Awareness Requirements*

Vítor E. Silva Souza[1], Alexei Lapouchnian[1],
William N. Robinson[2], and John Mylopoulos[1]

[1] Department of Inf. Engineering and Computer Science, University of Trento, Italy
{vitorsouza,lapouchnian,jm}@disi.unitn.it
[2] Department of Computer Information Systems, Georgia State University, USA
wrobinson@gsu.edu

Abstract. The functional specification of any software system operationalizes stakeholder requirements. In this paper we focus on a class of requirements that lead to feedback loop operationalizations. These *Awareness Requirements* talk about the runtime success/failure of other requirements and domain assumptions. Our proposal includes a language for expressing awareness requirements, as well as techniques for elicitation and implementation based on the EEAT requirements monitoring framework.

1 Introduction

There is much and growing interest in software systems that can adapt to changes in their environment or their requirements in order to continue to fulfill their mandate. Such adaptive systems usually consist of a system proper that delivers a required functionality, along with a monitor-analyze-plan-execute (MAPE [18]) feedback loop that operationalizes the system's adaptability mechanisms. Indications for this growing interest can be found in recent workshops and conferences on topics such as adaptive, autonomic and autonomous software (e.g., [7,23,14]).

We are interested in studying the *requirements* that lead to this feedback loop functionality. In other words, if feedback loops constitute an (architectural) solution, what is the requirements problem this solution is intended to solve? The nucleus of an answer to this question can be gleamed from any description of feedback loops: "... the objective ... is to make some output, say `y`, behave in a desired way by manipulating some input, say `u` ..." [10]. Suppose then that we have a requirement `r` = "supply customer with goods upon request" and let `s` be a system operationalizing `r`. The "desired way" of the above quote for `s` is that it *always* fulfills `r`, i.e., every time there is a customer request the system meets it successfully (here, the notion of "success" depends on the type of system: for software systems, it means completing the transaction without errors or exceptions, whereas for socio-technical systems "success" could involve the participation of human actors, e.g., goods are properly delivered to the customer). This means

* This is an extended version of the paper titled "Awareness Requirements for Adaptive Systems" published in the proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '11), pages 60–69. ACM, 2011.

R. de Lemos et al. (Eds.): Self-Adaptive Systems, LNCS 7475, pp. 133–161, 2012.

that the system somehow manages to deliver its functionality under all circumstances (e.g., even when one of the requested items is not available). Such a requirement can be expressed, roughly, as `r1` = "Every instance of requirement `r` succeeds". And, of course, an obvious way to operationalize `r1` is to add to the architecture of `s` a feedback loop that monitors if system responses to requests are being met, and takes corrective action if they are not. We can generalize on this: we could require that `s` succeeds more than 95% of the time over any one-month period, or that the average time it takes to supply a customer over any one week period is no more than 2 days. The common thread in all these examples is that they define requirements about the run-time success/failure/quality-of-service of other requirements. We call these *self-awareness requirements*.

A related class of requirements is concerned with the truth / falsity of domain assumptions. For our example, we may have designed our customer supply system on the domain assumption `d` = "suppliers for items we distribute are always open". Accordingly, if supplier availability is an issue for our system, we may want to add yet another requirement `r2` = "`d` will not fail more than 2% of the time during any 1-month period". This is also an awareness requirement, but it is concerned with the truth/falsity of a domain assumption.

The objective of this paper is to study Awareness Requirements (hereafter referred to as *AwReqs*), which are characterized syntactically as requirements that refer to other requirements or domain assumptions and their success or failure at runtime. *AwReqs* are represented in an existing language and can be directly monitored by a requirements monitoring framework. Although the technical contribution of this paper is focused on the definition and study of *AwReqs* and their monitoring at runtime, we do provide a discussion on how to go from *AwReqs* to adaptive systems, giving an overview of subsequent steps in this process.

Awareness is a topic of great importance within both Computer and Cognitive Sciences. In Philosophy, awareness plays an important role in several theories of consciousness. In fact, the distinction between self-awareness and contextual requirements seems to correspond to the distinction some theorists draw between higher-order awareness (the awareness we have of our own mental states) and first-order awareness (the awareness we have of the environment) [29]. In Psychology, consciousness has been studied as "self-referential behavior". Closer to home, awareness is a major design issue in Human-Computer Interaction (HCI) and Computer-Supported Cooperative Work (CSCW). The concept in various forms is also of interest in the design of software systems (security / process / context / location / ... awareness).

As part of our proposal's evaluation, which we detail in section 5, we have analyzed, designed and developed a simulation of a real-world system: an Ambulance Dispatch System (ADS), whose requirements have been documented by students of the University of Texas at Dallas [28]. We will use this application as running example throughout this paper.

The rest of the paper is structured as follows. Section 2 presents the research baseline; section 3 introduces *AwReqs* and talks about their elicitation; section 4 discusses their specification; section 5 talks about *AwReqs* monitoring

implementation and presents evaluation results from experiments with our proposal; section 6 summarizes related work; section 7 discusses the role of *AwReqs* in a systematic process for the development of adaptive systems based on feedback loops; finally, section 8 concludes the paper.

2 Baseline

This section introduces background research used in subsequent sections of this paper: Goal-Oriented Requirements Engineering (§2.1), feedback loops (§2.2) and requirements monitoring (§2.3).

2.1 Goal-Oriented Requirements Engineering

Our proposal is based on Goal-oriented Requirements Engineering (GORE). GORE is founded on the premise that requirements are stakeholder *goals* to be fulfilled by the system-to-be along with other actors. Goals are elicited from stakeholders and are analyzed by asking "why" and "how" questions [8]. Such analysis leads to goal models which are partially ordered graphs with stakeholder requirements as roots and more refined goals lower down. Our version of goal models is based loosely on $i^\star$ strategic rationale models [37]. Figure 1 shows a goal model for an Ambulance Dispatch System (ADS).

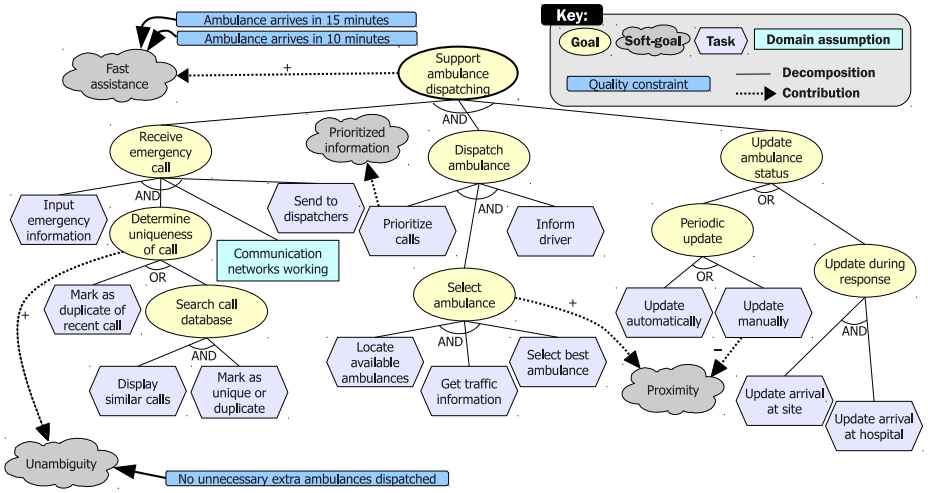

Fig. 1. Example goal model for an Ambulance Dispatch System

In our example, the main goal of the system is to support ambulance dispatching. Goals can be AND/OR refined. An AND-refinement means that in order to accomplish the parent goal, all sub-goals must be satisfied, while for an OR-refinement, only one of the sub-goals has to be attained. For example, to receive

an emergency call, one has to input its information, determine its uniqueness (have there been other calls for the same emergency?) and send it to dispatchers, all on the assumption that "Communication networks [are] working"[1]. On the other hand, periodic update of an ambulance's status can be performed either automatically or manually.

Goals are refined until they reach a level of granularity where there are tasks an actor (human or system) can perform to fulfill them. In the figure, goals are represented as ovals and tasks as hexagons. Note that we represent AND/OR *refinement* relations, avoiding the term *decomposition* as it usually carries a part-whole semantic which would constrain its use among elements of the same kind[2] (i.e., goal to goal, task to task, etc.). A refinement relation, on the other hand, can be applied between a goal and a task or a goal and a domain assumption and indicate how to satisfy the parent element: the goal is satisfied if all (AND) or any (OR) of its children are satisfied. In their turns, tasks are satisfied if they are executed successfully and domain assumptions are satisfied if they hold (the affirmation is true) while the user is pursuing its parent goal.

Softgoals are special types of goals that do not have clear-cut satisfaction criteria. In our example, stakeholders would like ambulance dispatching to be fast, dispatched calls to be unambiguous and prioritized, and selected ambulances to be as close as possible to the emergency site. Softgoal satisfaction can be estimated through qualitative contribution links that propagate satisfaction or denial and have four levels of contribution: break (- -), hurt (-), help (+) and make (++). E.g., selecting an ambulance using the software system contributes positively to the proximity of the ambulance to the emergency site, while using manual ambulance status update, instead of automatic, contributes negatively to the same criterion. Contributions may exist between any two goals (including hard goals).

Softgoals are obvious starting points for modeling non-functional requirements. To make use of them in design, however, they need to be refined to measurable constraints on the system-to-be. These are quality constraints (QCs), which are perceivable and measurable entities that inhere in other entities [17]. In our example, unambiguity is measured by the number of times two ambulances are dispatched to the same location, while fast assistance is refined into two QCs: ambulances arriving within 10 or 15 minutes to the emergency site.

Finally, domain assumptions (DAs) indicate states of the world that we assume to be true in order for the system to work. For example, we assume that communication networks (telephone, Internet, etc.) are available and functional. If this assumption were to be false, its parent goal ("Receive emergency call") would not be satisfied.

