A Language for Self-Adaptive System Requirements

Jon Whittle, Pete Sawyer, Nelly Bencomo
Computing Department,
InfoLab21,
Lancaster University,
Lancaster LA1 4AW, UK

Betty H.C. Cheng
Department of Computer Science and Engineering,
Michigan State University,
East Lansing, MI 48824, USA

ABSTRACT

Self-adaptive systems have the capability to autonomously modify their behaviour at run-time in response to changes in their environment. Such systems are now commonly built in domains as diverse as enterprise computing, automotive control systems, and environmental monitoring systems. To date, however, there has been limited attention paid to how to engineer requirements for such systems. As a result, self-adaptivity is often constructed in an ad-hoc manner. In this paper, we argue that a more rigorous treatment of requirements relating to self-adaptivity is needed and that, in particular, requirements languages for self-adaptive systems should include explicit constructs for specifying and dealing with the uncertainty inherent in self-adaptive systems. We present some initial thoughts on a new requirements language for self-adaptive systems and illustrate it using examples from the services domain.

1. Introduction

A system has goals that must be satisfied and, whether these goals are explicitly identified or not, system requirements should be formulated to guarantee goal satisfaction. This fundamental principle has served systems development well for several decades but is founded on an assumption that goals are fixed. In general, goals can remain fixed if the environment in which the system operates is stable. Hence, for example, using conventional development techniques and systems infrastructures, banking systems can be developed on the assumption that the fundamental characteristics of the financial services industry remain stable. However, in the 1980s, deregulation caused fundamental changes to the UK financial services industry. So significant were these environment changes that UK banking systems had new goals. Many existing systems had to undergo costly changes and completely new systems had to be developed to adapt to the new environment. Deregulation of the UK financial services industry caused fundamental changes to banks' operating environments but this change took place over a period of time that was sufficiently long to allow developers to make the necessary system adaptations. Increasingly, there is a demand for systems whose environment changes at a rate that necessitates the system to adapt in real time and with minimal intervention of developers.

As an example, a mobile device may need the ability to adapt in order to take advantage of new services as they come in range and become available. The goals of such a system may remain constant (e.g., to provide users with access to communication and information services) but subtle trade-offs in how they are satisfied may be necessary. For example, the choice of service to use may be constrained by a preference for certain service providers that may not always be available. The need for such trade-offs poses significant challenges for system developers but it is also possible to envisage systems where the environment is so volatile that the goals themselves change, and change so fast that systems need to adapt at run-time. (We acknowledge, however, that at the highest level of abstraction, invariant goals exist that indicate the fundamental services that must be provided by an adaptive system.)

Consider, for example, an autonomous vehicle designed for use in hazardous environments. One of its goals is to extinguish fires. It is a valuable vehicle so it also has a goal to preserve its capability to fight future fires, perhaps specified as (amongst others) a requirement that the temperature of the vehicle's casing must be maintained below some threshold value. If operating at the scene of a chemical fire (say) where the fire is in danger of reaching a tank of explosive chemicals, however, the second goal may have to be relaxed or disappear. Hence, in such circumstances, it may be better to allow the vehicle to remain in position spreading fire-suppressant chemicals for longer than guarantees the vehicle's survival, if this offers the best chance of avoiding an explosion. Note that there is an implicit trade-off between the two goals and it is easy to hypothesise circumstances where, even in non-emergency situations, the state of the environment might mean that achieving both goals was infeasible.

Quite aside from the obvious challenges of implementing such a self-adaptive system (although great strides in developing adaptive infrastructures have been made in recent years [9]), it is difficult to specify requirements to satisfy goals that:

- may change in the sense that their priority in relation to other goals may evolve;
- may disappear, as in the case of the self-preservation goal above;
- may have been unanticipated by the analyst.

