(Requirement) Evolution Requirements for Adaptive Systems

Vítor E. Silva Souza, Alexei Lapouchnian, John Mylopoulos
Department of Information Engineering and Computer Science - University of Trento
Via Sommarive, 14 - Trento, Italy - 38123
{*vitorsouza,lapouchnian,jm*}*@disi.unitn.it*

***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**

I. Introduction

It is often the case that the requirements elicited from stakeholders for a system-to-be are not carved in stone, never to change during the system's lifetime. Rather, stakeholders will often hedge with statements such as "If requirement *R* fails more than *N* times in a week, relax it to *R-*", or "If we find that we are fulfilling our target (requirement *S*), let's strengthen *S* by replacing it with *S+*", or even "Requirement *Q* no longer applies after 20/01/2014". These are all requirements in the sense that they come from stakeholders and describe desirable properties of the system-to-be. They are special requirements, however, in the sense that their operationalization consists of changing other requirements, as suggested by the examples above.

A requirements model defines a space of system behaviors, where each behavior fulfills system objectives. When system adaptation is performed, the new behavior is selected from this space of alternatives. In this paper, however, we concentrate on requirements that change that space, thereby defining a changed set of system behaviors. Such evolutions allow the system to utilize new alternative behaviors. We call such requirements *Evolution Requirements* (a.k.a. *EvoReqs*) since they prescribe desired evolutions for other requirements. The objective of this paper is to circumscribe this family of requirements and offer mechanisms for modeling and operationalizing them. At runtime, *EvoReqs* have an effect on the running components of the system with the purpose of meeting stakeholder directives. *EvoReqs* allow us to not only specify what other requirements need to change, but also when other strategies — such as "retry after some time" or "relax the requirement" — should be used.

Our approach is goal-oriented in the sense that vanilla requirements are modeled as goals that can be refined and correlated, while *EvoReqs* are modeled as Event-Condition-Action (ECA) rules that are activated if an event occurs and a certain 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 according to stakeholder wishes). Each operation effects 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. Furthermore, such operations can be combined using patterns in order to compose macro-level evolution strategies, such as *Retry* and *Relax*.

For our goal models, we have adopted ideas from Control Theory, notably the concepts of control variables and indicators (monitored variables). Such variables are associated to individual goals in a goal model. An indicator measures the degree of fulfillment of the goal it is associated with, e.g., the percentage of dispatches that arrived at their destination within 15 minutes. Control variables, on the other hand, measure the quantity of a resource that is available to the running system, e.g., the number of ambulance cars available for dispatching. The primitive actions that can modify a goal model can not only add/remove/change goals, but also change the values of control variables.

We validate our proposal with an experiment where an Adaptive Computer-aided Ambulance Dispatch (A-CAD) is designed using our approach and is then executed 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 [1] and some of the publications that analyzed the case (e.g., [2]). The A-CAD is also used as a running example throughout the paper.

The rest of the paper is organized as follows. Section II summarizes relevant earlier work and introduces the running example. Section III presents *EvoReqs* and discusses the use of reconfiguration as an adaptation strategy. In Section IV, we detail the run-time coordination process and our ECA-based implementation, while Section V focuses on the validation of the proposal. Section VI compares our proposal to related work in the areas of adaptive systems and software evolution. Finally, Section VII concludes.

