Empirical Resilience Evaluation of an Architecture-based Self-Adaptive Software System

Javier Cámara
Institute for Software Research
Carnegie Mellon University
jcmoreno@cs.cmu.edu

Pedro Correia
Department of Informatics Engineering
University of Coimbra
pcorreia@dei.uc.pt

Rogério de Lemos
School of Computing
University of Kent
r.delemos@kent.ac.uk

Marco Vieira
Department of Informatics Engineering
University of Coimbra
mvieira@dei.uc.pt

ABSTRACT

Architecture-based self-adaptation is considered as a promising approach to drive down the development and operation costs of complex software systems operating in ever changing environments. However, there is still a lack of evidence supporting the arguments for the beneficial impact of architecture-based self-adaptation on resilience with respect to other customary approaches, such as embedded code-based adaptation. In this paper, we report on an empirical study about the impact on resilience of incorporating architecture-based self-adaptation in an industrial middleware used to collect data in highly populated networks of devices. To this end, we compare the results of resilience evaluation between the original version of the middleware, in which adaptation mechanisms are embedded at the code-level, and a modified version of that middleware in which the adaptation mechanisms are implemented using Rainbow, a framework for architecture-based self-adaptation. Our results show improved levels of resilience in architecture-based compared to embedded code-based self-adaptation.

Categories and Subject Descriptors

D.2.11 [**Software**]: SOFTWARE ENGINEERING—*Software Architectures*; D.2.4 [**Software**]: SOFTWARE ENGINEERING—*Software/Program Verification*

General Terms

Experimentation, Measurement, Reliability

Keywords

Resilience evaluation, Architecture-based self-adaptation, Probabilistic model checking, Rainbow

QoSA'14, June 30–July 4, 2014, Marcq-en-Baroeul, France.

http://dx.doi.org/10.1145/2602576.2602577.

1. INTRODUCTION

During the last decade, the software industry has seen a continuous increase in the complexity of software systems, as well as in the uncertainty of the environments in which they have to operate. This trend has led to an increasing growth in the cost of both developing and operating such systems, but more importantly, has put their *resilience* (*i.e.*, their ability to provide service that can justifiably be trusted when facing changes [24]) in the spotlight as a central concern [13].

Initial approaches to tackle the development and run-time management of complex systems that must operate in ever-changing environments consisted either in making use of human oversight (which is expensive and unreliable), or in embedding low-level error-handling mechanisms in application code that trigger specific responses to anomalies observed in the system at run-time (*e.g.*, exceptions, timeouts). However, although the latter approach can be effective in specific situations, it lacks flexibility and is not well suited to dealing with more subtle, but important kinds of anomaly (*e.g.*, progressive performance degradation).

Autonomic computing, or self-adaptive systems [10, 13, 22], have emerged more recently as an alternative to overcome the shortcomings presented by the aforementioned approaches. In particular, architecture-based self-adaptation [17, 23, 25] leverages architecture models to enable high-level reasoning about the best way of adapting a system, and is regarded as a promising approach to building *resilient* software systems in a cost-effective manner. However, although architecture-based self-adaptation has been applied in practice and recent results indicate its potential benefits in terms of cost compared to embedded code-based adaptation mechanisms [8], there is still a lack of evidence supporting the arguments for its beneficial impact on system resilience [31].

This paper aims at providing empirical evidence that architecture-based self-adaptation has the potential to endow systems with improved levels of resilience, compared to the use of embedded code-based adaptation mechanisms. Our main contribution is an empirical study about the impact of architecture-based self-adaptation on resilience, which employs a framework for evaluating resilience in self-adaptive systems [7] incorporating the notion of *changeload* [1], which includes changes in the system and its environment.

In this study, our framework for resilience evaluation is applied to an adaptive industrial middleware system de-

veloped by Critical Software - called Data Acquisition and Control Service (DCAS), which is used to monitor and manage highly populated networks of devices in renewable energy production plants. Specifically, we compare the results of resilience evaluation between the original version of DCAS in which adaptation is implemented using embedded code-based mechanisms, and a modified version in which adaptation is implemented using the Rainbow framework for architecture-based self-adaptation [17]. Rainbow has been chosen for performing the experiments since its software has been widely available, its structure facilitates access to its internal components, its design is amenable to the injection of faults, and controllers built using it are fairly robust [9].

As a result of our analysis, we demonstrate that incorporating architecture-based self-adaptation in DCAS has a beneficial impact on resilience, mainly due to the: (i) higher level of abstraction of the information used by architecture-based self-adaptation, which enables better informed decision-making and flexibility, and (ii) reduced vulnerability of the external control layer to changes (*e.g.*, failures) occurring at the system level (in contrast with embedded code-based adaptation mechanisms, which are prone to be affected by system failures, thereby interfering with adaptation).

The rest of this paper is structured as follows. Section 2 presents the general structure and objective of DCAS middleware, as well as, its two different versions, *i.e.*, Original DCAS and Rainbow-DCAS, that are implemented using, respectively, embedded code-based and architecture-based adaptation mechanisms. Section 3 introduces resilience properties and their formalization, the notion of changeload, and provides an overview of the process followed for resilience evaluation. Next, Section 4 details the design of our experimental study, whereas Section 5 discusses experimental results. Section 6 discusses threats to validity. Section 7 describes related work. Finally, Section 8 presents some conclusions and indicates directions for future work.

2. DATA ACQUISITION AND CONTROL SERVICE

This section presents the general structure and objective of Data Acquisition and Control Service (DCAS) middleware, as well as, the adaptation mechanisms used in the two versions of the system employed for our study: (i) embedded code-based, and (ii) architecture-based.

2.1 Structure and objective

The Data Acquisition and Control Service (DCAS) [8] is a middleware from Critical Software that provides a reusable infrastructure to manage the monitoring of highly populated networks of devices. In particular, the middleware is designed to be seamlessly integrated with Critical's Energy Management System (csEMS)[1], which 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 with management capabilities that enable the operation of control centers independently of the underlying application (*e.g.*, wind, solar, etc.). The basic building blocks in a DCAS-based system (Figure 1) are: [2]

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

[2] We herein consider a simplified version of the DCAS architecture. Further details about DCAS can be found in [8].

