Robustness Evaluation of Controllers in Self-Adaptive Software Systems

Javier Cámara
University of Coimbra, Portugal
jcmoreno@dei.uc.pt

Rogério de Lemos
University of Kent, UK
r.delemos@kent.ac.uk

Nuno Laranjeiro, Rafael Ventura, Marco Vieira
University of Coimbra, Portugal
{cnl, ventura, mvieira}@dei.uc.pt

Abstract

*An increasingly important requirement for software-intensive systems is the ability to self-manage by adapting their structure and behavior at run-time in an autonomous way as a response to a variety of changes that may occur to the system, its environment, or its goals. In particular, self-adaptive (or autonomic) systems incorporate complex software components that act as controllers of a target system by executing actions through effectors, based on information monitored by probes. However, although these controllers are becoming critical in many application domains, it is still difficult to assess their robustness. The proposed approach for evaluating the robustness of controllers for self-adaptive software systems, is aimed at the effective identification of design faults. To achieve this objective, our proposal is based on a set of robustness tests that include the provision of mutated inputs to the interfaces between the controller and the target system (*i.e.*, probes). The feasibility of the approach is evaluated in the context of Znn.com, a case study implemented using the Rainbow framework for architecture-based self-adaptation.*

1. Introduction

What distinguishes a self-adaptive system from any other system is its ability to continuously deliver its services despite changes that may occur in the system, its environment or its goals. A key component that enables self-adaptive systems to handle changes at run-time is a controller that relies on a feedback loop for managing adaptations [3] by executing actions through effectors on the target system, based on information monitored by probes. In the context of complex software systems, these controllers consist usually of four distinct operational stages, namely, monitoring, analysis, planning and execution [13], that implement the traditional *sense-plan-act* architectures. Although major advances have been made, existing approaches in self-adaptation do not systematically address the need to determine if a self-adaptive system can deliver a service that can justifiably be trusted when facing changes (*i.e.*, that it will be *resilient* [16]). This lack of assurances is one issue that has hampered the widespread adoption of self-adaptive systems. A major problem associated with the provision of evidence is the combinatorial nature of the stateful aspects of a controller and the changes that may affect the system being controlled. Since the operational stages should be functionally independent from each other, a change might have a different impact on the controller depending on the state in which a controller is. Moreover, if the controller is expected to act upon a change when it occurs, there is a wide range of issues that needs to be considered when producing the appropriate action, including the place in which the change has occurred, the type and the frequency of the change, and whether it can be anticipated [2]. These factors need to be taken into account if assurances need to be provided about the services to be delivered by the target system, hence techniques need to be devised to uncover potential faults in the controller.

This paper describes an approach for evaluating the robustness of controllers for self-adaptive systems by abstracting away, in a first instance, from the state of the target system being controlled. The rationale behind this is the fact that the complexity associated with these controllers is such that we need first to devise novel means for evaluating the core logic that enables adaptation, before delving into the ensemble target system plus controller. Moreover, if the robustness evaluation is performed on the ensemble, some of the controller faults could be masked by the target system, or their effects upon the system could be more difficult to analyse. Hence the decision to define an approach that can be used in the robustness evaluation of different controllers, assuming that the core logic of the different op-

erational stages is basically the same on the different controllers [9]. In this way, we restrict the robustness tests to the inputs of the controller, which are characterized by the probes. Although the proposed approach abstracts away from the target system, we need to consider the stateful aspects of the controller, which are related to the operational stage of the controller.

This paper has two clear contributions, which can be seen from different viewpoints.

- from the perspective of resilience, the paper presents an approach for evaluating the robustness of controllers for self-adaptive software systems by defining a set of mutation rules that should be applied to the inputs of the controller, a tailored version of controller failure modes, and an experimental setup and testing procedure that is specific to self-adaptive systems.

- from the perspective of robustness evaluation, the paper presents an approach that enables the evaluation of controllers in different states by defining how the controller interface should be tested according to the target system changeload, and the operational stage of the controller.

The feasibility of the proposed approach for evaluating the robustness of controllers for self-adaptive software systems is evaluated in the context of Rainbow framework for architecture-based self-adaptation using the Znn.com case study [9]. Rainbow has been chosen for performing the experiments since its software has been widely available, its structure facilitates access to its internal components, its design is amenable to the injection of faults, and the logs Rainbow produces are suitable for analysing the effects of the injected faults upon the controller. For evaluating the robustness of Rainbow, all the faults are injected at its interface (*i.e.*, the probes that interface Rainbow with the system) based on an identified set of mutation rules.

The rest of this paper is structured as follows. Section 2 provides some background on self-adaptive systems and related work in the area of robustness testing. Section 3 introduces the Znn.com case study, which is used throughout the paper for illustrating the proposed approach. Section 4 describes our approach that is focused specifically on evaluating the robustness of controllers for self-adaptive systems. Section 5 presents the experimental results obtained from the evaluation of our approach. Finally, Section 6 concludes the paper and indicates future research directions.