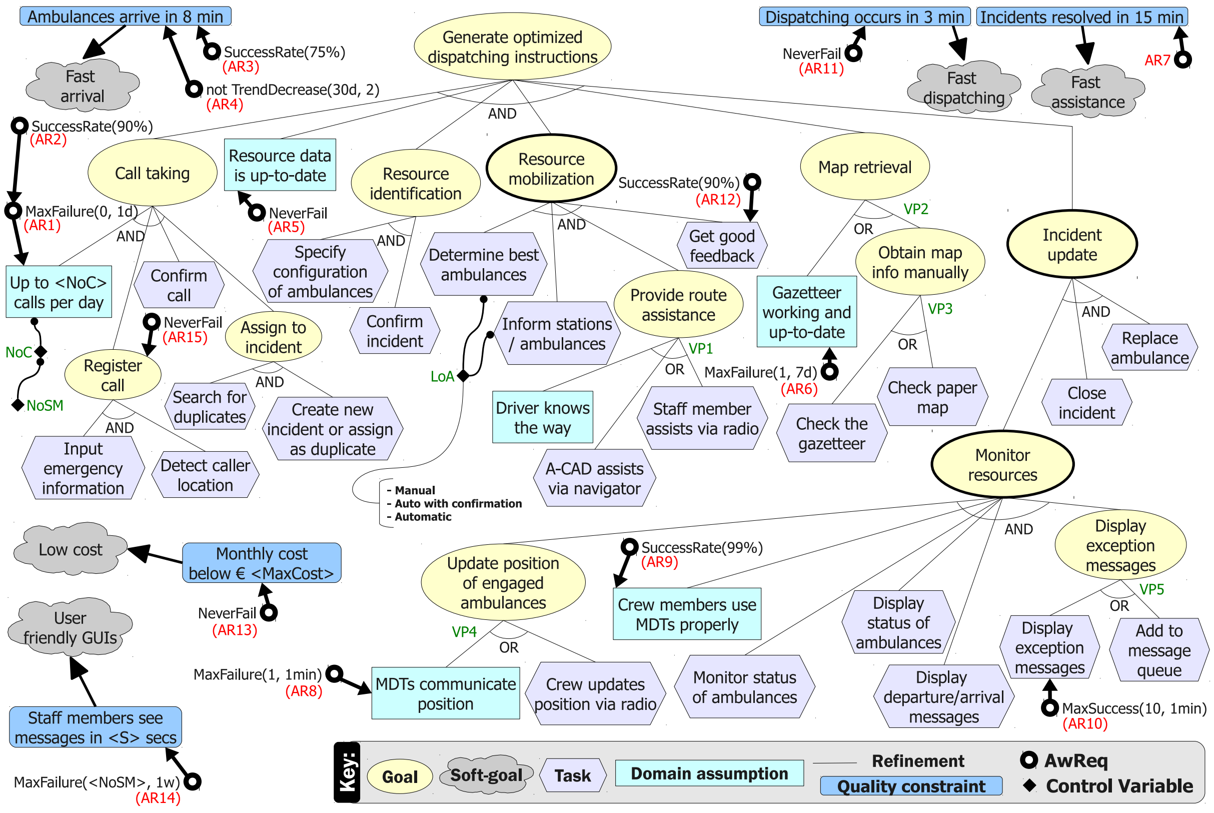

Figure 1. Goal model for the A-CAD after applying our systematic approach for the design of adaptive systems.

II. Baseline

We have previously proposed a systematic *System Identification* process [3] that starts with the elicitation of *Awareness Requirements* (*AwReqs*) [4], representing runtime indicators of requirements divergence. Our proposal in this paper is based on these works, which in turn builds on Goal-Oriented Requirements Engineering (GORE). This section summarizes this baseline through an example, an *Adaptive Computer-aided Ambulance Dispatch* (A-CAD), whose model is shown in Figure 1. For more details refer to [5].

A. The Initial Goal Model

Requirements are represented as *goals*, *tasks*, *softgoals*, *domain assumptions* (DAs) and *quality constraints* (QCs). These elements are part of the ontology proposed by Jureta et al. [6] for requirements and are supported by many requirements modeling approaches [7].

The space of alternatives for goal satisfaction is represented by Boolean AND/OR refinements with obvious semantics. Tasks, on the other hand, are directly mapped to functionality in the running system and are satisfied if executed successfully. Finally, DAs are satisfied if they hold (affirmed) while the system is pursuing its parent goal. In the example, the main goal of the system is to *Generate optimized dispatching instructions*. To be successful, the system is supposed to satisfy *Call taking*, *Resource identification*, *Resource mobilization*, *Map retrieval* and *Incident update*, all the while assuming that *Resource data is up-to-date*.