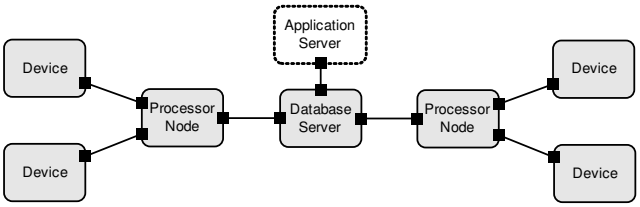

Figure 1: Architecture of a DCAS-based system

- **Devices** are equipped with one or more sensors to obtain data from the application domain (*e.g.*, from wind towers, solar panels, etc.). Each sensor has an associated *data stream* from which data can be read. Each type of device has its particular characteristics (*e.g.*, data polling rate, or expected value ranges) specified in a *device profile*.
- **Processor nodes** pull data from the devices at a rate configured in the device profile, and dispatch this data to the database server. Each processor node includes a set of processes called *Data Requester Processor Pollers* (DRPPs or *pollers*, for short) responsible for retrieving data from the devices. Communication between DRPPs and devices is synchronous, so DRPPs remain blocked until devices respond to data requests or a timeout expires. This is the main performance bottleneck of DCAS.
- **Database server** stores the data collected from devices by processor nodes.
- **Application server** is connected to the database server to obtain data, which can be presented to the operators of the system or processed automatically by application software. However, DCAS is application-agnostic, so the application server will not be discussed in the remainder of this paper.

The main objective of DCAS is collecting data from the connected devices at a rate as close as possible to the one configured in their device profiles, while making an efficient use of the computational resources in the processor nodes. Specifically, the primary concern in DCAS is providing service while maintaining acceptable levels of performance, measured in terms of processed data requests per second (rps) inserted in the database, while the secondary concern is optimizing the computational cost of operating the system, measured in number of active DRPPs in the processor nodes.

2.2 Adaptation Mechanisms

DCAS implements two adaptation mechanisms to keep an acceptable level of performance while making an efficient use of computational resources: (i) *Rescheduling* aims at avoiding performance degradation caused by devices which fail to respond in a timely manner when polled. It consists in decreasing the priority of the data streams associated with the failing devices, so that they are polled less often (thus reducing the average time that DRPPs remain blocked waiting for device data).(ii) *Scale up* aims at improving performance by exploiting as much as possible CPU and memory in processor nodes by (de)activating DRPPs as required.

Original DCAS. Scale up and rescheduling run in two separate control loops embedded in different sub-components of the processor node. Moreover, the C# adaptation logic that corresponds to these control loops is scattered across different parts of the code, and based on low-level information

that indirectly indicates which aspect of the system needs to be improved. In the case of scale up, if the size of a data request queue associated with a particular processor node remains close to zero consistently, the adaptation mechanism considers this as an indicator of good performance, implying that there are active DRPPs which probably are not necessary and have to be deactivated. On the contrary, if the queue size increases consistently, scale up tries to increase performance by activating new DRPPs.

Rainbow-DCAS. Scale up and rescheduling are implemented as adaptation strategies inside of a single control loop running in a controller external to the DCAS system. The adaptation logic is implemented in Stitch [12], a language specifically tailored for representing adaptation strategies in Rainbow. Stitch adaptation strategies make use of system information reified into an architecture model of the underlying DCAS system. Scale up and rescheduling adaptation strategies replicate functionality of the adaptation logic in Original DCAS. However, in the case of scale up, the strategy is also informed by a direct performance measure obtained by a probe that measures the rps in the database, in contrast to the embedded code-based mechanisms, which make use of an indirect performance indicator.

3. RESILIENCE EVALUATION

This section: (i) introduces some general concepts related to resilience evaluation, as well as, the kind of resilience properties that we deal with in our study and their formalization, (ii) describes the notion of changeload, which is central to the framework used for resilience evaluation, and (iii) describes the process followed for resilience evaluation.

3.1 Resilience Properties

A *resilient* system is one that delivers a service that *can justifiably be trusted when facing changes* [24]. In the context of self-adaptive systems, changes (which can occur in the system itself, its environment, or even in its goals) can induce anomalies in the system at run-time, changing its current *operational profile*. Specifically, within a self-adaptive system, we may distinguish between a *conventional operational profile* (COP) that corresponds to the region of the state-space in which the system is operating without experiencing any anomalies, and *non-conventional operational profiles* (NCOPs), associated with anomalies induced by changes in the system or its environment. NCOPs correspond to regions of the state-space in which the system is experiencing a particular anomaly [6]. When the self-adaptive system enters a NCOP, this typically triggers adaptation mechanisms whose purpose is driving the system back into its COP by performing some actions on the system to correct the experienced anomaly. Once the system has entered into a NCOP, resilience can be assessed by quantifying the probability of returning to the system's COP by a given time deadline.

To express resilience properties about the system, we use PCTL [3], a logic language inspired by CTL [15]. With the aid of PCTL, a designer can express properties about the system that are typically domain-dependent. Furthermore, to ease the formulation of probabilistic properties, we make use of property specification patterns [14] that describe generalized recurring properties in probabilistic temporal logics. In our case, we are interested in how the system responds to changes, so we restrict ourselves to properties that can be instanced by using probabilistic response patterns [19],

PCTL Formulation	Description
$\mathcal{P}_{\geq 1}[G(\Phi_1 \Rightarrow \mathcal{P}_{\bowtie p}(F^{\leq t}\Phi_2))]$	**Probabilistic Response.** After state formula Φ_1 holds, state formula Φ_2 must become *true* within time bound t, with a probability bound $\bowtie p$.
$\mathcal{P}_{\geq 1}[G(\Phi_1 \Rightarrow \mathcal{P}_{\bowtie p}(\neg\Phi_2 U^{\leq t}\Phi_3))]$	**Probabilistic Constrained Response.** After state formula Φ_1 holds, state formula Φ_3 must become *true*, without Φ_2 ever holding, within time bound t, with a probability bound $\bowtie p$.

Table 1: Probabilistic response patterns

adapted to PCTL syntax (see Table 1). These patterns include a premise Φ_1 representing the initial conditions after a change in the system or its environment occurs, and a subformula enclosed by the probabilistic operator $\mathcal{P}_{\bowtie p}(.)$, representing the response to that change expected from the system (with a probability bound p and a time bound t).

