Dissertation

Masters in Informatics Engineering

A Case study on Architecture-based Resilience Evaluation for Self-Adaptive Systems

Pedro Gonçalo Roque Correia
pcorreia@student.dei.uc.pt

Advisor:
Javier Cámara
Co-Avisor:
Prof. Marco Vieira

03-07-2013
Department of Informatics Engineering
Faculty of Science and Technology
UNIVERSITY OF COIMBRA

Abstract

One of the most promising approaches to face the increasing complexity of software systems is the use of self-adaptation, in order to enable software systems to deal with changes themselves, autonomously. It is presented as one of the means by which it is possible to provide systems that are scalable, support dynamic modifications and rigorous analysis, capable to respond to resource variability or user needs modifications, still, being flexible and robust.

Normally, by design, the methods for self-adaptation are at the system's source code or network level, but recently, architecture-based methods have been widely considered as more promising approaches.

One of the main barriers for greater implementation of architecture-based self-adaptation is the lack of evidence of the advantages and compensation of applying it in systems with built-in adaptation mechanisms.

A recent proposal to cope with this challenge uses an architecture-based approach which evaluates alternative adaptation mechanisms of a self-adaptive system by comparison, based on the identification of representative system and environmental conditions which may have a relevant impact on system resilience.

The present work has one major contribution: evaluate if the application of architecture-based self-adaptation can improve the resilience of an already adaptive system. The effectiveness of the above-mentioned approach is demonstrated by using Rainbow, an architecture-based platform for self-adaptation, and DCAS, an industrial software-intensive system used to monitor and manage highly populated networks of devices in renewable energy production plants.

The experimental evaluation showed that the application of architecture-based self-adaptation improved the resilience of the tested system. The overall runtime quality of the self-adaptive system can be greatly improved with acceptable costs.

Keywords

Self-adaptive systems, Resilience evaluation, Adaptation mechanisms, Dependability, Architecture-based self-adaptation.

Acknowledgments

I would like so start by thanking to my advisor Javier Cámara because without his support this work would not be possible. He proved to be an excellent advisor providing me with excellent advices and guidelines, always being patient and available to help, and above all transforming all errors committed in constructive criticism.

I also want to thank Professor Marco Vieira which gave me the opportunity to enter the field of scientific research, always providing support and suggestions to improve the quality of the work.

To all my laboratory and university colleagues, I would like to thank for all the fantastic moments that we already shared and will be sharing in the future.

Finally I have to thank with all my heart for the amazing family that I have, so physically distant in some points, but always extremely united. Especially I want to express my gratitude to my parents and brother. Without their backup would not be half the man I am today.

To my uncle Zé Barreto,

"Tio,

Obrigado pela pessoa que foste e que acima de tudo ainda és e sempre serás para mim! Foste um grande exemplo como Homem, Pai, Tio, Amigo. Sempre defendeste os teus ideais com unhas e dentes e mantendo a tua humildade e o teu orgulho nas tuas raízes humildes eras admirado e seguido por todos....

Sinto a tua falta....
não...
Todos sentimos a tua falta....
Sei que estarás sempre ao meu lado a dar aquele empurrão para seguir em frente ou aquele abraço
que sempre me fez tão bem!
Obrigado por tudo....."

List of Publications

This thesis is based on work presented in the following papers[1]:

1. J. Camara, P. Correia, R. de Lemos, D. Garlan, P. Gomes, B. Schmerl, e R. Ventura, «Evolving an Adaptive Industrial Software System to Use Architecture-based Self-Adaptation», SEAMS '13 Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems

[1] Publication available at http://dl.acm.org/citation.cfm?id=2487342

Table of Contents

Chapter 1 Introduction 14

1.1. Contributions.. 16

1.2. Thesis Structure.. 17

Chapter 2 Background and Related Work 18

2.1. The need for self-adaptation... 18

2.1.1. Tradittional maintenance and evolution approaches 18

2.1.2. Runtime adaptation 19

2.1.3. Self-adaptive systems.. 20

2.1.4. Architectural models 20

2.1.5. Adaptation strategies classification 21

2.1.6. Architecture-based self-adaptation challenges 23

2.2. Resilience evaluation.. 24

2.2.1. Definition of resilience 24

2.2.2. Fault Injection.. 25

2.2.3. Assessment Approaches... 26

Chapter 3 DCAS-Rainbow integration.. 28

3.1. Project ADAAS (Assuring Dependability in Architecture-based Adaptive Systems) 28

3.2. Data Acquisition and Control Service (DCAS).. 29

3.2.1. Objectives... 29

3.2.2. Architecture.. 29

3.2.3. Adaptation mechanisms.. 31

3.3. The Rainbow framework 33

3.4. Rainbow-DCAS Integration.. 34

3.4.1. Translation Infrastructure 34

3.4.2. Evolution of DCAS system 36

3.4.3. Scale Out implementation... 38

3.4.4. Integration conclusions 38

Chapter 4 Resilience Evaluation 40

4.1. Evaluation approach.. 40

4.1.1. Operational Profiles and Scenarios 41

4.1.2. Criteria Definition 42

4.1.3. Experimention and evaluation 42

4.2. Experimental evaluation 44

4.2.1. Changeload Identification 45

4.2.2. Runtime stimulation.. 47

4.2.3. System Resilience Evaluation 49

Chapter 5 Conclusions 60

References... 61

List of Figures

Fig. 1 - DCAS architecture overview 31
Fig. 2 - Rainbow framework.. 33
Fig. 3 - DCAS-Rainbow translation infrastructure.. 35
Fig. 4 - Non-conventional operational profile 42
FIg. 5 - Conventional operation profile 43
Fig. 6 - Experimental setup: Rainbow-DCAS (left) and DCAS (right) 44
Fig. 7 - Performance 48
Fig. 8 – Cost... 48
Fig. 9 - Performance 49

List of Tables

Table 1 System changes... 47
Table 2 Conventional Operation Profile - Rainbow DCAS 51
Table 3 Conventional Operation Profile - Original DCAS 52
Table 4 Non-Conventional Operation Profile - Original Dcas........................... 53
Table 5 Non-Conventional Operation Profile - Rainbow Dcas.......................... 54

Chapter 1
Introduction

Software systems are commonly developed based on a static approach. In other words, the system’s architecture is created based on static requirements and design decisions [1]. The decision process is carried out in most cases at design time, trying to predict the possible environments in which the software will run, as well as the available resources and the workload to which the system will be subjected. However, more and more of these software systems are running in highly uncertain and rapidly changing environments, normally requiring human supervision to continue operation in all conditions. Moreover, many types of systems (e.g., service-oriented, cloud-based) are becoming increasingly complex, leading all the maintenance tasks to become high costly and time-consuming procedures.

Most of the software systems (especially industrial) are designing to be extremely stable in order to be constantly reliable in a production environment, meaning that the system usually is not designed to be self-adaptive. This feature makes almost impossible for the users to apply changes in the system or tune its behavior in runtime.

The first approaches to tackle these problems consisted on simple mechanisms tightly coupled to the system’s source code (e.g., exceptions, fault tolerant protocols). These mechanisms are often highly specific to the application and as result, they were not able to reduce associated costs in building or modifying them, providing little more than local treatment of system faults.

To tackle this situation, a promising approach is run-time adaptability, that is, endowing systems with the ability to respond to changes at runtime in an autonomous manner, adapting successfully to subsequent changes in their runtime environment in order to maintain normal operation [2]. IBM’s Autonomic Computing [3] [4] initiative was one of the first successful proposals to address this concern, with the introduction of a self-adaptive layer responsible of managing the target system. This approach relies on a closed control loop known as the MAPE-K loop [3] (Monitoring, Analyzing, Planning, and Executing, through the use of a Knowledge base that informs the different activities). This loop can be used for the management of resources exposing sensor and effector interfaces. The control loop monitors the system’s state and its components, as well as the execution context, and identifies relevant changes that may prevent the system from achieving its goals and providing its intended service. As the next activity the system triggers the planning of potential alternative adaptations in order to respond to the new changes on environment and system conditions, executes them, and monitors that its goals are being achieved once again, if possible, without any interruption. All these stages make use of a common knowledge that guides them and that may be enriched by the experience earned during execution.

More recent approaches [5][6] using external self-adaptation also adopt the traditional control loop theory, since it has been used and proved to be an effective solution [7]. Additional studies describe framework-based approaches that provide reusable mechanisms to monitor and apply changes in the system dynamically [8][9], but these existing approaches still presented some flaws to resolve such as: still very high costs in adding external control to a system and limited reusability across systems

[10]. In order to respond to this problematic and improve the reuse of adaptation infrastructure across different systems and environments, some approaches were presented [5][11][12][13]. Some of these frameworks [5][13] resort to the use of architectural models which allowed the evolution from a previous static decision-making process to a dynamic process, where all the decisions can be carried out at run-time. This provides flexibility and reduces costs regarding operation and maintenance.

In spite of major advances in self-adaptive systems, building them in a predictable manner (i.e. fulfilling all the requirements and objectives) is a major engineering challenge that remains to be tackled. How to ensure that the application of self-adaptation mechanisms in a system under different environmental conditions will produce the expected results without any adverse effects (e.g., that the system behavior will not deteriorate instead of improving)? Can the system recover from a failure in a critical component within an acceptable period of time [14]? Summarizing, there is an important need to assess the ability of a self-adaptive system to recover in a timely manner when subjected to changes that influence negatively the provision of service, therefore guaranteeing that the system is persistently dependable in time despite changes either in the system itself, its environment, or its goals (i.e., that it is resilient [15]) .

In this work we evaluate the resilience of a particular kind of self-adaptive system, using an industrial software-intensive system as a case study. Concretely, this dissertation aims at answering the question: Does applying architecture-based self-adaptation improve the resilience of an already adaptive system?

To answer this question, we rely on an architecture-based approach which evaluates alternative adaptation mechanisms of a self-adaptive system by comparison, based on the identification of representative system and environmental conditions which may have a relevant impact on system resilience, which are used as the basis to establish a comparison. To apply architecture-based self-adaptation in the context of this work was necessary to create a prototype of our tested system, with its own new adaptations mechanisms and compare these mechanisms with the original built-in adaptation mechanisms of the tested system. By evaluating the results from the original adaptations against the new created adaptations (architecture-based) is how it is possible to conclude if the system's resilience is improved or not.

Concretely, this approach is applied in the context of the integration between DCAS (Data Acquisition and Control Service), an industrial middleware developed by Critical Software, and the Rainbow framework, which provides a reusable infrastructure for architecture-based self-adaptation.

DCAS aims at providing a reusable infrastructure to manage monitoring and (non-automatic) control of highly populated networks of devices. The main objective of DCAS is collecting data from the connected devices at a rate as close as possible to the one retrieved from the devices configuration, supporting at the same time as many connected devices as possible.

The prototype of our tested system was created using the Rainbow framework [5] This framework purpose is to decrease engineering effort by providing an explicit representation of adaptation knowledge. Rainbow provides the support for architecture-based self-adaptation mechanisms trough the following features: explicit architecture model of the target system, a collection of adaptation strategies, and utility preferences to guide adaptation.

1.1. **Contributions**

The goal of this dissertation work is to access if the application of architecture-based self-adaptation improves the resilience of an already adaptive system.

The contribution of this dissertation can be divided in two main objectives:

Rainbow-DCAS Integration:

- **Abstraction of existing adaptations from the system:** being a system already in use, several adaptations mechanisms are implemented inside DCAS. These mechanisms purpose is to maintain the performance of the system under different loads, responding to failing devices, the addition of new devices and modification in data rates. In some specific cases these mechanisms proved to be slow in recovering the system's performance. Our first challenge is removing these mechanisms and developing a prototype of DCAS (Rainbow-DCAS) in which different adaptation mechanisms for the system are implemented using Rainbow. These mechanisms are created to improve the system’s performance by providing a better and faster recovery.

Resilience Evaluation:

- **DCAS adaptations vs. Rainbow-DCAS adaptations:** compare the adaptations created using the Rainbow framework with the existing adaptations embedded within DCAS to assess whether the use of architecture-based self-adaptation really improves the results of the original adaptation mechanisms in DCAS.

As a secondary objective to complement the work performed after completing the main tasks:

- **Implementation of a sophisticated adaptation mechanism currently not implemented in DCAS (Scale out) using Rainbow**: support the deployment of several instances of the service within the same system if necessary. Currently, Scale Out can only be carried out manually by a human operator.

1.2. **Thesis Structure**

The document contents are divided in chapters, organized as follows.

Chapter 2 presents some background and related work on the relevant areas to this work. The main topics are background on self-adaptive software and resilience evaluation of software systems.

Chapter 3 presents a more detailed description of the context in which the work was developed, more concretely the main project that is the origin of this work: project ADAAS (Assuring Dependability in Architecture-based Adaptive Systems). The chapter includes a detailed description of the integration between the two main components of the work: The Rainbow platform, used to apply architecture-based self-adaptation, and DCAS, the software system used as a case study.

Chapter 4 presents the work on assessing the system resilience by comparing the new adaptation mechanisms, created using Rainbow with the original DCAS's adaptations. The chapter includes a detailed description of the approach and its experimental evaluation.

Finally, chapter 5 concludes this thesis and briefly introduces the future work.

Chapter 2
Background and Related Work

2.1. **The need for self-adaptation**

The successful launch of a software system can lead to unexpected changes in the desired path for the evolution of the system. It may bring new user demands and requirements that imply changes and improvements in the system. The opposite is also possible. The software may not get the desired success or contains serious faults and errors that were not detected during its implementation. These reasons make it almost inevitable the activity of constant maintenance and software evolution. Software maintenance was earlier defined as "the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment" [16].

An early survey [17] showed that around 75% of the maintenance effort is spent in adaptive (changes in the software environment) and perfective (new user requirements) maintenance while corrective maintenance consumed only about 21%. Later studies [18][19] confirmed the same results.

Nowadays, software systems have to deal with growing and highly dynamic and unpredictable environments, also gathered with new and more complex system requirements. Traditional approaches for the development and management of software systems are thus confronted with a very hard challenge to solve.

2.1.1. Tradittional maintenance and evolution approaches

Commonly, software maintenance activity was accomplished in two different ways.

- The first way was to be conducted by human operators and is referred as preventive maintenance (prevent problems in the future). Human supervision was used in order to maintain the intended system configuration in all conditions [20]. However, this approach reached its limits as systems become more complex, thus more difficult to maintain and understand. At the same time, human supervision entailed high costs.

- In the second technique. the software system suffers modification to the source code, introducing error-handling code (e.g. exception handling, timeouts) directly due either to a problem or a need to improve the system (referred as corrective maintenance) [13][21].This technique can already be classified as a self-adaptation technique, being classified as an **internal adaptation**. These error-handling techniques have the advantage of trapping an error at the moment of detection, and also are well supported by modern programming languages (e.g., Java exceptions) and run-time libraries (e.g., timeout function for jQuerys). However, these techniques present several weaknesses. They result in very complex interactions and are tightly coupled to the source code (e.g., exceptions, fault tolerant protocols). They showed to be useful for handling local adaptations; however, in general, it is necessary a global vision and information about the system and several actions/events to interact simultaneously in order to perform system adaptation.

2.1.2. Runtime adaptation

The lack of success of the aforementioned techniques lead to a key research challenge: Discovering new approaches for developing software that could be more easily and reliably changed during runtime. Also, more recent developed software systems, including some known as safety-critical systems [22] presented as a critical requirement continuous availability. A. G. Ganek et al. [23] presents a good example where one hour with offline service causes a loss of 1 Million USD volume of sales within bank industry and 2.8 Million USD within the energy industry.

