# Requirements and Architectural Approaches to Adaptive Software Systems: A Comparative Study

Konstantinos Angelopoulos[1], Vítor E. Silva Souza[2,1], João Pimentel[3,1]
[1] Department of Information Engineering and Computer Science, University of Trento, Italy
[2] Computer Science Department, Federal University of Espírito Santo (Ufes), Vitória, Brazil
[3] Centro de Informática, Universidade Federal de Pernambuco (UFPE), Recife, Brazil
angelopoulos@disi.unitn.it, vitorsouza@inf.ufes.br, jhcp@cin.ufpe.br

***Abstract*—The growing interest in adaptive software systems has resulted in a number of different proposals for the design of adaptive systems. Some approaches adopt architectural models, whereas others model adaptation options, at the level of requirements. This dichotomy has motivated us to perform a comparative study between two proposals for the design of adaptive systems: the Rainbow Framework (architecture-based) and our own proposal, *Zanshin* (requirements-based). This evaluation paper reports on our methodology and results. It also provides a comparison between the use of architectural and requirements models as centrepieces of adaptation, offering guidelines for the future research in the field of adaptive systems.**

***Index Terms*—Adaptive systems, adaptation, requirements, architecture, *Zanshin*, Rainbow, comparative study**

## I. Introduction

The past decade, has seen a growing interest in adaptive software systems, i.e., systems that can adapt to changes in their environment or their requirements in order to continue to fulfil their mandate. Researchers involved in this area have published several different proposals for the design of such systems, as can be seen in surveys and roadmap papers in the literature, such as [1], [2], [3], [4], [5].

Among the many proposals, intended to guide developers in the construction of adaptive systems, some focus on architectural models that capture architectural variability and support architectural reconfigurations, propagating the effects to the actual system, in response to certain situations. Instead, other approaches, advocate the use of requirements models to capture variability and support adaptation.

This dichotomy has motivated us to investigate whether these two types of approaches can produce the same results, what are their respective advantages and drawbacks, and study whether they are complementary.

As a first step in finding answers to these questions, in this paper we present a comparative study of one representative approach of each of the aforementioned categories, respectively: *Rainbow* [6] and *Zanshin* [7]. Our methodology consisted of applying both frameworks to the same exemplar: the *ZNN.com* case study presented in [8] for the *Rainbow* framework. Models of the system's adaptation rules were produced for each framework and adaptation scenarios based on an implementation of *ZNN.com* were executed.

The remainder of this paper is structured as follows: Section II summarizes the chosen adaptation approaches for this comparative study; Section III presents the *ZNN.com* exemplar and its solutions in the chosen architecture- and requirements-based approaches; Section IV details the methodology and the results of this study, discussing the questions mentioned above; Section V presents related work; finally, Section VI concludes.

## II. Selected Adaptation Approaches

Among existing approaches for the design of adaptive systems, some of which will be discussed in Section V, we are interested in comparing two kinds, namely:

- ***Requirements-based*** (henceforth ***RE-based***) approaches: extend Requirements Engineering techniques in order to represent the requirements of adaptation and/or the inherent uncertainty of the environment in which the system operates. These approaches may or may not include mechanisms for run-time reasoning and frameworks that operationalise the adaptation requirements, since they focus on capturing and analysing the problem rather than implementing solutions;
- ***Architecture-based*** approaches: concentrate on helping designers build architectures that support adaptation. They usually propose the use of an architectural model that shows system components and how they communicate amongst themselves through connectors. Such proposals often include the runtime software infrastructure on top of which to build the adaptive system, taking care of its adaptation rules and how to evolve its models.

For our comparative study, we have chosen one approach for each type, namely *Zanshin* and *Rainbow* respectively. These frameworks were chosen for a number of reasons. Firstly, they are good representatives of their respective schools of thought on building adaptive software systems. Secondly, they are fairly comprehensive and quite well documented in guiding the design of adaptive systems. Thirdly, there was code readily available for running our experiments. We summarise both approaches next.

### A. Rainbow

The *Rainbow* framework [6] is a prominent architecture-based approach for the design of adaptive systems. According to the proposal, adaptation rules are used to monitor the operational conditions of the system and define actions to be taken if the conditions are unfavorable. For example, given a



Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

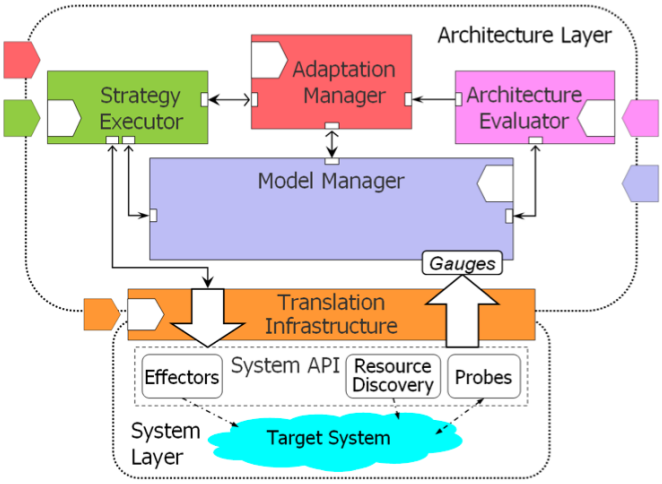


Fig. 1. The components of the *Rainbow* framework [8].

news website (which we will detail in Section III), if measured response times are too long, actions such as enlisting more servers or switching from multimedia to textual mode can be executed to try and improve response time.

The framework prescribes the use of the ACME architecture description language [9], which extends the usual component-connector representation with the concept of *families*, allowing designers to define different architectural variants and styles [10]. This allows for the specialization of the framework to specific application domains, defining style-specific architectural operators and repair strategies [11].

Fig. 1, adopted from [8], shows the elements that compose the *Rainbow* framework. Monitoring is done with a set of *Probes* deployed in the target system, which send observations to *Gauges* that interpret the probe measurements in terms of higher-level models. The *Model Manager* is responsible for tracking the changes in the models' states and keeping it consistent with the target system. Moreover, other components query the Model Manager for information about the current state of the model.

One of these components is the *Architecture Evaluator*, which detects changes in the status of the properties of the system's architecture and environment, validating such changes with respect to the constraints stated in the model. In case of a violation, it triggers the *Adaptation Manager* in order for it to select the most appropriate strategy, using Utility Theory (details in [8]) for the decision. Finally, the *Strategy Executor* coordinates the execution process, deciding the operators that should be applied through the *Effectors* at the *System Layer*.

For the final parts of the adaptation loop, *Rainbow* uses a language called *Stitch*, which captures routine human adaptation knowledge as explicit adaptation policies [12]. The language allows designers to specify what, when and how to adapt, thus automating the adaptation process. In Section III we will see some examples of Stitch applied to the exemplar chosen for our experiments, the news website *ZNN.com* [8].

