Runtime Monitoring and Resolution of Probabilistic Obstacles to System Goals

Antoine Cailliau and Axel van Lamsweerde
ICTEAM – Institute for Information & Communication Technologies, Electronics and Applied Mathematics
Université catholique de Louvain
Louvain-la-Neuve, Belgium
{antoine.cailliau, axel.vanlamsweerde}@uclouvain.be

***Abstract*—Software systems are deployed in environments that keep changing over time. They should therefore adapt to changing conditions in order to meet their requirements. The satisfaction rate of these requirements depends on the rate at which adverse conditions prevent their satisfaction. Obstacle analysis is a goal-oriented form of risk analysis for requirements engineering (RE) whereby obstacles to system goals are identified, assessed, and resolved through countermeasures yielding new requirements. The selection of appropriate countermeasures relies on the assessed likelihood and criticality of obstacles together with environmental assumptions. These various factors are estimated at RE time; they may however evolve during software development and at system runtime.**

To meet the system's goals under changing conditions, the paper proposes to defer obstacle resolution to system runtime. Following *Monitor–Analyze–Plan–Execute* cycles, techniques are presented for monitoring goal/obstacle satisfaction rates; deciding when adaptation should be triggered; and adapting the system on the fly to countermeasures that are more appropriate under the monitored conditions. The approach relies on a model where goals and obstacles are refined and specified in a probabilistic linear temporal logic. The proposed techniques allow for (a) monitoring the satisfaction rate of probabilistic leaf obstacles; (b) determining the severity of their consequences by up-propagating satisfaction rates through refinement trees from leaf obstacles to high-level probabilistic goals; and (c) dynamically shifting to alternative countermeasures that better meet the required satisfaction rate of the system's high-level goals under imposed cost constraints. Our approach is evaluated on fragments of an ambulance dispatching system.

Keywords - adaptive systems; runtime requirements monitoring; probabilistic goals; obstacle analysis; goal-oriented requirements engineering.

I. INTRODUCTION

Software systems are increasingly deployed in unpredictably varying environments. Autonomous vehicle control [12], disaster management [28], or adaptive security [41] are examples of problem domains where the system must adapt to changing environments to guarantee its goals.

For runtime system adaptation, *Monitor-Analyse-Plan-Execute* cycles are often followed [13, 31, 34]. The *Monitor* step collects, filters and aggregates data from the running system such as performance metrics and configuration characteristics. The *Analyze* step determines whether a change is required or not based on data analysis and reasoning about the running system. The *Plan* step structures the actions to apply in order to guarantee that the system will subsequently meet its objectives. During the *Execute* step, the system is updated with the planned actions.

Probabilistic requirements often emerge in the requirement engineering (RE) phase of system development. Typically, they require some target property to be satisfied in at least *X*% of cases [4, 7, 20, 23, 35]. At system runtime, they may not be satisfied due to adverse conditions. This paper focuses on runtime adaptation mechanisms to guarantee that the minimal thresholds required by probabilistic requirements are still met at system runtime in spite of environment changes.

In goal-oriented RE, adverse conditions to requirements satisfaction are called obstacles. An *obstacle* is a precondition to the non-satisfaction of a corresponding goal [20, 32, 33, 37]. While building an AND/OR goal refinement graph (called *goal model*), the analyst performs obstacle analysis cycles, each consisting of three steps: (i) obstacles are systematically *identified* from specifications of goals and domain properties [1, 32]; (ii) the likelihood and criticality of the identified obstacles are *assessed* [7, 9, 20]; (iii) the likely and critical obstacles are resolved through countermeasures integrated as new goals in the goal model [2, 8, 32]. The problem is that the selection of "best" countermeasures is based on environment assumptions and obstacle satisfaction rates determined at RE time. These influencing factors may turn to be different at system runtime. Some assumptions might no longer hold; new characteristics might emerge; experts' estimates at RE time for obstacle assessment might prove inaccurate at system runtime; other estimates might not be available at RE time; and so forth. For better fit to the system's goals under changing or originally unknown conditions, it might thus be better to defer decisions on selecting most appropriate countermeasures to system runtime [19].

The paper presents obstacle-driven runtime adaptation techniques for increased satisfaction of probabilistic system goals. These techniques are intended to address the following more specific objectives.

- *Precise semantics in terms of observed states and behaviors.* In contrast with [28, 39, 42], the monitored items have a clear and precise meaning. A formal semantics enables their interpretation as real-world phenomena which reduces subjective assessments. A formal semantics also allows model checking techniques to be used for monitoring probabilistic obstacles.

- *Traceability of monitored indicators and deployed countermeasures.* Unlike [24], the monitored indicators and decision criteria for system adaptation are traceable to system objectives; *why* such or such monitored information is required is thereby documented.
- *Model-based adaptation.* The adaptation process is driven by a goal/obstacle model; only those adaptations which are required to meet the probabilistic assertions from this model are made. Model-based adaptation also reduces the need for application-specific manipulations.
- *No explicit behavior modelling.* Unlike [25, 26], the model used is declarative; system behaviors need not be explicitly modelled. Building a consistent and complete behavior model for large distributed systems with many complex states and parallelism is often quite challenging.

The paper makes the following contributions.

- A formal characterization of satisfaction rates of probabilistic goals and obstacles is provided in terms of observed states and behaviors.
- Probabilistic obstacles are monitored at system runtime. Our approach extends the monitoring technique introduced in [6] for non-probabilistic linear temporal logic (LTL) to monitor *probabilistic* LTL assertions at system runtime. From such monitoring of leaf obstacles, the satisfaction rate of high-level goals is obtained by up-propagation through obstacle/goal refinement trees.
- Alternative countermeasures are selected on the fly, among those identified at RE time, when the satisfaction rate obtained for high-level goals is below their required threshold. A cost-benefit tradeoff analysis guides the selection to maximize satisfaction rates under cost constraints.
- The selected countermeasures are integrated into the goal model and deployed in the running software system.

The paper is organized as follow. Section II provides some necessary background on goal/obstacle modeling with probabilistic specifications and on monitoring non-probabilistic LTL assertions. Section III outlines the general approach proposed for runtime system adaptation. Section IV presents our technique for runtime monitoring of probabilistic assertions. Section V explains how satisfaction rates for high-level goals are obtained from monitored leaf obstacles in order to determine when an adaptation is required. Section VI describes how most appropriate countermeasures are selected to yield a required adaptation. Section VII discusses how the selected countermeasures are deployed into the running system. Section VIII reports on a preliminary evaluation of our techniques on a real ambulance dispatching system. Section IX discusses related work.

II. BACKGROUND

Goal-oriented system modeling [33]. A *goal* is a prescriptive statement of intent to be satisfied by cooperation of the agents forming the system. The word "system" refers to both the software and its environment, including people, legacy software, devices like sensors and actuators, etc. A *domain property* is a descriptive statement about the system, e.g., a physical law.