This research lead to the development of new approaches capable of enabling systems with the ability to reconfigure their structure and behavior at runtime [24][25][26][27] in order to perform repairs or improve their operation without any human intervention.

Gupta et al. [26] describes a formal framework for studying on-line software version adaptation, applied at the “statement-and procedure-level”. An online software replacement system replaces parts of the software while it is in execution, thus eliminating the shutdown. The technique is based on locating the program control points at which all variables affected by the adaptation are guaranteed to be redefined before use. They show that in general case locating all such control points is almost impossible and still based on analyzing techniques very connected to the source code and requires knowledge from the provider. Another problem is to apply this approach in large systems developed with complex programing languages. Languages such as Lisp and Smalltalk support this approach and provide the intended flexibility, but reduce the performance of the system. Furthermore, application behavior and dynamism are mixed and as result concerns regarding dynamic adaptation are crossed with system design, making the adaptation management extremely difficult.

Reynolds, J,H. [27] uses ACHOO (Automated Code Handler for Output Operations) for the runtime creation of code for printing simulation output. ACHOO supports a methodology whereby output requirements for a computer program, and the source code for the statements to support them, are determined at execution time based on user-supplied input. Thus, the user can tailor output requirements to a specific need or scenario. This activity, consisting of inserting calls to subroutines that ACHOO creates, eliminates the need for manual creation of output statements and accompanying format specifications during computer program development and subsequent maintenance.

Several other approaches were applied in real-world systems:

- fault-tolerant hardware [28] to cope with hardware failures;
- “hot pluggable” devices [29], to add capacity or replace faulty units without power cycling a machine;
- programming languages [30] and their runtimes to dynamically load, verify, and invoke code updates.
- system virtualization [31][32] to attain hardware fault isolation and improve resource utilization;

- tuning of operating system parameters to achieve optimal memory, CPU, and device utilization among application components [33];

2.1.3. Self-adaptive systems

A more recent approach has emerged with the goal to develop a new type of software systems. This kind of systems, which typically operate using an explicit representation of their structure and goals, has been studied within different research areas of software engineering (e.g., component-based development, requirements engineering, software architectures, etc.) and not only can recover from internal causes (e.g. failure) but are also able to deal with system and environment changes (e.g. increasing requests from users). Such systems are referred to as autonomic systems, self-adaptive systems, self-* computing, and self-healing systems [3][34][35] although for this work we use only one of the most general and popular among these names, namely self-adaptive systems.

One of the earliest approaches related to self-adaptive systems is autonomic systems and in fact, the terms are commonly used interchangeably within the self-adaptive systems community. The “self” prefix indicates that the systems decide autonomously (i.e., without or with minimal interference from human operators) about how to adapt or organize to accommodate changes in their contexts and environments. This reflects the vision of autonomic computing in which systems respond to change by evolving in a self-managed manner while running and providing service [3][36][37].

The key idea in the self-adaptation is the creation of a closed-loop system with a feedback loop aiming to adjust itself to changes during its operation. IBM’s Autonomic Computing [2] initiative proved to be a major breakthrough being the first approach presented to address this concern, with the introduction of a self-adaptive layer responsible for managing the target system. This approach relies on a closed-loop control known as the MAPE-K loop [3]. The four stages of the MAPE loop are enabled by knowledge combining assumptions and specification of the system. This knowledge, updated continually through environment and system monitoring, helps analyze whether the requirements specified by the user continue to be satisfied. When they are no longer satisfied, appropriate system changes are planned and executed automatically. Later studies proved the advantages in the use of this approach [38].

2.1.4. Architectural models

Although the work discussed over the years has provided much of what is useful in contributing towards self-management, it has not yet resolved some of the general and fundamental issues in order to provide a comprehensive and integrated approach.

One of the critical design issues when using a self-adaptive approach is how to model the system in order to provide the correct information for the control layer to determine when problems exist and choose an effective repair strategy. Several models for dynamic adaptation were presented along the time, mainly being classified into two different categories: Architecture style-based models such as *CHAM* [39] and *graph grammars* [40], and architecture description language (ADL) based models, such as *ACML* [41] and *Dynamic Wright* [42].

Several models approaches were presented, but there is one who stood out from the rest. Researchers and practitioners have discovered that architectural models are particularly well-suited for complex systems adaptation [13][35][43] as it brings several potential benefits:

- Software engineers can use the system's architecture as a tool to describe, understand, and reason about overall system behavior [44]. Leveraging the engineer's knowledge at this level of the system design holds can be an improvement in helping manage runtime adaptation.
- It gives the possibility, in principle, to select the system components responsible for problem detection and resolution of concerns and modify and extend adaptation mechanisms to be reused across different systems.
- The control over adaptation's application policy and scope can be made based on an understanding of the application requirements and semantics. Previous approaches to runtime adaptation either impose a single policy to be adopted by all systems or failed to separate application-specific functionality from runtime change considerations. As a result, concerns over runtime adaptation permeate system design.

Several studies prove the advantage of use of architecture-based self-adaptation [5][13][45][46], and in the work proposed below, we use one of these approaches as our starting point: Rainbow [5].

2.1.5. Adaptation strategies classification

One major issue in self-adaptation is related to the process of selection and creation of repair strategies. How to make the control layer able to create the connection cause-action in a given context? When presented with several strategies to apply, how to choose the best option among all available? Will the chosen strategy deteriorate instead of improve the system behavior? What action to take when the situation changes while applying a specific strategy? Should the available number of strategies be fixed from the beginning or can new strategies be created and added during runtime? These are difficult questions, and to date there has been relatively little work to answer them systematically.

When deciding how to apply the strategies, two main factors have to be considered: the **approach** and the **type** of the adaptation.

The **approach** to follow can be identified into two subsets:

- *Static/Dynamic*: The approaches based on static definition of strategies provide a set of predefined strategies with a fixed control structure and simple applicability conditions used to choose the best suited strategy. Dynamic software has the ability to adapt continuously in response to changes in the application objectives and the environment in which the software operates and to modify their architecture and enact the modifications during the system's execution.

Static strategies are known at design time and the conditions under which a strategy can be executed are well-specified. Thus, strategies can be revised and analyzed to determine their intended effects, and corrected. However, this approach has the major drawback that such repair strategies tend to be inflexible. When confronted with unexpected conditions for repair, it is possible that none of the known strategies will be able to adapt the system accordingly.

Dynamic change, which occurs while the system is operational, is far more demanding and requires that the system evolves dynamically, and that the adaptation occurs at runtime. These approaches have the reverse benefits and drawbacks; they manage to be extremely flexible, but do not allow knowing beforehand what strategies will actually be executed.

- *Internal/External: Internal* approaches are based on programming language features, such as conditional expressions, and exceptions. A good example is the second technique described in section 2.1.1. Error-handling code (e.g. exception handling, timeouts). In *External* approaches it is an external adaptation manager that contains the adaptation strategies. The manager implements the adaptation logic, mostly with the aid of a policy engine, or other application-independent mechanisms. ´

As described before, *internal* approaches result in very complex interactions and are tightly coupled to the source code (e.g., exceptions, fault tolerant protocols). They showed to be useful for handling local adaptations; however, in general, it is necessary a global vision and information about the system and several actions/events to interact simultaneously in order to perform system adaptation. However, the *external* approach has the advantage of allowing the customization of adaptations in order to be used across different systems.

Another important facet is the **type** of adaptation.

- *Close/Open*

A close-adaptive system has only a fixed number of adaptive actions, and no new behaviors and alternatives can be introduced during runtime. On the other hand, in open adaptation, self-adaptive software can be extended, and consequently, new alternatives can be added, and even new adaptable entities can be introduced to the adaptation mechanism

- *Model-Based/Free*

In *model-free* adaptation, the mechanism is not aware of the model for the environment and the system itself. The adaptation mechanism adjusts the system using the knowledge on the requirements, goals, and alternatives; On the other hand, in model-based adaptation the mechanism uses a model of the system and its context.

- *Specific/Generic*

Some of the existing solutions address only *specific* domains/applications, such as a database. However, *generic* solutions are also available, which can be configured by setting policies, alternatives, and adaptation processes for different domains.
This type addresses **where** and **what** concerns in addition to **how**, because the specific type only focuses on an adaptation of attributes of a particular part of the software system.

2.1.6. Architecture-based self-adaptation challenges

The pursued objective of self-adaptive systems is to create software capable of both self-adaptation to changes in its operating environment and continual verification of its requirements compliance.

It is important to note that with any type of architectural modification, concerns regarding the adaptation mechanisms must be separated from the effects of the adaptation on the particular application. An imprudent application of architectural modifications can compromise the integrity of the system. As result, such adaptations must be verified before being applied to a running system. It is imperative to maintain the correspondence between the architectural model and implementation of the system in order to ensure that architecture-based adaptations properly effect the system. Another important requirement is the provision of the necessary implementation infrastructure for runtime evolution facilities.

It is also necessary to develop a repertoire of techniques that provides timely reaction to detected system or environment changes by having both flexibility and predictability. These strategies will also have to be flexible enough so that they can be reused and applied in different systems. The use of architectural modeling and analysis tools is crucial in this regard, but dynamic generation of adaptations can be a significant performance concern, especially when dealing with a system's time-critical needs; this is further magnified if changes to the system's state cannot be treated in isolation and instead adaptations must be re-generated every time.

The work presented in this document addresses a concrete case study that combines the best of both approaches, providing both flexibility and predictability. While this work builds on past experience in self-adaptive systems – most notably the use of architectural models – it provides a key missing ingredient that is necessary for the success of the overall approach. *The provision of assurances that the system is resilient against changes that may occur either in the system or its environment at runtime.*

2.2. **Resilience evaluation**

One of the major engineering challenges in this research area is how to build self-adaptive systems in a cost-effectively and predictable manner. The goal is to provide self-adaptive systems with the ability to recover in useful time when subjected to changes. Laprie [15] defined resilience as the persistence of service delivery that can justifiably be trusted, when facing changes.

Before analyzing the existing approaches for resilience evaluation in computer systems it is necessary to clarify the possible meanings for resilience and what attributes and measures can be used to properly assess it.

2.2.1. Definition of resilience

The use of the term "resilience" has become more frequent in recent years in the information technology area, often being connected to other terms like "dependability", "security" and the RAMS (reliability, availability, maintainability and safety) concept. Different areas can have different approaches for resilience, depending on the intended evaluation, the metrics used and the measurement process.

Going back to the Latin origin, resilience means to rebound, recoil or return to the original form. Different areas and works provide several variations around the resilience theme. For instance in Physics, it is related to objects that are invulnerable to the impact of external forces. In Chemistry it is the capacity of a metal to return to its original form. In Engineering, resilience is a measure of a material's capacity to withstand impact, as well as to absorb and release energy through elasticity [47]. In Psychology it refers to a capacity to function in immensely demanding settings, as well as the ability to cope with stress [48]. Meanwhile, in Ecology, resilience has been used to measure the ability of an ecosystem to absorb change, continue to function and evolve [49].

It is obvious that there is plenty of scope for different approaches to what we mean by resilience, but today, resilience tends to be used to either mean a capacity to 'bounce back' or, more conservatively, a tendency to resist change. Narrowing the area of interest to the area of information and communication technologies, the term resilience is commonly used to describe a more flexible and dynamic approach to achieve dependability. Resilience is defined as "the ability to deliver, maintain, and improve service when facing threats and evolutionary changes" [50].

To evaluate the resilience of a system we need to ensure that the system is capable of being dependable through time. Dependability can be defined as "the ability to deliver service that can be justifiably trusted" [15], or the ability of a system to avoid service failures that are considered to be more severe and frequent from what is expected or considered as acceptable. Then we can affirm that **dependability** is a key requirement to achieve resilience.

Randell et al. [51] and J-C Laprie [15] consider that to evaluate the dependability of software systems, it is required to measure how reliably the system can be, the level of availability and the capacity to support modifications and repairs.

In section 2.1 we identified three main types of systems, regarding the system's ability for adaptation.

The first type of systems is non-adaptive systems. Initially, the existing approaches to assess the system's dependability dealt well with this type of systems that are relatively closed and unchanging. Basically repairing or evolving the software system usually required for the system to be shut down and then restarted. However, each time a new version of the system was created it was necessary to perform dependability evaluation all over again, turning the process of updating the software a very long one.

For these systems the majority of the conducted studies in dependability performed this evaluation during the development and validation phases of the software system. Guangsing Xu et al. [52] present manual software watermarking as method to fight software piracy. A manual watermark is inserted by the programmer of the application, rather than a using a third-party automatic tool.

Then, with the rapid evolution in software systems complexity, came the necessity of endowing these systems with the capability to adapt during runtime.

A good example of this evolution can also be related software watermarking. This task evolved from a manual operation to an semi-automatic operation where a programmer inserts markers into a program during development and the finished software is then augmented by a software watermarking tool(e.g. Sandmark [53]). A dynamic watermarking approach was introduced by C.Colberg et al. [54].

Although performed in different ways, the resilience of a watermark was tested against the same metrics: watermark should be **robust** - that is, resilient to semantics preserving transformations (such as optimizations).

2.2.2. Fault Injection

According to the dependability definition [15] changes here may refer to unexpected threats to the dependability of the system: failures, errors, faults.

A **failure** can be described as an event that occurs when de delivered service diverts from correct service, either because it does not comply with the functional specification, or because this specification did not adequately describe the system function.

Since a service is a sequence of the system's external sates, a service failure means that at least one (or more) external state of the system's deviates from the correct service state. The deviation is called an **error**. The possible cause of an error is called a **fault**. In most cases, a fault first causes an error in the service state of a component that is a part of the internal state of the system and the external state is not immediately affected. For this reason, the definition of an error is the part of the total state of the system that may lead to its subsequent service failure. If the fault leads to an error that it is considered as active otherwise is dormant.

Then one of the possible way to test the system resilience is by fault injection to detect if may lead to failures. The use of this fault injection [55] approach is quite common on self-adaptive systems for resilience evaluation and other metrics like security and system performance [56][57][58][59].

The purpose of injecting faults is to create emerging scenarios in which systems must run continuously and be capable of adapting autonomously the moment the conditions for adaptation are reached. Three major conditions for adaptation are recognized: system errors, changes in the environment, and changes in user preferences. Understanding these different conditions for self-adaptation directly affects the development of capabilities for measuring, modeling, and controlling the target system to support self-adaptation. These conditions share the common property of being a change that may not have been anticipated thus providing opportunities for runtime improvements to bring the system back within the boundaries of its requirements under the newly encountered conditions.

2.2.3. Assessment Approaches

Nowadays, software systems are commonly used in safety-critical areas and applications (safety-critical software systems), turning to be very complex and highly independent systems that provide essential services in our daily routine. This type of systems present as one of the major requirements, to be constantly available during time, thus implied that the system cannot be shutdown.

To better understand the conditions for self-adaptation it is necessary to focus on system properties that can be expressed quantitatively and require quantitative verification (such as reliability, performance, and energy consumption).

The analysis of the system properties can be performed by two different ways: either **modeling the system**, or **performing direct measurements** when the system is already implemented.

