2010 18th IEEE International Requirements Engineering Conference

Fuzzy Goals for Requirements-driven Adaptation

Luciano Baresi and Liliana Pasquale
Politecnico di Milano
Dip. di Elettronica e Informazione
piazza L. Da Vinci, 32 – 20133, Milano (Italy)
Email: {baresi | pasquale}@elet.polimi.it

Paola Spoletini
Università dell'Insubria
Dip. di Informatica e Comunicazione
via Ravasi, 2 – 21100, Varese (Italy)
Email: paola.spoletini@uninsubria.it

***Abstract*—Self-adaptation is imposing as a key characteristic of many modern software systems to tackle their complexity and cope with the many environments in which they can operate. Self-adaptation is a requirement per-se, but it also impacts the other (conventional) requirements of the system; all these new and old requirements must be elicited and represented in a coherent and homogenous way. This paper presents FLAGS, an innovative goal model that generalizes the KAOS model, adds *adaptive* goals to embed adaptation countermeasures, and fosters self-adaptation by considering requirements as *live*, runtime entities. FLAGS also distinguishes between *crisp* goals, whose satisfaction is boolean, and *fuzzy* goals, whose satisfaction is represented through fuzzy constraints. Adaptation countermeasures are triggered by violated goals and the goal model is modified accordingly to maintain a coherent view of the system and enforce adaptation directives on the running system. The main elements of the approach are demonstrated through an example application.**

***Keywords*-Goals, KAOS, Self-Adaptation, Fuzzy Goals**

I. Introduction

Self-adaptation is becoming a key feature of many software systems to tackle their increasing complexity and cope with the environments in which they are deployed. Systems must (self-)adapt because of new business needs, but also because of the volatility of their environments, which may lead to violating some requirements satisfied at deployment time. Many approaches [1], [2] have already tried to embed self-adaptation capabilities in existing systems or add them to new ones, but only few attempts [3], [4] have addressed this problem from the beginning. Adaptation capabilities should be identified as requirements per-se and must be related to the other, "conventional", requirements of the system. This allows us to identify which requirements need a specific adaptation when they are not fulfilled satisfactorily and guarantee that adaptations are coherent with the expectations of the stakeholders.

To address these issues, the paper presents FLAGS (Fuzzy Live Adaptive Goals for Self-adaptive systems), which is an innovative goal model able to deal with the challenges posed by self-adaptive systems. Goal models have been used for representing systems' requirements, and also for tracing them onto their underlying operationalization. However, goals do not directly address adaptation. To do this, one should describe how goals cope with changes and how they can modify themselves and, if needed, the whole goal model.

FLAGS generalizes the basic features of the KAOS model [5] (i.e., refinement and formalisation), and adds the concept of *adaptive* goal. These goals define the countermeasures that one must perform if one or more goals are not fulfilled satisfactorily. Each countermeasure produces changes in the goal model. Countermeasures may prevent the actual violation of a requirement, enforce a particular goal, or move to a substitute one. The selection at runtime depends on the satisfaction level of related goals and the actual conditions of the system and of the environment. As for goals' satisfaction, a crisp notion (yes or no) would not provide the flexibility necessary in systems where some goals cannot be clearly measured (soft goals), properties are not fully known, their complete specification —with all possible alternatives— would be error-prone, and small/transient violations must be tolerated. This is why, besides crisp goals, we also propose *fuzzy* goals, specified through fuzzy constraints, that quantify the degree x ($x \in [0, 1]$) to which a goal is satisfied/violated.

Moreover, goal models have been traditionally conceived as design-time abstractions, but this is not enough if we think of runtime changes and we want to keep the running system aligned with its business requirements. If we want to pursue the alignment between requirements and runtime entities, and we want to delegate decisions about adaptation to the goal model, we need to turn goals into *live* entities, as originally proposed by Kramer and Magee [6]. The idea is that the runtime infrastructure [7], not presented in this paper, feeds the goal model and triggers changes in its parts. These changes are translated into actual adaptation directives onto the running system.

This paper focuses on the description of fuzzy and adaptive goals; it only provides a general overview of how these goals can be handled as runtime abstractions. These concepts are exemplified through a simple laundry system that controls a set of washing machines. The system must decide how to use the different machines, select washing

This research has been funded by the European Commission, Programmes: IDEAS-ERC, Project 227977 SMScom, and FP7 Network of Excellence 215483 S-Cube.

programs, and activate washing cycles. The overall goal is to minimize energy consumption while accomplishing all tasks. The paper is organized as follows. Section II motivates our approach, and introduces the laundry system example. Section III illustratess the syntax and semantics of the language adopted to formalize fuzzy goals. Section IV describes adaptive goals and Section V explains the reasons why goals must be runtime abstractions. Section VI discusses related proposals and Section VII concludes the paper.

II. Overall Approach

This section introduces our approach and the motivations behind it through a simple laundry system. The aim is to introduce the key elements and also highlight the reasons why we decided to extend KAOS.

Figure 1 presents a KAOS model of the laundry system. The general objective of the system is to wash clothes (goal G1), which is AND-refined into the following subgoals: setup the washing machine (goal G1.1), complete a washing cycle (goal G1.4), consume a small amount of energy (goal G1.2) and keep the number of clothes that remain to be washed low (goal G1.3). The setup of a washing machine requires that clothes be inserted in the drum (goal G1.1.1), powder be added (goal G1.1.2), a washing program be selected (goal G1.1.3), and the washing machine be ready (goal G1.1.4). The completion of a washing cycle requires that the selected program be started (goal G1.4.1) and that the washing machine be emptied when it terminates the execution (goal G1.4.2).

Goals G1.2 and G1.3 are soft goals since there is not a clear-cut criterion to decide whether they are satisfied or not, that is, whether the energy consumption is "small" or the number of dirty clothes is "low". For this reason, these goals can only be partially fulfilled. Goal G1.3 is also in conflict with goal G1.2 (see the dotted line in Figure 1) since reducing the number of dirty clothes may imply the execution of more washing programs and thus more energy. Conflicting goals are usually associated with different priorities. In our case, goal G1.3 has a higher priority ($p = 4$) than goal G1.2 ($p = 2$), since reducing the amount of dirty clothes is more important than constraining energy consumption.

These goals use the following entities: *WashingMachine(id, program, powder, drumEmpty, washDuration, sens, state: {default, free, selected, ready, washCompleted})*, *Program(name, duration)*, *EnergyUnits(units)*, and *DirtyClothes(amount)*. As for events, we have *program_selected*, which signals that a program is selected for a washing machine, and *start_program*, which indicates that a particular program has just started on a washing machine.

