Noname manuscript No.
(will be inserted by the editor)

RELAX: A Language to Address Uncertainty in Self-Adaptive Systems Requirements

Received: October 18, 2009

Abstract Self-adaptive systems have the capability to autonomously modify their behavior at run-time in response to changes in their environment. Self-adaptation is particularly necessary for applications that must run continuously, even under adverse conditions and changing requirements; sample domains include automotive systems, telecommunications, and environmental monitoring systems. While a few techniques have been developed to support the monitoring and analysis of requirements for adaptive systems, limited attention has been paid to the actual creation and specification of requirements of self-adaptive systems. As a result, self-adaptivity is often constructed in an ad-hoc manner. In order to support the rigorous specification of adaptive systems requirements, this paper introduces RELAX, a new requirements language for self-adaptive systems that explicitly addresses uncertainty inherent in adaptive systems. We present the formal semantics for RELAX in terms of fuzzy logic, thus enabling a rigorous treatment of requirements that include uncertainty. RELAX enables developers to identify uncertainty in the requirements, thereby facilitating the design of systems that are, by definition, more flexible and amenable to adaptation in a systematic fashion. We illustrate the use of RELAX on smart home applications, including an adaptive assisted living system.

1 Introduction

As applications continue to grow in size, complexity, and heterogeneity, it becomes increasingly necessary for computing-based systems to dynamically self-adapt to changing environmental conditions. We call these systems *dynamically adaptive systems* (DASs). Example applications that require DAS capabilities include automotive systems, telecommunication systems, environmental monitoring, and power grid management systems. The distributed nature of DASs and changing environmental factors (including human interaction) make it difficult to anticipate all the explicit states in which the system will be during its lifetime. As such, a DAS needs to be able to tolerate a range of environmental conditions and contexts, but the exact nature of these contexts remains imperfectly understood. One overarching challenge in developing DASs,

Address(es) of author(s) should be given

therefore, is how to handle the inherent *uncertainty* posed by the respective application domains. This paper presents RELAX, a new requirements language for DASs, where explicit constructs are included to handle uncertainty. We illustrate the use of RELAX on a number of examples from the adaptive smart home domain.

The need for DASs is typically due to two key sources of uncertainty. First is the uncertainty due to changing environmental conditions, such as sensor failures, noisy networks, malicious threats, and unexpected (human) input; we use the term *environmental uncertainty* to capture this class of uncertainty. IBM, for example, originally proposed the area of *autonomic computing* [16] to handle environmental uncertainty, thereby enabling computing-based systems to use high-level application goals and requirements to guide run-time self-management, including self-monitoring, self-healing, and self-configuration.

A second form of uncertainty is *behavioral uncertainty.* Whereas environmental uncertainty refers to maintaining the same requirements in unknown contexts, behavioral uncertainty refers to situations where the requirements themselves need to change. For example, the requirements of a space probe may change mid-flight in order to pursue science opportunities not foreseen by the designers. It is difficult to know all requirements changes at design time and, in particular, it may not be possible to enumerate all possible alternatives.

This paper describes a requirements language called RELAX that supports the explicit expression of uncertainty in requirements. We have designed the vocabulary of RELAX to enable analysts to identify requirements that may change at run time when the environment changes. Hence, RELAX addresses both environmental and behavioral uncertainty. The system might wish, for example, to temporarily ignore a non-critical requirement in an emergency situation in order to ensure that critical requirements can still be met. When specifying RELAX-ed system requirements, the execution environment that affects system behavior is explicitly identified, and the components that enable the system to monitor those environmental conditions are specified. RELAX supports a declarative style for specifying both sources of uncertainty, rather than by enumerating all alternative requirements. Doing so enables adaptation modules to reason about requirement satisfaction at run time in such a way that critical requirements are never jeopardized but non-critical requirements may be deferred or even left unsatisfied.

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. For *non-invariant requirements* (i.e., those requirements that may not have to be satisfied at all times), we use RELAX operators to introduce flexibility into *SHALL* statements of a system.[1] Several dimensions of flexibility are supported, including duration and frequency of system states; possible states of a system; and configurations for a system. While the RELAX specifications are in the form of structured natural language with Boolean expressions, the semantics for RELAX have been defined in terms of temporal fuzzy logic.

We illustrate the use of RELAX with a case study based on an assisted living scenario obtained from an industrial collaborator. The remainder of the paper is organized as follows. Section 2 introduces the RELAX language and uses a smart office appli-

[1] *SHALL* statements are commonly used to specify requirements, indicating a contractual relationship between the customer and the developer as to what functionality should be included in the system.

cation to illustrate its features. The grammar for RELAX is introduced in Section 3, and the formal semantics for RELAX defined in terms of fuzzy logic is also presented. Section 4 gives a process for creating RELAX specifications based on traditional requirements stated in terms of *SHALL* statements. A detailed description of how we used RELAX to specify the requirements for an adaptive assisted living smart home is given in Section 5. Section 6 relfects on the RELAX language and process, offerings lessons learned and directions for future work. Section 7 overviews related work, and we conclude in Section 8.

2 RELAX Overview

This section presents the RELAX language, leveraging and extending preliminary work [40, 41] that introduced the basic concepts for and elements of the language. RELAX takes the form of a structured natural language, including operators designed specifically to capture uncertainty. These operators are introduced in this section and their formal semantics is given in Section 3. This paper refines the operators, formalizes them, and applies them to a case study provided by an industrial collaborator.

The focus in RELAX is on structured natural language requirements. 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, environmental uncertainty may mean that it is not always possible to achieve all of these *SHALL* statements and trade-offs between *SHALL* statements may be necessary and tolerated to relax non-critical statements in favor of other, more critical ones. The RELAX operators are designed to enable requirements engineers to explicitly identify requirements that should never change (invariants) as well as requirements that a system could temporarily relax under certain conditions. RELAX can also be used to specify constraints on how these requirements can be relaxed.

2.1 RELAX Vocabulary

Table 2.1 gives the set of RELAX operators, organized into modal, temporal, and ordinal operators and uncertainty factors. Note that RELAX includes standard operators from temporal logic, such as *EVENTUALLY* and *UNTIL*. (We do not include a NEXT operator in this paper, but the underlying semantic model supports it.) The contribution of RELAX is in the operators that support uncertainty – namely, those that include the phrase "as possible".

We retain the conventional modal verb *SHALL* for expressing a requirement, but the introduction of RELAX operators offers more flexibility in how and when that functionality may be delivered. More specifically, for a requirement that contributes to the satisfaction of goals that may be temporarily left unsatisfied, the inclusion of an alternative, temporal or ordinal RELAX-ation modifier will define the requirement as RELAX-able.[2] For example, one can write "the system *SHALL* do something *AS EARLY AS POSSIBLE*".

[2] Note that we take the liberty to use the RELAX name as a verb to indicate the insertion of RELAX operators.

RELAX operator	**Description**
Modal Operators	
SHALL	a requirement must hold
MAY . . . OR	a requirement specifies one or more alternatives
Temporal Operators	
EVENTUALLY	a requirement must hold eventually
UNTIL	a requirement must hold until a future position
BEFORE, AFTER	a requirement must hold before or after a particular event
IN	a requirement must hold during a particular time interval
AS EARLY, LATE AS POSSIBLE	a requirement specifies something that should hold as soon as possible or should be delayed as long as possible
AS CLOSE AS POSSIBLE TO [frequency]	a requirement specifies something that happens repeatedly but the frequency may be relaxed
Ordinal Operators	
AS CLOSE AS POSSIBLE TO [quantity]	a requirement specifies a countable quantity but the exact count may be relaxed
AS MANY, FEW AS POSSIBLE	a requirement specifies a countable quantity but the exact count may be relaxed
Uncertainty Factors	
ENV	defines a set of properties that define the system's environment
MON	defines a set of properties that can be monitored by the system
REL	defines the relationship between the **ENV** and **MON** properties
DEP	identifies the dependencies between the (relaxed and invariant) requirements

Table 1 RELAX Operators

Each of the relaxation operators define constraints on how a requirement may be relaxed at run time. In addition, it is important to indicate what *uncertainty factors* warrant a relaxation of these requirements, thereby requiring adaptive behavior. This information is specified respectively using the **MON** (monitor), **ENV** (environment), **REL** (relationship), and **DEP** (dependency) keywords. The environment properties capture the "state of the world" – i.e., they are characteristics of the operating context of the system. Often, however, environmental properties cannot be monitored directly because they are not observable. The **MON** keyword is used to define those properties that are directly observable and that may contribute information towards determining the state of the environment. RELAX is intended to specify the software requirements, once hardware constraints have already been defined. For example, **MON** is used to delimit the properties to be monitored by physical sensors.

The **REL** keyword is used to specify in what way the observables (given by **MON**) can be used to derive information about the environment (as given by **ENV**). The distinction between **ENV** and **MON** is analogous to elements from the field of control theory wherein parameters to be estimated cannot necessarily be directly observed. For example, an aircraft equipped only with direction finding equipment cannot directly estimate its position. Rather, it can observe its distance from a set of known waypoints and must compute its position from these measurements. In our parlance, the aircraft position is a property of the environment, whereas the distances from waypoints are monitorables. **REL** would be used to define how to compute the position from the distance measurements. Finally requirements dependencies are delimited by **DEP**, as it is important to assess the impact on dependent requirements after RELAX-ing a given requirement.

2.2 Illustrative Example

We illustrate RELAX here using a simple example from the smart office domain:

Alice's office detects her arrival every morning and initiates a data synchronization process to ensure that Alice's Blackberry, iPhone, and 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.

Given the task of deriving requirements for this smart office environment, a traditional requirements engineering process might result in the two following *SHALL* statements.

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.