[1] These requirements are for illustrative purposes and, thus, are quite simple. Real-world systems would probably have multiple domain assumptions, one for each level of communication service, or even have assumptions parameterized by control variables that can be tuned at runtime — see §7.1 for a discussion on control variables.

[2] One could argue that it makes no sense to consider a task or a domain assumption a part of a goal. In effect, we have received such criticism in the past, in more than one occasion.

2.2 Feedback Loops

The recent growth of software systems in size and complexity made it increasingly infeasible to maintain them manually. This led to the development of a new class of self-adaptive systems, which are capable of changing their behavior at runtime due to failures as well as in response to changes in themselves, their environment, or their requirements. While attempts at adaptive systems have been made in various areas of computing, Brun et al. [6] argue for systematic software engineering approaches for developing self-adaptive systems based on the ideas from control engineering [15] with focus on explicitly specified feedback loops. Feedback loops provide a generic mechanism for self-adaptation. To realize self-adaptive behavior, systems typically employ a number of feedback controllers, possibly organized into controller hierarchies.

The main idea of feedback control is to use measurements of a system's outputs to achieve externally specified goals [15]. The objective of a feedback loop is usually to maintain properties of the system's output at or close to its reference input. The measured output of the system is evaluated against the reference input and the control error is produced. Based on the control error, the controller decides how to adjust the system's control input (parameters that affect the system) to bring its output to the desired value. To do that, the controller needs to possess a model of the system. In addition, a disturbance may influence the way control input affects output. Sensor noise may be present as well. This view of feedback loops does not concentrate on the activities within the controller itself. That is the emphasis of another model of a feedback loop, often called the autonomic control loop [9]. It focuses on the activities that realize feedback: monitoring, analysis, plan, execution — MAPE [18].

The common control objectives of feedback loops are regulatory control (making sure that the output is equal or near the reference input), disturbance rejection (ensuring that disturbances do not significantly affect the output), constrained optimization (obtaining the "best" value for the measured output) [15]. Control theory is concerned with developing control systems with properties such as stability (bounded input produces bounded output), accuracy (the output converges to the reference input), etc. While most of these guidelines are best suited for physical systems, many can be used for feedback control of software systems.

Using the ADS as an example, a feedback loop would: (1) monitor particular indicators of the system which are of interest to the stakeholders — e.g., the time it takes for ambulances to arrive at the location of the incidents; (2) compare the monitored values of these indicators with reference values specified in the requirements — e.g., QCs in the ADS goal model indicate ambulances should arrive in 10 or 15 minutes; and (3) if the monitored values do not satisfy the requirements, do something to fix the problem — e.g., increase the number of ambulances, change their locations around the city, etc. In this paper we propose *Awareness Requirements* as indicators to be monitored by the feedback loop, whereas the other steps of the loop in the context of our research are briefly discussed in section 7. Our view of adaptive systems as control systems has also been featured in a recently published position paper [34].

2.3 Requirements Monitoring

Monitoring is the first step in MAPE feedback loops and, as will be characterized in section 3, since *AwReqs* refer to the success/failure of other requirements, we will need to monitor requirements at runtime.

Therefore, we have based the monitoring component of our implementation on the requirements monitoring framework EEAT[3], formerly known as ReqMon [24]. EEAT, an Event Engineering and Analysis Toolkit, provides a programming interface (API) that simplifies temporal event reasoning. It defines a language to specify goals and can be used to compile monitors from the goal specification and evaluate goal fulfillment at runtime.

EEAT's architecture is presented in more detail along with our implementation in section 5. In it, requirements can be specified in a variant of the Object Constraints Language (OCL), called OCL_{TM} — meaning OCL with Temporal Message logic [25]. OCL_{TM} extends OCL 2.0 [2] with:

- Flake's approach to messages [12]: replaces the confusing `^ message()`, `^^ message()` syntax with `sentMessage`/`s`, `receivedMessage`/`s` attributes in class `OclAny`;
- Standard temporal operators: ○ (`next`), • (`prior`), ◊ (`eventually`), ♦ (`previously`), □ (`always`), ■ (`constantly`), $\mathcal{W}$ (`always ... unless`), $\mathcal{U}$ (`always ... until`);
- The scopes defined by Dwyer et al. [11]: `globally`, `before`, `after`, `between` and `after ... until`. Using the scope operators simplifies property specification;
- Patterns, also in Dwyer et al. [11]: `universal`, `absence`, `existence`, `bounded existence`, `response`, `precedence`, `chained precedence` and `chained response`;
- Timeouts associated with scopes: e.g. `after(Q, P, '3h')` indicates that `P` should be satisfied within three hours of the satisfaction of `Q`.

Figure 2 shows an example of OCL_{TM} constraint on the ADS. The invariant `getsDispatched` determines that if a call receives the `confirmUnique` message, eventually an ambulance should get the message `dispatch` and both messages should refer to the same `callID` argument. Given an instrumented Java implementation of these objects and a program in which they exchange messages through method calls, EEAT is able to monitor and assert this invariant at runtime. In section 5, we describe in more detail how EEAT accomplishes this in the context of *AwReqs* monitoring.

Although in our proposal *AwReqs* can be expressed in any language that provides temporal constructs (e.g., LTL, CTL, etc.), examples of *AwReq* specifications in section 4 will be given using OCL_{TM}, which is also the language used for our proposal's validation, presented in section 5.

[3] `http://eeat.cis.gsu.edu:8080/`

```
context Call
  -- An ambulance is dispatched for each unique call received.
  def: uniqueCall: LTL::OclMessage = receivedMessage('confirmUnique')
  def: ambulanceDispatched: LTL::OclMessage  = receivedMessage('ads.Ambulance', 'dispatch')
  inv getsDispatched: after(eventually(uniqueCall <> null),
     eventually(ambulanceDispatched.argument('callID') = uniqueCall.argument('callID')))
```

Fig. 2. An example of OCL_{TM} constraint

3 Awareness Requirements

As we have mentioned in section 1, feedback loops can provide adaptivity for a given system by introducing activities such as monitoring, analysis (diagnosis), planning and execution (of compensations) to the system proper. We are interested in modeling the requirements that lead to this feedback loop functionality. In control system terms (see §2.2), the reference input in this case is the system fulfilling its mandate (its requirements). Feedback loops, then, need to measure the actual output and compare it to the reference input, in other words, verify if requirements are being satisfied or not.

Furthermore, Berry et al. [4] defined the *envelope of adaptability* as the limit to which a system can adapt itself: "since for the foreseeable future, software is not able to think and be truly intelligent and creative, the extent to which a [system] can adapt is limited by the extent to which the adaptation analyst can anticipate the domain changes to be detected and the adaptations to be performed."

In this context, to completely specify a system with adaptive characteristics, requirements for adaptation have to be included in the specifications. We propose a new kind of requirement, which we call Awareness Requirement, or *AwReq*, to fill this need. *AwReqs* promote feedback loops for adaptive systems to first-class citizens in Requirements Engineering.

In this section, we characterize *AwReqs* as requirements for feedback loops that implement adaptivity (§3.1); propose patterns to facilitate their elicitation, along with a way to represent them graphically in the goal model (§3.2); and discuss the elicitation of this new type of requirements (§3.3). We illustrate all of our ideas using our running example, the ADS (figure 1).

3.1 Characterization

AwReqs are requirements that talk about the run-time status of other requirements. Specifically, *AwReqs* talk about the states requirements can assume during their execution at runtime. Figure 3 shows these states which, in the context of our modeling framework, can be assumed by goals, tasks, DAs, QCs and *AwReqs* themselves. When an actor starts to pursue a requirement, its result is yet `Undecided`. Eventually, the requirement will either have `Succeeded`, or `Failed`. For goals and tasks, there is also a `Canceled` state.

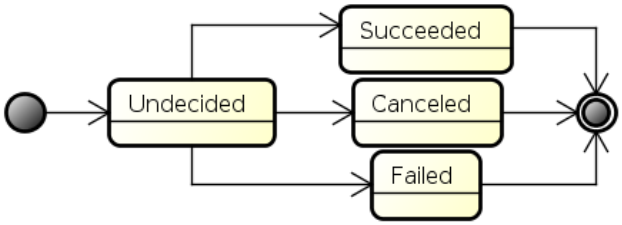

Fig. 3. States assumed by a requirement at runtime

Table 1. Examples of *AwReqs*, elicited in the context of the ADS

Id	Description	Type	Pattern
AR1	*Input emergency information* should never fail	–	`NeverFail(T-InputInfo)`
AR2	*Communications networks working* should have 99% success rate	Aggregate	`SuccessRate(D-CommNets Work, 99%)`
AR3	*Search call database* should have a 95% success rate over one week periods	Aggregate	`SuccessRate(G-Search CallDB, 95%, 7d)`
AR4	*Dispatch ambulance* should fail at most once a week	Aggregate	`MaxFailure(G-Dispatch Amb, 1, 7d)`
AR5	*Ambulance arrives in 10 minutes* should succeed 60% of the time, while *Ambulance arrives in 15 minutes* should succeed 80%, measured daily	Aggregate	`@daily SuccessRate( Q-Amb10min, 60%) and SuccessRate(Q-Amb15min, 80%)`
AR6	*Update automatically* should succeed 100 times more than the task *Update manually*	Aggregate	`ComparableSuccess( T-UpdAuto, T-UpdManual, 100)`
AR7	The success rate of *No unnecessary extra ambulances* for a month should not decrease, compared to the previous month, two times consecutively	Trend	`not TrendDecrease( Q-NoExtraAmb, 30d, 2)`
AR8	*Update arrival at site* should be successfully executed within 10 minutes of the successful execution of *Inform driver*, for the same emergency call	Delta	`ComparableDelta( T-UpdArrSite, T-InformDriver, time, 10m)`
AR9	*Mark as unique or duplicate* should be decided within 5 minutes	Delta	`StateDelta(T-MarkUnique, Undecided, *, 5m)`
AR10	`AR3` should have 75% success rate over one month periods	Meta	`SuccessRate(AR3, 75%, 30d)`
AR11	`AR5` should never fail	Meta	`NeverFail(AR5)`

Table 1 shows some of the *AwReqs* that were elicited during the analysis of the ADS. These examples illustrate the different types of *AwReqs*, which are discussed in the following paragraphs. Table 1 also indicates the pattern of each *AwReq* and we further elaborate on this matter on section 3.2.