KAOS allows one to formalize goals in LTL (Linear Temporal Logic, [8]). For example, Table I presents the formalization of the goals introduced for the example[1].

Goal	Formal definition
G1.1.1	$wm : WashingMachine,$ $@(wm.state = selected) \Rightarrow \Diamond_{t<x}(\neg wm.drumEmpty)$
G1.1.2	$wm : WashingMachine,$ $@(wm.state = selected) \Rightarrow \Diamond_{t<y}(wm.powder)$
G1.1.3	$wm : WashingMachine, p : wm.program,$ $(p.name = \text{``''}) \wedge @(wm.state = selected) \Rightarrow$   $\Diamond_{t<z}(wm.program \neq \text{``''} \wedge p.duration > 0)$
G1.1.4	$wm : WashingMachine,$ $((wm.state = selected) \wedge (\neg wm.drumEmpty)$ $\wedge(wm.program \neq \text{``''}) \wedge (wm.powder)) \Rightarrow$ $\circ(wm.state = ready)$
G1.2	$e : EnergyUnits,\ \Box(e.units \leq E_{MAX})$
G1.3	$dc : DirtyClothes,\ \Box(dc.amount < 5)$
G1.4.1	$wm : WashingMachine, p : wm.program$ $@(wm.state = ready) \Rightarrow$ $\Diamond_{t \leq p.duration}(@(wm.sens = \text{``green ''}) \wedge$ $@(wm.state = washCompleted))$
G1.4.2	$wm : WashingMachine,$ $(wm.state = washCompleted) \Rightarrow$ $\Diamond(wm.program = \text{``''} \wedge wm.state = free \wedge$ $\neg(wm.powder) \wedge \neg(wm.drumEmpty)$

Table I
FORMALIZATION OF SOME EXAMPLE GOALS.

Goal G1.1.3 must be achieved when a washing machine has been selected to perform a washing cycle but a washing program has not been chosen yet. In this case, a washing program must be selected and its duration must be set properly. This goal is achieved through operation Op3 [Select Program] defined as follows[2]:

Name:	$Op3$
In/Out:	$p : Program, wm : WashingMachine$
DomPre:	$wm.state = selected$
ReqPre:	$wm.program.name = \text{``''}$
TrigPre:	$program\_selected(wid, p)$
ReqPost:	$wm.program.name = p.name \wedge$ $wm.program.duration = p.duration \wedge$ $wid = wm.id$

This operation is activated when a washing program is selected for a particular washing machine (see $TrigPre$). Parameter p of event $program\_selected$ comprises the selected washing program ($p.name$) and its corresponding duration ($p.duration$). This information must be assigned to the washing machine chosen to perform the washing cycle (see $ReqPost$).

Goal G1.4.1 states that if a washing machine becomes ready to perform a washing program ($@(wm.state = ready)$), it must terminate it within a delay lower than that specified in the program duration ($wm.washDuration$). A program is completed when the sensor of the washing

[1]Operator @ has the following meaning [5]: $@P \equiv \bullet(\neg P) \wedge \circ P$

[2]Operations are expressed in terms of their input/output entities (In-/Out), domain preconditions (DomPre), domain postconditions (DomPost), required preconditions (ReqPre), triggering preconditions (TrigPre), and required postconditions (ReqPost).

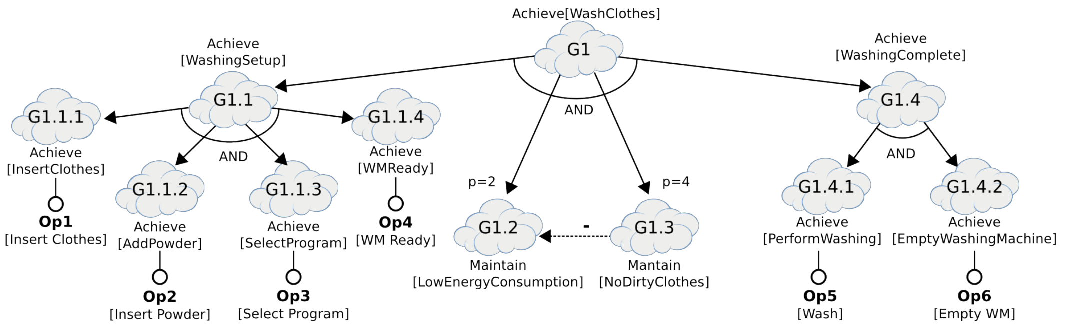

Figure 1. The KAOS goal model of the laundry system.

machine becomes green ($wm.sens = 'green'$). This goal is achieved through operation Op5 [Wash], which is defined as follows:

$$\begin{aligned}
\textbf{Name:}\ & Op5 \\
\textbf{In/Out:}\ & wm : WashingMachine \\
\textbf{DomPre:}\ & wm.state = ready \\
\textbf{DomPost:}\ & wm.state = washCompleted \\
\textbf{TrigPre:}\ & start\_program(wid, t) \wedge wm.id = wid \\
\textbf{ReqPost:}\ & wm(t').sens = \text{“}green\text{”} \wedge \\
& \wedge\, t' - t < wm.program.duration
\end{aligned}$$

The operation changes the state of the washing machine from $ready$ to $washCompleted$ (see $DomPre$ and $DomPost$). This operation is triggered when a washing program is started (event $start\_program(wid, t)$) for the specified washing machine ($wm.id = wid$).

Goals G1.2 and G1.3 rely on the operations shown in Figure 1 (i.e., Op1,..., Op6). To enforce the satisfaction of these goals we must add the following required post-conditions to each operation:

$$\begin{aligned}
\textbf{ReqPost}_{\mathbf{G1.2}}\textbf{:}\ & e.units < E_{MAX} \\
\textbf{ReqPost}_{\mathbf{G1.3}}\textbf{:}\ & dc.amount < 5
\end{aligned}$$

A. From KAOS to FLAGS

If we formalize goals in LTL, we can only assess whether they are fulfilled or not, and there is no way to say *“how much”* a goal is satisfied/violated. This may be sufficient for hard goals, but it is not satisfactory for soft-goals that can be satisfied up to a given degree, or when we need to tolerate small violations. For example, LTL is good for goal G1.1.3 since we are only interested in knowing whether a program is selected in z time units. In contrast, the adoption of LTL to formalize goal G1.4.1 would not be the best option since we may want to tolerate the case in which the washing machine terminates a program a bit later than expected ($@(wm.sens =$ “$green$”, at time $z' : z' - z = \epsilon$). This corresponds to considering these cases as “weak” violations. Furthermore, if we were able to track the level of satisfaction of soft goals, like G1.2 and G1.3, we would be able to adjust the behavior of the system accordingly. For example, since goal G1.2 (amount of energy consumed) and goal G1.3 (washed clothes) are in conflict, one may try to find a compromise between the two. The less critical requirements (G1.2) can be relaxed more than critical ones (G1.3) to provide viable solutions.