These requirements represent an ideal situation. Given these requirements, a designer might, for example, implement the synchronization process as a two-phase commit protocol that 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 data), the system might be able to find another way of reaching a malfunctioning device (e.g., by communication via a neighboring PDA or other networking medium, such as Bluetooth), or might temporarily tolerate inconsistent databases.

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-alt: 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 neighboring 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 behavior of the system for different environmental conditions. In particular, this approach would not allow for unanticipated environmental conditions and/or adaptations because possible behaviors are only those predefined by the set of enumerations.

Instead, RELAX can be used at development time to identify specific points of flexibility or uncertainty, but does not mandate a specific set of alternative requirements. 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 use RELAX to explicitly incorporate flexibilities into the requirements **S1** and **S2**. In essence, each requirement is examined to determine under which environmental conditions the requirement might not be satisfiable. For each such notable set of environmental conditions, the requirements engineer should then ask: (i) Is it essential for the requirement to be satisfied? If so, then the requirement is considered to be an *invariant* and should not be RELAX-ed. (ii) Is adaptivity required to enable satisfaction of the requirement? If (ii), then the requirement is augmented to use the RELAX vocabulary and to include environmental aspects to monitor.

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. We RELAX **S1** and **S2** to obtain requirements **S1'** and **S2'** as given below, where RELAX operators are italicized in all caps.

S1': The synchronization process *SHALL* be initiated *AS EARLY AS POSSIBLE AFTER* Alice enters the room and *AS CLOSE AS POSSIBLE TO* 30 minute intervals thereafter.
ENV: location of Alice; synchronization interval.
MON: motion sensors; network sensors
REL: motion sensors provide location of Alice; network sensors provide 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. *EVENTUALLY*, all devices *SHALL* use the same data.
ENV: number of consistent devices; time taken until consistency is reached.
MON: network sensors; device sensors
REL: network and device sensors provide number of consistent devices and time

S1' includes a characterization of the portion of the environment relevant to this requirement. For example, S1' requires that the system knows Alice's location and so her location is a key property of the environment. The **MON** information then defines how this environmental property can be monitored – in this case, by using motion sensors. The decision on this definition is made according to any design constraints imposed on the requirements engineer.

Consider the RELAX-ed requirement for **S2**. In fact, **S2'** supports a high degree of flexibility that goes well beyond the original requirements. Once the requirements engineer determines that indeed this level of flexibility can be tolerated, then the downstream developers, including the designers and programmers have the flexibility to incorporate the most suitable adaptive mechanisms to support the desired functionality. These decisions may be made at design time and/or run time [4, 18]. In this case, **S2'** makes use of two RELAX keywords – *AS MANY* and *EVENTUALLY* – to specify that temporary business contact data inconsistencies can be tolerated.

The RELAX-ed requirements declaratively specify (using the *SHALL* statements) where flexibility in the behavior is tolerated, and uses the keywords, **ENV**, **MON**, and **REL** to identify the parts of the environment that generate the uncertainty. By RELAX-ing the *SHALL* statements in this way, the requirements are less prescriptive and, in particular, give the flexibility for the run-time system to trade-off requirements when unknown situations are encountered at run time that were not known at design time.

3 RELAX Syntax and Semantics

This section presents a formal syntax and semantics for the RELAX language. The semantics is defined by mapping RELAX to fuzzy branching temporal logic (FBTL) [26].

We briefly motivate here the use of FBTL for formalizing the RELAX semantics. The use of a temporal logic is clearly required as many of the RELAX operators capture temporal information (e.g., *EVENTUALLY*, *UNTIL*, *BEFORE* and *AFTER*). In fact, RELAX relies upon standard operators of temporal logics, as will be seen below; for example, *EVENTUALLY* maps in a straightforward manner to the well-known future **F** operator (eventually, in the future, some property will hold on a given path of execution). But the decision to go beyond standard temporal logic and choose a fuzzy logic requires further elaboration.

Uncertainty is inherent in a RELAX specification. For example, the statement *AS EARLY AS POSSIBLE AFTER* e ϕ expresses the requirement that ϕ occurs after the occurrence of event e, but it is uncertain how much time it takes for ϕ to occur after event e has happened. The statement simply expresses a desire for the time period between the occurrences of e and ϕ to be as small as possible. A logic with built-in uncertainty is therefore necessary to formalize the RELAX semantics.

Note that probabilistic logics [1] do not capture our intended semantics. One could express a probability that ϕ becomes true within a specified time threshold after the occurrence of e; extending this, one could define a probability density function (pdf) that captures the probability of ϕ becoming true at each possible time threshold. However, this is not quite the semantics of *AS EARLY AS POSSIBLE*. The pdf represents the chance of ϕ occurring at a particular time point; it does not provide any way of deciding whether or not the time of occurrence of ϕ is "early enough". Put another way, given a set and uncertainty as to whether a variable is in the set, then fuzzy logic answers the question *how much* a variable is in the set, whereas probability theory answers the question *how probable* is it that a variable is in the set. In the latter case, the variable is either in the set or it is not. In the former case, the variable may be "roughly" in the set. Returning to the example of *AS EARLY AS POSSIBLE* , fuzzy logic allows one to define "as early as possible" as "roughly at zero time". In summary, fuzziness and probability are distinct yet complementary concepts [26].

3.1 Syntax

The syntax of RELAX expressions is defined by the grammar given below. Parameters of RELAX operators are typed as follows: p is an atomic proposition, e is an event, t is a time interval, f is a frequency and q is a quantity. An event is a notable occurrence that takes place at a particular instant in time. A time interval is any length of time bounded by two time instants. A frequency defines a number of occurrences of an event within a given time interval. If the number of occurrences is unspecified, then it is assumed to be one. A quantity is something measurable, that is, it can be enumerated. In particular, a RELAX expression ϕ is said to be quantifiable if and only if there exists a function Δ such that $\Delta(\phi)$ is a quantity.

A valid RELAX expression is any conjunction of statements $s_1; \ldots ; s_m$ where each s_i is generated by the following grammar.

$$\begin{aligned}
\phi := \;& \mathit{true} \mid \mathit{false} \mid p \mid \mathit{SHALL}\ \phi \\
\mid\;& \mathit{MAY}\ \phi_1\ \mathit{OR}\ \ \mathit{MAY}\ \phi_2 \\
\mid\;& \mathit{EVENTUALLY}\ \phi \ \mid \phi_1\ \mathit{UNTIL}\ \phi_2 \\
\mid\;& \mathit{BEFORE}\ e\ \phi \ \mid \mathit{AFTER}\ e\ \phi \ \mid \ \mathit{IN}\ t\ \phi \\
\mid\;& \mathit{AS\ CLOSE\ AS\ POSSIBLE\ TO}\ f\ \phi \\
\mid\;& \mathit{AS\ CLOSE\ AS\ POSSIBLE\ TO}\ q\ \phi \\
\mid\;& \mathit{AS}\ \{\mathit{EARLY,\ LATE,\ MANY,\ FEW}\}\ \mathit{AS\ POSSIBLE}\ \phi
\end{aligned}$$

It is straightforward to rewrite RELAX textual requirements in terms of this grammar, thereby making RELAX requirements amenable to tool support [17]. As an example, S2' from Section 2 would be represented as:
$\mathit{SHALL}\ (\ \mathit{AS\ MANY\ AS\ POSSIBLE}\ \ p);\ \mathit{EVENTUALLY}\ (\ \mathit{SHALL}\ q)$, where p denotes "distribute data to all connected devices", q denotes "all devices use the same data", and p is quantifiable with $\Delta(p)$ defined as the number of connected devices using the same data. This example illustrates the use of the Δ operator for quantification. It is a convenient way of capturing the fact that an expression is associated with a quantity – in this case, the number of connected devices.

3.2 Semantics

As mentioned previously, the semantics of RELAX expressions is defined in terms of fuzzy branching temporal logic (FBTL). FBTL can describe a branching temporal model with uncertain temporal and logical information.

A fuzzy set is a set whose elements have degrees of membership. In classical set theory, a member either belongs to a set or it does not. Fuzzy set theory permits the gradual assessment of membership of elements in a set, which is described using a membership function in the range of real numbers $[0, 1]$. In other words, a fuzzy set is a pair (A, m) where A is a set and $m : A \rightarrow [0, 1]$ captures the degree of membership of A.

A fuzzy number is a fuzzy subset of real numbers whose membership function is convex and normalized, i.e., $max(m(a)) = 1$. Typically, a fuzzy number is defined with a triangular membership function, in the sense that the membership graph describes a triangle with a vertex showing a degree of membership of value 1. For example, Figure 1 shows a fuzzy number two, which captures the fact that the precise value of the number is uncertain, or, in other words, that the number represents roughly two. The triangular membership function states that any value below 1.5 or above 2.5 is definitely not considered to be roughly 2, that 2.0 is absolutely considered to be roughly 2, whereas values in between 1.5 and 2.5 are roughly 2 with differing degrees of confidence. Note that the membership function need not be triangular but can be defined to fit the characteristics of the problem under study.

In FBTL, the concept of fuzzy number is extended to *fuzzy duration*. The duration $d \in \mathcal{R}^+$ is a fuzzy duration if there is fuzzy uncertainty about the exact length of the duration. That is, it is associated with a fuzzy number defining a fuzzy length of time. We can now define FBTL [26].

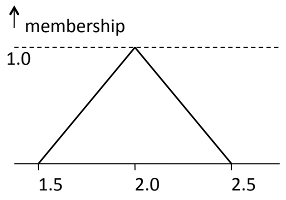

Fig. 1 Fuzzy numbers – "roughly 2".

