Monitoring Fuzzy Temporal Requirements for Service Compositions: Motivations, Challenges and Experimental Results*

Liliana Pasquale
Lero - Irish Software Engineering Research Ceneter
University of Limerick, Limerick (Ireland)
Email: liliana.pasquale@lero.ie

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

***Abstract*—Service compositions are an important family of self-adaptive systems, which need to cope with the variability of the environment (e.g., heterogeneous devices, changing context), and react to unexpected events (e.g., changing components) that may take place at runtime. To this aim, it is fundamental to continuously assess requirements while the system is executing and detect partial mismatches or handle uncertainty. Detecting the entity of a violation is very helpful, since it can guide the way applications adapt at runtime. This paper is based on the FLAGS language we already proposed in our previous work to represent requirements as fuzzy temporal formulas and identify partial violations at the temporal level. The paper illustrates the advantages of using the FLAGS language to express the requirements of service compositions, and proposes a technique to monitor them at runtime. The experimental evaluation demonstrates that the monitoring technique is feasible and the overhead introduced in the running system is negligible.**

***Keywords*-Fuzzy Goals, Requirements Monitoring, Service-Compositions, Self-Adaptive Systems**

I. Introduction

Service compositions require to be flexible and adaptable to cope with the variability of the environment where they are executed. The adoption of different devices, such as GPS, sensors, or smartphones, the ephemerality of service components, and the modifications of the context (e.g., location, user preferences, etc.), just to mention a few, are some factors of variability. In this scenario, requirements violations or context changes may take place unexpectedly while compositions are executing. For this reason, continuous monitoring is fundamental to detect these changes and apply proper reactions accordingly. Adaptation can be performed, for example, by reconfiguring the service components that can be discovered and composed at runtime.

Monitoring results are oftentimes uncertain, due to the imprecision of data collected by sensors, or the mismatch between the representation of requirements and the real users' preferences. Requirements have been traditionally associated with a crisp semantics ("yes" or "not") and are often expressed in LTL (Linear Temporal Logic). However, a crisp semantics is not flexible enough and does not allow us to represent partial mismatches or handle uncertainty. It does not provide any information regarding how much a requirement has been violated and, indeed, it does not give any insight regarding how a composition must adapt. For all these reasons, we are convinced that different semantics and monitoring capabilities must be investigated to express and assess requirements.

This paper is based on a fuzzy temporal language, the FLAGS language [1], which we have already proposed in our previous work. FLAGS extends traditional LTL by adding new operators to express transient/small violations in the temporal domain. This allows us to monitor partial deviations that can take place, for example, when an event happens a little bit later than expected, a property is always satisfied except for a small set of time instants, or a property is not maintained for a desired time interval. The contribution of this paper is to demonstrate how the FLAGS language can represent the requirements of service compositions and handle their uncertainty. We propose a technique to monitor the FLAGS requirements at runtime and explain the challenges faced to minimize the overhead given by monitoring fuzzy temporal formulas. The experimental evaluation demonstrates that the monitoring technique is feasible and the time to asses fuzzy formulas is comparable to that necessary to check LTL properties.

This paper is organized as follows. Section II introduces the FLAGS language. Section III explains the optimization technique to improve the monitoring performance. Section IV describes the implementation and the performance of the monitoring framework. Section V illustrates related work and Section VI concludes the paper.

II. The FLAGS Language

This Section illustrates an example that motivates the adoption of the FLAGS language and introduces the syntax and semantics of the language.

A. Motivating Example

To demonstrate the potentiality of the FLAGS language we examplify its usage on *GetStream*, a service based application that provides videos in streaming offered by different providers. In this case the context includes the user

* Supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1

location, and the kind of video that can be live (e.g., a live soccer match) or registered (e.g., a movie). For example, if we consider the scenario in which a customer uses*GetStream* to watch a live soccer game. He/she prefers not to have interruptions of the video during the whole match and can also tolerate a limited number of interruptions provided that they are isolated and quite far-between. Interruptions may be concretely represented as delays in the transmission rate ($rate$) of the required video. This way, *GetStream* must satisfy the following requirements.

