

AWARENESS REQUIREMENTS FOR ADAPTIVE SYSTEMS

Vítor E. Silva Souza, Alexei Lapouchnian, William N.
Robinson, John Mylopoulos

August 2010

Technical Report # DISI-10-049

The contents of this report were submitted for publication at the 33rd International
Conference on Software Engineering - ICSE 2011

Awareness Requirements for Adaptive Systems

Vítor E. Silva Souza
Department of Information
Engineering and Computer
Science - University of Trento
Via Sommarive, 14 - Trento,
Italy - 38123
vitorsouza@disi.unitn.it

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

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

John Mylopoulos
Department of Information
Engineering and Computer
Science - University 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 present a new type of requirement – called Awareness Requirement – that can refer to other requirements and their success/failures, constituting requirements for such feedback loops. We propose a way to elicit and formalize such requirements and validate our proposal using a monitoring framework. We further discuss how feedback loops could be implemented to provide adaptivity mechanisms to systems.

Categories and Subject Descriptors

D.2 [Software Engineering]: Requirements/Specifications

General Terms

Requirements, Adaptivity

Keywords

requirements engineering, modeling, self-adaptive systems, awareness, feedback loops, 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) 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., [8, 2, 7]).

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 are the requirements problems these solutions are intended to solve? The nucleus of an answer to this question can be gleaned 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 ..." [11]. Suppose then that we have a requirement r = "supply customer upon request" and let s be one of the operationalizations of r . Suppose further that the "desired way" of the above quote for this system is that every time it is fed another customer request it 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". We can generalize on this: we could require that the system 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 runtime success/failure or other requirements, including qualities. 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 the availability of suppliers is an issue for our system, we may want to add yet another requirement

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE '11 Honolulu, Hawaii, USA

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$10.00.

r2 = “d won’t fail more than 2% of the time during any 1-month period”. This is also an awareness requirement, but is concerned with the truth/falsity of domain assumptions. We might call these contextual awareness requirements.

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, which can be integrated into a requirements monitoring framework to fulfill the first stage of the MAPE feedback loop. We also provide discussion of a systematic process for generating architectural elements consisting of feedback loops to accommodate a given set of awareness requirements, providing the later steps of the MAPE feedback loop and, thus, providing adaptivity to our system. The main contribution of this paper, however, is the definition of a new class of requirements that impose constraint on the runtime behaviour of other requirements.

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) [28]. 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).

1.1 Running Example and Paper Outline

As part of our proposal’s evaluation, which we detail in section 4, we have analyzed, designed and developed a real-world application: an Ambulance Dispatch System (ADS), whose requirements have been documented by students of the University of Texas at Dallas [27]. 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 for our work; section 3 characterizes *AwReqs*, their elicitation and formalization; section 4 discusses implementation and presents evaluation results from experiments with our proposal; section 5 discusses ideas for a full MAPE feedback loop that provides adaptivity; section 6 summarizes related work; finally, section 7 concludes the paper.

2. BASELINE

In this section we briefly present research and technology used in our proposal.

2.1 Goal-Oriented Requirements Engineering

Our approach is based on Goal-Oriented Requirements Engineering (GORE). GORE was proposed and its popularity grew because of the inadequacy of previous approaches when dealing with very complex systems. These approaches did not capture the rationale behind the requirements being modeled, thus making them difficult to understand with respect to high-level concerns in the problem domain [18].

There are many different approaches for GORE. Four of the most well-known ones are summarized in [18]. Our re-

search, however, is not based on any specific framework, method or methodology, but on the main concepts that most GORE approaches present: goals, softgoals, quality constraints (QCs) and domain assumptions (DAs) [16]. These approaches are centered in the goal model, which captures the goals of the different stakeholders of the system. Figure 1 shows the goal model for the Ambulance Dispatch System.

Goals represent the objectives of the stakeholders and users. In our example, the main goal of the system is to support ambulance dispatching. Goals can be decomposed using Boolean decompositions: an AND-decomposition means that in order to accomplish the parent goal, all of its children (sub-goals, tasks and domain assumptions) must be satisfied, while for an OR, only one of them has to be attained. We can reason over these relationships at runtime [30]. For example, to receive an emergency call, one has to input its information, determine its uniqueness and send it to dispatchers (we explain the domain assumption “Communication networks working” next). On the other hand, to perform periodical update of an ambulance’s status, it is enough to do it either automatically or manually.

Goals are decomposed until they reach a level of granularity in which they can be seen as a relatively simple task to be carried out by an actor. The main difference between a task and a goal is that the former’s satisfaction can be inferred by observation of the system, while the latter’s is a logical consequence of the satisfaction of its children. We represent goals and tasks differently in our models.

Softgoals are special types of goals that represent objectives that don’t 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. Soft-goal satisfaction can be estimated through qualitative contribution links that propagate satisfaction or denial (noted with a D) and have four levels of contribution: break or deny (-), hurt (-), help (+) and make or satisfy (++)). In our example, selecting an ambulance using the software system contributes positively to the closeness 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).