EXAMPLE 1. *In DCAS, we are interested in assessing how the system reacts to low responsiveness in part of the devices connected to the system. Let* `rps` *be the variable associated with performance (defined as the number of requested data items inserted in the database per second), and* `drpps` *be the cost variable (defined as the number of active DRPPs in the system). We can then define the following predicates:*

$$\texttt{rpsViolation} \triangleq \texttt{rps} < \texttt{MIN\_RPS}$$
$$\texttt{hiCost} \triangleq \texttt{drpps} \geq \texttt{MAX\_POLLERS}$$

where `MIN_RPS` *is a threshold that establishes the minimum acceptable level of performance of the system, and* `MAX_POLLERS` *determines a maximum acceptable number of active pollers. Let us also express a predicate associated with the COP in DCAS as* $\texttt{dcasCOP} \triangleq \neg\texttt{rpsViolation} \wedge \neg\texttt{hiCost}$ *. Based on these predicates, we may instantiate the following PCTL property, using the probabilistic response pattern included in Table 1:*

$$\mathcal{P}_{\geq 1}[G(\texttt{rpsViolation} \Rightarrow \mathcal{P}_{\geq 0.9}(F^{\leq 100}\ \texttt{dcasCOP}\ ))]$$

This property reads as: "When performance falls below threshold `MIN_RPS` *, the probability of raising performance again above* `MIN_RPS` *with a cost below the* `MAX_POLLERS` *threshold within 100 seconds is greater or equal to 0.9". It is worth observing that the instantiation of the probabilistic response patterns, as well as, the predicates used for the specification of the properties can be more general or specific, depending on the particular aspect of the system resilience that we want to study (*e.g.*, we can employ* $\neg$`rpsViolation` *instead of* `dcasCOP` *in the property above if we are interested in evaluating resilience exclusively* w.r.t. *the performance of the system).*

3.2 Changeload

Evaluating the resilience of a self-adaptive system requires identifying the most relevant (sequences of) system or environmental changes that might affect system resilience. Such changes always occur under some system and environment conditions that provide a context for them. A *scenario* is a postulated sequence of events that captures the state of the system, its environment, and its goals during a given time frame, as well as 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 fall into two categories: *base scenarios* and *change scenarios*. A base scenario is defined in terms of typical

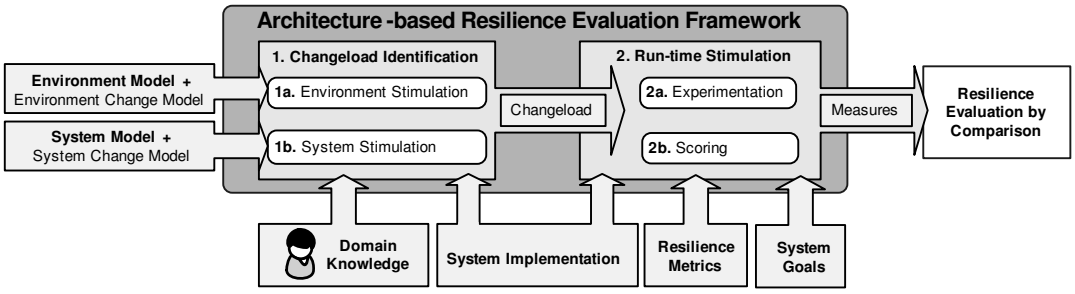

Figure 2: Overview of the resilience evaluation framework

conditions during the execution of the system, which includes: a typical (stable) state of the system and its environment, and a set of goals. [3]

The workload of a base scenario should be representative of the typical amount and type of work assigned to (or expected from) the system during 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, including 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 in which the system faces 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 are derived from a base scenario, but include a representative sequence of changes that may affect the system and its ability to achieve and maintain its goals.

A changeload, is a set of representative change scenarios, comprising changes both in the system and its environment.

3.3 Evaluation Process Overview

Our framework for evaluating the resilience of self-adaptive systems consists of two main stages (Figure 2):

1. **Changeload identification** consists in identifying the sequence of changes (*i.e.*, the changeload) relevant for the run-time stimulation of the system and its environment. This stage requires human intervention and is divided in:
   (a) **Environment Stimulation**, identifies the environmental changes required to drive the environment towards conditions that trigger system adaptations.
   (b) **System Stimulation**, identifies system changes that can affect the ability of the system to return to its COP when experiencing an anomaly.
2. **Run-time stimulation** of the system and its environment, according to the changeload identified. This stage is fully automatic and divided into:
   (a) **Experimentation**, during which the system and its environment are stimulated according to the changeload, and information regarding the system's execution is collected, according to the metrics, as sets of execution traces for each scenario in the changeload. The experimental profile, depicted in Figure 3, includes a set of runs, in which run 0 consists only of performing environmental stimulation to trigger adaptation to collect baseline information about adaptation behavior 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. During Runs 1...N the system is run in such a way that: (i) it executes without stimulation for a *steady state period* to reach a *steady state condition*, (ii) *environment stimulation* is applied to induce the *adaptation trigger condition* required for adaptation (iii) the system detects the adaptation trigger condition and starts the execution of adaptation after a period of *time to react*, (iv) system changes are injected on top of the environmental stimulation during the *time to adapt*, and (v) continues running without further disturbance after adaptation has finished execution during a *keep period* used to collect further data about the effects of adaptation until some *stop condition* is met (*e.g.*, time deadline, etc.). The *measurement interval* in every run starts when the adaptation trigger condition is met, and ends with the stop condition.

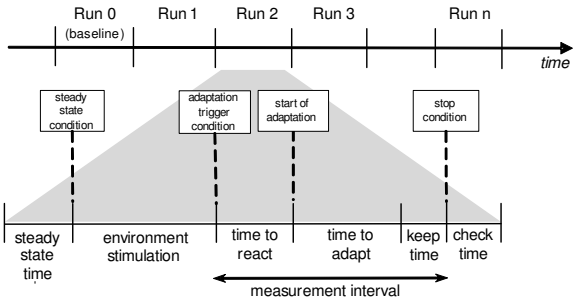

Figure 3: Experimental profile

(b) **Scoring**, where each set of collected traces for a particular scenario is transformed into a probabilistic response model of the system, and used as input to a probabilistic model-checker in addition to the resilience properties obtained from system goals to quantify resilience.

[3]Our approach to resilience evaluation assumes fixed goals.

Our framework enables the evaluation of adaptation by comparison, which makes use of relative resilience metrics to compare how different adaptive solutions respond to a particular set of (system or environmental) conditions. This approach for evaluating resilience is intended to be used by the developer of the system just before its deployment, since the process often involves putting the system through adverse conditions inadequate for production environments.

