**Computer Science - Research and Development manuscript No.**
(will be inserted by the editor)

# Requirements-Driven Software Evolution

**Vítor E. Silva Souza · Alexei Lapouchnian · Konstantinos Angelopoulos · John Mylopoulos**

Received: date / Accepted: date

**Abstract** It is often the case that stakeholders want to strengthen/weaken or otherwise change their requirements for a system-to-be when certain conditions apply at runtime. For example, stakeholders may decide that if requirement $R$ is violated more than $N$ times in a week, it should be relaxed to a less demanding one $R$-. Such *evolution requirements* play an important role in the lifetime of a software system in that they define possible changes to requirements, along with the conditions under which these changes apply. In this paper we focus on this family of requirements, how to model them and how to operationalize them at runtime. In addition, we evaluate our proposal with a case study adopted from the literature.

**Keywords** Requirements engineering; modeling; evolution; requirements; adaptive systems

## 1 Introduction

Adaptation and evolution are related concepts. In Biology, individuals adapt to better fit their environment, while species evolve when enough of their individual members adapt to a particular new trait. In the field of Software Engineering, Meir Lehman (1979), proposed that software evolution and maintenance processes change a software system in accordance with laws, much like physical laws prescribing physical phenomena. Adaptive and autonomic systems, on the other hand, include in their architecture mechanisms through which they can change their behavior at runtime in order to better fulfill their requirements (Kephart and Chess, 2003, Cheng et al, 2009a, Brun et al, 2009, Andersson et al, 2009).

Support for adaptation and evolution is especially important in Requirements Engineering (hereafter RE) since change in software systems is frequently triggered by change in stakeholder requirements (Zowghi and Offen, 1997). With this motivation, our research has studied different scenarios for requirements evolution, striving to develop adaptation and evolution mechanisms that support these scenarios. For example, Ernst et al (2011a, 2012) explore the case where unanticipated changes occur to the requirements of an operational system, such as a new law coming into effect, or stakeholders wanting additional functionality.

In this paper, we are focusing on requirements that cause the evolution of other requirements. For instance, such as requirement *evR1* = "If requirement $R$ fails more than $N$ times in a row, replace it with $R-$", or even *evR2* = "After January $1^{st}$ 2014, replace $R$ with $R+$". Here, both requirements *evR1* and *evR2* consist of a condition-action rule where the action involves changing (strengthening, weakening, abandoning, ...) another requirement. We call such requirements *evolution requirements* (*EvoReqs* for short). The main objective of this paper is to circumscribe and characterize such requirements and offer a prototype implementation for a software evolution mechanism that operationalizes *EvoReqs*. *EvoReqs* allow us to not only specify what other requirements need to change, but also when other strategies should be used, such as "retry after some time" or "abort current execution".

Notice that this is a big change with respect to previous works, which consider as evolution only unanticipated changes that, therefore, are not able to be modeled, let alone developed, a priori, e.g., (Bennett and

V. E. S. Souza, A. Lapouchnian, K. Angelopoulos, J. Mylopoulos
Department of Information Engineering and Computer Science, University of Trento, Via Sommarive 14, 38123, Trento, Italy
E-mail: {vitorsouza,lapouchnian,angelopoulos,jm}@disi.unitn.it

Rajlich, 2000, Ernst et al, 2011a). However, by defining requirements evolution as any change in the system's original requirements, be it anticipated or not, it follows immediately that evolving requirements is one way of adapting to system failures at runtime. Of course, for these evolutions to be done automatically, all involved requirements (in the previous example, $R$, $R-$ and $R+$) must have already been implemented.

Our approach is goal-oriented in the sense that requirements are modeled as goals that can be refined and correlated to each other, while *EvoReqs* are modeled as Event-Condition-Action (ECA) rules that are activated when an event occurs and a guard condition holds. The action component of an ECA rule consists of a sequence of primitive operations on a goal model (that evolve the goal model in accordance with stakeholder wishes). Each operation results in a primitive change to a goal model, e.g., removes/adds a goal at the class or instance level, changes the state of a goal instance, or undoes the effects of all executed actions for an aborted execution. Moreover, such operations can be combined using patterns in order to compose macro-level evolution strategies, such as the aforementioned *Retry* and *Abort* cases.

Our proposal is illustrated using the classic example of the Meeting Scheduler, improving on what has been presented in (Souza et al, 2011a). However, we validate our proposal with a larger experiment, in which an Adaptive Computer-aided Ambulance Dispatch (A-CAD) is designed using our approach and is then executed through simulations to see how reasonable its evolution is. Its requirements were based on the well-known London Ambulance Service Computer-Aided Despatch (LAS-CAD) failure report (Finkelstein, 1993) and some of the publications that analyzed the case, e.g., (Kramer and Wolf, 1996). For the simulations, we have developed a framework that operationalizes *EvoReqs* at runtime, called *Zanshin*.

This is an extended version of the paper (Souza et al, 2012a), titled "(Requirement) Evolution Requirements for Adaptive Systems" that appears in the *Proceedings of the 7$^{th}$ International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (SEAMS'12).

The rest of the paper is organized as follows: Section 2 presents the baseline for our research, namely Goal-Oriented Requirements Engineering, and introduces the running example; Section 3 summarizes our earlier proposals for the design of adaptive systems based on feedback loops; Section 4 focuses on *EvoReqs*, showing how to specify them in the system requirements and how they can be used to compose adaptation strategies, including reconfiguration; Section 5 details the operationalization of *EvoReqs* at runtime through an ECA-based process that executes adaptation strategies in response to failures; Section 6 describes the experiments conducted with the A-CAD, starting from its design as an adaptive system using our proposal until the execution of simulations that demonstrate how its requirements evolve at runtime; Section 7 compares our approach to related work in the fields of software adaptation and evolution; Section 8 discusses the work presented in this paper, pointing out some of its limitations and opportunities for improvement; finally, Section 9 concludes.

## 2 Goal-Oriented Requirements Engineering

Goal-Oriented Requirements Engineering is founded on the premise that the requirements stakeholders want to fulfill through a new system are goals (desired states-of-affairs), not functions that determine the functionality of the new system. Such goals need to be modeled and analyzed and through a systematic refinement process can lead to a functional specification for the system-to-be (Dardenne et al, 1993).

In our work, we use as primitives for building goal models the concepts included in the requirements ontology proposed in (Jureta et al, 2008). Apart from *goal*, the ontology includes *tasks* (aka actions or functions) that operationalize goals. As well, our modeling framework includes *softgoals*, that are non-functional requirements such as security and usability. Softgoals are in turn operationalized by *quality constraints*, such as "There will be no more than 5 security breaches per year". Finally, our framework includes *domain assumptions* that have to hold for a specification to fulfill requirements. For instance, we may generate through refinement a specification that fulfills the goal *Schedule meeting* through the execution of two tasks: *Collect timetables* and *Find time slot*. But for this plan to work, we need a domain assumption *Enough meeting rooms available* without which our plan may or may not work.

For our running example, shown in Figure 1, we start with the system's main goal, *Schedule meeting*, we refine the model by decomposing the goal into sub-goals (e.g., *Use local room* is decomposed in two sub-goals, *Find a local room* and *Book room*) and by operationalizing them into tasks and domain assumptions (e.g., *Collect automatically* is operationalized by domain assumption *Participants use the system calendar* and task *Collect from system calendar*).

A task operationalization is a requirement on an agent (human or software, not represented in the model) whereas a domain assumption operationalization is a requirement on the environment (it is assumed to be

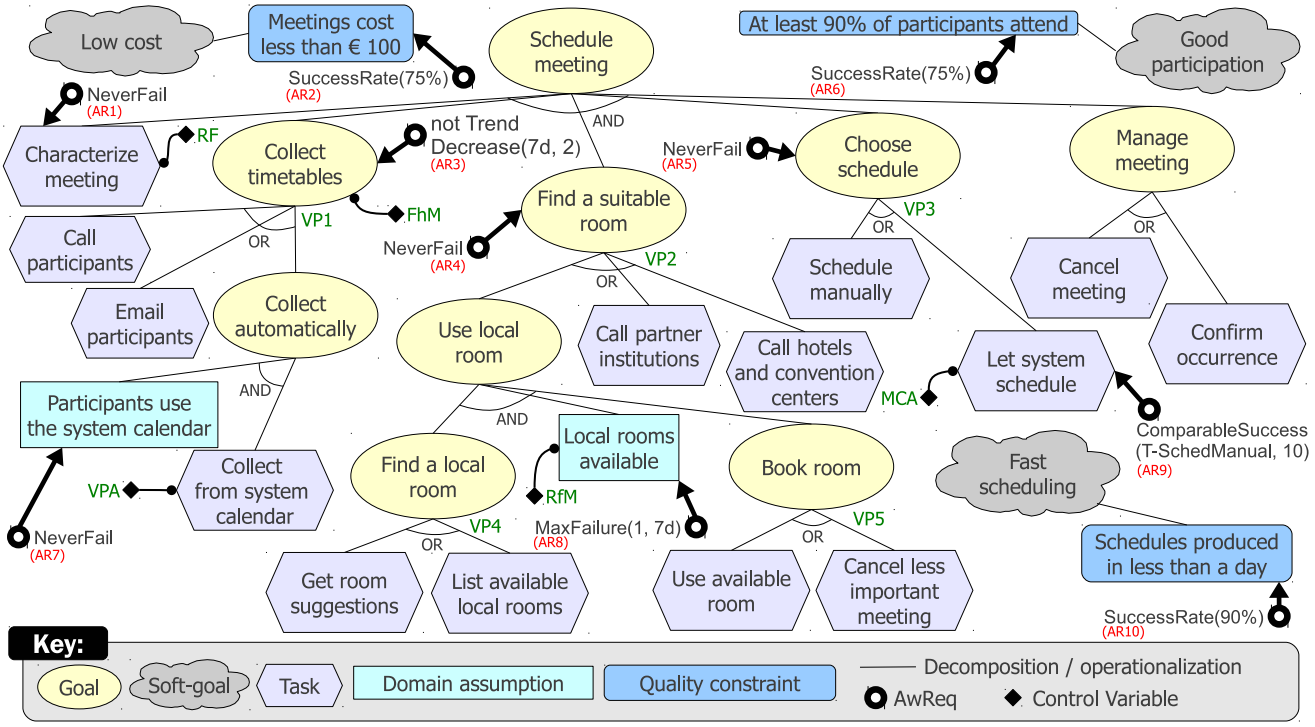


**Fig. 1** Goal model representing the requirements specification for a Meeting Scheduler.

true). Moreover, some non-functional requirements are also represented as softgoals, operationalized by quality constraints, giving them clear-cut criteria for satisfaction and avoiding a vague specification.