### B. Zanshin

*Zanshin* is an RE-based framework for the design of adaptive systems that exploits concepts of Control Theory to design adaptive software systems [7]. The core idea of the approach is to make the elements of the feedback loops that provide adaptivity first class citizens in the requirements models. An overview of the approach is shown in Fig. 2.

The approach is divided in two main steps that augment "vanilla" Goal-Oriented Requirements Engineering (GORE) with RE techniques that concern specifically adaptation requirements. Inspired by Control Theory, *Zanshin* supports **System Identification** of the target system in order to identify: (a) important indicators and the respective values the system should strive to maintain; (b) parameters that could be tuned at runtime to change the system's behavior; and (c) how changing parameters affect the value of the indicators [13].

Indicators are constrained by *Awareness Requirements* (*AwReqs*), which are requirements that refer to the states assumed by other requirements — such as their success or failure — at runtime [14]. Thus, *AwReqs* represent situations to which stakeholders would like the system to adapt. That way, they constitute the requirements for the monitoring component of the feedback loop that implements the adaptive capabilities of the system. Parameters have two flavors: *Variation Points*, which are the OR-refinements already present in goal models; and *Control Variables*, which are abstractions over OR-refinements that are too complex or tedious to model (we will see examples in Section III).

**Strategy Specification** focuses on the adaptation part of the feedback loop. Its objective is to associate one or more *adaptation strategies* (e.g., "Retry/delegate a task", "Relax a requirement", etc.) to each *AwReq* in order to have them executed in case of an *AwReq* failure at runtime. These strategies should also be elicited from the stakeholders and are represented by *Evolution Requirements* (*EvoReqs*). *EvoReqs* prescribe how other requirements of the model should evolve in response to an *AwReq* failure, and are specified using a set of primitive operations, each of which is associated with application-specific actions to be implemented in the system [15]. One strategy in particular, the *Reconfiguration* strategy, uses the information elicited during System Identification to reconfigure the system, also allowing designers to specify different reconfiguration algorithms depending on the amount of information available [16].

A prototype framework that operationalizes a feedback loop based on the models produced by *Zanshin* is available at https://github.com/sefms-disi-unitn/Zanshin. The experiments described in this paper (cf. Section IV) were conducted using this framework and can be repeated by the interested reader. In the next section, we will derive a goal model to represent the requirements of the *ZNN.com* exemplar used in the experiments and apply *Zanshin* to it.

## III. The *ZNN.com* Exemplar

An exemplar, or a *model problem*, is a shared, well-defined problem adopted by researchers of a specific field for pre-

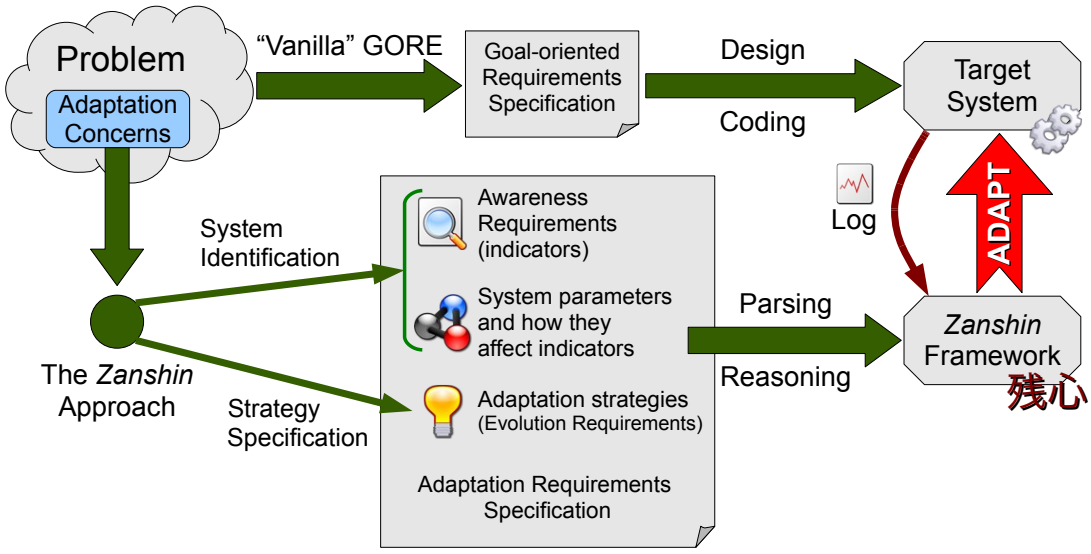


Fig. 2. An overview of the *Zanshin* approach.

senting and comparing proposals. The *Software Engineering for Adaptive and Self-Managing Systems* (SEAMS) research community has proposed some exemplars in their website,[1] among which we chose *ZNN.com* to perform the comparative study presented in this paper.

The choice of *ZNN.com* was also motivated by the fact that it had already been used in [8] as a case study for the proposal of the *Rainbow* framework. In this section, we present an overview of this model problem and how it was solved by *Rainbow*; then we apply *Zanshin* to it in order to be able to compare these two approaches.

### A. Overview of the Problem and Its Architectural Solution

*ZNN.com* is a news service that serves multimedia news content to its customers through a website. It is a simplified version of real sites such as cnn.com. It is presented in detail in [8] (§ 3.2) and also features as one of the exemplars in the SEAMS community website.

*ZNN.com*'s adaptive features are needed when the website experiences spikes in news requests due to, for instance, popular events. In these cases, response times for user requests might become unacceptable and the system has two possible adaptation strategies: enlisting new servers to divide the load of requests or switch from multimedia to text-mode to make each request quicker to respond. However, these strategies may cause problems in two other requirements of this system: first, the website managers would like to run the system at the lowest cost possible and adding new servers costs money; second, the users would like to see news with high content fidelity (i.e., high presentation quality), preferring multimedia over simple text.

Like most high demand websites, the architecture for *ZNN.com* includes a load balancer (or proxy) that distributes requests among multiple web servers and a database server. Current technology for load balancing and cloud computing already supports some level of adaptation, but automating trade-offs among multiple objectives like stated above is usually not supported [8]. For the *ZNN.com* case study, the operational target of the system is to keep a balance among its cost, performance and content fidelity.

The challenge of such systems is to achieve their mandate even when they operate under critical conditions. The difficulty lies in taking the right decision at the right time, in the sense that the problem should be detected promptly and the most efficient strategy to stabilize operation should be applied immediately. Under such circumstances, human intervention can be insufficient and automated mechanisms are required to carry out both decision making and adaptation.

