Robustness Evaluation of the Rainbow Framework for Self-Adaptation

Javier Cámara
Carnegie Mellon University
Pittsburgh, USA
jcmoreno@cs.cmu.edu

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

Nuno Laranjeiro
University of Coimbra
Coimbra, Portugal
cnl@dei.uc.pt

Rafael Ventura
University of Coimbra
Coimbra, Portugal
ventura@dei.uc.pt

Marco Vieira
University of Coimbra
Coimbra, Portugal
mvieira@dei.uc.pt

ABSTRACT

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. Despite the growing importance and criticality of controllers in many application domains, a central concern about them is the difficulty in assessing their robustness when architecting self-adaptive systems. In previous work, we proposed an approach for evaluating the robustness of controllers in self-adaptive systems. In this practical experience report, we describe a comprehensive evaluation of the robustness of a particular controller, in our case Rainbow, in the context of two case studies: a benchmark case study that reproduces the typical infrastructure for a news website, and an industrial middleware for monitoring populated networks of devices. The aim of this work is to assess to what extent the use of a different target system has an impact on the robustness of the controller, which has to be customized in different ways, and may need to resort to the activation of different features, depending on the particular target system. Our analysis concludes that the customization of Rainbow (the controller) has little impact on its robustness because of the way the controller was designed and built, and this modularization of non-functional requirements is indeed encouraging when architecting self-adaptive systems.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Self-adaptive systems, robustness testing, Rainbow

1. INTRODUCTION

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 [6, 9]. 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. These controllers consist usually of four distinct operational stages, namely, monitoring, analysis, planning and execution [12], that implement the traditional *sense-plan-act* architectures. Despite the growing importance and criticality of controllers in many application domains, 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* [14]). 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. 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 [1]. These factors need to be taken into account if assurances need to be provided about the services to be delivered by the target system.

In previous work, we proposed an approach for evaluating the robustness of controllers in self-adaptive systems, aiming at the effective identification of design faults [5]. In this paper, we assess the impact of the target system (*i.e.*, the system under control) upon the perceived robustness of the controller, which may need to be customized in different ways and resort to the activation of different features, depending on the particular target system. To achieve this, we carry out a comprehensive evaluation of the robustness of a controller, *i.e.*, Rainbow [11], an architecture-based approach that enables self-adaptation, in two different case studies: a full-fledged deployment of Znn.com, a benchmark case study that reproduces the typical infrastructure for a news website [11], and Data Acquisition and Control Service (DCAS), an industrial middleware for the monitoring

WOODSTOCK '97 El Paso, Texas USA

of highly populated networks of devices [4]. As a result of our analysis, we conclude that the customization of Rainbow has little impact on its robustness because of the way the controller was designed and built. Based on this, we can claim that, when architecting self-adaptive systems we are able to make reasonable assumptions regarding the robustness of the controller (at least in the case of Rainbow) since its robustness is not altered depending on the target system.

In the rest of this paper, Section 2 provides an introduction to the Rainbow framework, and summarizes the two case studies under consideration. In Section 3, we describe the experimental approach followed for the robustness evaluation of a controller (*i.e.*, Rainbow) for self-adaptive software systems. Section 4 presents the experimental results obtained from our study. Section 5 describes related work. Section 6 presents some conclusions and future work.

2. BACKGROUND

In this section, we briefly describe the Rainbow framework, and present the two case studies under consideration.

2.1 The Rainbow Framework

In this paper, we focus on Rainbow [11], an architecture-based platform for self-adaptation, which provides a substantial base of reusable infrastructure through customization, which aims at reducing 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.

Architecture models used in Rainbow employ a component-and-connector style of architecture descriptions made in the ACME ADL [10], which include relevant information about how the different architecture elements are connected in the system, as well as of their relevant properties. These models enable high-level reasoning about the structure and state of the system, in particular with the aim of determining the best course of action for its adaptation.

The framework defined by Rainbow includes mechanisms for (Figure 1): monitoring a target system and its environment (using the observations reported by probes 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 appropriate changes through system-level effectors. Rainbow's component-and-connector architecture model of the target system is one of the main elements used in its decision-making process.