Softgoals represent (usually non-functional) requirements that do not have clear-cut criteria for satisfaction, such as *Low cost* or *Fast dispatching*. Before our approach is applied, however, their satisfaction should be metricized by one or more QCs, which can be associated with run-time procedures that check their satisfaction. Figure 1 contains five softgoals, each of which is associated with one QC.

B. System Identification

After the initial goal model has been built, we conduct a systematic *System Identification* process [3] for the adaptive system-to-be. As in control systems, this process identifies relevant indicators that can be monitored, system parameters that can be tuned at runtime and the relation between them, i.e., how changes in parameters affect indicators.

The first step of system identification consists of eliciting *Awareness Requirements* (*AwReqs*), which are requirements about the states of other requirements — such as their success or failure — at runtime [4]. *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).

For the A-CAD experiment, fifteen *AwReqs* were elicited and added to the goal model, most of them based on the problems associated with the LAS-CAD demise. For example, one of these problems was related to crew members not properly using Mobile Data Terminals (MDTs) installed in the ambulances. *AwReq* `AR9` indicates that the system should be aware of this problem at runtime in order to adapt, specifying the DA *Crew members use MDTs properly* should be true 99% of the times. Another example is *AwReq* `AR11`, which indicates QC *Dispatching occurs in 3 min* should never fail. This indicates that stakeholders consider *Fast dispatch* a critical requirement.

At runtime, the elements of the goal model are represented as classes, being instantiated every time a user starts pursuing a requirement (in the case of goals and tasks) or when they are bound to be verified (in the case of DAs and QCs). 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 `AR11`) or at the class (aggregate) level (e.g., 99% of the instances created in a specified period of time should be in the state *Satisfied*, like `AR9`). More details on this monitoring infrastructure can be found in [4].

The next steps in the process consist of identifying parameters that, when changed, have an effect on the relevant indicators, modeling the nature of this effect using differential relations. Figure 1 shows A-CAD's five *variation points* (`VP1–VP5`) and three *control variables* (`LoA`/*Level of Automation*, `NoC`/*Number of Calls* and `NoSM`/*Number of Staff Members*). For instance, $\Delta\left(AR11/LoA\right) > 0$ represents the relation between `LoA` and `AR11` in a qualitative way: the higher the level of automation, the faster the dispatch.

When *AwReqs* fail at runtime, a possible adaptation strategy is to change parameter values in order to improve the failing indicators. In a recently submitted paper [8], 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 are presented, as illustrated in Section I. We present this new family of requirements next.

III. Evolution Requirements

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

- If a staff member fails to *Register call* (`AR15`), she should **retry** after a few seconds;
- If there is a negative trend on the success rate of *Ambulances arrive in 8 min* for two consecutive months (`AR4`), we can tolerate this at most once a year, **relaxing** the constraint to three months in a row;
- If we are receiving more calls than we can handle (`AR1` or `AR2`), we do not want the A-CAD to autonomously hire staff members (i.e., increase `NoSM`). This task should be **delegated** to the management.

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* (c.f. §II-B) 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 (there is, e.g., reconfiguration [8]). 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 (III-A), then defining patterns to represent common adaptation strategies using these operations (III-B) and how this framework can accommodate reconfiguration as one possible strategy (III-C).

A. *EvoReq Operations*

Figure 2 shows a conceptual architecture for a run-time adaptation framework. The *Monitor* component has been proposed in [4] 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. 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 IV). 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 I.

Table I
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 [9].
`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.

As can be seen in the table, adaptation instructions have arguments which can refer to, among other things, system actors, requirements classes (upper-case `R`) or instances (lower-case `r`) and system parameters. Actors can be provided by any diagram that models external entities that interact with the system (e.g., *i** Strategic Dependency models [10]). Requirements classes/instances are provided by the monitoring component [4], which represents the elements of the requirements model as UML classes each extending the appropriate class from the diagram shown in Figure 3. Run-time instances of these elements (such as the various incident calls and ambulance dispatches) are then represented as objects that instantiate these classes. Finally, parameters are elicited during system identification, as explained in Section II-B. Instructions `apply-config` and `find-config` also refer to configurations and algorithms, which will be further explained in Section III-C.