Clearly, this final category is beyond the capabilities of existing technologies, yet it is not outside the scope of the vision of fully autonomic computing such as that articulated in [6]. Existing requirements techniques are unsuited even to the expression of the first two classes of goal. Mandating goal or requirement satisfaction using the traditional "shall" (the vehicle *shall* ensure its own survival) is unhelpful because it does not allow requirements to be relaxed. Simply picking a different modal verb (the vehicle *should* ensure its own survival) doesn't help much. Fit Criteria [11] may be used to define different levels of requirement *satisficement* but their principal role is to define how to verify a requirement. It is important to retain this crucial role for fit criteria and not conflate it with the separate issue of the specification of uncertainty. Traditional behavioural modelling techniques are poorly suited also, although notations such as i* [15], that allow the analyst to model NFR (non-functional) trade-offs go some way to help. However, these typically support only enumeration of alternative goals that are known at design time.

This paper makes the first steps towards defining a requirements language that addresses the first two problems identified above. We sketch out a vocabulary that allows analysts to mark requirements that may be relaxed at run-time and embody this in a requirements engineering language, RELAX. Our ultimate aim is to address uncertainty in requirements to support self-adaptive systems development, in a way such that the uncertainty can be specified declaratively rather than by simply enumerating all alternative goals. Doing so will enable run-time adaptation modules to reason about goal satisfaction at run-time in such a way that critical goals are never jeopardised but non-critical goals may be left unsatisfied temporarily. The paper also outlines a process for translating traditional requirements into RELAX requirements. This process supports requirements engineers who must determine points of flexibility in their requirements.

The paper is structured as follows. Section 2 outlines RELAX. Section 3 applies it to an example from the smart office domain. Section 4 describes related work and is followed by conclusions and a discussion of further work.

2. RELAX: a Requirements Engineering Language for Self-Adaptive Systems

In this section, we present our initial work towards a requirements engineering language, RELAX, for self-adaptive systems. RELAX is based on two fundamental principles:

- In practice, most requirements are written as textual documents.
- Requirements for self-adaptive systems should build in flexibility in the way that goals are satisfied or traded-off against each other.

As a result, RELAX is a textual requirements language that includes a vocabulary for identifying explicit points where flexibility is allowed. Furthermore, since understanding where points of flexibility are allowed (or disallowed) is generally difficult, we propose a requirements process that guides engineers in modifying a set of traditional requirements into a set of RELAX requirements.

2.1 Vocabulary for Identifying Flexibility

Typically, textual requirements prescribe behavior using a modal verb such as *SHALL (*or *WILL)* that defines actions or functionality that a software system must always provide. For self-adaptive systems, however, such statements are overly prescriptive and requirements should instead mark explicit points where the system is free to trade-off or relax requirements. We propose a specific vocabulary for expressing these kinds of requirements. This vocabulary enables requirements engineers to explicitly identify requirements that should never change as well as requirements that may change under certain conditions. Our vocabulary currently includes the following keywords.

2.1.1 Modal verb:

SHALL – we retain the conventional modal verb for expressing a requirement. This is because, even in self-adaptive systems, some requirements are forever invariant.

However, for a requirement that contributes to the satisfaction of goals that may be temporarily left unsatisfied, the inclusion of a temporal or ordinal RELAX-*ation* modifier will define the requirement as RELAX-*able*. In such a case, the requirement statement should also define the conditions for RELAX-ation using the MONITOR and ENVIRONMENT keywords (which are explained under Section 2.1.4).

2.1.2 Temporal RELAX-ation

We currently offer three temporal modifiers, as follows.

AS CLOSE AS POSSIBLE TO *<frequency>* – expresses a requirement that something occurs repeatedly but the frequency may be relaxed. The ideal frequency of occurrence is defined but the modifier allows one to represent uncertainty about the frequency and which conditions in the environment will permit frequency relaxation. The requirement will be completely satisfied if the actual occurrence is periodic and at the ideal frequency. If this is unachievable, however, the system should ensure that the event occurs periodically at a frequency that is as close to the ideal as conditions permit, or aperiodically at a mean frequency that is as close as is achievable to the ideal. The clause implicitly mandates the system to opportunistically adapt in order to achieve as close to the ideal frequency and as close to being periodic as is feasible.