The main components of the framework are:

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

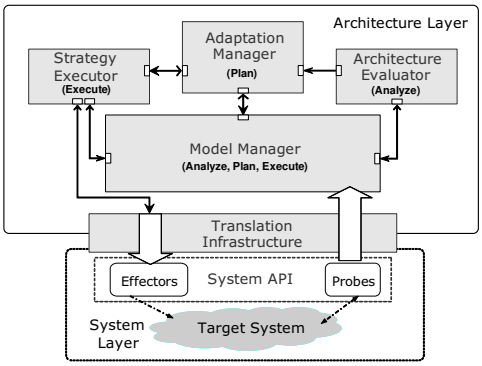

Figure 1: The Rainbow framework

2.2 ZNN.COM

Znn.com [7] is a case study portraying a representative scenario for the application of self-adaptation in software systems which has been extensively used to assess different research works in the self-adaptive systems community. Znn.com is able to reproduce the typical infrastructure for a news website, and has a three-tier architecture consisting of a set of servers that provide contents from backend databases to clients via front-end presentation logic (Figure 2). The system uses a load balancer to balance requests across a pool of replicated servers, the size of which can be adjusted according to service demand. A set of clients makes stateless requests, and the servers deliver the requested contents.

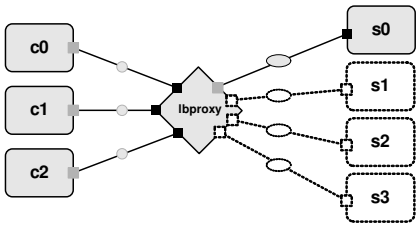

Figure 2: Znn.com system architecture

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, which is associated with the number of active servers.

In Znn.com, when response time becomes too high, Rainbow is able to increment server pool size if it is within budget to improve performance; or switch servers to textual mode (start serving text content) if cost is near budget limit.

2.3 Data Acquisition and Control Service (DCAS)

The Data Acquisition and Control Service (DCAS) [4] from Critical Software, is a middleware that provides a reusable

infrastructure to manage the monitoring of highly populated networks of devices. The middleware is integrated with Critical's Energy Management System (csEMS)[1], a platform that supports the operation of control centers for green power producing companies (*e.g.*, wind, solar, etc).

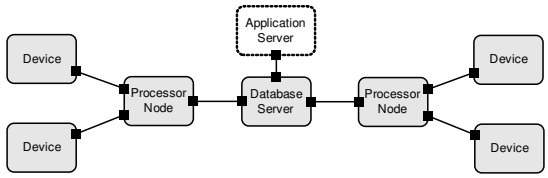

Figure 3: Architecture of a DCAS-based system

The building blocks in a DCAS system (Figure 3) are:

- **Devices** are equipped with one or more sensors to obtain data from the application domain (*e.g.*, from wind towers, solar panels, etc.). Each one of these sensors has an associated *data stream* from which data can be read. There may be different types of devices connected to the network, each type with its particular characteristics (*e.g.*, protocols, type of data, etc.). Each type of device has an associated *device profile* specifying the data polling rate and the expected value ranges for the data being collected.
- **Processor nodes** pull data from the devices at a given rate (configured in the device profile), and dispatch this data to the database server. Each processor node includes a set of processes called *Data Requester Processor Pollers* (DRPPs) responsible for retrieving data from the devices. Communication between the DRPPs and the devices is synchronous, so the DRPP remains blocked until the device responds to a request for data or a timeout expires. It is worth observing that this is the main performance bottleneck of DCAS.
- **Database server** stores the data pulled from devices.
- **Application server** is connected to the database server to obtain data, which can be presented to human operators or processed by application software. However, DCAS is application-agnostic, so the application server is not discussed in the remainder of this paper.

The main objective of DCAS is collecting data from the connected devices at a rate as close as possible to the one configured in their device profiles, while making an efficient use of the computational resources in the processor nodes. This is achieved by two adaptation mechanisms: (i) *rescheduling*, which decreases the priority of devices which fail to respond in a timely manner, so that they are polled less often (thus reducing the average time that DRPPs remain blocked waiting for device data), and (ii) *scale up*, which (de)activates DRPPs as required to exploit as much as possible the resources of processor nodes.

3. EXPERIMENTAL APPROACH

This section provides an overview of the approach used for evaluating the robustness of a controller in a self-adaptive software system, followed by a description of the design and setup for the experiments performed.

[1] http://solutions.criticalsoftware.com/products_services/csems/

3.1 Controller Robustness Evaluation

Our proposal for evaluating the robustness of a self-adaptive software system considers the model depicted in Figure 4. The *environment* consists of all non-controllable elements that determine the operating conditions of the system (*e.g.*, hardware, network, 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 required, *(iii)* applying a sequence of control actions through system-level *effectors*.