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

```
g' = new-instance(G_RegCall);
copy-data(g, g');
terminate(g);
rollback(g);
wait(5s);
initiate(g');
```

Here, **`g`** represents an instance of goal *Register call*, referred to by the instance of *AwReq* `AR15` that failed. The framework then creates another instance of the goal, tells the target system to copy the data from the execution session of the failed goal to the one of the new goal, to terminate the failing components and rollback any partial changes made by them. After $5s$, the framework finally instructs the target system to initiate the new goal (i.e., it starts to pursue its child tasks), 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 I, for each requirement in the system's model. Obviously, as with any other requirement in a specification, each $\langle operation, requirement\rangle$ pair can be implemented on an as-needed basis.

Revisiting the previous example, `copy-data` should tell the A-CAD to copy the data related to the goal that failed (e.g., information on the emergency that has already been filled in the system) to a new user session, `terminate` closes the screen that was being used by the staff member

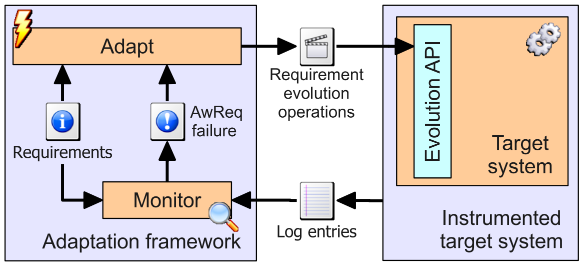

Figure 2. Conceptual architecture for a run-time adaptation framework.

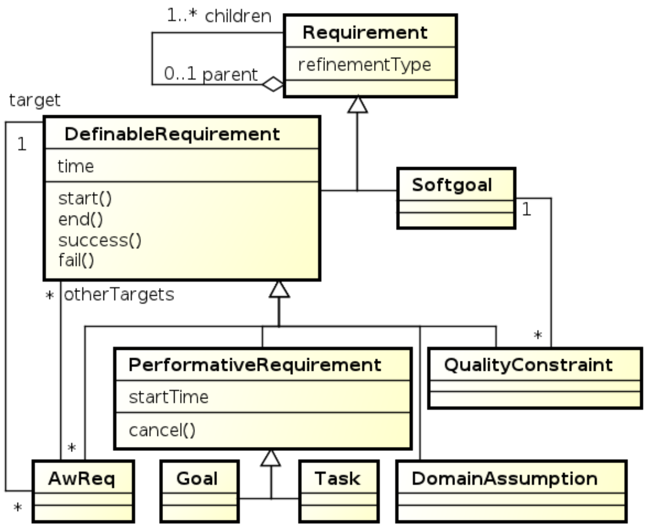

Figure 3. Class model for requirements in GORE, adapted from [4].

to register the call, `rollback` deletes any partial changes that might have been saved, `wait` shows a message asking the user to wait for $5s$ and, finally, `initiate` should open a new screen associated with the new user session so the staff member can try again. All this behavior is specific to the A-CAD and the task at hand and the way it will be implemented depends highly on the technologies chosen during its architectural design.

B. Adaptation Strategies as Patterns

The operations of Table I 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 that represent common adaptation strategies. Below we show the specification of the *Retry strategy*:

```
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');
}
```

Table II
ADAPTATION STRATEGIES ELICITED FOR THE A-CAD EXPERIMENT.