R1. The transmission rate must be always comprised between min_{rate} and max_{rate} (Kb/s) for all the duration of the match. The sooner an interruption takes place, the worse the violation will be.

R2. The transmission rate cannot be less than min_{rate} (Kb/s) for more than a few number of times.

R3. Every time the transmission rate becomes less than min_{rate} Kb/s, it must return to its normal values ($rate \in [min_{rate}, max_{rate}]$) almost within 1 minute. The longer an interruption takes, the worse the violation will be.

Assessing the satisfaction level of requirements may provide great insight about the adaptation action to select. For example, if interruptions happen during the first-half of the match (i.e., the satisfaction of R1 is low), *GetStream* may decide to contact another provider of the video. In case violations persist, *GetStream* may suggest the nearest bar, in the radius of 1 km, in which the match is transmitted, if possible. On the other hand, if interrruptions happen during the last minutes of the match (i.e., the satisfaction R1 is quite high) and the other requirements are violated, GetStream may consider more convenient to switch to a "play-by-play" description instead of using another provider of the video. As a matter of facts, it is not convenient to increase the provisioning costs, by contacting another provider, just to show the last minutes of the match.

B. The Language

The FLAGS language expresses requirements as fuzzy temporal formulas and assigns to them a satisfaction degree comprised between 0 and 1. Traditional FLTL (Fuzzy Linear Temporal Logic) [2] cannot embed the vagueness at the temporal level to express, for example, properties that must be verified "almost always", "within around t time instants", "always except for at most some x cases". The FLAGS language adds new fuzzy temporal operators and modifies the semantics of existing operators to overcome this limitation.

The grammar of the language is presented below [1].

$$\varphi ::= \mathcal{G}\ \varphi \mid \mathcal{F}_{\leq t}\ \varphi \mid \mathcal{G}_{\bullet t}\ \varphi \mid \mathcal{G}_{-x}\ \varphi \mid \varphi\ \mathcal{U}\ \varphi \mid \mathcal{L}_t\ \varphi \mid \mathcal{W}_t\ \varphi \mid \varphi_c \mid \pi_f$$

$$\pi_f ::= \varphi\ \curlywedge\ \varphi \mid \sim \varphi \mid \text{expr } F_{conn} \text{ expr}$$
$$F_{conn} ::= \succ \mid \prec \mid \succeq \mid \preceq \mid \approx \mid \not\approx$$
$$\varphi_c ::= \mathsf{G}\varphi_c \mid \mathsf{G}_{\bullet t}\ \varphi_c \mid \mathsf{F}\varphi_c \mid \mathsf{F}_{\bullet t}\ \varphi \mid \varphi_c\ \mathsf{U}\ \varphi_c \mid \mathsf{X}^t\ \varphi_c \mid \pi_c$$
$$\pi_c ::= \pi_c \wedge \pi_c \mid \neg\, \pi_c \mid \text{expr } C_{conn} \text{ expr}$$
$$C_{conn} ::= < \mid > \mid \geq \mid \leq \mid = \mid \neq$$
$$\text{expr} ::= \text{const} \mid \text{var} \mid \text{expr op expr}$$
$$\text{op} ::= + \mid - \mid * \mid /$$

φ is a fuzzy temporal formula (see operators $\mathcal{G}$, $\mathcal{F}_{\leq t}$, $\mathcal{G}_{\bullet t}$, $\mathcal{G}_{-x}$, $\mathcal{U}$, $\mathcal{L}_t$, $\mathcal{W}_t$), a crisp temporal formula (φ_c) or a fuzzy untimed formula (π_f). Terminals $const$ and var represent respectively a constant value and a variable. A crisp temporal formula (φ_c) follows the classical LTL syntax and can be associated with bounded/unbounded temporal operators, such as "always in the future" (G), "eventually in the future" (F), and boolean connectives (e.g., "and" ($\wedge$), and "not" ($\neg$)).