AS EARLY AS POSSIBLE AFTER <event> – expresses a requirement that an event occurs immediately after the defined event or, if that is not possible, with minimal delay. Again, implicit in the clause is the requirement that the system adapts to minimize the delay whenever an opportunity to do so occurs.

EVENTUALLY – expresses the requirement that something must occur. How soon it occurs is less important but eventual occurrence must be guaranteed, by adaptation of the system if necessary.

2.1.3 Ordinal RELAX-ation

RELAX also supports the following ordinal modifier.

AS {MANY|FEW} <subject> AS POSSIBLE – permits one to mandate that the system maximizes or minimizes some occurrence. Again, opportunistic adaptation is implicitly required of the system.

2.1.4 Alternative RELAX-ation

A straightforward case is to enumerate alternatives, any of which might satisfy a goal (and the system may choose autonomously):

MAY <behaviour> OR <behaviour>

2.1.5 Defining the conditions for RELAX-ation

RELAX-*able* requirements should also define conditions for when they can be relaxed. These conditions give guidance to the self-adaptive system as to when and how adaptation decisions can be taken. Each RELAX-*able* requirement specifies two additional pieces of information labeled by the following two condition keywords.

MONITOR <property$^+$> – defines the set of properties that need to be monitored by the executing system in order to evaluate the state of the environment.

ENVIRONMENT <property$^+$> – represents a projection on or a viewpoint of the set of properties that define the system's environment, and that are needed to evaluate the temporal or ordinal RELAX-*ation* conditions.

The MONITOR and ENVIRONMENT properties will be the same in many cases. Since ENVIRONMENT captures the "state of the world" and MONITOR defines properties to be monitored, then if the state can be monitored directly, ENVIRONMENT and MONITOR will coincide. Often, however, environmental variables cannot be monitored directly because, for example, they are not observable, and so MONITORable properties that provide evidence on the state must be monitored instead.

We make no claim that the RELAX vocabulary is complete. Rather, we have so far identified the temporal, ordinal and alternative categories of RELAX-*ation* modifiers. It is probable that others will be needed for some problem domains. The set of keywords identified here represents our first attempt to define a useful RELAX vocabulary, based on a number of example systems studied thus far.

2.2 Process for Deriving RELAX Requirements from Traditional Requirements

Defining a vocabulary for specifying flexible requirements is but the first step towards a requirements methodology for self-adaptive systems. Identifying points of flexibility is itself a difficult task. We therefore propose a process for developing RELAX requirements based on an existing set of traditional requirements in the form of SHALL statements. This process is based on the assumption that those requirements for which we will accept sub-optimal satisfaction are not somehow 'obvious'. Rather, we expect requirements engineers to be able to write down SHALL statements fairly easily, using standard elicitation techniques. The requirements engineer should then carefully examine each SHALL statement to see if there are opportunities to relax it and, indeed, if relaxation should be allowed. The steps in the process are given below and shown in Figure 1:

1. Derive a set of requirements expressed in the traditional way, by making every requirement implicitly mandatory using the modal verb SHALL.
2. For each SHALL statement, consider whether it must always be satisfied no matter what, or whether it could be relaxed under certain circumstances. In the former case, leave the SHALL statement as is. In the latter case, consider how well can it be satisfied under different conditions and at what cost to the available computational or data resources. Requirements for which sub-optimal satisfaction is acceptable (*RELAX-able* requirements) imply that some form of adaptation may be necessary to make the best use of the available resources according to the degree of satisfaction that is acceptable. Replace the SHALL with the appropriate RELAX modifier.
3. For each RELAX-*ed* SHALL statement, analyse the environment, with the express purpose of identifying environmental volatility that is likely to be manifested at run-time and that is likely to lead to relaxing the satisfaction of the requirement. The goal of this step is to identify characteristics of the environment which will determine whether or not the requirement should be relaxed. Ultimately, the environment states may be discrete (as in the case of *domains* in [4]) or continuous. Capture the relevant characteristics of the environment using the RELAX ENVIRONMENT keyword.
4. For each RELAX-*ed* SHALL statement, identify the observable properties of the environment which determine whether or not to relax the requirement. Capture these using the MONITOR keyword. As stated earlier, ENVIRONMENT and MONITOR will coincide except in the case when environmental properties cannot be directly sensed.

