Awareness Requirements for Adaptive Systems*

Vítor E. Silva Souza
Dept. Inf. Eng. and Computer Science - Univ. of Trento
Via Sommarive, 14 - Trento, Italy - 38123
vitorsouza@disi.unitn.it

Alexei Lapouchnian
Dept. of Computer Science - University of Toronto
10 King's College Road - Toronto, Canada - M5S 3G4
alexei@cs.toronto.edu

William N. Robinson
Dept. of Computer Inf. Sys. - Georgia State University
35 Broad St NW, Suite 927 - Atlanta, GA, USA - 30303
wrobinson@gsu.edu

John Mylopoulos
Dept. Inf. Eng. and Computer Science - Univ. of Trento
Via Sommarive, 14 - Trento, Italy - 38123
jm@disi.unitn.it

ABSTRACT

Recently, there has been a growing interest in self-adaptive systems. Roadmap papers in this area point to feedback loops as a promising way of operationalizing adaptivity in such systems. In this paper, we define a new type of requirement — called Awareness Requirement — that can refer to other requirements and their success/failures. We propose a way to elicit and formalize such requirements and offer a requirements monitoring framework to support them.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Requirements/Specifications

General Terms

Requirements, Adaptivity

Keywords

Requirements engineering, modeling, adaptive systems, awareness, monitoring

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 [9]) 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., [3]).

Feedback loops constitute an architectural prosthetic to a system proper, introducing monitoring, analysis/diagnosis, etc. functionalities to the overall system. 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` ..." [4]. 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. This means 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.

*An extended version of this paper is available in [16].

SEAMS '11, May 23–24, 2011, Waikiki, Honolulu, HI, USA

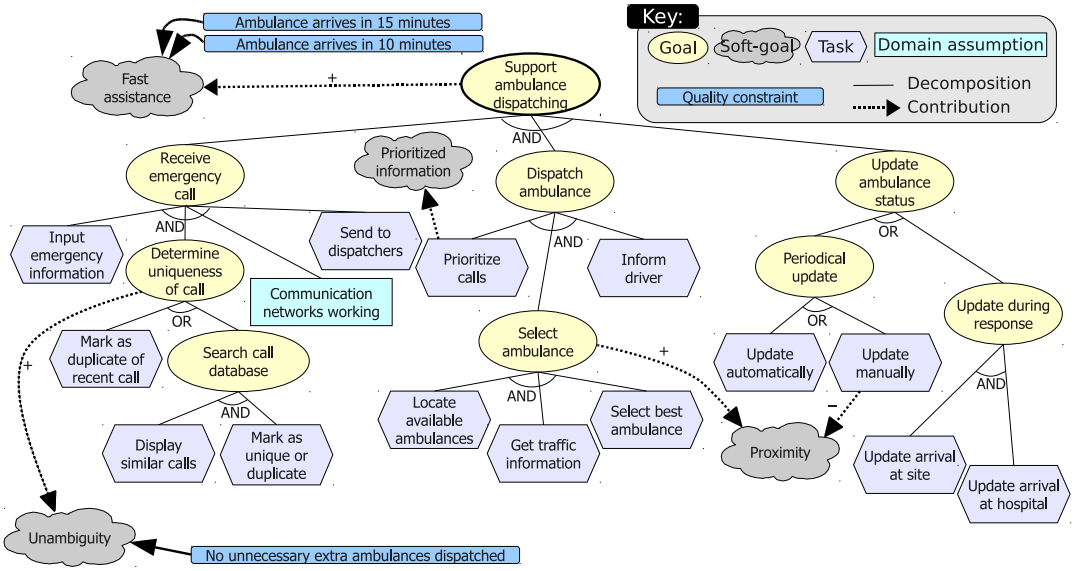

Figure 1: An example of a high-level goal model for an Ambulance Dispatch System.

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 a formal language and can be directly monitored by a requirements monitoring framework. In the future, we plan to propose a systematic process for designing full MAPE loops from a set of *AwReqs*. Here, however, our focus is on the definition and study of *AwReqs* and their monitoring at runtime.

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) [15]. In Psychology, consciousness has been studied as "self-referential behavior". Closer to home, awareness is a major design issue in HCI and CSCW. The concept in various forms is also of interest in the design of software systems (security / process / context / location / ... awareness).