4. EXPERIMENTAL DESIGN

Resilience is related to the system's ability of maintaining service provision without deviating from the fulfillment of system goals, despite changes that might affect the system or its environment. The primary goal in DCAS is maintaining an acceptable level of performance, therefore we designed our study with a focus on evaluating resilience *w.r.t.* performance. Hence, our experiments are designed to enable us to assess the system's ability to return to its COP, once it has been driven to the NCOP associated with the low performance anomaly `rpsViolation` (as defined in Example 1).

In this context, we formalize the following PCTL properties [4] in accordance with the objective of our study:

- *N1.* To address the primary performance related goal in DCAS, we formalize a property that enables us to quantify the probability of eliminating the `rpsViolation` anomaly (*i.e.*, raising performance again above threshold `MIN_RPS`) within t time units as: $\mathcal{P}(F^{\leq t}\ \neg\texttt{rpsViolation}\ )$.
- *N2.* Moreover, we also want to evaluate resilience in the context of the associated cost of improving performance, so we also quantify the ability of the system to free the resources used for adaptation once they are not required anymore (*i.e.*, deactivation of DRPPs below a threshold α that can be instanced with different values) by a given deadline t as: $\mathcal{P}(F^{\leq t}\ \texttt{drpps} \leq \alpha\ )$.

In our study, we analyze and compare the levels of resilience quantified on the properties described above for both Original DCAS and Rainbow-DCAS, in order to determine whether the incorporation of architecture-based self-adaptation in the system improves its resilience.

Moreover, we provide some context for our evaluation of resilience. We have complemented our study with additional evidence collected according to some of the run-time evaluation criteria proposed in [20] that indicate the ability of the system to return to its COP after a disturbance:

- *WAT.* Working *vs.* adaptivity time of the system, where the working time is the time needed to perform the usual function of the system, and the adaptivity time concerns the time needed to adapt to changes in the system or its environment. In our specific case, we consider as *working time* all the measurement interval in our experiments (time to react + time to adapt + keep time, Figure 3), whereas adaptivity time corresponds to "time to adapt" (working and adaptivity times overlap in DCAS, since the system continues to work while it is adapting).

$$WAT = \frac{WorkingTime}{AdaptivityTime}$$

- *TA.* Time for adaptation, or time required to return to a nominal behavior after a perturbation. In DCAS, this is the time required to return to the COP since the time instant in which the system enters the NCOP associated to anomaly `rpsViolation` .

Finally, we also provide further context for resilience evaluation by including a measure of *experimental availability* taken from dependability benchmarking [28] [5]. In this context, a system is considered to be available when it is able to provide service as specified (*e.g.*, in DCAS, when performance levels are above threshold `MIN_RPS`). We refer to the sum of the duration of all time periods in which the system is available during the working time of the experiments as *uptime*. Experimental availability is defined as:

$$EA = \frac{Uptime}{WorkingTime}$$

The rest of this section details the: (i) identification of the changeload, (ii) procedure followed for run-time stimulation, and (iii) experimental setup used for our study.

4.1 Changeload Identification

This section details first the environmental part of the changeload identified to drive DCAS towards adaptation conditions, followed by system stimulation, in which representative system changes that might affect the resilience of DCAS are identified.

4.1.1 Environment Stimulation

The first step to identify a representative changeload to compare alternative adaptation mechanisms in terms of resilience is determining which environmental changes can trigger adaptations. To this purpose, we need to identify the anomalies associated to NCOPs. In DCAS, the primary focus of adaptation mechanisms is recovering from situations in which the system is running under the acceptable levels of performance. Therefore, in this case we identified potential sources of system states in which anomaly `rpsViolation` holds.

The only representative source of low performance that can be attributed to the environment in DCAS corresponds to the case in which connected devices fail to respond data requests in a timely manner (*e.g.*, due to high network latency, failures in devices, etc.). Specifically, we identified two representative scenarios to stimulate the environment:

- *Device delay*, in which devices fail to respond in a timely manner (inducing a delay of 2 seconds in the response of 25% of connected devices when they are requested for data). The number of devices and the amount of delay applied have been chosen to recreate representative situations in which high network latency is given in DCAS-based systems deployed in the field.
- *Device failure*, where we replicate more extreme situations in which 25% of de devices fail to respond by inducing a 30-second delay in the devices. This causes a timeout in data requests performed by the DRPPs in processor nodes, simulating effectively device failure.

4.1.2 System Stimulation

To identify relevant stimulation of the system, we follow a risk-based approach based on the Software Risk Evaluation

[4]Specifically, we evaluate the results obtained from applying the probabilistic quantifier $\mathcal{P}(.)$ to system response according to the probabilistic response pattern displayed in Table 1. The implicit premise in the properties specified is `rpsViolation` .

[5]Assessing availability in global terms is difficult since it depends on many factors that influence the system Mean Time Between Failures (MTBF).

(SRE) method [32] that considers both the probability and impact of system changes. This is a manual process that uses field data (if available) and expert knowledge, and consists in identifying relevant system changes that might impact system goals during adaptation. Field data was unavailable for DCAS, so the probability and impact of system changes was analyzed with the help of field experts.

The candidate system changes for system stimulation were classified in a risk exposure matrix for the case of the NCOP associated to the anomaly `rpsViolation` (Table 2). Grayed-out cells in the exposure matrix area contain changes which are left out of the changeload due to their low representativeness when considering the combination of their impact and probability of occurring.

We can observe in the matrix that, when facing performance issues, failures in the components involved in adaptation have a high probability of occurring. Moreover, these failures might have a critical impact if they fail to respond properly since they are used intensively during adaptation. Such is the case of the effectors used to activate DRPPs or the probes to check the status of data request queues, which are intensively used during scale up adaptation. Other effectors and probes, such as, the ones used for rescheduling (device delay probe and change rate delay effector) have also a high probability of occurring, but a marginal or negligible impact on performance, since the effects of rescheduling are limited compared to those of scale up. The rest of system changes, included in the white area of the exposure matrix, involves failures in different system coarse-grained components (*e.g.*, processor node, database server), and finer-grained sub-components of the processor node (*e.g.*, service engine, data requester, data persister, polling scheduler) all with potentially either critical or even catastrophic impact on service provision[6] (*e.g.*, a crash in the database server would reduce rps to 0 by impeding insertions the database).