The examples illustrate a number of types of *AwReq*. `AR1` shows the simplest form of *AwReq*: the requirement to which it refers should never fail. Considering a control system, the reference input is to fulfill the requirement. If the actual output is telling us the requirement has failed, the control system must act (compensate, reconcile — out of the scope of this proposal and briefly discussed in section 7) in order to bring the system back to an acceptable state. `AR1` considers every instance of the referred requirement. An instance of a task is created every time it is executed and the "never fail" constraint is to be checked for every such instance. Similarly, instances of a goal exist whenever the goal needs to be fulfilled, while DA and QC instances are created whenever their truth/falsity needs to be checked in the context of a goal fulfillment.

Inspired by the three modes of control of the proportional-integral-differential (PID) controller, a widely used feedback controller type [10], we propose three types of *AwReqs*: *Aggregate AwReqs* act like the integral component, which considers not only the current difference between the output and the reference input (the control error), but aggregates the errors of past measurements. *Delta AwReqs* were inspired by how proportional control sets its output proportional to the control error. *Trend AwReqs* follow the idea of the derivative control, which sets its output according to the rate of change of the control error. We define and exemplify each type of *AwReq* in the following.

An ***aggregate AwReq*** refers to the instances of another requirement and imposes constraints on their success/failure rate. E.g., `AR2` is the simplest aggregate *AwReq*: it demands that the referred DA be true 99% of the time the goal *Receive emergency call* is attempted. Aggregate *AwReqs* can also specify the period of time to consider when aggregating requirement instances (e.g., `AR3`). The frequency with which the requirement is to be verified is an optional parameter for *AwReqs*. If it is omitted, then the designer is to select the frequency (if the period of time to consider has been specified, it can be used as default value for the verification frequency). `AR5` is an example of an *AwReq* with verification interval specified.

Another pattern for aggregate *AwReq* specifies the min/max success/failure a requirement is allowed to have (e.g., `AR4`). *AwReqs* can combine different requirements, like `AR5`, that integrates two QCs with different target rates. One can even compare the success counts of two requirements (`AR6`). This captures a desired property of the alternative selection procedure when deciding at runtime how to fulfill a goal.

`AR7` is an example of a ***trend AwReq*** that compare success rates over a number of periods. Trend *AwReqs* can be used to spot problems in how success/failure rates evolve over time. ***Delta AwReqs***, on the other hand, can be used to specify acceptable thresholds for the fulfillment of requirements, such as achievement time. `AR8` specifies that task *Update arrival at site* should be satisfied (successfully

finish execution) within 10 minutes of completing task *Inform driver*. This means that once the dispatcher has informed the ambulance driver where the emergency is, she should arrive there within 10 minutes.

Another delta *AwReq*, `AR9`, shows how we can talk not only about success and failure of requirements, but about changes of states, following the state machine diagram of figure 3. In effect, when we say a requirement "should [not] succeed (fail)" we mean that it "should [not] transition from `Undecided` to `Succeeded` (`Failed`)". `AR9` illustrates yet another case: the task *Mark as unique or duplicate* should be decided — i.e., should leave the `Undecided` state — within 5 minutes. In other words, regardless if they succeeded or fail, operators should not spend more than 5 minutes deciding if a call is a duplicate of another call or not.

Finally, `AR10` and `AR11` are the examples of ***meta-AwReqs***: *AwReqs* that talk about other *AwReqs*. As we have previously discussed, *AwReqs* are based on the premise that even though we elicited, designed and implemented a system planning for all requirements to be satisfied, at runtime things might go wrong and requirements could fail, so *AwReqs* are added to trigger system adaptation in these cases. In this sense, *AwReqs* themselves are also requirements and, therefore, are also bound to fail at runtime. Thus, meta-*AwReqs* can provide further layers of adaptation in some cases if needed be.

One of the motivations for meta-*AwReqs* is the application of gradual reconciliation/compensations actions. This is the case with `AR10`: if `AR3` fails (i.e., *Search call database* has less than 95% success rate in a week), tagging the calls as "possibly ambiguous" (reconciling `AR3`) might be enough, but if `AR3`'s success rate considering the whole month is below 75% (e.g., it fails at least two out of four weeks), a deeper analysis of the database search problems might be in order (reconciling `AR10`). Another useful case for meta-*AwReqs* is to avoid executing specific reconciliation/compensation actions too many times. For example, `AR5` states that 60% of the ambulances should arrive in up to 10 minutes and 80% in up to 15 and to reconcile we should trigger messages to all users of the ADS. To avoid sending repeated messages in case it fails again, `AR11` states that `AR5` should never fail and, in case it does, its reconciliation decreases `AR5`'s percentages by 10 points (to 50% and 70%, respectively), which means that a new message will be sent only if the emergency response performance actually gets worse. If sending this message twice a month were to be avoided, `AR11`'s reconciliation could be, for example, disabling `AR5` for that month. As mentioned before, reconciliation is discussed in section 7.

With enough justification to do so, one could model an *AwReq* that refers to a meta-*AwReq*, which we would call a meta-meta-*AwReq* — or third-level *AwReq*. There is no limit on how many levels can be created, as long as meta-*AwReqs* from a given level refer strictly to *AwReqs* from lower levels, in order to avoid circular references. It is important to note that the name meta-*AwReq* is due only to the fact that it consists of an *AwReq* over another *AwReq*. This does not mean, however, that multiple levels of adaptation loops are required to monitor them. As will be presented in section 5, monitoring is operationalized by EEAT, which does so by matching method calls to invariants described in OCL_{TM} (an

example of this was presented in section 2.3), regardless of the class of the object that is receiving the message (goal, task, *AwReq*, meta-*AwReq*, etc.).

3.2 Patterns and Graphical Representation

Specifying *AwReqs* is not a trivial task. For this reason we propose *AwReq* patterns to facilitate their elicitation and analysis and a graphical representation that allows us to include them in the goal model, improving communication among system analysts and designers.

Many *AwReqs* have similar structure, such as "something must succeed so many times". By defining patterns for *AwReqs* we create a common vocabulary for analysts. Furthermore, patterns are used in the graphical representation of *AwReqs* in the goal model and code generation tools could be provided to automatically write the *AwReq* in the language of choice based on the pattern. In section 5.1, we provide OCL_{TM} idioms for this kind of code generation. We expect that the majority (if not all) *AwReqs* fall into these patterns, so their use can relieve requirements engineers from most of the specification effort.

Table 2 contains a list of patterns that we have identified so far in our research on this topic. This list is by no means exhaustive and each organization is free to define its own patterns (with their own names and meanings). We have already shown the pattern representation of the *AwReqs* that were elicited for the ADS in the last column of table 1. For such representation, we have used the patterns of table 2, mnemonics to refer to the requirements and abbreviated amounts of time

Table 2. A non-exhaustive list of *AwReq* patterns

Pattern	**Meaning**
`NeverFail(R)`	Requirement `R` should never fail. Analogous patterns `AlwaysSucceed`, `NeverCanceled`, etc.
`SuccessRate(R, r, t)`	`R` should have at least success rate `r` over time `t`.
`SuccessRateExecutions (R, r, n)`	`R` should have at least success rate `r` over the latest `n` executions.
`MaxFailure(R, x, t)`	`R` should fail at most `x` times over time `t`. Analogous patterns `MinFailure`, `MinSuccess` and `MaxSuccess`.
`ComparableSuccess(R, S, x, t)`	`R` should succeed at least `x` times more than `S` over time `t`.
`TrendDecrease(R, t, x)`	The success rate of `R` should not decrease `x` times consecutively considering periods of time specified by `t`. Analogous pattern `TrendIncrease`.
`ComparableDelta(R, S, p, x)`	The difference between the value of attribute `p` in requirements `R` and `S` should not be greater than `x`.
`StateDelta(R, s1, s2, t)`	`R` should transition from state `s1` to state `s2` in less time than what is specified in `t`.
P_1 `and` / `or` P_2; `not` P	Conjunction, disjunction and negation of patterns.

like in OCL_{TM} timeouts [25]. Furthermore, it is important to note that when requirements engineer create patterns, they are responsible for their consistency and correctness and, unfortunately, our approach does not provide any tool to help in this task.

Given that *AwReqs* can be shortened by a pattern we propose they be represented graphically in the goal model along with other elements such as goals, tasks, softgoals, DAs and QCs. For that purpose, we introduce the notation shown in figure 4, which shows the goal model of the ADS with the addition of *AwReqs*, represented graphically in the model. *AwReqs* are represented by thick circles with arrows pointing to the element to which they refer and the *AwReq* pattern besides it. The first parameter of the pattern is omitted, as the *AwReq* is pointing to it. In case an *AwReq* does not fit a pattern, the analyst should write its name and document its specification elsewhere.

3.3 Sources of Awareness Requirements

Like other types of requirements, *AwReqs* must be systematically elicited. Since they refer to the success/failure of other requirements, their elicitation takes place after the basic requirements have been elicited and the goal model constructed. There are several common sources of *AwReqs* and, in this section, we discuss some of these sources. We do not, however, propose a systematic process for *AwReq* elicitation and requirements engineers should use existing requirement elicitation techniques to discover requirements that belong to this new class.