For the purposes of our research (writing *AwReqs*), we need to translate softgoals into things that can be measured. These are quality constraints (QCs), which are perceivable and measurable entities that inhere in other entities [16]. In our example, unambiguity is measured by the amount of times two ambulances are dispatched to the same location, while fast assistance is translated into two QCs: ambulances arriving in 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 provided and work properly. If this assumption were to be false, its parent goal (“Receive emergency call”) would not be satisfied.

2.2 Feedback Loops

The recent growth of software systems in size and complexity made it increasingly infeasible to control them manually. This led to the development of a new class of self-adaptive systems, which are capable of changing their be-

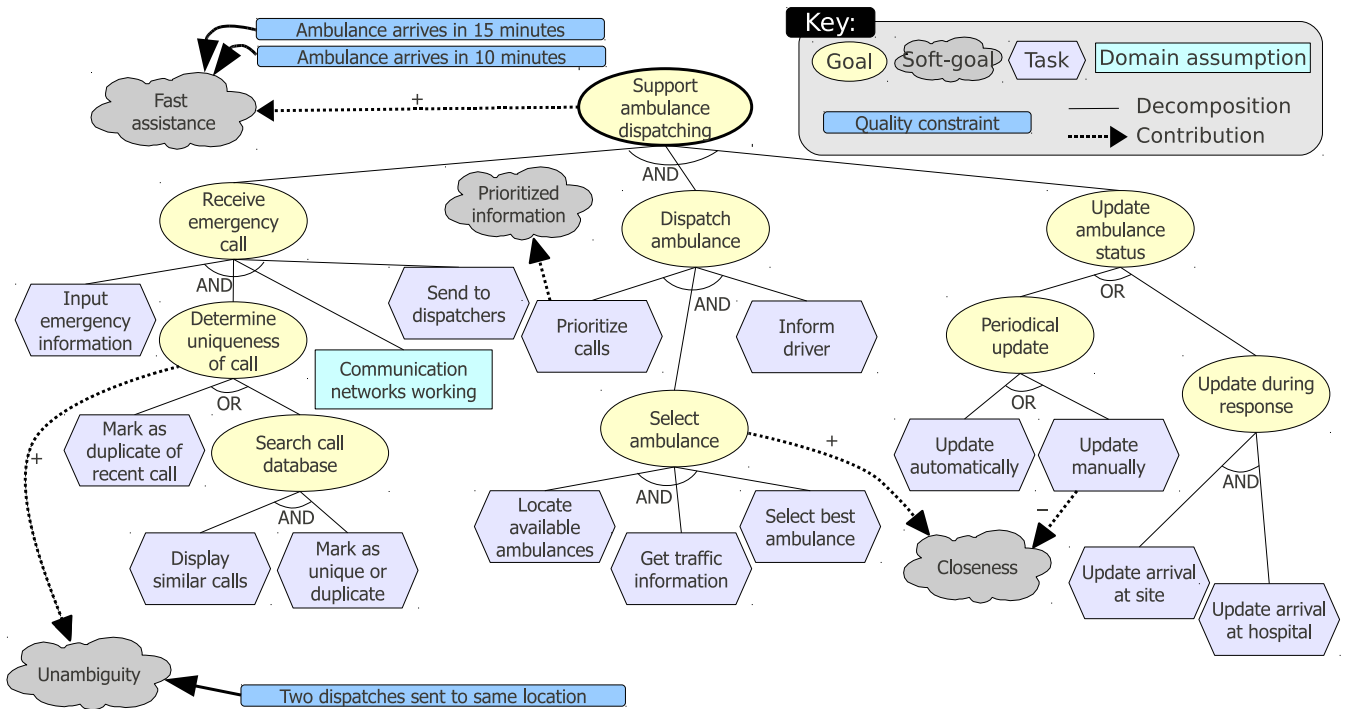


Figure 1: An example of a high-level goal model for an Ambulance Dispatch System

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. [5] argue for systematic software engineering approaches for developing self-adaptive systems based on the ideas from control engineering [14] 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 [14]. The goal 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 the 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 [10]. It focuses on the activities that realize feedback: monitoring, analysis, plan, execution.

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) [14]. 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.

2.3 Requirements Monitoring

In our proposal, as discussed in the previous section, adaptivity is to be implemented through MAPE feedback loops. Monitoring is the first step of this kind of loop and since *AwReqs* refer to the success/failure of other requirements, we will need to monitor requirements during runtime.

Therefore, we have based our implementation of feedback loops on requirements monitoring framework EEAT¹, formerly known as ReqMon [23]. 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 goals of the system during runtime.

EEAT’s architecture is presented in more detail along with our implementation in section 4. In EEAT, requirements can be specified in a variant of the Object Constraints Language (OCL), called OCL_{TM} – meaning OCL with Temporal Message logic [24]. OCL_{TM} extends OCL 2.0 [1] with:

- Flake’s approach to messages [13]: replaces the confusing `message()`, `^ message()` syntax with `sendMessage/s`, `receivedMessage/s` attributes in class `OclAny`;
- Standard temporal operators: \circ (next), \bullet (prior), \diamond (eventually), \blacklozenge (previously), \square (always), \blacksquare (constantly), \mathcal{W} (always ... unless), \mathcal{U} (always ... until);

⁹<http://eeat.cis.gsu.edu:8080/>

- The scopes defined by Dwyer et al. [12]: **globally**, **before**, **after**, **between** and **after ... until**. Using the scope operators simplifies property specification;
- Patterns, also in Dwyer et al. [12]: **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.