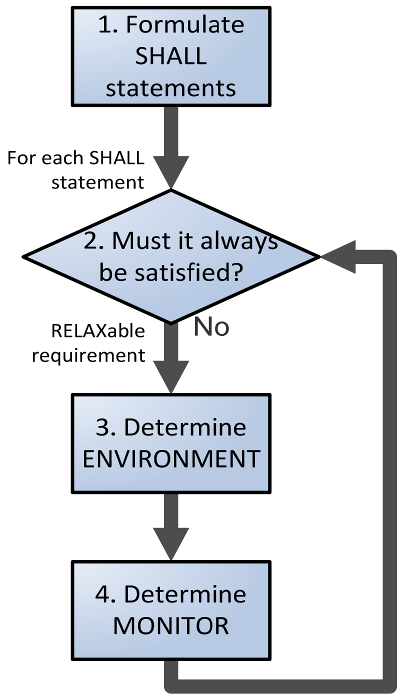

Figure 1: RELAX Process.

Note that implicit in this process is the likelihood that trade-offs in requirement satisfaction will be necessary. It is inevitable, therefore that several passes through the requirements will be necessary to establish what those trade-offs should be.

Note also that the process describes a way of incrementally building up a model of the environment. This is in contrast to including an explicit task to model the environment. The latter is difficult in practice because it is hard to understand which environmental factors might be relevant. Following the RELAX process, the relevant environmental characteristics are driven by the requirements, and, in particular, the relaxable requirements.

3. Example

We illustrate RELAX with an example from the smart office domain. The following scenario gives the context of the example:

Alice stores her personal office data using a number of handheld and fixed computing devices. She carries two PDAs with her: a Blackberry, which is principally used for business contacts, and an iPhone, which is mainly for personal contacts. In addition, Alice's desktop computer in her office maintains a business contact database and Alice's business partner, Joe, also maintains business contacts on both his desktop (in an adjacent office) and a set of PDAs. Alice's office is a state-of-the-art smart room, which detects Alice's arrival every morning and initiates a data synchronization process to ensure that Alice's Blackberry, iPhone and desktop, and Joe's desktop, all maintain a consistent list of business contacts. This synchronization process is repeated every 30 minutes as long as Alice is in the room. (Note that only business contacts which are also personal contacts are stored on Alice's iPhone.) The synchronization process is driven by Alice's desktop, which acts as a centralized controller for this process. The smart office environment, therefore, ensures data integrity and consistency at all times, enabling Alice to maximize her productivity without danger of losing important contact information.

In addition to this scenario, new devices (e.g., new PDAs) may be added at any time. This is done by a connection procedure initiated by Alice. Similarly, Alice may disconnect devices. Finally, Alice may act as an administrator and may abort the synchronization process if desired. This can be initiated from any of the connected devices.

Given the task of deriving requirements for this smart office environment, a traditional requirements engineering process might result in the list of requirements given in Table 1.

This set of requirements represents the ideal situation. Note the use of the SHALL keywords to prescribe behaviours that must be present in the final system. Given these requirements, a designer might implement the synchronization process as a two-phase commit protocol which would distribute data to all connected devices, except in the case of failure, in which case the system would roll back so that devices use a previous version of the data consistently.

The designers of the smart room, however, would like to build in self-adaptivity to make the system more flexible in an uncertain environment. For example, network outages or device malfunctions could mean that it may not always be possible to consistently synchronize all devices. In this case, instead of rolling back (which may result in Alice missing important new business contacts), the system might be able to find another way of reaching a malfunctioning device (e.g., by communicating via a neighboring PDA or other networking medium, such as Bluetooth).