This is why FLAGS distinguishes between crisp and fuzzy goals. The fulfillment of the former is boolean, while the latter can be satisfied up to a certain level ($x \in [0, 1]$). Crisp goals are rendered in LTL, while our proposal for fuzzy goals is presented in Section III. These two types of goals can easily coexist: crisp goals represent firm requirements, while fuzzy goals are more flexible.

During requirements elicitation we must also define how the system adapts itself at runtime. For example if goal G1.4.1 is violated because the washing machine turns off suddenly during a washing cycle, we must turn the washing machine on again. We must also restore the system in a state where the drum is already filled, the powder must be added and the program set again. This is the goal that states how the system should adapt itself by applying a suitable countermeasure, and thus we call it *adaptive* goal. Each countermeasure is associated with an event that *trigger*s it execution (goal G1.4 violated), a *condition* for its actual activation (the washing machine turns off suddenly), an *objective* it has to achieve (turn on the washing machine), and a sequence of basic *actions* that must be performed on the goal model, and the underlying system, to achieve the objective.

III. Crisp and Fuzzy Goals

This section presents the language we propose to specify goals. Crisp goals are still defined in LTL, while fuzzy goals use a fuzzy temporal language. The two options are unified in a single language whose grammar is presented below[3].

[3]As commonly done for LTL, we adopt the set of natural numbers as temporal domain. Furthermore, since we suppose that crisp and fuzzy untimed formulae are defined over finite domains, universal (∀) and existential (∃) quantifiers can be added without augmenting the expressive power of the language.

freq ::= $\mathcal{G}$freq | $\mathcal{F}_{<t}$freq | $\mathcal{G}_{\bullet t}$freq | $\mathcal{G}_{-x}$freq |
freq $\mathcal{U}$ freq | treq | fprop
fprop ::= fprop $\curlywedge$ fprop | $\sim$fprop | expr comp expr
expr ::= const | var | expr op expr
comp ::= $\succ$ | $\prec$ | $\succeq$ | $\preceq$ | $\approx$ | $\not\approx$
op ::= + | − | ∗ | /

$\mathcal{G}_{\bullet t} \in \{\mathcal{G}_{<t}, \mathcal{G}_{\leq t}, \mathcal{G}_{>t}, \mathcal{G}_{\geq t}\}$. $freq$ is a fuzzy temporal formula (see operators $\mathcal{G}$, $\mathcal{F}_{<t}$, $\mathcal{G}_{\bullet t}$, $\mathcal{G}_{-x}$, $\mathcal{U}$), a crisp temporal formula ($treq$) or a fuzzy untimed formula ($fprop$). $const$ represents a constant value and var indicates a variable. Since crisp formulae are LTL expressions, there is no need to further discuss them.

The semantics of the fuzzy language is inspired by the theory of fuzzy sets, originally proposed by Zadeh [9]. It adopts a membership function (μ) that assigns a degree of truth (codomain, $y \in [0,1]$) to each proposition. Possible values are: absolutely true (1), absolutely false (0), or an intermediate degree of truth (a value in $(0,1)$). The semantics of fuzzy relational operators ($\succ$, $\prec$, $\succeq$, $\preceq$, $\approx$, $\not\approx$) is shown in Figure 2(a); the corresponding crisp operators are presented in Figure 2(b). Relational operators can be described as impulse functions or one or more step functions. For example, $x = 0$ is absolutely true only in 0, that is the point in which the constraint is verified; it is absolutely false in all the other points. Similarly $x < 0$ is absolutely true in $]-\infty, 0[$ and is absolutely false in $[0, \infty[$. Fuzzy relational operators use a smooth function to assign a degree of satisfaction between 0 and 1 to those propositions that do not fully respect the condition, but are close to it. For example, $x \approx 0$ is absolutely true for the points close to 0 ($[-1,1]$), has a degree of truth between 0 and 1 in the points near 0 (e.g., $[-4,-1) \cup (1,4]$), and is absolutely false elsewhere. Note that give these definitions, crisp membership functions are a special case of the fuzzy ones and can only assume values 0 and 1. This means that fuzzy goals can be considered a generalization of crisp ones.

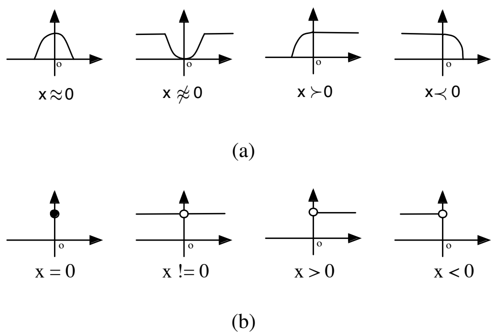

Figure 2. (a) Fuzzy membership functions for relational operators and (b) Crisp membership functions for relational operators.

This language allows us to modify the definition of goal G1.3 and consider the amount of dirty clothes explicitly as severity of the violation. Needless to say, the highest the number is, the worse the violation is. The goal can be redefined as follows:

$$\mathsf{G}\ dirty\_clothes \preceq 5$$

and the membership function of the relational operator $\preceq$ is similar to the fourth function shown in Figure 2(a), where the domain is the amount of dirty clothes and the points in which the membership function is completely satisfied are those in $(-\infty, 5]$, while it gradually decreases its truth degree for those points in $(5, \infty)$.