2. Background and Related Work

Over the past few years, run-time management of increasingly complex software-intensive systems has become a central concern in Software Engineering [5, 8]. Concretely, a major issue in this area is related to achieving conformance to functional and non-functional requirements in a dependable and cost-effective manner while changes may affect the system, its environment, and system goals.

One of the proposals addressing this concern was IBM's Autonomic Computing initiative [13], which has introduced a layer implementing what is known as the MAPE-K control loop to Monitor, Analyse, Plan, and Execute adaptation (with a Knowledge base acting as a cornerstone of the process) for purpose of managing a target system. Some successful approaches that rely on this closed-loop control paradigm for self-adaptation exploit architectural models for reasoning about the target system under management [9, 20]. In particular, Rainbow [9] provides a base of reusable infrastructure that can be applied to a wide range of systems through customisation. Section 4.4 provides further information about Rainbow, which is used for the experimental validation of our approach.

2.1. Resilience Evaluation in Self-Adaptive Systems

Though the field of self-adaptive software system is relatively new, there are already contributions regarding the provision of assurances, though the main focus of these contributions has been towards the ensemble target system plus controller regarding application. To the best of our knowledge, nothing has been done regarding the evaluation of controllers, though there is already some ground work pointing in this direction [5, 8].

One of the areas that are related to that of resilience evaluation is that of resilience benchmarking, which encompasses techniques from previous efforts in performance benchmarking [10], dependability benchmarking [12], and security benchmarking [18], due to its inherent relation to performance, dependability and security. Comparing to established benchmarks, a resilience benchmark may be specified following the same basic approach, but comprising a wide-ranging changeload (which will include, but will be not limited to, faults), as well as resilience metrics [1].

Other approaches deal with resilience evaluation through quantitative analysis using probabilistic model-checking [4], considering the system environment as the only source of change and leaving out changes that are internal to the system. The cited approaches quantitatively measure resilience in the self-adaptive system when facing changes either internal or external to the system. However, they do not deal with an additional source of problems from the perspective of resilience, which are robustness issues addressed by the techniques presented in the current paper.

2.2. Robustness Testing

Robustness testing allows the characterisation of the behaviour of a system or component in presence of erroneous input conditions. Robustness tests stimulate the system under testing in a way that may trigger internal errors and this allows developers to solve or wrap the identified problems. This technique can be used to differentiate systems according to the number and type of errors uncovered [19].

Ballista [14] uses a set of tests that combine acceptable and exceptional values on calls to kernel functions of operating systems. The parameter values used in each invocation are randomly extracted from a set of predefined tests and for each parameter a set of values of a certain data type is associated. Each operating system is classified in terms of its robustness and according to a predefined scale (the CRASH scale) that distinguishes several failure modes.

MAFALDA (Microkernel Assessment by Fault injection AnaLysis and Design Aid) [21] is a tool that enables the characterisation of the behaviour of microkernels in the presence of faults. Fault injection is performed at two levels: in the parameters of system calls and in the memory segments holding the target microkernel. However, only the former is relevant when the goal is robustness testing.

In previous work, we defined an approach to assess the behaviour of web services in the presence of tampered SOAP messages [24]. The proposed approach consists of a set of robustness tests based on invalid web services call parameters. The web services are classified according to the failures observed during the execution of the tests and using an adaptation of the CRASH scale [15].

The aforementioned works implement robustness testing approaches that do not consider the state of the system under test. In [7] the impact of state on robustness testing of a safety-critical operating system (OS) is investigated by including the OS state in test cases definition. Although system-specific, results show that the state can play an important role in testing, being able to cover more cases when compared to the traditional approaches.

An approach for robustness testing method of stateful Web Services, modelled with Symbolic Transition Systems, is presented in [23]. A test case generation method is proposed using unusual values and replacement and additions of operation names. States are transversed using different operations and starting from a system specification which, depending on the system being tested, may not always be available. The authors assume that messages sent and received are only SOAP messages and suggest that a web service could be considered as a grey box from which any type of message could be observed, increasing the potential of the technique.

3. Case Study

To illustrate our approach for robustness testing, we use the Znn.com case study [6], which is implemented using Rainbow, and is able to reproduce the typical infrastructure for a news website. It has a three-tier architecture consisting of a set of servers that provide contents from backend databases to clients via front-end presentation logic. Architecturally, it is a web-based client-server system that satisfies an N-tier style, as illustrated in Figure 1. The system uses a load balancer to balance requests across a pool of replicated servers, the size of which can be adjusted according to service demand. A set of client processes makes stateless requests, and the servers deliver the requested contents (*i.e.,* text, images and videos).