Table 1: Traditional Requirements for Alice's Smart Office.

Synchronization
S1: The synchronization process SHALL be initiated when Alice enters the room and at 30 minute intervals thereafter.
S2: The synchronization process SHALL distribute data to all connected devices in such a way that all devices are using the same data at all times.
Connection
C1: An authorized device SHALL be allowed to connect at any time.
C2: Once connected, current data SHALL be distributed to the device immediately.
Disconnection
D1: A device SHALL be allowed to disconnect at any time.
Aborting
A1: An administrator SHALL be allowed to abort data updates at any time. The system rolls back to the previously used data.

Of course, a requirements engineer could make an analysis of the existing requirements and derive specific instances where adaptivity, such as the example given above, might be desired. In such a case, one could easily reformulate the requirements. For example, S2 could be modified to the following statement:

> S2: The synchronization process SHALL distribute data to all connected devices in such a way that all devices are using the same data at all times. If a device is malfunctioning, synchronization SHALL be carried out by communication with a neighbouring device.

The problem with this approach is that the requirements engineer must enumerate all possible points where adaptivity might be required. The result, in effect, would be a tree of alternative requirements, where each path through the tree defines a possible behaviour of the system. In particular, this approach would not allow for unanticipated adaptations because possible behaviours are only those predefined by the set of enumerations.

Instead, the RELAX process allows for specific points of flexibility or uncertainty to be identified, but does not mandate a discrete set of alternatives. In this way, potentially unanticipated adaptations are allowed, as long as they conform to the declaratively specified flexibilities in the requirements.

We continue with the smart office example and show how to apply the RELAX process to incorporate explicit flexibilities into the requirements in Table 1. In essence, the process systematically examines each requirement and asks under which environmental conditions the requirement might not be satisfiable. For each such environmental condition, the requirements engineer should then ask: (i) Does it matter that the requirement cannot be satisfied? (ii) Is adaptation required to enable satisfaction of the requirement? If (ii), then the requirement is augmented to use the RELAX vocabulary and to include aspects to MONITOR and aspects of the ENVIRONMENT, as discussed in Section 2.1.

To illustrate, consider requirement S1. Now imagine that the requirement cannot be satisfied for some reason – perhaps, communication links are broken, or perhaps the smart office system is redeployed in a different environment where devices have different characteristics. In either case, synchronization may not be possible every 30 minutes. Following RELAX, we modify S1 to a new requirement S1' as follows:

S1': The synchronization process SHALL be initiated when Alice enters the room and AS CLOSE AS POSSIBLE TO 30 minute intervals thereafter. MONITOR: actual synchronization intervals. ENVIRONMENT: connection between devices that might adversely affect synchronization interval.

The MONITOR slot here specifies quantities that should be monitored by a design solution so that the inherent flexibility in the requirement can be achieved. In this case, the system would need to monitor how close the synchronization intervals are to 30 minutes and, if they repeatedly go beyond this threshold, it would need to choose an alternative design. Note that the requirement does not explicitly list alternative solutions and so a run-time adaptation module could even download new design solutions on-the-fly. The ENVIRONMENT slot determines aspects of the environment that affect satisfaction of the requirement. Taken as a whole, across all requirements, the ENVIRONMENT descriptions define a model of the environment relevant for the self-adaptive system.

In this example, the MONITOR and ENVIRONMENT descriptions are the same. However, this need not be the case. Generally, MONITOR defines variables that can be directly observed. ENVIRONMENT defines contextual characteristics that may not be possible to observe directly. Hence, there is an analogy with control systems in which sensors may only be able to measure particular variables but try to estimate true values for non-observable variables related to the measured variables in some way. (For example, estimate the position of an aircraft by measuring the distance from known reference points.)