Definition 1: Fuzzy Branching Temporal Logic (FBTL) has path formulae and state formulae defined inductively as follows.
The state formulae are defined as:

– if p is a proposition, then p is a state formula;
– if p and q are state formulae, then $\neg p$ and $p \wedge q$ are also state formulae;
– if p is a path formula, then $\mathbf{E}p$ and $\mathbf{A}p$ are state formulae.

And the path formulae are defined as:

– each state formula is also a path formula;
– if p and q are path formulae, then $\neg p$ and $p \wedge q$ are also path formulae;
– if p and q are path formulae, then $p\,\mathcal{U}q$ is also a path formula;
– if p is a path formula, then $\mathcal{X}_{Rd}p$ is also a path formula, where $R \in \{\leq, \geq, =, <, >\}$ and d is a normalized fuzzy duration on a time domain;

and the state formulae are the well-formed formulae of FBTL.

Path and state formulae are familiar from the usual definitions of branching temporal logics. Recall that, in branching temporal logic, state formulae quantify over states of the system, whereas path formulae quantify over possible execution paths of the system; that is, a chosen execution path defines a sequence of states through which the system passes. **E** and **A** are the usual "exist" and "all" operators, respectively. $\mathcal{U}$ denotes "until" as with standard temporal logic. $\mathcal{X}$, which takes the truth value of its formula after a time duration, is unique to FBTL and denotes a delay operator. The duration $d \in \mathcal{R}^+$ can be a fuzzy duration or a crisp (i.e., non-fuzzy) duration. The expression $\mathcal{X}_{=d}$ means "after exactly d"; $\mathcal{X}_{<d}$ represents "before d has passed"; and $\mathcal{X}_{>d}$ is "after d has passed". Therefore, if d is fuzzy, then the delay operator can be used to express relations with an uncertain time interval.

The shorthand notations customarily used in branching temporal logic are also available. In particular, $\mathbf{F}p =$ *true* $\mathcal{U}p$ and $\mathbf{G}p = \neg\mathbf{F}\neg p$, where **F** means eventually and **G** means always. Recall that in branching time logics, **E** and **A** quantify over execution paths, whereas **G** and **F** quantify over states within a given execution path.

For example, $\mathbf{G}p$ is true if p holds on the entire subsequent path, whereas $\mathbf{A}p$ is true if p holds on all paths starting from the current state. Highest binding power is given to the temporal operators $\mathbf{F}$ and $\mathbf{G}$ followed by $\mathcal{X}$ and $\mathcal{U}$. The logical operator $\neg$ is next, followed by $\wedge$. The formal semantics of FBTL is given in [26].

We are now ready to define the semantics of RELAX expressions in terms of FBTL. Table 2 gives the formal definitions using FBTL. The second column in the table gives an informal description of the meaning of each expression. The third column gives the interpretation as a FBTL formula.

We briefly explain the semantic definitions in Table 2. The first four entries are standard and so are not explained.

BEFORE and *AFTER* are defined in terms of the FBTL delay operator. In both cases, e_d defines a duration up until event e next occurs. Recall that a duration is a positive real number (i.e., a length of time). Hence, the formalization expresses that before this duration has passed (i.e., before e next occurs), ϕ holds. Similarly, for *AFTER*. Note that the duration in these cases is a crisp (i.e., non-fuzzy) duration; there is no uncertainty in these operators.

IN is defined in terms of *BEFORE* and *AFTER*. For ϕ to occur in a time interval, t, then it must occur either after the start of t or before the end of t. The start and end of the interval is defined using the events t_{start} and t_{end}, which define events capturing the beginning and end of the interval, respectively.

AS EARLY AS POSSIBLE relies on a fuzzy duration. The fuzzy logic definition formalizes the notion of "early enough". Intuitively, *AS EARLY AS POSSIBLE* is similar to *AFTER* except that ϕ should happen after the current point in time (rather than after an event e). Also, whereas *AFTER* is defined in terms of a crisp duration, *AS EARLY AS POSSIBLE* must be defined using a fuzzy duration because there is uncertainty about what constitutes "as early as possible". To capture this, a fuzzy duration is used whose membership function has its maximum value at 0 – i.e., at the current point in time. The membership function tails off gradually *ad infinitum* – i.e., it has a triangular membership graph that is asymptotic. This membership graph captures the meaning of *AS EARLY AS POSSIBLE* because if ϕ occurs immediately, then it is definitely "as early as possible". The later ϕ occurs, the less it satisfies "as early as possible". However, the statement *AS EARLY AS POSSIBLE* ϕ technically allows ϕ to become true at any point after the current time. Therefore, the membership function for the duration is never zero but approaches zero gradually as time increases.

AS LATE AS POSSIBLE is similar except that the duration has its membership function with minimum value at the current time and increasing gradually. This captures the fact that later is better; the statement "as late as possible" only becomes certain when $t = \infty$.

AS CLOSE AS POSSIBLE TO f ϕ says that ϕ occurs periodically and expresses a constraint that the period is as close as possible to f. Formally, this is defined using a fuzzy duration whose membership function has its maximum when the period is exactly f and tails off either side of f. The membership function tails off asymptotically to ∞ for values greater than f and converges to 0 for values less than f. Consider the formal definition in Table 2. It states that *AS CLOSE AS POSSIBLE TO* f ϕ holds if and only if $\mathcal{X}_{=v}\phi$ holds for every duration v that corresponds to a multiple of the frequency, f. In other words, "as close as possible" is satisfied more when the frequency of ϕ is closer f.

AS CLOSE AS POSSIBLE TO q ϕ is similar to the previous case but concerns closeness to a quantity. Note that ϕ must be quantifiable – i.e., there is a function Δ where $\Delta(\phi)$ is a quantity. This allows one to measure ϕ against the ideal quantity, q. The formal definition then simply compares the difference $(\Delta(\phi) - q)$ and asks whether it is roughly zero. The fuzzy set S defines the number "roughly zero".

AS MANY AS POSSIBLE and *AS FEW AS POSSIBLE* are similar to the previous case in that they compare a quantity with an ideal case using a fuzzy number.

4 A Process for Applying RELAX

Prior to applying RELAX, we assume that a conventional process of requirements discovery has been applied to yield a set of *SHALL* statements. Figure 2 overviews the process for RELAX-ing requirements; this process also identifies the invariant requirements. Each step is described.

For each *SHALL* statement, apply the following steps:

1. Must *SHALL* statement always be satisfied? For each *SHALL* statement, determine whether it must always be satisfied (e.g., safety property), or whether it could be relaxed under certain circumstances. In the former case, leave the *SHALL* statement as is, and denote it as an invariant requirement. A non-invariant requirement is potentially RELAX-able, thus implying that some form of run-time adaptation may be necessary to make the best use of the available resources while delivering acceptable behavior.
2. Identify uncertainty factors. For each potentially RELAX-able requirement:

- Identify and describe the part of the environment relevant to this requirement (**ENV**). The objective is to help the requirements engineer ascertain whether uncertainty exists in the **ENV**, thus potentially making satisfaction of the requirement problematic and necessitate its RELAX-*ation.*
- Identify the observable properties of the environment that can be monitored (**MON**)
- We expect the **ENV** and **MON** attributes to coincide except in cases where environmental properties cannot be directly observed. The **ENV/MON** relationship is made explicit by **REL**
- Requirements often make competing demands on resources. Thus requirements have inter-dependencies (**DEP**) and it is particularly important to understand these when assessing the uncertainty surrounding a requirement.

3. Must *SHALL* statement be RELAX-ed to handle uncertainty factors? Analyze the uncertainty factors to determine if sufficient uncertainty exists in the environment that makes absolute satisfaction of the requirement problematic or undesirable. If so, then this *SHALL* statement needs to proceed to the next step for introducing RELAX operators. If, however, the analysis reveals no uncertainty in its scope of the environment, then the requirement is potentially always satisfiable and therefore identified and maintained as an invariant.
4. Introduce RELAX operator(s). Given the sources of uncertainty, determine whether a requirement should be relaxed to introduce ordinal, temporal, or modal behavior flexibility at run time. Sources for uncertainty include: contention for resources, adverse environmental conditions, timing of events, and the duration of conditions.

RELAX Expression	**Informal**	**FBTL Formalization**
SHALL ϕ	ϕ is true in any state	$\mathbf{AG}\phi$
MAY ϕ_1 OR *MAY* ϕ_2	in any state, either ϕ_1 or ϕ_2 is true	$\mathbf{AG}(\phi_1 \vee \phi_2)$
EVENTUALLY ϕ	ϕ will be true in some future state	$\mathbf{AF}\phi$
$\phi_1 \ \mathcal{U} \ \phi_2$	ϕ_1 will be true until ϕ_2 becomes true	$\mathbf{A}(\phi_1 \ \mathcal{U} \ \phi_2)$
BEFORE e ϕ	ϕ is true in any state occurring prior to event e	$\mathbf{A}\mathcal{X}_{<e_d} \ \phi$ where e_d is the duration up until the next occurrence of e
AFTER e ϕ	ϕ is true in any state occurring after event e	$\mathbf{A}\mathcal{X}_{>e_d} \ \phi$
IN t ϕ	ϕ is true in any state in the time interval t	(*AFTER* t_{start} ϕ $\wedge$ *BEFORE* t_{end} ϕ) where t_{start}, t_{end} are events denoting the start and end of interval t respectively
AS EARLY AS POSSIBLE ϕ	ϕ becomes true in some state as close to the current time as possible	$\mathbf{A}\mathcal{X}_{\geq d}\phi$ where d is a fuzzy duration defined such that its membership function has its maximum at 0 (i.e., $m(0) = 1$) and decreases continuously for values > 0
AS LATE AS POSSIBLE ϕ	ϕ becomes true in some state as close to time $t = \infty$ as possible	$\mathbf{A}\mathcal{X}_{\geq d}\phi$ where d is a fuzzy duration defined such that its membership function has its minimum value at 0 (i.e., $m(0) = 0$) and increases continuously for values > 0
AS CLOSE AS POSSIBLE TO f ϕ	ϕ is true at periodic intervals where the period is as close to f as possible	$\mathbf{A}(\mathcal{X}_{=d}\phi \wedge \mathcal{X}_{=2d}\phi \wedge \mathcal{X}_{=3d}\phi \wedge \ldots)$ where d is a fuzzy duration defined such that its membership function has its maximum value at the period defined by f (i.e., $m(d) = m(2d) = \ldots = 1$) and decreases continuously for values less than and greater than d (and $2d, \ldots$)
AS CLOSE AS POSSIBLE TO q ϕ	there is some function Δ such that $\Delta(\phi)$ is quantifiable and $(\Delta(\phi) - q)$ is as close to 0 as possible	$\mathbf{AF}((\Delta(\phi) - q) \in S)$ where S is a fuzzy set whose membership function has value 1 at zero ($m(0) = 1$) and decreases continuously around zero. $\Delta(\phi)$ "counts" the quantifiable that will be compared to q.
AS MANY AS POSSIBLE ϕ	there is some function Δ such that $\Delta(\phi)$ is as close to ∞ as possible	$\mathbf{AF}(\Delta(\phi) \in S)$ where S is a fuzzy set whose membership function has value 0 at zero ($m(0) = 0$) and increases continuously around zero
AS FEW AS POSSIBLE ϕ	there is some function Δ such that $\Delta(\phi)$ is quantifiable and is as close as possible to 0	$\mathbf{AF}(\Delta(\phi) \in S)$ where S is a fuzzy set whose membership function has value 1 at zero ($m(0) = 1$) and decreases continuously around zero