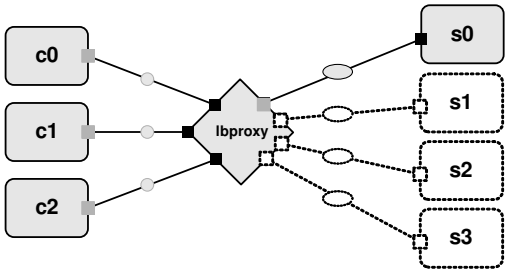

Figure 1. Znn.com system architecture

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

In the case of Znn.com, Rainbow is capable of analysing trade-offs among the different objectives, and execute different adaptations according to the particular run-time conditions of the system. For instance, when response time becomes too high, the system should increment server pool size if it is within budget to improve its performance; otherwise, servers should be switched to textual mode (start serving minimal text content) if cost is near budget limit.

4. Approach

Our proposal for evaluating the robustness of a self-adaptive software system considers the model depicted in

Figure 2. The *environment* consists of all non-controllable elements that determine the operating conditions of the system (*e.g.*, hardware, network, physical context, etc.). Regarding the system itself, we distinguish two main subsystems: a *target system*, which interacts with the environment by monitoring relevant variables associated with operating conditions, and a *controller* that manages the target system, driving adaptation whenever it is required. Concretely, the controller carries out its function by: *(i)* monitoring the target system and environment through *probes* that provide information about the value of relevant variables, *(ii)* deciding whether the current state demands adaptation, and if this is the case, *(iii)* applying a sequence of control actions through system-level *effectors*.

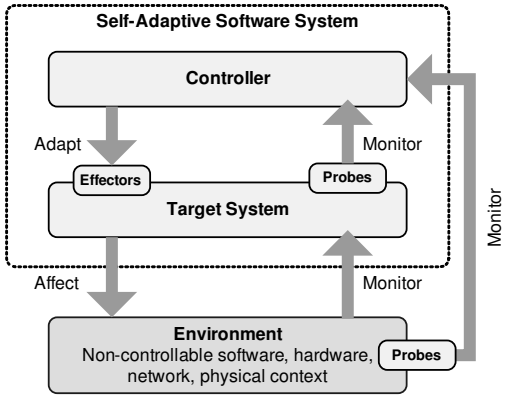

Figure 2. Self-adaptive software system

In this paper, instead of looking into the robustness of the entire system, we focus on the robustness of the controller, *i.e.,* we modify the probes' inputs into the controller with the intent of evaluating how robust is the controller regarding changes that may affect its interface. For this evaluation, the controller is considered to be stateful since for the same input, the controller's internal state may influence its output. In order to tackle this issue, we consider input mutation during the different operation stages of the controller (*i.e.*, analysis, planning, execution) to create an appropriate context for evaluating its robustness. The key elements of our approach are: *changeload*, which is a set of representative change scenarios, where changes are based on controller input mutations; *failure mode classification*, that characterizes the run-time behaviour of the controller while the target system is running in the presence of the changeload; and *robustness tests*, the mutation rules that are applied to the input probes into the controller.

In the following, in addition of describing the key elements of our approach, we also present how robustness tests are performed at the controller interface by mutating the inputs provided by the probes. To exemplify the principles of our approach, we have instantiated it into Rainbow [9].

4.1. Changeload Model

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

Definition 1 (Change Type) *A change type is a tuple* (src, m, A) *that characterises a change, where:*

- *src identifies the source probe type where a mutation rule is applied,*
- *m identifies the mutation rule applied on probe input,*
- $A = \langle a_1, \ldots, a_n \rangle$ *(possibly empty) is a vector of attributes that holds specific information about the mutation rule.*

Example 1 *In Znn.com, consider the change "Set an invalid timestamp date on a response time probe (type ClientProxyProbeT)". A possible change type definition for this would be:*

$$invalidDateCPP\_CT = (\ ClientProxyProbeT,\ TSInvalidDate,\ \langle date \rangle)$$

Definition 2 (Change) *Given a set of change types CT, a change is a tuple* $(ct, srcinst, V_A, ti, d)$ *that corresponds to an instantiation of a change type, where:*

- $ct = (src, m, A) \in CT$ *determines the change type to be instanced as a change,*
- *srcinst is the probe instance that is the source of change (*i.e.*, in which the input is mutated),*
- $V_A = \langle v_{A1}, \ldots, v_{An} \rangle$ *is a vector of attribute values instantiating the attributes in A,*
- $ti \in \mathbb{R}_0^+$ *determines the time instant in which the change is triggered,*
- $d \in \mathbb{R}^+$ *is the duration associated with the change.*

It is worth observing that while some specific changes may be transient, impacting the controller's input during a particular amount of time, in the definition above duration can be considered equal to ∞ if the change is permanent.

Example 2 *If we consider the change type described in Example 1, a possible instantiation of it could be:*

$$(invalidDateCPP\_CT,\ ClientProxyProbe1,\ \langle '2/29/1984' \rangle,\ 10,\ 2)$$

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

The main base concept in our changeload model is the *scenario*. A scenario is a postulated sequence of events that

captures the state of the system and its environment, system goals [1], and changes affecting all the aforementioned elements. It is defined in terms of state (system and environment) and changes applied to that state.