Although in our proposal *AwReqs* can be formalized in any language that provides temporal constructs (e.g. LTL), examples of *AwReq* formalization in section 3.2 will be given using OCL_{TM} , which was the language used for our proposal's validation, presented in section 4.

3. AWARENESS REQUIREMENTS

As we have mentioned in section 1, feedback loops can implement 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, we believe that in order to completely specify a system with adaptive characteristics, adaptivity requirements 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), formalize them (§3.2) and propose patterns to facilitate their elicitation, along with a way to represent them graphically in the goal model (§3.3). We illustrate all of our ideas using our running example, the ADS.

3.1 Characterization

AwReqs are requirements that talk about the success or failure of other requirements. In the context of GORE, requirements include goals, tasks, domain assumptions and quality constraints. As will be shown in this section, *AwReqs* can also refer to other *AwReqs*, constituting meta-*AwReqs*.

As with other requirements, *AwReqs* are obtained by talking to stakeholders and using the usual elicitation techniques, but specifically targeting adaptivity requirements. The following *AwReqs* were elicited during the analysis of the ADS.

AR1: task *Input emergency information* should never fail;

AR2: domain assumption *Communications networks working* should have 99% success rate;

AR3: goal *Search call database* should have a 95% success rate over one week periods;

AR4: goal *Dispatch ambulance* should fail at most once a week;

AR5: quality constraint *Ambulance arrives in 10 minutes* should succeed 60% of the time, while *Ambulance arrives in 15 minutes* should do it 80%, measured daily;

AR6: task *Update automatically* should succeed 100 times more than the task *Update manually*;

AR7: the success rate of QC *Two dispatches sent to the same location* for a month should not decrease, compared to the previous month, three times consecutively;

AR8: task *Update arrival at site* should succeed within 10 minutes of the successful execution of task *Inform driver*, for the same emergency call;

AR9: *AwReq AR3* should have 75% success rate over one month periods;

AR10: *AwReq AR5* should never fail.

AwReq AR1 shows the simplest form of *AwReq*: the requirement – the task *Input emergency information* – should never fail. Considering a control system, the requirement succeeding is the reference input. If the actual output is telling us the requirement has failed, the control system must act – compensate – in order to bring the system back to an acceptable state. Compensations are discussed in section 5.

AR1 considers every single instance of the referred requirement. An instance of a task exists every time an actor executes it using the system. For *AR1* this means that the constraint of "never failing" should be checked every time an operator attempts to input emergency information in the ADS. Similarly, instances of goals exist when instances of one of their subtasks exist (backward propagation of the execution), while DA instances are dependent on their parent goal/task and QC instances on goals/tasks specified by the analyst. This type of *AwReq* works for extremely critical requirements, ones that are supposed to be fulfilled no matter what. Compensations usually consist of alternative ways to reach the same results.

However, not all requirements are that critical. Most of them can tolerate some level of failure, considering the big picture. These kinds of *AwReqs* are called "aggregate *AwReqs*" because in order to determine their fulfillment you need to aggregate the instances of the referred requirement. *AR2* is an example of the simplest form of an aggregate *AwReq*: considering all the times an actor attempted to attain goal *Receive emergency call*, in 99% of them the assumption *Communication networks working* was indeed true.

If part of the stakeholders' requirements, aggregate *AwReqs* can also specify the period of time to consider when aggregating requirement instances. *AwReq AR3* exemplifies this case, indicating that instances of goal *Search call database* should be aggregated over one week periods. The frequency with which the requirement is to be verified is another optional parameter for *AwReqs*: *AR3* could be verified once a week or we could implement a daily verification of the past

seven days. In our case, since it’s not specified (i.e. it’s not an important factor according to stakeholders), it’s up to the designer to choose the best way to implement it. AR5 is an example of an *AwReq* with verification interval specified.

Another type of aggregate *AwReq* specifies not a percentage, but the minimum or maximum success/failure a requirement is supposed to have. For example, AR4 states that the goal *Dispatch ambulance* can fail at most once a week. *AwReqs* can combine different requirements, like AR5, where 60% of the ambulances should arrive in 10 minutes and 80% (i.e. another 20%) should arrive in at most 15 minutes. One can even compare the success counts of two requirements, as done in AR6: *Update automatically* should succeed at least 100 times more than *Update manually*.

Aggregate *AwReqs* work like the integral part of the proportional-integral-differential (PID) controller, a widely used feedback control loop [14]. Integral control considers not only the current difference between the output and the reference input (the control error), but aggregates the errors of the past measurements. Two other types of *AwReqs* were also inspired by the PID controller: “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.

AR7 is an example of a trend *AwReq*: the success rate of QC *Two dispatches sent to the same location* in a month should not be lower than the previous month for three months in a row. In other words, if r_i represents the success rate of month i , we would like to avoid the situation in which $r_{i+3} < r_{i+2} < r_{i+1} < r_i$. Trend *AwReqs* can be used to spot problems in how success/failure rates evolve through time.