In the architecture-based approaches for self-adaptation we already have a clear representation of the system in the architecture model. However in some cases, performing an accurate analysis implies relying in massive state-space models with high complexity and detail. Many existing analysis techniques rely on generating stochastic models of the system such as continuous-time Markov chains (CTMCs). They are able to capture various functional and stochastic dependencies among components and allow evaluation of various measures related to dependability and performance. These kinds of models can result in huge state spaces and computation time in not treated properly. Some analysis techniques were created in order to respond to the problem created by the increasing complexity of the models, by making use of efficient representation mechanisms. One of these techniques is known as symbolic model-checking [60] and supports various forms of (qualitative and quantitative) analysis.

One possible approach to verify system properties is quantitative analysis. Quantitative evaluation techniques have been mainly used to evaluate the impact of accidental faults on system dependability. In quantitative analysis, users define a finite mathematical model of a system and analyze the model's compliance with system requirements that are expressed formally in temporal logics [61][62] extended with probabilities and costs/rewards. Example requirements established through this analysis include the probability that a fault occurs within a specified time period and the expected response time of a software system under a given workload. The quantitative analysis of performance requirements can be performed using discrete-time Markov chains (DTMCs) to model behavioral aspects of the system, and probabilistic computation tree logic, or PCTL [63], to formalize requirements. Several tools using this “symbolic” approach, such as PRISM [64], SMART [65] and CASPA [66] are already successfully implemented.

Direct measurement, is used generally in systems that are already in use and refer to observations of systems in the operational phase. The main advantage in the results obtained is that they are collected under realistic operation conditions and environment, using a concrete workload, instead of a mere approximation. This represents a very important source of information when studying resilience properties, such as robustness, availability or dependability.

Some examples of direct measurement studies were performed based on software fault injection [67][68].

In our work, we also analyze system response due to fault injection. An approximate model is derived which enables one to account for the failures due to the design faults in a simple way when evaluating the system's dependability.

The approach used during our work focuses on quantitative analysis using direct measurements, since we already have the system implemented, and focuses on providing levels of confidence with respect to the self-adaptive capabilities of the system.

Chapter 3
DCAS-Rainbow integration

Before describing the process of integration between the architecture-based platform for self-adaptation (Rainbow), and the case study (DCAS), it is important to explain the project which is the starting point for the work presented in this document.

In section 3.1 is presented a more detailed explanation about the purpose of the project ADAAS[2] (Assuring Dependability in Architecture-based Adaptive Systems) and how it led to the selection of the two main components of this work. In section 3.2 and 3.3 is presented a more detailed analysis, respectively, on the Rainbow platform, used to apply architecture-based self-adaptation, and DCAS, the software system used as a case study.

The final section of this chapter contains a detailed description on the integration process and the steps carried out for the abstraction of the original adaptations and implementation of the Rainbow-based adaptations.

3.1. Project ADAAS (Assuring Dependability in Architecture-based Adaptive Systems)

The aim of the ADAAS project is to improve dependability and optimize performance in large-scale software systems, while reducing development and operational costs. To reach these goals, it resorts to the use of runtime analysis of architectural models, combined with analysis techniques which allow dealing with anticipated changes, concluding if the system goals are being achieved, and also endowing the system with the generation of adaptation strategies at runtime to respond to unanticipated changes.

The chosen approach was divided in 4 main boosts:

- Strategy language and platform: the definition of a language suitable for expressing adaptation strategies and able to address key quality attributes, like dependability.
- Support for analysis: techniques and tools for quantitative and qualitative analysis of adaptation strategies with respect to system goals
- Dynamic generation of adaptation strategies: support for generating strategies at runtime, according to goals and depending on resources.
- Case study and evaluation: Assessment of the effectiveness of the approach using a real world scenario.

It is precisely on this last point that the work presented in this document is included. It was necessary to evaluate the presented approach on an already implemented self-adaptive system. To perform the resilience assessment, Rainbow was chosen as the architecture-based platform and DCAS as the case study to be used.

The choice of this platform is justified since it provides generic self-adaptation through gauges (monitoring), a model manager, a constraint evaluator, an adaptation

[2] http://adaas.dei.uc.pt/adaas

engine (reasoning), an adaptation executor and effectors (adapting). It relies on sophisticated knowledge and manipulations of the system properties, constraint rules, adaptation strategies, and adaptation operators. Salehie et al. [20] Rainbow provided the ability of modeling the system's architecture (using AcmeStudio [69]) and a language for development of adaptation strategies (Stitch [70]). Combining these two features with fault injection for direct measurement of the systems metrics and with the use of PRISM [64] to perform the quantitative analysis of performance requirements, it was possible to assess the approach in real case study: DCAS.

Beside other possibilities, analyzed along with DCAS, this product provided by Critical Software fulfilled the requirements required for the assessment. It was an industrial software system already deployed in real environments, it was a product developed in Portugal (mandatory) and, of most importance, was already a self-adaptive system with built-in adaptation mechanisms. In the next sections is provided more information about Rainbow and DCAS.

3.2. **Data Acquisition and Control Service (DCAS)**

Data Acquisition and Control Service (DCAS) is a middleware created by Critical Software that provides a reusable infrastructure for management of monitoring and (non-automatic) control of highly populated networks of devices. In particular, the system is designed to be seamlessly integrated with Critical's Energy Management System platform (csEMS[3]). csEMS is a platform that provides asset management support for power producing companies based on renewable energy sources. The overall csEMS architecture aims at high scalability, flexibility, and customization to enable the operation of control centers with managing capabilities, independently of the underlying technology (wind, solar, etc.).

3.2.1. Objectives

The main objective of DCAS is to request data from the connected devices and save it into the database server at a rate as close as possible to the one configured in their device profiles, supporting as many connected devices as possible, using the system resources in the node where it is running whenever necessary (scale up), and, able to deploy additional instances of the service within the same system if necessary (scale out).

3.2.2. Architecture

To provide a clearer and easy understanding of the adaptation mechanisms is made a resume of the DCAS system and presented an overview of the system architecture. In Fig. 1 is presented a resume of the DCAS system and main components of the system.

[3] http://solutions.criticalsoftware.com/products_services/csEMS/

In this figure are presented the most important classes and modules. The DCAS system consists in 5 core modules:

- **Configuration Provider:** Data wrapper used to read service parameterization values. This module provides configuration values in a standard discrete way in order to encapsulate concrete data structures or business rules that may be subject to change in the future. This class implements a cache of configuration values in order to improve performance. It also implements and publishes a mechanism for it to be notified of configuration changes in order to invalidate and refresh those caches. In this class no adaptation mechanisms are applied. It’s also responsible of loading the list of requests from the database.

- **Polling Scheduler:** Scheduling mechanism to trigger scheduled data retrievals for devices. This class maintains a configured list of scheduled data items and enqueues them for processing at the specified sample rate. The initial array of “Data Streams” loaded from database is converted into an array of “IDataItems” arrays splitted and sorted by Sample Rate. Several pollers shall be instantiated, at least one for each sample rate, and a sub-set of “IDataItems" shall be provided. Each of the sub-sets is the responsibility of one specific poller.

- **Data Requester:** Main processing module responsible of requesting data from the devices. This main processor instantiates several secondary processors per device type and distributes requests to specific queues based on device type. The secondary processors have their own pollers on those queues.

- **Alarmer:** Module responsible for parsing and triggering functionality. This class checks each data item against configured thresholds and triggers alarms on violation.

- **DataPersister:** This module performs two distinct tasks. As a processing module it shall be placed at one of the ends of a processing sequence and shall enqueue and then persist the processed data items. As a "normal" class it is a wrapper for data access and so provides methods to return database data.

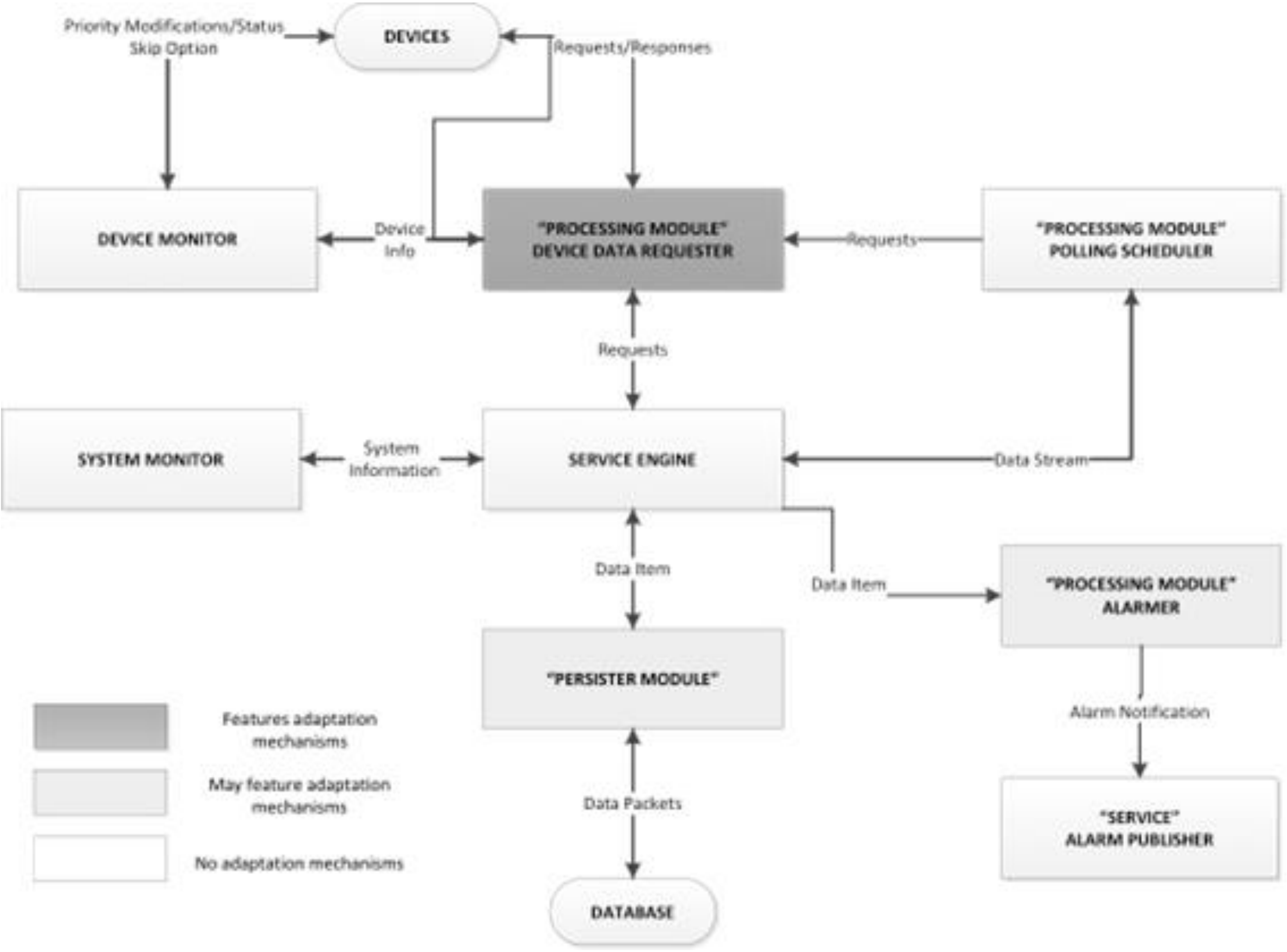

Fig. 1 - DCAS architecture overview

Besides the Core Modules it is useful to refer other important classes/components of the system:

- **Service Engine:** This class is the main control of DCAS system. This class shall be used as a singleton, for that all access shall be made through the Instance property. It's responsible of starting/stopping all the system components (core modules, extension modules, system monitor, etc.)

- **Device Monitor:** Class responsible for retrieving information about the devices status (not related to data gathered with requests).

- **System Monitor:** Class used to periodically collect system information that may be used by service optimization mechanisms (average cpu in usage and free memory).

3.2.3. **Adaptation mechanisms**

In this section we provide a basic idea of the built-in adaptation mechanisms in DCAS.. In a first step in the process of creating the new prototype of DCAS using

Rainbow, these three mechanisms were replicated in order to access if the monitoring and adaptation mechanisms could efficiently control the system. Three main mechanisms are addressed in this document: Scale Up, Rescheduling and Scale Out

Scale Up is a mechanism to add or remove threads that process requests in the queues. For each processing module there is a queue of incoming requests. A threshold value is defined to check the max numbers of requests per queue (queue size). According to the queue size the system can apply different actions:

- If the numbers of requests in the queue is below the defined threshold and near to 0 it means the system is running normally so no modifications are needed.
- If the queue size reaches the threshold, then new thread(s) will be added to the processing module.
- If the queue is empty during a specific time or number of checks it means that one or more threads are not required, so they will be removed to free system resources.

This was a very simple mechanism to implement using architecture based self-adaptation, since it only needed to check the queue size in each of the processing modules (ex: Data Requester). Using the Rainbow framework it was only required to probe each of the desired processing modules for its queue size, check with the defined limits and add or remove threads.

A different and more complex implementation was required since it was necessary to check if the system could provide the required resources (CPU, memory) to add more threads. The DCAS system has a component responsible for checking the systems resources (System Monitor), and in Rainbow was only necessary to probe the system for these values before taking any action, to check if was possible to create more threads, otherwise a different approach was necessary.

Rescheduling is a mechanism applied when the system detects a possible problem with one or more devices, and provides a way to handle device failures. The purpose is to ensure that if a device fails, data gathering from the other devices will continue to be executed. A device will be considered as in a failure state if it fails to respond to several consequent requests, or if the elapsed time to get data from a device is greater than the expected response time. After marking a device as in failure state, the system raises a failure event and reduces the scheduled requests (add a delay to the sample rate interval). Note that requests shall continue to be executed in order to allow failure recovery detection (reduce or remove delay). After a device is marked as in failure state, the system shall be able to detect when the device is back online and execute future scheduled requests as initially configured.

If the system is failing to reach or maintain the expected data rates and is already using all the system resources, it's necessary to create/ launch a new instance of the processing module in order to process requests.

Scale Out can be applied when it's not possible to add more threads in Scale Up. (Not enough cpu capacity or free memory).

According to DCAS documentation, each service instance should not be aware of the existence of others, so it's necessary that each instance gets only the data streams it should process (Data Streams will not be shared between different instances). The analysis on the scale out implementation using rainbow will be discussed at the end of this chapter, since it was one additional objective of this work.

3.3. **The Rainbow framework**

As we mentioned previously we use Rainbow to develop and apply alternative adaptation mechanisms is DCAS system. In a very brief description Rainbow is an architecture-based platform which focuses on two means of achieving cost-effective self-adaptation:

- An approach and mechanism to reduce engineering effort.
- An explicit representation of adaptation knowledge.

Since the work developed in the integration of Rainbow and DCAS was almost exclusively done on the evolution of DCAS we do not focus on providing a very analysis on Rainbow architecture.

Rainbow provides a framework to monitor a target system and its executing environment and reflect observations into an architecture model, detect opportunities for improvements, decide on a course of adaptation, and effect changes. By exploiting commonality between systems, Rainbow provides general, reusable infrastructures with explicit customization points to apply it to a wide range of systems. It also provides useful abstractions to focus engineers on adaptation concerns, facilitating its systematic customization to particular systems. To automate system adaptation, it provides a language, Stitch, to represent routine human adaptation knowledge using high-level adaptation concepts of strategies, tactics, and operators.

More detail information is provided in related papers [5][71] and in Shang-Wen Cheng PhD Thesis [72, p -], the creator of Rainbow.