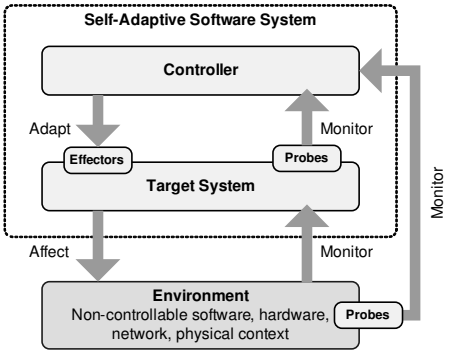

Figure 4: Self-adaptive software system

In a nutshell, the intent of the approach is evaluating how robust is the controller regarding changes that may affect its interface by modifying the probes' inputs into the controller. We consider that the controller has a stateful nature, since for the same input, the controller's internal state may influence its output. This is accounted for by considering input mutation during the different operational stages of the controller (*i.e.*, analysis, planning, execution) to create an appropriate context for evaluating its robustness.

In the remaining part of this subsection, we provide an overview of the key elements of our approach.

3.1.1 Changeload

The changeload is a set of representative change scenarios [3], where changes are based on controller input mutations according to a set of predefined *mutation rules* (Table 1). Mutation rules have been defined based on previous works on robustness testing [13, 19, 21], and explore limit conditions that are typically the source of robustness problems. Moreover, mutation rules affect the different parts of the input provided to the controller by a probe, which typically consists of: *(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. To build the changeload, we: *(i)* identify the workload and operational conditions of the system necessary to drive the controller through its different stages, *(ii)* identify the set of probes used during each controller stage, *(iii)* identify the set of applicable mutation rules for each probe based on the characteristics of the

probe, data type, and value range for the variable it updates, and *(iv)* combine the workload, operation conditions, and mutation rules to obtain change scenarios. For every tested probe and for each applicable mutation rule, we need to consider three different change scenarios related to the analysis, planning, and execution controller stages.

3.1.2 Failure mode classification

Characterizes the run-time behavior of the controller while the target system is running in the presence of the changeload. Specifically, the robustness of a controller for a self-adaptive system can be classified according to an adapted version of the CRASH scale [13] including the following failure modes:

- *Catastrophic*: the whole controller crashes or becomes corrupted (this might include the OS or machine on which the controller runs). No output is produced.
- *Restart*: the controller 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.
- *Abort*: abnormal behavior in the controller occurs due to a run-time exception inside of the controller.
- *Silent*: the controller fails to acknowledge an error, for instance by signalling an exception, which causes the controller to continue operating improperly.
- *Hindering:* the controller fails to return a correct error code, which may hinder error recovery. This differs from a silent failure, since the error is acknowledged by the controller but the returned error code is incorrect.

3.1.3 Robustness tests

Robustness tests mutate the input of the different probes (Figure 5). For each probe (which, at run-time is continuously delivering information to the tested controller) we apply a single change for each data sample provided by the probe. However, we apply (in the subsequent probe data samples) the same change for a given period of time, which gives us the possibility of further disturbing the system.

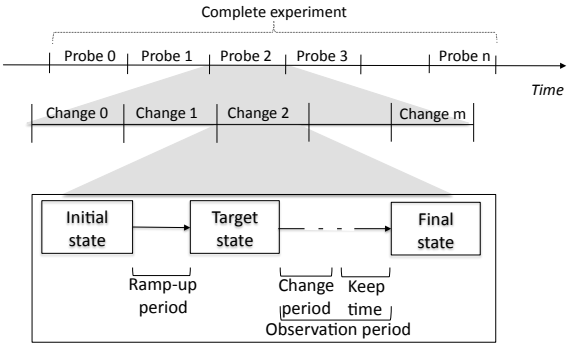

Figure 5: Robustness testing procedure

Each robustness test focuses on a single type of mutation rule, and is executed for each of the operational stages in the controller (analysis, planning, execution), since it enables us to cover more cases and potentially disclose more robustness problems. Hence, in each test we drive the system from an initial state to a target state by delivering a workload to the system for a given amount of time (*ramp-up period* in Figure 5). 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 stage, we start applying the changes (of the same type) during a *change period* in which the controller is on the target stage. This time period should be set to the typical time required for a transition to occur between the target controller stage to the next stage. After this probe mutation period there is a *keep time* which is the time required for the system to reach a final state, ending the test, and coinciding with the completion of the controller's execution stage. The keep time should at most be set to the worst case execution duration found in the specification of the adaptation to be executed. The *observation period* (change period+keep time) is used to register any deviations from expected controller behavior.