Table 2 Semantics of RELAX expressions

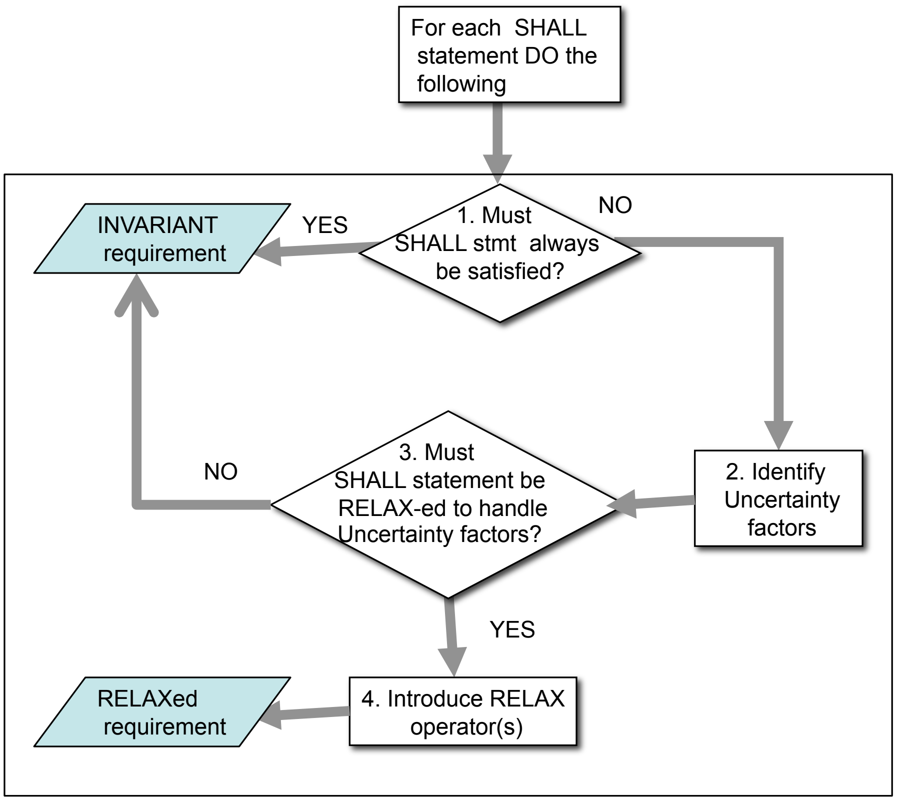
For each SHALL statement DO the following
1. Must SHALL stmt always be satisfied?
YES
NO
INVARIANT requirement
2. Identify Uncertainty factors
3. Must SHALL statement be RELAX-ed to handle Uncertainty factors?
NO
YES
4. Introduce RELAX operator(s)
RELAXed requirement

Fig. 2 RELAX Process

Note that the process describes a way of incrementally building up a model of the environment. This approach is in contrast to including an explicit task to model the environment. The latter is difficult in practice because it may not be clear which environmental factors might be relevant. It is also important to note that each iteration of the RELAX-ation process implicitly includes a form of regression assessment to ensure that the dependencies between the requirements are considered.

5 Example Application

To validate RELAX, we conducted a case study provided by Fraunhofer IESE in the form of an existing concept document describing a smart home for assisted living. [1] The concept document was written previously and independently of the RELAX research work. Below is an excerpt:

[1] See www.iese.fraunhofer.de/fhg/iese/projects/med_projects/aal-lab/index.jsp

Mary is a widow. She is 65 years old, overweight and has high blood pressure and cholesterol levels. Mary gets a new intelligent fridge. It comes with 4 temperature and 2 humidity sensors and is able to read, store, and communicate RFID information on food packages. The fridge communicates with the Ambient Assisted Living (AAL) system in the house and integrates itself. In particular, it detects the presence of spoiled food and discovers and receives a diet plan to be monitored based on what food items Mary is consuming.
An important part of Mary's diet is to ensure minimum liquid intake. The intelligent fridge partially contributes to it. To improve the accuracy, special sensor-enabled cups are used: some have sensors that beep when fluid intake is necessary and have a level to monitor the fluid consumed; others additionally have a gyro detecting spillage. They seamlessly coordinate in order to estimate the amount of liquid taken: the latter informs the former about spillages so that it can update the water intake level. However, Mary sometimes uses the cup to water flowers. Sensors in the faucets and in the toilet also provide a means to monitor this measurement.

Advanced smart homes, such as Mary's AAL, rely on adaptivity to work properly. For example, the sensor-enabled cups may fail, but since maintaining a minimum of liquid intake is a life-critical feature, the AAL should be able to respond by achieving this requirement in some other way.

5.1 Applying RELAX to the AAL

To apply RELAX, we first extracted a set of requirements from the concept document, structured as a list of *SHALL* statements. We then applied the process in Figure 2 to identify which of these *SHALL* statements should be relaxed. It is important to note that the decision of whether a requirement is invariant or not is an issue for the system stakeholders, aided by the requirements engineer. We have reverse-engineered the concept document description to simulate this decision point, but the analysis of the non-invariant requirements is accurately portrayed in the case study.

Requirements on the AAL at several abstraction levels can be extracted from the concept document. At the highest level is an implicit goal of keeping Mary healthy. From this goal, the following requirement can be identified:

R1: The system *SHALL* monitor Mary's health and *SHALL* notify emergency services in case of emergency.

It is then possible to identify a set of user-level requirements that support **R1**, where a small subset are given in Figure 3. These user-level requirements represent the essential properties of the AAL at a level of abstraction that is amenable to trade-off analysis. However, the RELAX process can be applied at different levels of granularity and might equally be applied at a lower level such as, for example, requirements specifying the detection of water spillage using the gyro-enabled cup.

Note that **R1.5** follows common practice of deferring the quantifying of a variable (in this case, the maximum period of no detected activity before the emergency services are alerted) until further analysis allows an appropriate value to be determined.

R1.1	:	The fridge *SHALL* detect and communicate with food packages.
R1.2	:	The fridge *SHALL* monitor and adjust the diet plan.
R1.3	:	The system *SHALL* ensure a minimum of liquid intake.
R1.4	:	The system *SHALL* minimize energy consumption during normal operation.
R1.5	:	The system *SHALL* raise an alarm if no activity by Mary is detected for **t.b.d.** hours during normal waking hours.
R1.6	:	The system *SHALL* minimize latency when an alarm has been raised.

Fig. 3 Subset of requirements that support **R1**

Once the requirements have been formulated as *SHALL* statements, the requirements engineer must work through the list, classifying the requirements as either invariant or relaxable. Starting with **R1.1**, the requirements engineer must determine whether or not to relax **R1.1**. To make this decision, the requirements engineer must ask whether the system will simply fail to satisfy **R1** if complete food information is not available, or if it is possible for the system to continue to operate but at a reduced capacity. Less than full functionality might be necessary to handle an emergency situation, e.g., where it might be preferable to divert resources from the intelligent fridge to support emergency functions, such as the latency requirement specified by **R1.5**. If **R1.1** were made an invariant statement, then an autonomous system will not have the flexibility to redirect resources in this way. Therefore, by relaxing **R1.1**, we allow for an adaptive system to balance resources in order to optimize global system parameters.

RELAX-ing **R1.1** gives the following requirement **R1.1'**. For all RELAX requirements that follow in this section, we present them in a box with a number of slots: the RELAX requirement itself; the **ENV**, **MON**, **REL**, and **DEP** specifications; the RELAX statement written in the RELAX grammar; the formalized version in FBTL; and any explanations for the formal definitions. We include the last three for completeness and so the reader can see how RELAX requirements are formalized in FBTL. Note that the requirements engineer does not specify these.