As originally proposed by Zadeh [3], the semantics of a proposition is expressed through a membership function (μ) that assigns a degree of truth (codomain, $y \in [0, 1]$) to it. The semantics of fuzzy relational operators ($\succ$, $\prec$, $\succeq$, $\preceq$, $\approx$, $\not\approx$) assigns 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. Ragarding fuzzy connectives, possible semantics for operators not ($\sim$), and ($\curlywedge$), and or ($\curlyvee$) are shown respectively in equations 1, 2 and 3.

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

The semantics of some of (fuzzy/crisp) temporal operators is intuitively expressed in natural language in Table I[2]. Note that each crisp operator is associated with its fuzzy counterpart, when possible.

Crisp op	NL expr	Fuzzy op	NL expr
G	Always	$\mathcal{G}$	Almost always
-	-	$\mathcal{G}_{-x}$	Almost always except at most some x cases
$\mathsf{F}_{<t}$	Within t	$\mathcal{F}_{<t}$	Almost within t
-	-	$\mathcal{W}_t$	Almost within around at most t instants

Table I
TEMPORAL OPERATORS.

According to the traditional semantics of fuzzy temporal operators [2], evaluating $\mathcal{F}_{<t}\,\varphi$ at an instant i is equivalent to

[1] $\bullet \in \{<, >, \geq, \leq\}$

[2] For reasons of space this paper just describes the semantics of the operators used to express the requirements of the example. Interested readers can refer to [4] for a complete description of all operators.

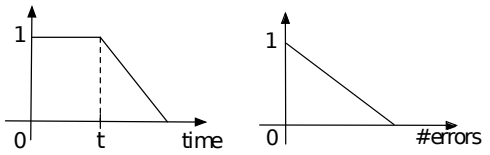
finding the most satisfactory truth value of φ in the interval $[i, i+t)$. This interpretation may be not satisfactory when φ has an high value of truth for a time instant t' slightly greater than $i+t$. For example, for R3 we may want to tolerate those situations in which the transmission rate takes a bit more than 1 minute to return to its normal range. For this reason, we added operator $\mathcal{W}_t$ to the FLAGS language.

$$\mathcal{W}_t\varphi = \curlyvee_{t'}(fEval(\varphi, t') * \mu_{\mathcal{W}_t}(t')), \;\; t' \in [i, i+w)$$

As shown above, $\mathcal{W}_t$ evaluates a formula φ for a time interval ($[i, i+w]$) a little bit longer than t, and weights each evaluation of its argument at instant t' by the value of the membership function $\mu_{\mathcal{W}_t}$ at the same instant. This membership function is evaluated on the offset between the current instant (t') and i. In particular, $\mu_{\mathcal{W}_t}(t')$ is equal to 1 when $t' \in [i, i+t)$ and decreases to 0 when $t' \geq (i+t)$, as shown in Figure 1(a) where we assume $i = 0$. This way we can express R3 as follows:

$$rate < x \Rightarrow \mathcal{W}_{60}(r \succ min_{rate} \curlywedge r \prec max_{rate})$$

where the membership function $\mu_{\mathcal{W}_{60}}$ is positive in the interval $[i, w)$ ($w > i + 60$).



(a) Memb. function $\mu_{\mathcal{W}}$. (b) Memb. function $\mu_{\mathcal{G}}$.

Figure 1.

According to the literature [2], the fuzzy temporal operator $\mathcal{G}\,\varphi$ is computed by finding the less satisfactory value of φ. This implies that if the truth value of φ becomes completely false (equals 0) for a certain time instant, due to a transient violation, $\mathcal{G}$ becomes 0, exactly like in the crisp case. Hence, this interpretation for operator $\mathcal{G}$ does not allow us to tolerate transient violations. For this reason, we use a different semantics: when the truth value of φ is under a certain threshold (th_{min}), it is not taken into account in the evaluation of $\mathcal{G}$. Instead, it is replaced 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 φ 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 1(b).

The semantics of operator $\mathcal{G}\varphi$ is formally described by the recursive function shown below.