Strategy	Description
Delegate	Delegates the solution of the problem to an external agent (human or system) and waits for the problem to be fixed.
Warning	Like *Delegate*, warns an external agent about the problem, but does not wait for the problem to be fixed. Useful in situations in which the system cannot or does not need to be blocked while the problem is being fixed.
Relax	Can be applied to any kind of requirement, including *AwReqs*. Inspired by the RELAX framework [12] (c.f. §VI), this strategy relaxes the satisfaction condition of a requirement, either at the instance level (relax only the current achievement of the requirement) or at the class level (relax future executions of the requirement). There are two flavors for this strategy: replacing the requirement with a relaxed version of itself and, in case of AND-refined requirements, disabling one or more children to facilitate the satisfaction of the parent.
Strengthen	The analogous counterpart of *Relax*, strengthens the satisfaction condition of a requirement and is useful in combination with *Relax* to create trade-offs among requirements.
Abort	Last resort, aborts the execution of the system as gracefully as possible.

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. 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. Given this strategy, and assuming that time is represented in milliseconds, the example from Section III-A could be more concisely expressed as **`Retry(5000)`**.

After strategies have been elicited and represented as patterns, they can be associated with *AwReqs* and added to the requirements specification (e.g, **`Retry(5000)`** and `AR15`). Due to space constraints, we do not propose a specific syntax for their inclusion in the models.

Other than *Retry*, in our experiments we have identified and formalized the adaptation strategies summarized in Table II (refer to [5] for their specifications). This list is not intended to be exhaustive and new strategies can be created as needed. For instance, one could take inspiration from design patterns for adaptation [11]. Furthermore, it does not include *Reconfiguration*, which in an important strategy.

C. Reconfiguration

Wang & Mylopoulos [13] provide a GORE-based definition of a system *configuration*: "a set of tasks from a goal model which, when executed successfully in some order, lead to the satisfaction of the root goal". We add to this definition the values assigned to each *control variable* (CV) elicited during system identification (c.f. § II-B). *Reconfigu-*

ration, 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 another paper, which has been recently submitted to review [8]. 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. To cite a couple of examples, [13] proposes algorithms that suggest a new configuration without the component that has been diagnosed as responsible for the failure; whereas [14] assign preference rankings to softgoals and determine the best configuration using a SAT solver. A more thorough review of reconfiguration proposals is included in [8]. Note that different reconfiguration algorithms may require different information from the model. For instance, [13] requires a goal model and a diagnosis pointing to the failing component, whereas [14] needs the preference rankings of softgoals. Analysts should provide the required information accordingly.

IV. The Adaptation Process

Using the language described in Section III, 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.

```
 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  }
```

Figure 4. Algorithm for responding to *AwReq* failures.

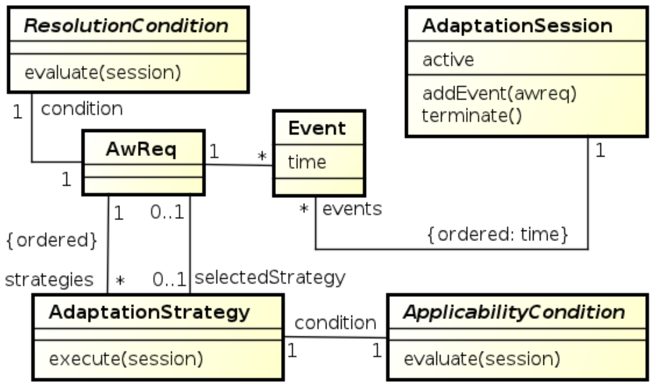

Figure 5. Entities involved in the ECA-based coordination process.

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 4, which manipulates instances of the classes represented in the class model of Figure 5.