4.2 Run-time Stimulation

To evaluate the resilience of the two different versions of DCAS, we carried out run-time stimulation using a changeload in which all scenarios include a workload and operating conditions characteristic of a typical deployment of a DCAS-based system in production. Scenarios have a duration of 40 minutes (2400s), which is enough to collect sufficient data to characterize system behavior and synthesize the probabilistic models required to quantify resilience. All experiments incorporate a workload that includes 100 data streams (devices) with a data polling rate of 1 second.

In our experiments, the system is driven towards the triggering of adaptation to improve performance, and in which scenarios conform to the following pattern: (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; and (iii) 1200s of normal activity. Scenarios in our experiments are divided in two groups:

1. Environment stimulation. Contains 2 scenarios including only environment stimulation, according to the cases described in Section 4.1.1 (device delay and device failure).
2. Environment+System stimulation. Contains 11 scenarios. Each scenario combines environmental stimulation (device delay) with one of the system changes identified as relevant in the exposure matrix shown in Table 2. Environment stimulation is used in these scenarios as a way to trigger system adaptation without interfering with the system. This avoids potential interactions between system changes triggering adaptation conditions, and those applied during the execution of adaptation. Specifically, we have favored the use of device delay over device failure as environmental stimulation because it is enough to trigger adaptation, but at the same time has a more moderate impact, enabling us to better assess the contribution of system changes to variation in resilience levels.

For every scenario, we built a probabilistic model of system behavior during a period of 900s, which corresponds to system traces collected during the time frame $[600, 1500]$ of every system run, out of which the first 600s correspond to the disturbance period. Each probabilistic model has a time discretization parameter of $\tau = 1s$, and quantization parameters for the performance and cost variables of $\eta_{\mathsf{rps}} = 10$ and $\eta_{\mathsf{drpps}} = 1$, respectively[7]. Each model is synthesized from data obtained from 30 different runs of the same scenario (*i.e.*, our experiments required (11+2)*30*2=780 runs).

4.3 Experimental Setup

For our experimental setup, we deployed both versions of DCAS across three different machines. In the case of Original DCAS (Figure 4, left), dcas-db acts as the backend database running on Oracle 10.2.0, dcas-main acts as a processor node, running DCAS, and (dcas-devs) is used to simulate the response of network devices from which DCAS retrieves information (device response simulation is implemented as a simple Web service whose response time can be set in a configuration file). In the case of Rainbow-DCAS (Figure 4, right), Rainbow's master is deployed in an additional machine (dcas-master). All machines run on Windows XP Pro SP3 (DCAS is deployed as a Windows service), and an Intel core i3 processor, with 1GB of RAM.

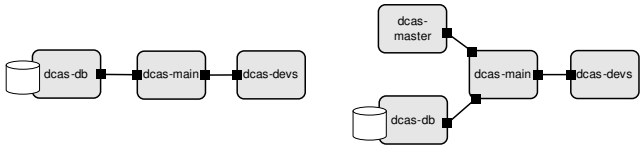

Figure 4: Experimental setup: Original DCAS (left) and Rainbow-DCAS (right)

5. EXPERIMENTAL RESULTS

In line with the system's goals of keeping an acceptable performance level while keeping down the cost of running the system, we study the resilience of Original DCAS and Rainbow-DCAS in the different scenarios identified in our changeload. Specifically, in this section we report on: (i) results regarding general run-time evaluation criteria related to the system's ability to return to its COP, and (ii) results regarding the evaluation of the resilience properties defined according to DCAS goals, described in Section 4.

5.1 General Run-time Evaluation Criteria

Table 3 displays the experimental results for WAT, TA, and EA both for Original DCAS and Rainbow-DCAS.

[6]A detailed discussion of the functionality of the different sub-components of the processor node can be found in [8].

[7]Details about time sampling, quantization, and the type of probabilistic models used for our study can be found in [7].

Impact	Probability: Very High	High	Low	Very Low
Catastrophic			DB Crash Service Engine Crash	Processor Node Crash
Critical		Add Poller Effector Crash DB-DCAS Conn. Shutdown Queue Status Probe Crash	Polling Scheduler Crash Data Persister Crash Data Requester Crash	
Marginal		ChangeRateDelay Effector Crash Remove Poller Effector Crash	System Monitor Crash	Alarmer Crash Alarmer Publisher Crash
Negligible		Device Delay Probe Crash	Device Monitor Crash	Alarmer Monitor Crash

Table 2: Exposure matrix for system changes in DCAS non-conventional operational profile `rpsViolation`

	WAT. Working *vs.* Adaptivity Time		TA. Time for Adaptation (s)		EA. Exp. Availability (%)	
	Orig.DCAS	Rainbow-DCAS	Orig.DCAS	Rainbow-DCAS	Orig.DCAS	Rainbow-DCAS
Device Delay	1.247	6.185	243	80	19	84
Device Failure	$\rightarrow 1$	$\rightarrow 1$	∞	∞	0	0
AddPoller Eff. Crash	$\rightarrow 1$	$\rightarrow 1$	∞	∞	0	0
Remove Poller Eff. Crash	1.200	7.058	362	88	15	45
ChangeRateDelay Eff. Crash	1.242	6.741	336	60	19	85
QueueStatus Probe Crash	1.226	8.695	291	62	18	88
DB-DCAS Conn. Shutdown	1.115	6.818	348	51	10	85
DB Crash	$\rightarrow 1$	$\rightarrow 1$	∞	∞	0	0
Service Engine Crash	$\rightarrow 1$	$\rightarrow 1$	∞	∞	0	0
Polling Scheduler Crash	$\rightarrow 1$	$\rightarrow 1$	∞	∞	0	0
Data Persister Crash	1.176	7.407	357	54	11	86
Data Requester Crash	1.123	6.060	315	72	15	83
Processor Node Crash	$\rightarrow 1$	$\rightarrow 1$	∞	∞	0	0

Table 3: Experimental results for general run-time evaluation criteria and availability

Regarding WAT, it can be observed that Rainbow-DCAS spends in most cases from 6 to 8 times more time working than adapting, whereas the original version of DCAS is prone to spend most of the time adapting to changes, with WAT values always close to 1. It is worth observing that some specific cases in which disturbances cannot be dealt with by adaptation mechanisms in both versions of DCAS (*e.g.*, device failure, service engine crash) make the system spend most of the working time also trying to (unsuccessfully) adapt to changes, resulting in WAT values that closely approach 1 (indicated by $\rightarrow 1$ in Table 3).

