RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems

Jon Whittle*, Pete Sawyer*, Nelly Bencomo*,
Betty H.C. Cheng† and Jean-Michel Bruel‡
*Computing Department, InfoLab21, Lancaster University, Lancaster LA1 4AW, UK
whittle, sawyer, nelly@comp.lancs.ac.uk
†Department of Computer Science and Engineering, Michigan State University, MI 48824, USA
chengb@cse.msu.edu
‡University of Toulouse, CNRS/IRIT Laboratory, 118 Rte de Narbonne, F-31062 Toulouse Cedex, France
bruel@irit.fr

Abstract

Self-adaptive systems have the capability to autonomously modify their behaviour 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 this paper, we argue that a more rigorous treatment of requirements explicitly 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 RELAX, a new requirements language for self-adaptive systems and illustrate it using examples from the smart home domain.

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 smart home 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, 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 the uncertainty imposed by 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* [10] 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.

The proposed RELAX language supports the explicit expression of environmental uncertainty in requirements. We have designed the vocabulary of RELAX to enable analysts to identify requirements that may be relaxed at run time when the environment changes. For example, it may be acceptable to temporarily relax 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 behaviour 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, we use RELAX operators to introduce flexibility

into *SHALL* statements of a system.[1] Several dimensions for uncertainty 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 application to illustrate its features. The grammar and semantics for RELAX are introduced in Section 3. 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 overviews related work, and we conclude in Section 7.

2. RELAX Overview

This section presents the RELAX 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. Preliminary ideas on RELAX, including a first suggestion of RELAX operators, were presented in [24]. This paper refines the operators, formalizes them, and applies them to a case study provided by an industrial collaborator.

The focus of RELAX is on structured natural language requirements. Typically, textual requirements prescribe behaviour 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. Therefore, it may be necessary to make trade-offs between *SHALL* statements 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 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*".

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 behaviour. This information is specified 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 identify properties that are directly observable and contribute information towards determining the state of the environment. RELAX is intended to be used at the software requirements phase once hardware constraints have already been defined. In particular, physical sensors (denoted by **MON**) are assumed to be known.

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** comes 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.

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.

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

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 *SHALL* statements below.

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 analyze 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 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, 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 incorporate explicit 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 environmental condition, 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) Can the requirement be made more flexible in order to maintain its satisfaction? If (ii), then the requirement is augmented to use the RELAX vocabulary and to include aspects to monitor and aspects of the environment.

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 to obtain requirement S1' as given below.

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

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 by the customer stakeholder.

Consider the RELAX-ed requirement for S2. 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* and *EVENTUALLY* – to specify that temporary business contact data inconsistencies can be tolerated.

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 (note that we are factoring in the part controlled by the system as well as environmental uncertainty).
MON: network sensors; device sensors
REL: network and device sensors provide number of consistent devices and time

The RELAX-ed requirements declaratively specify behavioural and environmental uncertainty: behavioural uncertainty by RELAX operators applied to *SHALL* statements; environmental uncertainty by defining **ENV**, **MON**, and **REL**. 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.

3. RELAX Syntax and Semantics

The syntax and semantics of RELAX are presented next.

3.1. Syntax

RELAX expressions are 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.

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

In the last three clauses ϕ must be quantifiable. It is straightforward to rewrite RELAX textual requirements in terms of this grammar, and thereby making RELAX requirements amenable to tool support [11]. As an example, S2' from Section 2 would be represented as: *SHALL* (*AS MANY AS POSSIBLE* p'); *EVENTUALLY* (*SHALL* q'), where p' denotes "distribute data to all connected devices to ensure they are using the same data at all times", 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.

3.2. Semantics

The semantics of RELAX expressions is defined in terms of fuzzy branching temporal logic (FBTL) [17]. FBTL can describe a branching temporal model with uncertain temporal and logical information. It is the representation of uncertainty in FBTL that makes it suitable

as a formalism for RELAX. For example, the statement *AS EARLY AS POSSIBLE AFTER* e ϕ expresses a 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 are not sufficient. One could express a probability that ϕ becomes true within a specified time threshold after the occurrence of e, but fuzzy logic additionally enables one to express uncertainty about what the time threshold is. Note fuzziness and probability are distinct concepts [17].

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 $m : A \rightarrow [0, 1]$.

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 triangular, in the sense that its membership graph describes a triangle with a vertex showing membership of 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. The concept of fuzzy number is easily 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 [17].

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.

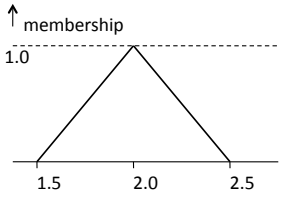

Figure 1. Fuzzy numbers – "roughly 2".