Definition 3 (Scenario) *A scenario is a tuple* (wl, oc, C)*, where:*

- *wl represents the workload, that is, the amount and type of work assigned to the system (not necessarily static),*
- *oc are the operational conditions of the system (including software and hardware resources needed for the system to perform its service),*
- *C is a set of changes applied to controller input in the presence of the workload and operational conditions.*

Based on the definition above, a *change scenario* is one which includes a non-empty set of changes ($C \neq \emptyset$).

Definition 4 (Changeload) *A changeload is a set of change scenarios.*

4.2. Controller Failure Modes

The robustness of a controller for a self-adaptive system can be classified according to an adapted version of the CRASH scale [15], which distinguishes the following failure modes:

1. **Catastrophic**: the whole controller crashes or becomes corrupted (this might include the OS or machine on which the controller is running). No output is produced.
2. **Restart**: the controllers execution hangs and may not issue any output commands, or send always the same command, within the worst case execution time associated with the adaptation cycle. The controller needs to be externally re-booted.
3. **Abort**: abnormal behavior in the controller occurs due to an exception raised at run-time inside of the controller.
4. **Silent**: the controller fails to acknowledge an error, for instance by signalling an exception, which causes the controller to continue operating improperly.
5. **Hindering:** the controller fails to return a correct error code, which may hinder error recovery. The difference between a silent failure and this case is that, here an error is acknowledged by the controller but the returned error code is incorrect.

In particular, it is worth observing that the tailored version of the CRASH scale for controllers in self-adaptive software systems includes an specific adaptation which is related with time (2).

4.3. Robustness Tests

The basis of the proposed approach for evaluating the robustness of controllers for self-adaptive software systems relies on stimulating the interface of the controller, which consists of probes that monitor both the target system and its environment (see Figure 2). For evaluating how robust is the controller, regarding changes that my affect its interface, the probes' inputs into the controller are modified according to a comprehensive set of mutation rules. Moreover, since the inputs of these probes may affect the different stages of a MAPE-K control loop, the evaluation needs to consider the controller as stateful. Although for evaluating the robustness of a controller we are able to abstract away from the application (target system), we nevertheless use the application to drive the evaluation.

4.3.1 Mutation Rules

The set of robustness tests performed is automatically generated by applying a set of predefined mutation rules to the messages sent by probes, which characterizes the monitoring stage of the controller. Although concrete message formats and additional elements may exist depending on the case, the basic input supplied by probes to the controller typically consists of three basic elements: *(i)* an identifier of the variable being monitored, *(ii)* the actual value for the variable, and *(iii)* a timestamp that provides a temporal context for the variable being monitored. For example, in the case of Rainbow, the kind of input received by the controller consists of simple messages encoded as text strings with the following format:

```
[ timestamp ] variable_name :  variable_value
```

Based on this general description of probe input, we propose a set of rules (Table 1), which have been defined based on previous works on robustness testing [15, 22, 24], and explore limit conditions that are typically the source of robustness problems.

4.3.2 Probe Usage Categories

The effect of applying mutation rules on the outputs generated by the probes may manifest in different ways (or not

[1] For the sake of simplicity, in this paper we abstract away from system goals, which are not required to deal with robustness evaluation of the controller.

Type	Rule Name	Description
Message	MsgNull	Replace by null value
	MsgEmpty	Replace by empty string
	MsgPredefined	Replace by predefined string
	MsgNonPrintable	Replace by string with non-printable characters
	MsgAddNonPrintable	Add non-printable characters to the string
	MsgOverflow	Add characters to overflow max string size
Timestamp	TSEmpty	Replace by empty timestamp
	TSRemove	Remove timestamp from response
	TSInvalidFormat	Replace by timestamp with invalid format
	TSDateMaxRange	Replace date in timestamp by maximum valid
	TSDateMinRange	Replace date in timestamp by minimum valid
	TSDateMaxRangePlusOne	Replace date in timestamp by maximum valid plus one
	TSDateMinRangeMinusOne	Replace date in timestamp by minimum valid minus one
	TSDateAdd100	Add 100 years to date in timestamp
	TSDateSubtract100	Subtract 100 years from date in timestamp
	TSInvalidDate	Replace date in timestamp by invalid date (*e.g.*, 2/29/1985)
Variable Name	VNRemove	Remove variable name
	VNSwap	Replace by different valid variable name of same type
	VNSwapType	Replace by different valid variable name of different type
	VNInvalidFormat	Replace by variable name with invalid format
	VNNotExist	Replace by non-existing variable name
Variable Value	VVRemove	Remove variable value
	VVInvalidFormat	Replace value by one with invalid format
	Number	
	VVNumAbsoluteMinusOne	Replace by -1
	VVNumAbsoluteOne	Replace by 1
	VVNumAbsoluteZero	Replace by 0
	VVNumAddOne	Add 1
	VVNumSubtractOne	Subtract 1
	VVNumMax	Replace by maximum number valid for type
	VVNumMin	Replace by minimum number valid for type
	VVNumMaxPlusOne	Replace by maximum number valid for type plus one
	VVNumMinMinusOne	Replace by minimum number valid for type minus one
	VVNumMaxRange	Replace by maximum number valid for variable
	VVNumMinRange	Replace by minimum number valid for variable
	VVNumMaxRangePlusOne	Replace by maximum number valid for variable plus one
	VVNumMinRangeMinusOne	Replace by minimum number valid for variable minus one
	Boolean	
	VVBoolPredefined	Replace by predefined value