Table 1: Mutation rules for probes

	Rule Name	Description
A. Message	1. MsgNull	Replace by null value
	2. MsgEmpty	Replace by empty string
	3. MsgPredefined	Replace by predefined string
	4. MsgNonPrintable	Replace by string of non-printable characters
	5. MsgAddNonPrintable	Add non-printable characters
	6. MsgOverflow	Add characters to overflow max string size
B. Timestamp	1. TSEmpty	Replace by empty timestamp
	2. TSRemove	Remove timestamp
	3. TSInvalidFormat	Replace by timestamp with invalid format
	4. TSDateMaxRange	Replace date by maximum valid
	5. TSDateMinRange	Replace date by minimum valid
	6. TSDateMaxRangePlusOne	Replace date by maximum valid plus one
	7. TSDateMinRangeMinusOne	Replace date by minimum valid minus one
	8. TSDateAdd100	Add 100 years to date
	9. TSDateSubtract100	Subtract 100 years from date
	10. TSInvalidDate	Replace by invalid date
C. Var. Name	1. VNRemove	Remove variable name
	2. VNSwap	Replace by different valid variable name of same type
	3. VNSwapType	Replace by different valid var. name of different type
	4. VNInvalidFormat	Replace by var. name with invalid format
	5. VNNotExist	Replace by inexistent var. name
D. Var. Value	1. VVRemove	Remove variable value
	2. VVInvalidFormat	Replace by invalid format value
	Number	
	3. VVNumAbsoluteMinusOne	Replace by -1
	4. VVNumAbsoluteOne	Replace by 1
	5. VVNumAbsoluteZero	Replace by 0
	6. VVNumAddOne	Add 1
	7. VVNumSubtractOne	Subtract 1
	8. VVNumMax	Replace by maximum type value
	9. VVNumMin	Replace by minimum type value
	10. VVNumMaxPlusOne	Replace by maximum number valid for type plus one
	11. VVNumMinMinusOne	Replace by minimum number valid for type minus one
	12. VVNumMaxRange	Replace by maximum number valid for variable
	13. VVNumMinRange	Replace by minimum number valid for variable
	14. VVNumMaxRangePlusOne	Replace by maximum number valid for variable plus one
	15. VVNumMinRangeMinusOne	Replace by minimum number valid for variable minus one
	Boolean	
	16. VVBoolPredefined	Replace by predefined value

3.2 Experiment Design

To build the changeload used for our experiments, we

identified the: workload and operating conditions, set of probes used during the different stages of the controller, and set of changes (based on mutation rules) applied to the set of probes identified in each case.

3.2.1 Workload and operating conditions

- *Znn.com.* Workload and conditions characteristic of a slashdot-type effect, based on a sample collected by Juric [2], previously used for an evaluation of the effectiveness of Rainbow in Znn.com [7]. Scenarios have been scaled down to a duration of 5 minutes, which is enough to drive the controller through its different operational stages and apply the robustness tests.

- *DCAS.* Workload and conditions in a typical deployment of a DCAS-based system. Scenarios have a duration of 5 minutes, which is enough to drive the controller through its different operational stages and apply the robustness tests. All experiments incorporate a workload that includes 100 data streams with a data polling rate of 1 second. Each experiment drives the controller towards the triggering of adaptation to improve performance, and conforms to the following pattern: (a) 50s of normal activity to let the system achieve a steady state; (b) 200s of disturbance, during which we induce low responsiveness in data streams (adding a 2-second delay in the response time of 25% of the data streams); and (c) 50s of normal activity.

3.2.2 Sets of probes

The three last columns of Table 2 indicate the set of probes used during the analysis (A), planning (P), and execution (E) stages of the controller for each case study. Probes were identified by inspecting architecture models and adaptation strategies. Specifically, the use of a probe during the analysis stage can be determined by checking whether the constraints specified for the architecture model are defined over variables updated by a given probe. Moreover, an analogous process is followed to identify probes used during the planning stage, which update information in variables used to specify applicability conditions of adaptation strategies. Finally, probes used during execution are identified by inspecting the predicates included in the code of the adaptation strategies.

3.2.3 Sets of changes to be applied on probes

Table 2 also indicates the set of non-applicable mutation rules to each of the probes in the two case studies, which are determined by the type of probe implemented, as well as by the data type and value range of the variables they update. Regarding probe implementation type, all the studied probes are implemented in Java, except for the ServerLoad and ServerFidelity probe types in Znn.com, which are implemented in Perl. In both cases, the length of strings is unrestrained, therefore mutation rule MsgOverflow (A6) is not applicable. In Perl probes, the null datatype does not exist, hence mutation MsgNull (A1) cannot be used. Regarding data types, all of the studied probes update numerical variables, disallowing the applicability of rule VVBoolPredefined (D16). The only exception is the ServerLoad probe type in Znn.com, which is not associated with a simple datatype and reports a message with a custom format in the variable value, preventing the use of mutation rules D3-16. Finally, the variables updated by some of the probes do not have a value range explicitly defined. In probe types RPS, QueueSize, ActivePollers, EllapsedTime and ClientProxy there is an implicit lower bound of zero, due to the semantics of the information in the variable (*e.g.*, negative times make no sense), but there is no upper bound, discarding the use of rules D12 and D14 that involve the maximum value in range. In QueueStatus and ServerLoad probes there are not even implicit upper or lower bounds for the value range, therefore in these cases rules D13 and D15 are also unapplicable.