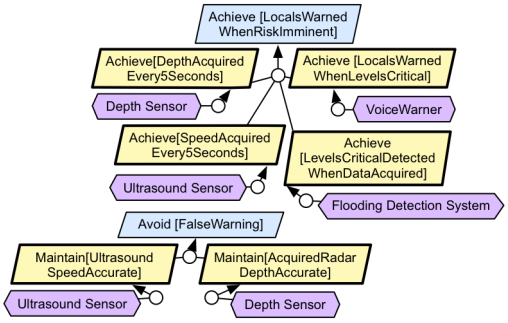

Figure 1. Goal model fragment for a flood detection system

The paper focuses on behavioral goals. Unlike soft goals, these goals can be satisfied in a clear-cut sense [33]. A behavioral goal defines a maximal set of behaviors declaratively and implicitly. A behavior violates a goal if it is not among the behaviors captured by the goal specification.

A metric linear temporal logic (MTL) is used for formalizing behavioral goals to enable their analysis [33]. The temporal operators include $\bigcirc$ (in the next state), $\Diamond$ (eventually), $\Diamond_{\leq d}$ (eventually before deadline d), $\Box$ (always in the future), $\Box_{\leq d}$ (always up to deadline d), W (always in the future unless). The standard logical connectives include $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (implies). A behavioral goal is of type *Achieve* or *Maintain*. The specification pattern for *Achieve* goals is $\Box$ **(C** $\rightarrow \Diamond$**T)**, where **C** and **T** refer to a current and a target condition, respectively. The specification pattern for *Maintain* goals is $\Box$ **(C** $\rightarrow$ **G)**, where **G** refers to a "good" condition.

A *goal model* is an AND/OR-graph showing how goals contribute positively or negatively to each other. An AND-refinement captures a combination of subgoals entailing the parent goal; an OR-refinement captures an alternative way of satisfying the parent goal. A goal may be refined into subgoals by asking "how" questions whereas it may be abstracted into parent goals by asking "why" questions. Leaf goals are assigned to specific system agents. A goal assigned to a single software agent is a *requirement* whereas a goal assigned to a single environment agent is an *assumption*.

Fig. 1 shows a goal model fragment for a flood detection system [30]. Goals and agents are represented by parallelograms and hexagons, respectively. The top goal in Fig. 1 is AND-refined into four leaf goals assigned to corresponding agents.

Obstacle Analysis [33]. An *obstacle O* to a goal *G* in the considered domain *Dom* is a satisfiable precondition for the non-satisfaction of this goal:

$\{O, Dom\} \models \neg G$ (obstruction)

$\{O, Dom\} \not\models false$ (domain consistency)

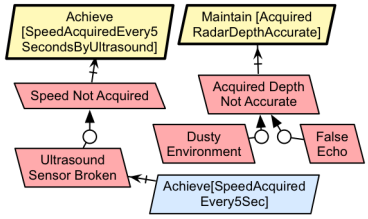

Figure 2. Obstacle model fragment with a countermeasure

Obstacles are also formalized in MTL. The specification pattern for an obstacle to an *Achieve* goal is $\Diamond$ **(C** $\wedge$ $\Box\neg$**T);** for an obstacle to a *Maintain* goal the pattern is $\Diamond$ **(C** $\wedge$ $\neg$**G)**.

Obstacles are also structured as AND/OR graphs rooted on negations of corresponding leaf goals. An obstacle AND-refinement captures a combination of sub-obstacles entailing the parent obstacle. An obstacle OR-refinement captures an alternative combination. The consequences of an obstacle are the falsification of the ancestors of the obstructed leaf goal.

Obstacles are resolved through *countermeasures* aimed at reducing their likelihood or mitigating their consequences. Fig. 2 shows two obstacle trees anchored on corresponding goals (obstacles are depicted by left parallelograms). The countermeasure goal Achieve[SpeedAcquiredEvery5Sec] there resolves the leaf obstacle UltraSoundSensorBroken obstructing the goal Achieve[SpeedAcquiredEvery5SecondsByUltrasound].

The integration of countermeasures to obstacles in a goal model increases the completeness of this model. The integration either adds a new goal in the model or replaces the obstructed goal or an ancestor of it by another goal [8]. A countermeasure goal *CG* is said to be *valid* if some ancestor goal *AG'* of the obstructed goal is entailed by it:

$\{CG, SG_1', \ldots, SG_n', Dom'\} \vDash AG'$ (ancestor entailment)

where *AG'* and SG_i' denote possibly weakened versions of the original ancestor *AG* with subgoals SG_i, respectively.

The *anchor* for a countermeasure goal is the lowest ancestor goal meeting the *ancestor-entailment* condition. In the augmented goal model, the countermeasure goal is anchored to this goal [8].

Various *strategies* are available for exploring alternative countermeasures [32, 33].

Probabilistic Goals and Obstacles [7]. Behavioral goals might in practice not be always satisfied in any possible situation. A *probabilistic goal* prescribes a minimal threshold for its satisfaction rate —e.g., "locals shall be warned when flooding is imminent *in at least 95% of cases*".

The *required degree of satisfaction (RDS)* of a goal is the minimal admissible satisfaction rate to be ensured by the system-to-be. It is often imposed from regulations, standards, common practice, and so forth.

The *estimated satisfaction rate* (ESR) of a leaf obstacle is the obstacle's satisfaction rate estimated by domain experts at RE time. The ESR of a goal is its satisfaction rate in view of its possible obstructions by obstacles. As seen in Section V, a goal's ESR can be obtained from the ESR of its leaf obstacles by up-propagation through the goal/obstacle model [9]. A goal's *ESR* shall ideally be greater than its *RDS*.

Monitoring Non-Probabilistic LTL Assertions [6]. Monitoring the runtime satisfaction of a LTL formula relies on the finite trace observed so far. The monitoring technique in Section IV extends the automata-based approach in [6] to *probabilistic* LTL assertions. The latter approach is chosen as it reports LTL formula satisfaction or violation as early as possible.

LTL_3, the LTL considered in [6], uses *true*, *false*, and *inconclusive* as truth values. A finite trace is labelled as *true* if any continuation of it satisfies the formula; *false* if any continuation falsifies the formula; and *inconclusive* otherwise. A LTL_3 formula monitor is a finite state machine (FSM) that reads finite traces and outputs the corresponding truth value. As suggested by Fig. 3, this FSM is built from the formula and its negation [6].

Figure 3. Steps for computing monitor FSM from input formula φ

(1) The associated non-deterministic Büchi automaton (NBA) is generated using a standard algorithm [15].
(2) A non-deterministic finite automaton (NFA) is then generated by performing an emptiness check for each NBA state. A state *s* is labelled with *true* if a continuation satisfying the formula exists (that is, if the language corresponding to the NFA with initial state *s* is not empty); it is *false* otherwise.
(3) The NFA is determinized to produce a deterministic finite automaton (DFA) using powersets.

