 \times 25\% \\
&= 67.5\%
\end{aligned}
$$

Discharge the Least Reliable

$$
\begin{aligned}
\mathrm{U}_{Rel} &= \text{Utility(predicted reliability)} = \text{Utility}(1.0) = 1.0 \\
\mathrm{U}_{AR} &= \text{Utility(active resources } + \text{Impact)} = \text{Utility}(3-1) = 0.85 \\
\mathrm{U}_{Load} &= \text{Utility(load } + \text{Impact)} = \text{Utility}(40\% + 20\%) = 0.5 \\
\text{Utility} &= \mathrm{U}_{Rel} \times \text{Availability Preference (50\%)} \\
&\quad + \mathrm{U}_{AR} \times \text{Cost Preference (25\%)} \\
&\quad + \mathrm{U}_{Load} \times \text{Utilization Preference (25\%)} \\
&= 1.0 \times 50\% + 0.85 \times 25\% + 0.5 \times 25\% \\
&= 83.75\%
\end{aligned}
$$

No Adaptation

$$
\begin{aligned}
\mathrm{U}_{Rel} &= \text{Utility(reliability)} = \text{Utility}(95.7\%) = 0.6 \\
\mathrm{U}_{AR} &= \text{Utility(active resources)} = \text{Utility}(3) = 0.55 \\
\mathrm{U}_{Load} &= \text{Utility(load)} = \text{Utility}(40\%) = 0.7 \\
\text{Utility} &= \mathrm{U}_{Rel} \times \text{Availability Preference (50\%)} \\
&\quad + \mathrm{U}_{AR} \times \text{Cost Preference (25\%)} \\
&\quad + \mathrm{U}_{Load} \times \text{Utilization Preference (25\%)} \\
&= 0.6 \times 50\% + 0.55 \times 25\% + 0.7 \times 25\% \\
&= 61.25\%
\end{aligned}
$$

The above formulas show the process of assessing the utility for each strategy before an adaptation is triggered. The utility values for each adaptation metric are obtained through utility functions outlined in Table 2. The results of our approach will be discussed and compared at the end of this section, after outlining the utility assessment of traditional planning algorithms.

5.4.2. Traditional Self-Adaptive

We denote 'traditional self-adaptive' to approaches that rely on test cases specified at design-time (*e.g.*, impact vectors). These test cases usually remain

static and may include uncertainty in strategy selection due to the number of environment and system conditions [2, 3, 14, 4].

From the architecture outlined in Figure 4 representing the system in a particular time-frame, traditional self-adaptive systems calculate the utility of every eligible strategy, as follows:

Enlist Server

$$
\begin{aligned}
\mathrm{U}_{Rel} &= \text{Utility(reliability } + \text{Impact)} = \text{Utility}(95.7\% + 10\%) = \\
&= \text{Utility}(100\%) = 1.0 \\
\mathrm{U}_{AR} &= \text{Utility(active resources } + \text{Impact)} = \text{Utility}(3 + 1) = 0.3 \\
\mathrm{U}_{Load} &= \text{Utility(load } + \text{Impact)} = \text{Utility}(40\% - 20\%) = 0.9 \\
\text{Utility} &= \mathrm{U}_{Rel} \times \text{Availability Preference } (50\%) \\
&\quad + \mathrm{U}_{AR} \times \text{Cost Preference } (25\%) \\
&\quad + \mathrm{U}_{Load} \times \text{Utilization Preference } (25\%) \\
&= 1.0 \times 50\% + 0.3 \times 25\% + 0.9 \times 25\% \\
&= 80.0\%
\end{aligned}
$$

Discharge the Least Reliable

$$
\begin{aligned}
\mathrm{U}_{Rel} &= \text{Utility(reliability } + \text{Impact)} = \text{Utility}(95.7\% + 1\%) = \\
&= \text{Utility}(96.7\%) \approx 0.68 \\
\mathrm{U}_{AR} &= \text{Utility(active resources } + \text{Impact)} = \text{Utility}(3 - 1) = 0.85 \\
\mathrm{U}_{Load} &= \text{Utility(load } + \text{Impact)} = \text{Utility}(40\% + 20\%) = 0.5 \\
\text{Utility} &= \mathrm{U}_{Rel} \times \text{Availability Preference } (50\%) \\
&\quad + \mathrm{U}_{AR} \times \text{Cost Preference } (25\%) \\
&\quad + \mathrm{U}_{Load} \times \text{Utilization Preference } (25\%) \\
&= 0.68 \times 50\% + 0.85 \times 25\% + 0.5 \times 25\% \\
&= 67.75\%
\end{aligned}
$$

No adaptation

$$
\begin{aligned}
\mathrm{U}_{Rel} &= \text{Utility(reliability)} = \text{Utility}(95.7\%) = 0.6 \\
\mathrm{U}_{AR} &= \text{Utility(active resources)} = \text{Utility}(3) = 0.55 \\
\mathrm{U}_{Load} &= \text{Utility(load)} = \text{Utility}(40\%) = 0.7 \\
\text{Utility} &= \mathrm{U}_{Rel} \times \text{Availability Preference (50\%)} \\
&\quad + \mathrm{U}_{AR} \times \text{Cost Preference (25\%)} \\
&\quad + \mathrm{U}_{Load} \times \text{Utilization Preference (25\%)} \\
&= 0.6 \times 50\% + 0.55 \times 25\% + 0.7 \times 25\% \\
&= 61.25\%
\end{aligned}
$$

5.4.3. Discussion of results

To compare the results of our approach and traditional self-adaptive ones we summarize the utility outcomes in Table 8.