One obvious source consists of the goals that are critical for the system-to-be to fulfill its purpose. If the aim is to create a robust and resilient system, then there have to be goals/tasks in the model that are to be achieved/executed at a consistently high level of success. Such a subset of critical goals can be identified in the process and *AwReqs* specifying the precise achievement rates that are required for these goals will be attached to them. This process can be viewed as the operationalization of high-level non-functional requirements (NFRs) such as Robustness, Dependability, etc. For example, the task *Input emergency information* is critical for this process since all subsequent activities depend on it. Also, government regulations and rules may require that certain goals cannot fail or be achieved at high rates. Similarly, *AwReqs* are applied to DAs that are critical for the system (e.g., *Communications networks working*).

As shown in section 3.1, *AwReqs* can be derived from softgoals. There, we presented a QC *Ambulance arrives in 10 minutes* that metricizes a high-level softgoal *Fast assistance*. Then, *AwReq* `AR5` is attached to it requiring the success rate of 60%. This way the system is able to quantitatively evaluate at runtime whether the quality requirements are met over large numbers of process instances and make appropriate adjustments if they are not.

Qualitative softgoal contribution labels in goal models capture how goals and tasks affect NFRs, which is helpful, e.g., for the selection of the most appropriate alternatives. In the absence of contribution links, *AwReqs* can be used to capture the fact that particular goals are important or even critical to meet NFRs and thus those goals' high rate of achievement is needed. This can be viewed as

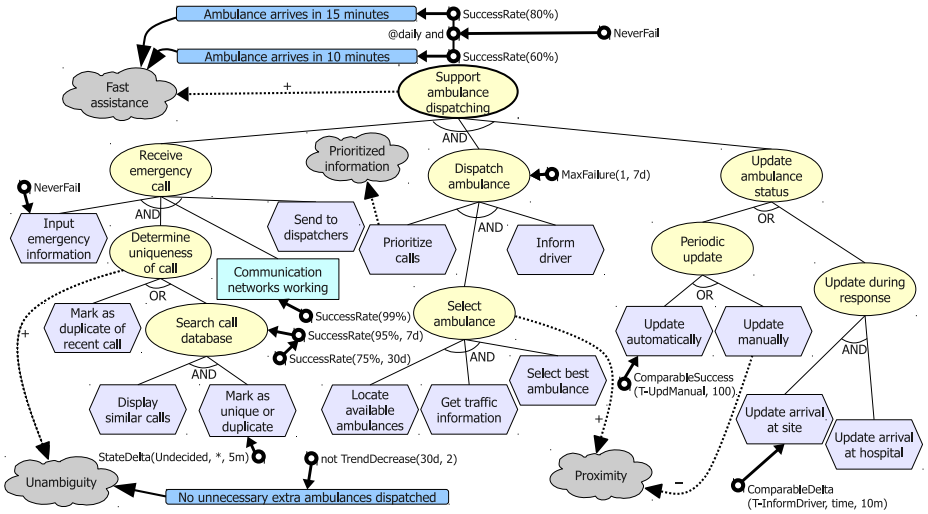

Fig. 4. Goal model of figure 1 with *AwReqs* represented graphically

an operationalization of a contribution link. For example, the task *Prioritize calls* in figure 1 positively affects the softgoal *Prioritized information* and can even be considered critical with respect to that softgoal. So, an *AwReq*, say, *SuccessRate(Prioritize Calls, 90%)*, can be added to the model to capture that fact. On the other hand, if a goal has a negative effect on an NFR, then an *AwReq* could demand a low success rate for it.

In Tropos [5] and other variations of goal modeling notation, alternatives introduced by OR-decomposed goals are frequently evaluated with respect to certain softgoals. The goal *Periodic updates* in figure 1 (of figure 4) is such an example. The evaluations are qualitative and show whether alternatives contribute positively or negatively to softgoals. In our approach, softgoals are refined into QCs and the qualitative contribution links are removed. However, the links do capture valuable information on the relative fitness of alternative ways to achieve goals. *AwReqs* can be used as a tool to make sure that "good" alternatives are still preferred over bad ones. For instance, the *AwReq* `AR6` states that automatic updates must be executed more often than manual ones, presumably because this is better for proximity of ambulances to target locations and due to the costs of manual updates. This way the intuition behind softgoal contribution links is preserved. If multiple conflicting softgoals play roles in the selection of alternatives, then a number of alternative *AwReqs* can be created since the selection of the best alternative will be different depending on the relative priorities of the conflicting NFRs.

One of the difficulties with *AwReqs* elicitation is coming up with precise specifications for the desired success rates over certain number of instances or during a certain time frame. To ease the elicitation and maintenance we recommend a gradual elicitation, first using high-level qualitative terms such as "medium"

or "high" success rate, "large" or "medium" number of instances, etc. Thus, the *AwReq* may originate as "high success rate of `G` over medium number of instances" before becoming *SuccessRate(G, 95%, 500)*. Of course, the quantification of these high-level terms is dependent on the domain and on the particular *AwReq*. So, "high success rate" may be mapped to 80% in one case and to 99.99% in another. Additionally, using abstract qualitative terms in the model while providing the mapping separately helps with the maintenance of the models since the model remains intact while only the mapping is changing.

4 Specifying Awareness Requirements

We have just introduced *AwReqs* as requirements that refer to the success or failure of other requirements. This means that the language for expressing *AwReqs* has to treat requirements as first class citizens that can be referred to. Moreover, the language has to be able to talk about the status of particular requirements instances at different time points. We have chosen to use an existing language, namely OCL_{TM}, over creating a new one, therefore inheriting its syntax and semantics. The subset of OCL_{TM} features available to requirements engineers when specifying *AwReqs* is the subset supported by the monitoring framework, EEAT, introduced in section 2.3. A formal definition of the syntax and the semantics of *AwReqs* is out of the scope of this paper.

Our general approach to using it is as follows: (i) design-time requirements — as shown in figure 1, but also the *AwReqs* of table 1 — are represented as UML classes, (ii) run-time instances of requirements, such as various ambulance dispatch requests, are represented as instances of these classes. Representing system requirements (previously modeled as a goal model) in a UML class diagram is a necessary step for the specification of *AwReqs* in any OCL-based language, as OCL constraints refer to classes and their instances, attributes and methods. Even though other UML diagrams (such as the sequence diagram or the activity diagram) might seem like a better choice for the representation of requirements and *AwReqs*, having instances of classes that represent requirements at runtime is mandatory for the OCL-based infrastructure that we have chosen.

Hence, we present in figure 5 a model that represents classes that should be extended to specify requirements. In other words, each requirement of our system should be represented by a UML class, extending the appropriate class from the diagram of figure 5. These classes have the same name as the mnemonics used in the pattern column of table 1. Moreover, the first letter of each class name indicates which element of figure 5 is being extended (`T` for `Task`, `G` for `Goal` and so forth). Note that the diagram of figure 5 does not represent a *meta-model* for requirements due to the fact that the classes that represent the system requirements are subclasses of the classes in this diagram, not instances of them as it is the case with meta-models. This inheritance is necessary in order for *AwReq* specifications to be able to refer to the methods defined in these classes, as they are inherited by the requirement classes.

Another important observation is that these classes are only an abstract representation of the elements of the goal model (figure 1) and they are part of the

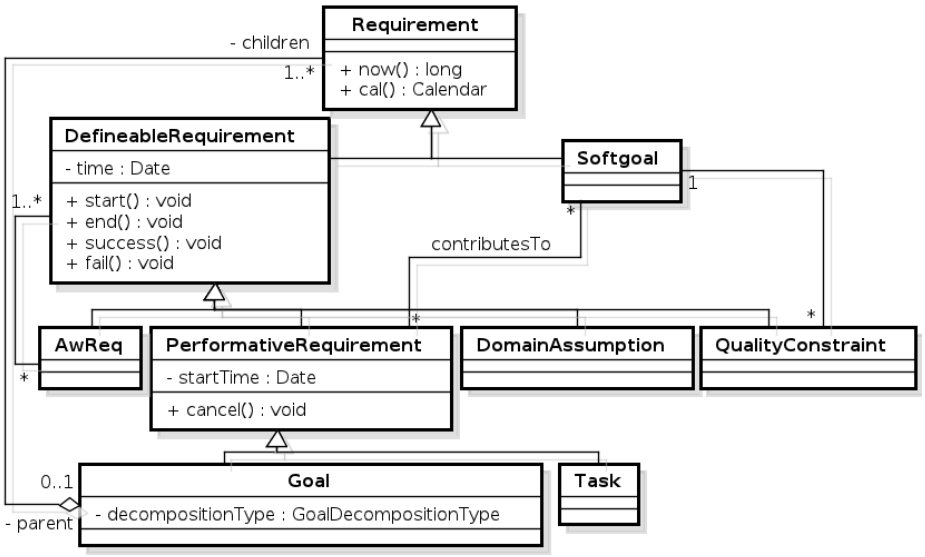

Fig. 5. Class model for requirements in GORE

monitoring framework that will be presented in section 5. They are not part of the monitored system (i.e., the ADS). In other words, the actual requirements of the system are not implemented by means of these classes.

Figure 6 shows the specification of some *AwReqs* of table 1 using OCL$_{TM}$. For example, consider `AR1`, which refers to a UML `Task` requirement. Figure 6 presents `AR1` as an OCL invariant on the class `T-InputInfo`, which should be a subclass of `Task` (from figure 5) and represents requirement *Input emergency information*. The invariant dictates that instances of `T-InputInfo` should never be in the `Failed` state, i.e., *Input emergency information* should never fail.