The monitor FSM results from the product of both DFAs. Let (s_1,s_2) denote a state in this product, where s_1 is the DFA state corresponding to the formula φ and s_2 the DFA state corresponding to its negation $\neg\varphi$. This monitor FSM state is labeled as:

- *true* if s_1 is labeled as *true* and s_2 as *false* (no continuation exists such that the formula is falsified);
- *false* if s_1 is labeled as *false* and s_2 as *true* (no continuation exists such that the formula is satisfied);
- *inconclusive* otherwise.

At runtime, the current state of the monitor FSM is updated according to the truth value of the observed predicates and state assertions on FSM transitions. More details about the approach can be found in [6].

III. Overview of the Approach

Based on the background outlined in the previous section, the objective in this paper is to let the system dynamically switch to more appropriate countermeasures to leaf obstacles in view of evolving environment conditions and obstacle satisfaction rates. The satisfaction rate of leaf obstacles was estimated at RE time and is now being observed at system runtime. The alternative countermeasure goals to those leaf obstacles were identified and specified at RE time.

A countermeasure should dynamically replace the current one when, unlike the latter, it makes the satisfaction rate of high-level goals exceed their required degree of satisfaction. The satisfaction rate of high-level goals is obtained from the monitored satisfaction rate of the leaf obstacles by up-propagation through the goal/obstacle model. The monitored satisfaction rate of a leaf obstacle is obtained by counting the observed behaviors.

Our approach comprises 6 steps detailed in the next sections.

(1) LTL_3 monitors for the leaf obstacles are built at RE time. The list of predicates to observe at runtime is thereby provided. (See Section IV.A.)
(2) At runtime, the states of the monitored system are observed at a regular pace. (The pace may be chosen to fit a specific domain.) At every observation, a new

virtual monitor for each leaf obstacle is started while existing monitors are updated. (See Section IV.B.)

(3) The monitored satisfaction rate of leaf obstacles is up-propagated through obstacle/goal refinement trees up to high-level goals. (See Section V.)

(4) Comparing the monitored satisfaction rates obtained for those goals with their RDS determines whether the current countermeasures to the monitored obstacles are still appropriate. If a monitored goal satisfaction rate falls below the goal's RDS, alternative more appropriate countermeasures are selected among those available. (See Section VI.A and Section VI.B.)

(5) The goal/obstacle model is updated accordingly by integrating the new current countermeasures and updating the propagation in Step 4. (See Section VI.)

(6) The software is automatically adapted according to the selected countermeasures. (See Section VII.)

IV. Monitoring Probabilistic Obstacles

At RE time, domain experts estimate the satisfaction rates of the leaf obstacles based on their knowledge of the system or their experience with similar systems [9]. These estimates might prove inaccurate at system runtime —they might be too rough or environment properties or assumptions might have changed in the meantime. Moreover, some variables might be hard to estimate at RE time —prior data might not be available or might be too costly to acquire; too many parameters might be involved; etc. Monitoring the *actual* satisfaction rate of leaf obstacles at system runtime helps filling this gap; former estimates may be made more accurate, and missing data may be made available.

Section IV.A provides a precise definition of satisfaction rates in terms of states and behaviors. Section IV.B explains how monitors for probabilistic assertions are built and used on top of non-probabilistic LTL_3 monitors.

A. Formal Framework for Probabilistic Assertions

As introduced in Section II, probabilistic goals might be satisfied only partially. A precise characterization in terms of observed states and behaviors enables the monitoring of their satisfaction rates.

An *Achieve* goal $\Box$ **(C** $\rightarrow \Diamond$**T)** requires all possible system states to satisfy $\mathbf{C} \rightarrow \Diamond \mathbf{T}$. As the goal might be satisfied only partially, a state s has a probability that the behaviors starting from it satisfy $\mathbf{C} \rightarrow \Diamond \mathbf{T}$.

The *state probability* of a non-probabilistic formula φ in state s, denoted by $Pr(s, \varphi)$, is defined as the ratio between (a) the number of possible behaviors from s satisfying φ, and (b) the number of possible behaviors from s. The notation $P^s_{\geq x}(\varphi)$ denotes the statement "*the state probability of* φ *in* s *is greater than* x", that is, $Pr(s, \varphi) \geq x$.

The *satisfaction rate* of an *Achieve* goal $\Box$**(C** $\rightarrow \Diamond$**T)** is the lowest state probability of $\mathbf{C} \rightarrow \Diamond \mathbf{T}$ for any possible state s. Note that the "$\Box$" goal prefix requires a lower bound as we focus on the *lowest* chance of goal satisfaction.

A goal $\Box$ **(C** $\rightarrow \Diamond$**T)** with satisfaction rate x states that the system satisfies the formula in at least x% of cases. This may be written as $\Box P_{\geq x}(\mathbf{C} \rightarrow \Diamond \mathbf{T})$ where the assertion $\Box P_{\geq x}(\varphi)$ is satisfied by a behavior if all states s along this behavior satisfy $P^s_{\geq x}(\varphi)$. The preceding definitions are similar for *Maintain* goals.

For example, consider a system with three states and the goal Achieve[LocalsWarnedWhenLevelsCritical], specified by

$\Box$ **(LevelsCritical** $\rightarrow \Diamond_{<5min}$ **LocalsWarned)**.

Assume that **LevelsCritical** $\rightarrow \Diamond_{<5min}$ **LocalsWarned** has a state probability *.1* in s_1; *.2* in s_2; and *.3* in s_3. The satisfaction rate for this goal is its lowest state probability, that is, *.1*. The system satisfies $\Box P_{\geq .1}$(**LevelsCritical** $\rightarrow \Diamond_{<5min}$ **LocalsWarned**).

An obstacle $\Diamond$ **(C** $\wedge \Box \neg$**T)** states that there is one future state at least that satisfies $\mathbf{C} \wedge \Box \neg \mathbf{T}$. A state has a probability that behaviors starting from it satisfy $\mathbf{C} \wedge \Box \neg \mathbf{T}$.

The *satisfaction rate* of an obstacle $\Diamond$ **(C** $\wedge \Box \neg$**T)** is the highest state probability of $\mathbf{C} \wedge \Box \neg \mathbf{T}$ for any possible state s. Dually to goals, the "$\Diamond$" obstacle prefix states an upper bound as we focus on the *highest* chance of goal violation.

An obstacle $\Diamond$ **(C** $\wedge \Box \neg$**T)** with satisfaction rate x states that the system satisfies the formula in at most x % of cases. This may be written as $\Box P_{\leq x}(\mathbf{C} \wedge \Box \neg \mathbf{T})$. The preceding definitions are similar for obstacles to *Maintain* goals.

In our example, consider the obstacle GSMNetworkDown, specified by

$\Diamond$ **(LevelsCritical** $\wedge \Box_{<5min}$ **GSMNetDown)**.

Let us assume that **LevelsCritical** $\wedge \Box_{<5min}$ **GSMNetDown** has a state probability *.9* in s_1; *.8* in s_2; and *.1* in s_3. The satisfaction rate for this obstacle is its highest state probability, that is, *.9*. The system satisfies the assertion $\Box P_{\leq .9}$(**LevelsCritical** $\wedge \Box_{<5min}$ **GSMNetDown**).