Table 8: Utility results from adaptation strategies

	Enlist Server	Discharge Least	No Adaptation Reliable
Our Approach	67.5%	83.75%	61.25%
Traditional Self-Adaptive	80.0%	67.75%	61.25%

The obtained results from the example of Figure 4 show that when a server is failing, the traditional self-adaptive approach based on static impact vectors, chooses to enlist a new server rather than discharging the failed one. This measure would add a new web-server, but not correct the failure.

On the other hand, our method performs reliability prediction to anticipate the impact of each strategy. This automated method allows architects to abstract from defining static vectors where it is hard to achieve a precise impact and also to go beyond the static approach in unexpected or untested conditions,

such as the above demonstration suggests.

One may argue that static impact vectors could be tuned or rearranged to account for this situation; however, another untested or unexpected condition may arise from this. These conditions are caused by the large state space of possibilities, which in this example one should account with three adaptation strategies, three runtime metrics and their utility values as well as preferences and impact vectors. Furthermore, our method relies on quantitative verification of reliability at runtime to predict the impact vectors and reduce the uncertainty at design time of estimating them. In conclusion, our approach makes better decisions by avoiding the constant impact vectors required in the traditional self-adaptive approach.

5.5. Workload

We tested our approach with a realistic workload which can trigger different adaptations. More precisely, our workload is based on an Internet phenomenon, known as *Slashdot effect* or *flash crowd*. This phenomenon is characterized by a low-traffic website which may suddenly be inundated by visitors for a period of time due to, for example, a dramatic news announcement or alternatively, it may be redirected from a highly-visited website.

Our workload is depicted in Figure 9 and was patterned after the collection of realistic traffic from the event. The collected data of the event lasted twenty-four hours which we scaled down to one hour, keeping a similarity to the 'visit traffic pattern' as follows:

1. 1 minute of low activity;
2. 5 minutes of sharp rise in incoming traffic;
3. 18 minutes of high peak requests;
4. 36 minutes of a linear decline in requests, also known as the ramp-down period.

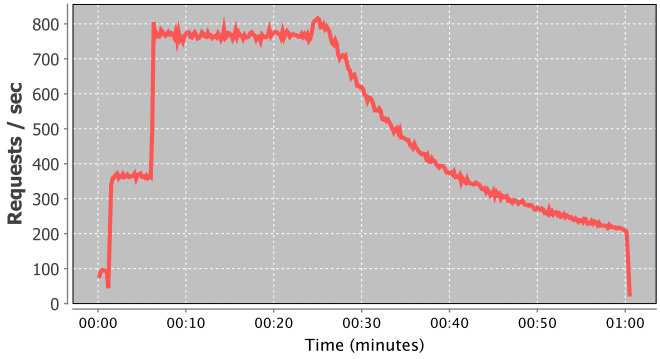


Figure 9: Request load of the *Slashdot effect*

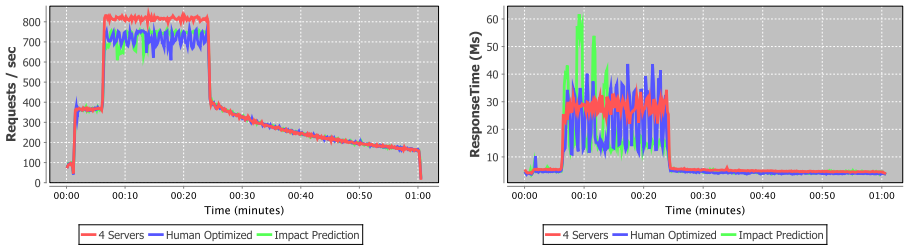
6. Evaluation

For validation purposes, we compared three different approaches: no-adaptation, traditional decision-making and our dynamic update of adaptation weights. In more specific terms, the first approach entails a non-adaptive solution in which all the four servers are active and ready to respond to client requests (*i.e.*, 4 Servers). The second approach consists of human configured values (*i.e.*, Human Optimized) to drive adaptations and shows the traditional decision-making algorithm. The third and last approach under comparison, includes the approach proposed in this paper and uses the runtime prediction of reliability to estimate the impact of each possible adaptation strategy (*i.e.*, Impact Prediction).

The experimental procedure consists of two testing scenarios: Control run and Fault-injection. In the former, we compare the approaches under normal conditions while, in the latter we inject faults to trigger adaptations.

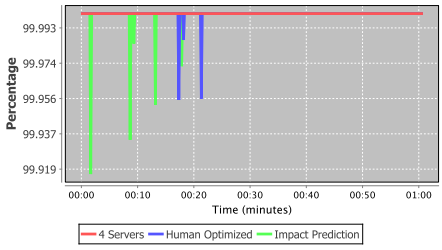
6.1. Control run

In this test, the system is in normal conditions without any injected fault or crash to ensure that both self-adaptive methods achieve their quality goals. The results are depicted in Figure 10 and as can be seen, there is a comparison between non-adaptive (4 servers), traditional self-adaptive (human optimized) and our self-adaptive proposal (impact prediction).