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, $\mathbf{F}p = \mathit{true}\ \mathcal{U}\ p$ 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 **F** and **G** followed by $\mathcal{X}$ and $\mathcal{U}$. The logical operator $\neg$ is next, followed by $\wedge$.

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. Definitions for those operators including uncertainty rely on a fuzzy duration or a fuzzy set. These typically have a maximum at a particular point and then tail off gradually *ad infinitum* – i.e., they have a triangular membership graph that is asymptotic. For example, in the case of *AS EARLY AS POSSIBLE* , the membership function has its maximum at the current time. 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.

We briefly explain the semantics in Table 2. The first four entries are standard. *BEFORE* and *AFTER* are defined in terms of the FBTL delay operator. In the former case, e_d defines a (crisp) duration up until event e next occurs. The formalization, therefore, expresses that before this duration has passed (i.e., before e next occurs), ϕ holds. Similarly, for *AFTER*. *AS EARLY AS POSSIBLE* is defined using a fuzzy duration whose membership function is maximum at 0 and

RELAX Expression	Informal	FBTL Formalization
SHALL ϕ	ϕ is true in any state	**AG**ϕ
MAY ϕ_1 OR *MAY* ϕ_2	in any state, either ϕ_1 or ϕ_2 is true	**AG**$(\phi_1 \vee \phi_2)$
EVENTUALLY ϕ	ϕ will be true in some future state	**AF**ϕ
ϕ_1 $\mathcal{U}$ ϕ_2	ϕ_1 will be true until ϕ_2 becomes true	**A**$(\phi_1$ $\mathcal{U}$ $\phi_2)$
BEFORE e ϕ	ϕ is true in any state occurring prior to event e	**A**$\mathcal{X}_{<e_d}$ ϕ 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	**A**$\mathcal{X}_{>e_d}$ ϕ
IN t ϕ	ϕ is true in any state in the time interval t	(*AFTER* t_{start} $\phi \wedge$ *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	**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	**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	**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	**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	**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	**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

decreases continuously for values greater than 0. Therefore, the expression $\mathcal{X}_{\geq d}\phi$ expresses that ϕ holds after some uncertain delay and that the fuzziness of the delay is such that it is most likely to be zero. This expression captures the intuition that ϕ holds *as early as possible* (i.e., as close as possible to a zero delay). Similarly, for *AS LATE AS POSSIBLE*. The remaining cases use fuzzy durations in the same manner. For *AS CLOSE AS POSSIBLE TO*, the duration's membership function has maximum value periodically with a period defined by f. In the last three entries, a fuzzy set is used instead of a fuzzy duration because the uncertainty is in the value of a quantity, not in the duration of a delay. However, the principle is the same: the membership function defines the "ideal" count and its triangular function is defined to decrease around the ideal.

4. A Process for Applying RELAX

The RELAX process assumes 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 behaviour.

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 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 (denoted by **REL**) except in cases when environmental properties cannot be directly sensed.
- Requirements often make competing demands on resources. Thus requirements have inter-dependencies (**DEP**) that must be understood 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 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 behaviour flexibility at run time. Sources for uncertainty include: contention for resources, adverse environmental conditions, timing of events, and the duration of conditions.

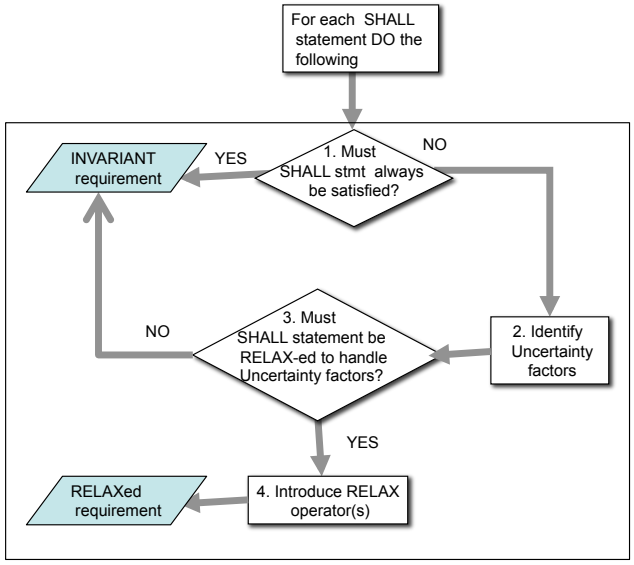

Figure 2. RELAX Process

Note that the process describes a way of iteratively and incrementally building up a model of the environment, where the starting point is a prose list of requirements. In contrast, in more recent work [4], we developed a goal-based approach to identifying requirements of a DAS, where we create a conceptual domain model that identifies the specific elements of the environment relevant to the system under development, thus scoping the uncertainty of the system. RELAX-ation is considered for each identified requirement. For both approaches, it is 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:

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.