R1.1': The fridge *SHALL* detect and communicate information with *AS MANY* food packages *AS POSSIBLE.*
ENV: Food locations, food item information (type, calories) & food state (spoiled, unspoiled).
MON: RFID readers; Cameras; Weight sensors.
REL: RFID tags provide food locations/food information/food state; Cameras provide food locations; Weight sensors provide food information (whether eaten or not).
DEP: **R1.1'** negatively impacts **R1.2'**; **R1.1'** positively impacts **R1.4** and **R1.6**

Grammar Expression: *SHALL* (*AS MANY AS POSSIBLE* p)
Formal RELAX expression: $\mathbf{AGF}(\Delta(p) \in S)$
Definitions. p is "The fridge detects and communicates information with food packages" and $\Delta(p)$ is the number of food packages

R1.1' conforms to the grammar expression *SHALL* (*AS MANY AS POSSIBLE* p), where p corresponds to *the fridge detects and communicates information with food packages*. In the FBTL expression of **R1.1'**, $\Delta(p)$ represents the number of food packages the fridge actually detects and resolves information about using the means available to it as defined in the requirement's RELAX uncertainty factors (discussed below). The set S is a fuzzy set whose membership function m has value 0 at zero ($m(0) = 0$) and increases continuously around zero to 1 at n, where n is the total number of food packages contained by the fridge. Hence, if all food packages in the fridge are detected, $m(\Delta(p)) = 1$. If none of the food packages in the fridge are detected, $m(\Delta(p)) = 0$. In the general case where *some* of the food packages in the fridge are detected, $0 < m(\Delta(p)) < 1$. In a scenario where there were 10 food packages in the fridge of which 8 were detected, $m(\Delta(p))$ would be close to but a little less than 1; it might be said to have a value of "roughly" 1. When combined with the quantifiers **AGF**, **R1.1'** can be read as: eventually, by using all the available sensors and the computational resources available to process the sensor data and fuse the partial data, the fridge will be able to maximise detection of the number of food packages and collation of information about those food packages, subject to uncertainties arising from resolution of sensor data, positioning of food, nature of the food's containers, etc.

The **DEP** attribute provides a place to describe how the relaxation of **R1.1** will impact other requirements, where this field may be updated as the requirements are RELAX-ed. In this case, relaxing **R1.1** will impair the system's ability to suggest an appropriate diet plan (**R1.2'**). However, it will support the requirement to minimize latency during emergency operation (**R1.5**) as well as minimizing energy consumption during normal operation (**R1.4**). The other three attributes for the relaxed **R1.1'** are **ENV**, **MON** and **REL** as explained in Section 2. In the case of **R1.1'**, the fridge needs to ascertain information about where food items are located and the nutritional information of these food items. To monitor these characteristics, the original concept document explicitly mentioned RFID tag monitoring. We suppressed mention of this solution technology when formulating **R1.1**, to maintain a separation of concerns between the specification of behaviour and selection of the solution. However, providing values for the **ENV** and **MON** attributes prompts the requirements engineer to consider the question of whether the system has the resources to sense its environment and thus is able to collect the data needed to make adaptation decisions. Sometimes, as is the case here, explicitly identifying **ENV** and **MON** forces the requirements engineer to posit solution technologies and provide a rationale for their choice. Hence, RFID tags on food packages would provide a partial solution, but it is likely that not all food items will have RFID tags and partially eaten food would be difficult to detect. Using cameras and weight sensors would permit data, albeit imperfect, to be collected about all food items, even if they were untagged.

R1.1' therefore states that the system should be able to tolerate incomplete information about food packages. Despite using RELAX to help identify a range of sensor types with which to monitor the food in the fridge, it may be impossible to gather complete information, forcing the system to work with incomplete data. Note that the incompleteness of **R1.1'** has important consequences: (1) on other requirements – can **R1.2** still be satisfied given incomplete information? (2) on design decisions – if we accept the incomplete information assumption, we need to design algorithms that can satisfice dietplans rather than simply calculate them.

The uncertainty about the presence (i.e., availability) of food identified in **R1.1** poses problems for the formulation of diet plans using the food available within Mary's house. Consequently, it was also necessary to relax **R1.2**, making explicit the task of the dietplan to optimise Mary's calorie intake, and the need to adapt the dietplan according to Mary's actual consumption. Mary's actual consumption formed the **ENV** property, measured by the fridge and trash can sensors specified by the **MON** property. Mary's actual calorie intake is in any case uncertain because of uncertainty that all food can be accurately sensed. Hence, for example, identifying untagged food that is discarded is problematic even if the trash can is instrumented with weight and RFID tag sensors.

R1.2': The fridge *SHALL* suggest a dietplan with total calories *AS CLOSE AS POSSIBLE TO* the daily ideal calories. The fridge *SHALL* adjust the dietplan in line with Mary's actual calorie consumption.
ENV: Mary's daily calorie consumption.
MON: RFID readers and weight sensors in fridge and trash can.
REL: RFID readers and weight sensors provide consumed items; items vanish from fridge and the items (if uneaten) or the packaging (if eaten) appears in trash can.
DEP: R1.2' is negatively impacted by **R1.1'**; **R1.2'** negatively impacts **R1**

Grammar Expression: *SHALL* (*AS CLOSE AS POSSIBLE TO* q p); *SHALL* r
Formal RELAX expression: $\mathbf{AGF}((\Delta(p) - q) \in S)$; $\mathbf{AG}$r
Definitions. p is "The fridge suggests a dietplan"; $\Delta(p)$ is the dietplan total calories and q is the daily ideal calories. r is "The fridge adjusts the dietplan in line with Mary's actual calorie consumption"

In the first clause of the FBTL expression of **R1.2'**, p is *the fridge suggests a dietplan*'; $\Delta(p)$ is the dietplan total calories and q is the daily ideal calories. The difference between $\Delta(p)$ and q is therefore the dietplan's deviation from the ideal expressed in calories. S is a fuzzy set whose membership function has value 1 at zero ($m(0) = 1$) and decreases continuously around zero. Hence, if the dietplan matches Mary's ideal calorie intake, the membership function returns 1, which is the ideal result. If there is only a small deviation in calories, the membership function will return a value less than but close to 1; of *about* 1. Combined with the quantifiers, the expression specifies that, subject to constraints about what is known about the contents of the fridge and depending on the computational and other resources that are available, the fridge will construct a dietplan that gives Mary as close to an optimal calorie consumption as is feasible. In the second clause r is *the fridge adjusts the dietplan in line with Mary's actual calorie consumption*, which is the subject of a conventional *SHALL* statement.

Note that relaxing **R1.2** affects satisfaction of **R1**. The same is true of **R1.3** that needs to be relaxed because the system could tolerate temporarily relaxing the requirement to monitor Mary's liquid intake. As with Mary's food consumption, it is not possible to estimate Mary's liquid consumption with complete certainty. We explore this problem in more detail below. Here however, it is sufficient to note that the intended effect of **R1.3'** is ensure that the AAL makes a best effort to make Mary's liquid intake adequate during the day, ensuring that this is so by the time Mary goes to bed at night.

R1.3': The system *SHALL* ensure that Mary's liquid intake is as *AS CLOSE AS POSSIBLE TO* ideal during the course of the day. The system *SHALL* ensure minimum liquid intake *BEFORE* bedtime.
ENV: Mary's daily liquid intake.
MON: fluid monitoring cups; orientation sensor-enabled cups; faucet sensors; flowerpot moisture sensors; timers correlating temporal events of different sensors - was cup emptied down sink, into flower pot or did Mary drink from it?
REL: cup sensors & moisture sensors & faucet sensors & sink outlet sensors & timers all interact to collaboratively determine Mary's daily liquid intake.
DEP: R1.3 negatively impacts **R1.**

Grammar Exp.: *SHALL* (*AS CLOSE AS POSSIBLE TO* q p); *SHALL* (*BEFORE* e r)
Formal RELAX expression: $\mathbf{AGF}((\Delta(p) - q) \in S)$; $\mathbf{AG}\mathcal{X}_{<_{e_d}}\ r$
Definitions. p is "The system ensures a liquid intake"; $\Delta(p)$ is the volume of liquid and q is the required minimum volume. r is "The system ensures minimum liquid intake". e is the "bedtime" event which occurs after a duration e_d.

R1.3' uses the same RELAX operators as **R1.2'**, but deals with liquid intake instead of calories. The FBTL expression of **R1.3'** therefore mirrors that of **R1.2'**.

Requirement **R1.4**, which specifies the minimization of energy consumption, is interesting because there is a potential for impact with **R1.6** that mandates that latency should be minimized during emergencies. The course of action chosen was to RELAX **R1.4** and treat **R1.6** as an invariant because of its implicit criticality to Mary's health. Thus, while **R1.6** was unchanged by the RELAX process, it would need to be treated as a critical non-fuctional requirement in the conventional way, beginning with quantification of the minimum acceptable level of latency (not shown here). By contrast, RELAX-ation of **R1.4** resulted in its reformulation using the RELAX syntax to become:

R1.4': The system *SHALL* consume *AS FEW* units of energy *AS POSSIBLE* during normal operation.
ENV: Total energy consumption.
MON: Smart energy monitors.
REL: Smart energy monitors can sense device energy consumption and sense activity within the AAL, and use these to control (e.g.) lighting and heating.
DEP: **R1.4'** is negatively impacted by **R1.5**

Grammar Expression: *SHALL* (*AS FEW AS POSSIBLE* p)
Formal RELAX expression: $\mathbf{AGF}(\Delta(p) \in S)$
Definitions. p is "The system consumes energy during normal operation" and $\Delta(p)$ is the units of energy consumed

In the FBTL expression of **R1.4'**, p is *the system consumes energy during normal operation* and $\Delta(p)$ is the units of energy consumed. S is a fuzzy set whose membership function has value 1 at zero ($m(0) = 1$) and decreases continuously around zero. Hence, in the ideal situation of the AAL consuming no energy, the membership function would return 1. By specifying that the membership function returns a value as close as possible to close to 1, the formal requirement constrains the AAL to be as energy-efficient as is feasible under normal operation and given uncertainty about Mary's habits (below).