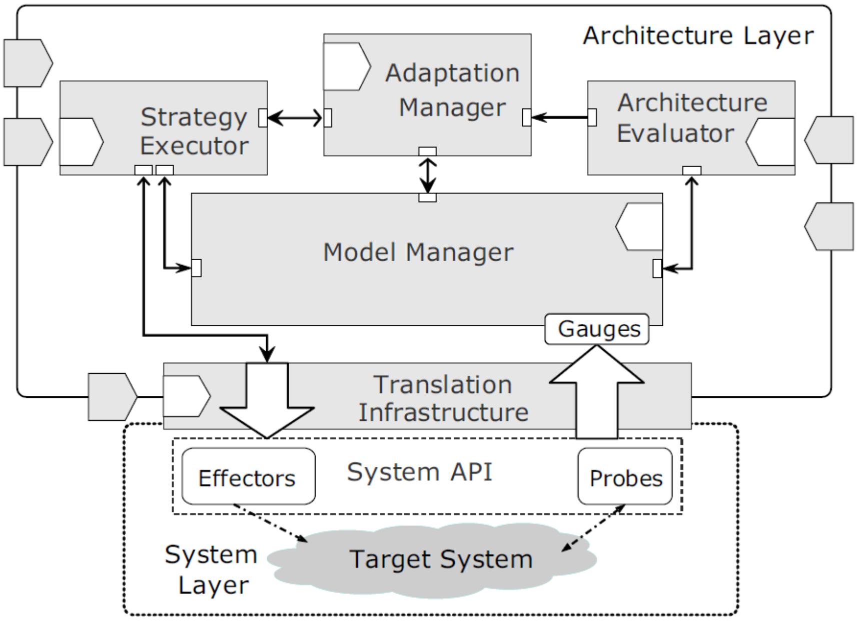

Fig. 2 - Rainbow framework

The focus of our work was in the translation Infrastructure. The monitoring mechanisms in the Translation Infrastructure – probes and gauges – observe the running target system and update properties of an architecture model managed by the Model Manager. The Strategy Executor applies the strategies previously chosen in the Adaptation Manager at the system level by executing effectors. Our work consisted in creating the translation infrastructure and creation of probes and effectors for the DCAS system.

3.4. **Rainbow-DCAS Integration**

In this section is described the process of removing the original adaptation mechanism from DCAS in order to use the new adaptation mechanisms created using Rainbow. We will refer to this process as “evolution of DCAS system”. However, it is important to describe how the translation infrastructure to establish the connection between DCAS and Rainbow was created, since it is tightly coupled to the adaptation mechanisms.

3.4.1. Translation Infrastructure

Implementing the translation infrastructure between DCAS and Rainbow required exposing part of the internal functionality in DCAS through a public interface, enabling communication with Rainbow for extracting system information through probes and effecting changes through system-level effectors. To achieve this, we implemented a lightweight server component embedded in DCAS that enables the exchange of information between a running instance of the DCAS service and Rainbow using TCP sockets. A new class named “TcpServer” was created inside the project to manage the exchange of information between a running instance of the DCAS service and Rainbow.

The purpose for the creation of this class was to provide a communication channel for information exchange between the running instance of DCAS and the Rainbow Framework responsible for monitoring DCAS. This way Rainbow can “probe” for the necessary information through the TcpServer, which will retrieve it from the system and send the answer with the latest information back to Rainbow.

Every time Rainbow needs to “effect” the system, adapting to any triggered condition it will send the required actions and new system values using the same communication channel, being the TcpServer responsible for treating the incoming messages and applying the necessary actions.

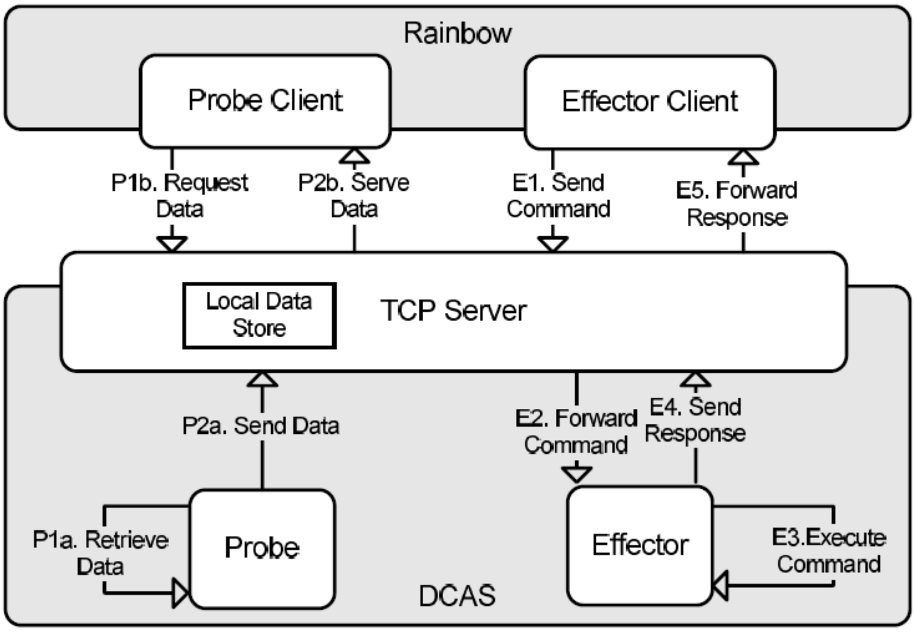

Fig. 3 - DCAS-Rainbow translation infrastructure

Figure 3 illustrates the translation infrastructure used between Rainbow and DCAS. Probes and effectors in Rainbow act as clients of the TCP server, which acts as a mediator between them and the actual probes and effectors embedded in DCAS:

- Probes embedded in DCAS keep the values of probed variables updated in a data store local to the TCP server, pushing updates whenever variables change (P1a and P2a). Then, when a probe client in Rainbow requests the value of a particular variable (P1b), it is directly served from the local data store to the probe client (P2b). This approach was chosen due to the difficulty of invoking the necessary operations to retrieve data in DCAS from the TCP server. Concretely, information such as queue sizes or number of active pollers in the data requester, as well as information relative to device data streams could not be obtained from the TCP Server, so different parts of DCAS code were instrumented to extract this information and update it in the TCP server data store.
- Effectors clients in Rainbow send requests for command execution to the TCP Server (E1), which forwards them to the effector embedded in DCAS (E2). Next, the effector executes the command (E3) and returns a response to the TCP server that states whether execution was successful (E4). Finally, the TCP server forwards the response to the effector client in Rainbow (E5).

In Figure 5 we see that the translation infrastructure is consisted by three main components: TcpServer, probes and effectors. Next we provide a little description on each of the components.

The **TcpServer** consists in a basic server using TCP Sockets, with a specific IP address and port that is continuously waiting and listening for possible client connections. The client will use server IP address to establish the connection (in this case the clients will always be Rainbow Probes or Effectors).

Every time a connection is accepted the incoming message is saved into a buffer. When the message is complete it is transformed into a string. As convention all the information between DCAS and Rainbow is send as String, being the information then treated when necessary. Since this document only focus on DCAS, the client approach will not be discussed.

Rainbow **probes** request the TcpServer for system values or important information of specific Processing Modules (ex: Data Requester processing module) necessary for triggering the adaptations mechanisms.

Some of these values, like the available memory or CPU usage, can be easily acceded, since the original implantation of DCAS has specific classes (in the case of memory and CPU is the System Monitor class) responsible of retrieving these values.

Regarding the information of the main components of the system (Core modules), the chosen approach was to create data structures inside TcpServer to save all the necessary information directly inside the class, providing a more direct access to the data. The same approach is used to store information regarding DataStreams and Devices.

All the processing modules featuring adaptation mechanisms, (ex: Device Data Requester) share the same base class ("Base Processing Module.cs"), and since a list with all the existing core modules is already created inside the server class, each time one of these modules is called during runtime, the TcpServer receives and saves the updated information about each of the modules (ex: queue sizes, number of pollers, etc.). This information is especially important in the "scale up" mechanism".

New methods were also created, related to the "rescheduling" mechanism in order to provide information related to the time required (elapsed time) to perform requests to the devices.

3.4.2. Evolution of DCAS system

Rainbow **effectors** are basically commands received from Rainbow regarding the possible adaptation mechanisms with the intended actions to take and new values to change.

In the initial implementation of DCAS the entire process of checking system values and possible violations (probing the system) and take actions when necessary (effecting the system) was referred as "throttling mechanism". This mechanism was associated with a timer function in which was possible to change the "gap" value between each checking period. With the integration of DCAS with Rainbow this process is now controlled from outside the system, and so the native methods of the "throttling mechanism" are no longer used. It was necessary to create new methods that would correspond to Rainbow decisions (Effectors).

Regarding the "Scale up" mechanism, new methods to add and remove pollers were created, using the information sent to the TcpServer from Rainbow.

```
public virtual void IncreasePollers()
    {

      Console.WriteLine("ADDING POLLERS FROM SERVER REQUEST");

     //  Console.WriteLine(this.ModuleName + " increasing pollers");

      int newPollerPriority = this.GetMaxPriorityQueueCount();
      this.IncreasePollers(newPollerPriority, 1);
      throttlingCheckNext = throttlingCheckCount + throttlingCheckGap;

      #region Log Info
      Log.Write(LogLevel.Information, LogCategory.NemoDebug,
        () => string.Format(CultureInfo.InvariantCulture,
          "{0} [{1}] - {2}.{3} : Grow pollers from {4} to {5} queue size: {6}, queueStatus: {7}",
          DateTime.Now, Thread.CurrentThread.ManagedThreadId, this.GetType().Name,
          MethodBase.GetCurrentMethod().Name, cyclingPollerThreads.Count - 1,
          cyclingPollerThreads.Count, this.concurrentQueue.Sum(q => q.Value.Count),
          GetInputQueueStatus()));
      #endregion Log Info
      //}
    }
```

Regarding “Rescheduling” all the new methods created, necessary to change the delay in Data Streams, priorities of Data Items in the Polling Scheduler, and threshold value were created using the information provided from the data structures created in the TcpServer.

```
case "updateDelay":

          if (!addRateDelayEffectorState)
          {
            StreamId = parts[1];
            int delay = Convert.ToInt32(parts[2]);
            //call method to change RateDelay ,get data stream and update SampleRateDelay
            long lValue = long.Parse(StreamId);
            dataStreams = DataStreamManager.GetDataStreams(new string[] { "S" });
            foreach (DataStream ds in dataStreams)
            {
              if (ds.Id == lValue)
              {
                ds.SampleRateDelay = delay;
                Console.WriteLine("updating delay in dataStream: " + ds.Id + " to: " + ds.SampleRateDelay);
              }
            }
          }
          else
            Console.WriteLine("effector failign!!!!");
          answer = "SUCCESS";
          break;
```

3.4.3. Scale Out implementation

As mentioned in section 3.2.3 if the system is failing to reach or maintain the expected data rates and is already using all the system resources, it’s necessary to create/ launch a new instance of the processing module in order to process requests. This mechanism is called Scale Out and should be applied when it’s not possible to add more threads in Scale Up. In the original implementation of DCAS this adaptation was performed by and human and the objective was to implement a version of this mechanism that would be triggered without human interaction.

Ideally this adaptation could be considered as an adaptation that would be triggered only in the case of failure in applying Scale Up (the system do not have additional free resource to add more pollers). In this situation a new instance of the service would be created, splitting the initial workload with the first instance. When referring to workload, we mean the number of DataStreams currently being processed. According to DCAS documentation, each service instance should not be aware of the existence of others, so it’s necessary that each instance gets only the data streams it should process (Data Streams will not be shared between different instances).

Applying this approach required that every time DataStreams were moved from one instance to another, all the instances should be notified to reload all the configurations and retrieve the new set of DataStreams specifically associated with that instance.

The effector to assign DataStreams to different system instances and the effector to signal the running instances to “restart” were created with success. The first one required the creation of a new class (dataBaseAcess.cs) to connect separately to the system’s database in order to run the respective SQL query to modify the DataStreams information.

However, a major problem for the successful implementation of this mechanism came from Rainbow. It was not possible to dynamically re-allocate components while the system was running. Each instance would have a maximum number of DataStreams associated. Basically, it was not possible to move DataStreams between different instances. This way it was not possible to test the mechanism with the pretended function.

3.4.4. Integration conclusions

In this section was addressed the first main objective on this work: the abstraction of existing adaptations from the system. We can conclude that the process of creation of the translation infrastructure between DCAS and Rainbow was successfully achieved, removing the original adaptation mechanism and successfully endowing Rainbow with full control over DCAS. All the probes to retrieve information from the system and all the effectors, connected to new created adaptation strategies were applied with success. This part of the work was crucial. Without integrating Rainbow and DCAS it was not possible to perform the architecture-based resilience evaluation.

Scale Out implementation was considered a secondary objective, but, although the objective was not completely achieved, that was due to external factors to what was committed to achieve. The required implementation on the DCAS prototype side was completed with success.

Chapter 4
Resilience Evaluation

In this section, we start by describing the approach followed for resilience evaluation, and then formally describe how the experiment evaluation was performed and the results obtained.

4.1. **Evaluation approach**

The followed approach focus in the evaluation by comparison of the adaptation mechanisms of a self-adaptive software system, relying on the identification of representative environmental and system changeloads (i.e., sequence of changes) used in the runtime stimulation of the system. This type of evaluation requires the use of relative resilience metrics in order to compare how different adaptive solutions respond to a particular set of (system or environmental) conditions;

This evaluation was performed in two steps:

a) **Changeload Identification:** Exploring the architectural model of DCAS prototype to identify and select the most relevant (sequences of) changes (i.e., the changeload) that have the best potential to unveil system or environmental faults during run-time stimulation. The first step for changeload generation is to identify environmental and system anomalies or sources of potential changes. To stimulate the system we need to represent the variables identified before into a set of system and environment changeloads. Environmental changeloads are used to lead the environment to reach the required conditions for adaptations to be used. System changeloads are used to assess the resilience of the adaptations mechanisms. This step was not required for the Original DCAS strategies since we already have the knowledge about the system metrics to evaluate and the adaptation mechanisms to test.

b) **Runtime Stimulation:** Stimulate the system and its environment during execution using the changeloads identified in the previous step, and collect information about the system's response in order to aggregate it into a probabilistic model of the system's behavior that is used to evaluate by comparison the alternative adaptation mechanisms and conclude which have the best results. We stimulate the system and its environment during execution using the changeloads identified in the previous step, and collect information about the system's response. For this we define the resilience metrics, evaluated during the stimulation, according to the system goals that are used to classify each of the adaptation mechanisms. The classification of each of the adaptations is then transformed into a probabilistic response model of the system which is used as input to a probabilistic model checker in addition to the resilience properties obtained from system goals. This probabilistic response model is used to evaluate by comparison the adaptation mechanisms and conclude which have the best results.

Although probabilistic behavioral models are obtained during runtime stimulation, they are built over the set of properties contained in the architectural model, and describe the evolution of the values of such properties obtained from monitored variables in the system at runtime. The use of probabilistic behavioral models enables the quantification of the probability of satisfaction of system properties (expressed in Probabilistic Computation-Tree Logic (PCTL) [63] when the system is subject to a particular stimulus.

4.1.1. Operational Profiles and Scenarios

To perform the two steps described previously it was necessary to define the conditions in which the system operates for changeload identification and then the conditions and additional configurations to perform runtime stimulation.

Two different operational profiles can be identified in self-adaptive systems [73].