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 from Section 4 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. Space permits us to present only a few illustrative requirements.

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.

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

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.

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* minimize latency during emergency operation.

Figure 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, where it might be preferable to divert resources from the intelligent fridge to support emergency functions (e.g., to satisfy **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 **1.1** gives the following requirement **R1.1'**:

R1.1': The fridge *SHALL* detect and communicate information with *AS MANY* food packages *AS POSSIBLE*.
ENV: Food locations & food information.
MON: RFID tags; Cameras; Weight sensors.
REL: RFID tags provide food locations/food information; 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 **R1.5**.

The **DEP** attribute provides a place to note 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 RELAX helping to 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 this incompleteness 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** (**ENV**, **MON**, **REL** and **DEP** attributes suppressed for brevity).

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.

Note that relaxing **R1.2** affects satisfaction of **R1**. The same is true of **R1.3** that was relaxed because the system could tolerate temporarily relaxing the requirement to monitor Mary's liquid intake. However, **R1.3** should eventually be satisfied otherwise Mary's health will be in jeopardy.

R1.3': The system *SHALL* ensure *AS CLOSE AS POSSIBLE TO* a minimum of liquid intake. The system *SHALL* ensure minimum liquid intake *EVENTUALLY*.

Of the remaining requirements in our list, **R1.4** and

R1.5 were deemed invariant requirements, given that the minimization requirements already expressed flexibility. Hence, it was enough simply to reformulate them to use the RELAX syntax. Thus **R1.4** became:

> **R1.4':** The system *SHALL* consume *AS FEW* units of energy *AS POSSIBLE* during normal operation.

6. Related Work

Recently, there has been a surge of interest in software engineering research for self-adaptive systems [3]. For requirements engineering, Berry *et al.* [1] 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 [9], [15], [18], [26] as well as for the specification of monitoring and switching between adaptive behaviours [20]. 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., [9], [16]).

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 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 likely be used in a complementary fashion. For example, RELAX could be integrated with goal modeling to *relax* goals in a non-enumerative way [4].

Irrespective of how a DAS's requirements are specified, the requirements must be properly integrated into a run-time requirement monitoring and adaptation infrastructure. Run-time monitoring dynamically assesses the conformance of run-time behaviour to the specified requirements [7], [8]. At run-time, a monitor runs concurrently with the system to detect violations of monitored assertions [21] and informs the choice of run-time adaptation. Using the ReqMon framework [19], for example, a DAS can send a warning when the system behaviour has violated the system requirements. RELAX-ed requirements could be mapped to monitors specified using monitor specification languages provided by tools and frameworks like ReqMon. Our vision is to enable a DAS to dynamically (i.e. during execution) reason about its own requirements and goals (a notion termed "requirements reflection" by Finkelstein [5]). 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.* [22], which monitors goals at run-time and applies AI diagnostic theories to diagnose failed goals.

Software engineering activities other than 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 [13] which discusses the state-of-the-art in software architecture for DASs. Work has also been done in providing assurance for adaptive systems [2], [12], [14], [27]–[29]. AI techniques for implementing DASs include approaches building on model-based diagnosis [6] and planning (e.g., [25]).

7. Conclusions

This paper presented a new requirements specification language called RELAX designed to explicitly address uncertainty for specifying the behaviour of dynamically adaptive systems. RELAX has three types of operators to handle uncertainty: temporal, ordinal, and modal. We focus on uncertainty due to the (possibly unexpected) changing conditions of the execution environment that require the system to adapt in order to ensure acceptable system behaviour. 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 behaviour. 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 behaviour. The invariants provide a point of reference for adaptive behaviour. 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. [23]). 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 impacts the effectiveness of the DAS.

Many possible directions for future work are possible. Additional uncertainty factors may facilitate the RELAX process. For example, we are using a combination of goal-based and threat modeling to identify sources of uncertainty explicitly [4]. We are studying how RELAX specifications can be used to guide the automatic generation of software models for adaptive systems, such as the evolutionary computation approach by Goldsby and Cheng [9]. We are also

exploring specification patterns for RELAX to be used in conjunction with Spider [11], a natural-language interface for specification patterns and analysis tools to further facilitate the use of RELAX. Finally, we are collaborating with a number of research groups to investigate the use of RELAX on requirements for adaptive systems, where traditional requirements languages have been used to describe requirements (e.g., natural language, use cases, and goal models).