Table 1. Mutation rules for probes

manifest at all) in the controller, depending on its internal state. This results from the stateful nature of the controller, which may use different inputs and in a different way, depending on its operation stage (*i.e.*, analysis, planning, or execution). Changes in the internal operation stage of the controller are also induced by input obtained from probes.

Table 2 distinguishes different probe categories, according to their use in the different operation stages of the controller. Different robustness issues may arise in the controller, depending on the particular stage/probe in which mutation rules are applied, even if the set of mutations rules applied are the same. The same probe can belong to different usage categories and be used during different stages in the controller. We consider the controller to be a gray box, while its different operation stages are black boxes on which probe mutation is applied. For the time being, we assume that each of these black boxes are stateless, even if that is not the case as far as the target system is concerned. The different stages in the controller are sequential, while monitoring is transversal to all of them.

4.4. Testing Procedure

As discussed previously, inputs to the Rainbow controller are delivered with the use of probes, which provide important system and environment information such as experienced response time, network latency, or server load. Robustness testing focuses on the controller's input points (*i.e.,* the probe information). Therefore, a complete robustness experiment must include a set of tests that focuses precisely on the information provided by each of the input probes.

Figure 3 represents the complete experimental procedure and, as we can see in the figure, each experiment includes several tests, each one focusing on a given probe. For each probe (which, at runtime is continuously delivering information to the controller under test) we apply a single change for each probe data sample. However, we apply (in the subsequent probe data samples) the same change for a given period of time, which potentially gives us the possibility of further disturbing the system under test.

Each robustness test focuses on a single mutation rule type, and having identified the three major controller operational stages (analysis, planning, execution), we must execute the tests with the controller in each of these stages, as it allows us to cover more cases and potentially disclose more robustness problems. Therefore, in each test, we must drive the system from an initial state to a target state by submitting the system to a changeload for a given amount of time (t1 in Figure 3). This target state is the one in which the system should be in order to start testing, and can correspond to any entry point to any of the three controller stages previously mentioned. With the controller in the target state, we

Probe Usage Category	Controller Stage	Input Usage	Example Rainbow/Znn.com
Analysis	The controller analyzes the current state of the target system for detecting anomalies, and triggering adaptation if needed.	Anomaly detection.	Rainbow checks whether the current response time (through response time probes) in Znn.com is above the maximum acceptable response time threshold.
Planning	The controller determines if any adaptation plans can be applied to the system, and selects the best alternative.	Adaptation plan selection.	If the maximum response time is above threshold, Rainbow detects anomaly and determines the best adaptation strategy (based on response time and network latency probes).
Execution	The controller executes the selected course of action.	Control action selection.	Rainbow executes the selected adaptation strategy for reducing response time (monitors response time and network latency probes).

Table 2. Probe categories

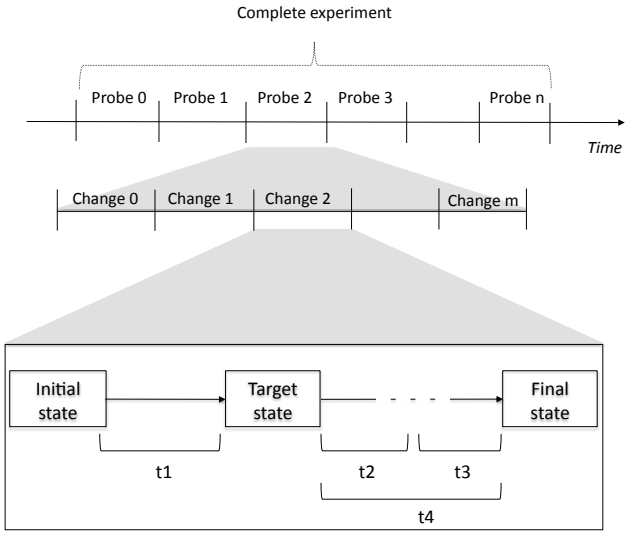

Figure 3. Robustness testing procedure

can start applying the changes (of the same type) during a t2 amount of time (see Figure 3) and while the controller is on the target state. This period of time should be set to the typical time required to transition from the target controller state for the test to the next state. After this probe mutation period there is a t3 period which is the time required for the system to reach a final state, which marks the end of the current test, and corresponds to the completion of the controller's execution stage. At most, t3 should be set to the worst case execution duration found in the adaptation strategies specification. The t4 period (composed by t3 and t2) is an observation period that can be used to register any deviations from expected controller behaviour.