What is specified by **R1.4'** is the same as **R1.4**. However, the uncertainty factors help to identify the means to achieve the desired behavior. Although energy consumption can be measured accurately, the behavior of Mary cannot be predicted or controlled completely. Thus, for example, if the periods when Mary is up and active and when she is asleep vary significantly, it may prove hard to optimize the heating and lighting.

Finally, **R1.5** was considered an invariant and was thus unchanged, since it specified behavior that may be critical to Mary's health.

5.2 Applying the Uncertainty Factors

RELAX-ation prompts the requirements engineer to understand the sources of environmental uncertainty and to document them using the RELAX uncertainty factors. The **ENV** and **MON** attributes are particularly useful for documenting whether the system has the means for monitoring the important aspects of the environment. By collecting together the various **ENV** and **MON** attributes, we can build up a model of the environment in which the system will operate, as well as a model of how the system will monitor its environment. Modeling the environment can be done with the aid of a conceptual model such as that shown in Figure 4. Figure 4 uses a UML class diagram to model the environment as subclasses of an **ENV** stereotype, a set of sensors modeled as subclasses of a **MON** stereotype, their relationships modeled as **REL** associations and the physical entities that the sensors are used to instrument.

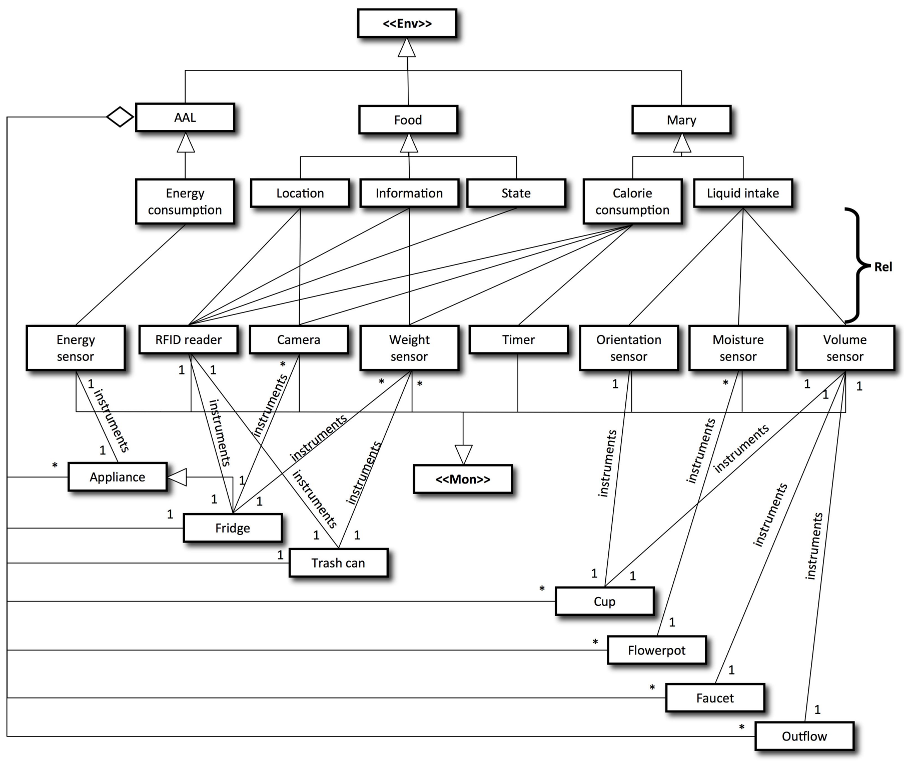

Fig. 4 AAL Conceptual Model

Tables 3 and 4 document and elaborate upon the domain model. Table 3 simply lists the environment variables. In Table 4, the first column gives the sensor, the second column states what data the sensors provide, and the third column states about which environment variables the sensor data provides information. These are marked as either complete or incomplete according to whether the data provides a complete characterization of the variable or whether additional data from other sensors is necessary.

The process of identifying and enumerating the uncertainty factors for a RELAX-ed requirement should stimulate a search for the means to mitigate them; if the uncertainty can be fully mitigated, then, in principle, the requirement can be completely satisfied and need not be RELAX-ed. Attempting to mitigate the uncertainty can be viewed as a validation of the requirement's RELAX-ation. If the uncertainty can't be mitigated, then run-time overhead of RELAX-ing the requirement in terms of (e.g.) monitoring, as well as the implied additional development costs of (e.g.) testing for emergent behavior, can be better justified.

Environment to be Monitored
Food locations
Food item information (type, calories)
Food state (spoiled, unspoiled)
Mary's daily liquid intake
Total energy consumption
Mary's daily calorie consumption

Table 3 System environment variables

Sensor	Data provided	Contributes to
RFID readers (in Fridge)	Food information (type, calories) Location of food item (if RFID tag detected, in fridge)	Food locations (complete) Food information (complete) Marys daily calorie consumption (incomplete)
Cameras in Fridge	Location of food item	Food locations (complete)
Trash can RFID reader	Location of food item; Food information (type, calories)	Food locations (complete); Food item information (complete); Mary's daily calorie consumption (incomplete)
Trash can weight sensor	Food item weight	Mary's daily calorie consumption (incomplete)
Fridge weight sensors	Food item wieght	Food information (incomplete)
Cup volume sensors	Presence of liquid	Mary's daily liquid intake (incomplete)
Cup orientations	Transfer of contents from cup	Mary's daily liquid intake (incomplete)
Flower pot moisture sensors	Presence of liquid	Mary's daily liquid intake (incomplete)
Faucet sensors	Water drawn off	Mary's daily liquid intake (incomplete)
Sink outlet sensors	Water poured away	Mary's daily liquid intake (incomplete)
Timers	Elapsed time	Mary's daily liquid intake (incomplete)
Energy meters	Energy usage	Total energy consumption (complete)

Table 4 System monitoring capabilities

6 Discussion

In this section, we reflect upon our experiences with applying RELAX. Although RELAX has not yet been applied independently by industrial partners, our comments are based both on the authors' experience of applying RELAX to a number of smart home adaptive applications as well as discussions with other researchers who have independently applied RELAX (e.g., [28, 29]). We perceive a number of benefits of RELAX, outlined briefly below, and also present some interesting research challenges with regards to how to use RELAX in a software engineering methodology for the development of self-adaptive systems.

6.1 Scoping Adaptive Behavior

RELAX gives us a means to establish the boundaries of adaptive behavior. That is, we must explicitly distinguish invariant from non-invariant requirements, identify and monitor the sources of uncertainty, and then describe what dimensions of the requirements can be relaxed and satisfied by adaptive behavior. The invariants provide a point of reference for adaptive behavior. The RELAX process also forced us to consider each non-invariant requirement in isolation, with the effect of incrementally revealing each requirements interdependencies and generating what is effectively trace information in the **DEP** attribute (cf. [39]).

In addition, by separately describing the environment and the monitoring, we can identify deficiencies in the monitoring infrastructure. Given that a DAS can only adapt based on its monitoring information, missing or insufficient sensors for the environment in question significantly impact the effectiveness of the DAS. An adaptive system senses its environment and then reacts accordingly. However, a tough question when designing an adaptive system is to determine the minimal monitoring infrastructure that is required. To be effective, the DAS must be equipped with enough sensors to adequately monitor the environment but resource constraints limit the monitoring infrastructure that can be provided. In practical cases, such as in Mary's smart home system, deciding on the optimal monitoring infrastructure is highly non-trivial. RELAX provides a systematic process to evaluate the current sensing infrastructure and to decide if there are adaptations that are required but are not possible without adding new monitors. For example, in **R1.1**, we found that RFID tags would not provide sufficiently complete data about the presence and consumption of food. With the original sensor resources, we were unable to address the uncertainty posed by the environment, thus prompting us to posit other sensor types that would help gather more data and potentially reduce the uncertainty.

We have begun initial work on formalizing this process of assessing the monitoring infrastructure as a variation of threat modeling. Threat modeling [32, 34, 37] is a technique used in security analysis of systems to systematically posit threats to a system architecture, to prioritize risks based on those threats, and then to extend the architecture to resist the most important threats. A similar process can usefully be applied in RELAX to uncover missing monitoring requirements or missing sensors. For each RELAX-ed requirement, the requirements engineer asks what information is necessary to invoke the adaptation corresponding to the requirement. If the current monitoring infrastructure does not provide enough information about the environment, then either the monitoring infrastructure should be extended or, if resources do not permit it, the adaptive capability must be reduced. This kind of trade-off analysis can be effectively modeled using goal-based requirements languages (in particular, obstacle analysis in KAOS [36] provides a variant of threat modeling). Cheng et al. [7] reports on our first attempts in this direction of work.

6.2 Implementing RELAX requirements

An important future question to consider is how to design and implement RELAX-ed requirements. Because of their adaptive nature, they must be implemented using an adaptive infrastructure. There are many possible ways of doing so – for example, using adaptive architectures (e.g., [13, 18]), using techniques based on run-time switching of features (*aka* dynamic software product lines) [15], using advanced AI techniques such as neural nets or genetic algorithms, or using optimization techniques (e.g., partial satisfaction planning [23]). An important driver when developing RELAX, however, was to maintain independence from any of these design techniques. We see this as a key advantage over previous attempts to derive requirements for adaptive systems. For example, Lapouchnian *et al.* [21] describe an approach based on goal-modeling in which sub-goals are used to derive alternative behaviors corresponding to adaptations. The alternatives can then be implemented as adaptations using an appropriate architecture. The approach requires an explicit enumeration of the alternative behaviors and hence does not support adaptive behaviors that have not been "designed in". The extent to which designers wish to hard-code adaptations in this way depends on the application domain. There is a range of adaptivity in DASs – from adaptive systems able to deal in only a minor way with unknown environmental conditions (e.g., because the alternatives are explictly designed and therefore limited) to those able to deal with the "so-called unknown unknowns" (e.g., highly adaptive systems based on AI). The main observation regarding RELAX is that it is agnostic to whichever style of design is chosen. The result is a very clear separation of adaptivity requirements and the adaptive infrastructure used to implement them.