Acknowledgments

The authors would like to thank LIUPPA Universit de Pau et des Pays de l'Adour for supporting this work in its early stages. In addition, Cheng is being supported in part by NSF grants EIA-0000433, CNS-0551622, CCF-0541131, IIP-0700329, CCF-0750787, Army Research Office, Ford Motor Company, and a Quality Fund Program grant from Michigan State University

References

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

[2] J. Bradbury, J. Cordy, J. Dingel, and M. Wermelinger. A classification of formal specifications for dynamics architectures. In *Proc. of ACM SIGSOFT 2004 12th FSE*, 2004.

[3] B. H. C. Cheng, H. Giese, P. Inverardi, J. Magee, and R. de Lemos. Software engineering for self-adaptive systems: A research road map, Dagstuhl-seminar on software engineering for self-adaptive systems. 2008.

[4] B. H. C. Cheng, P. Sawyer, N. Bencomo, and J. Whittle. A goal-based modeling approach to develop requirements for adaptive systems with environmental uncertainty. Technical Report MSU-CSE-09-22, Computer Science and Engineering, Michigan State University, Lancaster University, UK, East Lansing, Michigan, May 2009. (submitted for publication).

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

[6] J. De Kleer, A. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56(2-3):197 – 222, 1992.

[7] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. In *Proc. 9th Workshop Software Specification and Design*, pages 50–59, Apr 1998.

[8] S. Fickas and M. Feather. Requirements monitoring in dynamic environments. In *2nd IEEE International Symposium on Requirements Engineering (RE'95)*, 1995.

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

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

[11] S. Konrad and B. H. C. Cheng. Facilitating the construction of specification pattern-based properties. In *Proc. of International Requirements Engineering Conference (RE05)*, pages 329–338, Paris, France, August 2005.

[12] J. Kramer and J. Magee. Analysing dynamic change in software architectures: a case study. In *Proc. 2th International Conference on Configurable Distributed Systems*, pages 91–100, Annapolis, MA, 4–6 1998. IEEE.

[13] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering*, pages 259–268, 2007.

[14] S. S. Kulkarni and K. N. Biyani. Correctness of component-based adaptation. In *Proc. Component Based Software Engineering*, pages 48–58, 2004.

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

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

[17] S. Moon, K. Lee, and D. Lee. Fuzzy branching temporal logic. *Systems, Man, and Cybernetics, Part B, IEEE Transactions on*, 34(2):1045–1055, April 2004.

[18] M. Morandini, L. Penserini, and A. Perini. Modelling self-adaptivity: A goal-oriented approach. In *Proc. of 2Second Conf. on Self-Adaptive and Self-Organizing Systems (SASO '08)*, pages 469–470, 2008.

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

[20] M. Salifu, Y. Yu, and B. Nuseibeh. Specifying monitoring and switching problems in context. In *Proc. Requirements Engineering Conference (RE)*, pages 211–220, 2007.

[21] A. van Lamsweerde. Requirements engineering: from craft to discipline. In *Proc. 16th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, pages 238–249, 2008.

[22] Y. Wang, S. McIlraith, Y. Yu, and J. Mylopoulos. An automated approach to monitoring and diagnosing requirements. In *22nd Automated Software Engineering Conference (ASE 2007)*, pages 293–302, Atlanta, Georgia, 2007.

[23] K. Welsh and P. Sawyer. When to adapt? identification of problem domains for adaptive systems. In *Requirements Engineering Foundations for Software Quality (REFSQ)*, 2008.

[24] J. Whittle, P. Sawyer, N. Bencomo, and B. H. C. Cheng. Reassessing languages for requirements engineering of self-adaptive systems. In *RE Workshop SOCCER08*. 2008.

[25] B. Williams, M. Ingham, S. Chung, and P. Elliot. Model-based programming of intelligent embedded systems and robotic space explorers. *In Proc. IEEE: Special Issue on Modeling and Design of Embedded Software*, 91(1):212 – 237, 2003.

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

[27] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *In Proc. 28th International Conference on Software Engineering ICSE '06*, pages 371–380, 2006.

[28] J. Zhang and B. H. C. Cheng. Using temporal logic to specify adaptive program semantics. *Journal of Systems and Software (JSS), Architecting Dependable Systems*, 79(10):1361–1369, 2006.

[29] J. Zhang, H. Goldsby, and B. H. C. Cheng. Modular verification of dynamically adaptive systems. In *Proc. 8th International Conference on Aspect Oriented Software Development*, pages 161–172, 2009.