Refinements are of two types: *AND* or *OR*, carrying obvious Boolean semantics. OR-refinements allow us to expand the space of alternatives for goal satisfaction, adding variability to the specification. Variability is very important for adaptive systems, given that a possible means of adaptation is to explore the solution space for a suitable alternative (Souza et al, 2011a). Variability in requirements and goals has been explored in other works, e.g., (Lapouchnian and Mylopoulos, 2009).

Figure 1 also represents *Awareness Requirements* (*AwReqs*), *Variation Points* and *Control Variables*, which consist of new elements proposed in our past research on the design of adaptive systems based on a feedback loop architecture. This research is summarized next.

## 3 Feedback-Based Adaptation

Past research (Andersson et al, 2009, Brun et al, 2009, Cheng et al, 2009a) has pointed out the need for systematic software engineering approaches for developing self-adaptive systems based on ideas from control engineering with focus on explicitly specified feedback loops, which provide a generic mechanism for self-adaptation. Our research adopts a RE perspective, which motivated the question: what are the requirements that lead to feedback loop functionality? This question has led us to propose a new class of requirements, called *Awareness Requirements* (*AwReqs*), which talk about the runtime status of other requirements, such as their success or failure (Souza et al, 2011b).

Furthermore, given that feedback loops are a central element of control systems (Brun et al, 2009), we began to explore techniques from Control Theory that could be useful in the design of systems that use feedback loops as a mechanism for adaptation. We again applied our RE perspective and proposed that *System Identification* — the process of determining the rules that govern a system's dynamic behavior — should be conducted during the modeling of an adaptive system's requirements, using qualitative information to deal with uncertainty (Souza et al, 2011a).

The goal model shown earlier in Figure 1 is, in fact, the result of applying our systematic system identification process, which starts with *AwReqs* elicitation. *AwReqs* represent noteworthy situations where stakeholders may want the system to adapt. They also indicate how critical each requirement is by specifying the degree of failure that can be tolerated. In other words, *AwReqs* are used as indicators of requirements convergence at runtime (one could, however, adapt the process to use other kinds of indicators). Since *AwReqs* can

impose constraints on the success and failure of other requirements, they enforce our RE perspective at runtime by describing the desired behavior of the system in terms of its requirement models.

In our running example, ten *AwReqs* were identified (labeled *AR1–AR10*) and represented in the model using their proposed graphical syntax and patterns. For example, since *Characterize meeting* is very important (you cannot really do much without basic information on the meeting to schedule), *AR1* prescribes that it should never fail. A not-so-critical requirement is *Good participation*, so *AR6* enforces a 75% success rate on its quality constraint. *AwReqs* can also talk about the trend of a requirement's success rate, e.g., *AR3*: the success rate of *Collect timetables* should not decrease twice in a row, considering week periods.

At runtime, the elements of the goal model are represented as classes, being instantiated every time a user (or the system itself) starts pursuing a requirement (in the case of goals and tasks) or when they are bound to be verified (in the case of domain assumptions and quality constraints). Furthermore, the framework sends messages to these instances when there is a change of state (e.g. when they fail or succeed). Therefore, *AwReqs* can refer to requirements at the instance level (e.g., a single instance should not change its state to *Failed*, like *AR1*) or at the class (aggregate) level (e.g., 75% of the instances created in a specified period of time should be in the state *Satisfied*, like *AR6*).

It can be inferred from the above description that, in our approach, requirements (or domain assumptions) are not necessarily treated as invariants that must always be achieved (or should always be true). Instead, we accept the fact that the system may fail in achieving any of its initial requirements (or assumptions could turn out to be false) and, by considering feedback loops as first class citizens in the language, provide a way of specifying the level of criticality of each requirement and monitor the system to be aware of their failures. More details on this monitoring infrastructure and a list of *AwReq* patterns and their specification in an OCL-based language can be found in (Souza et al, 2011b).

The next step in the process consists of identifying parameters that, when changed, can have an effect on the relevant indicators. Parameters can be of two types. *Variation Points* consist of OR-refinements which are already present in high variability systems and just need to be labeled. According to Semmak et al (2008), this concept originally came from the field of feature modeling (Griss et al, 1998). For instance, in the Meeting Scheduler, the value of `VP1` specifies if timetables will be obtained by phone, via e-mail or automatically in the system's calendar. Figure 1 shows five variation points (`VP1`–`VP5`). The OR-refinement of goal *Manage meeting* was not considered a `VP` because it does not represent variants for the same purpose, but instead two possible outcomes for meetings: they are either canceled beforehand or, at their specified time, the secretary confirms if they occurred or not.

*Control Variables* are abstractions over large / repetitive variation points, e.g., `FhM` represents *From how Many* participants (a percentage) the system should collect timetables before considering the goal *Collect timetables* satisfied, abstracting over the (repetitive) OR-refinements that would have to be added in order to represent such variability. Other variables identified for the Meeting Scheduler are:

- *Required fields* (`RF`, attached to *Characterize meeting*) is an enumerated variable that can assume the values: *participants list only*, *short description required* or *full description required*;
- *Maximum Conflicts Allowed* (`MCA`, attached to *Let system schedule*) forces the system to choose a date in which the number of scheduling conflicts does not surpasses the value specified in this variable;
- *View private appointments* (`VPA`, attached to *Collect from system calendar*) can be either *yes* or *no*;
- *Rooms for Meetings* (`RfM`, attached to *Local rooms available*) indicate the number of rooms that the organization has made available for its employees to conduct meetings.

Having identified the parameters whose change can affect the indicators (represented by the *AwReqs*), the next step of the process is to model the nature of this effect using differential relations. For instance, $\Delta(AR8/RfM)[0, maxRooms] > 0$ represents the fact that, by increasing the number of *Rooms for Meetings*, the domain assumption *Local rooms available* will be satisfied more often. The numbers between square brackets indicate the interval in which this relation is valid ($maxRooms$ represents a qualitative value that should, eventually, be replaced by a precise number).

After modeling the effect of each indicator–parameter pair individually, a final refinement step analyzes the effects of the same indicators in combination to decide if they are cumulative and if they can be ordered (from greatest to smallest effect). More details on this process can be found in (Souza et al, 2011a).

Given the specification that results from this model, when *AwReqs* fail at runtime, a possible adaptation strategy is to perform reconfiguration, i.e., to change parameter values in order to improve the failing indicators, guided by the information represented in differential relations. In a recent paper (Souza et al, 2012b), we propose a framework that searches the solution space

for the best values to assign to system parameters, reconfiguring the system to adapt.

However, in this paper we focus on a different kind of strategy, one based on changing the requirements model — i.e., the problem space — as specified by *Evolution Requirements*. Unlike reconfiguration, which reasons over the model to try and find the best parameter values, *EvoReqs* prescribe specific requirement evolutions when certain situations present themselves, as illustrated in Section 1. We present this new family of requirements next.

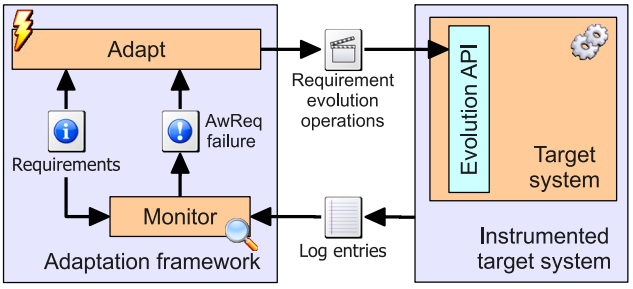


**Fig. 2** Conceptual architecture for a run-time adaptation framework.

## 4 Evolution Requirements

Evolution requirements specify changes to other requirements when certain conditions apply. For instance, suppose the stakeholders provide the following requirements:

- If the meeting organizer fails to *Characterize meeting* (*AR1*), she should **retry** after a few seconds;
- If there is a negative trend on the success rate of *Collect timetables* for two consecutive weeks (*AR3*), we can tolerate this at most once per trimester, **relaxing** the constraint to three weeks in a row;
- If local rooms are often unavailable (*AR8*), the Meeting Scheduler software cannot autonomously create new rooms (i.e., increase `RfM`). This task should be **delegated** to the management;
- If we realize that the domain assumption *Participants use the system calendar* is not true, **replace it** with a task that will enforce the usage of the system calendar.

We propose to represent these requirements by means of sequence of operations over goal model elements, in a way that can be exploited at runtime by an adaptation framework, which, acting like a controller in a control system, sends adaptation instructions to the target system. We call them *Evolution Requirements* (*EvoReqs*).

*EvoReqs* and *AwReqs* (cf. §3) complement one another, allowing analysts to specify the requirements for a feedback loop that operationalizes adaptation at runtime: *AwReqs* indicate the situations that require adaptation and *EvoReqs* prescribe what to do in these situations. It is important to note, however, that *EvoReqs* are not the only way to adapt to *AwReq* failures (we briefly discussed reconfiguration in the previous section). Analogously, *AwReq* failures are not the only event that can trigger *EvoReqs* (the framework proposed herein can be adapted to respond to, e.g., scheduled events).

The following subsections present *EvoReqs*, starting with low-level operations on requirements (4.1), then defining patterns to represent common adaptation strategies using these operations (4.2) and how this framework can accommodate reconfiguration as one possible strategy (4.3).

### 4.1 *EvoReq* Operations

Figure 2 shows a conceptual architecture for a runtime adaptation framework. The *Monitor* component has been proposed in (Souza et al, 2011b) and includes an instrumentation phase which augments the target system with logging capabilities. Here, the term *target system* is used as in Control Theory, i.e., the base system around which one defines a feedback loop (e.g., see (Hellerstein et al, 2004)). By analyzing the requirements (goal model with *AwReqs*, parameters, etc.) and the log entries, this component is able to conclude if and when certain *AwReqs* have failed.

These failures should then trigger an *Adapt* component that decides which requirement evolution operations the target system should execute (this decision process is further discussed in Section 5). These operations are obtained from the specification of *EvoReqs*, which are also part of the requirements depicted in Figure 2. *EvoReqs*, thus, are specified as a sequence of primitive operations which have an effect on the target system (TS) and/or on the adaptation framework (AF) itself, effectively telling them how to change (or, using a more evolutionary term, "mutate") the requirements model in order to adapt. The existing operations and their respective effects are shown in Table 1 (the set of operations could be extended if necessary).