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 formalization; section 5 talks about *AwReqs* monitoring implementation and presents evaluation results from experiments with our proposal; section 6 summarizes related work; finally, section 7 concludes the paper.

2. BASELINE

This section briefly presents research background on GORE and requirements monitoring.

2.1 Goal-Oriented Requirements Engineering

To model and analyze requirements, we adopt a goal-oriented approach: requirements are goals that stakeholders want to fulfill and constitute the mandate of a system-to-be. Goal-Oriented RE (GORE) approaches adopt as primitives concepts such as: goals, softgoals, quality constraints (QCs) and domain assumptions (DAs) [8]. Figure 1 shows a goal model for an Ambulance Dispatch System (ADS).

In our example, the main goal of the system is to support ambulance dispatching. Goals can be AND/OR decomposed with obvious semantics. 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". On the other hand, periodic update of an ambulance's status can be performed either automatically or manually. Goals are decomposed 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.

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 QCs, which are perceivable and measurable entities that inhere in other entities [8]. In our example, unambiguity is measured by the number of times two ambulances are dispatched to the same location,

```
context Buyer
  -- Seller responds with a quote for each Buyer quote request.
  def: buyerQuote: LTL::OclMessage = receivedMessage('quote')
  def: sellerQuoteRequest: LTL::OclMessage =
    receivedMessage('org.eeat.samples.order.Seller', 'rRequestQuote')
  inv getsQuote:
    after(eventually(sellerQuoteRequest <> null),
    eventually(sellerQuoteRequest.argument('qID') =
    buyerQuote.argument('qID')))
```

Figure 2: An example of OCL_{TM} constraint.

while fast assistance is refined into two QCs: ambulances arriving within 10 or 15 minutes to the emergency site.

Finally, domain assumptions 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.

2.2 Requirements Monitoring

In our proposal, as discussed in the introduction, adaptivity is to be implemented through MAPE feedback loops. Monitoring is the first step in this kind of loop and 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[1], formerly known as ReqMon [11]. 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 [12]. OCL_{TM} extends OCL 2.0 with: Flake's approach to messages [6], standard temporal operators ($\circ$, $\Diamond$, $\Box$, etc.), scopes and patterns as defined by Dwyer et al. [5] and timeouts that can be associated with such scopes.

Figure 2 shows an example of OCL_{TM} constraint, taken from a protocol definition for buyer and seller agents. The invariant `getsQuote` determines that if the buyer receives the `quote` message, eventually the seller should receive the `rRequestQuote` message and both messages should refer to the same `qID` argument. Given an instrumented Java implementation of these actors 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 formalized in any language that provides temporal constructs (e.g., LTL), examples of *AwReq* formalization in section 4 will be given using OCL_{TM}, which is also the language used for our proposal's validation, presented in section 5.

[1]http://eeat.cis.gsu.edu:8080/

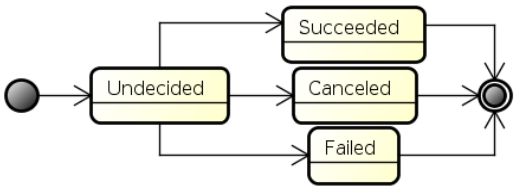

Figure 3: States assumed by requirements.

3. AWARENESS REQUIREMENTS

AwReqs are requirements that talk about the success or failure of other requirements. More generally, *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.

Table 1 shows some of the *AwReqs* that were elicited during the analysis of the ADS. These examples are presented to illustrate the different types of *AwReqs*, which are discussed in the following paragraphs, and also some patterns (§3.1) that can facilitate their elicitation (§3.2).

We can identify 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 — not discussed in this paper) 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.

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. `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

Table 1: Examples of *AwReqs*, elicited in the context of the ADS, and their types and patterns.

Id	Description	Type	Pattern
AR1	*Input emergency information* should never fail	Regular	NeverFail(T-InputInfo)
AR2	*Communications networks working* should have 99% success rate	Aggregate	SuccessRate(D-CommNetsWork, 99%)
AR3	*Search call database* should have a 95% success rate over one week periods	Aggregate	SuccessRate(G-SearchCallDB, 95%, 7d)
AR4	*Dispatch ambulance* should fail at most once a week	Aggregate	MaxFailure(G-DispatchAmb, 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)

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 min.

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.