Aggregate *AwReqs* place constraints over a collection of instances. In `AR3`, for example, all instances of `G-SearchCallDB` executed in the past 7 days are retrieved in a set named `week` (using date comparison as in [25]), then we use the `select()` operation again to separate the subset of the instances that succeeded and, finally, we compare the sizes of these two sets in order to assert that 95% of the instances are successful at all times (`always`).

Trend *AwReqs* are similar, but a bit more complicated as we must separate the requirements instances into different time periods. For `AR7`, the `select()` operation was used to create sets with the instances of `Q-NoExtraAmb` for the past three months to compare the rate of success over time.

Delta *AwReqs* specify invariants over single instances of the requirements. `AR8` singles out the instances of `T-UpdAtSite` that are related to `T-InformDriver` in the `related` set by comparing the `callID` argument using OCL$_{TM}$'s `arguments()` operation [25]. Its invariant states that eventually the `related` set should have exactly one element, which should both be successful and finish its execution within 10 minutes of `T-InformDriver`'s end time.

```
context T-InputInfo inv AR1: never(self.oclInState(Failed))
context G-SearchCallDB
  def: week : G-SearchCallDB.allInstances()->select(g | new Date().diff(g.time, DAYS) <= 7)
  def: success : week->select(d | d.oclInState(Succeeded))
  inv AR3: always(success->size() / week->size() >= 0.95)
context Q-NoExtraAmb
  def: all : Set = Q-NoExtraAmb.allInstances()
  def: now : Date = new Date()
  def: m1 : Set = all->select(q | now.diff(q.time, DAYS) <= 30)
  def: m2 : Set = all->select(q | (now.diff(q.time, DAYS) <= 60) and (now.diff(q.time, DAYS > 30)))
  def: m3 : Set = all->select(q | (now.diff(q.time, DAYS) <= 90) and (now.diff(q.time, DAYS > 60)))
  def: success1 : Set = m1->select(q | q.oclInState(Succeeded))
  def: success2 : Set = m2->select(q | q.oclInState(Succeeded))
  def: success3 : Set = m3->select(q | q.oclInState(Succeeded))
  inv AR7: never(((success3->size() / m3->size()) < (success2->size() / m2->size())) and
    ((success2->size() / m2->size()) < (success1->size() / m1->size())))
context T-InformDriver
  def: related : Set = T-UpdAtSite.allInstances()->select(t | t.arguments('callID') = self.arguments('callID'))
  inv AR8: eventually(related->size() == 1) and always(related->forAll(t | t.oclInState(Succeeded) and
    t.time.diff(self.time, MINUTES) <= 10))
context T-MarkUnique
  inv AR9: eventually(not self.oclInState(Undecided)) and never(self.time.diff(self.startTime, MINUTES) > 5)
```

Fig. 6. Examples of *AwReqs* expressed in OCL_{TM}

`AR9` shows how to specify the example in which we do not talk specifically about success or failure of a requirement, but its change of state: eventually tasks `T-MarkUnique` should not be in the `Undecided` state and the difference between their start and end times should be at most 5 minutes.

5 Implementation and Evaluation

To evaluate our proposal we have implemented a framework to monitor *AwReqs* at runtime. Such evaluation considers three aspects of this framework:

1. Can *AwReqs* be monitored? Specifically, can an automated monitor evaluate requirements types enumerated in table 2 at runtime? Applying a constructive experiment, we show this is true (§5.1);
2. Can the *AwReqs* framework provide value for the analysis of a real system? With simulation experiments, we demonstrate this is true for scenarios of the ADS (§5.2);
3. What is the impact of *AwReqs* monitoring in the overall performance of the monitored system? We discuss this in §5.3.

The first two items above represent the experimental and descriptive evaluation methods of Design Science, as enumerated by [16]. After this initial evaluation, two other experiments were conducted, modeling the *AwReqs* of systems that

are close to real-world applications: an Adaptive Computer-aided Ambulance Dispatch system [31] that is somewhat similar to the ADS, but was based on the requirements for the London Ambulance System Computer-Aided Despatch (LAS-CAD) [1]; and an Automatic Teller Machine [35]. Since these experiments involved simulations of running systems based on their requirements models, future evaluation efforts include experiment with actual running systems and conducting full-fledged case studies with partners in industry.

5.1 Monitoring Awareness Requirements Patterns

As mentioned in section 2.3, we have used EEAT to monitor *AwReqs* expressed in OCL_{TM}. In its current version, EEAT compiles the OCL_{TM} expression into a rule file that is triggered by messages exchanged by objects at runtime (i.e., method calls). For this reason, we have to transform the initial specification of the *AwReqs* to one based on methods received by the run-time instances which represent the requirements. Figure 7 shows some of the *AwReqs* previously presented in figure 6 in their "EEAT specifications".

```
context T-InputInfo
    inv AR1: between(receivedMessage('start') <> null, receivedMessage('end') <> null, never(receivedMessage('fail') <> null))

context G-SearchCallDB
    def: weekA : LTL::OclMessage = receivedMessage('newWeek')
    def: weekB : LTL::OclMessage = receivedMessage('newWeek')
    def: wS : Integer = receivedMessages('success')->select(m | now() - m.timestamp() < week())->size()
    def: wF : Integer = receivedMessages('fail')->select(m | now() - m.timestamp() < week())->size()
    inv AR3: between(weekA <> null, weekB <> null and cal().weekDiff(weekA.timestamp(), weekB.timestamp()),
        always(wS / (wS + wF) >= 0.95)

context goalmodel::Task
    def: sTUpdSite : LTL::OclMessage = receivedMessage('T-UpdAtSite', 'success')
    def: sTInfDriv : LTL::OclMessage = receivedMessage('T-InformDriver', 'success')
    inv AR8: after(eventually(sTUpdSite <> null), eventually(sTUpdSite.argument('callID') = sTInfDriv.argument('callID')), '10m')
```

Fig. 7. Specification of *AwReqs* for EEAT

For monitoring to work, then, the source code of the monitored system (in this case, the ADS) has to be instrumented in order to create the instances of the classes that represent the requirements at runtime and call the methods defined in classes `DefinableRequirement` and `PerformativeRequirement` from figure 5. Methods `start()` and `end()` should be called when the system starts and ends the execution of a goal or task (or the evaluation of a QC or DA), respectively. Together with the `between` clause (one of Dwyer et al. scopes, see §2.3), these methods allow us to define the period in which *AwReqs* should be evaluated, because otherwise the rule system could wait indefinitely for a given message to arrive.

Given the right scope, the methods `success()`, `fail()` and `cancel()` are called by the monitored system to indicate a change of state in the requirement from `Undecided` to one of the corresponding final states (see figure 3). These methods are then used in the "EEAT specification" of *AwReqs*. For example, we

define `AR1` not as never being in the `Failed` state, but as never receiving the `fail()` message in the scope of a single execution (between `start()` and `end()`).

An aggregate requirement, on the other hand, aggregates the calls during the period of time defined in the *AwReq*. For `AR3`, this is done by monitoring for calls of the `newWeek()` method, which are called automatically by the monitoring framework at the beginning of every week. Similar methods for different time periods, such as `newDay()`, `newHour()` and so forth, should also be implemented.

The last example shows the delta *AwReq* `AR8`, which uses OCL$_{TM}$ timeouts to specify that the `success()` method should be called in the `T-InformDriver` instance within 10 minutes after the same method is called in `T-UpdAtSite`, given that both instances refer to the same call ID, an argument that can be passed along the method. This can be implemented by having a collection of key-value pairs passed as parameters to the methods `start()`, `success()`, etc.

An automatic translator from the *AwReqs*' initial specification to their "EEAT specification" could be built to aid the designer in this task. Another possibility is to go directly from the *AwReq* patterns presented in section 3.2 to this final specification. Table 3 illustrates how some of the patterns of table 1 can be expressed in OCL$_{TM}$. These formulations are consistent with those shown in figure 7. The definitions and invariants are placed in the context of UML classes that represent requirements (see §4). For example, a `receiveMessage('fail')` for context `R`, denotes the called operation `R.fail()` for class `R`. Therefore, the invariant `pR` in the first row of table 3 is true if `R.fail()` is never called.

Table 3. EEAT/OCL$_{TM}$ idioms for some patterns

Pattern	**OCL$_{TM}$ idiom**
NeverFail(R)	`def: rm: OclMessage = receiveMessage('fail')` `inv pR: never(rm)`

SuccessRate(R, r, t)	`def: msgs: Sequence(OclMessage) = receiveMessages()->`
	`  select(range().includes(timestamp()))`
	`-- Note: these definitions are patterns that are assumed in`
	`  the following definitions`
	`def: succeed: Integer = msgs->select(methodName = 'succeed'))->size()`
	`def: fail: Integer = msgs->select(methodName = 'fail'))->size()`
	`inv pR: always(succeed / (succeed + fail) > r)`

ComparableSuccess (R, S, x, t)	`-- c1 and c2 are fully specified class names` `inv pR: always(c1.succeed > c2.succeed * x)`
MaxFailure(R, x, t)	`inv pR: always(fail < x)`
P_1 and/or P_2; not P	`-- arbitrary temporal and real-time logical expressions are` `  allowed over requirements definitions and run-time objects`

Of course, the patterns of table 1 represent only common kinds of expressions. *AwReqs* contain the range of expressions where a requirement `R1` can express properties about requirement `R2`, which include both design-time and run-time requirements properties. OCL$_{TM}$ explicitly supports such references, as the following expressions illustrate:

```
def: p1: PropertyEvent = receivedProperty('p:package.class.invariant')
inv p2: never(p1.satisfied() = false)
```

In OCL_{TM}, all property evaluations are asserted into the run-time evaluation repository as `PropertyEvent` objects. The definition expression of `p1` refers to an invariant (on a UML class, in a UML package). Properties about `p1` include its run-time evaluation (`satisfied()`), as well as its design-time properties (e.g., `p1.name()`). Therefore, in OCL_{TM}, requirements can refer to their design-time and run-time properties and, thus, *AwReqs* can be represented in OCL_{TM}.