As can be seen in the table, adaptation instructions have arguments which can refer to, among other things, system actors (`A`), requirements classes (uppercase `R`) or instances (lower-case `r`) and system parameters (`p`) and their values (`v`). Actors can be provided by any diagram that models external entities that interact with the system, e.g., $i^\star$ Strategic Dependency models (Yu et al, 2011). Requirements classes/instances are

**Table 1** Requirement evolution operations and their effect on the target system (TS) and/or the adaptation framework (AF).

| Instruction | Effect |
|---|---|
| `abort(ar)` | TS should "fail gracefully", which could range from just showing an error message to shutting the entire system down, depending on the system and the *AwReq* `ar` that failed. |
| `apply-config(C, L)` | TS should change from its current configuration to the specified configuration `C`. Argument `L` indicates if the change should occur at the class level (for future executions) and/or at the instance level (for the current execution). |
| `change-param([R\|r], p, v)` | TS should change the parameter `p` to the value `v` for either all future executions of requirement `R` or the requirement instance `r` currently being executed |
| `copy-data(r, r')` | TS should copy the data associated with performative requirement instance `r` (e.g., data provided by the user) to instance `r'`. |
| `disable(R), suspend(r)` | TS should stop trying to satisfy requirement instance `r` in the current execution, or requirement `R` from now on. If `r` (or `R`) is an *AwReq*, AF should stop evaluating it. |
| `enable(R), resume(r)` | TS should resume trying to satisfy requirement instance `r` in the current execution, or requirement `R` from now on. If `r` (or `R`) is an *AwReq*, AF should resume evaluating it. |
| `find-config(algo, ar)` | AF should execute algorithm `algo` to find a new configuration for the target system with the purpose of reconfiguring it. Other than the *AwReq* instance `ar` that failed, AF should provide to this algorithm the system's current configuration and the system's requirements model. |
| `initiate(r)` | TS should initialize the components related to `r` and start pursuing the satisfaction of this requirement instance. If `r` is an *AwReq* instance, AF should immediately evaluate it. |
| `new-instance(R)` | AF should create a new instance of requirement `R`. |
| `rollback(r)` | TS should undo any partial changes that might have been effected while the satisfaction of performative requirement instance `r` was being pursued and which would leave the system in an inconsistent state, as in, e.g., Sagas (Garcia-Molina and Salem, 1987). |
| `send-warning(A, ar)` | TS should warn actor `A` (human or system) about the failure of *AwReq* instance `ar` |
| `terminate(r)` | TS should terminate any component related to `r` and stop pursuing the satisfaction of this requirement instance. If `r` is an *AwReq* instance, AF should no longer consider its evaluation. |
| `wait(t)` | AF should wait for the amount of time `t` before continuing with the next operation. TS is also informed of the wait in case changes in the user interface are in order during the waiting time. |
| `wait-for-fix(ar)` | TS should wait for a certain condition that indicates that the problem causing the failure of *AwReq* `ar` has been fixed. |

provided by the monitoring component (Souza et al, 2011b), which represents the elements of the requirements model as UML classes each extending the appropriate class from the diagram shown in Figure 3. As mentioned in Section 3, run-time instances of these elements (such as the various meetings being scheduled) are then represented as objects that instantiate these classes. Finally, parameters are elicited during system identification, as also explained in Section 3.

Instructions `apply-config` and `find-config` also refer to configurations (`C`) and algorithms (`algo`), which will be further explained in Section 4.3.

Below, we show the specification of one of the examples presented earlier in this section: retry a goal when it fails.

```
t' = new-instance(T_CharactMeet);
copy-data(t, t');
terminate(t);
rollback(t);
wait(5s);
initiate(t');
```

Here, `t` represents an instance of task *Characterize meeting*, referred to by the instance of *AwReq AR1* that failed. The framework then creates another instance of the task, tells the target system to copy the data from the execution session of the failed task to the one of the

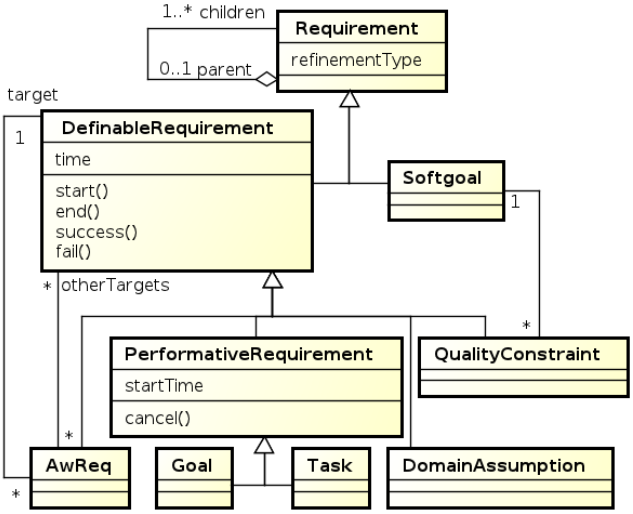


**Fig. 3** Class model for requirements in GORE, adapted from (Souza et al, 2011b).

new task, to terminate the failing components and rollback any partial changes made by them. After 5*s*, the framework finally instructs the target system to initiate the new task, thus accomplishing "retry after a few seconds".

Although evolution operations are generic, their effects on the target system are application-specific. For example, instructing the system to try a requirement again could mean, depending on the system and the requirement, retrying some operations autonomously or showing a message to the user explaining that she should repeat the actions she has just performed. Therefore, in order to be able to carry out these operations, the target system is supposed to implement an *Evolution API* that receives all operations of Table 1, for each requirement in the system's model. Obviously, as with any other requirement in a specification, each operation–requirement pair can be implemented on an as-needed basis.

Revisiting the previous example, `copy-data` should tell the Meeting Scheduler to copy the data related to the task that failed (e.g., information on the meeting that has already been filled in the system) to a new user session, `terminate` closes the screen that was being used by the meeting organizer to characterize the meeting, `rollback` deletes any partial changes that might have been saved, `wait` shows a message asking the user to wait for 5*s* and, finally, `initiate` should open a new screen associated with the new user session so the meeting organizer can try again. All this behavior is specific to the Meeting Scheduler and the task at hand and the way it will be implemented depends highly on the technologies chosen during its architectural design.

## 4.2 Adaptation Strategies as Patterns

The operations of Table 1 allow us to describe different *adaptation strategies* in response to *AwReqs* failures using *EvoReqs*. However, many *EvoReqs* might have similar structures, such as "wait $t$ seconds and try again, with or without copying data". Therefore, to facilitate their elicitation and modeling, we propose the definition of patterns[1] that represent common adaptation strategies. Table 2 shows the specification for some *EvoReq* patterns.

A strategy is defined by a name, a list of arguments that it accepts (with optional default values) and an algorithm (composed of Java$^{TM}$-style pseudo-code and evolution operations) to be carried out when the strategy is selected. For instance, `Retry` is defined with two parameters: `copy`, of Boolean type and with default value `true`; and `time`, of long integer type and no default value (which makes it mandatory when this pattern is used).

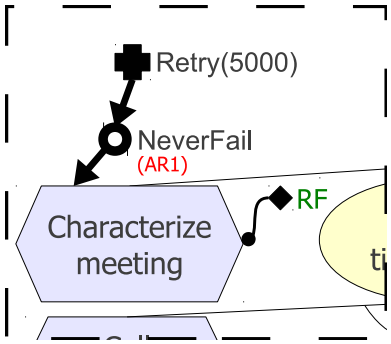


**Fig. 4** Graphical representation of an adaptation strategy in response to an *AwReq* failure.

Moreover, strategies are usually associated to failures of *AwReqs* and, therefore, we can also refer to the instance of the *AwReq* that failed using the keyword `awreq` in the pseudo-code. In other words, `awreq` is an implicit parameter that is available in all strategy definitions. Taking the `Retry` strategy again as an example, we can see that the failed *AwReq*'s target is assigned to variable `r` in the first line of the pseudo-code. Consequently, assuming that time is represented in milliseconds, the example from Section 4.1 could be more concisely expressed as `Retry(5000)`.

It is important to note, however, that the list in Table 2 is not intended to be exhaustive and new strategies can be created as needed. For instance, one could take inspiration from the design patterns for adaptation cataloged by Ramirez and Cheng (2010). After strategies have been elicited and represented as patterns, they can be associated with *AwReqs* and added to the requirements specification.[2] The use of patterns also allows us to add adaptation strategies to the goal model, as shown in Figure 4. This portion of the Meeting Scheduler's model shows the `Retry(5000)` pattern associated with failures of *AwReq AR1*.

## 4.3 Reconfiguration

According to Wang and Mylopoulos (2009), a system *configuration* is "a set of tasks from a goal model which, when executed successfully in some order, lead to the

[1] Here, we use the term *pattern* in its more general sense: "a form or model proposed for imitation" or "something designed or used as a model for making things" (cf. `http://www.merriam-webster.com/dictionary/pattern`). The reader should not confuse it with *design pattern*, a more common use for this word in Software Engineering.

[2] Note that, for consistency reasons, even a very simple *EvoReq* like aborting (which consists of a single operation) is represented as a strategy through the use of the pattern `Abort`.

**Table 2** Some *EvoReq* patterns and their specifications based on the evolution operations of Table 1.

```
Abort() {
  abort(awreq);
}
```

```
Delegate(a : Actor) {
  send-warning(a, awreq);
  wait-for-fix(awreq);
}
```

```
RelaxDisableChild(r : Requirement = awreq.target; level : Level = INSTANCE; child : Requirement
    ) {
  if ((level == CLASS) || (level == BOTH)) {
    disable(child.class);
  }

  if ((level == INSTANCE) || (level == BOTH)) {
    suspend(r);
    terminate(child);
    if (child.class = PerformativeRequirement) rollback(child);
    suspend(child);
    resume(r);
  }
}
```

```
Replace(r : Requirement = awreq.target; copy : boolean = true; level : Level = INSTANCE; r' :
    Requirement) {
  R = r.class;
  R' = r'.class;
  if ((level == CLASS) || (level == BOTH)) {
    disable(R);
    enable(R');
  }

  if ((level == INSTANCE) || (level == BOTH)) {
    if (R = PerformativeRequirement) && (R' = PerformativeRequirement) && (copy) copy-data(r, r
        ');
    terminate(r);
    if (R = PerformativeRequirement) rollback(r);
    suspend(r);
    initiate(r');
  }
}
```

```
Retry(copy: boolean = true; time: long) {
  r = awreq.target; R = r.class;
  r' = new-instance(R);
  if (copy) copy-data(r, r');
  terminate(r); rollback(r);
  wait(time);
  initiate(r');
}
```

```
StrengthenEnableChild(r : Requirement = awreq.target; level : Level = INSTANCE; child :
    Requirement) {
  if ((level == CLASS) || (level == BOTH)) {
    enable(child.class);
  }

  if ((level == INSTANCE) || (level == BOTH)) {
    suspend(r);
    resume(child);
    initiate(child);
    resume(r);
  }
}
```

```
Warning(a : Actor) {
  send-warning(a, awreq);
}
```

satisfaction of the root goal". We add to this definition the values assigned to each *control variable* elicited during system identification (cf. § 3). *Reconfiguration*, then, is the act of replacing the current configuration of the system with a new one in order to adapt.