5. Experimental Evaluation

The aim of our experiments is assessing the validity of our approach to evaluate controller robustness in self-adaptive systems. In particular, we evaluate the robustness of Rainbow's controller (*i.e., Rainbow master*) on an implementation of the Znn.com case study described in Section 3.

5.1. The Rainbow Framework

In this paper, we focus on Rainbow [9], an architecture-based platform for self-adaptation, which provides a substantial base of reusable infrastructure through customization, which aims to reduce the cost of self-adaptive system development. Rainbow has distinctive features: an explicit architecture model of the target system, a collection of adaptation strategies, and utility preferences to guide adaptation.

The framework defined by Rainbow includes mechanisms for (Figure 4): monitoring a target system and its environment (using the observations for updating the architectural model of the target system), detecting opportunities for improving the system's quality of services (QoS), deciding the best course of adaptation based on the state of the system, and effecting the most appropriate changes. Rainbow's component-and-connector architectural model of the target system is one of the main elements used in its decision-making process, using it to update monitored system information and reason about appropriate adaptation mechanisms for a particular situation.

The main components of the framework are:

- **Architecture Evaluator:** evaluates the model upon update to ensure that the system is operating within an acceptable range. If the evaluator determines that the system is not operating within the accepted range, it triggers the adaptation.
- **Adaptation Manager:** chooses a suitable strategy based on current state of the system (reflected in the architectural model).
- **Strategy Executor:** executes the strategy chosen by the adaptation manager on the running system via system-level effectors.
- **Model Manager:** updates the architecture model using the information observed in the system via probes.

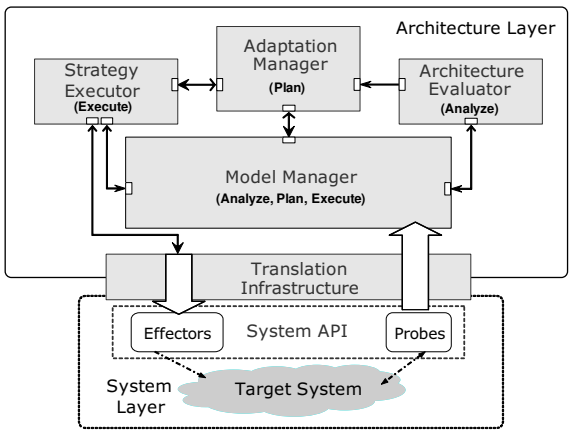

Figure 4. The Rainbow framework

5.2. Experimental Setup

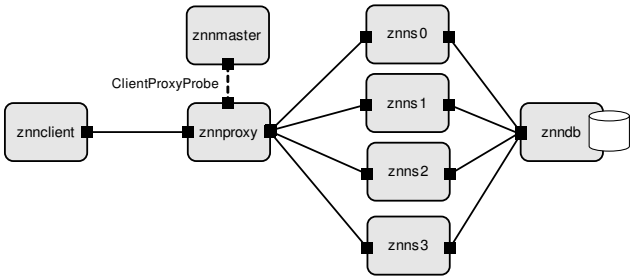

Figure 5. Znn.com experimental setup

For our experimental setup, we deployed Rainbow and the corresponding implementation of Znn.com across seven different machines (Figure 5): znns0-3 are the four content servers running Apache v2.2.16, znndb is a common backend database running mySql v14.14d5.1.61, from which the different servers extract the contents, and znnproxy is the proxy machine that runs the load balancing software (Apache running mod_proxy_balancer v2.1). The controller is deployed in a separate machine (znnmaster). All machines run Debian Linux v6.0.4, and have 512MB of memory. Moreover, an additional machine znnclient running JMeter v2.5.1 generates the traffic during the execution of the system.

To build the changeload used for our experiments, we identified the:

1. Workload and operating conditions for our change scenarios characteristic of a slashdot-type effect, based on a sample collected by Juric [11], previously used for a general evaluation of the effectiveness of Rainbow in Znn.com [6]. In this case, scenarios have been scaled down to a duration of five minutes, which is enough to drive the controller through its different operational stages and apply the robustness tests.

2. Set of probes used during the analysis, planning, and execution stages of the controller. For the sake of simplicity, in this particular case we chose to deploy a configuration of Znn.com that only uses the *response time probe* (type ClientProxyProbeT) . Still, this probe is used through the three operational stages, and since our goal is not to carry out a complete evaluation of Rainbow, this is enough to demonstrate the approach.

3. Set of changes to be applied on our set of probes. We identified 32 mutation rules listed in Table 1 that are applicable to the response time probe (response time is a float with domain [0-Float.MAX_VALUE]).