In situations where the system is running without any anomalies, we are in the presence of *conventional operation profiles*. *Non-conventional operational profiles* are associated with changes in the system or its environment that induce anomalies in the system (typically triggering adaptations).

The systematic identification and classification of change types is fundamental to support the definition of change scenarios. One of the main base concepts is that of a scenario. A scenario is a required sequence of events that captures the state of the system and its environment, system goals, and changes affecting all the aforementioned elements. It is defined in terms of state (system and environment), changes applied to that state, and system goals. Scenarios can be classified into two groups: base scenarios and change scenarios [73].

A *base scenario* is defined in terms of typical conditions during the execution of the system, which includes: a typical (stable) state of the system and its environment, and a set of fixed goals.

The workload of a base scenario should be representative of the typical amount and type of work assigned to (or expected from) the system in a specified time period. Typical operation conditions comprise the typical setup of systems in the domain, as well as representative characterization of the system's environment, and the hardware and software resources typically used. Hence, a base scenario reflects the operational characteristics of systems in the domain while running a typical workload and operating in the absence of changes, setting the baseline for comparison with situations when the system is faced with changes that may drive it into an adaptation process.

It should be noted that "typical" does only imply a stable state of the system with no abnormal conditions, not that the workload or operation conditions cannot be dynamic.

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

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

Since the purpose of this approach is the evaluation by comparison of adaptation mechanisms, the use of base scenarios was discarded, since it didn't fostered the conditions to apply the adaptation mechanisms.

4.1.2. Criteria Definition

Before starting the process for system stimulation it is necessary to define the resilience metrics to be evaluated, according to the system goals that are used to classify each of the adaptation mechanisms. These metrics will be used to classify and compare each of the adaptation mechanisms, as the objective is to understand how effective each adaptation alternative is in terms of assuring that the system satisfies the predefined goals after adaptation. Referring again to Laprie, a resilient system is defined as one whose service can justifiably be trusted when facing changes [21], that is, a system that fulfills its goals in a dependable and persisting manner in spite of changes in its environment or the system itself. Therefore, the definition of resilience incorporates the fulfillment of system goals. In other words, the metrics are strongly related to the goals targeted by the system and have to characterize it in a useful and meaningful way.

.

4.1.3. Experimention and evaluation

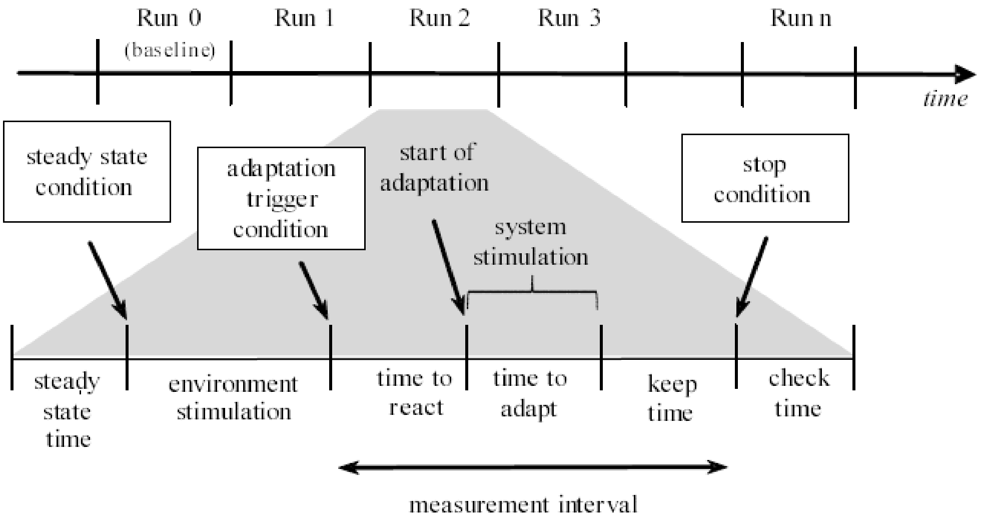

Fig. 4 - Non-conventional operational profile

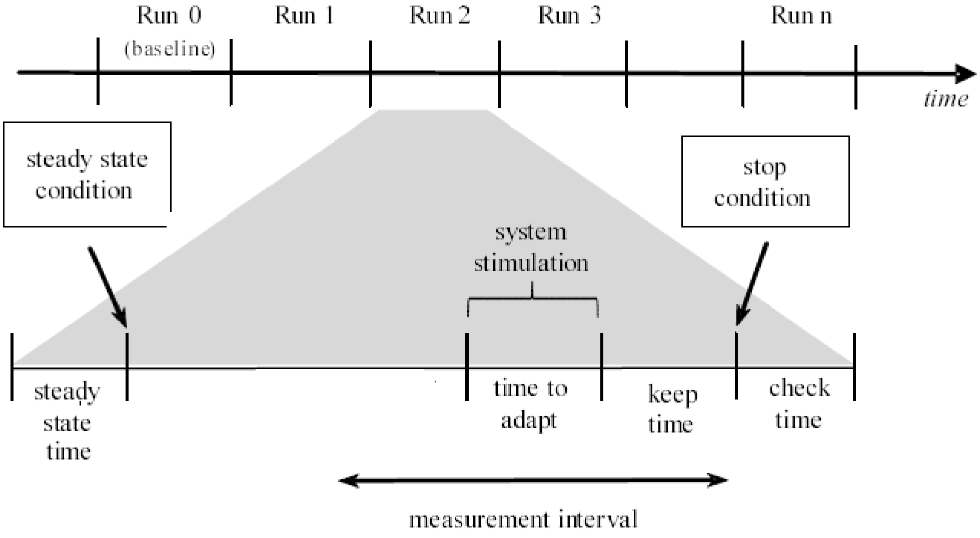

FIg. 5 - Conventional operation profile

The experimental profiles used for the evaluation can be divided in two cases (see Figure 4 and Figure 5). Each experiment, regardless of the targeted operational case (NCOP or COP), includes a set of runs.

Run 0 consists of performing environmental stimulation to trigger adaptation, without performing system stimulation. The goal is to collect baseline information about the behavior of the adaptation alternative in the absence of system changes. This baseline will be used later as reference to understand the impact of system changes in the execution of the adaptation strategies.

Figure 4 represents the case where the goal is to assess the target system when adaptations is required (i.e., the system is running in a nonconventional operational profile).

Figure 5 reflects the case were no adaptation is required in the target system (i.e., the system is running in a conventional operational profile). During Runs 1. . .N the system is run in such a way that environmental stimulation will lead to triggering adaptation, and then changes are injected on top of the environmental stimulation during adaptation to measure their impact in the different adaptation alternatives. In order to assure that each run portraits a realistic scenario as much as possible, and at the same time assures that important properties such result repeatability and representativeness of results are met, the definition of the profile of the run has to follow several rules. The following points summarize those rules (see Figure 4):

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

2. The tests are conducted with the system in a steady state condition, which is achieved after a given time executing transactions (steady state time).

3. Environmental stimulation is conducted after the system achieves the steady state to reach an adaptation trigger condition in the first place, and then keeps on going throughout the execution of adaptation, keeping the conditions of the environment to

make adaptation valid. The existence of a time to react is considered as there may be a lag between the trigger condition and the activation of the adaptation strategy.

4. Adaptation is exercised by injecting system changes during its execution.

5. When the adaptation completes, the system must continue to run during a keep time in order to characterize the system speedup after adaptation. Since we are interested in characterizing only the behavior of the system during adaptation, the measurement interval starts in each run only when adaptation starts, and ends when the stop condition is met.

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

4.2. **Experimental evaluation**

The objective of this work is the evaluation by comparison of the adaptation mechanisms of a self-adaptive software system, relying on the identification of representative environmental and system changeloads. In particular, we compare the performance and efficiency of architecture-based self-adaptation mechanisms with those achieved by DCAS built-in adaptation mechanisms.

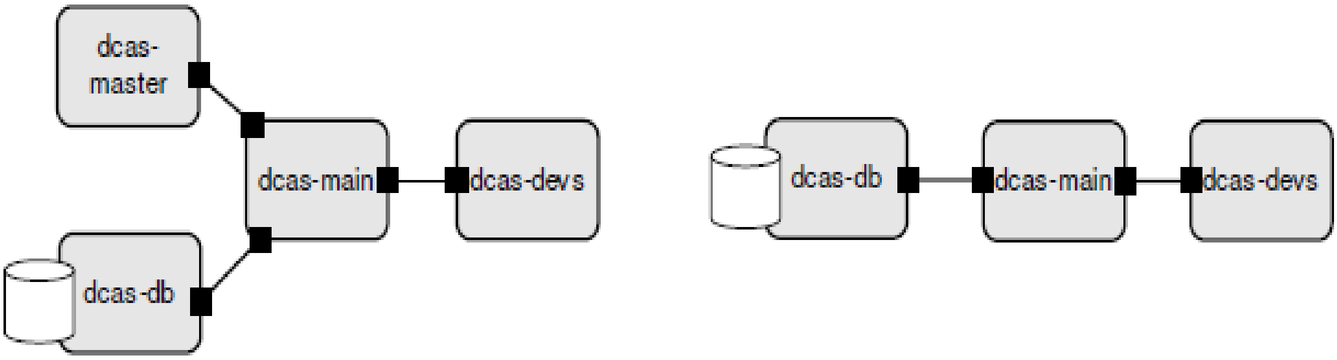

Fig. 6 - Experimental setup: Rainbow-DCAS (left) and DCAS (right)

For our experimental setup, we deployed both versions of DCAS across three different machines (Figure 6): dcasdb 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. In the case of Rainbow-DCAS (Figure 8, left), 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 memory.

Our experiments include 100 data streams with a sample rate of 1 second. The duration of each test is 40 minutes (2400s), and the pattern followed is:

Non-conventional operational profile:

1) Environment changeload:

   i. 600s of normal activity to let the system achieve a steady state;

   ii. 600s of disturbance, during which we induce low responsiveness in data streams (by artificially adding a 2-second delay in the response time of 25% of the data streams);

   iii. 1200s of normal activity to let the system stabilize.

2) System changeload:

   i. 600s of normal activity to let the system achieve a steady state;

   ii. 600s of disturbance, during which we induce low responsiveness in data streams (by artificially adding a 2-second delay in the response time of 25% of the data streams) but with the injection of system change after starting the disturbance period (normally 1s after injecting the disturbance);

   iii. 1200s of normal activity to let the system stabilize.

Conventional operational profile:

1) System changeload:

   i. 600s of normal activity to let the system achieve a steady state;

   ii. t= 601s ; injection of system change

   iii. remaining period of normal activity

4.2.1. Changeload Identification

The first task performed consisted in identifying changeloads that would lead the system towards adaptation (environment changeload). Different quality attributes, such as performance, cost or information quality, can serve as the base context for applying adaptation.

Environment changeload

Analyzing the original DCAS implementation, the non-functional requirements are mainly concerned about two quality objectives: Performance and cost (scalability).In DCAS performance is measured by the number of requests processed by time unit, more specifically, the requests per second (rps) stored in the database center. Cost analysis is related to the number of active pollers used in data requesters.

The number of requests processed by the system is highly affected by the time each device takes to respond. If a device responds with delay or even fails to respond it will lead to a performance drop, and this way the triggering conditions for application of the adaptation mechanisms will be reached. Thus we concluded that adding/removing delay into devices should be used as the environment changeload to lead the system towards adaptation. In the previous work performed over DCAS [74] it was determined that the workload required for this purpose should induce a 2-seconds delay to 25% of devices responses

An additional scenario was also tested using a similar workload but inducing a 30-seconds delay instead of 2-seconds. This scenario was used in order to assess the impact of device failures in the system performance.

System Changeload

Normally, the identification of the system changeload is performed based on a risk-based approach [75] that considers the probability and impact of system changes, and is divided of three steps:

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

Taking the change types, we then identify specific system changes that may impact the system goals. This was essentially a manual process that data collected from direct measurements and expert knowledge, and basically consists of finding, for each change type, tangible changes that may affect the system during adaptation.

Furthermore, instantiating a change required the specification of concrete attribute values (depending on each change) and of the trigger instant, as well as the duration (when applicable) of each change.

Dynamic Analysis. The goal of this step is to understand the impact of each system change in the target system. The environmental changeload identified previously is used to stimulate the system towards conditions that trigger adaptation, in conjunction with system changes identified before, which are injected in the system while undergoing adaptation. Thus, each of the identified system changes is run individually on the system under typical workload and operational conditions, gathering data about the variables included identified for environment stimulation, during a particular time frame [0; t]. As in for the Environment Stimulation, each change is executed a number of times under similar conditions to obtain a set of traces statistically representative of the behavior of the system variables while undergoing the change. This information is then used to build an impact model for each system change.

Filtering/Cutoff. The goal of this step is deciding which system changes should be included in the changeload. This is needed as the number of potential system changes usually is very large (especially for complex systems) and, thus may become impracticable.

Moreover, it is expected that many of the changes identified present a low probability of occurrence and/or may have a low impact in the system. This way, following a risk-based approach [25], is proposed the use of an exposure matrix that allows understanding how relevant each change is. The goal is prioritizing the changes, selecting a Top-N (i.e., the most representative ones), based on a cut-off level.

As mentioned before, field data can be used to support the process of classifying the impact and probability of changes. However, in most cases that data is not available, demanding for expert judgment.

Since we already had the knowledge about the system metrics to evaluate and the adaptation mechanisms to test for original DCAS, we focused in identifying the system variables and operators used in adaptation mechanism for the Rainbow-DCAS prototype.).In DCAS performance is measured by the number of requests processed by time unit, more specifically, the requests per second (rps) stored in the database center. Cost analysis is related to the number of active pollers used in data requesters.

AddPoller Effector
RemovePoller Effector
ChangeRateDelay Effector
QueueStatus Probe
Rps Probe
DataPersisterModule
DataRequesterModule
Polling Schedduler Module
Service Engine Module
Database-DCAS Connection
Processor Node
Database

Table 1 System changes

In Table 1 are presented the created system changes. The selection of these changes was based in the existing adaptation mechanisms and the components of the system considered relevant for its behavior. As described in section 3.4. the monitoring mechanisms in the Translation Infrastructure – probes and gauges – observe the running target system and update properties of an architecture model managed by the Model Manager. The Strategy Executor applies the strategies previously chosen in the Adaptation Manager at the system level by executing effectors. If the properties used to trigger the adaptations are not properly updated, the adaptation manager will not detect that a possible triggering condition has been achieved. The variation of the input queue size in the DataRequesterProcessor module (queueStatus) and the current number of requests per second being processed (rps) are conditional variables used for triggering and selection of adaptation mechanisms. This way, injecting failures in the process of updating properties of the architecture model (probing) and the application of the adaptation strategies (effecting) may impact the system behavior.

The application of the adaptation mechanism is mainly related to the change of the number of active pollers (add/remove pollers) and the scheduling delay associated to a device when presenting (or not) problems in the response time (change rate delay). Being the number of pollers and rate delay the direct targets of the adaptation strategies, injecting failures in these effectors may result in a critical impact in the system behavior. The remaining changes presented are related to the injection of failures in the main components of the system architecture. We pretend to assess the impact of each case in the system behavior when under adaptation process (non-conventional profile) and when running under normal conditions (conventional profile). It is important to conclude if the system is able to recover in the presence of these changes or if it leads to a catastrophically crash of the system.

In our approach no fileting/cutoff was made since we concluded that the results on analyzed changes could be useful for future studies.