Delta *AwReqs*, on the other hand, can be used to specify reference values for parameters of the requirements, most commonly execution time. AR8 specifies that task *Update arrival at site* should be satisfied (successfully finish execution) within 10 minutes of the execution of task *Inform driver*, meaning once the dispatcher has informed the ambulance driver where the emergency is, she should arrive there in less than 10 min. What needs to be verified is the difference Δ (hence the name) between the reference and the actual value: $\Delta = V_R - V_A$. If $\Delta < 0$, the *AwReq* is not satisfied.

Finally, AR9 and AR10 show us examples of meta-*AwReqs*. A meta-*AwReq* talks about the success/failure of another *AwReq*, resulting in feedback loops organized hierarchically. One of the motivations for meta-*AwReqs* is the application of gradual compensations. This is the case with AR9: if AR3 fails (i.e., *Search call database* has less than 95% success rate in a week), tagging the calls as “possibly ambiguous” (AR3’s compensation) might be enough, but if AR3’s success rate considering the whole month is below 75% (e.g. fails at least two out of four weeks), a deeper analysis of the problems of database searching might be in order (AR9’s compensation).

Another useful case for meta-*AwReqs* is to avoid executing a given compensation action 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 it’s compensation indicates that a failure in this QC should trigger messages to all users of the ADS. To avoid sending repeated messages in case it fails again, AR10 states that AR5 should never fail and, in case it does, its compensation decreases AR5’s percentages by 10 points (to 50% and 70%, respectively), which means

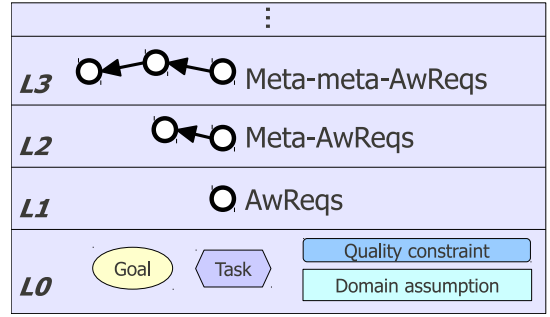


Figure 2: Requirements (in the context of GORE) in their respective stratum.

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, AR10’s compensation 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 one could model fourth-level *AwReqs*, fifth-level *AwReqs* and so on. To avoid circular references we organize requirements in different strata, like depicted in figure 2, and enforce a constraint that allows *AwReqs* to only reference requirements from the stratum directly below.

3.2 Formalization

We have just characterized *AwReqs* as requirements that refer to the success or failure of other requirements, thus making requirements themselves first-class citizens in the requirements language by allowing expressions about them. Furthermore, *AwReqs* may refer to requirements in specific periods of time. Our proposal is not coupled with any specific language, as long as it provides these two essential characteristics of *AwReqs*, with both design and runtime properties available in expressions. The examples of this section, however, use *OCL_{TM}* (see §2.3), as it was the language we have used for our proposal’s validation (more details in §4).

We’re interested in writing statements for five types of requirement: goals, tasks, domain assumptions, quality constraints and *AwReqs*. The first two are considered *long-running requirements* because actors pursue their fulfillment during a period of time. Also, together with domain assumption, goals and tasks are *child requirements*, which are the ones into which goals can be decomposed. Finally, all five types of requirement are *decidable*, meaning there are clear-cut criteria that indicate if they have succeeded or failed. Softgoals, not having such criteria, are clearly not *decidable requirements*.

The requirements model illustrated in figure 3 can be used to specify requirements. For example, consider AR1 (§3.1), which refers to a UML Task requirement. Figure 4 presents AR1 as an OCL invariant on the UML class *TInputInfo*, which is a subclass of *Task* (from figure 3) and represents requirement *Input emergency information*. An operation, i.e. a *receivedMessage*, is called on an instance of *TInputInfo* when it succeeds, fails, or is canceled. AR1 is satisfied when an instance of *TInputInfo* does not receive a *fail()* call, between calls for *start()* and *end()*. The *between* clause,

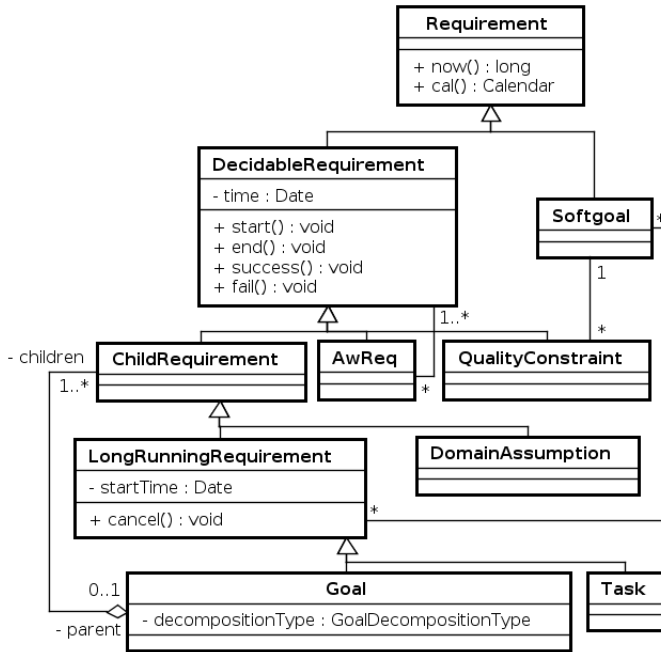


Figure 3: Class model for requirements in GORE.

one of Dwyer et al. scopes (see §2.3), allows us to say that `fail()` was never called, otherwise we would have to wait until the whole system shuts down (end of the global context) to make sure the objects never received such message.