Different semantics [9] are available for fuzzy boolean connectives (and $\curlywedge$)) and the connectives derived from them (or $\curlyvee$, and implies $\Rrightarrow$). For operator not $\sim$ we adopt the definition shown in equation 1. For operator $\curlywedge$ we can decide between the definitions provided in equations 2 and 3. The actual choice depends on the application we are dealing with, since the former definition assigns a more optimistic degree of satisfaction to the values of the domain than the other.

$$\mu(\sim \pi) = 1 - \mu(\sim \pi) \quad (1)$$
$$\mu(\pi_1 \curlywedge \pi_2) \equiv min(\mu(\pi_1), \mu(\pi_2)) \quad (2)$$
$$\mu(\pi_1 \curlywedge \pi_2) \equiv \mu(\pi_1) * \mu(\pi_2) \quad (3)$$

Once a semantics for operators $\sim$ and $\curlywedge$ is chosen, operator $\curlyvee$ is obtained by applying the classical De Morgan law, and implication ($\Rrightarrow$) is computed as a residuum of a T-norm. An intuitive semantics for temporal operators is given in Table II, where crisp and fuzzy temporal operators are described in natural language (*NL expr*).

Besides considering the fuzzy connectives and propositions, we are also interested in adding an intrinsic vagueness to temporal operators. Due to lack of space, in this paper we only discuss the semantics of the temporal operators that are required in the laundry system example[4]. The semantics is defined in an operational way through function $fEval(freq, i)$ that evaluates every possible fuzzy formula at instant i by adopting other auxiliary functions.

According to the traditional semantics of fuzzy temporal operators [10], evaluating $\mathcal{F}_{<t}freq$ at instant i would be equivalent to finding the most satisfactory truth value of $freq$ in the interval $[i, i+t)$. This interpretation is not satisfactory for us since we also want to take into account the situations in which $freq$ has a high value of truth for a time instant x slightly greater than $i+t$. For example, for goal G1.4.1, where we want to tolerate the fact that a washing machine takes a bit more time to complete a washing program. This corresponds to evaluating the satisfaction of goal G1.4.1 by taking into account also some time instants greater than $(i + wm.program.duration)$ for which proposition $(wm.sens = 'green')$ is true.

[4]The complete semantics of the language is available at http://home.dei.polimi.it/pasquale/publications/FuzzyLanguage.pdf

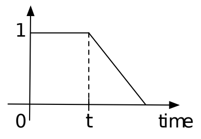

Figure 3. Membership function $\mu_{\mathcal{F}_{<t}}$.

To this aim, we need to fuzzify the temporal interval $[i, i+t)$ by introducing a membership function $\mu_{\mathcal{F}_{<t}}$ that is evaluated on the offset between the current instant (x) and i. In particular, $\mu_{\mathcal{F}_{<t}}(x)$ is equal to 1 when $x \in [0,t)$ and decreases to zero when $x \geq t$, as shown in Figure 3.

If the membership function $\mu_{\mathcal{F}_{<t}}$ is positive in the interval $[i, w)$, the operational semantics for $\mathcal{F}_{<t}(freq)$ at instant i is given by evaluating $freq$ for each instant x ($x \in [i,w)$) and by weighting it with respect to the value returned by the membership function $\mu_{\mathcal{F}_{<t}}$ at instant x. $\mathcal{F}_{<t}$ becomes the maximum among these evaluations. The semantics of operator $\mathcal{F}_{<t}$ is also expressed by the following recursive function:

```
$fEval(\mathcal{F}_{<t}(freq), i) = val1(freq, t, i, 0)$
  $float\ val1(freq, t, i, timer)\{$
    $if(\mu_{\mathcal{F}_{<t}}(timer) > 0)$
          $return\ max(\mu_{\mathcal{F}_{<t}}(timer) * fEval(freq, i),$
             $val1(freq, t, i+1, timer+1))$
    $else\ return\ 0;\}$
```

According to this semantics, goal G1.4.1 is redefined as follows:

$$@(wm.state = ready) \Rightarrow \mathcal{F}_{t<p.duration}(wm.sens = \text{“}green\text{”} \wedge @(wm.state = washCompleted))$$

Informally, it states that if the washing machine becomes ready, the washing program must complete ($wm.sens = 'green'$) within some z time instant, from that moment ($z = wm.program.duration$).

According to the literature [10] the fuzzy temporal operator $\mathcal{G}$ is evaluated in t as follows:

$$eval(\mathcal{G}freq, i) = eval(freq, i) \curlywedge eval(\mathcal{G}freq, i+1)$$

where function $eval(freq, t)$ assigns a truth value to $freq$ in i. In this definition, $\mathcal{G}$ is computed by aggregating, through operator $\curlywedge$, the truth values of $freq$ for each instant i. This implies that if the truth value of $freq$ becomes completely false (equals 0) for a certain time instant, due to a transient violation, $\mathcal{G}$ becomes 0 consequently, exactly like in the crisp case. Hence this interpretation for operator $\mathcal{G}$ does not allow us to tolerate transient violations. For example when the number of dirty clothes for certain time instants exceeds five, goal G1.3 would be violated.

Crisp op	NL expr	Fuzzy op	NL expr
G	Always	$\mathcal{G}$	Almost always
$G_{<t}$	Lasts t instants from now	$\mathcal{G}_{<t}$	Lasts hopefully t instants from now
$G_{>t}$	Always from exactly t instants	$\mathcal{G}_{>t}$	Almost always from exactly t instants
-	-	$\mathcal{G}_{-x}$	Always except at most some x cases
F	Eventually	-	-
$F_{<t}$	Within t	$\mathcal{F}_{<t}$	Within around t
$F_{>t}$	Eventually from exactly t instants	-	-
U	Until	$\mathcal{U}$	Almost Until

Table II
TEMPORAL OPERATORS.

For this reason, we adopt a different semantics: when the truth value of $freq$ is under a certain threshold (th_{min}), it is not taken into account in the evaluation of $\mathcal{G}$. Instead, it is substituted by a satisfaction value, computed by a membership function ($\mu_{\mathcal{G}}$), that depends on the number of past violations (i.e., the truth value of $freq$ is less than th_{min}). The membership function $\mu_{\mathcal{G}}$ returns a truth value that is inversely proportional to the number of violations already occurred ($\#\ errors$), as shown in Figure 4.

The semantics of operator $\mathcal{G}freq$ is described by the following function:

```
$fEval(\mathcal{G}freq, i) = val2(freq, i, 0)$
  $float\ val2(freq, i, error)\{$
    $if(fEval(freq, i) < th_{min})\{$
          $error++;$
        $if(\mu_{\mathcal{G}}(error) == 0)\ return\ 0;$
        $else\ return\ \frac{\mu_{\mathcal{G}}(error)}{\mu_{\mathcal{G}}(error-1)} * val2(freq, i+1, error);\}$
    $else\ return\ fEval(freq, i) * val2(freq, i+1, error)\}$
```

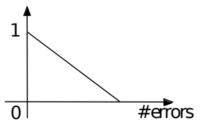

Figure 4. Membership function $\mu_{\mathcal{G}}$.