4.2.2. Runtime stimulation

The impact of the selected environment stimulation scenario (device delay) on the system's performance and cost are described in Fig.7 and Fig.8. Results show that Rainbow-DCAS is able to recover faster than DCAS. Concretely, immediately after starting disturbance period (2-second delay add to 25% of the total devices), the performance of both DCAS and Rainbow-DCAS drops, going from average values (near 130 rps) to values in the range 0-50. However, by t=700s, performance in Rainbow-DCAS has been restored to normal levels. In contrast, DCAS takes more time to recover, only going back to normal almost when the disturbance is removed by time t=1200s. Regarding efficiency, Rainbow-DCAS is faster in reacting to the disturbance, since the adaptation strategies were modified to activate pollers more aggressively when low responsiveness appears in data streams. This comes at the cost of more active pollers, but it was considered an acceptable solution given that the main priority of the system is performance. It is important to remind that these tests were performed using a non-conventional operation profile (NCOP).

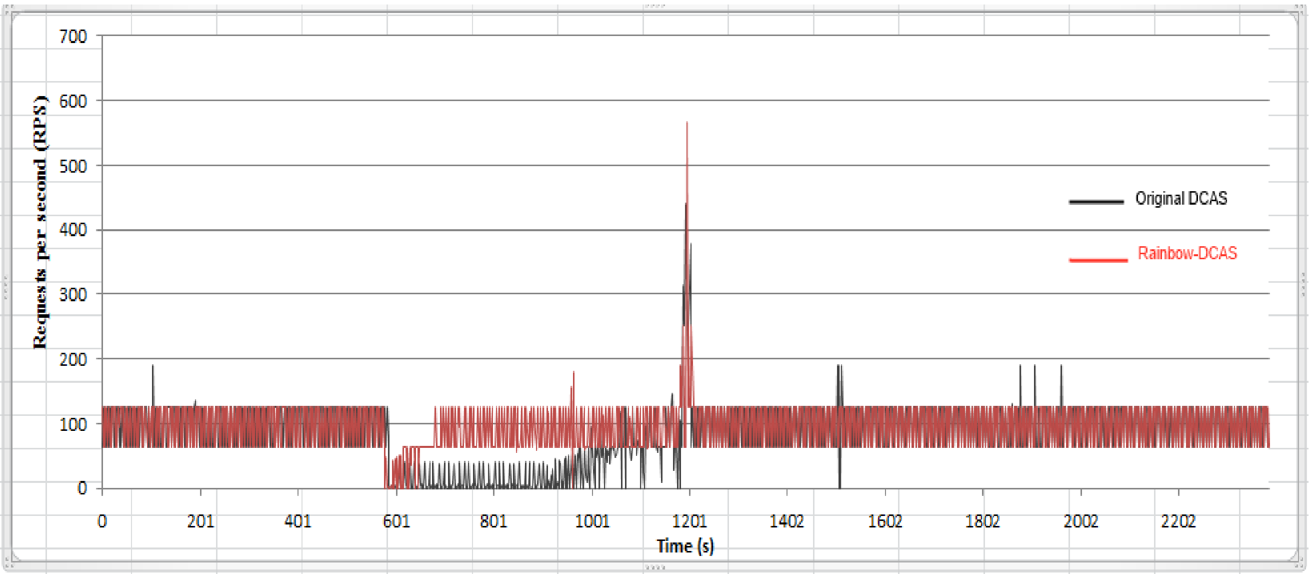

Fig. 7 - Performance

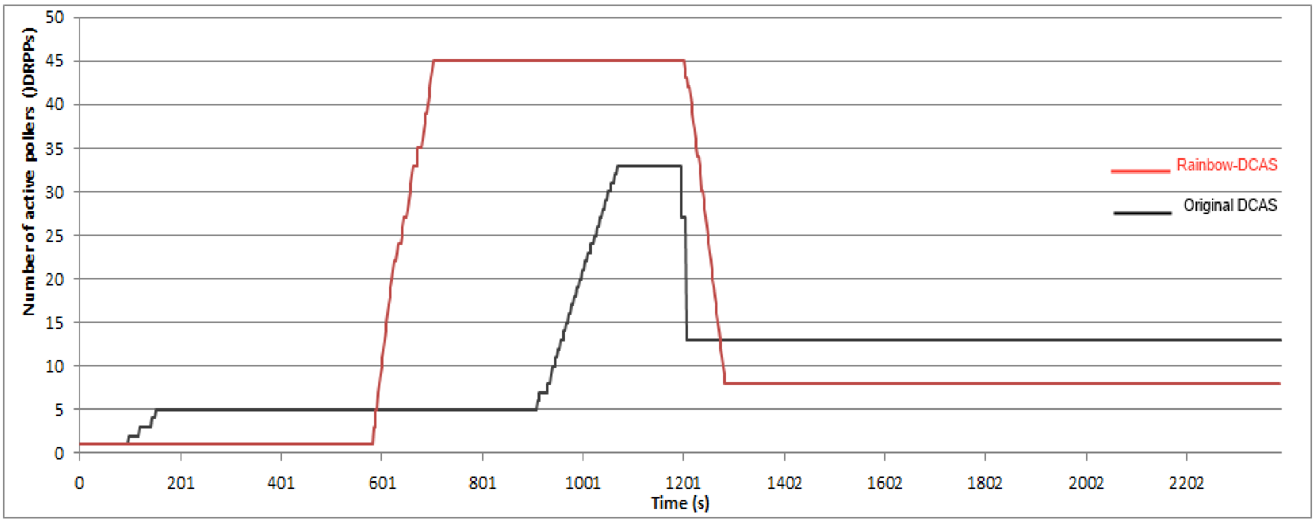

Fig. 8 – Cost

In Fig.9 is presented an example of a system changeload scenario, more concretely in the case of AddPoller efector. In this scenario the failure injected consisted in blocking the adaption mechanism application by not allowing the addition of pollers. In both versions, during the entire test, only one poller was active, proving the successful injection of the change. It is easily concluded that the performance of the system is drastically affected during the disturbance period, with rps values remaining close to 0 most of the interval. After removing the disturbance the system is capable of recovering the normal performance.

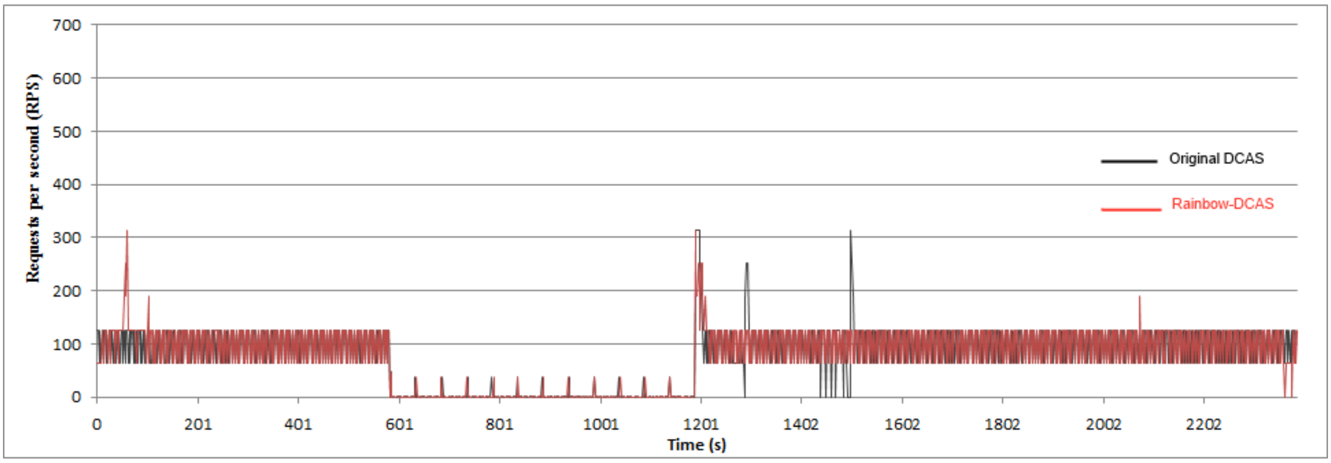

Fig. 9 - Performance

4.2.3. System Resilience Evaluation

The main objective of the system is to achieve an acceptable level of performance (rps) while keeping down the cost of running the system (number of active pollers). To assess this objective we study the resilience of the system in the presence of the different failures comparing the application of the different adaptation mechanisms. To achieve our goal, we built a set of probabilistic models from all scenarios that uncovered adaptation failures using a time frame [0; 600]. Each model is synthesized from data obtained from 30 different runs of each scenario (COP and NCOP) resulting in a total of (4)*30=120 runs. Using these models, we quantify the levels of system resilience while the system is in:

- **Conventional Operational Profile**: The system is operating steadily with good levels of performance and within cost, so adaptation is not required (Table 2 and Table 3). In this case, we analyze resilience in terms of whether injecting failures make the target system deviate from its COP (e.g., caused by the triggering of unnecessary adaptations). Deviation from the system's COP can be either in terms of performance or cost. Concretely, we quantify**: (Ƥ1)** the probability of performance level falling below the MIN RPS threshold in specific time interval (P=? [F (rps<120)&(time<X))], **(Ƥ2**) the probability of the number of active pollers raising above a specific value (P=? [F pollers>Y], and **(Ƥ3)** the probability of the performance level being above the MIN RPS threshold using a limited number of active pollers (P=? [F (rps>120)&(pollers<Z)])Probability values displayed in the table for **Ƥ1** and **Ƥ2** are the complementary, so that lower values indicate better resilience. An additional reward attribute is evaluated (uptime) in which we access the number of seconds during the disturbance period were the performance is considered acceptable (above MIN RPS threshold). In **Ƥ1** the total time interval (0-600s) is divided in sub-intervals, with 50 seconds each in order to identify in every interval the property value. In **Ƥ2** are defined 5 different (10, 20, 30, 40 and 50)

values in order to assess the probability of the number of active pollers being above than the tested value. **Ƥ3** uses the values defined in **Ƥ2** to assess the number of active pollers being used when the system recovers to normal performance.

- **Non-Conventional Operational Profile** (associated with anomaly rpsViolation). The system is underperforming, so adaptation has been triggered and the controller is in its planning or execution stage (Table 4 and 5). We analyze resilience in terms of whether the system can recover, returning to its COP by a given time interval. Properties **Ƥ2 and Ƥ3** are checked again in this profile under the same conditions, such as the reward property "uptime". A small modification is made in property **Ƥ1.** Now we acess the probability of performance level going above the MIN RPS threshold in specific time interval (P=? [F (rps<120)&(time<X))].
  This way it is possible to know the specific amount of time until the system's performance is above the MIN_RPS threshold.

	Table 2: Conventional Operation Profile - Rainbow Dcas																						
	P1=? [F (rps<120)&(time<X)]												**P2**=? [F pollers>Y]					**P3**=? [F (rps>120)&(pollers<Z)]					R{"uptime"}=? [F (time=600)]
	X=50	X=100	X=150	X=200	X=250	X=300	X=350	X=400	X=450	X=500	X=550	X=600	Y=10	Y=20	Y=30	Y=40	Y=50	Z=10	Z=20	Z=30	Z=40	Z=50	
Property: **Test Case:**	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**	**P11**	**P12**	**P13**	**P14**	**P15**	**P16**	**P17**	**P18**	**P19**	**P20**	**P21**	**P22**	
AddPollerEffector Failure	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**100**	**100**	**100**	**100**	**100**	**600.0**
RemovePollerEffector Failure	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**100**	**100**	**100**	**100**	**100**	**600.0**
ChangeRateDelay EffectorFailure	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**100**	**100**	**100**	**100**	**100**	**600.0**
DataPersisterModule Crash	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**100**	**100**	**100**	**100**	**100**	**599.9**
DataRequesterModule Crash Failure	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**100**	**100**	**100**	**100**	**100**	**600.0**
Polling Schedduler Module Crash	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**-**
Service EngineModule Crash	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**-**
Db-dcas Connetion Shutdown	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**100**	**100**	**100**	**100**	**100**	**600.0**
RpsProbe Failure (dcas rpsProbe)	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**0**	**0**	**0**	**0**	**0**	**36**	**36**	**36**	**36**	**36**	**-**
DcasTotal Failure	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**-**
Database Crash	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**-**
QueueStatusProbe Failure	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**100**	**100**	**100**	**100**	**100**	**599.99**

Table 2 Conventional Operation Profile - Rainbow DCAS

	Table 3 - Conventional Operation Profile - Original Dcas																						
	Ƥ1: P=? [F (rps<120)&(time<X)]												**Ƥ2:**P=? [F pollers>Y]					**Ƥ3:**P=? [F (rps>120)&(pollers<Z)]					R{"uptime"}=? [F (time=600)]
	X=50	**X=100**	**X=150**	**X=200**	**X=250**	**X=300**	**X=350**	**X=400**	**X=450**	**X=500**	**X=550**	**X=600**	**Y=10**	**Y=20**	**Y=30**	**Y=40**	**Y=50**	**Z=10**	**Z=20**	**Z=30**	**Z=40**	**Z=50**	
Property: **Test Case:**	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**	**P11**	**P12**	**P13**	**P14**	**P15**	**P16**	**P17**	**P18**	**P19**	**P20**	**P21**	**P22**	
AddPollerEffector Failure	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**100**	**100**	**100**	**100**	**100**	**599.93**
RemovePollerEffector Failure	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**33**	**0**	**0**	**0**	**0**	**100**	**100**	**100**	**100**	**100**	**599.56**
ChangeRateDelay EffectorFailure	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**34**	**0**	**0**	**0**	**0**	**100**	**100**	**100**	**100**	**100**	**598.76**
DataPersisterModule Crash	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**100**	**100**	**100**	**100**	**100**	**599.33**
DataRequesterModule Crash Failure	**3**	**7**	**13**	**16**	**16**	**16**	**16**	**16**	**16**	**16**	**16**	**16**	**23**	**0**	**0**	**0**	**0**	**100**	**100**	**100**	**100**	**100**	**596.1**
Polling Schedduler Module Crash	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**-**
Service EngineModule Crash	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**-**
Db-dcas Connetion Shutdown	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**3**	**0**	**0**	**0**	**0**	**90**	**100**	**100**	**100**	**100**	**599.99**
DcasTotal Failure	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**-**
Database Crash	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**100**	**79**	**68**	**47**	**21**	**100**	**100**	**100**	**100**	**100**	**-**
QueueStatusProbe Failure	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**100**	**100**	**100**	**100**	**100**	**598.76**