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 III: say the monthly success rate of *Ambulances arrive in 8 min* has decreased twice in a row, causing the failure of `AR4` 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 1). As shown in Figure 5, an *adaptation session* consists on 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 `AR4` 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 `AR4` 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 A-CAD designers have associated two strategies to `AR4`. First, relax it by replacing `AR4` with `AR4'`, which verifies if the success rate has decreased not in two, but in three consecutive months (i.e., `not TrendDecrease(30d, 3)`). This strategy is associated with a condition that constraints its applicability to at most once a year. Second, the *Warning* strategy is also associated with `AR4`, sending a message to the ambulance service managers 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 `AR4` fails in the current year, it will be relaxed to `AR4'`, 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 `AR4` example, the session would remain active until another month has been passed and `AR4'` is checked. If the success rate increases then, `AR4'` will be satisfied, triggering another call to `processEvent()`, which would find `AR4`'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 month. Later, when we discuss the framework's implementation, this coordination process is depicted 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. As with *EvoReqs*, we do not propose any particular syntax for the inclusion of this information in the specification. 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.

V. Implementation and Evaluation

This section presents an evaluation of our proposal, based on Design Science methods [15]. In §V-A, we show that *EvoReqs* can be operationalized at runtime using the ECA-based process, bringing value to the target adaptive system. §V-B briefly discusses the performance of this solution.

A. Operationalizing EvoReqs

To demonstrate the value *EvoReqs* can bring to the development of adaptive systems, we have built the adaptation framework depicted earlier in Figure 2, together with a simulation component as the target system that mimics failure situations that could occur in the A-CAD.

The framework was implemented as OSGi bundles (Core, Logging, Monitoring, Adaptation and Simulation) and their source code is available for download (github.com/vitorsouza/Zanshin). The Core bundle exposes four service interfaces, 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*

```
<?xml version="1.0" encoding="UTF-8"?>
<acad:AcadGoalModel ...>
 <rootGoal xsi:type="acad:G_GenDispatch">
  <children xsi:type="acad:G_CallTaking">
   <children xsi:type="acad:D_MaxCalls"/>
   <children xsi:type="acad:G_RegCall">
    <children xsi:type="acad:T_InputInfo"/>
    <children xsi:type="acad:T_DetectLoc"/>
   </children>
   ...
  </children>
  ...
 </rootGoal>
 ...
 <awReqs xsi:type="acad:AR15" target="//@rootGoal/
     @children.0/@children.1">
  <condition xsi:type="model:SimpleResolutionCondition
      "/>
  <strategies xsi:type="model:RetryStrategy" time="
      5000">
   <condition xsi:type="
       model:MaxExecApplicabilityCondition"
       maxExecutions="1"/>
  </strategies>
  <strategies xsi:type="
      model:RelaxDisableChildStrategy" child="//
      @rootGoal/@children.0/@children.1/@children.1">
   <condition xsi:type="
       model:MaxExecApplicabilityCondition"
       maxExecutions="1"/>
  </strategies>
 </awReqs>
</acad:AcadGoalModel>
```

Figure 6. The A-CAD requirements specified as an EMF model.

```
AF: Processing state change: AR15 -> Failed
AF: (S1) Created new session for AR15
AF: (S1) The problem has not yet been solved...
AF: (S1) RetryStrategy is applicable.
AF: (S1) Selected: RetryStrategy
AF: (S1) Applying strategy RetryStrategy(true; 5000)
TS: Received: new-instance(G_RegCall)
TS: Received: copy-data(iG_RegCall, iG_RegCall)
TS: Received: terminate(iG_RegCall)
TS: Received: rollback(iG_RegCall)
TS: Received: wait(5000)
TS: Received: initiate(iG_RegCall)
AF: (S1) The problem has not yet been solved...
----------------------------------------------------
AF: Processing state change: AR15 -> Failed
AF: (S1) Retrieved existing session for AR15
AF: (S1) The problem has not yet been solved...
AF: (S1) RetryStrategy is not applicable
AF: (S1) RelaxDisableChildStrategy is applicable.
AF: (S1) Selected: RelaxDisableChildStrategy
AF: (S1) Applying strategy RelaxDisableChildStrategy(
     G_RegCall; Instance level only; T_DetectLoc)
TS: Received: suspend(iG_RegCall)
TS: Received: terminate(iT_DetectLoc)
TS: Received: rollback(iT_DetectLoc)
TS: Received: resume(iG_RegCall)
AF: (S1) The problem has not yet been solved...
----------------------------------------------------
AF: Processing state change: AR15 -> Succeeded
AF: (S1) Retrieved existing session for AR15
AF: (S1) The problem has been solved. Terminate S1.
```