To determine if the *AwReq* patterns can be evaluated at runtime, we constructed scenarios for each row of table 3. Each scenario includes three alternatives, which should evaluate to true, false, and indeterminate (non-false) during requirements evaluation. We had EEAT compile the patterns and construct a monitor. Then, we ran the scenarios. In all cases, EEAT correctly evaluated the requirements.

To illustrate how EEAT evaluates OCL_{TM} requirements in general, the next subsection describes in detail a portion of the evaluation of the ADS' monitoring system, which was generated from the requirements of table 1.

5.2 Evaluating an Awareness Requirement Scenario

The requirements of the ADS provide a context to evaluate the *AwReq* framework. The ADS is implemented in Java. Its requirements (table 1) are represented as OCL_{TM} properties, using patterns like those presented in table 3 and figure 7. Scenarios were developed to exercise each requirement so that each of them should evaluate as failed or succeeded. When each scenario is run, EEAT evaluates the requirements and returns the correct value. Thus, all the scenarios that test ADS requirements presented here evaluate correctly.

Next, we describe how this process works for one requirement and one test. Consider a single vertical slice of the development surrounding requirement `AR1`, as shown in figure 8:

1. Analysts specify the *Emergency input information* task of figure 1 (i.e., `T-InputInfo`) as a task specification (e.g., input, output, processing algorithm) along with *AwReqs* such as `AR1`;
2. Developers produce an input form and a processor fulfilling the specification. In a workflow system architecture, `T-InputInfo` is implemented as a XML form which is processed by a workflow engine. In our standard Java application, `T-InputInfo` is implemented as a form that is saved to a database. In any case, the point at which the input form is processed is the instrumentation point;
3. Validators (i.e., people performing requirements monitoring) instrument the software. Five events are logged in this simple example: (a) `T-InputInfo.start()`, (b) `T-InputInfo.end()`, (c) `T-InputInfo.success()`, (d) `T-InputInfo.fail()`, and (e) `T-InputInfo.cancel()`. Of course, the developers may have chosen a different name for `T-InputInfo` or the five methods, in which case, the validator must introduce a mapping from the run-time object and methods to the requirements classes and operations. Given the

rise of domain-driven software development, in which requirements classes are implemented directly in code, the mapping function is often relatively simple — even one-to-one;

4. The EEAT monitor continually receives the instrumented events and determines the satisfaction of requirements. In the case of `AR1`, if the `T-InputInfo` form is processed as succeed or cancel, then `AR1` is true.

The architecture and process of EEAT provides some context for the preceding description. EEAT follows a model-driven architecture (MDA). It relies on the Eclipse Modeling Framework (EMF) for its meta-model and the OSGi component specifications. This means that the OCL_{TM} language and parser is defined as a variant of the Eclipse OCL parser by providing EMF definitions for operations, such as `receivedMessage`. The compiler generates Drools rules, which combined with the EEAT API, provide the processing to incrementally evaluate OCL_{TM} properties at runtime.

EEAT provides an Eclipse-based UI. However, the run-time operates as a OSGi application, comprised as a dynamic set of OSGi components. For these experiments, the EEAT run-time components consist of the OCL_{TM} property evaluator, compiled into a Drools rule system, and the EEAT log4j feed, which listens for logging events and adds them to the EEAT repository. The Java application was instrumented by Eclipse TPTP to send CBE events via log4j to EEAT, where the event are evaluated by the compiled OCL_{TM} property monitors. For a more complete description of the language and process of EEAT, see [26,27].

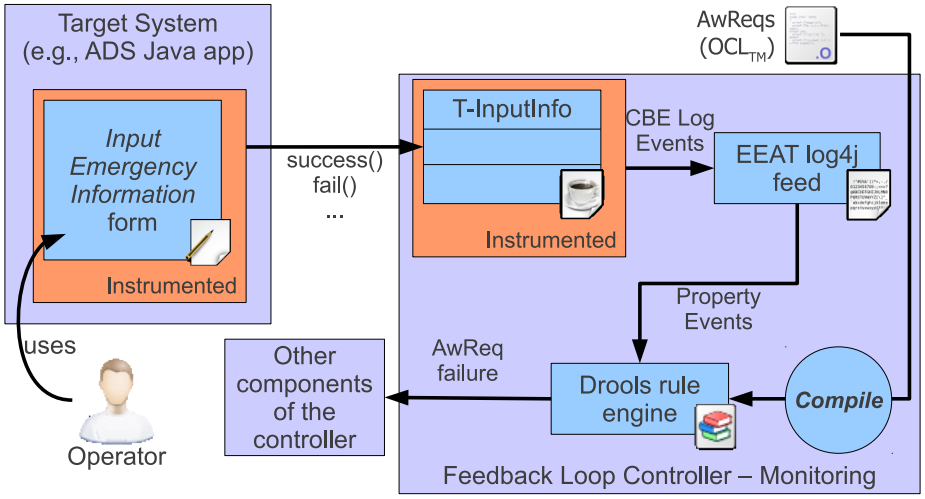

Fig. 8. Overview of the *AwReqs* monitoring framework

5.3 Monitor Performance

Monitoring has little impact on the target system, mostly because the target system and the monitor typically run on separate computers. The TPTP Probekit provides optimized byte-code instrumentation, which adds little overhead to some (selected) method calls in the target system. The logging of significant events consumes no more than 5%, and typically less than 1% overhead.

For real-time monitoring, it is important to determine if the target events can overwhelm the monitoring system. A performance analysis of EEAT was conducted by comparing the total monitoring runtime vs. without monitoring using 40 combinations of the Dwyer et al. temporal patterns [11]. For data, a simple two-event sequence was the basis of the test datum; for context, consider the events as an arriving email and its subsequent reply. These pairs were continuously sent to the server 10,000 times. In the experiment, the event generator and EEAT ran in the same multi-threaded process. The test ran as a JUnit test case within Eclipse on a Windows Server 2003 dual core 2.8 GHz with 1G memory. The results suggest that, within the test configuration, sequential properties (of length 2) are processed at 137 event-pairs per second [26]. This indicates that EEAT is reasonably efficient for many monitoring problems.

6 Related Work

In the literature, there are many approaches for the design of adaptive systems. A great deal of them, however, focus on architectural solutions for this problem, such as the Rainbow framework [13], the proposal of Kramer & Magee [19], the work of Sousa et al. [30], the SASSY framework [22], among others. These approaches usually express adaptation 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 early requirements (goal) models, allowing stakeholders and requirements engineers to reason about adaptation on a higher level of abstraction. Furthermore, *AwReqs* can be associated not only to non-functional characteristics of the system (represented by quality constraints), but also to functional requirements (goals, tasks) and even domain assumptions. The rest of this section focuses on recent approaches that share a common focus with ours in early requirements models.

A number of recent proposals offer alternative ways of expressing and reasoning about partial requirements satisfaction. RELAX by Whittle, et al. [36] is one such approach aimed at capturing uncertainty (mainly due to environmental factors) in the way requirements can be met. Unlike our goal-oriented approach, RELAX assumes that structured natural language requirements specifications (containing the SHALL statements that specify what the system ought to do) are available before their conversion to RELAX specifications. The modal operators available in RELAX, SHALL and MAY...OR, specify, respectively, that requirements must hold or that there exist requirements alternatives. We, on the other hand, capture alternative requirements refinement using OR decompositions of goals.

In RELAX, points of flexibility/uncertainty are specified declaratively, thus allowing designs based on rules, planning, etc. as well as to support unanticipated adaptations. Some requirements are deemed invariant — they need to be satisfied no matter what. This corresponds to the *NeverFail(R) AwReq* pattern in our approach. Other requirements are made more flexible in order to maintain their satisfaction by using "as possible"-type RELAX operators. Because of these, RELAX needs a logic with built-in uncertainty to capture its semantics. The authors chose fuzzy branching temporal logic for this purpose. It is based on the idea of fuzzy sets, which allows gradual membership functions. E.g., the function for fuzzy number 2 peaks at 1 given the value 2 and slopes sharply towards 0 as we move away from 2, thus capturing "approximately 2". Temporal operators such as *Eventually* and *Until* allow for temporal component in requirements specifications in RELAX.

Our approach is much simpler compared to RELAX. The *AwReqs* constructs that we provide just reference other requirements. Thus, we believe that it is more suitable, e.g., for requirements elicitation activities. Our specifications do not rely on fuzzy logic and do not require a complete requirements specification to be available prior to the introduction of *AwReqs*. Also, our language does not require complex temporal constructs. However, the underlying formalism used for *AwReqs* — OCL_{TM} — provides temporal operators, as does EEAT, so temporal properties can be expressed and monitored. Most of the work on generating OCL_{TM} specifications can be automated through the use of patterns.

With each relaxation RELAX associates "uncertainty factors": properties of the environment that can or cannot be monitored, but which affect uncertainty in achieving requirements. Our future work includes such integration of domain models in our approach.

Using *AwReqs* we can express approximations of many of the RELAX-ed requirements. For instance, `AR5` from table 1 can be used as a rough approximation of the requirement "ambulances must arrive at the scene AS CLOSE AS POSSIBLE to 10 minutes' time". The general pattern for approximating fuzzy requirements is to first identify a number of requirements that differ in their strictness, depending on our interpretation of what "approximately" means. E.g., `R1` = "ambulance arrives in 10 min", `R2` = "ambulance arrives in 12 min", `R3` = "ambulance arrives in 15 min". Then, we assign desired satisfaction levels to these requirements. For instance, we can set success rate for `R1` to 60% (as in `AR5`), `R2` to 80%, and `R3` to 100%. This means that all ambulances will have to arrive within 10–15 min from the emergency call. The *AwReq* will then look like `AR12` = *SuccessRate(R1, 60%) AND SuccessRate(R2, 80%) AND SuccessRate(R3, 100%).* On the other hand, `AR13` = *SuccessRate(R1, 80%) AND SuccessRate(R2, 100%)* provides a much stricter interpretation of the fuzzy duration with all ambulances required to arrive within 12 minutes.