So, in order to formalize *AwReqs* in such manner, each requirement of our system is represented by a UML class, extending the appropriate class in the diagram of figure 3, like the `TInputInfo` example we’ve just described. For space constraints, we don’t present a diagram showing all the classes that represent the requirements of the ADS, however one can deduce which requirement is being represented from the mnemonic used as class name. 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 4. 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.

Then, for monitoring to work, the monitored system should be instrumented in order to create instances of these classes when the requirements are being achieved, as we have briefly explained in the previous subsection. To delimit the scope, the system should call methods `start()` and `end()`. For *long-running requirements* these are called when the actors start and end pursuing that requirement, respectively, while for other types of requirement they should be called immediately before and after their validation. Between calls of these two methods, the system would call methods `success()`, `fail()` or `cancel()` when the requirement succeeded, failed or was purposefully canceled by the user (this last one only in the case of *long-running requirements*).

Aggregate *AwReqs* are a little more complicated, as we have to specify a period of time in which success and failure of requirements would be considered. As we can see in the formalization of `AR3` in figure 4, this is accomplished with the use of methods `now()`, which provides the current times-

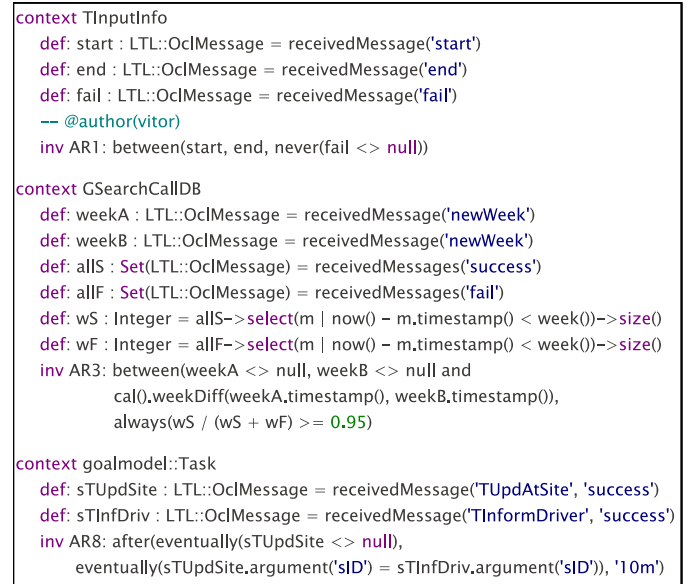


Figure 4: *AwReqs* formalized in OCL_{TM} .

tamp, and `cal()`, which provides a calendar utility, together with a clock component in the monitoring framework, which should send messages to our objects indicating the beginning of specific periods of time, such as `newHour`, `newDay` and, in the case of `AR3`, `newWeek`. Then, the `between` clause is used again to hold off the evaluation of the `always` clause until it has received messages `weekA` and `weekB`, which should represent a week period (the calendar utility is used to verify that). Once `weekB` is received, the number of `success()` calls (`wS`) and the number of `fail()` calls (`wF`) in the past week are computed and a simple calculation tells us whether or not the success rate was above the intended 95%.

Sometimes we need to refer to a domain-specific attribute of a requirement, as it’s the case of delta *AwReq* `AR8`: the tasks *Update arrival at site* and *Inform driver* should refer to the same emergency call. We use a session id (`sID`) argument attached to the messages to verify that. This can be implemented by having a collection of key-value pairs passed as parameters to the methods `start()`, `success()`, etc.

Finally, notice in figure 4 that OCL annotations are supported and can be used to add meta-data to *AwReqs*. Requirements attributes which refer to design or runtime properties can be useful in many different contexts. As a simple example, `AR1`’s invariant is annotated with the author of that specific *AwReq*.

The remaining *AwReqs* in the ADS can be formalized analogously to the ones presented in figure 4 and their formalization isn’t shown for reasons of space limitation.

3.3 Patterns and Graphical Representation

As we’ve seen in the previous subsection, formalizing *AwReqs* is not a trivial task. For this reason we propose *AwReq* patterns to facilitate the elicitation and analysis of *AwReqs*, and a graphical representation that allows us to include them in the goal model, improving the communication among system analysts and designers.

Many *AwReqs* have similar structure, such as “something must succeed so many times”. By defining patterns for

Table 1: *AwReq* patterns.