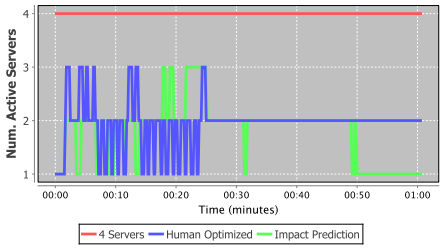


(a) Throughput

(b) Response Time



(c) Reliability



(d) Active Resources

Figure 10: Graph results for the Control Run

Figure 10(a) shows the throughput in number of processed requests (successful and unsuccessful requests) during each 10 second time-frame. As can be seen, the results show consistency between the tested scenarios and there is only a difference between them during the high-peak period of requests, when the scenario with more active servers will respond to more requests. Table 9 supports this claim by including a higher number of processed requests for the scenario with no adaptation and four active servers.

Figure 10(b) shows the response time for each scenario in milliseconds with a granularity of ten seconds. An average response time is calculated for each ten second time-frame. An increase in the response time occurs during the high-peak period of requests, and returns to normal values in the ramp-down. Moreover, all the graph series have similar results, although between 6 and 24

Table 9: Control Run results

	N. Requests	**Availability** (%)	**VMs Hour**
4 Servers	1532989	100.0	4.0
Human optimized	1425479	99.999	1.9
Impact prediction	1437780	99.999	1.8

minutes (the high peak period) both self-adaptive approaches have unstable results. This instability is due to the enlisting and discharging servers, although the results still remain within the threshold of acceptable response times (*i.e.*, below 2000 milliseconds).

Figure 10(c) shows the reliability through the rate of successful requests for each time-frame. The non-adaptive approach has a constant 100% of reliability throughout the test while the self-adaptive ones show some low peaks. These low peaks represent a very low number of failures (7 in the *human optimized* and 13 in the *impact prediction*) and are due to lost requests between switching servers on and off. Table 9 supports these statements by providing the long-run availability values and it can be seen that both self-adaptive approaches have five nines, while the non-adaptive does not register any failure.

The number of active servers during the experimentation is shown in Figure 10(d). The non-adaptive approach has a constant number of active servers, although self-adaptive ones have variations especially when there is a high demand for requests.

In conclusion, Figure 10 and Table 9 show that both self-adaptive approaches achieve similar results when compared with the most expensive non-adaptive solution. During this test, there are no significant advantages in applying our method for predicting the impact for each strategy and this was hardly the purpose of this experimentation. The set test is designed to show that both self-adaptive solutions achieve their adaptation goals, and result in similar and

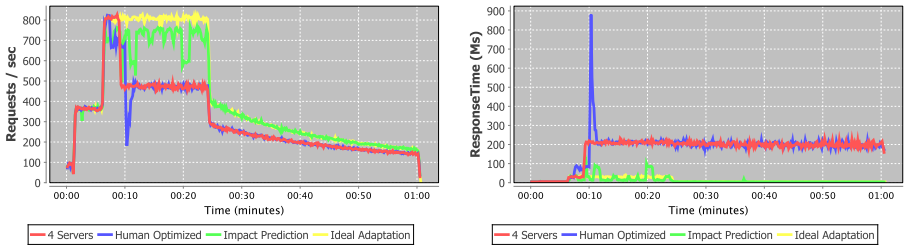
comparable results to the most expensive and available non-adaptive solution. However, the results for both the self-adaptive approaches are similar and thus it can be concluded that runtime modeling and prediction of a quality attribute do not have an adverse effect on the performance of the system or the achievement of quality goals.

6.2. Fault injection

Human operators are often considered the weak link and the proportion of errors that can be attributed to people ranges from 0.1% to 30%, depending on whether the operator is handling simple routine operations or undergoing a high level of stress [39]. In view of this, we set up an experiment that injects a fault in a PHP file that corresponds to a mistake introduced by the developer. The fault consists of a delay introduced in each request that leads to service degradation by increasing the time each request is resolved by between 1.5 and 2.5 seconds, following an uniform distribution. The rationale behind the specified values is that a request is regarded as unsuccessful if it takes more than two seconds to be resolved. Thus, the introduced delay allow some requests to be resolved within the time regarded as successful and others to exceed the reasonable amount of time leading to failures ($> 2000ms$, defined in Section 5.3.2).

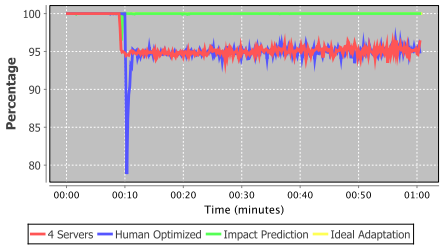
The injected fault only affects one server and is introduced 10 minutes after the start of the experiment. Figure 11 shows the results for this experiment in which we compared the non-adaptive (4 servers), both self-adaptive approaches and a run that we consider to be the *ideal adaptation*. This *ideal adaptation* assumes that the system knows when and where the failure will occur, so it proceeds by disabling the failing server and enlisting a spare one. This 'ideal' adaptation is considered to be unrealistic in the real world, since it assumes knowing *a priori* when and where to apply a recovery action. The goal of this test is to keep a record of the best possible adaptations and identify how close other adaptation methods get to this ideal adaptation. Table 10 includes a complete list of the performed tests.