*Rainbow* tackles this challenge through a software architecture model of *ZNN.com* written in the ACME language [9]. The model includes elements representing *clients*, *servers*, *connections* and the *proxy*; as well as properties for the client's *experienced response time*, the connection's *bandwidth* and the server's *cost*, *load* and *fidelity*. Moreover, operations for (de)activating a server and setting its fidelity allowed architectural designers to create four tactics that can be applied when adaptation is necessary: enlisting/discharging servers and raising/lowering the fidelity [17].

Tactics such as these are combined in strategies, written in Stitch to form high level adaptation processes. The exact definition of the adaptation strategies used in *ZNN.com* are described in Appendix C of Cheng's thesis [8]. We show one of these strategies in Fig. 3 and summarize them below:

- **`SimpleReduceResponseTime`**: In case a client experiences response time above a predefined threshold then the fidelity is lowered by one step. In case response time remains high, fidelity is decreased again one more step;

[1] See http://seams.self-adapt.org/wiki/Exemplars.

```
strategy SmarterReduceResponseTime
[ styleApplies && cViolation ] {
  define boolean unhappy = numUnhappyFloat/numClients >
M.TOLERABLE_PERCENT_UNHAPPY;

  t0: (unhappy) -> enlistServers(1) @[500 /*ms*/] {
    t1: (!cViolation) -> done;
    t2: (unhappy) -> enlistServers(1) @[2000 /*ms*/] {
      t2a: (!cViolation) -> done;
      t2b: (unhappy) -> lowerFidelity(2, 100) @[2000 /*ms*/] {
        t2b1: (!cViolation) -> done;
        t2b2: (unhappy) -> do[1] t2;
        t2b3: (default) -> TNULL;  // in this case, we have no more steps to take
      }
    }
  }
}
```

Fig. 3. Strategy **`SmarterReduceResponseTime`** in Stitch [8].

- **`SmarterReduceResponseTime`**: If an unacceptable percentage of clients experiences high response time, then enlist one server, then enlist another server and finally lower fidelity by one step. Repeat twice the last two actions until response time is restored;
- **`ReduceOverallCost`**: If server cost is higher than a threshold value then reduce the number of servers by one. If response time is low and cost remains high repeat the previous action, until cost is returned to normal;
- **`ImproveOverallFidelity`**: If content fidelity level is below threshold then raise fidelity of all servers by one step. If response time is low and fidelity remains low then raise fidelity level one more step.

The strategies above compose the possible options of the *Adaptation Manager* we described earlier when it is required to restore the system's invariants such as cost, performance and content fidelity to their desired levels.

Given the availability of *Rainbow* models for the *ZNN.com* exemplar, to conduct the comparative study we needed to produce models of the system according to *Zanshin*. The results of this effort are reported next.

### *B. An RE-Based Solution to ZNN.com using Zanshin*

Using available documentation, we have elicited requirements for the *ZNN.com* exemplar, producing the model shown in Fig. 4. Of course, the figure does not represent complete requirements for a news service (which would include concerns such as adding news, searching, managing advertisement, etc.), but concentrates on the adaptation scenario described earlier.

Requirements for the system are represented using Goal-Oriented Requirements Engineering (GORE) elements such as *goals*, *tasks*, *softgoals*, *quality constraints* and *refinement* relations that indicate how (soft)goals are satisfied using Boolean semantics [18]. One of *ZNN.com*'s goals is to *Serve news* to its visitors, which can be accomplished using *text-only*, *low resolution* or *high resolution* contents. Three non-functional requirements also compose this simple scenario:

- *Cost-efficiency*: the system should either be operating using a single server, unless response times are above a certain minimum threshold ($MIN_{RT}$), which would justify the addition of extra servers;
- *High fidelity*: analogously, the system should prefer high resolution content over lower ones, unless response times are above the minimum threshold;
- *High performance*: response time and server load should be under a certain maximum threshold ($MAX_{RT}$).

Quantitative values for $MIN_{RT}$ and $MAX_{RT}$ should eventually be provided by the stakeholders, but are not essential to the discussion herein.

On top of this base model, we have applied *Zanshin* to elicit requirements for adaptation as well. In the model of Fig. 4, these are represented by *AwReqs AR1*, *AR2* and *AR3*, variation point *VP1* and control variable *NoS* (*Number of Servers*). Moreover, by applying System Identification to *ZNN.com* we have come up with the following qualitative relations among indicators (*AwReqs*) and parameters:

$$\Delta\left(AR1/NoS\right)\left[0, maxServers\right] < 0 \quad (1)$$

$$\Delta\left(AR3/NoS\right)\left[0, maxServers\right] > 0 \quad (2)$$

The equations tell us that increasing the number of servers will hurt cost-efficiency (1), but contribute towards higher performance (2). Relations between variation point *VP1* and *AwReqs AR2* and *AR3* were also identified, but are not necessary to produce the *ZNN.com* scenarios presented in the previous subsection (we come back to those in Section IV-C). Finally, Fig. 5 shows the complete specification for *AwReqs AR1–AR3*, based on *Rainbow*'s **`SimpleReduceResponseTime`** strategy presented earlier.

Assuming initial values $NoS = 4$ and $VP1 = high\ resolution$, *AR1* will never actually fail (the simple scenario does not include enlisting of servers) and *AR2* (checked for every user request) will not fail initially. When *ZNN.com* experiences spikes in news requests it may cause *AR3* (related to *High performance* and also checked at every request) to fail, starting an adaptation session for it.

The first adaptation strategy (AS3.1), applicable only for the first failure of the session, is to change parameter *VP1*, which will take the value $low$. For the next $1000ms$, other requests with response time over threshold will activate strategy 3.2 (*Do Nothing*), simulating the waiting period of *Rainbow*'s strategy.

If *AR3* keeps failing for more than a second, AS3.2 will cease to be applicable, giving turn to AS3.3, which switches the fidelity to text-only mode and becomes immediately inapplicable. Further failures in the next 3 seconds will activate AS3.4 to simulate another waiting period. If the problem is not solved during this entire time, the *Abort* strategy (by default, the last resort in all *AwReq* failures) will take place and close the session.