[2] http://www.astro.princeton.edu/universe/slashdotting/

3.3 Experimental Setup

3.3.1 Znn.com

We deployed Rainbow and the corresponding implementation of Znn.com across seven different machines (Figure 6): znns0-3 are the four content servers running Apache v2.2.16, znndb is a common backend database running mySql v14.1, 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 RAM. An additional machine znnclient running JMeter v2.5.1 generates the traffic during the execution of the system.

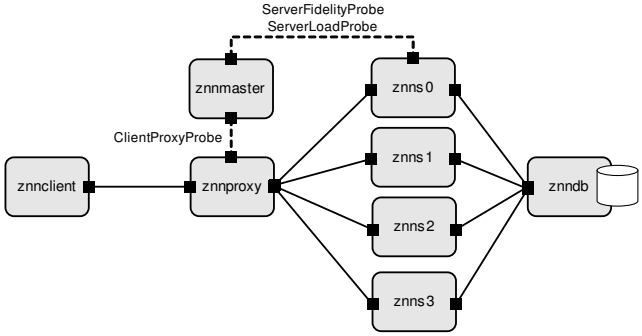

Figure 6: Znn.com experimental setup

3.3.2 DCAS

We deployed Rainbow and DCAS across four machines (Figure 7): dcas-db acts as the backend database running on Oracle 10.2.0, dcas-main acts as a processor node, running DCAS, and (dcas-devs) is used to simulate the response of network devices from which DCAS retrieves information. Moreover, the controller (Rainbow's master) is deployed in a separate machine (dcas-master). All machines run on Windows XP Pro SP3 (DCAS is deployed as a Windows service), and an Intel Core i3 processor, with 1GB of RAM.

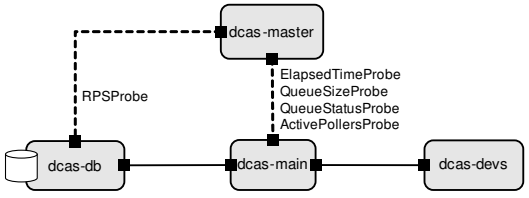

Figure 7: DCAS experimental setup

Table 2: Probe use per controller stage and applicable mutation rules for DCAS and ZNN

	Probe Type	Description	Non-applicable mutation rules	A	P	E
DCAS	RPS	Measures performance (processed data requests per second inserted in database).	A6,D12,D14,D16	x	x	
	QueueSize	Data request queue size in processor node.	A6,D12,D14,D16	x	x	x
	QueueStatus	Data request queue growth rate in processor node.	A6,D12-16			x
	ActivePollers	Number of active Data Requester Processor Pollers in processor node.	A6,D12,D14,D16		x	
	ElapsedTime	Measures device response time when polled for data.	A6,D12,D14,D16	x	x	
ZNN	ClientProxy	Measures experienced response time in proxy.	A6,D12,D14,D16	x	x	x
	ServerLoad	Measures the load of a given server.	A1,A6,D3-16	x		x
	ServerFidelity	Reports the fidelity level of the contents in a server.	A1,A6,D16		x	x

4. EXPERIMENTAL RESULTS

This section presents and discusses the outcome of the experiments carried out on the two case studies.

4.1 Experimental Results for Znn.com

Table 3 details the experimental results obtained for the Znn.com case study. For this case study, 108 out of the 209 conducted tests uncovered robustness issues (51.6%). None of these robustness issues included catastrophic, restart, or hindering cases. Specifically, only 2.7% of the issues uncovered correspond to abort failures, which only occur on tests based on the mutation MsgNull (in this case, in the ClientProxy probe type, which is the only one implemented in Java). Specifically, this abort case consists of the same unhandled java.lang.NullPointerException in each of the three stages of the controller during the parsing of probe response with a regular expression matcher. The remaining 97.3% of the cases are different silent failures, which mostly correspond to incorrect updates (or the lack thereof) of property values in the architecture model of the target system which are not acknowledged by the controller. In probes implemented in Perl (ServerLoad and ServerFidelity), when incorrect input is received by the controller, the update is ignored in all cases, and the property in the model is not updated. In contrast, in the Java probe (ClientProxy), properties are updated with clearly incorrect values (such as negative values in the case of ClientProxy with mutations VVNumAbsoluteMinusOne or VVNumMin), or not updated in some other cases (*e.g.*, mutations MsgNonPrintable or VNRemove).