As mentioned before, *EvoReqs* are the focus of this work and we have proposed a reconfiguration framework in a separate publication (Souza et al, 2012b). However, the *EvoReqs* framework proposed herein was designed in a way to facilitate the integration with one or more reconfiguration components. This is done by considering $Reconfiguration$ a type of adaptation strategy. *EvoReqs* can, thus, be used to specify that stakeholders would like to use reconfiguration, in one of two ways:

1. If stakeholders wish to apply a specific reconfiguration for a given failure, instructions like `change-param`, `enable`/`disable` and `initiate`/`terminate` can be used to describe the precise changes in requirements at class and/or instance level;
2. Instead, if there is no specific way to reconfigure, a reconfiguration algorithm that is able to compare the different alternatives should be executed using the `find-config` instruction, after which `apply-config` is called to inform the target system about the new configuration.

Below, we show the pattern that describes the adaptation strategy of option 2. The strategy receives as arguments an algorithm to find the new configuration, the *AwReq* that failed and thus triggered the strategy and the level at which the changes should be applied: class (future executions), instance (current execution) or both.

```
Reconfigure(algo: FindConfigAlgorithm, ar:
    AwReq, level: Level = INSTANCE) {
  C' = find-config(algo, ar)
  apply-config(C', level)
}
```

The state-of-the-art on goal-based adaptive systems provides several algorithms that are capable of finding a new system configuration. Wang and Mylopoulos (2009) propose algorithms that suggest a new configuration without the component that has been diagnosed as responsible for the failure; Nakagawa et al (2011) developed a compiler that generates architectural configurations by performing conflict analysis on KAOS goal models (van Lamsweerde, 2009); Fu et al (2010) use reconfiguration to repair systems based on an elaborate state-machine diagram that represents the life-cycle of goal instances at runtime; Peng et al (2010) assign preference rankings to softgoals (which can be dynamically changed at runtime) and determine the best configuration using a SAT solver; Khan et al (2008) apply Case-Based Reasoning to the problem of determining the best configuration; Dalpiaz et al (2012) propose an algorithm that finds all valid variants to satisfy a goal and compares them based on their cost and benefit. Moreover, in (Dalpiaz et al, 2010), reconfiguration is discussed in terms of interaction among autonomous, heterogeneous agents based on commitments.

Note that different reconfiguration algorithms may require different information from the model. For instance, (Wang and Mylopoulos, 2009) requires a goal model and a diagnosis pointing to the failing component, whereas (Peng et al, 2010) needs the preference rankings of softgoals. Analysts should provide the required information accordingly.

## 5 The *Zanshin* Framework

To operationalize *EvoReq* adaptation strategies at runtime in response to *AwReq* failures, we have developed a prototype framework called *Zanshin* (named after a term used in the Japanese martial arts that refers to a state of awareness). *Zanshin* receives notifications from the monitoring component about *AwReq* failures and executes an adaptation process that is explained in sub-Section 5.1. Then, sub-Section 5.2 presents more details on the framework's implementation.

### 5.1 The Adaptation Process

Using the language described in Section 4, requirements engineers can specify stakeholders' *EvoReqs* in a precise way (based on clearly-defined primitive operations) that can also be exploited at runtime by an adaptation framework (e.g., Figure 2). However, more than one *EvoReq* can be associated to each requirement divergence, which prompts the need for a process that coordinates their execution.

Here, we propose a process based on ECA rules for the execution of adaptation strategies in response to system failures. This process is summarized in the algorithm shown in Figure 5, which manipulates instances of the classes represented in the class model of Figure 6.

The process is triggered by *AwReq* evaluations, independent of the *AwReq* instance's final state (*Success*, *Failed* or *Canceled*). For instance, let us recall one of the examples in the beginning of Section 4: say the weekly success rate of *Collect timetables* has decreased twice in a row, causing the failure of *AR3* and starting the ECA process.

The algorithm begins by obtaining the *adaptation session* that corresponds to the class of said *AwReq*, creating a new one if needed (line 2). As shown in Figure 6,

```
1  processEvent(ar : AwReq) {
2    session = findOrCreateSession(ar.class);
3    session.addEvent(ar);
4    solved = ar.condition.evaluate(session);
5    if (solved) break;
6
7    ar.selectedStrategy = null;
8    for each s in ar.strategies {
9      appl = s.condition.evaluate(session);
10     if (appl) {
11       ar.selectedStrategy = s;
12       break;
13     }
14   }
15
16   if (ar.selectedStrategy == null)
17     ar.selectedStrategy = ABORT;
18
19   ar.selectedStrategy.execute(session);
20   ar.condition.evaluate(session);
21 }
```

**Fig. 5** Algorithm for responding to *AwReq* failures.

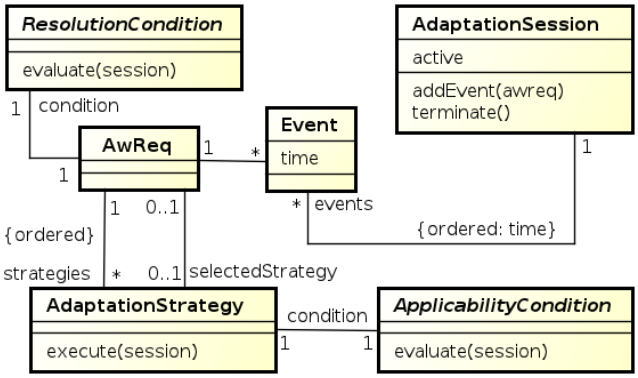


**Fig. 6** Entities involved in the ECA-based adaptation process.

an *adaptation session* consists of a series of events, referring to *AwReq* evaluations. This time-line of events can be later used to check if a strategy is applicable or if the problem has been solved (i.e., if the adaptation has been successful). Active sessions are stored in a repository (e.g., a hash table indexed by *AwReq* classes attached to the user session) which is managed by the `findOrCreateSession()` procedure. In the example, assuming it is the first time *AR3* fails, a new session will be created for it.

Then, the process adds the current *AwReq*'s evaluation as an event to the active session, immediately evaluates if the problem has been solved — this is done by considering the *AwReq*'s *resolution condition*, which analyzes the session's event time-line — and stops the process if the answer is affirmative (3–5). For example, the trivial case is considering the problem solved if the (next) *AwReq* evaluates to success, but this abstract class can be extended to provide different kinds of *resolution conditions*, including, e.g., involving a human-in-the-loop to confirm if the problem has indeed been solved, organizing conditions into AND/OR-refinement trees (like in a goal model), etc. For the running example, let us say that *AR3* has been associated with the aforementioned simple resolution condition. Since the *AwReq*'s state is *Failed*, the session is not considered solved and the algorithm continues.

If the current *AwReq* evaluation does not solve the issue, the process continues to search for an applicable *adaptation strategy* to execute in order to try and solve it (7–14). It does so by going through the list of strategies associated with the *AwReq* that failed in their predefined order (e.g., preference order established by the stakeholders) and evaluating their *applicability conditions*, breaking from the loop once an applicable strategy has been found. As with `ResolutionCondition`, `ApplicabilityCondition` is also abstract and should be extended to provide specific kinds of evaluations. For instance, apply a strategy "at most $N$ times per session/time period", "at most in $X\%$ of the failures/executions", "only during specified periods of the day", AND/OR-refinements, etc. (patterns can be useful here). Some conditions might even need to refer to some domain-specific properties or contextual information. If no applicable strategy is found, the process falls back to the *Abort* strategy (16–17).

Back to the running example, imagine now that the Meeting Scheduler designers have associated two strategies to *AR3*. First, relax it by replacing *AR3* with *AR3'*, which verifies if the success rate has decreased not in two, but in three consecutive weeks (i.e., `not TrendDecrease(G_CollectTime, 7d, 3)`). This strategy is associated with a condition that constraints its applicability to at most once a trimester. Second, the *Warning* strategy is also associated with *AR3*, sending a message to the IT support staff so they can take corrective action. To this strategy a simple applicability condition is associated, which always returns true. Therefore, if this is the first time *AR3* fails in the past three months, it will be relaxed to *AR3'*, otherwise the *Warning* strategy will be selected.

After the strategy is selected, it is executed and the session is given another chance to evaluate its resolution (sometimes we would like to consider the issue solved after applying a specific strategy, independent of future *AwReq* evaluations, e.g. when we use *Abort*). When an *adaptation session* is considered resolved, it should be *terminated*, which marks it as no longer being active. At this point, future *AwReq* evaluations would compose new *adaptation sessions*. Instead, if the algorithm ends without solving the problem, the framework will continue to work on it when it receives another *AwReq* evaluation and retrieves the same *adaptation session*, which is still active. Some *adaptation strategies* can force a re-evaluation of the *AwReq* when executed, which guarantees the continuity of the adaptation process.

For the *AR3* example, the session would remain active until another month has passed and *AR3'* is checked again. If the success rate increases, then *AR3'* will be satisfied, triggering another call to `processEvent()`, which would find *AR3*'s session and, according to the resolution condition, consider it solved and terminate it. If the rate decreases one more time, though, the *Warning* strategy is used and the session remains active until the following week. In Section 6, when we discuss the experiments with an ambulance dispatch system (A-CAD), this adaptation process is depicted once again with another example.

As this example illustrated, information on resolution and applicability conditions should be present in the requirements specification in order for the adaptation framework to use this process. We do not propose any particular syntax for the inclusion of this information in the specification (as will be shown later, in our experiments we have used a simple tabular notation). Furthermore, the ECA-based process is only one possible solution for the coordination and execution of adaptation strategies in response to *AwReq* failures at runtime. It can be replaced or combined with other processes that use *EvoReqs* and any extra specification necessary (e.g. applicability and resolution conditions) to: (a) select the best strategy to apply; (b) execute it; (c) check if the problem has been solved; (d) loop back to the start if it has not.

### 5.2 Implementation

To demonstrate the value *EvoReqs* can bring to the development of adaptive systems, we have developed the *Zanshin* framework together with a simulation component as the target system that mimics failure situations that could occur at runtime. In what follows, we provide more detail on the framework implementation. Section 6 describes experiments with the A-CAD and how it was simulated in *Zanshin*.