The problem is considered solved when its resolution condition becomes true. This means that *AR3* was evaluated as being satisfied for a request that happened after one of the useful strategies (i.e., the ones that are not *Do Nothing*) was applied. Until this is true and *AR3*'s session is closed, failures of *AR2* will not lead to its strategy being executed, given its applicability condition. Once *AR3* is done and response time

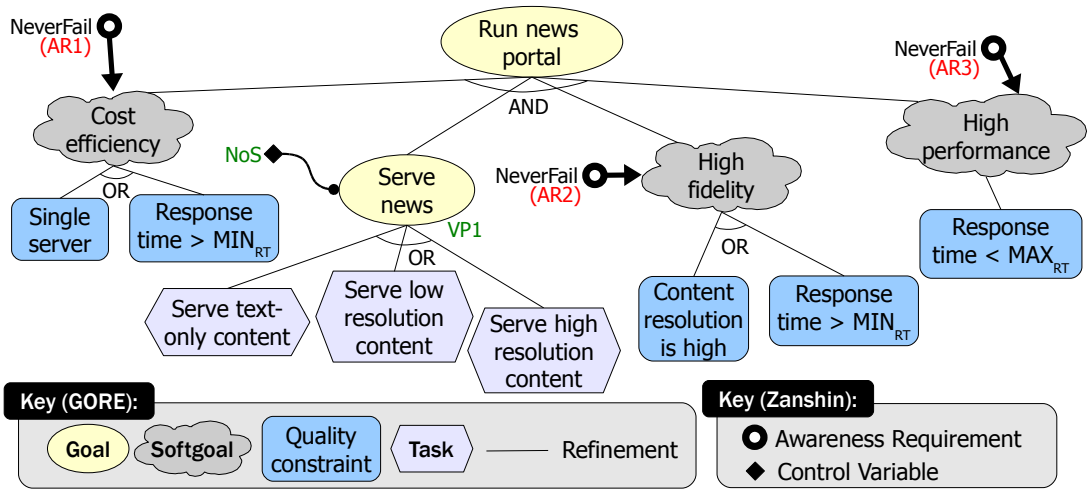


Fig. 4. Goal model for the *ZNN.com* exemplar, mirroring the adaptation scenarios modeled in *Rainbow*.

```
AwReq AR1: softgoal Cost efficiency should never fail
    ├ Checked at: every second
    ├ Adaptation Strategy 1.1: Reconfigure(∅)
    │   ├ Applicability Condition: there are no active sessions for AR3

AwReq AR2: softgoal High fidelity should never fail
    ├ Checked at: every request
    ├ Adaptation Strategy 2.1: ChangeParam(VP1, high)
    │   ├ Applicability Condition: there are no active sessions for AR3

AwReq AR3: softgoal High Performance should never fail
    ├ Checked at: every request
    ├ Adaptation Strategy 3.1: ChangeParam(VP1, low)
    │   ├ Applicability Condition: this is the first failure
    ├ Adaptation Strategy 3.2: Do Nothing
    │   ├ Applicability Condition: AS3.1 applied last, less than 1s ago
    ├ Adaptation Strategy 3.3: ChangeParam(VP1, text-only)
    │   ├ Applicability Condition: AS3.1 applied last, more than 1s ago
    ├ Adaptation Strategy 3.4: Do Nothing
    │   ├ Applicability Condition: AS3.3 applied last, less than 3s ago
    ├ Resolution Condition: AR3 was satisfied AND:
    │   ├ AS3.1 was applied last, more than 1s ago OR
    │   ├ AS3.3 was applied last, more than 3s ago
```

Fig. 5. Specification of the **`SimpleReduceResponseTime`** strategy with *Zanshin*.

```
AwReq AR3: HP should not fail more than MAX_unhappy% of the clients
    ├ Checked at: every request
    ├ Adaptation Strategy 3.1: Do Nothing
    │   ├ Applicability Condition:
    │   │   ├ Only one useful AS applied, less than 0.5s ago OR
    │   │   ├ At least two useful ASs applied, last was less than 2s ago
    ├ Adaptation Strategy 3.2: Reconfigure(∅)
    │   ├ Applicability Condition:
    │   │   ├ This is the first or second failure OR
    │   │   ├ Last useful AS applied was AS3.3 OR
    │   │   ├ VP1 == text-only
    ├ Adaptation Strategy 3.3: ChangeParam(VP1, low)
    │   ├ Applicability Condition: VP1 == high
    ├ Adaptation Strategy 3.4: ChangeParam(VP1, text-only)
    │   ├ Applicability Condition: VP1 == low
    ├ Resolution Condition:
    │   ├ AR3 was satisfied AND
    │   ├ AS3.1 is not applicable (i.e., we are not waiting)
```

Fig. 6. Specification of *AR3* for the **`SmarterReduceResponseTime`** strategy.

goes under $MIN_{RT}$, the *AR2*'s strategy will be applicable and, as a result of its execution, *ZNN.com* will go back to serving multimedia content.

As the description above shows, the models produced by applying *Zanshin* can produce, at runtime, the same result as the *Rainbow*'s **`SimpleReduceResponseTime`** strategy. It is also possible to model the **`SmarterReduceResponseTime`** strategy, by changing the specification of *AR3*, as follows.

First, its definition would change from "*High performance* should never fail" to "*High performance* should not fail for more than the $MAX_{unhappy}$% of the clients", where $MAX_{unhappy}$ represents the percentage of clients who experience response times that are higher than the tolerated threshold before something has to be done. Second, its adaptation strategies and resolution condition should also change, as shown in Fig. 6. The new specification of *AR3* represents, in a declarative way, the same algorithm described for **`SmarterReduceResponseTime`** in the previous subsection: reconfiguration (enlisting of additional servers) is applied at first with half a second of wait, then multiple times more (as long as the number of servers does not overcome $maxServers$) interposed with gradual reductions of fidelity.

The above exercise of mapping *ZNN.com*'s *Rainbow* specification to *Zanshin* indicates that the latter, although using a different representation, has at least equivalent expressiveness to the former. We will come back to this in the discussion of Section IV-C.

The expressiveness of *Zanshin* is due to its extensibility, allowing for new *AwReq*/*EvoReq* patterns and applicability/resolution conditions to be created. In effect, most of the conditions used in the examples of Fig. 5–6 did not yet exist when we started this experiment.

## IV. Comparison Between Rainbow and Zanshin

In the previous section we represented the adaptation strategies that *Rainbow* implements using Stitch with the *EvoReqs* of *Zanshin*. This allowed us to repeat the same experiment that simulates a scenario of highly increasing traffic that was already implemented for *Rainbow* [17], but this time assigning the adaptation control to *Zanshin*.

It is important to point out that the purpose of this work is not to compare the performance of the two frameworks or provide a better solution for the case study, but to compare and contrast the two approaches. To this end we mirror the solution of the *ZNN.com* case study using the *Zanshin* framework in order to make an apples-to-apples comparison. In what follows, we describe the methodology of our experiment, present its result and discuss the two frameworks based on our experience.

### A. Methodology

For purposes of this work, we implemented in *Zanshin* the goal model shown in Fig. 4, along with the specification of the **`SimpleReduceResponseTime`** strategy described in Fig. 5. For the base system (the *ZNN.com* website) we used the source code available on the SEAMS community website.