The definitions presented here differ from those in [7] by relying on state probabilities. They are needed for extending the LTL_3 monitoring technique [6] to probabilistic obstacles.

B. Monitoring-Based Estimation of Satisfaction Rates

As the satisfaction rate of an obstacle $\Diamond \varphi$ is the upper bound among the state probabilities of φ, we may at runtime count the number of observed behaviors satisfying φ from states s; this estimates the corresponding state probability. The automata-based monitoring procedure for LTL_3 determines at runtime whether φ is satisfied from s.

The *monitored satisfaction rate* of an obstacle or a goal is its actual satisfaction rate as observed in the running system.

For an obstacle $\Diamond \varphi$, the monitored satisfaction rate is determined from the monitored state probabilities. The latter are obtained by monitoring the satisfaction of φ for all observed states.

Fig. 4 shows the process of monitoring the satisfaction rate of the obstacle $\Diamond(\Box_{>2s}$ dustyEnvironment) during 8 observations. The squares represent states of the observed system. Three states *A*, *B*, *C* are observed; *A* and *B* satisfy the predicate dustyEnvironment while *C* does not. The circles show the label of the current state of the LTL_3 monitors.

As the semantics of our language is synchronous [36], observations are made at a regular pace. If observations are performed every second, the MTL formula φ inside the $\Diamond$-operator can be transformed into an LTL conjunction:

φ: dustyEnvironment $\wedge$ o dustyEnvironment $\wedge$ o o dustyEnvironment

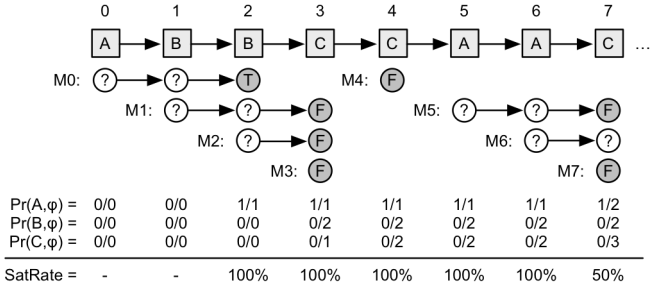

Figure 4. Monitoring a probabilistic obstacle

Fig. 5 shows the corresponding LTL_3 monitor. The top left monitor state labelled with *?* is the initial state. Each transition is labelled with a state formula. The label on states are: *T* for *true*, *F* for *false*, *?* for *inconclusive*.

For a monitored leaf obstacle, at each observation of the running system, an LTL_3 monitor is started to check whether the behavior from the current state satisfies φ. As seen below, virtual copies are used in practice to avoid creating new monitors at runtime.

Let us have a closer look at the example in Fig. 4.

- At observation *0*, we start a monitor M_0 to check whether the future system behavior satisfies φ.
- At observation *1*, the monitor M_0 is still inconclusive; a new monitor M_1 is started.
- At observation *2*, the current state of the monitor M_0 is updated to *true*. For state *A*, one observed behavior so far satisfies φ (see "1/1" at the bottom of Fig. 4). A new monitor M_2 is started.
- At observation *3*, the current state of the monitors started at observations 1 to 3 is labeled *false*. For state *B*, two behaviors were observed to not satisfy φ (see "0/2" at the bottom of Fig. 4). For state *C*, one behavior is seen to not satisfy φ ("0/1").
- At observation *7*, one observed behavior from state *A* satisfies φ among the two observed ones (the third one is still inconclusive). The two observed behaviors from *B* violate the formula. The three observed behaviors from *C* also violate the formula.

In this setting, the *monitored state probability* of state *s* is the ratio between (a) the number of monitors started in *s* whose current state is labelled as *true*, and (b) the number of monitors started in *s* whose current state is labeled as *true* or *false*.

In our example, the monitored state probability of state *A* is not available for the two first observations; it is equal to 1 for the five next observations, and to .5 for the last one. The satisfaction rate for our obstacle is the upper bound of these state probabilities. It changes from 'not available' to 100% at observation *2*, then decreases from 100% to 50% at observation *7* (see the bottom of Fig. 4).

In practice, creating a new LTL_3 monitor at each observation is clearly unrealistic as the complexity is $O(2^{2^n})$ where n is the size of the formula [6]. To avoid creating multiple instances of the same monitor, one LTL_3 monitor is built at RE time; the monitors being started at runtime are virtual copies of the former. A *virtual copy* only contains a pointer to the current state of the LTL_3 monitor. The complexity of starting a "new" virtual monitor is thus $O(1)$.

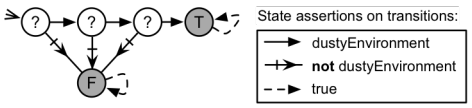

Figure 5. LTL_3 monitor for *DustyEnvironment*

The complexity of updating all monitors is $O(n)$ where n is the number of virtual monitors. This number depends on the number of observations. The worst-case situation corresponds to a system where all observed states are unique and all monitors remain inconclusive forever. Such system is unlikely. Our running example and the validation case study in Section VIII suggest that monitors have a short life which reduces the cost of updating monitors.

To implement the monitoring of leaf obstacles, a list of monitors is kept in memory for each observed state. To increase efficiency, the number of behaviors satisfying the formula φ and the number of behaviors violating it are kept in registers. Once a monitor reaches a monitor state labelled as *True* or *False*, it can be removed from the list and the corresponding register updated. Computing the state probability is then reduced to arithmetic operations on these registers.

To mitigate the risk of unnecessary system adaptations, "enough" observations should be made before deciding whether an adaptation is required. Otherwise, decisions would be based on non-statistically significant data, possibly leading to adaptations that deteriorate the system instead of improving it. To address this problem, standard statistical techniques may be used to compute the number of observations required to achieve a specified level of accuracy. Achieving such statistical significance imposes limits on the rate at which the system can adapt. Details are skipped here for lack of space; they can be found in [10].

Note that other monitoring techniques such as [17, 27, 29, 50] might be used to determine the satisfaction of φ. LTL_3, however, reports both violation and satisfaction of the formula as early as possible. Its three-value semantics distinguishes cases where a formula is satisfied, not satisfied, or none applies. Techniques such as [17] amalgamate the last two cases.

V. Obstacle-Based System Adaptation

A system adaptation is required at runtime when the current configuration of countermeasures does not guarantee the required degree of satisfaction (RDS) of the system's high-level goals. The actual satisfaction rate of these goals must therefore be determined from the monitored satisfaction rates of leaf obstacles. When falling below their RDS, alternative countermeasures maximizing the satisfaction rate of these goals should replace the current configuration.

The model up-propagation procedure in [9] is borrowed for determining the satisfaction rate of high-level goals. We summarize it here for a single high-level goal. Multiple ones are handled by use of a weighted sum combining goal satisfaction rates, where the weights capture goal priorities.