Table 2 gives the full set of modified requirements for the smart office application. Consider the RELAX requirement for S2:

> S2': The synchronization process SHALL distribute data to all connected devices in such a way that AS MANY devices AS POSSIBLE are using the same data at all times. Those devices not updated SHALL know it. EVENTUALLY, all devices should use the same data. MONITOR: number of non-updated devices. ENVIRONMENT: number of consistent devices.

S2' in fact supports a high degree of flexibility that goes well beyond the original requirements. It is up to the requirements engineer, of course, to decide if such flexibility is really desired. S2' makes use of two RELAX keywords – AS MANY AS and EVENTUALLY – to specify that temporary inconsistencies can be tolerated.

We briefly comment on the remaining RELAX requirements in Table 2. C1' and D1' both relax the immediacy constraint on connection or disconnection of devices. To keep track of these requirements, the average device (dis)connection times should be monitored. C2' is similar but concerns relaxation of constraints on distributing data to new devices. A1 has been left as is to illustrate that it is not mandatory to relax requirements. It is perfectly acceptable, if the analyst deems it so, to mandate strict requirements that should never be made more flexible – that is, an invariant of the system.

Table 2: RELAX Requirements for Alice's Smart Office.

Synchronization
S1': The synchronization process SHALL be initiated when Alice enters the room and AS CLOSE AS POSSIBLE TO 30 minute intervals thereafter. MONITOR: synchronization interval at each iteration. ENVIRONMENT: synchronization interval.
S2': The synchronization process SHALL distribute data to all connected devices in such a way that AS MANY devices AS POSSIBLE are using the same data at all times. Those devices not updated SHALL know it. EVENTUALLY, all devices should use the same data. MONITOR: number of non-updated devices. ENVIRONMENT: number of consistent devices.
Connection
C1': A device SHALL be allowed to connect AS EARLY AS POSSIBLE after it requests it. MONITOR: average device connection times. ENVIRONMENT: number of devices requesting a connection.
C2': Once connected, current data SHALL be distributed to the device AS EARLY AS POSSIBLE. MONITOR: average device data update times. ENVIRONMENT: number of devices requesting a connection.
Disconnection
D1': A device SHALL be allowed to disconnect AS EARLY AS POSSIBLE after it requests it. MONITOR: average device disconnection times. ENVIRONMENT: number of devices requesting disconnection.
Aborting
A1: An administrator SHALL be allowed to abort data updates at any time. The system rolls back to the previously used data.

3.1 Towards Flexible Designs

RELAX requirements make no assumptions about how to satisfy them in a design. RELAX requirements should be mapped into a set of alternative designs between which the run-time system can choose. The challenge is to structure the design alternatives in such a way that design choices can be composed easily and that supports the maximum degree of flexibility. Existing methods from component-based design and/or software product lines could be used for this purpose, as well as techniques used for structuring knowledge-based systems, such as [13] in which "open" components are described by preconditions and have open slots where their behavior can be altered. No matter which design strategy is employed, there needs to be a set of guidelines for mapping the RELAX vocabulary concepts down to a chosen structuring method (e.g., variation points in software product lines).

4. Related Work

There has been growing interest in the specification of requirements for self-adaptive systems (see, for example, [1,2]), particularly using goal based models [4,7,8,10,14]. One of the most prolific research efforts in this area is that undertaken by Mylopoulos et al. [7, 14, 8]. Goal models and feature models are used to specify possible behaviours of autonomic systems, emphasizing the self-configuration, self-healing, and self-optimization aspects present in self-adaptive systems. They use goal models to capture variability in the problem domain. Furthermore, they "*try to capture all the different ways the system's goals can be achieved in that domain*" [8]. This last objective is very different from our aim since all possible alternatives must be enumerated. Hence, unanticipated adaptations are not possible. However, the approaches are complementary as the goal models work cited above, including our own work [4], can be leveraged and benefit from the RELAX specifications proposed here.