Pattern	Meaning
NeverFail(R)	Requirement R should never fail. Analogous pattern AlwaysSucceed, NeverCanceled, etc.
SuccessRate(R, r, t)	R should have at least success rate r over time t, where t is optional.
SuccessRateExecutions(R, r, n)	R should have at least success rate r over the latest n executions.
ComparableSuccess(R, S, x, t)	R should succeed at least x times more than S over time t, where t is optional.
MaxFailure(R, x, t)	R should fail at most x times over time t. Analogous patterns MinFailure, MinSuccess and MaxSuccess.
P_1 and/or P_2 ; not P	Conjunction, disjunction and negation of patterns.

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 the next section, 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 OCL coding.

Table 1 shows a non-exhaustive list of patterns. Each organization is free to define its own patterns and use them during requirements analysis. Using the patterns of table 1, mnemonics to refer to the requirements and abbreviated amounts of time like in OCL_{TM} timeouts [24], we can represent some of the ADS’ *AwReqs* as: *NeverFail(InputInfo)* (AR1), *SuccessRate(CommNetsWork, 99%)* (AR2), *MaxFailure(DispatchAmb, 1, 7d)* (AR4), *SuccessRate(Amb10min, 60%)* and *SuccessRate(Amb15min, 80%)* (AR5), *ComparableSuccess(UpdAuto, UpdManual, 100)* (AR6), etc.

Given that *AwReqs* can be shortened by a pattern we propose *AwReqs* 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 5. For reasons of space, 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* doesn’t fit a pattern, the analyst should write its name and document its OCL formalization elsewhere.

4. IMPLEMENTATION AND EVALUATION

Our evaluation of the *AwReqs* proposal considers two aspects of the framework:

1. Can *AwReqs* be monitored? Specifically, can an automated monitor evaluate requirements types enumerated in table 1 at runtime? Applying a constructive experiment, we show this is true;
2. Can the *AwReqs* framework provide value for the analysis of a real system? With simulation experiments, we

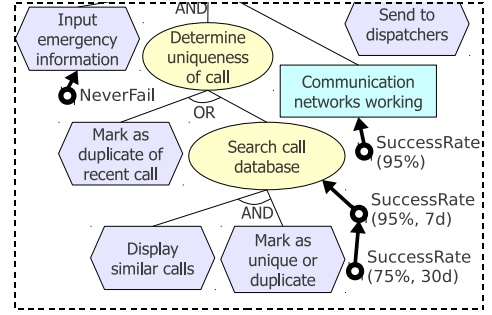


Figure 5: Portion of the ADS goal model showing the graphical representation of *AwReqs* AR1, AR2, AR3 and meta-*AwReq* AR9

demonstrate this is true for scenarios of the ADS.

These evaluation methods represent the experimental and descriptive evaluation methods of Design Science, as enumerated by [15].

4.1 Monitoring *AwReq* Patterns

Table 2 illustrates how each of the patterns of table 1 can be expressed in OCL_{TM} . These formulations are consistent with that shown in figure 4. The definitions and invariants are placed in the context of UML classes that represent requirements (see §3.2). 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 are 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 runtime 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 runtime 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 runtime evaluation (`satisfied()`), as well as its design-time properties (e.g., `p1.name()`). Therefore, in OCL_{TM} , requirements can refer to the design-time and runtime properties of requirements. 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.

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 section 3.1.

4.2 Evaluating an AwReq Scenario

The requirements of the Ambulance Dispatch System (ADS) provide a context to evaluate the *AwReq* framework. The ADS is implemented in Java. Its requirements (see §3.1) are represented as OCL_{TM} properties, using patterns like those presented in table 2 and figure 4. 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 (a.k.a. **TInputInfo**) 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, **TInputInfo** is implemented as a XML form which is processed by a workflow engine. In our standard Java application, **TInputInfo** 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., a person performing the requirements monitoring) instrument the software. Five events are logged in this simple example: (a) **TInputInfo.start()**, (b) **TInputInfo.end()**, (c) **TInputInfo.success()**, (d) **TInputInfo.fail()**, and (e) **TInputInfo.cancel()**. Of course, the developers may have chosen a different name for **TInputInfo** or the five methods. In which case, the validator must introduce a mapping from the runtime 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 calculates the value of requirements. In the case of **AR1**, if the **TInputInfo** form is processed as succeed or cancel, then **AR1** is true.

The architecture and process of EEAT provides some context for the proceeding 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 user interface. However, the runtime operates as a OSGi application, comprised as a dynamic set of OSGi components. For these experiments, the EEAT runtime 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

Table 2: OCL_{TM} idioms for the patterns of table 1.

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)
SuccessRateExecutions(R, r, n)	def: stream: Sequence(OclMessage) = receiveMessages()->select(m methodName = 'succeed' or methodName = 'fail') def: msg: Sequence(OclMessage) = msgs->select(m indexOf(m) > stream.size() - n) -- see above def's for succeed and fail 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 runtime objects

them to the EEAT repository. For our experiments, the Java application is 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 [25, 26].

5. DISCUSSION

The previous section explained how EEAT was used to monitor requirements and provide feedback if *AwReqs* are successful or not. This is, however, only the first part of a MAPE feedback loop that operationalizes the system's adaptivity. To complete it, we must also implement analysis (diagnosis), planning and execution (of compensations). Here is how we propose this could be done.