The throughput results are given in Figure 11(a) and show an abrupt fall

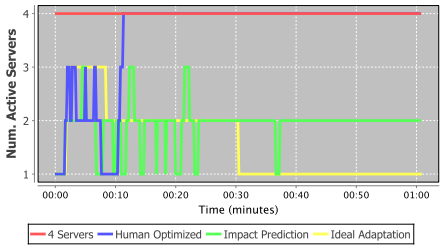


(a) Throughput

(b) Response Time



(c) Reliability



(d) Active Resources

Figure 11: Graph results for the Fault Injection experiment

Table 10: Fault injection results

	N. Requests	**Availability (%)**	**VMs Hour**
4 Servers	1110584	95.752	4.0
Human optimized	1096751	95.682	3.6
Impact prediction	1417786	99.981	1.9
Ideal adaptation	1523411	99.999	1.6

in the number of processed requests at 10 minutes due to the introduction of a delay. It can be seen that the non-adaptive (4 Servers) approach cannot recover from the 'failing behavior', and leads to an increase of response time and a decline in reliability as shown in Figures 11(b) and 11(c), respectively.

In both *4 servers* and in *human optimized* approaches, the number of processed requests shown in Figure 11(a) falls sharply. The reason for this is that the load-balancing policy distributes the same amount of work among the various active servers. To keep equality among the servers, the load-balancer waits for all the responses before distributing a new set of requests. For this reason, if the failing server remains in the pool, the load-balancer has to wait for its response which causes a delay and, thus, reduces the number of processed requests.

With regard to the *human optimized* run, Figures 11(a) and 11(c) show an abrupt fall in the number of processed requests as well as in reliability. Figure 11(d) illustrates its adaptation process which consists of increasing the number of resources to cope with the introduced delay. However, since it just enables more servers and keeps the failing server active, the reliability and performance will always be affected, since the failing server has to respond to requests and the others will have to wait for it.

On the other hand, in the *impact prediction* there is clearly also a fall in reliability and throughput. However, by predicting the impact of each adaptation strategy, it first decides to discharge the failing server and then adds more resources to cope with the demand for requests, as seen in Figure 11(d) and demonstrated by the utility calculations set out in Section 5.4.1. As a result, our method quickly triggers proper adaptations to cope with this kind of erratic behavior, by providing a high number of processed requests and a low number of failures throughout the rest of the experiment.

These results confirm that unexpected or untested conditions may have a negative effect on the achievement of quality goals when using constant weights to trigger adaptations. Table 10 suggests that the *impact prediction* has good overall results with lower cost and better availability than the other approaches.

Moreover, it can also be confirmed that runtime modeling and prediction of quality attributes positively influence decision-making in unexpected or untested conditions.

The consequences of injecting a fault are only undetectable in the *ideal adaptation*. This is because the presence of the fault is known beforehand, and thus appropriate measures are taken to repair the fault before it can lead to a failure. This means we can achieve an ideal result for the self-adaptive system under fault injection. However, this scenario might be unrealistic, since we assume that the system knows when and where the fault will be injected. In Table 10 it can be observed that our approach, *impact prediction*, obtains excellent values regarding availability and costs which are close to the ideal and unrealistic results of the *ideal adaptation*.

In short, both *human optimized* and *impact prediction* solutions try to recover from the erratic behavior, although the chosen adaptation strategies differ. More precisely, both adaptive approaches have configured the enlisting server strategy to improve availability (as outlined in Section 5.3.4). Although this assumption may have a positive effect in most cases, it is not always true as shown by this experiment. Hence, when the failure occurs, the *human optimized* solution selects the enlisting server strategy to increase availability, as shown in Figure 11(d), until it reaches the maximum size of the server pool. If our setup scenario had more available servers, the *human optimized* approach would have reached the maximum pool size too. However, our approach predicts the impact of choosing each strategy and when the failure occurs, it decides to discharge the failing server. As a result, it can be concluded from this experimental procedure that by predicting the impact of each strategy, the self-adaptive system is able to make informed decisions and achieve the desired adaptation goals.

6.3. Effectiveness and scalability of impact prediction

Formal approaches that require an analysis of large state spaces may often be time-consuming, and may lead to considerable overheads. To address this issue, we implemented our approach to complete the execution within the ten-

second time window. This requirement ensures that there is never a different overlapping analysis. Table 11 shows the time that each prediction takes to complete.

Table 11: Time, in seconds, taken to predict the impact of each strategy

Number of servers	Mean	Std. Dev.	95^{th} Percentile
4 Servers	0.15	0.14	0.27
25 Servers	0.22	0.25	0.44
50 Servers	0.40	0.26	0.50
75 Servers	0.55	0.39	0.79
100 Servers	0.79	0.59	1.15

It can be observed that in the case-study with four active servers, each analysis takes, on average, 0.15 s and the 95^{th} percentile takes 0.27 s. Our approach generates and solves a stochastic model for each strategy, so the total time in the analysis is obtained through $Num.\ Strategies \times Time\ spent\ on\ prediction$. In our case-study, there are three strategies, which means analysis usually executes in less than one second ($0.27s \times 3 = 0.81s$), well below the ten-second requirement.

In an attempt to evaluate scalability, we tested our implementation for an increasing number of servers, and the results are illustrated in Figure 12. To be more explicit, we use the 'control run' experiment without injecting any fault.