Figure 7. Adaptation framework execution log for the `AR15` simulation.

instances, submitting these to the Adaptation Service. This component is further described in [4];

- *Adaptation Service*: implements the ECA-based coordination process described in Figure 4 (§IV), 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 I, 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 meta-models: the Core component provides the basic GORE classes (c.f., Figure 3) and the classes involved in the ECA-based process (c.f., Figure 5). These meta-models are extended by the Simulation bundle to provide classes representing the A-CAD requirements, i.e., one EMF class for each requirement of the goal model shown earlier in Figure 1, extending the appropriate GORE/ECA classes. Finally, the A-CAD requirements specification can be written as an EMF model, as shown in Figure 6.

This model excerpt shows the specification of goal *Register call*, its child tasks and ancestor goals, and *AwReq AR15*, which refers to *Register call* as its target using EMF's syntax for references within a model. *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: **`Retry(5000)`** (mentioned back in Section III-B) and **`RelaxDisableChild(T_DetectLoc)`** (which relaxes the satisfaction of the goal by disabling task *Detect caller location*). Both strategies are applicable at most once during an adaptation session, as can be seen in the specification.

When the simulation is ran, the A-CAD specification is read and represented in memory as Java^{TM} objects (using EMF's API). Once the framework detects *AR15* has changed its state (again, details in [4]), it conducts the ECA-based coordination process, producing a log similar to the one shown in Figure 7. 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.

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.

B. Performance

To evaluate the performance of the implementation, we have developed a simulation in which goal models of different sizes (100–1000 elements) are built and have an *AwReqs* 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.

Both the target system and the adaptation framework threads were timed and results showed that the impact of the implementation on the former is minimal, whereas the latter scales linearly with the size of the goal model. Space constraints prevent us from presenting more complete results in this paper, but 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.

VI. Related Work

Our work relates both to the Adaptive Systems and Software Evolution areas. Here, however, we restrict ourselves to approaches that, like ours, are focused on requirements.

Our approach is quite similar to FLAGS [16]. 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 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 [12] is similar to FLAGS, although it does not provide a runtime framework that operationalizes adaptation.

Another similar work is proposed by Fu et al. [17]. 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 & Perini [18] focus on service-based applications and adapt by searching for new services at runtime. Brown et al. [19] encapsulates Adapt operator extended LTL in specifications, which allows the system to switch between operational domains. Approaches that perform adaptation by reconfiguration, such as the ones cited in Section III-C, also fall into this category. Our work, on the other hand, proposes to adapt by changing the requirements (problem) space instead.

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" [20]) 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. [21] 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 [22]. 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 [23], 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 in our approach can be carefully predicted. Another important aspect of requirement evolution is the completeness and consistency of requirements models. E.g., to address this, [24] 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.

VII. Conclusions and Future Work

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, e.g., requirements failures.

Finally, it is important to consider the limitations of our proposal. ECA rulesets have a number of inherent problems, that are the subject of active research. One is the lack of confluence (i.e., invoking the same set of rules in different order will produce different results), while another is the possibility that conflicting rules (e.g., those assigning the same non-sharable resource to two different processes) may fire at the same time, thus requiring conflict resolution (e.g., using rule priorities or utility functions). Such conflicts may manifest themselves only at runtime, thereby complicating consistency checking among ECA rules.

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. Also, 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, among others.

Acknowledgment

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

References

[1] *Report of the inquiry into the London Ambulance Service*. South West Thames Regional Health Authority, 1993.

[2] J. Kramer and A. L. Wolf, "Succeedings of the 8th International Workshop on Software Specification and Design," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 5, pp. 21–35, Sep. 1996.