The deployment configuration is similar to the one described in [17] for the evaluation of the *Rainbow* framework and includes five Apache web servers (four replicated hosts and one proxy) and a MySql database server running on a Debian-flavored operating system. A Java™ application called JMeter, which is used to perform stress tests on web applications, is instantiated in one additional machine that plays the role of the clients who send requests to the server. The workload we created for the experiment (equivalent to that of [17]) simulates a real world case that many websites like *ZNN.com* deal with on a regular basis. The traffic scenario is as follows:

1) Slow start with 6 visits/min;
2) Sudden increase for five minutes where the traffic increases by 120 visits/min every minute until it reaches 600 visits/min;
3) Hold the load for 18 minutes;
4) For the remaining 36 minutes reduce the workload by 15 visits/min every minute.

After running the experiment and evaluating the effectiveness of *Zanshin*, we compare the characteristics of the two approaches by indicating weaknesses and advantages. The comparison points we set include a) adaptation type b) the kind of models used by each approach, c) the adaptation actions, d) the adaptation triggering, e) the adaptation selection and f) how each framework deals with adaptation failures. The outcome of this comparison can be exploited by the ongoing research on adaptive systems, leading to adaptation frameworks that would combine the maximum set of advantages of the current approaches.

### B. Experimental Results

We conducted two trials, one without any adaptation process and another applying *Zanshin*'s adaptation strategies. The results we extracted from JMeter's output are presented in figures 7 and 8, produced by the online service Loadosophia[2]. The *Baseline* trial represents the one without the adaptation process, whereas the one referred as *Test* represents the trial where *Zanshin* controls *ZNN.com*.

Fig. 7 shows that the response time has been improved by 67.4% after applying the adaptation strategies and the throughput has been improved by 8.7%. While in Fig. 8 the distribution of the experienced response times is depicted. From the latter we notice that in the case where *Zanshin* is present the distribution of the low response times is higher than in the case where an adaptation mechanism is absent.

To evaluate the efficiency of our approach we measured the failures of the *AwReqs* for every trial. The results have shown that without the use of an adaptation framework the AR3 failed 518 times, while with the use of *Zanshin* it failed 408 times but also AR2 failed 214 times. The improvement in the performance is obvious but it came with the cost of not providing high fidelity content for the whole duration of the experiment. The analysts of the system could use these metrics to evaluate their strategies and apply the suitable thresholds.

### C. Discussion

In this subsection we juxtapose the core ideas of the architecture-based approach followed by *Rainbow* and the RE-based approach followed by *Zanshin*.

We start by noticing that both approaches base their adaptation process on a closed loop, where the system monitors its output, detects possible malfunctions and changes its parameters in order to keep fulfilling its mandate. The necessity of the closed loop in software engineering has been pointed out by [4] as a tool that will give the opportunity to produce systems based on the principles of *Control Theory*. Another common point of the two frameworks, is that the control is external, which means that the target system does not implement any part of the control loop. In [19] it is mentioned that by delegating the control of the system to an external mechanism, higher generality, cost-effectiveness and composability can be achieved.

The main difference of the two frameworks lies on the different kinds of models they utilise to support their adaptation mechanism. The architectural model of *Rainbow* gives information about the capabilities and the restrictions of the system, which later on will be exploited as operators and adaptation conditions accordingly. Furthermore, basing the adaptation strategies on an architectural model that describes a family of systems makes them reusable to any target system that conforms to the same architecture.

However, having as a starting point the architectural model of the system can result in capturing only low level requirements about it. On the other hand, a requirements model as the one *Zanshin* uses can capture every requirement that comes from the stakeholders. Nevertheless, technical restrictions and

[2]https://loadosophia.org

| | Baseline Value | Test Value | Difference | Hint |
|---|---|---|---|---|
| Test Duration : | 1 hour, 3 minutes, 59 seconds | 1 hour, 4 minutes, 14 seconds | 15 seconds | Duration difference 0.4% is small enough, comparison is valid |
| Average Virtual Users : | 387.91 | 405.552 | 17.642 | +8.7%, Test served more requests per second |
| Average Rate (TPS) : | 7.10732 | 7.7234 | 0.6161 | |
| Average Response Time (ms) : | 7544 | 2461 | -5083 | -67.4%, Test was faster |

Fig. 7. Summary comparison report

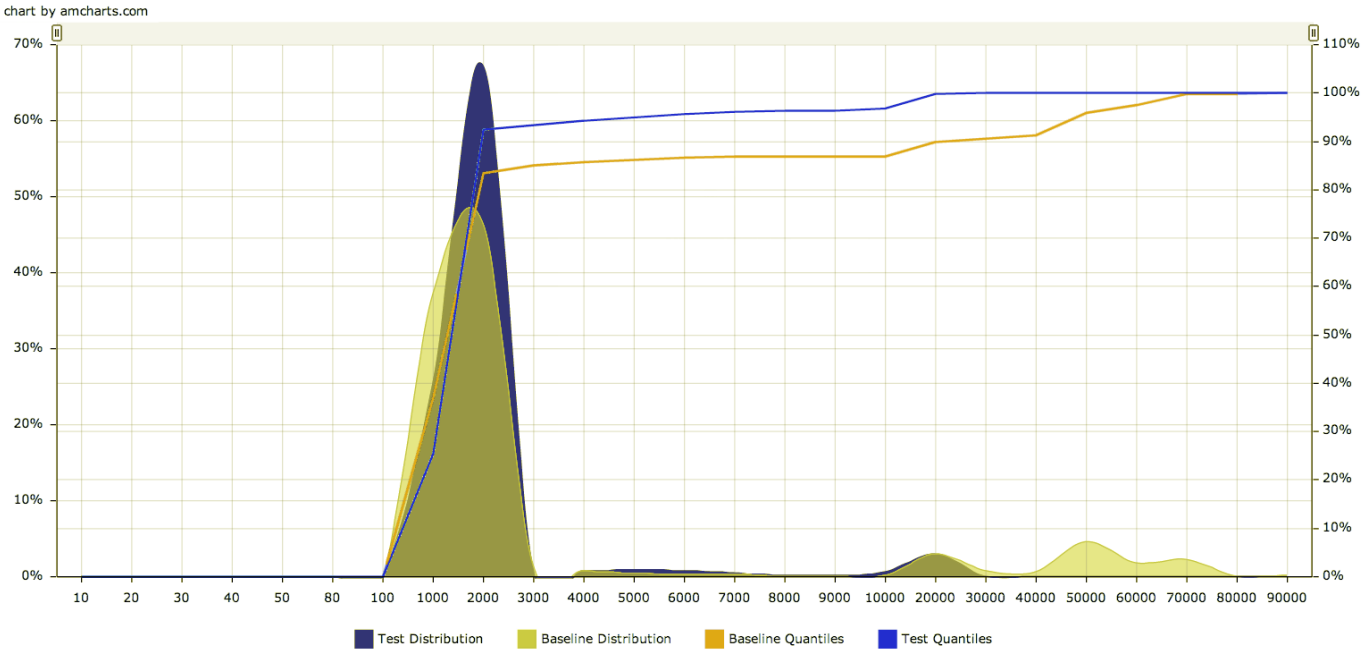


Fig. 8. Distribution of response times

properties can be revealed only at a later stage of the requirements analysis process and sometimes important details are overlooked unintentionally.