We ran each experiment 30 times and collected the mean and the standard deviation which are shown in the diagram. We can observe an increase in the time spent in each prediction, but the analysis scales well for a system with up to 100 active servers. If there is a scenario with 100 servers, our approach would take less than 10 seconds to execute, given the prediction of the three

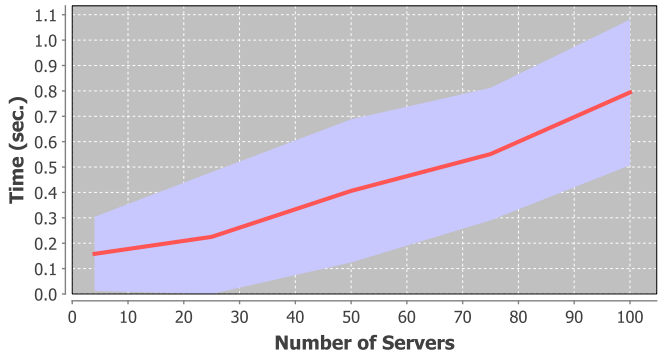


Figure 12: Scalability of our approach regarding the number of servers

adaptation strategies ($1.15s \times 3 = 3.45s$). This would be the case not only for the 95^{th} percentile but also for the maximum time observed.

6.4. Discussion

In the control run experiment, we tested the two self-adaptive solutions against a non-adaptive one. The goal was to determine whether both self-adaptive systems achieve the non-functional requirements, and maintain high availability while reducing the usage of computational resources. The results show that both self-adaptive solutions achieve similar results with the most expensive and highly available non-adaptive solution. However, self-adaptive approaches outperform the non-adaptive one by using 50% less computational resources.

When failures occur, the experimental results show that the two self-adaptive approaches adopt different strategies. In particular, the *human optimized* approach uses static impact vectors while the *impact prediction* approach uses stochastic models to predict the failure behavior of each adaptation strategy. As a result, our method selects the best strategy to recover from failures, and achieves an increased performance and availability, while reducing the usage of computational resources.

Regarding the hypotheses specified in Section 4:

1st hypothesis This hypothesis is confirmed in the *fault injection* experiment. Although the results of the *human optimized* are near-ideal in cases for which the conditions were anticipated, they fail to select the best strategy under unexpected or untested conditions.

2nd hypothesis The *fault injection* experiment allows to conclude that our approach based on reliability prediction recovered from the erroneous behavior, achieving better results when compared to other solutions. Moreover, under normal and known conditions (as in the *control-run* experiment) both self-adaptive approaches present similar results. As such, we can validate the second hypothesis, since using quantitative prediction methods at runtime improves the ability to reach quality goals under unanticipated conditions, while maintaining similar ability in known ones.

3rd hypothesis1 In Section 6.3, we evaluated the performance and scalability of our approach showing overall good results. Moreover, the difference of performance between our *impact prediction* approach and the traditional *human optimized* show no difference in the results presented in *control-run* and *fault-injection* experiments. Thus, validating the third hypothesis by observing that our method maintains performance while meeting the adaptation goals.

7. Contributions

Our approach encompasses a set of contributions to the research community which are following summarized:

- Assess the impact of adaptation strategies – This approach presents a novel method by generating stochastic models to predict the quality impact of each adaptation strategy at runtime. We apply the changes that are imposed by each strategy at the architecture level and predict the quality outcome of such changes to support the planning phase of a self-adaptive

system. From the results presented in Section 6, one can conclude that our approach presents similar results when compared to traditional self-adaptive solutions. However, in unexpected or untested conditions our approach surpasses traditional solutions, approaching the *ideal adaptation*.

- Formal description of the translation from an ADL to DTMC – Section 3.4 proposes a formal description of the translation from an architectural description to a stochastic model. The goal is to support an universal interpretation for the translation, provides rigor to other researchers understand and replicate experiments and also allows the translation to be extended (*e.g.*, encompass different ADLs, stochastic models, quality attributes or the translation of architectural styles).

- Performance assessment – In this study we generate and solve stochastic models which are tasks generally related to time-consuming and state-space explosion issues. This occurs when the model has a large number of states and a great number of transitions between those states exceeding the available memory. With this in mind and as presented in Section 6.3, we tested our approach for larger state-space sizes (*e.g.*, 100 servers) showing good scalability and performance results. This third contribution shows future researchers that is possible to incorporate runtime prediction of quality attributes into their work. This prediction generates and solves stochastic models at runtime while not affecting the system performance and at the same time, complying with the strict time requirements to conduct adaptation planning (in the presented case-study the planning should perform under ten seconds).

8. Discussion

Our approach may raise a set of questions regarding its design, translation or implementation. Those questions are detailed below along with a description on how to address them.

- Accounting with structural changes – Our approach supports structural changes in the architecture thanks to the generation of stochastic models at runtime. Specifically, we build those models from an architecture description containing information about the system, its properties, structure and architectural styles. This way, if the architecture gets updated by adding or removing components, changing their interconnections or use different architectural styles, our approach will generate a proper stochastic model.

- Statistical relevance of the experiments – Each experiment presented in Section 6 was tested for five runs and the data extracted was consistent between runs.

- Threats to validity – Section 6 outlines the validation of our approach through the application of our approach into an actual self-adaptive case-study. One may argue that the results are only specific for the presented scenario and they might be different if we use load-balancing tools to reroute requests or tune adaptation operators. However, our approach aims to avoid static adaptation operators by selecting proper adaptation strategies even under unexpected or untested conditions. In short, the experimentation procedure could be changed, but other unexpected situations may arise leading to a degradation of the system quality goals. To conclude, our validation is not tied to a specific scenario, but rather shows that is possible to present similar results as traditional self-adaptive approaches, while surpassing them under unexpected situations.

- Applicability to other self-adaptive solutions – Although our approach can be applied to other self-adaptive systems, it requires a specific adaptation according to each solution which leads to an increased development effort. We regard this as a limitation factor due to the lack of standard methods described in the self-adaptive software community to bind external analysis approaches to enrich the decision-making of each solution.