Another related approach called FLAGS is presented in [3]. FLAGS requirements models are based on the KAOS framework [20] and are targeted at adaptive systems. It proposes crisp (Boolean) goals (specified in linear-time temporal logic, as in KAOS), whose satisfaction can be easily evaluated, and fuzzy goals

that are specified using fuzzy constraints. In FLAGS, fuzzy goals are mostly associated with non-functional requirements. The key difference between crisp and fuzzy goals is that the former are firm requirements, while the latter are more flexible. Compared to RELAX, FLAGS is a goal-oriented approach and thus is closer in spirit to our proposal.

To provide semantics for fuzzy goals, FLAGS includes fuzzy relational and temporal operators. These allow expressing requirements such as something be almost always less than `X`, equal to `X`, within around `t` instants of time, lasts hopefully `t` instants, etc. As was the case with the RELAX approach, *AwReqs* can approximate some of the fuzzy goals of FLAGS while remaining quite simple. The example that we presented while discussing RELAX also applies here. Whenever a fuzzy membership function is introduced in FLAGS, its shape must be defined by considering the preferences of stakeholders. This specifies exactly what values are considered to be "around" the desired value. As we have shown above with `AR12` and `AR13`, *AwReqs* can approximate this "tuning" of fuzzy functions while not needing fuzzy logic and thus remaining more accessible to stakeholders.

Additionally, in FLAGS, adaptive goals define countermeasures to be executed when goals are not attained, using event-condition-action rules. Using a similar approach, we have recently published a proposal to complement *AwReqs* with adaptation strategies that provide compensation for failures [33]. Discussion in section 3 illustrates how *AwReqs* and meta-*AwReqs* could be used to enact the required compensation behavior, including relaxation of desired success rates. We further comment on these aspects on section 7.2.

Letier and van Lamsweerde [21] present an approach that allows for specifying partial degrees of goal satisfaction for quantifying the impact of alternative designs on high-level system goals. Their partial degree of satisfaction can be the result of, e.g., failures, limited resources, etc. Unlike FLAGS and RELAX, here, a partial goal satisfaction is measured not in terms of its proximity to being fully satisfied, but in terms of the probability that it is satisfied. The approach augments KAOS with a probabilistic layer. Here, goal behavior specification (in the usual KAOS temporal logic way) is separate from the quantitative aspects of goal satisfaction (specified by quality variables and objective functions). Objective functions can be quite similar to *AwReqs*, except they use probabilities. For instance, one such function presented in [21] states that the probability of ambulance response time of less than 8 min should be 95%. Objective functions are formally specified using a probabilistic extension of temporal logic. An approach for propagating partial degrees of satisfaction through the model is also part of the method.

Overall, the method can be used to estimate the level of satisfaction of high-level goals given statistical data about the current or similar system (from rather low-level measurable parameters). Our approach, on the other hand, naturally leads to high-level monitoring capabilities that can determine satisfaction levels for *AwReqs*.

There is a fundamental difference between the approaches described above and our proposal. There, by default, goals are treated as invariants that must

always be achieved. Non-critical goals — those that can be violated from time to time — are relaxed. Then, the aim of those methods is to provide the machinery to conclude at runtime that while the system may have failed to fully achieve its relaxed goals, this is acceptable. So, while relaxed goals are monitored at runtime, invariant ones are analyzed at design time and must be guaranteed to always be achievable at runtime.

In our approach, on the other hand, we accept the fact that a system may fail in achieving any of its initial (stratum 0) requirements. We then suggest that critical requirements are supplemented by *AwReqs* that ultimately lead to the introduction of feedback loop functionality into the system to control the degree of violation of critical requirements. Thus, the feedback infrastructure is there to reinforce critical requirements and not to monitor the satisfaction of expendable (i.e., relaxed) goals, as in RELAX/FLAGS. The introduction of feedback loops in our approach is ultimately justified by criticality concerns.

7 From Awareness Requirements to Feedback Loops

As stated in section 1, our intention in this proposal is to identify and explore requirements that lead to the introduction of feedback loop functionality into adaptive systems. In section 3.3, we discussed the sources of *AwReqs*, while section 5 explained how EEAT can be used to monitor *AwReqs* at runtime to determine if they are attained or not. In this section, we present the overview of the role of Awareness Requirements in our overall approach for feedback loop-based requirements-driven adaptive systems design.

Figure 9 shows a variant of a feedback controller diagram adapted for requirements-driven adaptive systems. Here, system requirements play the role of the reference input, while indications of requirements convergence signaling if the

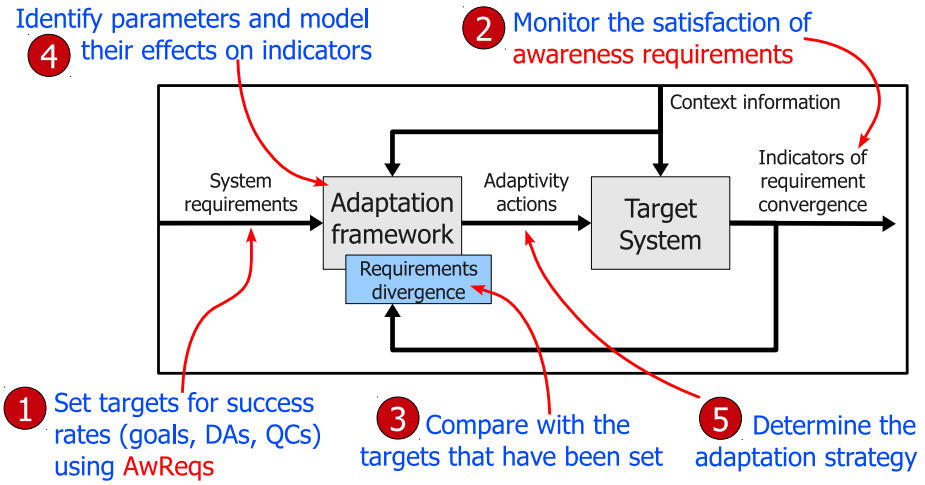

Fig. 9. A feedback loop illustrating the steps of the proposed process

requirements have been met replace the traditional monitored output of the controller. The controller itself is represented by a requirements-driven adaptation framework that controls the target system through executing adaptation actions that correspond to the control input in traditional feedback control schemes. Dynamically changing context corresponds to the disturbance input of the control loop. Finally, the measure of requirements divergence is the control error.

Furthermore, the phases of our proposed approach are added to the feedback loop diagram in figure 9, labeled 1 through 5. Step 1 is to set the targets for system to achieve/maintain at runtime. *AwReqs*, as discussed here, are used for this purpose. For step 2, the EEAT monitoring framework presented in section 5 is used to monitor whether the *AwReqs* are attained at runtime. Given the values for the *AwReq* attainment at runtime, in step 3 we calculate requirements divergence. If the targets are not met, this warrants a system adaptation. The system identification process (step 4) is aimed at linking system configuration parameters with indicators of requirements convergence and can be used to determine possible system reconfigurations. This process is further discussed in section 7.1. Finally, adaptation strategies/actions (step 5 in figure 9) are used by the adaptation framework to actually adapt the target system. These are further discussed in section 7.2.

7.1 System Identification

As we have shown throughout sections 3 to 5, *AwReqs* can be used to determine when requirements are not being satisfied, much the same way a control system calculates the control error, i.e., the discrepancy between the reference input (desired) and the measured output (outcome). The next step, then, is to determine the control input based on this discrepancy, i.e., determine what could be done to adapt the target system to ultimately satisfy the requirements.

In Control Theory (e.g., [15]), the first step towards accomplishing this is an activity called *System Identification*, which is the process of determining the equations that govern the dynamic behavior of a system. This activity is concerned with: (a) the identification of system parameters that, when manipulated, have an effect on the measured output; and (b) the understanding of the nature of this effect. Afterwards, these equations can guide the choice of the best way to adapt to different circumstances. For example, in a control system in which the room temperature is the measured output, turning on the air conditioner lowers the temperature, whereas using the furnace raises it. If the heating/cooling systems offer different levels of power, there is also a relation between such power level and the rate in which the temperature in the room changes.

In [32] we propose a systematic process for conducting *System Identification* for adaptive software systems, along with a language that can be used to represent how changes in system parameters affect the indicators of requirements convergence. After *AwReqs* have been elicited as *indicators*, the *System Identification* process consists of three activities:

1. **Identify parameters:** determine points of variability in the system (OR-decompositions, parameters related to system goals or tasks) whose change

of value affects any of the indicators. For instance, the set of required fields (an enumerated parameter) affects *AwReq* `AR1` (see table 1) — less required fields makes inputting information easier; the number of ambulances, as well as operators and dispatchers working, affects *AwReq* `AR5` — the higher the number, the higher the chances of fast assistance;

2. **Identify relations:** for each indicator–parameter pair (not only the ones identified in the previous step, but the full $\{indicators\} \times \{parameters\}$ Cartesian product), verify if there is a relation between changes in the parameter and the value of the indicator. For each existing relation, model qualitative information about the nature of the effect using differential equations. For example, $\Delta(AR1/RequiredFields) < 0$ indicates that decreasing the required fields (assuming the enumerated values form a totally ordered set) increases the success of `AR1`; $\Delta(AR5/NumberOfAmbulances) > 0$ states that increasing the number of ambulances also increases the success of `AR5`;
3. **Refine relations:** after identifying initial relations, the model can be refined by comparing and combining those that refer to the same indicator. For example, $\Delta(AR5/NumberOfAmbulances) > \Delta(AR5/NumberOfOperators)$ tells us that buying more ambulances is more effective than hiring more operators when considering how fast ambulances get to emergency sites.