As future work, we are investigating how to map RELAX requirements to a variety of different adaptive infrastructures. The mapping cannot be defined independently from the choice of infrastructure. For example, most existing work on software architectures for adaptive systems works by defining specific thresholds which, when reached, trigger an adaptation. These thresholds consist of concrete values for sensor attributes and are deliberately not defined in RELAX in order to maintain the separation of requirements and design. Therefore, to map to such an architectural style would require these thresholds to be defined as part of the mapping process. Equally, however, one could map RELAX to a less prescriptive architecture in which thresholds are not concrete, fixed values but are (e.g.) fuzzy, derived at runtime using machine learning techniques, or evolve autonomously using genetic algorithms.

6.3 Analyzing RELAX requirements

We have not yet implemented any analysis techniques for RELAX requirements. The formal semantics is a prerequisite for this, of course. In general, there are a number of different kinds of analysis that could be applied to RELAX, ranging from simple type-checking to more sophisticated model checking. The typed grammar (see Section 3) permits straightforward type-checking algorithms to be applied to RELAX statements. A more involved analysis would be to check whether a RELAX specification is in fact satisfiable (or satisfice-able). Given the highly declarative nature of RELAX statements, it is possible to write a specification that is impossible to implement because (e.g.) two RELAX requirements directly conflict with each other. Traditional model checking or constraint satisfaction techniques could be usefully applied to uncover such problems.

More interestingly, heuristic checking methods may be desired to highlight specifications that, whilst technically satisfiable, are not well-specified or meaningful. For example, one RELAX statement might require *AS MANY AS POSSIBLE* of a particular quantity whereas another requires *AS FEW AS POSSIBLE* . Whilst this is implementable (by simply choosing the midpoint value of the quantity), it may not be the best way to write the specification. Heuristic methods that could point out *po-*

tential issues could serve as a useful aid when developing the specification. Note that this issue is not particular to RELAX but is because of the very nature of adaptive systems, which must by necessity involve trade-offs.

6.4 Requirements Reflection

RELAX is the first step towards a longer term vision which we are calling "requirements reflection"[3]. Computational reflection is the ability of a program to observe and possibly modify its design [25]. When source code is compiled, information about the structure of the program is normally lost as lower level code is produced. If a system supports reflection, the structure is preserved as metadata.

In a similar way, requirements reflection refers to a case where knowledge of the structure and content of the system requirements is available at runtime to support dynamic requirements introspection and analysis. Currently, systems have no explicit knowledge of their requirements. At best, adaptive systems monitor for threshold values which have been derived from requirements, but these systems are not cognizant of the requirements themselves. Requirements reflection aims to make requirements first-class entities at runtime. By so doing, software implemented using requirements reflection will be able to reconsider decisions at runtime when more precise, complete, and up-to-date information can be obtained through observation of the system execution. Furthermore, in order for users to trust adaptive systems, it is essential that the system be able to explain its adaptations in terms that are meaningful to users. Requirements reflection will support this ability by allowing the system to explain its changes in terms of elements in its requirements model, such as the context that motivated the change and the expected impact of the change on the levels of requirements satisfaction.

Requirements reflection relies on an explicit runtime representation of requirements (in the same way that architectural reflection explicitly represents a system's architecture [25]). We see RELAX as the first step towards tackling this challenge since RELAX scopes the envelope of adaptivity as discussed earlier. Precise requirements languages, which include support for specifying adaptivity, are required for this purpose. RELAX is not yet the answer to do this but future work will investigate how to combine RELAX with goal modeling languages such as KAOS.

7 Related Work

Recently, there has been a surge of interest in software engineering research for self-adaptive systems [6]. For requirements engineering, Berry *et al.* [3] have defined a framework of discourse for DAS requirements. Goal-based modeling has been used for specifying the adaptation choices that a DAS must make [14, 22, 27, 43] as well as for the specification of monitoring and switching between adaptive behaviors [31]. A particular strength of goal-based modeling is that it supports the modeling of non-functional trade-offs, which can be used to capture some elements of environmental uncertainty. This is well illustrated by work on partial goal satisfaction (e.g., [14, 24]).

A feature common to these works is that they assume that all adaptation choices are known and enumerated at design time. Hence, *unanticipated* adaptations are difficult

[3] a term coined by Anthony Finkelstein at a Dagstuhl workshop on software engineering for self-adaptive systems, January 2008

to specify and analyze. RELAX avoids this problem by specifying declaratively the ways in which a requirement may be RELAX-ed. RELAX does not require all possible alternative adaptations to be specified during requirements engineering. This flexibility leaves open the design choice as to how to achieve adaptation and therefore supports designs based on adaptation rules, planning algorithms, control theory algorithms, etc. Having said this, RELAX and goal-based approaches can be used in a complementary fashion. Recently, we have used KAOS [36] to show how obstacle modeling can be used to reason about uncertainty and identify where it impacts on the goal hierarchy [7]. We performed an initial goal refinement and then used obstacle analysis modeling (as described in Section 5) to capture the uncertainty in the environment. Once the impact of the uncertainty on the goals is understood, RELAX-ation can be applied to mitigate the uncertainty.

Regardless of how a DAS's requirements are specified, the requirements must be properly integrated into a run-time requirements monitoring and adaptation infrastructure. Run-time monitoring dynamically assesses the conformance of runtime behavior to the specified requirements. At run-time, a monitor runs concurrently with the system to detect violations of monitored assertions [35] and informs the choice of run-time adaptation. In the web services domain, the idea of run-time monitoring has rapidly gained ground, driven by web services' capacity for dynamic (re-)binding and the presumed availability of alternative, functionally equivalent services offered by competing service providers [2, 33]. In web services, the requirements that are monitored are typically qualities of service (QoS), such as response time, which are specified as service-level agreements (SLAs). Web services, however, may be viewed as a special case where a restricted but fairly general class of QoS requirements has wide relevance, where standards have emerged that support machine-interpretable SLA specification, and where monitors may exploit defined message formats with which service-based systems communicate. The choice of run-time adaptation in response to detected SLA violations is similarly bounded; typically dynamic renegotiation of the SLA or binding to an alternative service.

Outside the domain of web services, where standards are weaker and the run-time platforms potentially more heterogeneous, the question of what and how to monitor is more open. Nevertheless, a number of requirements monitoring frameworks have been defined, including [11, 12, 30], in which system developers must specify a set of conditions the software should monitor, a set of tuneable system parameters and design alternatives, and a set of adaptation rules defining when and how to tune the system parameters and switch to an alternative design based on the monitored conditions. Using, for example, the ReqMon framework [30], RELAX-ed requirements could be mapped to monitors specified using a monitor specification language. The RELAX **ENV**, **MON** and **REL** uncertainty factors have been specifically designed to facilitate this mapping. Our ultimate vision is to enable a DAS to dynamically (i.e. during execution) reason about its own requirements and goals, that is, achieve "requirements reflection" [8]). Explicit run-time representations of system requirements incorporating uncertainty are crucial for this vision. A good step in this direction is provided by the work of Wang *et al.* [38], which monitors goals at run-time and applies AI diagnostic theories to diagnose failed goals.

Software engineering activities downstream from requirements are much better represented when it comes to architecting DASs. Without fully elaborating on the large body of work, we refer readers to a roadmap paper [18] which discusses the state-of-the-art in software architecture for DASs. Progress has also been made in addressing assurance of adaptive systems [5, 19, 20, 44–46]. AI techniques for implementing DASs include approaches building on model-based diagnosis [10] and planning (e.g., [42]).

8 Conclusions

This paper has presented a new requirements specification language called RELAX designed to explicitly address uncertainty for specifying the behavior of dynamically adaptive systems. RELAX has three types of operators to handle uncertainty: temporal, ordinal, and modal. We consider two key sources of uncertainty. Environmental uncertainty refers to the (possibly unexpected) changing conditions of the execution environment. Behavioral uncertainty refers to the need to change system behavior at run time in response to the environmental uncertainty. In some cases, behavioral uncertainty can also be due to lack of sufficient information about the application's intended behavior at development time, thus still requiring run-time adaptation. We introduced a process for using the RELAX language that incrementally builds up a view of the execution environment, while introducing RELAX operators to the non-invariant requirements.

After applying RELAX to a number of smart home adaptive applications, we observed several key benefits. First, RELAX gives us a means to establish the boundaries of adaptive behavior. That is, we must explicitly distinguish invariant from non-invariant requirements, identify and monitor the sources of uncertainty, and then describe what dimensions of the requirements can be relaxed and satisfied by adaptive behavior. The invariants provide a point of reference for adaptive behavior. Second, the RELAX process forced us to consider each non-invariant requirement in isolation, with the effect of incrementally revealing each requirements interdependencies and generating what is effectively trace information in the **DEP** attribute (cf. [39]). Third, by separately describing the environment and the monitoring, we can identify deficiencies in the monitoring infrastructure. Given that a DAS can only adapt based on its monitoring information, missing or insufficient sensors for the environment in question significantly impact the effectiveness of the DAS.