- Reliability Vs. Availability – In this paper we use reliability prediction to achieve a particular adaptation goal: high availability. Reliability is usually defined as the "continuity of correct service" [24] or as "it is not doing the wrong thing" [25]. However, we consider as correct service or the "right thing" in this context to be the successfulness of a request being responded (see Section 5.3.2). The concept of reliability is only applied for each ten second time-frame in which we collect information and it is used as an **adaptation metric**. On the other hand, availability is the probability that a system is operating properly at any given time. In this case, we consider availability as the probability that the system is not failed or undergoing a repair action when it is invoked to serve. Specifically, it is calculated by considering the failure rates in all time-frames encompassed during the run and expressing the system probability of readiness for correct service. In this study, availability is used as an **adaptation goal** while reliability is used as an *adaptation metric*.

- Hidden VM cost – Each machine that is activated or deactivated takes less than 3 seconds to begin serving requests. However, in the presented Graphs which are related to the number of active servers (cost in VMs Hour), we only consider that a machine is active if it responds to requests. Thus, the cost of activation and deactivation, representing less than 3 seconds, is hidden from those results.

- Finding the *ideal adaptation* – In Section 6 we introduced the *ideal adaptation* run aiming to compare the results of an ideal perfect adaptation with other adaptive approaches. This adaptation was determined through testing and encompasses the corrective behavior to recover from the introduced failure. Moreover, we consider this adaptation as unrealistic and impossible to be applied to self-adaptive approaches, since it requires the knowledge of where and when the failure is injected in the system. This assumption makes it unmanageable to be applied to real world systems and our purpose is to compare other adaptation results with this perfect, ideal

and unrealistic corrective behavior. To conclude, we regard this type of adaptation as a novel approach in the research community for comparing the effectiveness of different adaptive solutions.

- 'Unexpected' or 'untested' conditions – We consider normal conditions as being a set of circumstances that the architect was expecting to find, designing the system to behave accordingly. However, self-adaptive systems are deployed in dynamic and unpredictable environments which may lead to the occurrence of unexpected circumstances. Moreover, the configuration of such systems encompasses a large number of metrics, impacts, adaptation preferences and utilities leading to a large spectrum of testing possibilities. The lack of automated tools to test and assure such possibilities result in untested conditions. In these 'unexpected' or 'untested' circumstances, the system becomes uncertain of which adaptation to choose to meet its goals and may lead to select a strategy that degrades the system and deviates it from the defined goals.

9. Implications for practice

In this section we share several insights from our work, contributing with answers to future research on the topic of improving decision-making on self-adaptive systems through stochastic models.

- Influence of our work on the current techniques – Our approach shows that is possible to predict the quality behavior of an adaptation allowing the controller to perform informed decisions according to the designed adaptation goals. In addition, our method adds value to current techniques by generating and solving stochastic models at runtime in an automated fashion.

- Correctness of the generated mathematical model. In a previous study [27] we show the correctness of the translation procedure from an architectural

description to the stochastic model. Since in this work we use the same translation method, we assume that the generated mathematical models are correct, representing correctly the system's behavior and providing accurate reliability values.

- Application to other quality goals – In the generated mathematical model we predicted reliability, but it can be applied to other quality goals as demonstrated by Gallotti *et al.* [15].

- Integration in self-adaptive approaches. Our work motivated Rainbow development team to include in subsequent releases a feature that allows to dynamically update the impact of each strategy from runtime prediction of quality goals.

10. Conclusion

Self-adaptive systems are becoming more common in our daily tasks, although this is sometimes unnoticed, until a failure occurs. When these systems affect human lives, like self-driving cars, they only need to make one wrong decision to fail and become discredited by their users and cease to be dependable, especially when human lives are at risk. Bearing this in mind, we applied our automated method to predict reliability from a software architecture to enhance the planning phase of self-adaptive systems. Our goal was to show the applicability of our approach and its effectiveness in improving real world issues.

Evidence of the limitations of current decision-making approaches and the validity of our method was obtained by conducting a realistic experiment based on a news infrastructure hosted in a cloud environment. The experiment which is discussed in Section 6.2, confirms that traditional decision-making approaches (*i.e.*, human optimized) fail to select the best strategy in unexpected or untested conditions and this leads to the degradation of the delivered service. In addition, the same experiment enabled us to conclude that our approach (*i.e.*, impact prediction) can recover from erroneous behavior, by validating the use of

quantitative prediction methods at runtime. This method improves the ability to reach quality goals in unforeseen circumstances, while maintaining a similar ability in known ones.

Our approach entails the generation of stochastic models and the means of solving them, which are tasks usually seen as time-consuming and inefficient. It can be concluded from the experiment in Section 6.2 that our approach (*i.e.*, impact prediction) performs as well as other approaches by providing similar values of resolved requests, throughput and resource consumption. Furthermore, we conducted an experiment to assess the performance and scalability of our approach. The obtained data shows that our method performs under one second and if the system could scale to one hundred web-servers, our approach would still fulfill the performance requirements.

To conclude, the experimental work covered in this chapter can be of value to the self-adaptive community by employing a method that allows a correct and context-sensitive strategy to be adopted to achieve the specified quality goals. Given that self-adaptive systems are recognized as a solution for dealing with highly complex environments, we expect our approach to further improve the current solutions in unforeseen circumstances.