The framework was implemented as OSGi bundles (Core, Logging, Monitoring, Adaptation and Simulation) and their source code is available for download (`http://github.com/vitorsouza/Zanshin`). The Core bundle exposes four service interfaces, based on the conceptual architecture shown in Figure 2, each of which implemented by a different bundle:

- *Monitoring Service*: monitors the log provided by the target system and detects changes of state in *AwReq* instances, submitting these to the Adaptation Service. This component is further described in (Souza et al, 2011b);
- *Adaptation Service*: implements the ECA-based adaptation process described in Figure 5 (§5.1), analyzing the requirements specification and deciding which adaptation strategy to execute next;
- *Target System Controller Service*: implemented by the Simulation bundle, serves as a bridge between the adaptation framework and the target system, by implementing the operations of Table 1, which are called by the executed adaptation strategies;
- *Repository Service*: implemented by the Core bundle itself, stores the instances of the requirements models that are used by the other services.

Requirements models are specified using Eclipse Modeling Framework (EMF) meta-models: the Core component provides the basic GORE classes (cf. Figure 3) and the classes involved in the ECA-based process (cf. Figure 6). These meta-models are extended by the Simulation bundle to provide classes representing the requirements of the target system. For example, for the Meeting Scheduler there would be one EMF class for each requirement of the goal model shown earlier in Figure 1, extending the appropriate GORE/ECA classes.

Finally, the target system' requirements specification can be written as an EMF model, to be read by the framework, represented in memory as Java™ objects (using EMF's API) and stored in the Repository Service when the target system is executed. This way, the EMF model represents the requirements at the *class* level, whereas the objects stored in the Repository Service for each execution represent the requirements at the *instance* level.

At runtime, when the Monitoring Service detects an *AwReq* has changed state, it notifies the Adaptation Service, which executes the adaptation process described earlier in Section 5.1. When the adaptation strategy is chosen and executed, *EvoReq* operations are sent to the Target System Controller Service, which is responsible for adapting the target system. An example of this entire process is illustrated in the next section.

## 6 Experiments

An important aspect of any research proposal is validation. Hevner et al (2004) describe five categories of evaluation methods in Design Science: Observational, Analytical, Experimental, Testing and Descriptive. Methods range from simple description of scenarios up to full-fledge case studies which are conducted in business environments. In the previous section, we have used descriptive methods — in the form of *scenarios* and *informed argument* — to illustrate the usefulness of our approach. In this section, we describe the application

of experimental methods of evaluation — in particular, controlled experiments and simulations — that have been conducted using a larger experiment.

### 6.1 The A-CAD System

In order to provide initial validation of our proposed approach, we have conducted an experiment on the design of an Adaptive Computer-aided Ambulance Dispatch (A-CAD) system and its simulation at runtime using the *Zanshin* framework. The A-CAD was based on the report on the failure of the London Ambulance Service Computer Aided Despatch (LAS-CAD) System (Finkelstein, 1993). This case study was first presented at the $8^{th}$ International Workshop on Software Specification and Design (Finkelstein and Dowell, 1996) and became an exemplar in the Software Engineering community, being further analyzed in other venues such as the European Journal of Information Systems (Beynon-Davies, 1995), the Journal of the Brazilian Computer Society (Breitman et al, 1999), ACM SIGSOFT Software Engineering Notes (Kramer and Wolf, 1996), etc.

Being a real system and having so much available information — due to its failure and subsequent inquiry — makes the LAS-CAD a good choice for validation of new research proposals. In the case of our research on adaptive systems, this is especially true, given that the success of the LAS-CAD "would depend on the near 100% accuracy and reliability of the technology in its totality. Anything less could result in serious disruption to LAS operations" (Finkelstein, 1993). An adaptive CAD system could try and avoid the "serious disruptions" in its operations through adaptation[3].

In a technical report (Souza, 2012), we describe in detail the requirements elicitation process (both early and late requirements) and the application of our approach for the design of adaptive systems, including the identification of sixteen Awareness Requirements (*AR1–AR16*), five variation points (`VP1`–`VP5`), four control variables (`NoC` / *Number of Calls*, `NoSM` / *Number of Staff Members*, `MST` / *Minimum Search Time*, `LoA` / *Level of Automation*) and over thirty differential relations between indicators and parameters. Figure 7 shows the final (GORE-based) requirements specification for the A-CAD.

Analyzing the failure report and the different publications mentioned above, we have identified several failures which were considered as possible causes for the LAS-CAD demise, such as system misusage, transmission problems, unreliable software, call flooding, slow response speed, problems with the use of Mobile Data Terminals (MDTs), etc. Then, *AwReqs* were modeled so a feedback loop controller would be aware of these possible failures and would try to adapt to them. For instance, to address slow response speeds, four *AwReqs* were attached to the quality constraints of softgoals *Fast dispatch*, *Fast arrival* and *Fast assistance*; to try and mitigate call flooding, *AwReqs* have been associated with domain assumption *Up to <NoC> calls per day* (where the *Number of Calls* is calculated given the *Number of Staff Members* working, which allows management to increase it by hiring new employees), and so forth.

After parameters and differential relations have been modeled during System Identification, adaptation strategies were associated with each *AwReq*, specifying the adaptation part that follows monitoring in the feedback loop. Table 3 specifies an ordered list of adaptation strategies to be executed, in the specified order, in case their associated *AwReq* fail. *AwReq* patterns refer to elements of the goal model using mnemonics (the letter that precedes the underscore indicates the type of element — `Goal`, `Task`, `Domain` assumption, or `Quality` constraint). Moreover, the `Reconfigure()` pattern had its parameters omitted due to our focus here being on *EvoReqs*. The complete table can be seen in (Souza, 2012) and details on the reconfiguration algorithms that were used are in (Souza et al, 2012b).

### 6.2 Experiments with the A-CAD

With the parts of the specification of the A-CAD that are provided by Figure 7 and Table 3, we were able to simulate run-time failures of this system to evaluate the response of the *Zanshin* framework and the effectiveness of our proposals.

The first step consists in encoding the specification of the A-CAD in an EMF model so, as explained in Section 5.2, the framework can parse it and create an in-memory representation of the system requirements for every system execution. Figure 8 shows parts of this EMF model — points of ellipsis (...) indicate sections of the model that were omitted for brevity.

This model excerpt shows the specification of goal *Register call*, its child tasks (*Input emergency infor-*

[3] Note, however, that is not our intention to prove that the LAS would not have failed if it had been built as an adaptive system using our proposal. Many of the analyses conducted over the failure indicate that the procurement and the development processes were flawed, producing a bad quality system in general. Hence, if adaptation mechanisms had been developed to work with the LAS, there is no guarantee these would have been properly developed and have good quality and would therefore also be prone to failure. Our objectives here are to learn from the problems detected in the LAS in order to identify critical requirements and use those to develop a new system which would, in theory, be designed properly and have good quality in general.