Concerning TA, Rainbow-DCAS also takes considerably less time to eliminate the anomaly, with times that range between 50 and 90 seconds *vs.* the 240-350 seconds required by Original DCAS, depending on the case. Cases in which adaptation mechanisms were unable to eliminate the performance anomaly are indicated with an infinite TA in the table. An interesting phenomenon that can be initially perceived as somewhat strange in the case of Rainbow-DCAS is that there are cases that yield a shorter TA in the presence of system and environment changes (*e.g.* data persister crash) *w.r.t.* cases in which there is only environment stimulation (device delay). This happens because adverse system conditions can influence the decision-making process in Rainbow (*e.g.*, the controller can try to further compensate the adverse situation by increasing the rate at which DRPPs are activated), resulting in a more aggressive adaptation *w.r.t.* cases in which decision-making is not influenced by such unfavorable changes. This phenomenon is in line with previous experience using Rainbow in other case studies [7].

Finally, it can also be observed that availability is much higher in Rainbow-DCAS, with values ranging between 80-90% in most cases in which the system is able to adapt, compared to the 10-20% in Original DCAS.

5.2 Resilience Evaluation

Table 4 shows the experimental results for resilience evaluation in the NCOP associated with the `rpsViolation` anomaly for Original DCAS (top), and Rainbow-DCAS (bottom).

Performance-related property $N1$ is instanced with different time bounds (t), in intervals of 50s, and up to time 600s, coinciding with the end of environmental stimulation. Each instance of the property describes the probability of recovering an acceptable performance by the given time bound.

Results for property $N1$ show that in general, resilience values obtained in Rainbow-DCAS tend to be higher than in Original DCAS in many of the scenarios.

Regarding scenarios that only include environment stimulation, we can observe that in the case of the device delay scenario, Rainbow-DCAS reacts much faster to the anomaly with values for $N1$ already of 90% by $t = 100$, whereas Original DCAS only reaches values above 90% by $t = 400$. This is motivated by the fact that in Rainbow-DCAS, the controller can exploit an explicit model of the system's expected behavior (in this case expected performance), as well as high-level information about the current performance of the system (updated at run-time in the system's architecture model). Having this explicit information readily available to the controller enables early detection of anomalies in the case of architecture-based self-adaptation. In contrast, embedded code-based adaptation mechanisms in Original DCAS do not have a global picture of the system's state, and for that, they have to rely on low-level local information that implicitly indicates potential performance problems (*e.g.*, data request queue size growth rate). In this case, there is a time gap between the occurrence of the anomaly causing the performance problem, and the time by which queue growth rate increases enough to trigger adaptation mechanisms. This results in late detection of the anomaly and increased reaction time of adaptation, hindering the ability of the system to recover its intended performance levels in a timely manner.

In the case of device failure, the results obtained are similar for both versions of the system since the changeload cannot be accommodated with the computational resources available, independently of the adaptation mechanism used.

In the case of scenarios that combine system and environment stimulation (device delay), we can observe that in some of the scenarios that involve component crashes in the system (*e.g.*, Data Persister and Data Requester), Rainbow-DCAS does not show any noticeable degradation of performance with respect to the device delay scenario. However, performance in Original DCAS is further degraded, showing values of 50% and 68% by $t = 400$ in the scenarios for the Data Persister and Data Requester crashes respectively,

		N1.$\mathcal{P}(F^{\leq t}$ ¬rpsViolation)												N2.$\mathcal{P}(F^{\leq 600}$drpps $\leq \alpha$)			
		t=50	t=100	t=150	t=200	t=250	t=300	t=350	t=400	t=450	t=500	t=550	t=600	α=10	α=20	α=30	α=40
Orig. DCAS	Device Delay	0	7	20	37	58	70	85	93	97	97	97	100	0	93	100	100
	Device Failure	0	0	0	0	0	0	0	0	0	0	0	0	53	93	100	100
	AddPoller Eff. Crash	0	0	0	0	0	0	0	0	0	0	0	0	100	100	100	100
	Remove Poller Eff. Crash	0	0	3	7	26	30	37	50	53	53	56	93	20	33	71	100
	ChangeRateDelay Eff. Crash	0	0	0	0	13	26	45	47	53	76	78	98	27	100	100	100
	QueueStatus Probe Crash	0	0	7	17	35	40	50	59	88	94	96	100	20	97	100	100
	DB-DCAS Conn. Shutdown	0	0	0	3	7	17	23	25	33	49	60	95	100	100	100	100
	DB Crash	0	0	0	0	0	0	0	0	0	0	0	0	43	93	100	100
	Service Engine Crash	0	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-
	Polling Scheduler Crash	0	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-
	Data Persister Crash	0	0	0	0	17	28	40	50	60	64	78	100	28	100	100	100
	Data Requester Crash	0	7	13	26	40	53	59	68	74	85	90	99	16	95	100	100
	Processor Node Crash	0	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-
Rainbow-DCAS	Device Delay	0	90	100	100	100	100	100	100	100	100	100	100	100	100	100	100
	Device Failure	0	0	0	0	0	0	0	0	0	0	0	0	100	100	100	100
	AddPoller Eff. Crash		0	0	0	0	0	0	0	0	0	0	0	100	100	100	100
	Remove Poller Eff. Crash	0	92	100	100	100	100	100	100	100	100	100	100	0	0	10	100
	ChangeRateDelay Eff. Crash	0	96	97	100	100	100	100	100	100	100	100	100	100	100	100	100
	QueueStatus Probe Crash	13	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
	DB-DCAS Conn. Shutdown	0	91	100	100	100	100	100	100	100	100	100	100	100	100	100	100
	DB Crash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Service Engine Crash	0	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-
	Polling Scheduler Crash	0	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-
	Data Persister Crash	0	93	100	100	100	100	100	100	100	100	100	100	100	100	100	100
	Data Requester Crash	0	77	100	100	100	100	100	100	100	100	100	100	100	100	100	100
	Processor Node Crash	0	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-

Table 4: Experimental results for resilience evaluation

v.s. the 93% shown for that same time bound in the device delay case. Considering that the Data Persister and Data Requester components are in charge of requesting data to devices and dispatching the processed data to the database, respectively, one might expect a radical drop of the values in $N1$ to 0% throughout these scenarios. However, both versions of DCAS include a redundancy mechanism that detects the absence of a Data Persister or a Data Requester working properly, and automatically instances a new one, facilitating the progressive recovery of performance. In Original DCAS, performance recovery is slower because the implementation of the scale up adaptation mechanism is embedded within the failing component, and therefore needs to be restarted and run for some time until it stabilizes again. In contrast, scale up in Rainbow-DCAS is implemented in the Rainbow controller external to the target system, which is not directly affected by the failure in the component.