Once we have observed that the system is not behaving the way it is meant to be, diagnosis consists on identifying the components which, when assumed to be functioning abnormally, will explain our observations [22]. In our case, once the monitoring component has identified the failure of an *AwReq*, the diagnosis component should tell us which part of our system is responsible for the failure in the requirements.

In a feedback loop, knowing this will help us determine the best compensation to be effected during the planning stage. Taking **AR1** once again as example, many different parts of the ADS could be responsible for the failure of task *Input*

emergency information, such as a database error, a network failure, system misuse by the operator, etc. Taking the *envelope of adaptability* (see §3) into account, all these possibilities and their respective compensation actions should also be elicited along with the *AwReqs*. GORE approaches could be adopted, using contexts (e.g. [19]) to describe possible diagnoses and modeling the compensation of a given *AwReq* failure as a goal that can be decomposed into sub-goals and tasks, and so forth.

This diagnosis component could also be implemented as an OSGi bundle integrated with EEAT, like the monitoring component presented in section 4.2. Instead of observing events such as the success and failure of goals and tasks, however, the diagnosis bundle's rule file would match log events produced by the monitoring bundle that indicate the failure of an *AwReq*.

As with the monitoring component, the diagnosis bundle also raises events which are ultimately fed into the planning component, which could also be implemented as an OSGi bundle. Given the part of our system that was responsible for the failure, this component should plan and ultimately execute a compensation that would, finally, operationalize adaptivity in the monitored system. Planning would take into consideration requirement models. Approaches such as finding an alternative path that could satisfy the same goal [31] or finding an external agent that can do it instead [9] could be applied. Finally, some kind of callback function from the feedback loop to the monitored system should be provided, as compensations are usually domain-specific and would be implemented as part of the system itself.

6. RELATED WORK

The Dagstuhl Seminar on “Software Engineering for Self-Adaptive Systems” [6] discussed the state-of-the-art and challenges in this area. Challenges proposed for Requirements Engineering include a new requirements language to deal with uncertainty, systematic methods for refining this new language into an architecture, requirements reflection and traceability from requirements to implementation [8]. Andersson et al. [3] consider that “a major challenge is to accommodate a systematic engineering approach that integrates both control-loop approaches with decentralized agents inspired approaches.” Brun et al. [5] notice that “while [some] research projects realized feedback systems, the actual feedback loops were hidden or abstracted. [...] With the proliferation of self-adaptive software systems it is imperative to develop theories, methods and tools around feedback loops.” We believe our proposal is a starting point to face these challenges.

A work with similar purpose to ours is the RELAX language by Whittle et al. [32]. RELAX aims at capturing uncertainty declaratively with modal, temporal, ordinal operators and uncertainty factors provided by the language, whose semantics are formalized in Fuzzy Branching Temporal Logic. Instead, we offer an extension to a graphical GORE language with a process aimed at coming up with high-level adaptive architecture based on feedback loops, at identifying adaptation situations and at inferring feedback controller requirements from them. While RELAX aims at supporting unanticipated adaptations, our approach is targeting domains where predictability is important (e.g., business process management).

Proposals for self-adaptability have also come from the

Tropos research group. Morandini et al. [20] propose extensions to the architectural design phase of Tropos to model adaptive systems based on the Belief-Desire-Intention (BDI) model as a reference architecture. Qureshi & Perini [21] present a goal-based characterization of adaptive requirements that aids the analyst in modeling these kinds of requirements for a self-adaptive system. Dalpiaz et al. [9] propose an architecture that, based on requirements models, adds self-reconfiguring capabilities to a system using a monitor-diagnose-compensate loop. While our proposal is similar to these works in many aspects, it differs from them in that it promotes feedback loops to first-class citizens during Requirements Engineering, introducing a new class of requirements, *AwReqs*, which are requirements for feedback loops that provide adaptivity to a software system.

Schmitz et al. [29] uses goals to model the requirements of control systems and proposes a process to derive mathematical models from the requirements. Although our work also targets requirements for control loops, it is not restricted to control systems development and allows modeling of requirements for any adaptive system.

Other proposals in the literature, such as [17], focus at the architecture and design of self-adaptive system. While our proposal focuses on the requirements, the state-of-the-art on self-adaptive architecture could play an important role in the future steps of our research.

7. CONCLUSION

The main contribution of this paper is the definition of a new class of requirements that impose constraint on the runtime behavior of other requirements. The technical details of the contribution include a language for expressing such requirements (OCL_{TM}), a methodology for generating feedback from such requirements, as well as fragments of a prototype implementation founded on an existing requirements monitoring framework.

For future research, we propose to complete and further evaluate the EEAT implementation as presented in section 5. Also, to extend the class of *AwReqs* to include adaptivity requirements, such as “If requirement r fails more than N times over a time period, relax it”. Expressing such requirements calls for some extensions to the OCL_{TM} language. This line of research is closely related to the RELAX proposal [32], though the technical details of our approach are very different.