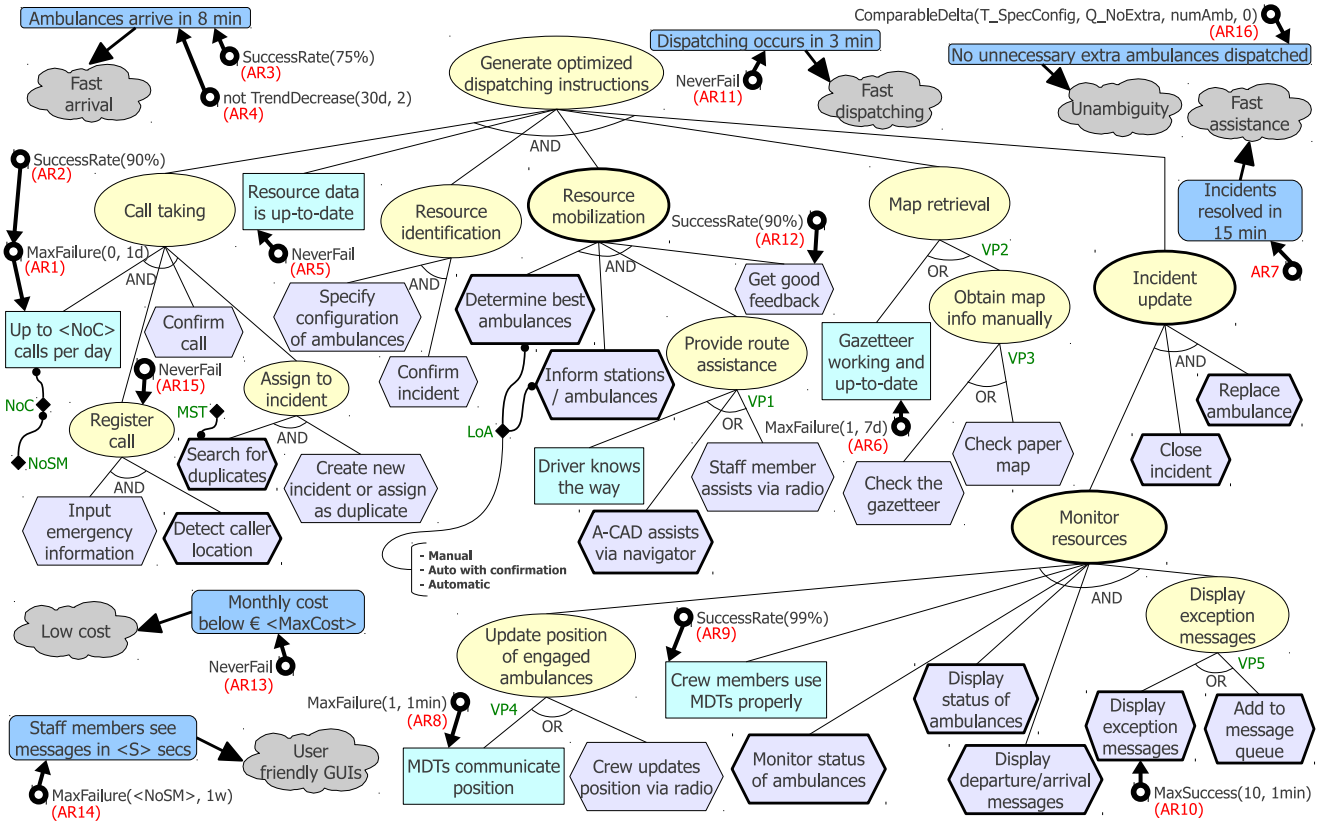


**Fig. 7** The goal model for the A-CAD system, after applying our approach for the design of adaptive systems.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <acad:AcadGoalModel ...>
3   <rootGoal xsi:type="acad:G_GenDispatch">
4    <children xsi:type="acad:G_CallTaking">
5     <children xsi:type="acad:D_MaxCalls"/>
6     <children xsi:type="acad:G_RegCall">
7      <children xsi:type="acad:T_InputInfo
          "/>
8      <children xsi:type="acad:T_DetectLoc
          "/>
9     </children>
10    ...
11   </children>
12   ...
13  </rootGoal>
14  ...
15  <awReqs xsi:type="acad:AR15" target="//
        @rootGoal/@children.0/@children.1">
16   <condition xsi:type="model:
        SimpleResolutionCondition"/>
17   <strategies xsi:type="model:
        RetryStrategy" time="5000">
18    <condition xsi:type="model:
         MaxExecApplicabilityCondition"
         maxExecutions="1"/>
19   </strategies>
20   <strategies xsi:type="model:
        RelaxDisableChildStrategy" child
        ="//@rootGoal/@children.0/@children
        .1/@children.1">
21    <condition xsi:type="model:
         MaxExecApplicabilityCondition"
         maxExecutions="1"/>
22   </strategies>
23  </awReqs>
24 </acad:AcadGoalModel>
```

**Fig. 8** The A-CAD requirements specified as an EMF model.

*mation* and *Detect caller location*) and ancestor goals (*Call taking* and *Generate optimized dispatching instructions*) in lines 3–9. Line 15 contains the specification of *AwReq AR15*, which refers to *Register call* as its target using EMF's syntax for references within a model (i.e., starting at the root goal, navigate to the child with index 0, then in that element navigate to the child of index 1).

*AR15* is specified to have a simple resolution condition — i.e., if the *AwReq* evaluation succeeded, the problem is solved — and two associated adaptation strategies, as specified in Table 3: `Retry(5000)` and `RelaxDisableChild(T_DetectLoc)`. Both strategies are applicable at most once during an adaptation session, as can be seen in the specification. The rationale behind this specification is the following: if the staff member cannot register the call, first assume it is a glitch in the input form and just try again. If the goal is still not satisfied, check if it is a problem with caller detection and disable that part (the staff member should then insert the location of the caller manually), checking if the goal is satisfied this way.

After the A-CAD specification has been represented in EMF, an implementation of the Target System Controller Service (cf. §5.2) specifically for the A-CAD sim-

**Table 3** Specification of *EvoReqs* elicited for the A-CAD experiment.

| ***AwReq*** | ***AwReq* Pattern** | **List of *EvoReq* Adaptation Strategies** |
|---|---|---|
| AR1 | `NeverFail(T_InputInfo)` | 1. *Warning("AS Management")*<br>2. *Reconfigure()* |
| AR2 | `SuccessRate(AR1, 90%)` | 1. *Warning("AS Management")*<br>2. *Reconfigure()* |
| AR3 | `SuccessRate(Q_AmbArriv, 75%)` | 1. *Reconfigure()* |
| AR4 | `not TrendDecrease(Q_AmbArriv, 30d, 2)` | 1. *Replace(AR4, AR4_60Days) + StrengthenReplace(AR3, AR3_80Pct)*<br>2. *Reconfigure()* |
| AR5 | `NeverFail(D_DataUpd)` | 1. *Delegate("Staff Member")* |
| AR6 | `MaxFailure(D_GazetUpd, 1, 7d)` | 1. *Reconfigure()* |
| AR7 | | 1. *Reconfigure()* |
| AR8 | `MaxFailure(D_MDTPos, 1, 1min)` | 1. *Replace(D_MDTPos_20Secs)*<br>2. *Replace(AR8, AR8_45Secs)*<br>3. *Replace(AR8_45Secs, AR8_30Secs)*<br>4. *Retry(60000)*<br>5. *Reconfigure()* |
| AR9 | `SuccessRate(D_MDTPos, 1, 1min)` | 1. *Reconfigure()* |
| AR10 | `MaxSuccess(T_Except, 10, 1min)` | 1. *Reconfigure()* |
| AR11 | `NeverFail(Q_Dispatch)` | 1. *Reconfigure()* |
| AR12 | `SuccessRate(T_Feedback, 90%)` | 1. *Reconfigure()* |
| AR13 | `NeverFail(Q_MaxCost)` | 1. *Reconfigure()* |
| AR14 | `MaxFailure(Q_MsgTime, <NoSM>, 1w)` | 1. *Reconfigure()* |
| AR15 | `NeverFail(G_RegCall)` | 1. *Retry(5000)*<br>2. *RelaxDisableChild(T_DetectCaller)* |
| AR16 | `ComparableDelta(T_SpecConfig, Q_NoExtra, numAmb, 0)` | 1. *Reconfigure()* |

ulation has to be provided. In a real setting, this controller would be the connection between the running A-CAD and *Zanshin*, effecting the application-specific changes related to each *EvoReq* operation (cf. §4.1). In our experiments, however, we have instead implemented simulations of the A-CAD system, which call the life-cycle methods expected by the monitoring infrastructure (Souza et al, 2011b) and acknowledges the reception of *EvoReq* operations, changing the requirements model as instructed.

When this simulation is ran, the A-CAD specification is read and stored in the repository and life-cycle methods referring to tasks *Input emergency information* and *Detect caller location* are sent by the simulated system. The monitoring infrastructure detects *AR15* has changed its state (again, details in Souza et al (2011b)), and *Zanshin* conducts the ECA-based coordination process, producing a log similar to the one shown in Figure 9. In the figure, messages are prefixed with `TS` and `AF` to indicate if they originate from the target system or the adaptation framework, respectively, which run in separate threads. This is done to resemble more closely a real life situation, in which the target system is a separate component from the adaptation framework.

The log shows the adaptation framework receiving notification of *AR15*'s failure (line 1), creating a new adaptation session `S1` for it (2) and searching for a suitable adaptation strategy to be applied, executing the `Retry(5000)` strategy (4–6). Then the simulated target system acknowledges the reception of the commands included in that pattern's definition (7–12) — see Table 2 —, and the adaptation framework verifies that the problem has not yet been solved (13).

After a while, the monitoring component notifies one more failure of *AR15* (line 15), prompting the adaptation framework to retrieve the same adaptation session `S1` as before, realizing that it has not yet been solved (16–17). *Zanshin* then proceeds to searching for a suitable adaptation strategy, but `Retry(5000)` cannot be used again in the same session due to its applicability condition (18). The framework ends up selecting

```
1  AF: Processing state change: AR15 ->
       Failed
2  AF: (S1) Created new session for AR15
3  AF: (S1) The problem has not yet been
       solved...
4  AF: (S1) RetryStrategy is applicable.
5  AF: (S1) Selected: RetryStrategy
6  AF: (S1) Applying strategy RetryStrategy(
       true; 5000)
7  TS: Received: new-instance(G_RegCall)
8  TS: Received: copy-data(iG_RegCall,
       iG_RegCall)
9  TS: Received: terminate(iG_RegCall)
10 TS: Received: rollback(iG_RegCall)
11 TS: Received: wait(5000)
12 TS: Received: initiate(iG_RegCall)
13 AF: (S1) The problem has not yet been
       solved...
14 ------------------------------------------

15 AF: Processing state change: AR15 ->
       Failed
16 AF: (S1) Retrieved existing session for
       AR15
17 AF: (S1) The problem has not yet been
       solved...
18 AF: (S1) RetryStrategy is not applicable
19 AF: (S1) RelaxDisableChildStrategy is
       applicable.
20 AF: (S1) Selected:
       RelaxDisableChildStrategy
21 AF: (S1) Applying strategy
       RelaxDisableChildStrategy(G_RegCall;
       Instance level only; T_DetectLoc)
22 TS: Received: suspend(iG_RegCall)
23 TS: Received: terminate(iT_DetectLoc)
24 TS: Received: rollback(iT_DetectLoc)
25 TS: Received: resume(iG_RegCall)
26 AF: (S1) The problem has not yet been
       solved...
27 ------------------------------------------