An *obstruction set* for a goal captures an AND-combination of leaf obstacles that prevents the goal from

being satisfied. A goal may have multiple alternative obstruction sets. An *obstruction superset* for goal *G*, denoted by *OS(G)*, is the set of all its obstruction sets.

To obtain the monitored satisfaction rate of a goal, we need to compute its obstruction superset by up-propagation of satisfaction rates through the goal/obstacle model, from leaf obstacles to root obstacles to leaf goals to root goal. This is done at RE time; it needs not be repeated at runtime.

1. *From leaf obstacle to root obstacle.* Let *LG*, *RO* and *LO* denote a leaf goal, obstructing root obstacle, and corresponding leaf obstacles, respectively. Let *OS(LG|RO)* denote the obstruction superset for *LG* considering all sub-obstacles in the tree rooted on *RO*. This obstruction superset is computed by structural induction:

   $OS(LG|LO) = \{LO\}$ (leaf obstacle)

   $OS(LG|O) = OS(LG|SO_1) \times OS(LG|SO_2)$ (for AND-Refinement)

   $OS(LG|O) = OS(LG|SO_1) \cup OS(LG|SO_2)$ (for OR-Refinement)

   where $\times$ represents the Cartesian Product over sets.
2. *From root obstacle to leaf goal.* For leaf goal *LG* obstructed by root obstacle *RO*, the obstruction superset is simply given by $OS(LG) = OS(LG|RO)$.
3. *From leaf goals to root goal.* The obstruction superset for a root goal is obtained by bottom-up propagation along AND-refinements in the goal model according to the rule

   $$OS(PG) = OS(SG_1) \cup OS(SG_2)$$

   for an AND-refinement with two subgoals SG_1 and SG_2.

The obstruction superset for a root goal captures an AND/OR combination of leaf obstacles. The corresponding Boolean formula is encoded as a binary decision diagram (BDD) to enable efficient subsequent manipulations. The internal BDD nodes correspond to leaf obstacles. A positive (resp. negative) edge indicates that the leaf obstacle is (resp. is not) in the combination —a combination being a path of positive/negative edges from root to terminal node. The terminal nodes indicate whether the combination obstructs the goal.

At system runtime, the probability for a goal's obstruction superset is computed from the monitored values for the leaf obstacles. The satisfaction rate *SR(G)* for goal *G* is given by $SR(G) = 1 - Pr(OS(G))$ where Pr(*OS(G)*) denotes the probability of *OS(G)*. To compute Pr(*OS(G)*), the positive BDD edges are decorated with the monitored satisfaction rates *SR(LO)* for leaf obstacles *LO* whereas negative edges are decorated with $1 - SR(LO)$. The probability Pr(*OS(G)*) is then computed bottom-up from the leaves of the BDD to its root. More details can be found in [9].

The satisfaction rate obtained for the high-level goal is compared with the goal's RDS. When falling below, an adaptation is required through alternative countermeasures maintaining the monitored satisfaction rate above the RDS.

Back to our running example, consider the goal Maintain [AcquiredRadarDepthAccurate]. Its obstruction superset is:

$OS(G) = \{ \{ DustyEnvironment \} , \{ FalseEcho \} \}$,

corresponding to the formula DustyEnvironment $\vee$ FalseEcho. A candidate BDD is the following:

DustyEnvironment
→ Positive edge: Terminal node 1
→ Negative edge: **FalseEcho**
→ Positive edge: Terminal node 1
→ Negative edge: Terminal node 0

Let us assume that the probability for FalseEcho is 2%. The positive edge is decorated with *.02* and the negative one with *.98*. The propagated probability for that internal node will be

$1 \times .02 + 0 \times .98 = .02$.

Assume that the probability for Dusty Environement is 5%; the propagated probability for that internal node will be

$1 \times .05 + 0.02 \times .95 = .069$.

The monitored satisfaction rate for the goal Maintain [Acquired RadarDepthAccurate] is found to be $1 - .069 = 93.1\%$. If the goal's RDS is 92%, an adaptation is required.

VI. Selecting Most Appropriate Countermeasures To Critical Obstacles

Section VI.A describes how the impact of alternative countermeasures is assessed. Section VI.B clarifies what are "most appropriate" countermeasures to be selected and deployed when an adaptation is required. Section VI.C explains how such selection is computed.

A. Assessing the Impact of Countermeasures

The satisfaction rate of an alternative countermeasure goal to be considered impacts on the satisfaction rate of the system's high-level goals. Such impact needs to be quantified in order to decide whether the countermeasure should be selected.

Assessing a countermeasure's maximal impact at RE time. A countermeasure goal may itself be refined down to leaf goals assignable to single agents. As the latter might be obstructed by new obstacles, a new cycle of obstacle analysis may be needed at RE time. At runtime, the new corresponding leaf obstacles might be monitored as well for more accurate assessment of their satisfaction rate.

To determine whether a new obstacle analysis cycle is required, the *maximal change in satisfaction rate* of the considered high-level goal should be considered. At best, the satisfaction rate of the countermeasure goal is *1* (no possible obstruction of it). By up-propagation, this satisfaction rate leads to a satisfaction rate sr_1 for the high-level goal. At worst, the satisfaction rate is *0*. This leads to another satisfaction rate sr_0 for the high-level goal. The countermeasure's maximal impact is obtained by taking the difference $sr_1 - sr_0$. Based on this, the analyst may decide at RE time whether a new cycle is needed or not.

For example, consider the countermeasure goal Achieve [LocalsWarnedBySMSWhenLevelsCritical]. If the satisfaction rate of this countermeasure is *0*, the satisfaction rate of the high-level goal Achieve [LocalsWarnedWhenRiskImminent] is 52.3%. If the satisfaction rate of this countermeasure is *1*, the satisfaction rate of the high-level goal is 61.6%. The maximal impact on the high-level goal is thus 9.2%. As the impact is important, it may be worth spending time in studying obstacles to this countermeasure goal at RE time to better estimate the impact of its deployment.

Depending on the selected obstacle resolution strategy and associated countermeasure integration schema, the obstructed goal is *removed* from the goal model or *kept* [8, 32]. In the former case, the obstructed goal is replaced by the

countermeasure goal; in the latter case, a new refinement is introduced involving both the obstructed goal and the countermeasure goal.

Assessing countermeasure impact at runtime when the obstructed goal is replaced. In this case, when the monitored satisfaction rate of the high-level goal is computed at runtime, the propagation procedure shall use the satisfaction rate of the countermeasure goal in place of the satisfaction rate of the replaced goal.

For example, the goal Achieve [SpeedAcquiredEvery 5SecByCamera] was generated using the goal substitution strategy. When integrated, it replaces the goal Achieve [Speed AcquiredEvery5SecByUltrasound]. The computation of the satisfaction rate of the parent goal Achieve [LocalsWarnedWhen RiskImminent] uses the satisfaction rate of this countermeasure goal rather than the satisfaction rate of the replaced goal. When integrated, the satisfaction rate for the top goal increases from 52.3% to 58.1%.