Acknowledgments

This research was supported by a grant from the Carnegie Mellon|Portugal project AFFIDAVIT (PT/ELE/0035/2009). FCT – Fundação para a Ciência e a Tecnologia also supported this work in the scope of the project DECAF: An Exploratory Study of Distributed Cloud Application Failures (reference EXPL/EEI-ESS/2542/2013) and a doctoral grant [SFRH/BD/89702/2012]. The authors would also like to thank the contribution from Paulo Casanova.

APPENDIX

Appendix A. Prism model for the architectural changes for the Enlist Server strategy

Following we present the generated Prism code encompassing the DTMC for the Enlist Server Strategy illustrated in Figure 5.

```
dtmc

// Number of components: 6 + 2 Absorbing states
global s : [1..8] init 1;

const double p_Web0 = 1.0;
const double p_Web2 = 0.9;
const double p_DB = 1.0;
const double p_Web3 = 1.0;
const double p_LB0 = 1.0;
const double p_Web1 = 1.0;

// Component name - LB0
module LB0
    [] s=1 -> p_LB0*0.25:(s'=2) + p_LB0*0.25:(s'=3) +
        p_LB0*0.25:(s'=4) + p_LB0*0.25:(s'=5) +
        (1-p_LB0):(s'=8);
endmodule

// Component name - Web2
module Web2
    [] s=2 -> p_Web2*1.0:(s'=6) + (1-p_Web2):(s'=8);
endmodule
```

```
// Component name - Web3
module Web3
    [] s=3 -> p_Web3*1.0:(s'=6) + (1-p_Web3):(s'=8);
endmodule

// Component name - Web1
module Web1
    [] s=4 -> p_Web1*1.0:(s'=6) + (1-p_Web1):(s'=8);
endmodule

// Component name - Web0
module Web0
    [] s=5 -> p_Web0*1.0:(s'=6) + (1-p_Web0):(s'=8);
endmodule

// Component name - DB
module DB
    [] s=6 -> p_DB:(s'=7) + (1-p_DB):(s'=8);
endmodule

// Absorbing states
module absorbingStates
    // Final states
    [] s=7 -> (s'=7);
    [] s=8 -> (s'=8);
endmodule

label "available" = (s=7);
label "unavailable" = (s=8);
```

Appendix B. Prism model for the architectural changes for the Discharge the Least Reliable

Following we present the generated Prism code encompassing the DTMC for the Discharge Server Strategy illustrated in Figure 6.

```
dtmc
// Number of components: 4 + 2 Absorbing states
global s : [1..6] init 1;

const double p_DB = 1.0;
const double p_Web0 = 1.0;
const double p_LB0 = 1.0;
const double p_Web1 = 1.0;

// Component name - LB0
module LB0
    [] s=1 -> p_LB0*0.5:(s'=2) + p_LB0*0.5:(s'=3) +
        (1-p_LB0):(s'=6);
endmodule

// Component name - Web0
module Web0
    [] s=2 -> p_Web0*1.0:(s'=4) + (1-p_Web0):(s'=6);
endmodule
```

```
// Component name - Web1
module Web1
    [] s=3 -> p_Web1*1.0:(s'=4) + (1-p_Web1):(s'=6);
endmodule

// Component name - DB
module DB
    [] s=4 -> p_DB:(s'=5) + (1-p_DB):(s'=6);
endmodule

// Absorbing states
module absorbingStates
    // Final states
    [] s=5 -> (s'=5);
    [] s=6 -> (s'=6);
endmodule

label "available" = (s=5);
label "unavailable" = (s=6);
```

References

[1] S.-W. Cheng, Rainbow: Cost-effective software architecture-based self-adaptation, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (2008).

[2] E. M. Fredericks, A. J. Ramirez, B. H. C. Cheng, Towards run-time testing of dynamic adaptive systems, in: Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 169–174.

[3] F. D. Macías-Escrivá, R. Haber, R. del Toro, V. Hernandez, Self-adaptive systems: A survey of current approaches, research challenges and applications, Expert Systems with Applications 40 (18) (2013) 7267–7279.

[4] M. Salehie, L. Tahvildari, Self-adaptive software: Landscape and research challenges, ACM Trans. Auton. Adapt. Syst. 4 (2) (2009) 14:1–14:42.

[5] R. Lemos, H. Giese, H. A. Muller, M. Shaw, J. Andersson, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Goeschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, M. Litoiu, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezze, C. Prehofe, W. Schäfer, R. Schlichting, B. Schmerl, D. B. Smith, J. P. Sousa, G. Tamura, L. Tahvildari, N. M. Villegas, T. Vogel, D. Weyns, K. Wong, J. Wuttke, Software Engineering for Self-Adaptive Systems: A Second Research Roadmap, in: R. de Lemos, H. Giese, H. Müller, M. Shaw (Eds.), Software Engineering for Self-Adaptive Systems, Vol. 7475 of Dagstuhl Seminar Proceedings, Springer, 2013, pp. 1–26.

[6] IBM Corp., An architectural blueprint for autonomic computing, IBM Corp., USA, 2004.

[7] T. M. King, A. E. Ramirez, R. Cruz, P. J. Clarke, An integrated self-testing

framework for autonomic computing systems., Journal of Computers 2 (9) (2007) 37–49.

[8] N. Bencomo, A. Belaggoun, V. Issarny, Dynamic decision networks for decision-making in self-adaptive systems: A case study, in: Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 113–122.