A more detailed explanation of the *System Identification* process and the proposed language for modeling relations between indicators and parameters can be found in [32]. However, the basic examples above already give us the intuition that this kind of information is very important in order to determine the best way to adapt the target system and, therefore, the models produced by *System Identification* can be used by the adaptation framework for this purpose. Adaptation strategies are discussed next.

7.2 Adaptation Strategies

There are several ways a system can be changed as a result of its failure to attain the requirements. We call one such possibility adaptation. Here, the system's configuration (the values of its parameters) is changed in attempt to achieve the indicator targets. This can be viewed as parameter tuning. There can be a number of possible reconfiguration strategies based on the amount of information available in the system identification model. The more information is available and the more quantitative it is, the more precise and advanced the reconfiguration strategies can become. The reconfigurations involve changing the values of the system parameter(s), which affect indicator(s) that failed to achieve their target values. With the absence of a fully quantitative model relating parameters and indicators, an adaptation strategy may involve a number of such reconfigurations that are performed in succession in attempt to bring the indictor value to its target. When more precise information is available, quantitative approaches, e.g., mimicking the PID controller [15] can be used. Detailed specification and analysis of these strategies is one of the subjects of our current research.

In addition to reconfiguring a system, *Evolution Requirements*, which describe evolutions of other requirements, can be used to identify specific changes to the

system requirements under particular conditions (usually requirements failures, negative trends on achieving requirements, or opportunities for improvement). Unlike reconfigurations discussed above, evolution requirements may change the space of alternatives available for the system. In our recent work [33], we have identified a number of adaptation strategies, including *abort*, *retry*, *delegate* to an external agent, *relax/strengthen* the requirement, etc., constructed from the basic requirements evolution operations such as *initiate* (a requirement instance), *rollback* (changes due to an attempt to achieve a requirement), etc. These adaptation strategies can be applied at the requirements instance level (thus, fixing/improving a particular system instance) and/or type level (thereby changing the behavior of all subsequent system instances). Reconfiguration is considered as one possible adaptation strategy. It can be applied at both levels. Further, [33] proposes an ECA-based process for executing adaptation strategies in response to failures. Triggered by *AwReq* failures, this process attempts to execute the possibly many adaptation strategies associated with the *AwReq* in their preference order, while defaulting to the abort strategy if others do not prove successful.

We stress here that Awareness Requirements are absolutely crucial in our vision for requirements-driven adaptive systems design. They serve both as the means to specify targets to be met by the system (i.e., reference inputs for the feedback controller) and as the indicators of requirements convergence (i.e., the monitored outputs), with their failures triggering the above-described adaptation strategies.

8 Conclusions

The main contribution of this paper is the definition of a new class of requirements that impose constraints on the run-time success rate of other requirements. The technical details of the contribution include linguistic constructs for expressing such requirements (reference to other requirements, requirement states, temporal operators), expression of such requirements in OCL_{TM}, as well as portions of a prototype implementation founded on an existing requirements monitoring framework. We have also discussed the role of *AwReqs* in a complete process for the development of adaptive systems using a feedback loop-based adaptation framework that builds on top of this monitoring framework.

Other than working towards the full feedback loop implementation discussed in section 7, future steps in our research include the integration of domain models in the approach (as mentioned in section 6) and improvements in the definition and specification of *AwReqs*. Other questions also present themselves as opportunities for future work in the context of this research: what is the role of contextual information in this approach? How could we add predictive capabilities or probabilistic reasoning in order to avoid failures instead of adapting to them? Could this approach help achieve requirements evolution? These and other questions show how much work there is still to be done in this research area.

References

1. Report of the inquiry into the London Ambulance Service. South West Thames Regional Health Authority (1993)

2. Object Constraint Language, OMG Available Specification, Version 2.0 (2006), http://www.omg.org/cgi-bin/doc?formal/2006-05-01
3. Baresi, L., Pasquale, L., Spoletini, P.: Fuzzy Goals for Requirements-driven Adaptation. In: Proc. of the 18th IEEE International Requirements Engineering Conference, pp. 125–134. IEEE (2010)
4. Berry, D.M., Cheng, B.H.C., Zhang, J.: The Four Levels of Requirements Engineering for and in Dynamic Adaptive Systems. In: Proc. of the 11th International Workshop on Requirements Engineering: Foundation for Software Quality, pp. 95–100 (2005)
5. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An Agent-Oriented Software Development Methodology. Autonomous Agents and Multi-Agent Systems 8(3), 203–236 (2004)
6. Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Engineering Self-Adaptive Systems through Feedback Loops. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009)
7. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
8. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed Requirements Acquisition. Science of Computer Programming 20(1-2), 3–50 (1993)
9. Dobson, S., et al.: A Survey of Autonomic Communications. ACM Transactions on Autonomous and Adaptive Systems 1(2), 223–259 (2006)
10. Doyle, J.C., Francis, B.A., Tannenbaum, A.R.: Feedback Control Theory. Macmillan Coll Div (1992)
11. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification. In: Proc. of the 21st International Conference on Software Engineering, pp. 411–420. ACM (1999)
12. Flake, S.: Enhancing the Message Concept of the Object Constraint Language. In: Proc. of the 16th International Conference on Software Engineering & Knowledge Engineering, pp. 161–166 (2004)
13. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. Computer 37(10), 46–54 (2004)
14. Giese, H., Cheng, B.H.C. (eds.): Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. ACM (2011)
15. Hellerstein, J.L., Diao, Y., Parekh, S., Tilbury, D.M.: Feedback Control of Computing Systems, 1st edn. Wiley (2004)
16. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design Science in Information Systems Research. MIS Quarterly 28(1), 75–105 (2004)
17. Jureta, I., Mylopoulos, J., Faulkner, S.: Revisiting the Core Ontology and Problem in Requirements Engineering. In: Proc. of the 16th IEEE International Requirements Engineering Conference, pp. 71–80. IEEE (2008)
18. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
19. Kramer, J., Magee, J.: A Rigorous Architectural Approach to Adaptive Software Engineering. Journal of Computer Science and Technology 24(2), 183–188 (2009)

20. van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications, 1st edn. Wiley (2009)
21. Letier, E., van Lamsweerde, A.: Reasoning about Partial Goal Satisfaction for Requirements and Design Engineering. In: Proc. of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, vol. 29, pp. 53–62. ACM (2004)
22. Menasce, D.A., Gomaa, H., Malek, S., Sousa, J.A.P.: SASSY: A Framework for Self-Architecting Service-Oriented Systems. IEEE Software 28(6), 78–85 (2011)
23. Parashar, M., Figueiredo, R., Kiciman, E.E. (eds.): Proceedings of the 7th International Conference on Autonomic Computing. ACM (2010)
24. Robinson, W.N.: A requirements monitoring framework for enterprise systems. Requirements Engineering 11(1), 17–41 (2006)
25. Robinson, W.N.: Extended OCL for Goal Monitoring. Electronic Communications of the EASST 9 (2008)
26. Robinson, W.N., Fickas, S.: Designs Can Talk: A Case of Feedback for Design Evolution in Assistive Technology. In: Lyytinen, K., Loucopoulos, P., Mylopoulos, J., Robinson, B. (eds.) Design Requirements Engineering. LNBIP, vol. 14, pp. 215–237. Springer, Heidelberg (2009)
27. Robinson, W.N., Purao, S.: Monitoring Service Systems from a Language-Action Perspective. IEEE Transactions on Services Computing 4(1), 17–30 (2011)
28. Rohleder, C., Smith, J., Dix, J.: Requirements Specification - Ambulance Dispatch System. Tech. rep., Software Engineering (CS 3354) Course Project, University of Texas at Dallas, USA (2006), http://www.utdallas.edu/~cjr041000/
29. Rosenthal, D.: Consciousness and Mind, 1st edn. Oxford University Press (2005)
30. Sousa, J.P., Balan, R.K., Poladian, V., Garlan, D., Satyanarayanan, M.: A Software Infrastructure for User–Guided Quality–of–Service Tradeoffs. In: Cordeiro, J., Shishkov, B., Ranchordas, A., Helfert, M. (eds.) ICSOFT 2008. CCIS, vol. 47, pp. 48–61. Springer, Heidelberg (2009)
31. Souza, V.E.S.: An Experiment on the Development of an Adaptive System based on the LAS-CAD. Tech. rep., University of Trento (2012), http://disi.unitn.it/~vitorsouza/a-cad/
32. Souza, V.E.S., Lapouchnian, A., Mylopoulos, J.: System Identification for Adaptive Software Systems: A Requirements Engineering Perspective. In: Jeusfeld, M., Delcambre, L., Ling, T.-W. (eds.) ER 2011. LNCS, vol. 6998, pp. 346–361. Springer, Heidelberg (2011)
33. Souza, V.E.S., Lapouchnian, A., Mylopoulos, J.: (Requirement) Evolution Requirements for Adaptive Systems. In: Proc. of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 155–164. IEEE (2012)
34. Souza, V.E.S., Mylopoulos, J.: From Awareness Requirements to Adaptive Systems: a Control-Theoretic Approach. In: Proc. of the 2nd International Workshop on Requirements@Run.Time, pp. 9–15. IEEE (2011)
35. Tallabaci, G.: System Identification for the ATM System. Master thesis, University of Trento (to be submitted, 2012)
36. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.-M.: RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In: Proc. of the 17th IEEE International Requirements Engineering Conference, pp. 79–88. IEEE (2009)
37. Yu, E.S.K., Giorgini, P., Maiden, N., Mylopoulos, J.: Social Modeling for Requirements Engineering, 1st edn. MIT Press (2011)