These three non-regular types of *AwReq* bear similarities with the three components of the proportional-integral-differential (PID) controller, a widely used feedback controller type [4]. 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, while trend *AwReqs* follow the idea of the derivative control, which sets its output according to the rate of change of the control error.

Finally, `AR10` and `AR11` are the examples of meta-*AwReqs*: *AwReqs* that talk about other *AwReqs*. 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 DB 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.

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 a third-level *AwReq*). There is no limit on how many levels can be created and to avoid circular references we organize requirements in different strata, like depicted in figure 4, allowing *AwReqs* to only reference requirements from the stratum directly below.

3.1 Patterns and Graphical Representation

Formalizing *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 the communication among system analysts and designers.

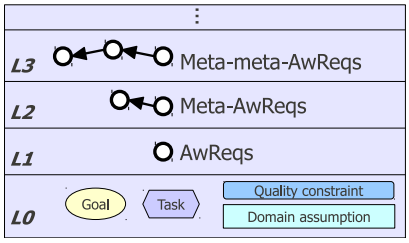

Figure 4: Strata for Awareness Requirements.

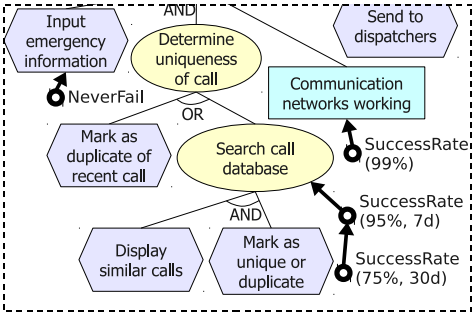

Figure 5: Graphical representation of *AwReqs*.

The last column in table 1 shows the patterns for each of the *AwReqs* elicited for our running example. The list of patterns shown in the table is by no means exhaustive and each organization is free to define its own patterns. By using such patterns we create a common vocabulary for analysts and code generation tools could be provided to automatically write *AwReqs* in the language of choice based on the pattern, relieving requirements engineers from most of the OCL coding. In section 5.1, we provide EEAT/OCL$_{TM}$ idioms for this kind of code generation.

Furthermore, patterns are used in the graphical representation of *AwReqs* 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 5. Due to space limitations, we show only a small portion of the goal model with three *AwReqs* and a meta-*AwReq*. *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 formalization elsewhere.

3.2 Awareness Requirements Elicitation

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*.

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, *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/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 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 [2] and other variations of goal modeling notation, alternatives introduced by OR-decomposed goals are frequently evaluated with respect to certain softgoals. The goal *Periodical updates* in figure 1 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. E.g., 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