[9] C. Ghezzi, L. S. Pinto, P. Spoletini, G. Tamburrelli, Managing non-functional uncertainty via model-driven adaptivity, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 33–42.

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

[11] B. Schmerl, J. Cámara, J. Gennari, D. Garlan, P. Casanova, G. A. Moreno, T. J. Glazier, J. M. Barnes, Architecture-based self-protection: Composing and reasoning about denial-of-service mitigations, in: HotSoS 2014: 2014 Symposium and Bootcamp on the Science of Security, Raleigh, NC, USA, 2014.

[12] J. C. Moreno, A. Lopes, D. Garlan, B. Schmerl, Formal Aspects of Component Software: 11th International Symposium, FACS 2014, Bertinoro, Italy, September 10-12, 2014, Revised Selected Papers, Springer International Publishing, Cham, 2015, Ch. Impact Models for Architecture-Based Self-adaptive Systems, pp. 89–107.

[13] J. Cámara, D. Garlan, B. Schmerl, A. Pandey, Optimal planning for architecture-based self-adaptation via model checking of stochastic games, in: Proceedings of the 10th DADS Track of the 30th ACM Symposium on Applied Computing, Salamanca, Spain, 2015.

[14] R. Calinescu, C. Ghezzi, M. Kwiatkowska, R. Mirandola, Self-adaptive software needs quantitative verification at runtime, Communications of the ACM 55 (9) (2012) 69.

[15] S. Gallotti, C. Ghezzi, R. Mirandola, G. Tamburrelli, Quality prediction of service compositions through probabilistic model checking, in: Proceedings of the 4th International Conference on Quality of Software-Architectures: Models and Architectures, QoSA '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 119–134.

[16] M. Kwiatkowska, G. Norman, D. Parker, Prism: Probabilistic model checking for performance and reliability analysis, ACM SIGMETRICS Performance Evaluation Review 36 (4) (2009) 40–45.

[17] ISO/IEC/IEEE, Systems and software engineering – Architecture description, ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000) (2011) 1–46.

[18] J. Camara, R. de Lemos, Evaluation of resilience in self-adaptive systems using probabilistic model-checking, in: Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on, 2012, pp. 53–62.

[19] T. Zheng, M. Woodside, M. Litoiu, Performance model estimation and tracking using optimal filters, Software Engineering, IEEE Transactions on 34 (3) (2008) 391–406.

[20] A. Filieri, C. Ghezzi, A. Leva, M. Maggio, Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements, 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011) 0 (2011) 283–292.

[21] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, G. Tamburrelli, Dynamic qos management and optimization in service-based systems, Software Engineering, IEEE Transactions on 37 (3) (2011) 387–409.

[22] I. Epifani, C. Ghezzi, R. Mirandola, G. Tamburrelli, Model evolution by run-time parameter adaptation, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 111–121.

[23] J. M. Spivey, The Z Notation: A Reference Manual, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[24] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Transactions on Dependable and Secure Computing 1 (1) (2004) 11–33.

[25] J. Gray, Why do Computers Stop and What Can be Done About It?, Tech. Rep. TR 85.7, Tandem Computers (June 1985).

[26] K. Goševa-Popstojanova, K. Trivedi, Architecture-based approach to reliability assessment of software systems, Performance Evaluation 45 (2) (2001) 179–204.

[27] J. M. Franco, R. Barbosa, M. Zenha-Rela, Automated Reliability Prediction from Formal Architectural Descriptions, in: 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, IEEE computer society, Helsinki, Finland, 2012, pp. 302–309.

[28] M. Kwiatkowska, G. Norman, D. Parker, Stochastic model checking, in: Proceedings of the 7th International Conference on Formal Methods for Performance Evaluation, SFM'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 220–270.

[29] D. Garlan, R. Monroe, D. Wile, Acme: An architecture description interchange language, in: Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON '97, IBM Press, Toronto, Ontario, Canada, 1997, pp. 7–.

[30] R. N. Taylor, N. Medvidovic, E. M. Dashofy, Software Architecture: Foundations, Theory, and Practice, Wiley Publishing, 2009.

[31] M. Kwiatkowska, G. Norman, D. Parker, PRISM 4.0: Verification of probabilistic real-time systems, in: G. Gopalakrishnan, S. Qadeer (Eds.), Proc. 23rd International Conference on Computer Aided Verification (CAV'11), Vol. 6806 of LNCS, Springer, 2011, pp. 585–591.

[32] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, A. L. Wolf, E. L. Wolf, An architecture-based approach to self-adaptive software, IEEE Intelligent Systems 14 (1999) 54–62.

[33] D. Weyns, S. Malek, J. Andersson, FORMS: Unifying reference model for formal specification of distributed self-adaptive systems, ACM Trans. Auton. Adapt. Syst. 7 (1) (2012) 8:1–8:61.

[34] N. M. Villegas, H. a. Müller, G. Tamura, L. Duchien, R. Casallas, A framework for evaluating quality-driven self-adaptive software systems, Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems - SEAMS '11 1 (2011) 80.

[35] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, P. Steenkiste, Rainbow: architecture-based self-adaptation with reusable infrastructure, Computer 37 (10) (2004) 46–54.

[36] S.-W. Cheng, D. Garlan, 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, IEEE Computer Society, Washington, DC, USA, 2009, pp. 132–141.

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

[38] N. R. Storey, Safety Critical Computer Systems, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[39] B. Kirwan, A Guide To Practical Human Reliability Assessment, Vol. 1, Taylor & Francis, 1994.