Regarding the cost-related property *N2*, we can observe that the probabilities of deactivating DRPPs that are not required is 100% even below the minimum threshold of pollers ($\alpha = 10$) almost in all cases of Rainbow-DCAS. There are two cases which are an exception. First, as expected, in the case in which the remove poller effector crashes no pollers can be removed and the system remains at the maximum level of active pollers that were required while scaling up (always above 20, as indicated by the probability 0% of $N2$ when $\alpha = 20$). Second, in the case in which the database crashes, the lack of data item insertions in the database is perceived by the controller as a lack of performance that has to be corrected, impeding the deactivation of pollers. Cases in which the number of pollers cannot be monitored due to the malfunction of the failing processor node (sub)component (cases service engine crash, polling scheduler crash, and processor node crash) are indicated by "-" in the table.

In Original DCAS, the probability of reducing the number of pollers below all values of threshold α tends to be smaller than in Rainbow-DCAS, mainly because detecting when pollers are not needed anymore by looking exclusively at queue growth rates is not as efficient as factoring in explicit performance information. However, the cases for the database crash and the remove poller effector crash are exceptions. In the database crash, the fact that Original DCAS does not use performance information probed in the database to (de)activate pollers avoids the activation of additional pollers by the scale-up mechanism. For the crash in the remove poller effector, these higher probabilities are a consequence of the scale-up mechanism not activating initially as many pollers as in Rainbow-DCAS for the same situation.

6. THREATS TO VALIDITY

Regarding the internal validity of our study, the main concern is related to determining whether the improvement observed in resilience values in the case of Rainbow-DCAS *w.r.t.* Original DCAS is indeed caused by incorporating architecture based self-adaptation *vs.* other factors, such as, the lack of equivalence between the adaptation logic implemented in Original DCAS and the Stitch strategies implemented in Rainbow-DCAS. In this regard, rescheduling and scale-up adaptation logic in Rainbow-DCAS replicates as closely as possible the original adaptation logic in their respective embedded code-based counterparts. However, it is important to identify two main differences between the alternative implementations of adaptation mechanisms:

1. In Original DCAS, each adaptation mechanism resides in its own independent control loop in different sub-components of the processor node. This is an imposition of the OOP paradigm used to develop DCAS, which enforces encapsulation and information hiding, favoring good modularization, but also constrains the access of embedded adaptation mechanisms to information (e.g., for anomaly detection) and restricts actuation to their local scope (hampering coordinated adaptation). In contrast, the two adaptation mechanisms in Rainbow-DCAS reside within the same control loop in the external control layer that decides which one should be used, in a coordinated manner.
2. As a consequence of the limited scope of embedded adaptation mechanisms, Original DCAS can only use low-level information that indirectly indicates the system's performance (e.g., queue sizes). However, Rainbow-DCAS has access to high level information about whether performance goals are being met.

In our opinion, the abovementioned differences in the im-

plementation of adaptation mechanisms in Rainbow-DCAS *w.r.t.* Original DCAS do not undermine the internal validity of the study. Regarding (1), it could be argued that since Rainbow is a centralized controller, its resilience compared to a decentralized control alternative might be worse, because it represents a single point of failure. However, in our study we consider resilience in the presence of changes only at the target system level and its environment (leaving controller failures out of the scope of this paper). Moreover, it is worth emphasizing that in Original DCAS, adaptive mechanisms are embedded in different target system components and are therefore prone to be affected by target system failures. Concerning (2), the ability to factor high level information, like performance, into the decision-making process is possible because of architectural descriptions. These descriptions allow systematical reasoning in terms of the actual goals of the system, rather than ad-hoc about low-level, indirect indicators.

Regarding external validity, the main concern might be the limited scope of our study, since it is restricted to:

1. A particular class of systems. Our results are set in the context of DCAS and Rainbow. Generalization requires experimenting with further types of controllers and systems. However, despite the recent appearance of other frameworks for developing self-adaptive systems [27, 2], Rainbow is the only one that has been widely available and evaluated in real-world scenarios [11, 8].
2. A set of (representative) change scenarios. This restriction stems from the large number of potential changes (especially in complex systems). This is an issue pervasive to many testing techniques in which not all inputs to or paths through a program can be tested. In practice, many of the changes identified by DCAS engineers present a low probability of occurrence or have a low impact in the system. Hence, the adequate use of a risk-based approach analogous to the Software Risk Evaluation (SRE) method, proposed by SEI, enables the selection of the most representative change scenarios.

7. RELATED WORK

There are some recent contributions that deal with the evaluation of self-adaptive systems.

The criteria for evaluation of adaptive properties presented in [20] and [30] aim at assessing the impact of self-* properties on different aspects of the system, such as, performance, in addition to comparing the adaptive features of different systems. Concretely, the criteria in [20] are grouped in different categories, among which "run-time evaluation" is the one that has a stronger relation with resilience. The definition of the different criteria relies on concepts such as *self-* situations* and *nominal situation*, comparable to NCOPs and COP, respectively (even if they are not formally defined).

The set of metrics presented in [26] aims at evaluating the adaptability of software at the architectural level, defining a relationship between the values of the proposed metrics and QoS levels that the system must guarantee. Although this approach is intended to help architects in the generation of adaptable architectures at development-time, the authors propose as future work its integration at run-time, extending it with the metrics presented in [20].

Other contributions, based either on probabilistic modeling or direct measurement of an existing system rely on the analysis of non-functional properties. Modeling approaches are useful during development, but heavily rely on parameter estimations obtained from domain experts or existing similar systems. For instance, Calinescu and Kwiatkowska [5] introduce an autonomic architecture that uses Markov-chain quantitative analysis to dynamically adjust the parameters of an IT system according to its environment and goals. However, this approach requires the specification of Markov-chains for describing the probabilistic behavior of components of the system. Concerning direct measurement, Epifani *et al.* [16] present a framework to keep models alive updating their internal parameters with run-time data. The framework uses Discrete-Time Markov Chains (DTMCs) and Queuing Networks to reason about reliability and performance. The approach by Calinescu *et al.* [4] combines [5] and [16] for defining a framework for developing adaptive service-based systems in which QoS requirements are translated into probabilistic temporal logic formulae used for identifying optimal system configurations.