Function $fEval$, which is used to describe the semantics of $\mathcal{G}freq$, adopts an auxiliary function $val2$ that takes the sub-formula ($freq$) on which $\mathcal{G}$ is applied, the instant i at which the formula is computed, and a violation counter ($error$) initialized to 0. Every time a violation occurs, $error$ is incremented. If the updated value of $error$ makes $\mu_{\mathcal{G}}$ return a value equal to zero, it means that $\mathcal{G}freq$ has a truth value equal to 0 ($return\ 0$). Otherwise the current truth value of $\mathcal{G}freq$ is multiplied by factor $\frac{\mu_{\mathcal{G}}(error)}{\mu_{\mathcal{G}}(error-1)}$. Note that if no violation occurred, the truth value of $freq$ (given by $fEval(freq, i)$) directly affects the final evaluation of $\mathcal{G}freq$.

This way, we can redefine goal G1.3 as follows

$$\mathcal{G}\ dirty\_clothes \preceq 5$$

to say that the number of dirty clothes must almost always be less than or equal to five.

Operator $\mathcal{G}_{-x} freq$ requires that $freq$ has a satisfactory truth value with the exception of "at most some" x cases. This is useful when we want to admit a certain number of violations. For example, goal G1.2, due to peaks in energy consumption, could accept up to a maximum of some ten violations. This way, goal G1.2 can be redefined as follows:

$$\mathcal{G}_{-10}\ energy \leq E_{MAX}$$

to mean that the amount of consumed energy must always be less than or equal to E_{MAX}, but we accept up to ten exceptions.

Operator $\mathcal{G}_{-x}$ is semantically similar to $\mathcal{G}$, with the only difference that $\mathcal{G}_{-x}$ adopts a slightly modified membership function ($\mu_{\mathcal{G}_{-x}}$). This function returns 1, when it is evaluated for a number of violations in the interval $[0, x]$, and returns a value between 0 and 1 if the number of violations (k) is greater than x.

Operator $\mathcal{G}_{>t}$ has a semantics similar to that of $\mathcal{G}$, since evaluating $\mathcal{G}_{>t}$ in i is equivalent to evaluating $\mathcal{G}$ in $i + t$:

$$fEval(\mathcal{G}_{>t} freq, i) = fEval(\mathcal{G} freq, i + t)$$

Note that $fEval(fprop, i)$ evaluates a fuzzy proposition ($fprop$) by adopting the membership functions of the relational operators shown in Figure 2(a), while $fEval(treq, i)$ checks expression $treq$ according to the usual LTL semantics.

A. Tuning the membership functions

Every time the formalization of a goal introduces a new membership function, we must define its shape by considering the preferences of the different stakeholders.

Our approach considers that membership functions be limited and continuous, and in most of the cases they have a trapezoidal shape (but can also degenerate into triangles). For each of these trapezoidal functions, we must define the domain ($[d, D]$) over which it can assume values between 0 and 1. We must also specify two key points in the domain: the minimum (m) and maximum (M) values for which the corresponding crisp formula would be true (i.e., the upper parallel side of the trapezoid). These points obey to the following inequality: $d \leq m \leq M \leq D$, with the general case when $d < m \leq M < D$. The user can tune the severity of these membership functions by expressing the gradient of the segments that go from point $(x, 0)$, with $x \in [d, m)$, to point $(m, 1)$, and from point $(M, 1)$ to $(y, 0)$,

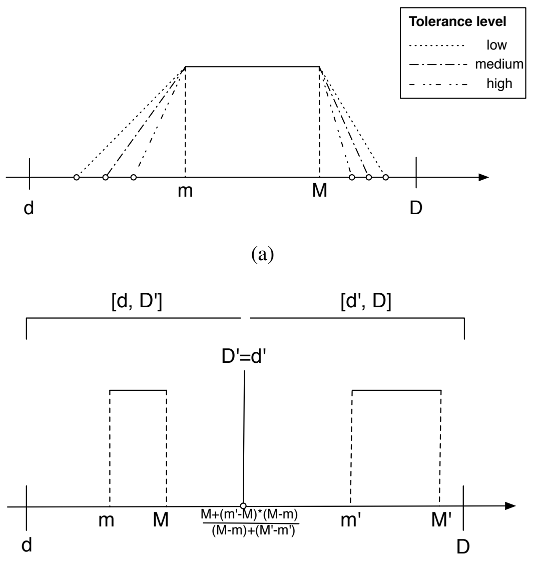

Figure 5. (a) Customization of a trapezoidal membership function and (b) Splitting of the membership function into smaller trapezoidal functions.

with $y \in (M, D]$[5]. This is equivalent to choosing a value for x and y. Non expert users can just specify a severity level among {low, medium, high} when they specify goals. This corresponds to fixing $x = \frac{m-d}{3} + d$ and $y = (D - M) * \frac{2}{3} + M$ for low, $x = \frac{m-d}{2} + d$ and $y = \frac{D-M}{2} + M$ for medium, and $x = (m - d) * \frac{2}{3} + d$ and $y = \frac{D-M}{3} + M$ for high. These possibilities are shown in Figure 5(a).

The membership function that does not present a trapezoidal shape corresponds to a formula for which the crisp version is true in more than one interval of the domain. However these cases can be reduced to the trapezoidal case by decomposing the problem into smaller trapezoidal problems obtained by segmenting the domain in subsets, each containing exactly one interval in which the membership function is one. This case is shown in Figure 5(b).

IV. Adaptive Goals

FLAGS proposes *adaptive* goals as means to conveniently describe adaptation countermeasures in a parametric way, that is with respect to the satisfaction level of the other goals or the environmental conditions. Each countermeasure comes with a set of constraints (*trigger* and *condition*s) that identify the execution points where it must be performed, an *objective* to be achieved, and a sequence of *action*s applied on the goal model to fulfill the aforementioned objective.