We saw earlier that the possible adaptation actions of *Rainbow* are defined by the set of basic operators provided by the target system, e.g., activate server and change fidelity. These operators can be combined in tactics, which are then combined in strategies expressed in Stitch language. These strategies are intended to encapsulate human expertise on specific situations, where external intervention is required to restore a malfunctioning system. On the other hand, *Zanshin* provides two kinds of adaptation: *reconfiguration* and *evolution*. A reconfiguration can either change a parameter of the system (control variable) or switch to an alternative selected for a variant point (OR-refinement on the goal model). The new configuration is informed to the system, which can then take further actions related to this change. It is also important to point out that the ability of self-inspecting in *Zanshin* provides a lot of expression power for its adaptation strategies. However, as it is usually the case in any modeling language, this should be used with care in order not to make models that are very difficult to manage. These modifications are based on the differential relations mined during the system identification process and let the system compose its own adaptation strategies given the holding conditions.

*EvoReqs*, however, are modeled as Event-Condition-Action (ECA) rules, where the actions are composed of sixteen basic operations [15]. From these, thirteen are system-specific thus must be implemented in the target system. These operations allow, among other things, to retry a given goal, to change the parameters of the system, to delegate the issue to an actor and to relax the awareness requirements (meta-adaptation). In the previous section we managed to express the *Rainbow* strategies with *EvoReqs* and Reconfiguration using the applicability and the resolution conditions. Defining a formal transformation from one approach to the other, though, is not an easy task. The adaptation strategies composed by *EvoReqs* are more close to those of the Rainbow framework written in Stitch as they both

capture static administrative operations while the latter gives a more clear representation of the priorities of the objectives using Utility Theory.

Regarding the monitoring part (malfunction detection), in *Rainbow* an adaptation is triggered when any invariant in the ACME model fails. An example of invariant is $response\ time < MAX_{RT}$. Thus, if the *response time* gets equal or higher than the maximum allowed, an adaptation is triggered. Instead of invariants, *Zanshin* utilizes *AwReqs* in the requirements model to reason about the status of the target system's operation.

For instance, considering a quality constraint of $response\ time < MAX_{RT}$, an *AwReq* may state that this should be the case in at least 90% of the time. If the percentage goes below that threshold, an adaptation is triggered. We can say that both frameworks are based on the models that they use as a centerpiece for their adaptation, in order to define the variables of the system which should be monitored. However, the variety of *AwReqs* offer a higher level of expressiveness to represent objectives to be satisfied at runtime, than the simple conditions of the architectural model.

Another comparison point is adaptation triggering. In *Rainbow*, it is guided by pre-conditions for the execution of adaptation strategies. If more than one is applicable, the best one is selected according to an aggregate attribute vector that considers a) the cost-benefit of the tactics in a strategy, b) the weights of predefined criteria, and c) the likelihood of each tactic being applicable. In *Zanshin*, *EvoReqs* have a similar format: there are pre-conditions that define whether a strategy applies or not. If more than one is applicable, the first one is selected (according to the order on which the *EvoReqs* were defined).

However, when *Zanshin* uses reconfiguration, the new values for the system's parameters are defined based on a control-theoretic approach. Differential relations are used to define the impact of parameter changes on the *AwReqs* (benefit). Different adaptation algorithms can be used to select which reconfiguration to perform. In Section III-B, given that we were mirroring the scenario implemented in *Rainbow*, only one parameter (*NoS*) was used in the process and reconfiguration was trivial. We could have, however, included differential relations about *VP1* as well:

$$\Delta\left(AR2/VP1\right) > 0 \tag{3}$$

$$\Delta\left(AR3/VP1\right) < 0 \tag{4}$$

These equations represent the fact that an increase in the fidelity level would contribute positively to the success of *AR2* (3) but at the same time decrease the success rate of the *AR3* (4). Considering these equations together with the ones presented earlier, we can prioritize the relations that involve the same indicator to declare which parameters have greater impact on it. For example, $\Delta\left(AR3/NoS\right)[0, maxServers] > \Delta\left(AR3/VP1\right)$ would mean that by increasing the number of servers the probability to have high performance (*AR3*) is increasing faster than by decreasing the fidelity. This way, *Zanshin* can provide dynamic adaptation based on control theory principles, while the adaptation process in *Rainbow* is in this sense static.

Finally, we contrast how the two frameworks deal with adaptation failures. In *Rainbow* a tactic fails when a) its pre-conditions are not satisfied, b) the execution of its operators fail, or c) the result of the tactic is different than expected (which is assessed through post-conditions). These failures are predicted in the strategies, which can request alternative tactics to be executed when a given tactic fails. If all the possible tactics were applied, but the goal of the strategy is not achieved, a termination condition is triggered and the strategy ends. Then *Rainbow* will recalculate which is the most suitable strategy to apply. Similarly, in *Zanshin* when none of the applicability conditions of the *EvoReqs* for a given adaptation is satisfied, the adaptation is aborted. After an adaptation action is performed, but the *AwReq* that triggered the adaptation still fails, the adaptation selection is performed again.

## V. Related Work

As we have previously mentioned, the purpose of this paper is to compare RE-based and architecture-based approaches to the design of adaptive systems. To this end, we have chosen one representative of each type, respectively the *Zanshin* (cf. §II-B) and the *Rainbow* (cf. §II-A) frameworks.

However, there are several approaches that fit these two categories, many of which are cited in surveys and roadmap papers in the area of adaptive systems [1], [2], [3], [4], [5]. In what follows, we summarize some of these approaches.

A well-known **RE-based** approach is RELAX [20], which aims at capturing uncertainty declaratively with modal, temporal and ordinal operators applied over SHALL statements (e.g., "the system SHALL ... AS CLOSE AS POSSIBLE to ..."). A similar approach, but based on the goal-oriented language KAOS [21], is FLAGS [22]. This approach extends the linear temporal logic (LTL) used in KAOS with fuzzy relational and temporal operators, allowing some goals to be satisfied even if values are "around" but not exactly equal to the desired ones. FLAGS also proposes an operationalisation of its models in a service-oriented infrastructure. It could also be a candidate for a more extended comparison since the approach involves mechanisms for capturing the problem and the adaptive solutions similar to *Zanshin*. The operationalisation though of the approach would require to tailor the case study accordingly.