Assessing countermeasure impact at runtime when the obstructed goal is kept. In this case, the anchor goal is refined into two subgoals; one subgoal is the obstructed goal conjoined with the negation of the obstacle condition; the other subgoal is the countermeasure goal [8]. (To preserve the correctness of goal/obstacle refinements, the new conjunct is propagated down to leaf obstacles.) The satisfaction rate for the anchor goal is therefore a combination of the satisfaction rates of those two subgoals.

For example, integrating the countermeasure goal Achieve [LocalsWarnedWhenEmergencySituation] causes the root goal to be refined in two subgoals: Achieve [LocalsWarned**When No**EmergencySituation] and Achieve [LocalsWarnedWhenEmergencySituation]. The satisfaction rate of the root goal increases from 52.3% to 65.4%: The down-propagation of changes in the goal graph reduces the formal specification of the obstacle LowAcquisitionRateInEmergencySituation to *false*; The satisfaction rate of the latter is therefore *0*.

B. What are Most Appropriate Countermeasures?

Countermeasure selection at runtime should at reasonable cost increase the satisfaction rate of high-level goals above their RDS.

A *safe selection* of countermeasures is a set of countermeasures that, once integrated, guarantees that the monitored satisfaction rate of the high-level goals is greater than their respective RDS.

Achieve [LocalsWarned ByEmailWhenLevelsCritical]	0		0	1	**1**		1
Achieve [LocalsWarned BySMSWhenLevelsCritical]	0		1	0	**0**		0
Achieve [SpeedAcquired Every5SecondsByCamera]	0		0	0	**0**		1
Achieve [SpeedAcquired Every10SecondsByUltrasound]	0		1	1	**0**		0
Achieve [localsWarned WhenRiskImminent AndEmergencySituation]	0	…	0	0	**1**	…	1
Maintain [CameraSpeed Accurate]	0		0	0	**0**		1
Achieve [LocalsWarned WhenRiskImminent]	.52		.68	.72	**.81**		.77
Cost	0		2	2	**2**		4

TABLE I. COMBINING COUNTERMEASURES

A selection of countermeasures must be *consistent*; conflicting countermeasures shall not be selected together:

$\{CM_i, CM_j, Dom\} \not\models false \quad \text{for } i \neq j$ (selection consistency)

Countermeasures come with a cost. The latter are elicited with regard to the countermeasure's contributions to soft goals in the goal model —such as Minimize[PowerConsumption] or Maximize[Speed Computation] [33]. For ease of presentation, only single costs are considered here. Multiple costs are combined by a weighted sum where weights correspond to the priorities of the considered soft goals.

The *resolution cost* of a selection is the sum of the costs associated with the selected countermeasures.

A selection is *cost-optimal* if no other selection increases the probability of satisfaction of the high-level goal without increasing the resolution cost.

The selection of *most appropriate countermeasures* is defined as the one having the lowest resolution cost such that the satisfaction rate for the high-level goals is maximized while being safe, consistent, and cost-optimal.

C. Selecting Countermeasures at Runtime

The selection of most appropriate countermeasures amounts to solving two optimization problems, namely,

- finding the minimal cost for guaranteeing the RDS of the high-level goals;
- finding the selection that maximizes the satisfaction rate of the high-level goals given this cost.

This cannot be done at RE time as the satisfaction rates of obstacles might be unavailable or estimated inaccurately. If no solution is found, the system might leave the self-adaptation mode to prompt for manual adaptation.

Computing the minimal cost. We may iteratively generate all possible selections and keep the selection that minimizes cost while guaranteeing that the RDSs are met. The complexity of this naïve approach is $O(2^n)$ where n is the number of countermeasures.

Computing the cost-optimal selection. We may then generate all possible selections, compute their cost, and keep the selection with a minimal cost and the largest satisfaction rate for the high-level goals. The complexity of this naïve computation is $O(2^n)$ for n countermeasures.

In our example, the minimal cost for guaranteeing the RDS of 80% for the goal Achieve [LocalsWarnedWhen RiskImminent] is *2*. There are 8 possible combinations guaranteeing the RDS with costs ranging from *2* to *4*, partially shown in grey in Table I. The first 6 rows show the selection of countermeasures: *1* for selected goals, *0* otherwise. The row before the last one is the satisfaction rate for the root goal. The best selection costs *2* and maximizes the satisfaction rate, shown in bold in Table I.

The problem of finding the cost-optimal selection shares similarities with the NP-hard Knapsack optimization problem [16]. The latter is concerned with filling a bag with valued items without exceeding a maximal weight while maximizing value. The problem here differs in that (i) the value and weight of items do not simply sum; and (ii) adding a new countermeasure goal does not necessarily increase the satisfaction rate of a high-level goal. Improvements of our

naïve algorithms are however expected as a pseudo-polynomial algorithm exists for the Knapsack problem [16]. Other techniques such as [38] might also improve selections.

VII. Runtime Deployment of Most Appropriate Countermeasures

When most appropriate countermeasures are integrated in the goal model and selected, the running software system must be adapted to match the updated goal model. The software component responsible for adapting the running system is named *Adaptor* in the following discussion. This component keeps track of a current selection of countermeasures. When the most appropriate countermeasures are computed, it identifies the countermeasures that are (i) added —that is, not found in the current selection but in the selection of the most appropriate ones; and (ii) removed —that is, no longer in the selection of the most appropriate ones.

To adapt the software system, the Adaptor calls the activation and deactivation procedures associated with the countermeasures to be added or removed. The *activation procedure* is the code-related procedure responsible for the deployment of the corresponding countermeasure in the running system. The *deactivation procedure* is responsible for its removal. These procedures are used to: add, remove or replace a running component; update configuration parameters; activate hardware components; and so forth. The countermeasure goals are decorated with these procedures.

For example, let us consider that the monitored satisfaction rate of the leaf obstacle UltrasoundSensorBroken For5Sec increases, as shown in dashed line in Fig. 6. This increase causes a decrease in the monitored satisfaction rate of the high-level goal Achieve[LocalsWarnedWhenRiskImminent] (in solid line in Fig. 6) below its RDS. This decrease causes the countermeasure Achieve [SpeedAcquiredEvery5SecBy Camera] to be selected as most appropriate countermeasure. The activation procedure for Achieve [SpeedAcquiredEvery5Sec ByCamera] is called and replaces the software component acquiring the speed by the camera-related component. As Fig. 6 shows, the satisfaction rate of the high-level goal increases above its RDS after software adaptation.

VIII. Validation

Our approach was applied to a benchmark commonly used for evaluating obstacle analysis techniques [1, 7, 8]. The goal/obstacle model for the ambulance dispatching system is based on [32]. Section VIII.A outlines this model. Section VIII.B overviews the monitored software and the monitoring infrastructure. Section VIII.C presents the results of our simulations. These results are discussed in Section VIII.D.

A. Goals and Obstacles in Ambulance Dispatching

The goal model contains 44 goals, 18 refinements and 5 agents. The obstacle model includes 8 root obstacles and 158 leaf obstacles. The full model can be found in [11].