Table 3 Conventional Operation Profile - Original DCAS

Table 4 Non-Conventional Operation Profile - Original Dcas

	Ƥ1=? [F (rps>120)&(time<X)]												Ƥ2=? [F pollers>Y]					Ƥ3=? [F (rps>120)&(pollers<Z)]					R{"uptime"}=? [F (time=600)]
	X=50	X=100	X=150	X=200	X=250	X=300	X=350	X=400	X=450	X=500	X=550	X=600	Y=10	Y=20	Y=30	Y=40	Y=50	Z=10	Z=20	Z=30	Z=40	Z=50	
Property: **Test Case:**	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**	**P11**	**P12**	**P13**	**P14**	**P15**	**P16**	**P17**	**P18**	**P19**	**P20**	**P21**	**P22**	**P23**
Device Delay	0	7	20	37	58	70	85	93	97	97	97	100	100	100	64	0	0	0	93	100	100	100	119.4
Device Failure	0	0	0	0	0	0	0	0	0	0	0	0	100	100	97	60	47	53	93	100	100	100	0.97
AddPollerEffector Failure	0	0	0	0	0	0	0	0	0	0	0	55	0	0	0	0	0	100	100	100	100	100	3.45
RemovePollerEffector Failure	0	0	3	7	26	30	37	50	53	53	56	93	77	67	57	0	0	20	33	71	100	100	90.83
ChangeRateDelay EffectorFailure	0	0	0	0	13	26	45	47	53	76	78	98	93	90	60	0	0	27	100	100	100	100	103.33
DataPersisterModule Crash	0	0	0	0	17	28	40	50	60	64	78	100	93	90	62	0	0	28	100	100	100	100	85.99
DataRequesterModule Crash Failure	0	7	13	26	40	53	59	68	74	85	90	99	100	94	41	0	0	16	95	100	100	100	76.77
Polling Schedduler Module Crash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-
Service EngineModule Crash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-
Db-dcas Connetion Shutdown	0	0	0	3	7	17	23	25	33	49	60	95	100	100	100	100	0	100	100	100	100	100	58.89
DcasTotal Failure	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-
Database Crash	0	0	0	0	0	0	0	0	0	0	0	0	89	64	57	39	18	0	0	0	0	0	0.0
QueueStatusProbe	0	0	7	17	35	40	50	59	88	94	96	100	100	100	62	0	0	20	97	100	100	100	100.13

Table 5 Non-Conventional Operation Profile - Rainbow Dcas

	P1=? [F (rps>120)&(time<X)]												P2=? [F pollers>Y]					P3=? [F (rps>120)&(pollers<Z)]					R{"uptime"}=? [F (time=600)]
	X=50	**X=100**	**X=150**	**X=200**	**X=250**	**X=300**	**X=350**	**X=400**	**X=450**	**X=500**	**X=550**	**X=600**	**Y=10**	**Y=20**	**Y=30**	**Y=40**	**Y=50**	**Z=10**	**Z=20**	**Z=30**	**Z=40**	**Z=50**	
Property: **Test Case:**	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**	**P11**	**P12**	**P13**	**P14**	**P15**	**P16**	**P17**	**P18**	**P19**	**P20**	**P21**	**P22**	
Device Delay	**0**	**90**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**0**	**100**	**100**	**100**	**100**	**100**	**287.34**
Device Failure	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**100**	**100**	**100**	**100**	**0**	**100**	**100**	**100**	**100**	**100**	**0.20**
[redacted]																							
AddPollerEffector Failure	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**59**	**0**	**0**	**0**	**0**	**0**	**100**	**100**	**100**	**100**	**100**	**4.10**
RemovePollerEffector Failure	**0**	**92**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**0**	**0**	**0**	**10**	**100**	**100**	**269.36**
ChangeRateDelay Effector Failure	**0**	**96**	**97**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**0**	**100**	**100**	**100**	**100**	**100**	**283.00**
DataPersisterModule Crash	**0**	**93**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**0**	**100**	**100**	**100**	**100**	**100**	**283.5**
DataRequesterModule Crash	**0**	**77**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**0**	**100**	**100**	**100**	**100**	**100**	**256.89**
PollingSchedduler Module Crash	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**100**	**100**	**100**	**100**	**0**	**0**	**0**	**0**	**0**	**0**	**1.59**
Service Engine Module Crash	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**100**	**100**	**100**	**100**	**0**	**0**	**0**	**0**	**0**	**0**	**1.19**
RpsProbe Failure (dcas rpsProbe)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**100**	**100**	**100**	**100**	**0**	**0**	**0**	**0**	**0**	**0**	**0.0**
Db-dcas Connetion	**0**	**91**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**0**	**100**	**100**	**100**	**100**	**100**	**282.79**

DcasTotal Failure	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**-**
Database Crash	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**100**	**100**	**100**	**100**	**0**	**100**	**100**	**100**	**100**	**100**	**-**
QueueStatusProbe	**13**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**0**	**100**	**100**	**100**	**100**	**100**	**299.79**

The probabilities displayed in the table have been quantified using probabilistic models synthesized from sets of 30 different execution for each scenario included. Our analysis will focus on the NCOP results since is in under this profile that the different adaptation mechanisms were tested. Finally we will analyze the results obtained using COP to assess the impact of the system changes when no adaptations are being used.

First we perform the evaluation by comparing the two different systems (Rainbow-DCAS and Original DCAS) in each case and conclude with an overall analysis of the results.

- **Device Delay.** 25% of the devices fail to respond in a timely manner (2-second delay induced in their respective DataStreams). While in original DCAS the system recovers performance progressively in property P1 (only reaching a 100% probability of recovering above minimum performance threshold after the delay from the devices is removed at time t=600), in Rainbow-DCAS we can observe that that same 100% is achieved by t=150. Regarding property P2, we can observe that original DCAS is less aggressive in adding pollers than Rainbow DCAS, since the probability of adding more than 40 pollers in original DCAS is 0%, compared to the 100% in Rainbow-DCAS. However, property P3 indicates that Original DCAS is not efficient at removing unnecessary pollers after they have been used during the low responsiveness period of the devices, since the probability of going below 10 active pollers after this period is 0%, whereas in the case of DCAS, it is 100%.
- **Device Failure.** In this test the performance of the system is severally affected in both cases, with the system not recovering performance before the delay from the devices is removed. However in property P3 the same analyses from the previous test can be applied. Rainbow-DCAS is more efficient at removing unnecessary pollers.
- **AddPoller Failure.** In this test, without the possibility of adding pollers to recover performance, Original DCAS does not recover during the full period. Rainbow-DCAS only recovers in 59% of the cases before t=600. It becomes clear that the affection of this specific adaptation mechanism has a great impact on the performance of the system.
- **RemovePoller Failure.** Values of properties P1 and P2 seem to be not affected by this failure. This is expected, since they are concerned only with performance and the addition of pollers. However, it is interesting that in P3, the values are of course affected in a negative way. In fact, in this particular case, the probability of reducing the number of pollers below a particular threshold after they are not needed anymore is higher in original DCAS. This is explained because the number of pollers that it adds in the first place is smaller, compared to Rainbow-DCAS which adds pollers more aggressively. Also, Original DCAS can remove several pollers at the same time and Rainbow DCAS only remove one each time)
- **Change RateDelay.** The values of Property P1 for original DCAS degrade noticeably in this case, since the rescheduling mechanism is effectively disabled.

However, this degradation in performance is negligible in the case of Rainbow-DCAS, which compensates the unavailability of the rescheduling mechanism by making a more intensive use of scale-up. This happens because Rainbow can dynamically balance the use of its alternative adaptation mechanisms to achieve the same goal, whereas the hardwired version of the mechanisms in DCAS lacks this capability. The values of properties P2 and P3 are not affected by this failure.

- **DataPersister Failure.** The probability of recovering performance in Original DCAS is degraded initially, since the failure of the DataPersister impedes the writing of device data in the database. However, although one might expect a permanent value of 0% for P1 throughout the experiment, DCAS includes a redundancy mechanism that automatically detects the absence of a DataPersister working properly, and automatically instances a new one, facilitating the progressive recovery of performance. The same happens in the case of Rainbow-DCAS, since it uses the same recovery procedure as original dcas ,although in this case the recovery is much faster, thus the overall behavior will not be affected seriously). Values of P2 and P3 are basically unaffected.
- **DataRequester Failure.** The values of all properties are unaffected both in Original DCAS and Rainbow DCAS. In this case, DCAS implements a similar redundancy mechanism to the one used for the DataPersister. In this case, the effects on performance are less noticeable, since the downtime of the DataPersister until replaced with a new instance does not prevent the writing of values to the database which was already sent to the DataPersister queue before the failure.
- **PollingScheduler.** In this case, performance drops completely in both cases, since the polling scheduler is in charge of triggering requests for data on the data requesters, and without this, the system cannot provide its intended service. In contrast with the data requester and the DataPersister, there is not a redundancy mechanism put in place for the polling scheduler. One interesting difference that can be observed in this case between original DCAS and Rainbow-DCAS is in the values of property P2. Since Rainbow-DCAS has a permanent value of 100% (pollers are added to the maximum supported by the processor node), original DCAS does not add any pollers (0%). This is explained by the fact that the triggering conditions for scale-up are different in DCAS and Rainbow-DCAS. While Rainbow-DCAS looks at the low RPS values and the fact that queues in Data Requesters are not shrinking (trying to solve it by adding more pollers), DCAS looks exclusively at the size of queues in the data requester (which is of course 0). Since the size of the queues is not growing, DCAS considers that adding new pollers is not necessary.
- **ServiceEngine Failure.** Same behavior as in PollingScheduler. Since all data requests travel from the polling scheduler to the data requester through the service engine, obtaining the same results is expected.

- **Database-DCAS connection Failure.** In this case values of all properties are unaffected both in Original DCAS and Rainbow DCAS. Although we should expect permanent value of 0% for P1 throughout the experiment, DCAS includes a protection mechanism that automatically detects if the connection was closed and creates a new connection.
- **DcasTotal Failure.** In this case, performance drops completely in both cases, since all the core modules are shutdown, resulting of an permanent failure on the service.
- **Database Crash.** Since the values used for performance analysis depends on the rps values, crashing the database results in no data being written to the database. This results in a complete drop of the performance. However it is interesting to highlight the P2 properties. Both Rainbow-DCAS and Original DCAS use a great number of pollers by caused by different reasons. Original DCAS add pollers due to the growth in the size of the input Queue. That fact that the requests are not saved into the database results slows the process rate of the requests, affecting the performance. Rainbow-DCAS adds the maximum number of pollers trying to recover the performance to normal values, caused by the rps is 0 to the rest of the test.
- **QueueStatus Probe Failure** Values of properties P1 and P3seem to not be affected by this failure in both cases. However in Rainbow-DCAS, we have a big number active of pollers, even under normal performance. This is caused because Rainbow uses the queueStatus values as a condition to remove pollers. After injecting the failure, the queueStatus value remains high, thus not allowing the system to remove unnecessary pollers.
- **RpsProbe Failure.** This test is used for Rainbow-DCAS since it is required to trigger the adaptation mechanisms. Original DCAS focuses in the inputQueue size as the major variable for triggering conditions. Rainbow DCAS focus in the rps value, using the MIN RPS threshold as a condition to add pollers. Thus, the performance of the system is severally affected in a similar way of the Database Crash. Since the rps value remains under the threshold, Rainbow will continuously try to add pollers.

Uptime property

We define uptime as the total number of seconds where the performance of the system remained above the MIN RPS threshold, thus under normal performance. The time interval for the creation of the probabilistic models was of 600s, so a value of uptime near 600s means the system remained under normal performance most of test. A value near 0s means the system performance depredated and the adaptations didn't manage to recover the performance.

In COP, environment simulation is not applied, thus the system operates under normal conditions, until a system change is injected. Both Original-DCAS and Rainbow-DCAS have uptime values near 600s when confronted with system changes with small impact on the system. However Rainbow-DCAS shows betters results, almost with no modification in the system’s performance. Changes with catastrophic impact on the system (e.g. database Crash) results in a complete failure in both cases, thus with the system not capable of providing its service (uptime value was 0 or even not possible to measure).

In NCOP, when applying environment and system stimulation Rainbow DCAS presented better results than original DCAS, meaning higher values of Uptime. Thus, we can conclude that, under the same conditions and stimulations, the adaptation mechanisms developed in Rainbow-DCAS are more effective than original adaptations, enabling the system to quickly recover performance and at the same time without a large cost associated.

Chapter 5
Conclusions

In this work we evaluated experimentally the effectiveness of an architecture-based approach that evaluates by comparison if the application of architecture-based self-adaptation can improve the resilience of an already adaptive system

. To achieve our goals we used a prototype developed using Rainbow, an architecture-based platform for self-adaptation, and DCAS, an industrial software-intensive system used to monitor and manage highly populated networks of devices in renewable energy production plants, using a quantitative approach based on probabilistic model checking to perform resilience evaluation.

In the first place we developed a prototype of DCAS (Rainbow-DCAS) with different adaptation mechanisms, based on architecture based self-adaptation, removing the existing adaptations from the system.

In second place we compared the adaptations created using Rainbow with the existing adaptations embedded within DCAS to assess whether the use of architecture-based self-adaptation really improved the results of the original adaptation mechanisms in DCAS.

The experimental evaluation showed the effectiveness of evaluating, by comparison, the resilience of adaptation mechanisms when using probabilistic behavioral models for quantifying the probability of system properties being satisfied when under stimulation.

Results showed that applying architecture-based self-adaptation proved to be more effective than original adaptations, improving the resilience of the system. The overall runtime quality of the self-adaptive system can be greatly improved with acceptable costs.

References

[1] A. Jansen e J. Bosch, «Software Architecture as a Set of Architectural Design Decisions», em *5th Working IEEE/IFIP Conference on Software Architecture, 2005. WICSA 2005*, 2005, pp 109 –120.

[2] J. O. Kephart, «Research challenges of autonomic computing», em *27th International Conference on Software Engineering, 2005. ICSE 2005. Proceedings*, 2005, pp 15 – 22.

[3] J. O. Kephart e D. M. Chess, «The vision of autonomic computing», *Computer*, vol 36, n 1, pp 41 – 50, Jan 2003.

[4] R. Sterritt, «Autonomic computing», *Innovations Syst Softw Eng*, vol 1, n 1, pp 79–88, Abr 2005.

[5] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, e P. Steenkiste, «Rainbow: architecture-based self-adaptation with reusable infrastructure», *Computer*, vol 37, n 10, pp 46 – 54, Out 2004.

[6] Y. Qun, Y. Xian-chun, e X. Man-wu, «A framework for dynamic software architecture-based self-healing», em *2005 IEEE International Conference on Systems, Man and Cybernetics*, 2005, vol 3, pp 2968–2972 Vol. 3.

[7] J. L. Hellerstein, «Self-managing systems: a control theory foundation», em *29th Annual IEEE International Conference on Local Computer Networks, 2004*, 2004, p 708–.

[8] M. Tauber, G. Kirby, e A. Dearle, «Self-Adaptation Applied to Peer-Set Maintenance in Chord via a Generic Autonomic Management Framework», em *2010 Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshop (SASOW)*, 2010, pp 9 –16.

[9] F. André, E. Daubert, e G. Gauvrit, «Towards a Generic Context-Aware Framework for Self-Adaptation of Service-Oriented Architectures», em *2010 Fifth International Conference on Internet and Web Applications and Services (ICIW)*, 2010, pp 309 –314.

[10] S.-W. Cheng, D. Garlan, e B. Schmerl, «Making Self-Adaptation an Engineering Reality», em *Self-star Properties in Complex Information Systems*, O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. van Moorsel, e M. van Steen, Eds Springer Berlin Heidelberg, 2005, pp 158–173 [Online]. Disponível em: http://link.springer.com/chapter/10.1007/11428589_11. [Acedido: 27-Fev-2013]

[11] M. Zouari, M. Segarra, e F. André, «A Framework for Distributed Management of Dynamic Self-adaptation in Heterogeneous Environments», em *2010 IEEE 10th International Conference on Computer and Information Technology (CIT)*, 2010, pp 265 – 272.

[12] R. Asadollahi, M. Salehie, e L. Tahvildari, «StarMX: A framework for developing self-managing Java-based systems», em *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09*, 2009, pp 58 –67.

[13] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, e A. L. Wolf, «An architecture-based approach to self-adaptive software», *IEEE Intelligent Systems and their Applications*, vol 14, n 3, pp 54 –62, Jun 1999.