Function $fEval$ uses an auxiliary function ($val1$) that takes sub-formula φ, on which $\mathcal{G}$ is applied, instant i at which the formula is computed, and a violation counter

$$fEval(\mathcal{G}\varphi, i) = val1(\varphi, i, 0)$$

$$\begin{array}{l} float\ val1(\varphi, i, error)\{ \\ \quad if(fEval(\varphi, i) < th_{min})\{ \\ \qquad error++; \\ \qquad if(\mu_{\mathcal{G}}(error) == 0)\ return\ 0; \\ \qquad else\ return\ \frac{\mu_{\mathcal{G}}(error)}{\mu_{\mathcal{G}}(error-1)} * val1(\varphi, i+1, error);\} \\ \quad else\ return\ fEval(\varphi, i) * val1(\varphi, i+1, error)\} \end{array}$$

($error$) that is 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 0, it means that $\mathcal{G}\varphi$ has a truth value equal to 0 ($return\ 0$). Otherwise, the current truth value of $\mathcal{G}\varphi$ is multiplied by factor $\frac{\mu_{\mathcal{G}}(error)}{\mu_{\mathcal{G}}(error-1)}$. Note that if no violation occurs, the truth value of φ (given by $fEval(\varphi, i)$) directly affects the final evaluation of $\mathcal{G}\varphi$.

This new definition allows us tolerate some transient problems, such as a certain number of interruptions during the transmission of the match and, indeed, we can represent R2 as follows:

$$\mathcal{G}(rate > min_{rate})$$

III. Monitoring

Many approaches [5]–[7] have been already proposed to monitor service compositions. For untimed formulas (e.g., properties expressed in propositional logic), monitoring results can be produced instantaneously, as soon as data collection is performed, with a negligible delay on the running system. Conversely, the evaluation of temporal formulas at a specific time instant (i) cannot be performed instantaneously, since some necessary data will be only available at subsequent time instants with respect to i. For this reason, monitoring results cannot be produced "on-the-fly" and the monitoring procedure may add considerable overhead on the system. Bounded temporal operators (such as, $\mathsf{F}_{\bullet t}$, $\mathcal{F}_{<t}$, $\mathsf{G}_{\bullet t}$, $\mathcal{G}_{\bullet t}$, $\mathcal{L}_t$, $\mathcal{W}_t$)[3] require a limited temporal window to be evaluated, when they do not include any future operator as argument. Instead, for unbounded temporal operators only temporary results can be provided, since the monitoring procedure must be theoretically performed over an infinite temporal window.

The number of threads used to evaluate a temporal formula may grow when it includes a set of temporal sub-formulas. For example, if we consider requirement R1 ($(rate < min_{rate}) \Rightarrow \mathcal{W}_{<60}(rate > min_{rate})$), every time the antecedent is satisfied, a new thread is created to evaluate temporal sub-formula $\mathcal{W}_{60}(\ldots)$. If the antecedent is satisfied every time instant, we will have at most 60 active thread instances at the same time that evaluate sub-formula $\mathcal{W}_{60}(\ldots)$. This problem seems even more complex when we deal with unbounded operators that can never terminate

[3] $\bullet \in \{\leq, <\}$

and may depend on other infinite threads that evaluate their sub-formulas at the same time. If the number of threads is too high, the overhead introduced in the system, in terms of time and memory consumption, may make the monitoring procedure unfeasible. Besides, fuzzy temporal operators provided by FLAGS further increase the complexity of the whole monitoring procedure. Every time a fuzzy temporal formula is evaluated, it is also necessary to aggregate the partial results of its sub-formulas.

The implementation of a monitor to evaluate each temporal operator is straightforward, since it can be easily derived from the operational semantics partially described in the previous section. On the other end, finding a technique to reduce the number of threads used during the evaluation of temporal operators is not simple and a solution that is semantically sound and practically feasible is required to solve this problem.

A. Monitoring crisp operators

Our optimization technique is based on the ideas that have been already proposed in [8] to optimize the monitoring of crisp temporal operators. This section briefly recaps these ideas, while the next section explains which optimizations can be performed to monitor fuzzy temporal operators.