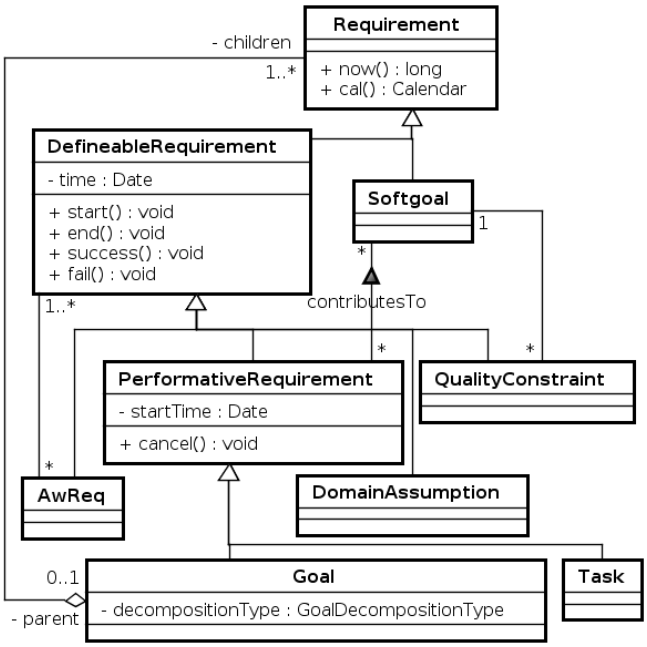

Figure 6: Class model for requirements in GORE.

```
context T-InputInfo
  inv AR1: never(self.oclInState(Failed))
context G-SearchCallDB
  def: all : G-SearchCallDB.allInstances()
  def: week : all->select(g | new Date().difference(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.difference(q.time, DAYS) <= 30)
  def: m2 : Set = all->select(q | (now.difference(q.time, DAYS) <= 60)
    and (now.difference(q.time, DAYS > 30)))
  def: m3 : Set = all->select(q | (now.difference(q.time, DAYS) <= 90)
    and (now.difference(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))
  def: rate1 : Double = success1->size() / m1->size()
  def: rate2 : Double = success2->size() / m2->size()
  def: rate3 : Double = success3->size() / m3->size()
  inv AR7: never((rate3 < rate2) and (rate2 < rate1))
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.difference(self.time, MINUTES) <= 10))
context T-MarkUnique
  inv AR9: eventually(not self.oclInState(Undecided)) and
    never(self.time.difference(self.startTime, MINUTES) > 5)
```

Figure 7: Some *AwReqs* formalized in OCL$_{TM}$.

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. FORMALIZING AWREQS

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. The language we have chosen is OCL$_{TM}$. 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 the system requirements (previously modeled as a goal model) in a UML class diagram is a necessary step for the formalization of *AwReqs* in any OCL-based language, as OCL constraints refer to classes and their instances, attributes and methods.

The model illustrated in figure 6 can be extended to specify requirements. For example, consider `AR1` (table 1), which refers to a UML `Task` requirement. Figure 7 presents `AR1` as an OCL invariant on the class `T-InputInfo`, which is a subclass of `Task` (from figure 6) 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.

Each requirement of our system is represented by a UML class, extending the appropriate class in the diagram of figure 6, like the `T-InputInfo` example we have just described. Mnemonics were used to name these classes so one can deduce which requirement of figure 1 is being represented from the class name. Moreover, the first letter of each class name indicates which element of figure 6 is being extended (`T` for `Task`, `G` for `Goal` and so forth). It is important to note that these classes are only an abstract representation of the elements of the goal model and they are part of the 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 7 also shows examples of formalization for different categories of *AwReqs*: aggregate (`AR3`), trend (`AR7`) and delta (`AR8` and `AR9`). Meta-*AwReqs* also belong to one of the above categories and, thus, are formalized similarly.

Aggregate *AwReqs* place constraints over a collection of instances. In `AR3`, all instances of `G-SearchCallDB` are retrieved in a set, then we use the `select()` operation 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`). Also, we used date comparison as in [12] to indicate the evaluation should be done considering only the past week. Trend *AwReqs* are similar, but 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-TwoDispatches` 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-UpdAt-`