4.2 Experimental Results for DCAS

Table 4 details the experimental results obtained for DCAS. To begin with, 160 out of the 295 conducted tests uncovered robustness issues (54.2%). Out of all the failures, only 5.3% correspond to abort, whereas the remaining 94.7% are silent failures. Regarding abort failures, only tests based on the mutation MsgNull caused this type of failure by raising the unhandled java.lang.NullPointerException in the different probes tested, which are all implemented in Java. In the case of silent failures, all probes present failure patterns similar to the one described for the Java probe ClientProxy in Znn.com. In general, there are either incorrect or no property updates in architecture model properties in the presence of exceptional input provided by the probe. However, there are some slight differences in patterns, such as in the RPS probe, which presents additional failures in the presence of mutations VVNumMax and VVNumMin, due to the data type of the variable it updates (floating point vs. integers in the rest of the probes). Specifically, the values of Float.MAX_VALUE and Float.MIN_VALUE are transliterated into the probe response message in exponential notation, causing the incorrect parsing of the message upon reception in the controller. The other probe that presents a slightly different failure pattern is QueueStatus, which is not affected by mutations that modify the value of the variable with negative numbers (*e.g.*, VVNumAbsoluteMinusOne or VVNumMin).

4.3 Discussion

Our approach for robustness testing was able to discover a relevant number of failures in both case studies ($\gtrsim 50\%$ of tests uncovered at least one robustness issue). However, none of these failures fell into the categories of catastrophic, reboot, or hindering [3]. Moreover, most failures identified in both case studies ($> 90\%$) correspond to silent failures and follow similar patterns across the different probes. Specifically, mutations that pertain the overall probe response message, as well as variable name and value (first, third, and fourth groups in Tables 3 and 4, respectively) present the highest concentration of silent failures. In contrast, silent failures concerning timestamps occur only when it is removed (mutations TSEmpty and TSRemove). This is a consequence of the way in which Rainbow parses messages from probes, checking only the presence of a timestamp in the message without further syntactic nor semantic checks.

Despite the similar failure patterns across probes, we have observed that there are slight differences related with their type of implementation: *(i)* all abort failures are given when mutating Java probes, and *(ii)* silent failures when mutating Perl probes always stop the updates of properties in the architecture model, in contrast with Java probes, in which incorrect updates of properties can also appear.

Moreover, regardless of the similar failure patterns for the same probe across different controller stages, the specific failure instances discovered in the different controller stages are different. An instance of this is the mutation of Java probes with the MsgNull, which results in the properties of the architecture model being updated with null values in tests conducted during the analysis stage, whereas in the planning and execution stages the last valid value on the model freezes when the mutation is applied on the probe.

Finally, the similarity of the failure patterns observed in the two case studies, in which the customization of Rainbow was carried out by two independent teams, indicates that the customization of Rainbow for different target systems has little impact in the robustness of the resulting controller. This is a consequence of Rainbow's architecture, in which the model manager component acts as a safeguard for the rest of the components that handle the different stages of the feedback control loop. Concretely, these other components use the information as updated in the architecture model, in contrast with using input directly supplied by probes.

[3]Despite this, these failure modes portray relevant behaviors that might be uncovered when testing other controllers.

Table 3: Experimental results for Znn.com

	Failures (A=Abort, S=Silent)																	
	Analysis						Planning						Execution					
	ClientProxy		ServerLoad				ClientProxy		ServerFidelity				ClientProxy		ServerFidelity		ServerLoad	
Mutation Rule	A	S	A	S			A	S	A	S			A	S	A	S	A	S
A1. MsgNull	1	1					1	1					1	1		1		
A2. MsgEmpty		1		1				1		1				1		1		1
A3. MsgPredefined		1		1				1		1				1		1		1
A4. MsgNonPrintable		1		1				1		1				1		1		1
A5. MsgAddNonPrintable		1		1				1		1				1		1		1
B1. TSEmpty		1		1				1		1				1		1		1
B2. TSRemove		1		1				1		1				1		1		1
C1. VNRemove		1		1				1		1				1		1		1
C2. VNSwap				1						1						1		1
C4. VNInvalidFormat				1						1						1		1
C5. VNNotExist				1						1						1		1
D1. VVRemove		1		1				1		1				1		1		1
D2. VVInvalidFormat		1		1				1		1				1		1		1
D3. VVNumAbsoluteMinusOne		1						1		1				1		1		
D8. VVNumMax		1						1						1				
D9. VVNumMin		1						1		1				1		1		
D10. VVNumMaxPlusOne		1						1		1				1		1		
D11. VVNumMinMinusOne		1						1		1				1		1		
D15. VVNumMinRangeMinusOne		1						1		1				1		1		
TOTAL/PROBE	1	16	0	12			1	16	0	18			1	16	0	18	0	12
TOTAL/STAGE	**A: 1, S: 28**						**A: 1, S: 34**						**A: 1, S: 46**					