Our optimization technique is based on the following assumptions:

- The satisfaction of temporal formulas is measured by a thread that is active during the temporal window in which the argument of the formula must be evaluated.
- The threads that evaluate a temporal formula update the (temporary) monitoring result only when at least one of the necessary variables changes its value.
- Each sub-formula is associated with a flag that indicates whether an active thread which evaluates that sub-formula already exists. For a subset of the temporal operators this flag is used to prevent the generation of multiple threads for the same sub-formula.
- If a temporary monitoring result is already available and the variables included in the formula to be evaluated do not change their value, the monitoring result continues to keep the same value.

All formulas that have G and F (both bounded and unbounded) as the most external operators do not require more than one active thread to be evaluated. Indeed, once operator G needs to be verified, its thread remains active until the argument of the formula (and, consequently, the whole formula) becomes false. In the meanwhile, if another occurrence of the same formula needs to be checked at a subsequent instant of time, it can rely on the same thread previously created. Symmetrically, the thread that evaluates operator F remains active until its argument becomes true. If in the meanwhile another instance of the same formula needs to be evaluated, the same thread can be used. The same rule is applied to monitor unbounded operators $\mathsf{G}_{>t}$ and $\mathsf{F}_{>t}$, with the only difference that their evaluation is delayed of t time instants.

Bounded versions of temporal operators may also require a single thread. As a matter of facts, in case an instance of $\mathsf{G}_{<t}$ (and $\mathsf{G}_{\leq t}$) is already active and another one is required to be evaluated at the current time instant, we do not need to create a new thread. Instead, the existing thread just enlarges the temporal window during which the formula is evaluated.

Optimizing operator $\mathsf{F}_{\leq t}$ (and $\mathsf{F}_{<t}$) is more tricky. We can use a single monitor only when we want to stop the monitoring procedure after the first error occurs. For example, if we are monitoring a critical property, we may be just interested to detect the first violation of the corresponding formula to apply an adaptation. In this case, we can use only one thread to monitor several instances of this operator that start at different time instants. For example, if we take into account formula $\mathsf{F}_{<10}A$ and evaluate it at instants i and $i+3$, we can have three possibilities. If A is satisfied between $(i, i+3]$ we have just 1 thread since the thread that evaluates the first instance of the formula terminates just before the second one starts. In case A is satisfied in $(i+3, 10]$ the thread that evaluates the first formula is also used to evaluate the second one. We do not take into account the case when A is satisfied in $[i+10, i+13]$ since, in this case, the first instance of the formula has been already violated. All these assumptions cannot be applied when we prefer to count the number of errors since, in this case, a separate thread must be used to evaluate different instances of the same formula.

B. Monitoring fuzzy operators

The semantics of fuzzy temporal operators is given by accumulating or filtering all the evaluations of their argument during a limited/unlimited temporal window. Since each instance of the same fuzzy temporal operator depends on a different temporal window, it requires a different thread to be evaluated. However, as we will demonstrate in Section IV, monitoring fuzzy temporal operators is still feasible in practice, even if no optimization can be applied.

To evaluate temporal operators $\mathcal{G}$ (and analogously $\mathcal{G}_{-x}$) it is necessary to accumulate the satisfaction value of the argument at each time instant, and scale it depending on the errors that already occurred. If an active thread that checks an instance of this formula already exists and another one has to be checked, a new thread must be created. The new thread cannot reuse the partial monitoring result already computed by the previous thread, since it accumulates all satisfaction of its argument starting from value 1. The previous thread is reused only when no errors happened and the temporary monitoring result is still equal to 1. The bounded versions of the previous operators ($\mathcal{G}_{<t}$ and $\mathcal{G}_{\leq t}$) suffer of a similar problem.

The evaluation of operator $\mathcal{F}_{<t}$ (and $\mathcal{F}_{\leq t}$) is equal to the maximum value of its argument during a time window of length t. If more than one instance must be checked, a

single thread is enough only if during the intersection of the different time windows the argument is evaluated to 1 or to the maximum value for all temporal windows considered separately. However this optimization is not feasible in practice, since the maximum value of the argument in the different temporal windows cannot be known in advance.