A top goal in this model is Achieve [IncidentResolved]. It is refined into two subgoals: Achieve [AmbMobilizedWhen IncReported] and Achieve [PatientAtHospitalWhenAmbMobilized]. The former is further refined into two subgoals: Achieve [AmbAllocatedWhenIncReported] and Achieve [AllocatedAmb Mobilized]. These goals are in turn refined until they are assignable to single agents.

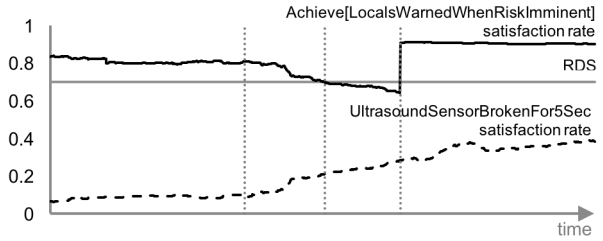

Figure 6. Adaptation after an increase in the satisfaction rate of obstacle *UltrasoundSensorBrokenFor5Sec*

The *Allocator* software agent is assigned to the goal Achieve [AmbAllocatedBasedOnAmbulanceInfo], a subgoal of Achieve [AmbAllocatedWhenIncReported]. The *Mobilizator* software agent is assigned to the goal Achieve [MobilizationOrderSentWhenAmbAllocated], a sub-goal of Achieve [AllocatedAmbMobilized].

Obstacles to leaf goals were identified and refined. Here is a small sample of obstacle trees:

Obstacles to *Achieve [AmbulanceOnSceneWhenMobilized]*:
MobilizedAmbulance**Not**OnSceneInTime
← AmbulanceStuckInTrafficJamTowardIncident
← DestinationUnreachable

Obstacle to Achieve [AtStationStatusEncoded]
← AtStationButton**Not**Pressed

Obstacles such as AtStationButton**Not**Pressed were further refined for different ambulances: Amb1AtStationButton **Not**Pressed, Amb2AtStationButton**Not**Pressed, and so forth.

Countermeasures to leaf goals were explored using available resolution strategies [32]. For example, the obstacle AmbStuckInTrafficJamTowardIncident is resolved by the countermeasure goal Achieve [AmbulanceReallocatedWhen StuckInTrafficJam] by use of the *strong mitigation* strategy.

B. Ambulance Dispatching: the Software and its Monitors

The *ambulance dispatching software* (ADS) was developed in C# [11]. Among its components, the default ambulance allocator can be replaced by an alternative *AtStationAllocator* which first allocates ambulances that are available at station. Extra components include *TrafficJamAllocator* or *StatusDetector*. The former re-allocates incidents for which the allocated one is stuck in a traffic jam; the latter automates the status reporting.

The *environment simulator* simulates the behavior of ambulance staff, dispatchers and other environment agents. For example, it "presses" buttons on mobile data terminals (MDT), "drives" the ambulance to the incident location, and so forth. The simulator runs in a separate process and communicates with the ADS through a network socket.

The *adaptation tool*, developed in C#, implements our techniques [11]. The tool accepts a textual representation of the goal/obstacle model as input together with their formal specifications. It runs in a separate process.

C. Results

We simulated the operation of the ambulance dispatching system in Brussels, considering 15 ambulances and 10 simultaneous incidents. Over a 4-hour simulation, a total of 100 incidents was reached. In comparison, the real Brussels

ambulance dispatching system handles about 25 ambulances and 250 incidents per day. The simulation was performed on a MacBook Pro 3Ghz equipped with 16Gb of RAM.

At RE time, our monitoring tool built 158 monitors for all probabilistic leaf obstacles in about 10 seconds. At runtime, every second, 351 predicates were monitored, the monitors were updated, and the satisfaction rate of the high-level goals Achieve [IncidentResolved] and Avoid [AmbulanceMobilizedOnRoad] was computed. Every 5 minutes, the tool compared the monitored satisfaction rate of these goals with their respective RDS. When required, it computed the most appropriate countermeasures. Later on, the tool called the corresponding activation/deactivation procedure for reconfiguring the ADS. The optimization process took about 2 minutes. During simulation, however, one elapsed second corresponds to ten seconds in reality. This explains why Fig. 7 exhibits a delay between the time at which an adaption is found necessary and the time at which it is deployed.

We ran three simulations over 4 hours to cover the following three scenarios.

1. Rush Hour. During rush hour, the obstacles AmbulanceStuckInTrafficJamTowardsIncident and AmbulanceStuckInTrafficJamTowardsHospital have an increased satisfaction rate. This caused the countermeasures Achieve [AmbStuckInTrafficJamReAlloc] and Achieve [PatientToHospitalWhenOnSceneAndTrafficJam] to be selected, integrated and deployed in the running system. Fig. 7(a) shows the satisfaction rate of the root goal increasing again beyond its RDS threshold, as it is no longer obstructed by the two obstacles. The deployment of the first countermeasure changes the software. The second countermeasure does not change it; the goal specification is only relaxed to allow more time between the incident scene and the hospital.

2. On-Road Mobilization. At night, the RDS for the goal Avoid [AmbulanceMobilizedOnRoad] is .5. During the day, however, ambulance staff prefer to not intervene on multiple incidents without going back to their station. The RDS is increased to .8, as Fig. 7(b) shows. As a result, the countermeasure Achieve [AmbAllocAtStationWhenIncReported] was deployed during the day. The adaptation replaced the *DefaultAllocator* component. The new allocation strategy reduced the satisfaction rates of the obstacle AmbMobilizedOnRoad and guaranteed the goals's RDS.

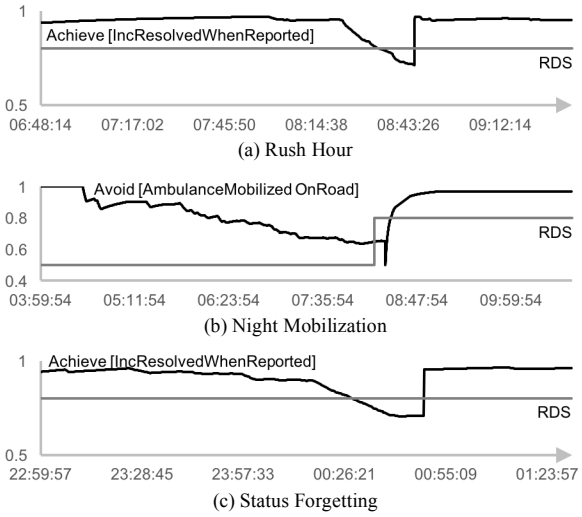

Figure 7. Simulation Scenarios

3. Forgetting Ambulance Status. Late at night and early in the morning, ambulance staff tend to forget to push buttons. This results in an increase in the satisfaction rate of obstacles such as StatusAtHospitalNotEncoded. During that period, the goals Achieve [AutomatedOnSceneDetection] and Achieve [AutomatedAtHospitalDetection] were selected, integrated and deployed. This caused their satisfaction rate to increase again beyond their RDS threshold. Fig 8(c) shows the satisfaction rate of Achieve[IncResolvedWhenReported].