A related topic is run-time monitoring of the environment to assess requirements conformance (e.g., [3]). An important contribution in this direction is the ReqMon framework [1]. ReqMon is used by analysts in the development of requirements monitors in the domain of enterprise services. Using ReqMon, the system can send a warning when the system has failed to satisfy a specified requirement. ReqMon seeks to define a language for requirements and definition of monitors, and analyze monitor feedback. Complementary to this work, we could map RELAX requirements to monitors specified using the monitor specification language provided by ReqMon. Such a combination of efforts could be a step towards enabling a self adaptive system with run-time awareness of requirements (a notion called "requirements reflection" by Finkelstein [2]).

5. Conclusions

This paper presented initial work defining RELAX, a new language for the requirements specification of self-adaptive systems. The RELAX philosophy and the vocabulary specified so far explicitly acknowledges the need to deal with the levels of uncertainty, which are unavoidable when introducing self-adaptation capabilities to systems. To illustrate the viability and benefits of languages like RELAX we have presented a realistic example in the service-oriented domain.

RELAX allows analysts to specify "incomplete" systems, where "incomplete" here does not imply poor specifications, but instead acknowledges the possibility of a valid, yet unknown, specific behaviour.

Future work will continue to expand the vocabulary of RELAX based on a number of industry case studies. We also plan to explore the integration of RELAX specifications with goal-based approaches that offer principled ways to structure the different design alternatives that we speculate can be derived from RELAX requirements. Furthermore, our vision is to rely on run-time infrastructures that could support the realization of those designs. More work is needed to support increasingly sophisticated monitoring needs, such as adaptive monitoring to respond to the changing environmental and system conditions. Finally, we will explore the use of probabilistic logics [5] and multi-valued logics to define the semantics of the RELAX language.

6. References

[1] D.M. Berry, B.H.C. Cheng, and J. Zhang. The four levels of requirements engineering for and in dynamic adaptive systems. In *11th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'05)*, Porto, Portugal, 2005.

[2] B. H. C. Cheng, J. Whittle, A. Finkelstein, N. Bencomo, J. Magee, J.Kramer, S. Park, and S. Dustdar. Requirements engineering section of software engineering for self-adaptive systems: A research road map. 2008.

[3] S. Fickas and M.S. Feather. Requirements monitoring in dynamic environments. In *Second IEEE International Symposium on Requirements Engineering (RE'95)*, 1995.

[4] H. J. Goldsby, P. Sawyer, N. Bencomo, D. Hughes, and B. H.C. Cheng. Goal-based modeling of dynamically adaptive system requirements. In *15th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS)*, 2008.

[5] L. Grunske. Specification patterns for probabilistic quality properties. In *Proceedings of the 30th international Conference on Software Engineering, ICSE '08*, Leipzig, Germany, 2008.

[6] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *IEEE Computer 36(1), (2003)*, 36(1), 2003.

[7] A. Lapouchnian, S. Liaskos, J. Mylopoulos, and Y. Yu. Towards requirements-driven autonomic systems design. In *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, 2005.

[8] A. Lapouchnian, Y. Yu, S. Liaskos, and J. Mylopoulos. Requirements-driven design of autonomic application software. In *Proceedings of CASCON 2006*, 2006.

[9] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H.C. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, 2004.

[10] 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*, 2008.

[11] S. Robertson and J. Robertson. *Mastering the Requirements Process*. Addison-Wesley, 1999.

[12] W. Robinson. A requirements monitoring framework for enterprise systems. *Requirements Engineering*, 11(1):17 – 41, 2005.

[13] J. Whittle and J. Schumann. Automating the implementation of kalman filter algorithms. *ACM Transactions on Mathematical Software*, 30(4):434–453, 2004.

[14] Y. Yijun, A. Lapouchnian, S. Liaskos, J. Mylopoulos, and J. Leite. *From Goals to High-Variability Software Design*, volume 4994. Springer Berlin / Heidelberg, 2008.

[15] E. S. K. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *RE '97: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97)*, Washington, DC, USA, 1997.