8. REFERENCES

- [1] Object constraint language, omg available specification, version 2.0, <http://www.omg.org/cgi-bin/doc?formal/2006-05-01>, 2006.
- [2] *Proceedings of the 6th international conference on Autonomic computing*, Barcelona, Spain, June 2009.
- [3] J. Andersson, R. de Lemos, S. Malek, and D. Weyns. *Modeling Dimensions of Self-Adaptive Software Systems*, volume 5525/2009, pages 27–47. Springer-Verlag, Berlin, Heidelberg, 2009.
- [4] D. M. Berry, B. H. C. Cheng, and J. Zhang. The four levels of requirements engineering for and in dynamic adaptive systems. In *REFSQ '05: Proceedings of the 11th International Workshop on Requirements Engineering: Foundation for Software Quality*, pages 95–100, Porto, Portugal, 2005.

- [5] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. *Engineering Self-Adaptive Systems through Feedback Loops*, volume 5525/2009, pages 48–70. Springer-Verlag, Berlin, Heidelberg, 2009.
- [6] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors. *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
- [7] B. Cheng et al. Seams 2009: Software engineering for adaptive and self-managing systems. In *Proceedings of the 2009 31st International Conference on Software Engineering: Companion Volume*, pages 463–464, Washington, DC, USA, 2009. IEEE Computer Society.
- [8] B. Cheng et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. *Software Engineering for Self-Adaptive Systems*, 5525/2009:1–26, 2009.
- [9] F. Dalpiaz, P. Giorgini, and J. Mylopoulos. An Architecture for Requirements-Driven Self-reconfiguration. *Advanced Information Systems Engineering*, 5565/2009:246–260, 2009.
- [10] S. Dobson, F. Zambonelli, S. Denazis, A. Fernández, D. Gaiiti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, and N. Schmidt. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems*, 1(2):223–259, 2006.
- [11] J. Doyle, B. Francis, and A. Tannenbaum. *Feedback Control Theory*. McMillan Publishing, 1990.
- [12] 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.
- [13] S. Flake. Enhancing the Message Concept of the Object Constraint Language. In F. Maurer and G. Ruhe, editors, *SEKE '04: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, pages 161–166, Banff, Canada, 2004.
- [14] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.
- [15] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.
- [16] 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.
- [17] J. Kramer and J. Magee. A Rigorous Architectural Approach to Adaptive Software Engineering. *Journal of Computer Science and Technology*, 24:183–188, 2009.
- [18] A. Lapouchnian. Goal-oriented requirements engineering: An overview of the current research. Technical Report <http://www.cs.toronto.edu/~alexei/pub/Lapouchnian-Depth.pdf>, Univ. of Toronto, 2005.
- [19] A. Lapouchnian and J. Mylopoulos. Modeling domain variability in requirements engineering with contexts. In *ER '09: Proceedings of the 28th International Conference on Conceptual Modeling*, pages 115–130, Gramado, Brazil, 2009. Springer.
- [20] M. Morandini, L. Penserini, and A. Perini. Towards goal-oriented development of self-adaptive systems. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 9–16, Leipzig, Germany, 2008. ACM Press.
- [21] N. A. Qureshi and A. Perini. Engineering adaptive requirements. In *SEAMS '09: 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 126–131. IEEE, 2009.
- [22] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, Apr. 1987.
- [23] W. N. Robinson. A requirements monitoring framework for enterprise systems. *Requirements Engineering*, 11(1):17–41, Mar. 2006.
- [24] 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.
- [25] W. N. Robinson and S. Fickas. Designs can talk: A case of feedback for design evolution in assistive technology. In W. Aalst, J. Mylopoulos, N. M. Sadeh, M. J. Shaw, C. Szyperski, K. Lyytinen, P. Loucopoulos, J. Mylopoulos, and B. Robinson, editors, *Design Requirements Engineering: A Ten-Year Perspective*, volume 14 of *Lecture Notes in Business Information Processing*, pages 215–237. Springer Berlin Heidelberg, 2009.
- [26] W. N. Robinson and S. Puro. Monitoring service systems from a language-action perspective. *IEEE Transactions on Services Computing*, 2010.
- [27] C. Rohleder, J. Smith, and J. Dix. Requirements specification - ambulance dispatch system. Available online: <http://www.utdallas.edu/~cjr041000/> (last access: March 4th, 2010), Feb. 2006.
- [28] D. Rosenthal. *Consciousness and Mind*. Oxford University Press, USA, Jan. 2006.
- [29] D. Schmitz, M. Zhang, T. Rose, M. Jarke, A. Polzer, J. Palczynski, S. Kowalewski, and M. Reke. *Mapping Requirement Models to Mathematical Models in Control System Development*, volume 5562/2009 of *Lecture Notes in Computer Science*, pages 253–264. Springer-Verlag, Berlin / Heidelberg, June 2009.
- [30] R. Sebastiani, P. Giorgini, and J. Mylopoulos. Simple and minimum-cost satisfiability for goal models. *Advanced Information Systems Engineering*, 3084/2004:675–693, 2004.
- [31] Y. Wang and J. Mylopoulos. Self-repair Through Reconfiguration: A Requirements Engineering Approach. In *ASE '09: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, Auckland, New Zealand, 2009.
- [32] 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.