[14] J. Cámara, P. Correia, R. de Lemos, D. Garlan, P. Gomes, B. Schmerl, e R. Ventura, «Evolving an Adaptive Industrial Software System to Use Architecture-based Self-Adaptation», apresentado na SEAMS 2013 (submitted to conference)), p 10.

[15] J.-C. Laprie, «Resilience for the Scalability of Dependability», em *Fourth IEEE International Symposium on Network Computing and Applications*, 2005, pp 5 –6.

[16] «IEEE Standard for Software Maintenance», *IEEE Std 1219-1993*, p 0_1, 1993.

[17] B. P. Lientz, E. B. Swanson, e G. E. Tompkins, «Characteristics of application software maintenance», *Commun. ACM*, vol 21, n 6, pp 466–471, Jun 1978.

[18] J.-M. Desharnais e A. April, «Software maintenance productivity and maturity», em *Proceedings of the 11th International Conference on Product Focused Software*, New York, NY, USA, 2010, pp 121–125 [Online]. Disponível em: http://doi.acm.org/10.1145/1961258.1961289. [Acedido: 30-Ago-2013]

[19] A. April e A. Abran, *Software Maintenance Management: Evaluation and Continuous Improvement*. John Wiley & Sons, 2012.

[20] M. Salehie e L. Tahvildari, «Self-adaptive software: Landscape and research challenges», *ACM Trans. Auton. Adapt. Syst.*, vol 4, n 2, pp 14:1–14:42, Mai 2009.

[21] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, e E. Gjorven, «Using architecture models for runtime adaptability», *IEEE Software*, vol 23, n 2, pp 62–70, 2006.

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

[23] A. G. Ganek e T. A. Corbi, «The dawning of the autonomic computing era», *IBM Syst. J.*, vol 42, n 1, pp 5–18, Jan 2003.

[24] M. M. Gorlick e R. R. Razouk, «Using weaves for software construction and analysis», em *, 13th International Conference on Software Engineering, 1991. Proceedings*, 1991, pp 23 –34.

[25] L. Sha, R. Rajkumar, e M. Gagliardi, «Evolving dependable real-time systems», em *, 1996 IEEE Aerospace Applications Conference, 1996. Proceedings*, 1996, vol 1, pp 335 –346 vol.1.

[26] D. Gupta, P. Jalote, e G. Barua, «A formal framework for on-line software version change», *IEEE Transactions on Software Engineering*, vol 22, n 2, pp 120 –131, Fev 1996.

[27] J. H. Reynolds, «The runtime creation of code for printing simulation output», em *Simulation Conference, 1990. Proceedings., Winter*, 1990, pp 220 –223.

[28] G.-C. Cardarilli, M. Ottavi, S. Pontarelli, M. Re, e A. Salsano, «A fault tolerant hardware based file system manager for solid state mass memory», em *Proceedings of the 2003 International Symposium on Circuits and Systems, 2003. ISCAS '03*, 2003, vol 5, pp V–649–V–652 vol.5.

[29] S. H. Abbas e S.-H. Hong, «A top-down approach to add hot-pluggable asynchronous devices to RAPIEnet infrastructure», em *9th International Symposium on Communications and Information Technology, 2009. ISCIT 2009*, 2009, pp 128–133.

[30] F. Schmied e A. Cyment, «Aspect-oriented weaving and the .NET common language runtime», *IET Software*, vol 1, n 6, pp 251–262, 2007.

[31] P. Padala, «Resource Management in VMware Powered Cloud: Concepts and Techniques», em *2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, 2013, pp 581–581.

[32] K. Mathew, P. Kulkarni, e V. Apte, «Network bandwidth configuration tool for Xen virtual machines», em *2010 Second International Conference on Communication Systems and Networks (COMSNETS)*, 2010, pp 1–2.

[33] M. Badel, E. Gelenbe, J. Leroudier, e D. Potier, «Adaptive optimization of a time-sharing system's performance», *Proceedings of the IEEE*, vol 63, n 6, pp 958–965, 1975.

[34] J. Kramer e J. Magee, «Self-Managed Systems: an Architectural Challenge», em *Future of Software Engineering, 2007. FOSE '07*, 2007, pp 259 –268.

[35] E. M. Dashofy, A. van der Hoek, e R. N. Taylor, «Towards architecture-based self-healing systems», em *Proceedings of the first workshop on Self-healing systems*, New York, NY, USA, 2002, pp 21–26 [Online]. Disponível em: http://doi.acm.org/10.1145/582128.582133. [Acedido: 30-Ago-2013]

[36] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, e M. Shaw, «Engineering Self-Adaptive Systems through Feedback Loops», em *Software*

Engineering for Self-Adaptive Systems, B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, e J. Magee, Eds Springer Berlin Heidelberg, 2009, pp 48–70 [Online]. Disponível em: http://link.springer.com/chapter/10.1007/978-3-642-02161-9_3. [Acedido: 28-Fev-2013]

[37] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. D. M. Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, e J. Whittle, «Software Engineering for Self-Adaptive Systems: A Research Roadmap», em *Software Engineering for Self-Adaptive Systems*, B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, e J. Magee, Eds Springer Berlin Heidelberg, 2009, pp 1–26 [Online]. Disponível em: http://link.springer.com/chapter/10.1007/978-3-642-02161-9_1. [Acedido: 28-Fev-2013]

[38] D. Weyns, M. U. Iftikhar, e J. Söderlund, «Do external feedback loops improve the design of self-adaptive systems? a controlled experiment», em *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, Piscataway, NJ, USA, 2013, pp 3–12 [Online]. Disponível em: http://dl.acm.org/citation.cfm?id=2487336.2487341. [Acedido: 30-Ago-2013]

[39] P. Inverardi e A. L. Wolf, «Formal specification and analysis of software architectures using the chemical abstract machine model», *IEEE Transactions on Software Engineering*, vol 21, n 4, pp 373–386, 1995.

[40] D. Le Metayer, «Describing software architecture styles using graph grammars», *IEEE Transactions on Software Engineering*, vol 24, n 7, pp 521–533, 1998.

[41] M. Luckey, B. Nagel, C. Gerth, e G. Engels, «Adapt cases: extending use cases for adaptive systems», em *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, New York, NY, USA, 2011, pp 30–39 [Online]. Disponível em: http://doi.acm.org/10.1145/1988008.1988014. [Acedido: 30-Ago-2013]

[42] R. Allen, R. Douence, e D. Garlan, «Specifying and analyzing dynamic software architectures», em *Fundamental Approaches to Software Engineering*, E. Astesiano, Ed Springer Berlin Heidelberg, 1998, pp 21–37 [Online]. Disponível em: http://link.springer.com/chapter/10.1007/BFb0053581. [Acedido: 30-Ago-2013]

[43] J. C. Georgas, «Knowledge-based architectural adaptation management for self-adaptive systems», em *Proceedings of the 27th international conference on Software engineering*, New York, NY, USA, 2005, pp 658–658 [Online]. Disponível em: http://doi.acm.org/10.1145/1062455.1062592. [Acedido: 30-Ago-2013]

[44] M. Shaw e D. Garlan, *Software architecture: perspectives on an emerging discipline. 1996*. Prentice-Hall.

[45] D. Garlan, S.-W. Cheng, e B. Schmerl, «Increasing System Dependability through Architecture-Based Self-Repair», em *Architecting Dependable Systems*, R. de Lemos, C. Gacek, e A. Romanovsky, Eds Springer Berlin Heidelberg, 2003, pp 61–89 [Online]. Disponível em: http://link.springer.com/chapter/10.1007/3-540-45177-3_3. [Acedido: 28-Fev-2013]

[46] T. Batista, A. Joolia, e G. Coulson, «Managing Dynamic Reconfiguration in Component-Based Systems», em *Software Architecture*, R. Morrison e F. Oquendo, Eds Springer Berlin Heidelberg, 2005, pp 1–17 [Online]. Disponível em: http://link.springer.com/chapter/10.1007/11494713_1. [Acedido: 28-Fev-2013]

[47] S. Wolff, «The Concept of Resilience», *Aust N Z J Psychiatry*, vol 29, n 4, pp 565–574, Dez 1995.

[48] D. Fletcher e M. Sarkar, «Psychological resilience: A review and critique of definitions, concepts, and theory», *European Psychologist*, vol 18, n 1, pp 12–23, 2013.

[49] R. Plummer e D. Armitage, «A resilience-based framework for evaluating adaptive co-management: Linking ecology, economics and society in a complex world», *Ecological Economics*, vol 61, n 1, pp 62–74, Fev 2007.

[50] J. P. G. Sterbenz, D. Hutchison, E. K. Çetinkaya, A. Jabbar, J. P. Rohrer, M. Schöller, e P. Smith, «Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines», *Computer Networks*, vol 54, n 8, pp 1245–1265, Jun 2010.

[51] Brian Randell, J.-C. Laprie, e A. Avizienis, «Fundamental Concepts od Dependability». [Online]. Disponível em: Basic Concepts and Taxonomy of Dependable and Secure Computing

[52] G. Xu e G. Xiang, «A method of software watermarking», em *2012 International Conference on Systems and Informatics (ICSAI)*, 2012, pp 1791 –1795.

[53] C. Collberg, G. R. Myles, e A. Huntwork, «Sandmark-A tool for software protection research», *IEEE Security Privacy*, vol 1, n 4, pp 40 – 49, Ago 2003.

[54] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, e M. Stepp, «Dynamic path-based software watermarking», em *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, New York, NY, USA, 2004, pp 107–118 [Online]. Disponível em: http://doi.acm.org/10.1145/996841.996856. [Acedido: 28-Jan-2013]

[55] T. Naughton, W. Bland, G. Vallee, C. Engelmann, e S. L. Scott, «Fault injection framework for system resilience evaluation: fake faults for finding future failures», em *Proceedings of the 2009 workshop on Resiliency in high performance*, New York, NY, USA, 2009, pp 23–28 [Online]. Disponível em: http://doi.acm.org/10.1145/1552526.1552530. [Acedido: 28-Fev-2013]

[56] L. Blasi, R. Savola, H. Abie, e D. Rotondi, «Applicability of security metrics for adaptive security management in a universal banking hub system», em *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, New York, NY, USA, 2010, pp 197–204 [Online]. Disponível em: http://doi.acm.org/10.1145/1842752.1842792. [Acedido: 28-Fev-2013]

[57] R. J. Rodríguez, J. Merseguer, e S. Bernardi, «Modelling and analysing resilience as a security issue within UML», em *Proceedings of the 2nd International Workshop on Software Engineering for Resilient Systems*, New York, NY, USA, 2010, pp 42–51 [Online]. Disponível em: http://doi.acm.org/10.1145/2401736.2401741. [Acedido: 28-Fev-2013]

[58] S. Winter, C. Sarbu, B. Murphy, e N. Suri, «The impact of fault models on software robustness evaluations», em *2011 33rd International Conference on Software Engineering (ICSE)*, 2011, pp 51 –60.

[59] A. Schaeffer-Filho, P. Smith, e A. Mauthe, «Policy-driven network simulation: a resilience case study», em *Proceedings of the 2011 ACM Symposium on Applied Computing*, New York, NY, USA, 2011, pp 492–497 [Online]. Disponível em: http://doi.acm.org/10.1145/1982185.1982293. [Acedido: 28-Fev-2013]

[60] K. L. McMillan, «Symbolic Model Checking», em *Symbolic Model Checking*, Springer US, 1993, pp 25–60 [Online]. Disponível em: http://link.springer.com/chapter/10.1007/978-1-4615-3190-6_3. [Acedido: 02-Set-2013]

[61] E. M. Clarke e E. A. Emerson, «Design and synthesis of synchronization skeletons using branching time temporal logic», em *Logics of Programs*, D. Kozen, Ed Springer Berlin Heidelberg, 1982, pp 52–71 [Online]. Disponível em: http://link.springer.com/chapter/10.1007/BFb0025774. [Acedido: 02-Set-2013]

[62] E. M. Clarke, E. A. Emerson, e A. P. Sistla, «Automatic verification of finite-state concurrent systems using temporal logic specifications», *ACM Trans. Program. Lang. Syst.*, vol 8, n 2, pp 244–263, Abr 1986.

[63] C. Baier e J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[64] A. Hinton, M. Kwiatkowska, G. Norman, e D. Parker, «PRISM: A Tool for Automatic Verification of Probabilistic Systems», em *Tools and Algorithms for the Construction and Analysis of Systems*, H. Hermanns e J. Palsberg, Eds Springer Berlin Heidelberg, 2006, pp

441–444 [Online]. Disponível em: http://link.springer.com/chapter/10.1007/11691372_29. [Acedido: 02-Set-2013]

[65] G. Ciardo e A. S. Miner, «A data structure for the efficient Kronecker solution of GSPNs», em *The 8th International Workshop on Petri Nets and Performance Models, 1999. Proceedings*, 1999, pp 22–31.

[66] J. Bachmann, M. Riedl, J. Schuster, e M. Siegle, «An Efficient Symbolic Elimination Algorithm for the Stochastic Process Algebra Tool CASPA», em *SOFSEM 2009: Theory and Practice of Computer Science*, M. Nielsen, A. Kučera, P. B. Miltersen, C. Palamidessi, P. Tůma, e F. Valencia, Eds Springer Berlin Heidelberg, 2009, pp 485–496 [Online]. Disponível em: http://link.springer.com/chapter/10.1007/978-3-540-95891-8_44. [Acedido: 02-Set-2013]

[67] J. Christmansson e R. Chillarege, «Generation of an error set that emulates software faults based on field data», em *, Proceedings of Annual Symposium on Fault Tolerant Computing, 1996*, 1996, pp 304–313.

[68] J. A. Duraes e H. S. Madeira, «Emulation of Software Faults: A Field Data Study and a Practical Approach», *IEEE Transactions on Software Engineering*, vol 32, n 11, pp 849–867, 2006.

[69] B. Schmerl e D. Garlan, «AcmeStudio: supporting style-centered architecture development», em *26th International Conference on Software Engineering, 2004. ICSE 2004. Proceedings*, 2004, pp 704–705.

[70] S.-W. Cheng e D. Garlan, «Stitch: A language for architecture-based self-adaptation», *Journal of Systems and Software*, vol 85, n 12, pp 2860–2875, Dez 2012.

[71] S.-W. Cheng, D. Garlan, e B. Schmerl, «Evaluating the effectiveness of the Rainbow self-adaptive system», em *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09*, May, pp 132–141.

[72] S.-W. Cheng, *Rainbow: Cost-Effective Software Architecture-based Self-adaptation*. ProQuest, 2008.

[73] J. Cámara, R. de Lemos, M. Vieira, R. Almeida, e R. Ventura, «Architecture-based resilience evaluation for self-adaptive systems», *Computing*, vol 95, n 8, pp 689–722, Ago 2013.

[74] J. Cámara, P. Correia, R. De Lemos, D. Garlan, P. Gomes, B. Schmerl, e R. Ventura, «Evolving an adaptive industrial software system to use architecture-based self-adaptation», em *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, Piscataway, NJ, USA, 2013, pp 13–22 [Online]. Disponível em: http://dl.acm.org/citation.cfm?id=2487336.2487342. [Acedido: 28-Ago-2013]

[75] R. Williams, G. Paandelios, e S. Behrens, *Software Risk Evaluation (SRE) Method Description: Version 2.0. Technical report*. Carnegie Mellon University, Software Engineering Institute, 1999.