A *condition* specifies properties of the system (e.g., satisfaction levels of both conventional and adaptive goals, goals' priorities, countermeasures already performed) or of

[5]The user can express the gradient of only one of these segments when $d = m$ or $M = D$.

the environment (e.g. domain assumptions) that must be true to allow a specific countermeasure to be activated. A *trigger* expresses a constraint on the satisfaction of a leaf goal and activates the execution of a countermeasure if the corresponding *condition*s are satisfied too. As for *objective*, we can:

- Enforce the satisfaction of a leaf goal[6] without modifying its definition, and its membership functions;
- Enforce a weaker version of a goal by relaxing its membership function or changing its definition completely;
- Prevent situations in which the satisfaction level of goals would be too low.

Note that a countermeasure that enforces the original goal gives more guarantees than a countermeasure that enforces a weaker version of the goal, which gives more guarantees than a countermeasure that just prevents a goal from being violated. The objective of a countermeasure allows us to assess the success of an adaptation at runtime and evaluate whether other countermeasures must be performed. *Actions* can change the goal model by:

- Adding/Removing goals;
- Modifying the definition of a leaf goal (i.e., changing the membership function adopted for its evaluation);
- Adding/Removing operations;
- Modifying the pre- and postconditions of operations;
- Adding/Removing entities, events, or agents.

These actions can be tuned according to the satisfaction level of goals or the execution context, by applying more or less severe adaptations accordingly. If we deal with critical goals, or if a goal largely deviates from the desired objective, a countermeasure that enforces a goal satisfaction without modifying its membership function can be performed. While for those situations in which a goal cannot be completely satisfied (i.e., its satisfaction is always under a certain threshold), we must substitute the existing goal with a new one or relax its membership function.

Countermeasures may conflict the same ways as conventional goals do. We can have possible conflicts:

- Among countermeasures that can be applied on the same goal at the same time due to overlapping conditions;
- Among countermeasures associated with conflicting goals (the conflict is made explicit in the goal model).

For the first type of conflicts, our policy is to trigger the stronger countermeasure, while the others can only be performed if the previous one fails. If there is more than one equivalent countermeasure, all of them can be triggered at the same time in case they do not produce an incoherent goal model. As explained in Section II, conflicting goals in general are associated with different priorities (the most critical ones have the highest priorities). This allows us to solve the second type of conflicts since the countermeasures of the adaptive goals with higher priorities are triggered. Despite these potentials, the mechanism adopted to handle conflicts is still too simplistic in certain situations. For example a vicious cycle may exist when a countermeasure A has a negative side effect on another goal, and that goal's countermeasure B has a negative side effect on the first goal as well. These cases can be handled by tuning the conditions of the countermeasures involved, which would become pretty complex. For example, to solve the aforementioned situation, we should state that countermeasure A cannot be executed if it has been already performed in the past, and B was performed as an effect of A.

[6]This assumption does not reduce the expressive power of our approach since adaptations applied on higher level goals can always be reduced to violations of leaf ones [5].

The laundry system uses the adaptive goals shown in Figure 6. Adaptive goal AG1.2 is associated with goal G1.2, since it is triggered when the total amount of consumed energy is too high. AG1.2 is decomposed into a single countermeasure ($C1$). $C1$ is activated if the violation level of goal G1.2 is lower than 0.8 as stated in its associated condition ($Sat(G1.2) < 0.8$). Obviously when the satisfaction level is higher than 0.8, no countermeasures are taken. This countermeasure can enforce the satisfaction of the original goal by substituting operation $Select\ Program$ with operation $Select\ Ecoprogram$, which would choose cheap washing programs. These programs must have a duration inversely proportional to the actual amount of consumed energy, that is, $2 - \sim (Sat(G1.2))$ hours. This example highlights how adaptation can be tuned depending on the satisfaction level of stated requirements. In our case, the duration of a selected washing program will be inversely proportional to the satisfaction level of goal G1.2: the more energy is consumed, the shorter the selected washing program is. If a suitable eco-program does not exist, a washing cycle cannot be performed. Countermeasure $C1$ applies the following actions on the goal model: $removeOperation(Select\ Program)$, and $addOperation(Select\ Ecoprogram)$. This last operation is equal to operation $Select\ Program$ but the required postcondition becomes:

$$\textbf{ReqPost}_{\mathbf{G1.1.3}}\textbf{:}\quad \begin{array}{l} wm.program.name = p.name \wedge \\ wm.program.duration = p.duration \wedge \\ wm.program.duration \leq 2 - \sim Sat(G1.2) \end{array}$$

We associate adaptive goal AG1.3 with goal G1.3. AG1.3 is operationalized through 2 countermeasures ($C2$ and $C3$). $C2$ is activated when the satisfaction of goal G1.3 is lower than 0.8 (its condition is $Sat(G1.3) < 0.8$). It relaxes the membership function of goal G1.2, making it less strict, as defined in Figure 7. This way, we reduce the number of times countermeasure $C1$ is triggered, avoiding to be blocked waiting for a suitable washing program to be selected again.

Countermeasure $C3$ is activated when the satisfaction level of goal G1.3 is lower than 0.6, and if $C2$ was already

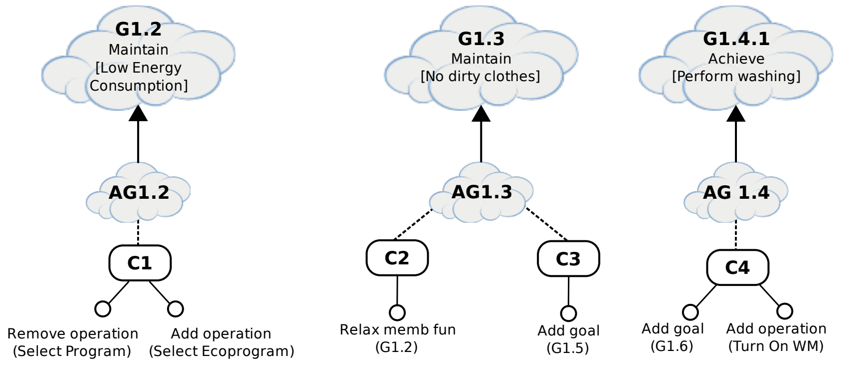

Figure 6. Adaptive goals for the laundry system

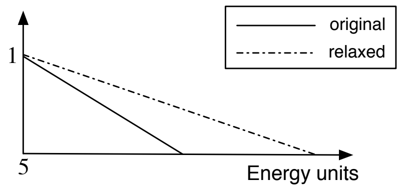

Figure 7. A relaxed version of the membership function of goal G1.2.

triggered without achieving its objective, as stated in its condition ($Sat(G1.3) < 0.6 \curlywedge \sim C2$). This countermeasure adds a new goal G1.5 to use another washing machine if there are other clothes to wash. Goal G1.5 is defined as follows:

$$\mathsf{G}((d.amount > 0 \curlywedge \sim (wm.state = free) \curlywedge \\ \exists wm' \sim (wm.id = wm'.id) \curlywedge wm'.state = free) \Rightarrow \\ \mathsf{F}(wm'.state = selected))$$

Since goals $G1.2$ and $G1.3$ are in conflict, adaptive goals AG1.2 and AG1.3 may have conflicts of type 2. In this case, only the countermeasures associated with the most critical goal are triggered (i.e., $C2$ and $C3$). Furthermore $C2$ and $C3$ can also be in conflict, but countermeasure $C2$ is triggered first since it is explicitly specified in the condition of $C3$.