Many possible directions for future work are possible. Additional uncertainty factors may facilitate the RELAX process. For example, recently we have used a variation of threat modeling techniques to assess ways in which uncertainty from the changing environment may impact goals [7]. Different mitigation strategies are used to address the uncertainty, including RELAXing a given goal. A natural progression is to explore how RELAXed goals can be used to guide the design refinement process for adaptive systems. One approach is to use RELAX specifications to guide the automatic generation of software models for adaptive systems, such as the evolutionary computation approach by Goldsby and Cheng [14] that returns a suite of solutions for a set of system requirements. We are exploring specification patterns for RELAX to be used in conjunction with Spider [17], a natural-language interface for specification patterns and analysis tools to further facilitate the use of RELAX by the DAS community. Finally, we are investigating different requirements monitoring options that will enable RELAX requirements to be monitored at run time.

References

1. Fahiem Bacchus, *Representing and reasoning with probabilistic knowledge: a logical approach to probabilities*, MIT Press, Cambridge, MA, USA, 1990.
2. Luciano Baresi, Carlo Ghezzi, and Sam Guinea, *Smart monitors for composed services*, ICSOC, 2004, pp. 193–202.
3. D.M. Berry, Betty H. C. Cheng, and J. Zhang, *The four levels of requirements engineering for and in dynamic adaptive systems*, 11th Int. Work. on Requirements Engineering: Foundation for Software Quality (REFSQ'05) (Porto, Portugal), 2005.
4. Gordon Blair, Nelly Bencomo, and Robert France, *Models at run.time*, Computer **36** (2003), no. 1, Special issue on Models at Run Time.
5. J.S. Bradbury, J.R. Cordy, J. Dingel, and M. Wermelinger, *A classification of formal specifications for dynamic architectures*, 2004.
6. Betty H. C. Cheng, Holger Giese, Paola Inverardi, Jeff Magee, and Rogerio de Lemos, *Software engineering for self-adaptive systems: A research road map, Dagstuhl-seminar on software engineering for self-adaptive systems*, Software Engineering for Self-Adaptive Systems (Betty H. C. Cheng, Holger Giese, Paola Inverardi, Jeff Magee, and Rogerio de Lemos, eds.), Springer, 2008, Lecture Notes in Computer Science, 5525.
7. Betty H. C. Cheng, Pete Sawyer, Nelly Bencomo, and Jon Whittle, *A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty*, MoDELS '09: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems, Springer-Verlag, 2009.
8. Betty H. C. Cheng, Jon Whittle, Nelly Bencomo, Anthony Finkelstein, Jeff Magee, Jeff Kramer, Sooyong Park, and Schahram Dustda, *Software engineering for self-adaptive systems: A research road map, requirements engineering section*, Software Engineering for Self-Adaptive Systems (Betty H. C. Cheng, Holger Giese, Paola Inverardi, Jeff Magee, and Rogerio de Lemos, eds.), Springer, 2008, Lecture Notes in Computer Science, 5525.
9. Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau (eds.), *Component-based software engineering, 7th international symposium, CBSE 2004, Edinburgh, uk, may 24-25, 2004, proceedings*, Lecture Notes in Computer Science, vol. 3054, Springer, 2004.
10. J. De Kleer, A.K. Mackworth, and R. Reiter, *Characterizing diagnoses and systems*, Artificial Intelligence **56** (1992), no. 2-3, 197 – 222.
11. M.S. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, *Reconciling system requirements and runtime behavior*, Ninth International Workshop Software Specification and Design, Apr 1998, pp. 50–59.
12. S. Fickas and M.S. Feather, *Requirements monitoring in dynamic environments*, Second IEEE International Symposium on Requirements Engineering (RE'95), 1995.
13. David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste, *Rainbow: Architecture-based self-adaptation with reusable infrastructure*, IEEE Computer **37** (2004), no. 10, 46–54.
14. Heather Goldsby, Pete Sawyer, Nelly Bencomo, Danny Hughes, and Betty H. C. Cheng, *Goal-based modeling of dynamically adaptive system requirements*, 15th Annual IEEE Int. Conf. on the Engineering of Computer Based Systems (ECBS), 2008.
15. Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg, *On the notion of variability in software product lines*, WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01), IEEE Computer Society, 2001, p. 45.
16. Jeffrey O. Kephart and David M. Chess, *The vision of autonomic computing*, Computer **42** (2009), no. 10, 41–50.
17. Sascha Konrad and Betty H. C. Cheng, *Facilitating the construction of specification pattern-based properties*, Proc. of IEEE International Requirements Engineering Conference (RE05) (Paris, France), August 2005, pp. 329–338.
18. J. Kramer and J. Magee, *Self-managed systems: an architectural challenge*, Future of Software Engineering, 2007, pp. 259–268.
19. Jeff Kramer and Jeff Magee, *Analysing dynamic change in software architectures: a case study*, Proceedings. Fourth International Conference on Configurable Distributed Systems (Annapolis, MA), IEEE, 4–6 1998, pp. 91–100.
20. Sandeep S. Kulkarni and Karun N. Biyani, *Correctness of component-based adaptation.*, in Crnkovic et al. [9], pp. 48–58.
21. Alexei Lapouchnian, Sotirios Liaskos, John Mylopoulos, and Yijun Yu, *Towards requirements-driven autonomic systems design*, Proceedings of ICSE 2005 Workshop on Design and Evolution of Autonomic Application Software (St. Louis, Missouri), May 2005.

22. Alexei Lapouchnian, Yijun Yu, Sotirios Liaskos, and John Mylopoulos, *Requirements-driven design of autonomic application software*, Proceedings of CASCON 2006, 2006.
23. Emmanuel Letier and Axel van Lamsweerde, *Reasoning about partial goal satisfaction for requirements and design engineering*, SIGSOFT Softw. Eng. Notes **29** (2004), no. 6, 53–62.
24. Emmanuel Letier and Axel van Lamsweerde, *Reasoning about partial goal satisfaction for requirements and design engineering*, Proc. of 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2004, pp. 53–62.
25. Pattie Maes, *Concepts and experiments in computational reflection*, Proceedings of the Conference on Object-Oriented Programming Syst ems, Languages, and Applications (OOPSLA) (New York, NY) (Norman Meyrowitz, ed.), vol. 22, ACM Press, 1987, pp. 147–155.
26. S. Moon, K.H. Lee, and Doheon Lee, *Fuzzy branching temporal logic*, Systems, Man, and Cybernetics, Part B, IEEE Transactions on **34** (2004), no. 2, 1045–1055.
27. Mirko Morandini, Loris Penserini, and Anna Perini, *Modelling self-adaptivity: A goal-oriented approach*, SASO '08: Proc. of 2008 Second IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems, 2008, pp. 469–470.
28. Liliana Pasquale and Luciano Baresi, *Applying RELAX to requirements for service-oriented architectures*, 2009, Private Communication.
29. Nauman Ahmed Qureshi and Anna Perini, *Applying RELAX to softgoals of adaptive systems*, 2009, Private Communication.
30. William Robinson, *A requirements monitoring framework for enterprise systems*, Requirements Engineering **11** (2005), no. 1, 17 – 41.
31. M. Salifu, Yijun Yu, and B. Nuseibeh, *Specifying monitoring and switching problems in context*, IEEE Int. Requirements Engineering Conference, Oct. 2007, pp. 211–220.
32. Bruce Schneier, *Attack Trees - Modeling security threats*, Dr. Dobb's Journal (1999).
33. George Spanoudakis, *Non intrusive monitoring of service based systems*, International Journal of Cooperative Information Systems **15** (2006), 325–358.
34. Frank Swiderski and Window Snyder, *Threat modeling*, Microsoft Press, Redmond, WA, USA, 2004.
35. Axel van Lamsweerde, *Requirements engineering: from craft to discipline*, Proc. of 16th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering, 2008, pp. 238–249.
36. Axel van Lamsweerde, *Requirements engineering: From system goals to UML models to software specifications*, Wiley, 2008.
37. Axel van Lamsweerde and Emmanuel Letier, *Handling obstacles in goal-oriented requirements engineering*, IEEE Trans. Softw. Eng. **26** (2000), no. 10, 978–1005.
38. Y. Wang, S. McIlraith, Y. Yu, and J. Mylopoulos, *An automated approach to monitoring and diagnosing requirements*, Automated Software Engineering Conference (Atlanta, Georgia), 2007, pp. 293–302.
39. Kristopher Welsh and Pete Sawyer, *Requirements tracing to support change in dynamic adaptive systems*, Requirements Engineering Foundations for Software Quality (REFSQ), 2009.
40. Jon Whittle, Pete Sawyer, Nelly Bencomo, and Betty H. C. Cheng, *Reassessing languages for requirements engineering of self-adaptive systems*, RE Workshop for SOCCER08, Technical Report, 2008.
41. Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel, *RELAX: Incorporating uncertainty into the specication of self-adaptive systems*, Proceedings of IEEE International Requirements Engineering Conference (RE09) (Atlanta, Georgia), 2009.
42. B.C. Williams, M.D. Ingham, S.H. Chung, and P.H. Elliot, *Model-based programming of intelligent embedded systems and robotic space explorers*, Proc. of IEEE **91** (2003), no. 1, 212 – 237.
43. Yu Yijun, Alexei Lapouchnian, Sotirios Liaskos, John Mylopoulos, and Julio Leite, *From goals to high-variability software design*, vol. 4994, Springer Berlin / Heidelberg, 2008.
44. Ji Zhang and Betty H. C. Cheng, *Model-based development of dynamically adaptive software*, ICSE '06: Proceeding of the 28th international conference on Software engineering (New York, NY, USA), ACM Press, 2006, pp. 371–380.
45. Ji Zhang and Betty H. C. Cheng, *Using temporal logic to specify adaptive program semantics*, Journal of Systems and Software (JSS), Architecting Dependable Systems **79** (2006), no. 10, 1361–1369.
46. Ji Zhang, Heather J. Goldsby, and Betty H. C. Cheng, *Modular verification of dynamically adaptive systems*, AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development (New York, NY, USA), ACM, 2009, pp. 161–172.