[3] V. E. S. Souza, A. Lapouchnian, and J. Mylopoulos, "System Identification for Adaptive Software Systems: A Requirements Engineering Perspective," in *Proc. ER 2011*. Springer, 2011, pp. 346–361.

[4] V. E. S. Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos, "Awareness Requirements for Adaptive Systems," in *Proc. SEAMS '11*. ACM, 2011, pp. 60–69.

[5] V. E. S. Souza, "An Experiment on the Design of an Adaptive System based on the LAS-CAD," University of Trento, Italy, Tech. Rep. http://disi.unitn.it/∼vitorsouza/a-cad/, 2012.

[6] I. Jureta, J. Mylopoulos, and S. Faulkner, "Revisiting the Core Ontology and Problem in Requirements Engineering," in *Proc. RE 2008*. IEEE, 2008, pp. 71–80.

[7] A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour," in *Proc. RE 2001*. IEEE, 2001, pp. 249–262.

[8] V. E. S. Souza, A. Lapouchnian, and J. Mylopoulos, "Requirements-driven Qualitative Adaptation," in *submitted for publication (under review)*, 2012.

[9] H. Garcia-Molina and K. Salem, "Sagas," *ACM SIGMOD Record*, vol. 16, no. 3, pp. 249–259, 1987.

[10] E. S. K. Yu, P. Giorgini, N. Maiden, and J. Mylopoulos, *Social Modeling for Requirements Engineering*. MIT Press, 2010.

[11] A. J. Ramirez and B. H. C. Cheng, "Design Patterns for Developing Dynamically Adaptive Systems," in *Proc. SEAMS 2010*. ACM, 2010, pp. 49–58.

[12] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.-M. Bruel, "RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems," in *Proc. RE 2009*. IEEE, 2009, pp. 79–88.

[13] Y. Wang and J. Mylopoulos, "Self-repair Through Reconfiguration: A Requirements Engineering Approach," in *Proc. ASE 2009*, 2009.

[14] X. Peng, B. Chen, Y. Yu, and W. Zhao, *Self-Tuning of Software Systems through Goal-based Feedback Loop Control*. IEEE, 2010.

[15] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design Science in Information Systems Research," *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004.

[16] L. Baresi and L. Pasquale, "Adaptive Goals for Self-Adaptive Service Compositions," in *Web Services (ICWS), 2010 IEEE International Conference on*, july 2010, pp. 353–360.

[17] L. Fu, X. Peng, Y. Yu, and W. Zhao, "Stateful Requirements Monitoring for Self-Repairing of Software Systems," Fudan University, China, Tech. Rep. FDSE-TR201101, available at http://www.se.fudan.sh.cn/paper/techreport/1.pdf, 2010.

[18] N. A. Qureshi and A. Perini, "Requirements Engineering for Adaptive Service Based Applications," in *Proc. RE 2010*. IEEE, 2010, pp. 108–111.

[19] G. Brown, B. H. C. Cheng, H. Goldsby, and J. Zhang, "Goal-oriented specification of adaptation requirements engineering in adaptive systems," in *Proc. SEAMS 2006*, 2006, pp. 23–29.

[20] A. I. Antón and C. Potts, "Functional paleontology: system evolution as the user sees it," in *Proc. ICSE 2001*. IEEE, 2001, pp. 421–430.

[21] S. Harker, K. Eason, and J. Dobson, "The change and evolution of requirements as a challenge to the practice of software engineering," in *Proc. RE 1993*, 1993, pp. 266–272.

[22] N. Ernst, A. Borgida, and I. Jureta, "Finding Incremental Solutions for Evolving Requirements," in *Proc. RE 2011*, 2011, pp. 15–24.

[23] W. Lam and M. Loomes, "Requirements Evolution in the Midst of Environmental Change: A Managed Approach," in *Proc. Euromicro 1998*. IEEE, 1998, pp. 121–127.

[24] D. Zowghi and R. Offen, "A Logical Framework for Modeling and Reasoning About the Evolution of Requirements," in *Proc. RE 1997*. IEEE, 1997, pp. 247–257.