Adaptive goal $AG1.4.1$ is associated with goal G1.4.1 and is operationalized through countermeasure $C4$. $C4$ is activated when the washing machine turns off suddenly during a washing cycle. This is defined in $AC5$'s condition: (*wm.state = off*). The objective of this countermeasure is to restore the satisfaction of goal G1.4.1 by turning on the washing machine and restoring it in a state in which the drum is filled, but the powder must be added newly and the program must be selected. To this aim this countermeasure adds goal G1.6, which is defined as follows:

$$\mathsf{F}(wm.state = selected \curlywedge \sim (wm.powder) \curlywedge \\ \sim (wm.drumEmpty) \curlywedge wm.program = \text{“”})$$

This goal can be operationalized through operation *Turn On WM*:

Operation:	*Turn On WM*
Input:	$wm : WashingMachine$
Output:	$wm : WashingMachine$
DomPre:	$wm.state = off$
DomPost:	$wm.state = selected$
TrigPre:	$turn\_on$
ReqPre:	$@(wm.state = off)$
ReqPost:	$turn\_on \wedge \neg(wm.powder) \wedge$ $\neg(wm.drumEmpty) \wedge (wm.program = \text{“”})$

The application of a countermeasure may also modify existing adaptive goals by eliminating some countermeasures that are no longer useful. In general, if a countermeasure removes a goal, an operation, or an object (agent/entity/event), all the countermeasures that modify these elements are invalidated. For example, if countermeasure $C1$ is applied, all countermeasures that modify or remove operation $SelectProgram$ are invalidated. Furthermore, if a countermeasure modifies/relaxes a goal's definition, some countermeasures are triggered less frequently, since the possibility of a goal to be satisfied under a certain threshold is reduced. For example, since countermeasure $C2$ relaxes the membership function of goal G1.2, countermeasure $C1$ will be triggered less frequently. Note that adding or removing countermeasures also requires that the conflicts among existing countermeasures be updated.

V. Goals at runtime

Besides being used to "generate" the actual application, the goal model is also used to oversee its runtime adaptation. A live goal model allows us to efficiently handle changes due to new requirements or modifications to the context. It also allows us to reason on countermeasures and understand whether and how they are feasible on the different application instances. Devised countermeasures behave differently in the different situations, and thus they may not achieve expected results. Similarly, designed membership functions may not correspond to the actual needs of the stakeholders. For example, countermeasure C3, which tries to use a free washing machine, may become useless if all washing machines are always busy, or conversely, the membership function to represent the constraint of goal G1.2 does not trigger an adaptation when needed, e.g., because its membership function is too optimistic. All these situations must be properly supervised at runtime. We foresee that each instance of the application is associated with a live goal model that receives field data and decides how to change the application, if needed. Field data are used to access the satisfaction of conventional goals, evaluate the triggers and conditions associated with adaptive goals, and select the actual countermeasures that must be executed to keep the application on track. Adaptation actions are properly translated into statements at application level, whose execution is triggered by the goal model. Changes can either refer to the single instance, and thus they only affect the particular instance and its goal model, or to the entire application, that is, they modify the definition of the goal model permanently, and future instances will start from the new version.

VI. Related Work

The concept of fuzzy requirements [11] is not new. Fuzziness is mainly adopted to express uncertain requirements [4], [12] and to perform trade-off analysis among conflicting functional requirements [13]. Liu et al. [12] introduce a methodology to elicit non-functional requirements through fuzzy membership functions that allows one to represent the uncertainty of human perception. Instead, Whittle et al. [4] propose RELAX, a notation to express uncertain requirements, whose assessment is affected by the imprecision of measurement, and integrate it with a goal modeling methodology [14]. The adoption of fuzziness for trade-off analysis [13] aims to identify aggregation functions to combine correlated requirements into higher-level ones. Fuzziness can also guide the selection of COTS components during requirements elicitation. For example, Alves et al. [15] use the degree of satisfaction of one or more goals to select components and analyze possible mismatches.

In this paper, we exploit fuzziness to express and assess the satisfaction degree of requirements with the idea of preventing some violations and tolerating small/transient deviations. We think that reasoning with satisfaction levels allows us to effectively select and tune adaptation. Letier et al. [16] have also investigated partial satisfaction of goals. They annotate goal refinement models with some objective functions, that are measured and evaluated with respect to a target value. The satisfaction degree of a goal is expressed in a probabilistic way based on the frequency with which its objective function is satisfied. Conversely, in our work we measure the satisfaction level in terms of the proximity of a goal to being completely satisfied, and not in terms of the probability it is satisfied.

Our approach also proposes the definition of adaptation capabilities while eliciting requirements, as it has been proposed in other works [17], [18]. Lapouchnian et al. [17] exploit alternative paths in the goal model to derive a set of possible system's behaviors. This way, when a requirement is violated, these alternatives are ready to be selected to perform adaptation. Goldsby et al. [18] use goal models to represent the non-adaptive behavior of the system (business logic), the adaptation strategies (to handle environmental changes) and the mechanisms needed by the underlying infrastructure to perform adaptation. These proposals only handle adaptation by enumerating all alternative paths at design time. In contrast, we support the continuous evolution of the goal model by tracing adaptive goals onto the underlying implementation.

Adapting the specification of the system-to-be according to changes in the context was originally proposed by Salifu et al. [19]. It was also extensively exploited in different works [20], [21] that handled context variability through the explicit modeling of alternatives. Penserini et al. [20] model the availability of execution plans to achieve a goal (called ability), and the set of pre-conditions and context-conditions that can trigger those plans (called opportunity). Ali et al. [21] detect the parameters that come from the environment (context) and associate them with specific variation points in the goal model. This way different execution plans are selected at runtime based on the actual context. Conversely, in our approach adaptation does not only refer to variation points but it allows us to apply modifications in the goal model. Furthermore the factors that can trigger such modifications depend on: (1) the context, (2) the actual satisfaction level of goals, and (3) the countermeasures already performed on the goal model.