5.3. Experimental Results and Discussion

Each change scenario of the changeload results from combining the workload and operating conditions with a single change type based on an applicable mutation rule. In our changeload, each mutation rule gives way to three change scenarios (*i.e.,* applied during the analysis, planning, and execution stages, respectively), which are triggered in the time instant in which the controller enters the corresponding stage, and their duration is permanent. Overall,

Mutation Rule	**Failure Mode (A=Abort, S=Silent)**					
	Analysis		**Planning**		**Execution**	
	A	S	A	S	A	S
MsgNull	1	1	1	1	1	1
MsgEmpty		1		1		1
MsgPredefined		1		1		1
MsgNonPrintable		1		1		1
MsgAddNonPrintable		1		1		1
TSEmpty		1		1		1
TSRemove		1		1		1
VNRemove		1		1		1
VVRemove		1		1		1
VVInvalidFormat		1		1		1
VVNumAbsoluteMinusOne		1		1		1
VVNumMax		1		1		1
VVNumMin		1		1		1
VVNumMaxPlusOne		1		1		1
VVNumMinMinusOne		1		1		1
VVNumMinRangeMinusOne		1		1		1
TOTAL	1	16	1	16	1	16

Table 3. Robustness issues uncovered by the experiments

we run 96 robustness tests using our experimental setup (32 applicable mutation rules × 3 controller operational stages).

Table 3 details the experimental results obtained from the tests that apply the change scenarios based on each of the identified applicable mutation rules at each one of the controller stages. To begin with, 48 out of the 96 conducted tests uncovered robustness issues (50%). Moreover, one of the first observations that can be made is that no catastrophic, restart, nor hindering failures have been identified during the tests. Although no catastrophic, restart, or hindering failures have been identified during our tests, these failure modes are still needed, as they portray relevant behaviours of the controller. Regarding abort failures, only tests based on the mutation MsgNull have uncovered a single failure in each one of the controller stages. Concretely, this consisted in the raising of the same unhandled java.lang.NullPointerException in each controller stage. In particular, the exception was produced during the parsing of the probe's response with a regular expression matcher. It is worth mentioning that additional unhandled exceptions have been detected during the course of the experiments. However, these have not been considered in the results table, since they have been originated outside of the controller (concretely, on the response time probe itself).

Silent failures are by far the most frequent failure type discovered during the tests. As it can be observed, mutations that pertain the overall probe response message and the variable value (first and fourth group in Table 3, respectively) present the highest concentration of silent failures. In contrast, mutations that concern timestamps and variable names present silent failures only in cases in which the concrete element is removed (mutations TSEmpty, TSRemove and VNRemove). This is a consequence of the way in which the Rainbow master processes inputs from the probes. Messages sent from the probes are parsed in such a way that only the presence of a variable name and a timestamp in the message is assessed, but their concrete values are not checked syntactically nor semantically. However, this does not prevent the correct update of values in the architectural model of the system inside of the controller, which uses a unique probe identifier to update the value in the correct place in spite of incorrect variable names or timestamps in probe input. Moreover, all the observed silent failures correspond to incorrect updates of values in the architectural model which are not acknowledged by the controller in any way. However, it is worth mentioning that **the issues discovered in the different controller stages are different**. In particular, while the response time value is updated with null values in tests conducted during the analysis stage, in the planning and execution stages, the last valid value on the model becomes frozen when the mutation rule is applied on the probe, and this can lead to completely different effects when considering the ensemble controller plus system.

Summarizing, although in general terms Rainbow master is fairly robust, experimental results have shown that our approach has been able to uncover a relevant set of robustness issues in the controller. Although in this particular case the identified pattern of robustness issues at the different stages of the controller differs only to a limited extent, this can be attributed to the particular architecture of Rainbow, which uses its model manager as a safeguard for the logic in the rest of the components used throughout the different operational stages. Moreover, the obtained results align with previous research, which has shown that robustness testing may disclose a small number of different issues, despite of their potentially high relevancy to the particular system being tested [17].

6. Conclusions

In this paper, we have presented a novel approach that enables the stateful evaluation of controllers for self-adaptive software systems. This has been achieved by defining how the controller's interface should be tested according to the target system's changeload and the operational stage of the controller. Results were characterised using an adapted version of the CRASH scale, and the approach was validated in the context of Rainbow framework for architecture-based self-adaptation and the Znn.com case study. Experimental results have shown that our approach is able to uncover a relevant set of faults in the controller that might hinder the resilience of the self-adaptive system.

The main limitation uncovered in our approach has been the inability of simulating some of the failure modes of the CRASH scale. This might be related to the architectural robustness of the controller, the restricted observability of the controller's internal behaviour (we rely mainly on the logs being generated by Rainbow rather than on tailored error detectors), the non-removal of the faults associated with the errors detected (which might prevent exercising other parts of the code), and the simplicity of the case study (which prevents exercising the entire functionality of Rainbow).