28 AF: Processing state change: AR15 ->
       Succeeded
29 AF: (S1) Retrieved existing session for
       AR15
30 AF: (S1) The problem has been solved.
       Terminate S1.
```

**Fig. 9** *Zanshin* execution log for the *AR15* simulation.

`RelaxDisableChild(T_DetectCaller)` and executing it (19–21), which again is recognized by the target system controller (22–26).

Finally, the monitoring infrastructure indicates that *AR15* has been satisfied (line 28), so the adaptation process can retrieve session `S1`, mark the problem as solved and terminate it. From this point on, further failures of *AR15* from the same user will create a new adaptation session.

As the log shows, the framework is able to execute the specified adaptation strategies, sending *EvoReq* operations to the target system, which should then adapt according to the instructions. Other than demonstrating the usefulness of our proposed approach, such operationalization of *EvoReqs* can help in the development of adaptive systems by separating the adaptation concerns into a specific component.

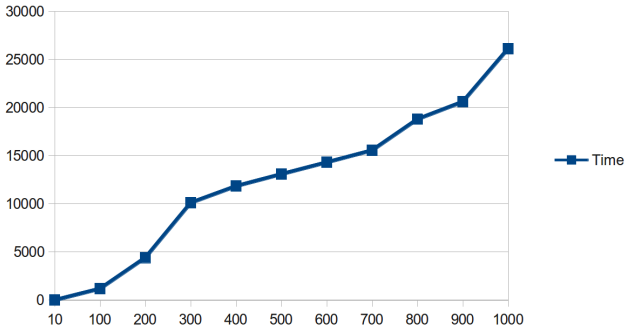


**Fig. 10** Results of the scalability tests of *Zanshin*.

### 6.3 Performance

Other than demonstrating the usefulness of our approach using the A-CAD experiment, we have also evaluated the performance of *Zanshin*'s implementation, by developing a simulation in which goal models of different sizes (100–1000 elements) are built and have an *AwReq* failing at runtime. The framework applies the adaptation strategy that is also included in the specification and the target system (i.e., the simulation) acknowledges it. The simulation was ran ten times for each goal model size and the running time of the framework was calculated. Average times in milliseconds for each goal model size are shown in Figure 10 (the running time of the target system was irrelevant in comparison and, therefore, not included in the graph).

As the graph shows, the adaptation framework scales linearly with the size of the goal model. The interested reader can experiment the simulations for themselves by downloading its source code. Furthermore, the target system and adaptation framework can be ran in a separate computers, reducing the impact of the adaptation process even further.

## 7 Related Work

The Rainbow framework (Garlan et al, 2004, Cheng et al, 2009b) adopts an architectural approach, using an architectural model as centerpiece for adaptation. Adaptation rules monitor operational conditions for the system and define actions if the conditions are unfavorable. For example, given a news website as target system, the adaptation mechanisms should keep the balance among three quality objectives: (a) performance (b) cost and (c) content fidelity. During a case that the response time is low a triggered adaptation strategy can either enlist more servers or switch off the multimedia mode causing the cost or content fidelity requirements respectively to fail. To restore failed requirements, there are adaptation strategies that when response time in-

creases discharge the extra servers or switch on the multimedia mode increasing the fidelity.

The adaptation mechanism in Rainbow is similar to our proposal here in that it consists of rules that monitor the target system, evaluate recorded data and if a constraint violation is detected then an adaptation strategy is triggered. Their strategies involve changing the components of the architecture, whereas our rules change requirements. An advantage of the Rainbow approach is the reusability of evolution rules for systems that share the same architecture and similar requirements.

Several other approaches for the design of adaptive systems focus on architectural solutions for this problem, such as the proposal of Kramer and Magee (2009), the work of Sousa et al (2009), the SASSY framework (Menasce et al, 2011), among others. These approaches usually express adaptation / evolution requirements in a quantitative manner (e.g., utility functions) and focus on quality of service (i.e., non-functional requirements). In comparison, our research is focused on requirements (goal) models, allowing stakeholders and requirements engineers to reason about adaptation on a higher level of abstraction. For this reason, in the rest of this section we restrict ourselves to approaches that, like ours, are focused on requirements.

Our approach is quite similar to FLAGS (Baresi and Pasquale, 2010). This service-oriented approach allows for the definition of adaptive goals which, when triggered by a goal not being satisfied, execute a set of adaptation actions that can change the system's goal model in different ways — add/remove/modify goals or agents, relax a goal, etc. — and in different levels — in transient or permanent ways. FLAGS is based on Linear Temporal Logic (LTL) and our approach is less heavy-handed in the formalism that is used than logic-based formalisms such as LTL, which has been found to be difficult in many practical settings. Furthermore, our approach is more general, offering a more varied set of operations over the goal model and allowing for extensible applicability/resolution conditions for adaptation strategies. On the other hand, FLAGS deals with synchronization and conflict resolution of adaptation goals, whereas *EvoReqs* just delegate these issues to the target system, sending instructions according to the specification of adaptation strategies. Considering these issues is a good opportunity for future work. The RELAX framework (Whittle et al, 2009) is similar to FLAGS, although it does not provide a runtime framework that operationalizes adaptation.

Another similar work is proposed by Fu et al (2010). Their approach represents the life-cycle of instances of goals at runtime using a state-machine diagram and, based on it, an algorithm can prevent possible failures or repair the system in case of requirements deviation. Their proposal, however, works at the instance-level only and does not change the system in a "from now on" fashion. Moreover, the list of possible adaptation strategies is fixed, whereas *EvoReqs* offers a fixed set of operations that can compose many different kinds of adaptation strategies. Failure prevention can also be implemented in our approach by specifying *AwReqs* not only on system failures but also on indications they are about to occur (if possible). *EvoReqs* associated with these *AwReqs* could then enact preventive measures, avoiding the failure altogether.

Most requirements-based adaptive systems proposals focus on the solution space. Qureshi and Perini (2010) focus on service-based applications and adapt by searching for new services at runtime. Brown et al (2006) extends LTL with an *Adapt* operator, encapsulating A-LTL formalisms in specifications, which allows the system to switch between operational domains. Approaches that perform adaptation by reconfiguration, such as the ones cited in Section 4.3, also fall into this category. Our work, on the other hand, proposes to adapt by changing the requirements (problem) space instead.

Another important aspect of adaptation related to our work is modeling and managing variability among applicable executions of a target system. We introduced earlier the concepts of variation points and control variables to model variability at the requirements level. Griss et al (1998) introduce variability as the key for exploiting reusable software features and propose the terms variation point and variant. Hallsteinsen et al (2008) point out that Software Product Lines (SPLs) lead to the development of applications for different domains composed in terms of reusable parts. However, as applications become more demanding, it is hard to foretell required variability at runtime. Therefore, there is a need for Dynamic Software Product Lines (DSPLs) where adaptation becomes critical in order to cope with changes in user requirements and the environment.

An interesting approach for developing adaptive systems based on DSPLs is the FeautureAce framework (Rosenmüller et al, 2011). FeatureAce allows manual or automatic adaptation by selecting a set of features from a static SPL, but also allowing adaptation rules that are triggered at runtime. Selected features are bound dynamically and the final reconfiguration is validated by a SAT solver to spot inconsistencies. A similar feature-oriented approach (Shen et al, 2011) divides available features represented via alternative and optional constructions. Features are inter-related through dependency and other constraints that define a space of possible variations. The adaptation process is based on

ECA (Event-Condition-Action) rules that mandate at runtime an optimal reconfiguration. Dinkelaker et al (2010) also express variability at runtime by specifying DSPLs through constraint relations among the features that constitute the software. As in previous approaches the features are activated or deactivated at run-time resulting in reconfigurations. Instead of declared rules and satiafiability checks, the validity of these reconfigurations is established with the use of aspect oriented models taking that way into account the context of the application's execution.

The problem of requirements evolution has mainly been addressed in the context of software maintenance. Thus, most research on this topic treats it as a post-implementation phenomenon (e.g., "evolution of requirements refers to changes that take place in a set of requirements after initial requirements engineering phase" (Antón and Potts, 2001)) caused by changes in the operational environment, user requirements, operational anomalies, etc. A lot of research has been devoted to the classification of types of changing requirements such as mutable, adaptive, emergent, etc. (Harker et al, 1993) and factors leading to these changes. Generally, these changes are viewed as being unanticipated and thus as not being able to be modeled a priori (Ernst et al, 2011a). Our work is quite different in this respect as we use *EvoReqs* to define trajectories for possible runtime requirements changes under particular circumstances. Clearly, not all requirements changes can be anticipated, but in this work we focused on modeling those that capture what the system should do in case it fails to meet its objectives. The triggers for these changes are clearly identifiable as requirements divergences can be anticipated. Nevertheless, these changes represent requirements evolution as they modify the original system requirements.

Requirements evolution research has focused on modeling requirements change and its impact on the system. For instance, in (Lam and Loomes, 1998), environment changes are propagated through requirements changes and down to design. Each triggered requirements change is analyzed in terms of its risks and the impact it has on the users' needs. Since we are dealing with anticipated and explicitly specified requirements changes, the analysis of their impact on the system can be carefully predicted. Another important aspect of requirement evolution is the completeness and consistency of requirements models. For instance, to address this, (Zowghi and Offen, 1997) proposes a formal approach based to requirements evolution utilizing non-monotonic default logics with belief revision. In our approach, we assume that the responsibility for requirements consistency rests with the modeler.

## 8 Discussion

One of the hallmarks of goal-oriented RE is its ability to systematically elicit, capture and analyze alternative ways to refine requirements. *EvoReqs* do not provide this opportunity since they are currently represented in a non-intentional way, as ECA rules. While in the method presented in this paper there are ways to support various ways of evolving system requirements with different conditions specifying their applicability, the current approach does not provide for a full-fledged analysis of alternative system evolutions. Thus, modeling and trade-off analysis of possible requirements evolutions using the common quality criteria of cost, customer satisfaction, etc. or the criteria especially relevant for system adaptation, such as the familiarity of the new solution (Ernst et al, 2011b), is not supported.

For instance, after a failure of the previously mentioned *AR3*, the two adaptation strategies available are to relax it or to send a warning to a member of the IT staff for manual intervention. Obviously, albeit risky, postponing any action (i.e., relaxation) may prove to be the most cost-effective strategy in case the negative trend is reversed in the following month. However, the costly manual intervention is less risky. Depending on the relative importance of risk vs. cost, the ordering/application of the two strategies will be different. Thus, the framework for systematic elicitation and analysis of adaptation strategies given the relevant quality criteria would be a welcome addition to the approach presented here.

A failure of an *AwReq* attached to a domain assumption indicates that the environment is in a different state than anticipated, i.e., in a different context. From this point of view, adaptation strategies associated with such failures represent ways to adapt the running system to the new context. Therefore, exploring connections between this framework and the goal-oriented context approaches (e.g., (Lapouchnian and Mylopoulos, 2009)) as a way to support context awareness in adaptive systems seems to be a worthy endeavor.