The LoREM approach [23], also based on KAOS, uses an extension of LTL that includes an *Adapt* operator and defines a systematic process for performing goal-oriented RE for adaptive systems. Later, Cheng et al. [24] integrate this approach with the RELAX language in order to explore environmental uncertainty using threat modeling.

There are also a few RE-based approaches for the design of adaptive systems based on $i^\star$ [25] and Tropos [26]. Tropos4AS [27] is a methodology for the design of agent-based

adaptive systems founded on the Belief-Desire-Intention (BDI) model. As run-time infrastructure, Tropos4AS proposes the mapping of goal models to Jadex.[3] The CARE method [28] also bases itself on Tropos, but focuses on service-based applications. Adaptive requirements are specified at design time and a run-time infrastructure based on environment monitoring, service selection and customization is provided. Dalpiaz et al. [29] propose an architecture that adds self-reconfiguring capabilities to a system using a Monitor-Diagnose-Compensate (MDC) loop based on the system's requirements models in $i^\star$. Different reconfiguration algorithms are proposed on top of this architecture.

On the **architecture-based** side, one of the first proposals for an architecture that is well-suited for systems that can adapt themselves was IBM's autonomic computing initiative [30]. According to it, systems are made of interactive collections of autonomic elements delivering services to users and to other elements according to specified goals and constraints. Each element follows an internal MAPE-K loop which performs monitoring, analysis, planning and execution of actions, based on knowledge about the environment, policies, etc. Another seminal work is that of Oreizy et al. [31], which proposed an infrastructure that relies on software agents, explicit representation of software components and the environment, plus messaging and event services that coordinate the adaptation.

A well-known architecture-based approach is that of Kramer & Magee [32], which proposes a reference architecture for self-adaptive systems based on a three-layer architecture — from bottom to top: *Component Control* (reports events and status to the upper layer and supports modification of current component configuration); *Change Management* (performs changes in the bottom layer based on situations reported by the latter or new goals introduced by it, relying on the top layer in case of unplanned situations); and *Goal Management* (produces change management plans in response to requests from the middle layer or the introduction of new goals). Several subsequent proposals followed this architectural foundation.

Sousa et al. [33] focus on allowing users to control Quality of Service (QoS) trade-offs and coordinate the use of resources in a distributed environment composed of several applications. Utility functions for each QoS dimension express user preferences in terms of thresholds for satiation and starvation. The SASSY framework [34] also focuses on QoS tuning, targeting service oriented systems. The approach uses a BPMN[4]-based language to represent the correct behavior of the system, allowing domain experts to annotate such model with QoS goals.

## VI. Conclusions

In this paper we conducted a comparative study between two adaptation approaches, one architecture-based and one RE-based. As a reference point we used the *ZNN.com* exemplar, applying both frameworks to provide adaptation mechanisms according to its described scenarios. Results have shown that both frameworks can provide significant improvement to the system's operation, without any human intervention.

We also performed a side by side comparison of the core elements of both approaches. The outcome of this comparison is that architecture models can capture all the properties and technical restrictions of the target system and by using them as a guide to develop adaptation strategies the reusability of the adaptation mechanism becomes applicable.

More specifically, *Rainbow* captures the human experience and expertise in its strategies and, by applying techniques from decision theory, selects the one that is most suitable. Therefore, the control level of *Rainbow* does not exceed the one of the human intervention but automates it, offering better reaction time and eliminating human errors.

On the other hand, a requirements model captures more explicitly the objectives of the systems and, with the use of quality constraints, can also express the technical restrictions. However, the exact values of the thresholds of these constraints can be provided either by the instantiation of the architecture model or by the expertise of the system analyst. Moreover, some practitioners may consider that detailed architectural information do not belong in requirements models.

Regarding the adaptation process, the RE-based approach presented in this paper provides higher variability by applying *EvoReqs* or letting the system adjust its parameters through a reconfiguration strategy. In this way, the system relies its adaptation process not only on human expertise but also on well-founded principles of control theory. The current state of *Zanshin* applies these principles using qualitative control, although deriving quantitative factors that would give precise information about the impact of the parameter changes on the system's output is part of our research agenda.

In summary, this study revealed the advantages and the vulnerabilities of two well-known approaches in the field of software adaptation. The results suggest that requirement and architectural models should be combined in order to capture every detail of the target system's adaptation needs. The purpose of this combination is to mine all the alternatives that are embedded in the solution and the problem space. The requirements models can provide a broader set of alternatives (e.g., in the case of *ZNN.com*, delegate the video hosting to an external service, such as YouTube), while the architectural models can provide variability in the deployment of the solution. Moreover, *AwReqs* and parameters can indicate the specific components of the system or variables of the environment that should be monitored, instead of putting probes empirically.

For the reasons illustrated above, our current research focuses on the derivation of architecture models from requirements models that include control parameters and indicators. We are using the STREAM-A approach [35] as baseline for this new line of work.

[3] A BDI Agent System, see http://jadex-agents.informatik.uni-hamburg.de/.
[4] Business Process Model and Notation, see http://www.bpmn.org/.

## ACKNOWLEDGMENT

We are grateful to Bradley Schmerl for providing us with valuable information and material about the Rainbow implementation to facilitate our comparative study.

This work has been supported by the ERC advanced grant 267856 "Lucretius: Foundations for Software Evolution" (April 2011 – March 2016, http://www.lucretius.eu) as well as Brazilian foundations FAPES (http://www.fapes.es.gov.br) through the PRONEX grant #52272362 and CAPES (http://www.capes.gov.br).

## REFERENCES

[1] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl, "A journey to highly dynamic, self-adaptive service-based applications," *Automated Software Engineering*, vol. 15, no. 3, pp. 313–341, 2008.

[2] M. C. Huebscher and J. A. McCann, "A survey of Autonomic Computing—Degrees, Models, and Applications," *ACM Computing Surveys*, vol. 40, no. 3, pp. 1–28, 2008.

[3] M. Salehie and L. Tahvildari, "Self-Adaptive Software: Landscape and Research Challenges," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, no. 2, pp. 1–42, 2009.

[4] B. H. C. Cheng *et al.*, "Software Engineering for Self-Adaptive Systems: A Research Roadmap," in *Software Engineering for Self-Adaptive Systems*, ser. Lecture Notes in Computer Science, B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer, 2009, vol. 5525, pp. 1–26.

[5] R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, Eds., *Software Engineering for Self-Adaptive Systems II*, ser. Lecture Notes in Computer Science. Springer, 2013, vol. 7475.

[6] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[7] V. E. S. Souza, "Requirements-based Software System Adaptation," PhD Thesis, University of Trento, Italy, 2012.