As for future work, in addition of solving the above mentioned limitations, there are several lines of research that could be exploited based on the groundwork setup by this paper. First, we need to employ different controllers and additional case studies for assessing our approach in terms of its efficiency in uncovering faults in the controller of a self-adaptive software system. Second, while the focus of this paper was the evaluation of the controller, there is the need for considering the self-adaptive system in its entirety, and this would inevitably lead to new challenges, such as, the necessity to consider the full state of the target system when evaluating the robustness of the entire system, *i.e.*, the controller plus the target system. Finally, in the long term, since the structure of a self-adaptive software system

is expected to evolve during run-time, one should be able to perform the type of evaluation described in this paper at run-time rather than development-time.

Acknowledgements

Co-financed by the Foundation for Science and Technology via project CMU-PT/ELE/0030/2009 and by FEDER via the "Programa Operacional Factores de Competitividade" of QREN with COMPETE ref.: FCOMP-01-0124-FEDER-012983.

References

[1] R. Almeida and M. Vieira. Changeloads for resilience benchmarking of self-adaptive systems: a risk-based approach. In *Proceedings of EDCC 2012*.

[2] J. Andersson, R. Lemos, S. Malek, and D. Weyns. Modeling Dimensions of Self-Adaptive Software Systems. In: *Software engineering for self-adaptive systems*, LNCS volume 5525, pages 27–47. Springer, 2009.

[3] Y. Brun et al. Engineering self-adaptive systems through feedback loops. In: *Software Engineering for Self-Adaptive Systems*, LNCS volume 5525, pages 48–70. Springer, 2009.

[4] J. Cámara and R. de Lemos. Evaluation of Resilience in Self-Adaptive Systems Using Probabilistic Model-Checking. In *Proceedings of SEAMS 2012*, pages 53–62. IEEE, 2012.

[5] B. H. Cheng et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: *Software engineering for self-adaptive systems*, LNCS volume 5525, pages 1–26. Springer, 2009.

[6] S.-W. Cheng, D. Garlan, and B. R. Schmerl. Evaluating the Effectiveness of the Rainbow Self-Adaptive System. In *SEAMS*, pages 132–141. IEEE, 2009.

[7] D. Cotroneo, D. Di Leo, R. Natella, and R. Pietrantuono. A case study on state-based robustness testing of an operating system for the avionic domain. In: *Computer Safety, Reliability, and Security*, LNCS volume 6894, pages 213–227. Springer, 2011.

[8] R. de Lemos et al. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In *Software Engineering for Self-Adaptive Systems 2*, LNCS volume 7475, pages 1–32, Springer, 2013.

[9] D. Garlan, S.-W. Cheng, A.-C. Huang, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Computer*, 37(10):46–54, 2004.

[10] J. Gray. *Benchmark Handbook: For Database and Transaction Processing Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[11] M. Juric. Slashdotting of mjuric/universe. `http://www.astro.princeton.edu/~mjuric/universe/slashdotting/`, 2004.

[12] K. Kanoun and L. Spainhower. *Dependability Benchmarking for Computer Systems*. Wiley-IEEE CS, 2008.

[13] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36:41–50, 2003.

[14] P. Koopman and J. DeVale. Comparing the robustness of POSIX operating systems. In *29th International Symposium on Fault-Tolerant Computing*, pages 30–37, 1999.

[15] P. Koopman and J. DeVale. Comparing the robustness of posix operating systems. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, FTCS '99, pages 30–, Washington, DC, USA, 1999. IEEE CS.

[16] J.-C. Laprie. From Dependability to Resilience. In *DSN Fast Abstracts*. IEEE CS, 2008.

[17] N. Laranjeiro, M. Vieira, and H. Madeira. Experimental robustness evaluation of JMS middleware. In *IEEE International Conference on Services Computing (SCC 2008)*, pages 119–126. IEEE CS, July 2008.

[18] H. Madeira. Towards a security benchmark for database management systems. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, pages 592–601, Washington, DC, USA, 2005. IEEE CS.

[19] A. Mukherjee and D. Siewiorek. Measuring software dependability by robustness benchmarking. *Transactions on Software Engineering*, 23(6):366–378, 1997.

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

[21] M. Rodríguez, F. Salles, J.-C. Fabre, and J. Arlat. MAFALDA: microkernel assessment by fault injection and design aid. In *3rd European Dependable Computing Conference*, pages 143–160. Springer, 1999.

[22] M. Rodríguez, F. Salles, J.-C. Fabre, and J. Arlat. Mafalda: Microkernel assessment by fault injection and design aid. In *Proceedings of the Third European Dependable Computing Conference*, EDCC-3, pages 143–160, London, UK, UK, 1999. Springer.

[23] S. Salva and I. Rabhi. Stateful web service robustness. In *Fifth International Conference on Internet and Web Applications and Services*, Barcelona, Spain, May 2010. Iaria.

[24] M. Vieira, N. Laranjeiro, and H. Madeira. Benchmarking the robustness of web services. In *13th IEEE Pacific Rim Dependable Computing Conference (PRDC 2007)*, pages 322–329, Melbourne, Victoria, Australia, Dec. 2007. IEEE.