Operator $\mathcal{W}_t$ also depends on the time instant in which the formula is evaluated. The thread that checks this operator accumulates all evaluations of the corresponding argument during a temporal window and uses a membership function to scale it depending on the time instant in which the check is performed. Since different instances of the same formula are started at different time instants, they use different membership functions to weight the evaluation of their argument and their results are totally independent. For this reason, yet again, separate threads must be used.

IV. Implementation and Evaluation

This section describes the implementation of the monitoring framework and illustrates its performance when both crisp and fuzzy temporal formulas are evaluated.

A. Implementation of the Monitoring Framework

The architecture of the monitoring framework is shown in Figure 2. The *Monitor* is devoted to evaluate the formulas expressed in the FLAGS language. The *Configuration Manager* directly interacts with the designer of the system (who defines the requirements), and sends the corresponding FLAGS formulas to the *Monitor* (1). The *Data Collector* collects the data coming from the *Execution Environment* (2.a) and notifies the *Monitor* in case changes have taken place (2.b). The Monitor retrieves the modified data (2.b) and re-evaluates the formulas (2.c), if necessary. When the framework is started, the *Monitor* receives a set of data that represent the initial conditions of the system.

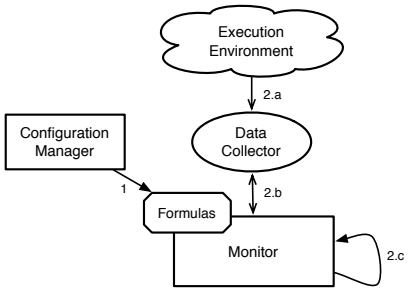
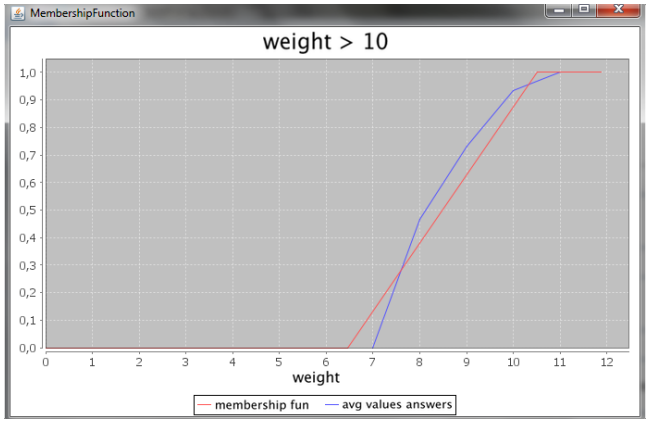


Figure 2. The architecture of the monitoring framework.

The *Monitor* creates a different thread depending on the operator that must be evaluated. In case a temporal formula includes other temporal subformulas, the thread that evaluates the most external operator also decides when and how create other threads according to the optimizations presented in Section III. We used Xtext [9] to define the syntax of the FLAGS language. Xtext allows us to automatically derive a parser that recognizes a FLAGS formula and generates its corresponding AST (Abstract Syntax Tree). We implemented a visitor that traverses the AST and generates the threads that will check the formula at runtime.

To express fuzzy temporal formulas it is also necessary to provide information regarding the shape of the membership functions. We assumed that membership functions always have a triangular/trapezoidal shape and interpolate the preferences of the users expressed in a questionnaire. We provide a graphical tool that allows users to fill the questionnaires (see Figure 3(a)) and analyzes the users' answers to generate a membership function that represents them (see Figure 3(b)).

(a) User questionnaiers



(b) Generated membership function

Figure 3.

B. Performance Evaluation

We evaluated the performance of the monitoring framework by computing the time and number of threads created to check both crisp and fuzzy formulas. We simulated the *Data Collector* through a suitable thread that updates the values of the variables necessary to produce the monitoring results and notifies the *Monitor* that, in its turn, re-evaluates the formulas. All experiments have been performed on a PC with processor x86 dual core, 4GB RAM and operative

system Windows 7. For all experiments each time instant is equivalent to 1 millisecond.