Our resilience evaluation framework focuses on quantitative analysis using measurements, and does not assume the existence of Markov-chains describing the behavior of system components. Moreover, while other proposals deal with estimates of the future system behavior for optimizing its operation, our approach focuses on evaluating levels of confidence *w.r.t.* the self-adaptive capabilities of the system.

Another area related to resilience evaluation is resilience benchmarking, which encompasses techniques from prior efforts in performance [18], dependability [21], and security benchmarking [29]. Compared to established benchmarks, a resilience benchmark is specified following the same basic approach, but comprising a wide-ranging changeload (including, but not limited to, faults) and resilience metrics [1].

In our study, we use an architecture-based framework that evaluates by comparison the resilience of adaptation mechanisms of a self-adaptive software system [7].

8. CONCLUSIONS

In this paper, we have reported on our experience evaluating the resilience of a self-adaptive industrial middleware (DCAS) employed for collecting and processing data in highly populated networks of devices. Concretely, we have compared the resilience of the original version of DCAS (Original DCAS) in which adaptation mechanisms are implemented as embedded code-based logic, against a version in which adaptation is performed by an external controller implemented using the Rainbow framework (Rainbow-DCAS), which relies on architecture-based self-adaptation.

The empirical evidence of this evaluation indicates a positive impact of architecture-based self-adaptation on resilience, showing a substantial increment in the resilience values obtained in Rainbow-DCAS *w.r.t.* Original DCAS. In particular, resilience evaluation results regarding performance reveal a much faster recovery of acceptable performance levels in half of the scenarios used for our experiments. These results are consistent with the measures obtained for general run-time evaluation criteria and availability, which also show a remarkable improvement in Rainbow-DCAS, thus reinforcing the results of resilience evaluation.

We identified two main factors that influence improvement in resilience evaluation results in the case of Rainbow-DCAS. First, the controller has access to a global picture of the state of the system and its environment, reflected in the architec-

ture model of the system updated at run-time. This enables faster anomaly detection and better-informed decision making based on an explicit model of the expected behavior of the system *vs.* the use of low-level, local information that is only an indirect indicator of whether the system goals are being met. Second, the fact that the adaptation logic resides within a control layer external to the target system reduces its vulnerability to failures at the target system. This can be observed if we consider for instance the crash of a component in Original DCAS that includes embedded adaptation logic. In such cases, not only the capability of proper repair cannot be relied upon (since part of the adaptation logic is compromised), but even if the system can recover, the affected part of the adaptation logic has to be restarted, requiring a ramp-up period until it achieves stable operation.

Regarding future work, we plan on: (i) investigating how further run-time evaluation criteria for adaptive properties presented in [20] and [30] can be instanced in the context of our resilience evaluation framework, and (ii) evaluate these additional properties in DCAS. Studying such properties in the context of DCAS will not only enable us to explore any potential correlations between resilience and adaptive properties, but will also provide further evidence about the potential improvement on other adaptive properties resulting from incorporating architecture-based self-adaptation. Finally, we also plan to include other controller types and systems to confirm the generality of our results.

9. ACKNOWLEDGMENTS

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

10. REFERENCES

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

[2] R. Asadollahi, M. Salehie, and L. Tahvildari. Starmx: A framework for developing self-managing java-based systems. In *SEAMS*. IEEE, 2009.

[3] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[4] R. Calinescu et al. Dynamic QoS Management and Optimization in Service-Based Systems. *IEEE Trans. Software Eng.*, 37(3), 2011.

[5] R. Calinescu and M. Z. Kwiatkowska. Using Quantitative Analysis to Implement Autonomic IT Systems. In *ICSE*. IEEE, 2009.

[6] J. Cámara and R. de Lemos. Evaluation of Resilience in Self-Adaptive Systems Using Probabilistic Model-Checking. In *SEAMS*. IEEE, 2012.

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

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

[9] J. Cámara et al. Robustness evaluation of controllers in self-adaptive software systems. In *LADC*. IEEE, 2013.

[10] B. H. Cheng et al. SEfSAS. volume 5525 of *LNCS*, chapter Software Engineering for Self-Adaptive Systems: A Research Roadmap. Springer, 2009.

[11] S.-W. Cheng et al. Evaluating the Effectiveness of the Rainbow Self-Adaptive System. In *SEAMS*. IEEE, 2009.

[12] S.-W. Cheng and D. Garlan. Stitch: A language for architecture-based self-adaptation. *J. Syst. Software*, 85(12), 2012.

[13] R. de Lemos et al. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In *SEfSAS 2*, number 7475 in LNCS. Springer, 2012.

[14] M. B. Dwyer et al. Patterns in Property Specifications for Finite-State Verification. In *ICSE*, 1999.

[15] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3), 1982.

[16] I. Epifani et al. Model Evolution by Run-Time Parameter Adaptation. In *ICSE*. IEEE CS, 2009.

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

[18] J. Gray. *Benchmark Handbook: For Database and Transaction Processing Systems*. Morgan Kaufmann, 1992.

[19] L. Grunske. Specification Patterns for Probabilistic Quality Properties. In *ICSE*. ACM, 2008.

[20] E. Kaddoum et al. Criteria for the evaluation of self-* systems. In *SEAMS*. ACM, 2010.

[21] K. Kanoun and L. Spainhower. *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society Pr, 2008.

[22] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36, 2003.

[23] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE*, 2007.

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

[25] P. Oreizy et al. An architecture-based approach to self-adaptive software. *IEEE Intell. Syst.*, 14, 1999.

[26] D. Perez-Palacin et al. Software architecture adaptability metrics for qos-based self-adaptation. In *QoSA/ISARCS*. ACM, 2011.

[27] G. Tamura et al. Improving context-awareness in self-adaptation using the dynamico reference model. In *SEAMS*, 2013.

[28] M. Vieira and H. Madeira. A dependability benchmark for oltp application environments. In *VLDB*. VLDB Endowment, 2003.

[29] M. Vieira and H. Madeira. Towards a security benchmark for database management systems. In *DSN*. IEEE CS, 2005.

[30] N. M. Villegas et al. A framework for evaluating quality-driven self-adaptive software systems. In *SEAMS*. ACM, 2011.

[31] D. Weyns and T. Ahmad. Claims and evidence for architecture-based self-adaptation: A systematic literature review. In *ECSA*, volume 7957 of *LNCS*. Springer, 2013.

[32] R. Williams et al. *Software Risk Evaluation (SRE) Method Description: Version 2.0*. CMU-SEI, 1999.