Table 4: Experimental results for DCAS

	Failures (A=Abort, S=Silent)																	
	Analysis						Planning								Execution			
	RPS		QueueSize		ElapsedTime		RPS		QueueSize		ElapsedTime		ActivePollers		QueueSize		QueueStatus	
Mutation Rule	A	S	A	S	A	S	A	S	A	S	A	S	A	S	A	S	A	S
A1. MsgNull	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
A2. MsgEmpty		1		1		1		1		1		1		1		1		1
A3. MsgPredefined		1		1		1		1		1		1		1		1		1
A4. MsgNonPrintable		1		1		1		1		1		1		1		1		1
A5. MsgAddNonPrintable		1		1		1		1		1		1		1		1		1
B1. TSEmpty		1		1		1		1		1		1		1		1		1
B2. TSRemove		1		1		1		1		1		1		1		1		1
C1. VNRemove		1		1		1		1		1		1		1		1		1
C2. VNSwap						1				1		1		1		1		1
C4. VNInvalidFormat		1		1		1		1		1		1		1		1		1
C5. VNNotExist		1		1		1		1		1		1		1		1		1
D1. VVRemove		1		1		1		1		1		1		1		1		1
D2. VVInvalidFormat		1		1		1		1		1		1		1		1		1
D3. VVNumAbsoluteMinusOne		1		1		1		1		1		1		1		1		
D8. VVNumMax		1						1										
D9. VVNumMin		1		1		1		1		1		1		1		1		
D10. VVNumMaxPlusOne		1		1		1		1		1		1		1		1		
D11. VVNumMinMinusOne		1						1										
D15. VVNumMinRangeMinusOne		1		1		1		1		1		1		1		1		
TOTAL/PROBE	1	18	1	16	1	17	1	18	1	17		17	1	17	1	17	1	13
TOTAL/STAGE	**A: 3, S: 51**						**A: 4, S: 69**								**A: 2, S: 40**			

5. RELATED WORK

Robustness testing consists in stimulating a system with erroneous input conditions to trigger internal errors. This enables testers to differentiate systems according to the number and type of errors uncovered, providing developers with information to solve the identified problems [17].

Ballista [13] 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. Each operating system is classified in terms of its robustness and according to a predefined scale (the CRASH scale [13]) that distinguishes several failure modes.

MAFALDA (Microkernel Assessment by Fault injection AnaLysis and Design Aid) [19] is a tool that enables the characterization of the behavior of microkernels in the presence of faults injected in the parameters of system calls.

Robustness testing techniques have been applied also at the middleware layer and targeting different types of systems. Robustness testing of high availability middleware is discussed in [16]. The paper presents a testing framework that integrates previous testing techniques (*e.g.*, scenario-based testing and test result classification). The case study conducted on OpenAIS (an open implementation of the Application Interface Specification -AIS-) showed that simple techniques can identify robustness problems.

Ballista was also adapted to middleware systems. Concretely, [18] studies the robustness of various CORBA ORB implementations. The failure modes were adapted to better characterize the CORBA context and the authors were able to reveal several issues in the middleware.

In [15], an experimental approach for the robustness evaluation of three JMS middleware providers is presented. The study exposes serious problems, highlighting the importance of applying robustness testing to real-world systems.

The abovementioned works implement robustness testing approaches that do not consider the state of the system under test. In [8] 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 since they are able to cover more cases when compared to the traditional approaches.

In [5] we present an approach to evaluate the robustness of

controllers for self-adaptive software systems, aiming at the identification of design faults. The approach is based on robustness tests that provide mutated inputs to the interfaces between controller and target system (*i.e.*, probes).

6. CONCLUSIONS

In this paper, we have reported on our experience evaluating the robustness of controllers for self-adaptive software systems. For that, we have applied the Rainbow framework, which is an architecture-based approach for supporting self-adaptation both to a benchmark case study that reproduces the typical infrastructure for a news website, and an industrial middleware for monitoring device networks.