As already mentioned, here we treat system reconfiguration as a possible adaptation strategy. Our qualitative requirements-driven adaptation approach (Souza et al, 2012b) that relies on the equations produced using the system identification process itself has a complex adaptation loop that needs to be integrated into the feedback loop presented in this paper in order to consistently and reliably combine requirements-based system evolution and adaptation.

The use of rule sets in our framework constitutes a significant limitation. Large rule sets are hard to evolve, as it becomes increasingly difficult to understand what

does a change entail. Moreover, attention needs to be paid to the case where conflicting rules fire at the same time. Therefore, our method puts a lot of responsibility on the target system's designers, who need to be concerned with issues such as consistency, correctness and completeness.

In addition, the operationalization of *EvoReqs* assumes the target system can be appropriately instrumented, which might make the approach difficult to apply on legacy systems or systems that rely heavily on third-party components/services. Furthermore, when stakeholder requirements are very complex, representing them using adaptation strategies, applicability and resolution conditions can make the model difficult to read. Finally, our current implementation deals with *AwReq* failures separately and is not able to handle multiple concurrent failures.

All these limitations provide opportunities for further research, which may also include an experiment with the complete framework and a real application for further validation, the development of a CASE tool to help in model design and analysis.

## 9 Conclusions

In this paper, we have characterized a new family of requirements, called *Evolution Requirements*, which specify changes to other requirements when certain conditions apply. We have also proposed an approach to model this type of requirement and to operationalize them at runtime in order to provide adaptivity capabilities to a target system. This approach allows us to explicitly and precisely model changes to requirements models in response to certain conditions, such as requirements failures.

We are currently studying design techniques for controllers that would react to undesirable situations (e.g., a failed requirement) by changing one or more control variables, thereby changing the behavior of the system and/or the state of the environment. We are also beginning to study the adoption of ideas from architecture-based adaptation frameworks, such as the Rainbow project (Garlan et al, 2004), so that proposed adaptations take into account what is a feasible change at the architectural level and what is not.

**Acknowledgements** This work has been supported by the ERC advanced grant 267856 "Lucretius: Foundations for Software Evolution" (unfolding during the period of April 2011 – March 2016) — http://www.lucretius.eu.

## References

Andersson J, de Lemos R, Malek S, Weyns D (2009) Modeling Dimensions of Self-Adaptive Software Systems. In: Cheng BHC, de Lemos R, Giese H, Inverardi P, Magee J (eds) Software Engineering for Self-Adaptive Systems, Lecture Notes in Computer Science, vol 5525, Springer, pp 27–47

Antón AI, Potts C (2001) Functional Paleontology: System Evolution as the User Sees It. In: Proc. of the 23rd International Conference on Software Engineering, IEEE, pp 421–430

Baresi L, Pasquale L (2010) Adaptive Goals for Self-Adaptive Service Compositions. In: Proc. of the 2010 IEEE International Conference on Web Services, IEEE, pp 353–360

Bennett KH, Rajlich VT (2000) Software Maintenance and Evolution: a Roadmap. In: Proc. of the 22nd International Conference on Software Engineering, ACM, pp 73–87

Beynon-Davies P (1995) Information systems 'failure': the case of the London Ambulance Service's Computer Aided Despatch project. European Journal of Information Systems 4(3):171–184

Breitman KK, Leite JCSP, Finkelstein A (1999) The world's a stage: a survey on requirements engineering using a real-life case study. Journal of the Brazilian Computer Society 6(1)

Brown G, Cheng BHC, Goldsby HJ, Zhang J (2006) Goal-oriented Specification of Adaptation Requirements Engineering in Adaptive Systems. In: Proc. of the 2006 International Workshop on Self-adaptation and Self-managing Systems, ACM, pp 23–29

Brun Y, et al (2009) Engineering Self-Adaptive Systems through Feedback Loops. In: Cheng BHC, de Lemos R, Giese H, Inverardi P, Magee J (eds) Software Engineering for Self-Adaptive Systems, Lecture Notes in Computer Science, vol 5525, Springer, pp 48–70

Cheng BHC, et al (2009a) Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng BHC, de Lemos R, Giese H, Inverardi P, Magee J (eds) Software Engineering for Self-Adaptive Systems, Lecture Notes in Computer Science, vol 5525, Springer, pp 1–26

Cheng SW, Garlan D, Schmerl B (2009b) 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, pp 132–141

Dalpiaz F, Chopra AK, Giorgini P, Mylopoulos J (2010) Adaptation in Open Systems: Giving Interaction Its Rightful Place. In: Parsons J, Saeki M, Shoval P, Woo C, Wand Y (eds) Conceptual Modeling – ER

2010, Lecture Notes in Computer Science, vol 6412, Springer, pp 31–45

Dalpiaz F, Giorgini P, Mylopoulos J (2012) Adaptive socio-technical systems: a requirements-based approach. Requirements Engineering pp 1–24

Dardenne A, van Lamsweerde A, Fickas S (1993) Goal-directed Requirements Acquisition. Science of Computer Programming 20(1-2):3–50

Dinkelaker T, Mitschke R, Fetzer K, Mezini M (2010) A Dynamic Software Product Line Approach Using Aspect Models at Runtime. In: Proc. of the 1st International Workshop on Composition: Objects, Aspects, Components, Services and Product Lines, CEUR, pp 11–18

Ernst NA, Borgida A, Jureta I (2011a) Finding Incremental Solutions for Evolving Requirements. In: Proc. of the 19th International Requirements Engineering Conference, IEEE, pp 15–24

Ernst NA, Borgida A, Mylopoulos J (2011b) Requirements Evolution Drives Software Evolution. In: Proc. of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution, ACM, pp 16–20

Ernst NA, Borgida A, Mylopoulos J, Jureta I (2012) Agile Requirements Evolution via Paraconsistent Reasoning. In: Proc. of the 24th International Conference on Advanced Information Systems Engineering (to appear)

Finkelstein A (1993) Report of the inquiry into the London Ambulance Service (Electronic Version). Tech. rep., South West Thames Regional Health Authority

Finkelstein A, Dowell J (1996) A Comedy of Errors: the London Ambulance Service case study. In: Proc. of the 8th International Workshop on Software Specification and Design, IEEE, pp 2–4

Fu L, Peng X, Yu Y, Zhao W (2010) Stateful Requirements Monitoring for Self-Repairing of Software Systems. Tech. rep., FDSE-TR201101 (available online: `http://www.se.fudan.sh.cn/paper/techreport/1.pdf`), Fudan University, China

Garcia-Molina H, Salem K (1987) Sagas. ACM SIGMOD Record 16(3):249–259

Garlan D, Cheng SW, Huang AC, Schmerl B, Steenkiste P (2004) Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. Computer 37(10):46–54

Griss ML, Favaro J, D'Alessandro M (1998) Integrating feature modeling with the RSEB. In: Proc. of the 5th International Conference on Software Reuse, IEEE, pp 76–85

Hallsteinsen S, Hinchey M, Park S, Schmid K (2008) Dynamic Software Product Lines. Computer 41(4):93–95

Harker SD, Eason KD, Dobson JE (1993) The Change and Evolution of Requirements as a Challenge to the Practice of Software Engineering. In: Proc. of the 1993 IEEE International Symposium on Requirements Engineering, IEEE, pp 266–272

Hellerstein JL, Diao Y, Parekh S, Tilbury DM (2004) Feedback Control of Computing Systems, 1st edn. Wiley

Hevner AR, March ST, Park J, Ram S (2004) Design Science in Information Systems Research. MIS Quarterly 28(1):75–105

Jureta I, Mylopoulos J, Faulkner S (2008) Revisiting the Core Ontology and Problem in Requirements Engineering. In: Proc. of the 16th IEEE International Requirements Engineering Conference, IEEE, pp 71–80

Kephart JO, Chess DM (2003) The vision of autonomic computing. Computer 36(1):41–50

Khan MJ, Awais MM, Shamail S (2008) Enabling Self-Configuration in Autonomic Systems using Case-Based Reasoning with Improved Efficiency. In: Proc. of the 4th International Conference on Autonomic and Autonomous Systems, IEEE, pp 112–117

Kramer J, Magee J (2009) A Rigorous Architectural Approach to Adaptive Software Engineering. Journal of Computer Science and Technology 24(2):183–188

Kramer J, Wolf AL (1996) Succeedings of the 8th International Workshop on Software Specification and Design. ACM SIGSOFT Software Engineering Notes 21(5):21–35

Lam W, Loomes M (1998) Requirements Evolution in the Midst of Environmental Change: a Managed Approach. In: Proc. of the 2nd Euromicro Conference on Software Maintenance and Reengineering, IEEE, pp 121–127

van Lamsweerde A (2009) Requirements Engineering: From System Goals to UML Models to Software Specifications, 1st edn. Wiley

Lapouchnian A, Mylopoulos J (2009) Modeling Domain Variability in Requirements Engineering with Contexts. In: Laender A, Castano S, Dayal U, Casati F, de Oliveira J (eds) Conceptual Modeling - ER 2009, Lecture Notes in Computer Science, vol 5829, Springer, pp 115–130

Lehman MM (1979) On understanding laws, evolution, and conservation in the large-program life cycle. Journal of Systems and Software 1:213–221

Menasce DA, Gomaa H, Malek S, Sousa JaP (2011) SASSY: A Framework for Self-Architecting Service-Oriented Systems. IEEE Software 28(6):78–85

Nakagawa H, Ohsuga A, Honiden S (2011) gocc: A Configuration Compiler for Self-adaptive Systems Using Goal-oriented Requirements Description. In: Proc. of

the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, ACM, pp 40–49

Peng X, Chen B, Yu Y, Zhao W (2010) Self-Tuning of Software Systems through Goal-based Feedback Loop Control. In: Proc. of the 18th IEEE International Requirements Engineering Conference, IEEE, pp 104–107

Qureshi NA, Perini A (2010) Requirements Engineering for Adaptive Service Based Applications. In: Proc. of the 18th IEEE International Requirements Engineering Conference, IEEE, pp 108–111

Ramirez AJ, Cheng BHC (2010) Design Patterns for Developing Dynamically Adaptive Systems. In: Proc. of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, ACM, pp 49–58

Rosenmüller M, Siegmund N, Pukall M, Apel S (2011) Tailoring Dynamic Software Product Lines. In: Proc. of the 10th ACM International Conference on Generative Programming and Component Engineering, ACM, pp 3–12

Semmak F, Gnaho C, Laleau R (2008) Extended Kaos to Support Variability for Goal Oriented Requirements Reuse. In: Ebersold S, Front A, Lopistéguy P, Nurcan S (eds) Proc. of the International Workshop on Model Driven Information Systems Engineering: Enterprise, User and System Models, CEUR, pp 22–33

Shen L, Peng X, Liu J, Zhao W (2011) Towards Feature-Oriented Variability Reconfiguration in Dynamic Software Product Lines. In: Schmid K (ed) Top Productivity through Software Reuse, Lecture Notes in Computer Science, vol 6727, Springer, pp 52–68

Sousa JaP, Balan RK, Poladian V, Garlan D, Satyanarayanan M (2009) A Software Infrastructure for User–Guided Quality–of–Service Tradeoffs. In: Cordeiro J, Shishkov B, Ranchordas A, Helfert M (eds) Software and Data Technologies, Communications in Computer and Information Science, vol 47, Springer, pp 48–61

Souza VES (2012) An Experiment on the Development of an Adaptive System based on the LAS-CAD. Tech. rep., University of Trento (available at: `http://disi.unitn.it/~vitorsouza/a-cad/`)

Souza VES, Lapouchnian A, Mylopoulos J (2011a) System Identification for Adaptive Software Systems: a Requirements Engineering Perspective. In: Jeusfeld M, Delcambre L, Ling TW (eds) Conceptual Modeling – ER 2011, Lecture Notes in Computer Science, vol 6998, Springer, pp 346–361

Souza VES, Lapouchnian A, Robinson WN, Mylopoulos J (2011b) Awareness Requirements for Adaptive Systems. In: Proc. of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, ACM, pp 60–69

Souza VES, Lapouchnian A, Mylopoulos J (2012a) (Requirement) Evolution Requirements for Adaptive Systems. In: Proc. of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, IEEE, pp 155–164

Souza VES, Lapouchnian A, Mylopoulos J (2012b) Requirements-driven Qualitative Adaptation. In: Proc. of the 20th International Conference on Cooperative Information Systems (to appear), Springer

Wang Y, Mylopoulos J (2009) Self-Repair through Reconfiguration: A Requirements Engineering Approach. In: Proc. of the 2009 IEEE/ACM International Conference on Automated Software Engineering, IEEE, pp 257–268

Whittle J, Sawyer P, Bencomo N, Cheng BHC, Bruel JM (2009) RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In: Proc. of the 17th IEEE International Requirements Engineering Conference, IEEE, pp 79–88

Yu ESK, Giorgini P, Maiden N, Mylopoulos J (2011) Social Modeling for Requirements Engineering, 1st edn. MIT Press

Zowghi D, Offen R (1997) A Logical Framework for Modeling and Reasoning about the Evolution of Requirements. In: Proc. of the 3rd IEEE International Symposium on Requirements Engineering, IEEE, pp 247–257