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

context G-SearchCallDB
    def: weekA : LTL::OclMessage = receivedMessage('newWeek')
    def: weekB : LTL::OclMessage = receivedMessage('newWeek')
    def: allS : Set(LTL::OclMessage) = receivedMessages('success')
    def: allF : Set(LTL::OclMessage) = receivedMessages('fail')
    def: wS : Integer = allS->select(m | now() - m.timestamp() < week())->size()
    def: wF : Integer = allF->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')
```

Figure 8: Formalization of *AwReqs* for EEAT.

`Site` that are related to `T-InformDriver` in the `related` set by comparing the `callID` argument using OCL$_{TM}$'s `arguments()` operation [12]. 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. `AR9` shows how to formalize 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

This section presents an evaluation of our proposal. Using experimental and descriptive evaluation methods of Design Science [7], we show that *AwReqs* can be monitored at runtime (§5.1) and that this monitoring framework can provide value for the analysis of a real system (§5.2). We also briefly discuss the performance of this solution (§5.3).

5.1 Monitoring AwReq Patterns

As mentioned in section 2.2, 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 formalization of the *AwReqs* to one based on methods received by the run-time instances which represent the requirements. Figure 8 shows three of the five *AwReqs* formalized previously in figure 7 in their "EEAT formalizations".

For monitoring to work, then, the source code of the monitored system (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 `DefineableRequirement` and `PerformativeRequirement` from figure 6. 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 §2.2), 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 formalization" 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 `begin()` 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 formalization to their "EEAT formalization" could be built to aid the designer in this task. Another possibility is to go directly from the *AwReq* patterns presented in section 3.1 to this final formalization. Table 2 illustrates how some of the patterns of table 1 can be expressed in OCL$_{TM}$. These formulations are consistent with those shown in figure 8. 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, invariant `pR` in the first row of table 2 is true if `R.fail()` is never called.

Of course, the patterns of table 1 represent only common kinds of expressions. *AwReqs* include 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 2. 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.

Table 2: 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`

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 AwReq 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 2 and figure 8. Scenarios were developed to exercise each requirement so that each of them should evaluate as fail or success. 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`:

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 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 runtime 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 [13, 14].

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 five percent, and typically less than 1 percent 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 [5]. 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 [13]. This indicates that EEAT is reasonably efficient for many monitoring problems.

6. RELATED WORK

A number of recent proposals offer alternative ways of expressing and reasoning about partial requirements satisfaction. RELAX by Whittle, et al. [17] 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 [1]. FLAGS requirements models are based on the KAOS framework 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. We are currently working on integrating compensations into our approach and a full-fledged compensation language remains future work. However, 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.

Letier and van Lamsweerde [10] 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 vars. and objective functions). Objective functions can be quite similar to *AwReqs*, except they use probabilities. For instance, one such function presented in [10] 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. DISCUSSION AND CONCLUSIONS

The main contribution of this paper is the definition of a new class of requirements that impose constraints on the run-time behavior 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 fragments of a prototype implementation founded on an existing requirements monitoring framework.

The real usefulness of this new class of requirements for adaptive systems, however, comes from a full implementation of the feedback loop that provides adaptivity, most likely instantiating the well-known monitor-analyze-plan-execute feedback loop proposed by autonomic computing researchers [9]. A conceptual architecture for such a loop is presented and discussed in some detail in the extended version of this paper [16]. Defining a systematic approach for the design of adaptive systems starting on *AwReqs* and integrating such systems in a framework that implements this feedback loop architecture is at the core of our future work.

Concerning *AwReqs* themselves, 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 formalization of *AwReqs*. For instance, we plan on developing support for *adaptivity AwReqs* such as "if requirement `r` fails more than `N` times over a time period, relax it", which are closely related to the RELAX proposal.

8. REFERENCES

[1] L. Baresi, L. Pasquale, and P. Spoletini. Fuzzy goals for requirements-driven adaptation. *IEEE International Conference on Requirements Engineering*, pages 125–134, 2010.

[2] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[3] B. Cheng et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. *Software Engineering for Self-Adaptive Systems*, 5525/2009:1–26, 2009.

[4] J. Doyle, B. Francis, and A. Tannenbaum. *Feedback Control Theory*. McMillan Publishing, 1990.

[5] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, Los Angeles, USA, 1999. ACM Press.

[6] S. Flake. Enhancing the Message Concept of the Object Constraint Language. In *SEKE '04: Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering*, pages 161–166, Banff, Canada, 2004.

[7] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.

[8] I. J. Jureta, J. Mylopoulos, and S. Faulkner. Revisiting the Core Ontology and Problem in Requirements Engineering. In *RE '08: 16th IEEE International Requirements Engineering Conference*, pages 71–80, Barcelona, Spain, 2008. IEEE.

[9] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[10] E. Letier and A. van Lamsweerde. Reasoning about partial goal satisfaction for requirements and design engineering. In *SIGSOFT '04/FSE-12: 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 53–62, 2004.

[11] W. N. Robinson. A requirements monitoring framework for enterprise systems. *Requirements Engineering*, 11(1):17–41, Mar. 2006.

[12] W. N. Robinson. Extended OCL for Goal Monitoring. In *Ocl4All '07: Proceedings of the 7th International Workshop on Ocl4All: Modelling Systems with OCL*, Nashville, USA, 2007. Springer Berlin / Heidelberg.

[13] W. N. Robinson and S. Fickas. Designs can talk: A case of feedback for design evolution in assistive technology. In W. A. et al., editor, *Design Requirements Engineering: A Ten-Year Perspective*, volume 14 of *Lecture Notes in Business Information Processing*, pages 215–237. Springer, 2009.

[14] W. N. Robinson and S. Purao. Monitoring service systems from a language-action perspective. *IEEE Transactions on Services Computing*, 2010.

[15] D. Rosenthal. *Consciousness and Mind*. Oxford University Press, USA, Jan. 2006.

[16] V. E. S. Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos. Awareness requirements for adaptive systems: Extended report. In *University of Trento Technical report DISI-11-352*, 2011.

[17] J. Whittle, P. Sawyer, N. Bencomo, B. H. Cheng, and J.-M. Bruel. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In *RE '09: Proceedings of the 17th IEEE International Requirements Engineering Conference*, pages 79–88, Atlanta, USA, 2009. IEEE.