The first experiments are aimed to assess the concrete benefits of the optimization technique (expecially in the crisp case) on the whole performance. We expressed the same formulas with crisp and fuzzy operators to compare the overhead generated for their evaluation. We can notice that, despite the number of threads created for the evaluation of these formulas is higher when no optimization is applied, the response time still maintains acceptable values that are comparable with the theoretic ones. We used the same history (i.e., sequence of data) for each couple of experiments. For operator "Eventually" (fuzzy and crisp) the evaluation terminates after a time threshold specified in advance or as soon as its argument becomes true. While for operator "Always" we selected an history able to make the formula false after a certain time threshold. As expected, the evaluation of crisp formulas generates less threads than their fuzzy counterpart. Furthermore, fuzzy formulas have higher delays compared to the theoretical one (e.g., the highest delay is 379 milliseconds).

We performed other experiments to measure the number of threads created while crisp and fuzzy temporal operators are evaluated over long time intervals. To this aim, we compared the perfomance obtained to monitor a crisp ($G(G(s < 100))$) and a fuzzy formula ($\mathcal{G}(\mathcal{G}(s < 100))$) that adopt unbounded operators. In Figure 4 we can observe that the increasing side of the graph represents the phase in which threads are generated to evaluate the formula, while the decreasing side represents the phase when threads are eliminated after some variables change their value and some threads are not necessary anymore. Every time data change all threads that are not necessary are deleted. As expected, the number of threads created to evaluate the crisp formula is less than the number of threads needed to evaluate its fuzzy counterpart. In the crisp case the maximum number of threads is 300 (at instant 351), while in the fuzzy case it is greater than 2500 (at instant 497).

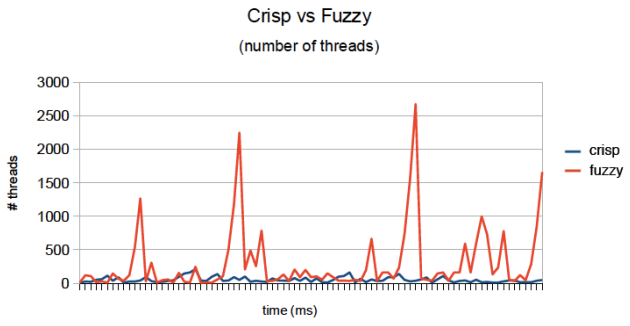


Figure 4. Number of threads created over time

Note that these results are highly influenced by the formula adopted in the experiments. However, in real cases requirements are expressed as simple formulas that do not have many inner sub-formulas. For this reason, the situations in which the number of active thread is too high are negligible. Generally we can claim that monitoring fuzzy requirements at runtime is feasible since it introduces an overhead, in terms of monitoring time, that is slightly greater than that measured to monitor crisp formulas.

V. Related Work

The idea of monitoring requirements was originally proposed by Fickas et al. [10]. The authors claim that requirements monitoring is necessary to know when and how the system has to evolve in case requirements assumptions become not valid. Feather et al. [11] improve this idea expressing monitoring requirements as temporal combinations of events that can be translated into monitoring code.

Robinson [7] provides a framework, ReqMON, to verify requirements on web services. It distinguishes between the design-time model, where business goals and their possible obstacles are defined, and the runtime model, where logical monitors are derived from the obstacles and are applied onto the running system. Monitors are represented in terms of LTL formulas. Conversely, our approach directly monitors requirements that are expressed as fuzzy formulas. It also infers logical monitors, that evaluate requirements at runtime, directly from the definition of requirements.

Wang et al. [12] propose a solution for monitoring system requirements, but are more interested in diagnosing the cause of requirements violations. Requirements are represented through a TROPOS model augmented with pre- and post-conditions that can be associated with goals or tasks to indicate the constraints that must be fulfilled before and after a goal is satisfied or a task is successfully executed. The authors also tune the monitoring granularity by adopting suitable switches that indicate whether the corresponding goal/task is to be monitored. Differently from our approach, requirements monitoring is not performed at runtime and log data are used to infer the denial of requirements and detect problematic components. Diagnosis is reduced to a satisfiability problem (SAT) that identifies what can be the causes (i.e., which leaf goal has failed) of a higher level goal to be not satisfied.