The similarity in the failure patterns observed in our results enabled us to conclude that the customization of the Rainbow framework for different target systems has little impact in the robustness of the resulting controller. This is a result of the architecture of Rainbow, where the model manager component acts as a safeguard for the rest of the components that handle the different stages of the feedback control loop. Specifically, these other components make use of the information as updated in the architecture model, in contrast with using input directly supplied by probes.

These results indicate that some assumptions regarding the robustness of the resulting controller can be made when developing self-adaptive systems through the customization of a reusable framework, such as Rainbow. It is worth observing that this claim is only validated by our results in the context of Rainbow and that its generalization would require experimentation with further self-adaptive systems including different types of controllers. However, although there have been some recent works in reusable frameworks that aim at obtaining controllers through customization in self-adaptive software systems [20, 2], none of them are mature nor widely available. To the best of our knowledge, Rainbow is the only of such frameworks evaluated in the context of different real-world case studies [7, 4].

Regarding future work, we aim at developing a framework for resilience evaluation of self-adaptive software systems by applying our technique for evaluating the robustness of controllers, basing upon previous work on resilience evaluation presented in [3]. This will enable us to explore how robustness issues in the controller influence the resilience of the overall self-adaptive system.A second direction consists in extending our robustness evaluation approach into the internal components of the controller that implement the MAPE-K loop. The idea is to test the interfaces between its components, in contrast with just focusing on the interface between the controller and the target system.

7. ACKNOWLEDGMENTS

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.

8. REFERENCES

[1] J. Andersson et al. Modeling dimensions of self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems*, LNCS, pages 27–47. Springer, 2009.

[2] R. Asadollahi et al. Starmx: A framework for developing self-managing java-based systems. In *SEAMS*, pages 58–67. IEEE, 2009.

[3] J. Cámara et al. Architecture-based resilience evaluation for self-adaptive systems. *Computing*, 95(8):689–722, 2013.

[4] J. Cámara et al. Evolving an Adaptive Industrial Software System to Use Architecture-based Self-Adaptation. In *SEAMS*, pages 13–22. IEEE, 2013.

[5] J. Cámara et al. Robustness Evaluation of Controllers in Self-Adaptive Software Systems. In *LADC*, pages 411–420. IEEE, 2013.

[6] B. H. Cheng et al. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, LNCS, pages 1–26. Springer, 2009.

[7] S. Cheng et al. Evaluating the Effectiveness of the Rainbow Self-Adaptive System. In *SEAMS*, pages 132–141. IEEE, 2009.

[8] D. Cotroneo et al. A case study on state-based robustness testing of an operating system for the avionic domain. In *SAFECOMP*, volume 6894 of *LNCS*, pages 213–227. Springer, 2011.

[9] R. de Lemos et al. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In *Software Engineering for Self-Adaptive Systems 2*, number 7475 in LNCS. Springer, 2013.

[10] D. Garlan et al. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, chapter 3, pages 47–67. Cambridge University Press, 2000.

[11] D. Garlan et al. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, 37(10):46–54, 2004.

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

[13] P. Koopman and J. DeVale. Comparing the robustness of POSIX operating systems. In *FTCS*, pages 30–37. IEEE CS, 1999.

[14] J.-C. Laprie. From Dependability to Resilience. In *DSN Companion Volume*. IEEE, 2008.

[15] N. Laranjeiro et al. Experimental robustness evaluation of JMS middleware. In *SCC*, pages 119–126. IEEE, 2008.

[16] Z. Micskei et al. Robustness testing techniques for high availability middleware solutions. In *Int. Workshop on Engineering of Fault Tolerant Systems*, 2006.

[17] A. Mukherjee and D. Siewiorek. Measuring software dependability by robustness benchmarking. *IEEE Trans. Software Eng.*, 23(6):366–378, 1997.

[18] J. Pan et al. Robustness testing and hardening of CORBA ORB implementations. In *DSN*, pages 141–150. IEEE CS, 2001.

[19] M. Rodríguez et al. MAFALDA: microkernel assessment by fault injection and design aid. In *EDCC*, volume 1667 of *LNCS*, pages 143–160. Springer, 1999.

[20] G. Tamura et al. Improving context-awareness in self-adaptation using the dynamico reference model. In *SEAMS*, pages 153–162. IEEE, 2013.

[21] M. Vieira et al. Benchmarking the robustness of web services. In *PRDC*, pages 322–329. IEEE, 2007.