All these works address adaptation at requirements level, but they mainly target context-aware applications and context-based adaptation. They do not consider adaptations required by the inability to satisfy some goals. A different approach is instead adopted by Wang et al. [3] who provide reconciliation mechanisms when requirements are violated. They generate system reconfigurations, guided by OR-refinements of goals, after the violation of a goal is diagnosed. They choose the configuration that contributes most positively to the non-functional requirements of the system and also has the lowest impact on its current configuration. Again, our work is different since it foresees a wider set of adaptations that can change the goal model in different ways and these changes can be traced onto the underlying system. Furthermore we allow one to tune adaptation by considering the satisfaction level of fuzzy goals.

VII. Conclusions

The paper presents FLAGS, an innovative goal model, based on KAOS, for specifying the requirements and adaptation capabilities of self-adaptive systems. Its key innovations are the transformation of goals into live entities, the distinction between crisp and fuzzy goals, with which one can associate different satisfaction levels, and the definition of adaptation strategies as if they were goals. All these elements help embed self-adaptability in software systems from the very beginning (requirements elicitation), and reason on possible consequences.

This is the first complete formalization of the goal model, and our future work will concentrate on further shaping and refining it. We will better investigate the conflict resolution mechanism among countermeasures and the training of membership functions through significant examples. In parallel, we will also investigate simpler notations to provide practitioners with a sound and usable modeling means. Finally, we plan to adopt our goal model to specify some real(istic) self-adaptive systems to gain more experience, identify further requirements for it, and empirically assess its usefulness.

VIII. Acknowledgements

The authors wish to thank John Mylopoulos for his fruitful comments and the anonymous reviewers for the suggestions that helped us improve the paper.

References

[1] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based Runtime Software Evolution," in *Proc. of the 20th Int. Conf. on Software Engineering*, 1998, pp. 177–186.

[2] S.-W. Cheng, A.-C. Huang, D. Garlan, B. R. Schmerl, and P. Steenkiste, "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," in *Proc. of the 1st Int. Conf. on Autonomic Computing*, 2004, pp. 276–277.

[3] Y. Wang, S. A. Mcilraith, Y. Yu, and J. Mylopoulos, "Monitoring and Diagnosing Software Requirements," *Automated Software Engineering*, vol. 16, no. 1, pp. 3–35, 2009.

[4] J. Whittle, P. Sawyer, N. Bencomo, and B. H. C. Cheng, "RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems," in *Proc. of the 17th Int. Requirements Engineering Conf.*, 2009, pp. 79–88.

[5] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*. John Wiley, 2009.

[6] J. Kramer and J. Magee, "Self-Managed Systems: an Architectural Challenge," in *Proc. of Future of Software Engineering*, 2007, pp. 259–268.

[7] L. Baresi, S. Guinea, and L. Pasquale, "Integrated and Composable Supervision of BPEL Processes," in *Proc. of the 6th Int. Conf. of Service Oriented Computing*, 2008, pp. 614–619.

[8] A. Pnueli, "The Temporal Logic of Programs," in *Proc. of the 18th Annual Symposium on Foundations of Computer Science*, 1977, pp. 46–57.

[9] L. A. Zadeh, "Fuzzy sets," *Information and Control*, vol. 8, no. 3, pp. 338–353, 1965.

[10] K. Lamine and F. Kabanza, "Using Fuzzy Temporal Logic for Monitoring Behaviour-based Mobile Robots," in *Proc. of IASTED Int. Conf. on Robotics and Applications*, 2000, pp. 116–121.

[11] X. F. Liu, "Fuzzy Requirements," *IEEE Potentials*, pp. 24–26, 1998.

[12] X. F. Liu, M. Azmoodeh, and N. Georgalas, "Specification of Non-functional Requirements for Contract Specification in the NGOSS Framework for Quality Management and Product Evaluation," in *Proc. of the 5th Int. Workshop on Software Quality*, 2007, pp. 36–41.

[13] X. Liu and J. Yen, "An Analytic Framework for Specifying and Analyzing Imprecise Requirements," in *Proc. of the 18th Int. Conf. on Software Engineering*, 1996, pp. 60–69.

[14] B. H. C. Cheng, P. Sawyer, N. Bencomo, and J. Whittle, "A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty," in *Proc. of the 12th Int. Conf. on Model Driven Engineering Languages and Systems*, 2009, pp. 468–483.

[15] C. Alves, X. Franch, J. P. Carvallo, and A. Finkelstein, "Using Goals and Quality Models to Support the Matching Analysis During COTS Selection," in *Proc. of the 4th Int. Conf. on COTS-Based Software Systems*, 2005, pp. 146–156.

[16] E. Letier and A. van Lamsweerde, "Reasoning About Partial Goal Satisfaction for Requirements and Design Engineering," *Software Engineering Notes*, vol. 29, no. 6, pp. 53–62, 2004.

[17] A. Lapouchnian, Y. Yu, S. Liaskos, and J. Mylopoulos, "Requirements-driven Design of Autonomic Application Software," in *Proc. of the Conf. of the Centre for Advanced Studies on Collaborative Research*, 2006, pp. 80–94.

[18] H. J. Goldsby, P. Sawyer, N. Bencomo, B. H. C. Cheng, and D. Hughes, "Goal-Based Modeling of Dynamically Adaptive System Requirements," in *Proc. of the 15th Int. Workshop on Engineering of Computer-Based Systems*, 2008, pp. 36–45.

[19] M. Salifu, Y. Yu, and B. Nuseibeh, "Specifying Monitoring and Switching Problems in Context," in *Proc. of the 15th International Requirements Engineering Conf.*, 2007, pp. 211–220.

[20] L. Penserini, A. Perini, A. Susi, and J. Mylopoulos, "High Variability Design for Software Agents: Extending Tropos," *ACM Transactions on Autonomous Adaptive Systems*, vol. 2, no. 4, pp. 75–102, 2007.

[21] R. Ali, F. Dalpiaz, and P. Giorgini, "A Goal Modeling Framework for Self-Contextualizable Software," in *Proc. of the 14th Int. Conf. on Exploring Modeling Methods in Systems Analysis and Design*, vol. 29, 2009, pp. 326–338.