[8] S.-W. Cheng, "Rainbow: Cost-Effective Software Architecture-based Self-adaptation," Ph.D. dissertation, Carnegie Mellon University, 2008.

[9] D. Garlan, R. Monroe, and D. Wile, "Acme: an architecture description interchange language," in *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, ser. CASCON '97. IBM Press, 1997, pp. 7–. [Online]. Available: http://dl.acm.org/citation.cfm?id=782010.782017

[10] B. Schmerl and D. Garlan, "Exploiting Architectural Design Knowledge to Support Self-Repairing Systems," in *Proc. of the 14$^{\text{th}}$ International Conference on Software Engineering and Knowledge Engineering*. ACM, 2002, pp. 241–248.

[11] D. Garlan, S.-W. Cheng, and B. Schmerl, "Increasing System Dependability through Architecture-Based Self-Repair," in *Architecting Dependable Systems*, ser. Lecture Notes in Computer Science, R. de Lemos, C. Gacek, and A. Romanovsky, Eds. Springer, 2003, vol. 2677, pp. 61–89.

[12] S.-W. Cheng and D. Garlan, "Stitch: A language for architecture-based self-adaptation," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2860–2875, 2012.

[13] V. E. S. Souza, A. Lapouchnian, and J. Mylopoulos, "System Identification for Adaptive Software Systems: a Requirements Engineering Perspective," in *Conceptual Modeling – ER 2011*, ser. Lecture Notes in Computer Science, M. Jeusfeld, L. Delcambre, and T.-W. Ling, Eds. Springer, 2011, vol. 6998, pp. 346–361.

[14] V. E. S. Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos, "Awareness Requirements for Adaptive Systems," in *Proc. of the 6$^{\text{th}}$ International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2011, pp. 60–69.

[15] V. Souza, A. Lapouchnian, K. Angelopoulos, and J. Mylopoulos, "Requirements-driven software evolution (online first)," *Computer Science - Research and Development*, pp. 1–19, 2012.

[16] V. E. S. Souza, A. Lapouchnian, and J. Mylopoulos, "Requirements-driven Qualitative Adaptation," in *Proc. of the 20th International Conference on Cooperative Information Systems (to appear)*. Springer, 2012.

[17] S.-W. Cheng, D. Garlan, and B. Schmerl, "Evaluating the Effectiveness of the Rainbow Self-Adaptive System," in *Proc. of the ICSE 2009 Workshop on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2009, pp. 132–141.

[18] I. Jureta, J. Mylopoulos, and S. Faulkner, "Revisiting the Core Ontology and Problem in Requirements Engineering," in *Proc. of the 16$^{\text{th}}$ IEEE International Requirements Engineering Conference*. IEEE, 2008, pp. 71–80.

[19] S.-W. Cheng, D. Garlan, and B. Schmerl, "Self-star properties in complex information systems," O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, and S. Leonardi, Eds. Berlin, Heidelberg: Springer-Verlag, 2005, ch. Making self-adaptation an engineering reality, pp. 158–173. [Online]. Available: http://dl.acm.org/citation.cfm?id=2167575.2167589

[20] J. Whittle, P. Sawyer, N. Bencomo, B. Cheng, and J.-M. Bruel, "RELAX: a language to address uncertainty in self-adaptive systems requirement," *Requirements Engineering*, vol. 15, no. 2, pp. 177–196, 2010.

[21] A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-directed Requirements Acquisition," *Science of Computer Programming*, vol. 20, no. 1-2, pp. 3–50, 1993.

[22] L. Baresi, L. Pasquale, and P. Spoletini, "Fuzzy Goals for Requirements-driven Adaptation," in *Proc. of the 18$^{\text{th}}$ IEEE International Requirements Engineering Conference*. IEEE, 2010, pp. 125–134.

[23] H. J. Goldsby, P. Sawyer, N. Bencomo, B. H. C. Cheng, and D. Hughes, "Goal-Based Modeling of Dynamically Adaptive System Requirements," in *Proc. of the 15$^{\text{th}}$ Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*. IEEE, 2008, pp. 36–45.

[24] B. H. C. Cheng, P. Sawyer, N. Bencomo, and J. Whittle, "A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, A. Schürr and B. Selic, Eds. Springer, 2009, vol. 5795, pp. 468–483.

[25] E. S. K. Yu, P. Giorgini, N. Maiden, and J. Mylopoulos, *Social Modeling for Requirements Engineering*, 1st ed. MIT Press, 2011.

[26] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos, "Tropos: An Agent-Oriented Software Development Methodology," *Autonomous Agents and Multi-Agent Systems*, vol. 8, no. 3, pp. 203–236, 2004.

[27] M. Morandini, L. Penserini, and A. Perini, "Operational Semantics of Goal Models in Adaptive Agents," in *Proc. of the 8$^{\text{th}}$ International Conference on Autonomous Agents and Multiagent Systems*. ACM, 2009, pp. 129–136.

[28] N. A. Qureshi and A. Perini, "Requirements Engineering for Adaptive Service Based Applications," in *Proc. of the 18$^{\text{th}}$ IEEE International Requirements Engineering Conference*. IEEE, 2010, pp. 108–111.

[29] F. Dalpiaz, P. Giorgini, and J. Mylopoulos, "Adaptive socio-technical systems: a requirements-based approach," *Requirements Engineering*, pp. 1–24, 2012.

[30] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[31] P. Oreizy *et al.*, "An Architecture-Based Approach to Self-Adaptive Software," *IEEE Intelligent Systems*, vol. 14, no. 3, pp. 54–62, 1999.

[32] J. Kramer and J. Magee, "Self-Managed Systems: an Architectural Challenge," in *Future of Software Engineering (FOSE '07)*. IEEE, 2007, pp. 259–268.

[33] J. P. Sousa, R. K. Balan, V. Poladian, D. Garlan, and M. Satyanarayanan, "A Software Infrastructure for User–Guided Quality–of–Service Tradeoffs," in *Software and Data Technologies*, ser. Communications in Computer and Information Science, J. Cordeiro, B. Shishkov, A. Ranchordas, and M. Helfert, Eds. Springer, 2009, vol. 47, pp. 48–61.

[34] D. A. Menasce, H. Gomaa, S. Malek, and J. a. P. Sousa, "SASSY: A Framework for Self-Architecting Service-Oriented Systems," *IEEE Software*, vol. 28, no. 6, pp. 78–85, 2011.

[35] J. Pimentel, M. Lucena, J. Castro, C. Silva, E. Santos, and F. Alencar, "Deriving software architectural models from requirements models for adaptive systems: the STREAM-A approach," *Requirements Engineering*, vol. 17, no. 4, pp. 259–281, 2012.