Souza et al. [13] introduce the concept of awareness requirements that may refer to the success/failures of other requirements or concerned with the truth/falsity of a domain assumption. The main advantage of this approach is to introduce linguistic constructs for expressing requirements and temporal operators. For example, requirements can represent: success/failure of other requirements, aggregate values (e.g., success/failure rate or min/max success/failure of other requirements), or a trend of another requirement over a temporal period. This really eases the designer task that do not have to specify them manually. However the

Table II
PERFORMANCE COMPARISON BETWEEN CRISP AND FUZZY OEPRATORS.

Formula	Theoretical monitoring time (ms)	Monitoring time (ms)	# Threads
$\mathsf{G}(Gltl(b > 85))$	120000	120175	3
$\mathcal{G}(\mathcal{G}(b > 85))$	120000	120213	121
$\mathsf{F}_{20}(\mathsf{F}_{20}(\mathsf{F}_{20}(b < 85)))$	76000	76116	7
$\mathcal{F}_{20}(\mathcal{F}_{20}(\mathcal{F}_{20}(b < 85)))$	76000	76379	8421

authors do not address uncertainty and requirements are still represented in LTL (OCL_{TM}).

VI. CONCLUSIONS

This paper describes the FLAGS languages to express requirements for service compositions. It fuzzyfies the temporal domain and allows us to express uncertain temporal properties. It provides a motivating example, a monitoring technique to assess FLAGS formulas at runtime, and justifies the viability of the approach through some experiments.

However our approach has some drawbacks. The FLAGS language is not intuitive: common users cannot specify their requirements directly and the intervention of the system designer is necessary. We just overcome this limitation for the definition of membership functions, that are generated by interpolating the answers given by the users to a questionnaire. This solution is not completely satisfactory and a mechanism that allows users to easily define FLAGS requirements, while hiding the complexity of the language, is still missing.

REFERENCES

[1] L. Baresi, L. Pasquale, and P. Spoletini, "Fuzzy Goals for Requirements-Driven Adaptation," in *Proc. of the 18th Int. Req. Eng. Conf.*, 2010, pp. 125–134.

[2] 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.

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

[4] L. Pasquale, "A Goal Oriented Methodology for Self-Supervised Service Compositions," Ph.D. dissertation, Politecnico di Milano, 2011.

[5] L. Baresi and S. Guinea, "Self-Supervising BPEL Processes," *IEEE Trans. on Softw. Eng.*, vol. 37, no. 2, pp. 247–263, 2011.

[6] G. Spanoudakis and K. Mahbub, "Non-intrusive monitoring of service-based systems," *Int. J. Cooperative Inf. Syst.*, vol. 15, no. 3, pp. 325–358, 2006.

[7] W. Robinson, "A Requirements Monitoring Framework for Enterprise Systems," *Req. Eng.*, vol. 11, pp. 17–41, 2006.

[8] L. Baresi, D. Bianculli, S. Guinea, and P. Spoletini, "Keep It Small, Keep It Real: Efficient Run-Time Verification of Web Service Compositions," in *Proc. of the Int. Joint Conf. FMOODS/FORTE*, 2009, pp. 26–40.

[9] "Xtext Language Development Framework," http://www.eclipse.org/Xtext/.

[10] S. Fickas and M. S. Feather, "Requirements Monitoring in Dynamic Environments," in *Proc. of the 2nd Int. Symposium on Req. Eng.*, 1995, pp. 140–147.

[11] M. S. Feather, S. Fickas, A. Van Lamsweerde, and C. Ponsard, "Reconciling System Requirements and Runtime Behavior," in *Proc. of the 9th Int. Workshop on Softw. Spec. and Design*, 1998, pp. 50–59.

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

[13] V. E. Silva Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos, "Awareness Requirements for Adaptive Systems," in *Proc. of the 6th Int. Symposium on Softw. Eng. for Adaptive and Self-managing Systems*, 2011, pp. 60–69.