D. Discussion

Our techniques were felt to help significantly for the following reasons.

Precise semantics in terms of behaviors. Thanks to formal specifications being anchored on real-world phenomena, the mapping between predicates and the running software system was straightforward. For example, the evaluation of the predicate *ambulanceA9OnScene* triggered a simple query in the database.

All monitored items had a clear, precise meaning. Surprising results were easily understandable, e.g., our technique identified inaccurate specifications with missing conditions or unrealistic time constraints.

Traceability of monitored indicators and deployed countermeasures. The monitored satisfaction rates of high-level goals significantly helped understanding whether an increase in the satisfaction rate of an obstacle is critical. Comparing their monitored satisfaction rate with their RDS provided a traceable criterion for system adaptation.

For example, the activation of the *TrafficJamAllocator* software component in the ADS is directly traceable to its countermeasure goal Achieve [AmbulanceInTrafficJamReAlloc]. The latter in turn is traceable to high-level goals such as "an ambulance shall be on scene within 10 minutes".

Model-based adaptation. The selection of most appropriate countermeasure was driven by the goal/obstacle model. As the results showed, their dynamic selection ensured that the high-level probabilistic goals remained satisfied. Without our technique, the monitoring and adaptation of the ambulance dispatching software would have required dedicated, application-specific code to be written.

No explicit behavior modelling. The ambulance dispatching system exhibits complex states and parallelism among processes. Building a complete, consistent, and adequate state machine model for this system appears quite hard.

Other benefits. The formalization effort is kept minimal as only leaf obstacles need to be formalized. In addition, a change in the formal specification of a leaf obstacle does not require source code modification.

This validation case study also highlighted areas for improvement. In particular, the satisfaction rates estimated by experts [8] might be used to improve the accuracy of the monitored satisfaction rates in case only few data are available. The monitored satisfaction rates should also be

filtered to smooth out the noise caused by a low number of observations. The technique in [22] might be used to improve the quality of monitoring.

As in many monitoring-based self-adapting systems, the monitoring task may impact on the performance of the monitored system. Our preliminary experience suggests that most of the impact may be transferred to a separate computer to reduce the footprint on the monitored system.

IX. Related work

Earlier research efforts were devoted to self-adaptation driven by runtime goal replacement in view of obstacle-based uncertainty [4, 14, 19]. In particular, [14] describes an overall process, with accompanying tactics, whereby goals specified in the RELAX language can be added or weakened to address obstacle-based uncertainty. Claims may be further added to record the rationale for decisions under uncertainty; when falsified they may be RELAXed or alternative goals may be selected [45]. In [4], adaptive goals are introduced to trigger adaptations when they are violated. Both [4] and [45] capture partially satisfied goals in fuzzy logic. All those efforts differ from ours in that our goals and obstacles are probabilistic; moreover, they are specified in a temporal logic enabling: a precise semantics in terms of observed states and behaviors; model checking techniques for probabilistic obstacle monitoring; formal up-propagation of obstacle consequences through the goal model; and formal goal replacements at runtime.

Awareness requirements capture requirements about the runtime success or failure of other requirements [48]. Runtime monitors can be produced to check whether such requirements are satisfied. In [49], evolution requirements are introduced to specify changes to other requirements. The focus there is on meta-requirements on requirements.

Other RE frameworks were introduced to reason about requirements of self-adaptive systems. In [39], TROPOS is extended for modeling self-adaptive systems with faults. Requirements on self-adaptive systems were also studied [42, 43, 44, 40]. Alternatives are selected there to meet high-level requirements and user preferences. These efforts are somewhat limited by lack of precise characterization in terms of observed states and behaviors.

Other efforts have focused on non-functional requirements that can be probabilistically quantified [18, 21, 23, 25, 26]. The software is modeled as a dynamic product line; different configurations define the space of possible adaptations. Runtime verification drives the selection of most appropriate configurations at system runtime. Our approach shares similarities in terms of probabilistic formalization and use of formal verification techniques. It does however not require building an explicit behavior model, and benefits from the goal/obstacle refinement structure.

Monitoring techniques such as [5, 46, 51] focus on failure detection; our focus in on probabilistic assessment. In [46, 47], the observed trace is divided into smaller samples so that properties can be checked on each sample; statistical reasoning is then applied to extract a probability. Our approach proposes an alternative without sampling the observed behavior into fixed-size samples.

Goals with partial degrees of satisfaction were also considered in [35] to evaluate alternative options and derive quantitative requirements at RE time. In contrast with our probabilistic framework, the reasoning there relies on ad-hoc domain-specific variables and equations.

X. Conclusion

Software systems should adapt to changing environmental conditions in order to keep their goals satisfied. The paper proposed an obstacle-driven runtime adaptation approach aimed at increasing the actual satisfaction rate of probabilistic system goals. Leaf obstacles are monitored at runtime to let the system dynamically switch to more appropriate countermeasure goals that increase the satisfaction rate of the system's high-level goals under the current conditions. The approach ensures that the required degree of satisfaction of high-level goals remains satisfied when obstacle satisfaction rates are changing.

Monitored satisfaction rates are defined precisely in terms of observed states and behaviors. Our monitoring technique extends the LTL_3 approach [6] to support monitoring of probabilistic assertions. The monitors are built at RE time from the formal specification of leaf obstacles; at runtime, virtual copies of LTL_3 monitors keep track of obstacle satisfaction. State probabilities for each observed state can thereby be obtained to yield satisfaction rates for the monitored obstacles. The latter are propagated through the obstacle/goal model up to the system's high-level goals. The satisfaction rates obtained for these goals are compared with their required degree of satisfaction; when the former falls below the latter, more appropriate countermeasures replace the current ones to remain above the required threshold.

Our techniques were applied to two non-trivial mission critical systems: a flood detection system and an ambulance dispatching system. They are supported by a prototype tool in C# [11]. The tool enables the monitoring of C# programs and triggers adaptations of these. The software for the flood detection system and for the ambulance dispatching system are both available to replicate our experiments [11].

A next possible step for runtime adaptation would monitor the costs of countermeasures, such as performance cost or energy cost, and their activation/deactivation costs. Supporting expert estimates with uncertainty margins might also improve the selection of countermeasures. More advanced reasoning based on stochastic processes to anticipate changes and trigger adaptation *before* the goals become unsatisfied would be worth investigating as well. Last but not least, machine learning techniques might usefully complement our approach to support "unknown unknowns", where new obstacles and their resolution would be identified at system runtime.

Dynamic countermeasure replacement raises problems of consistency and stability of the running software. These problems are common to self-adapting systems; they should be better understood and further studied.

Acknowledgment. This work was supported by the RICOSORE project (FRS Nr. T.0134.14). Thanks to S. Busard and B. Lambeau for useful discussions, and to the reviewers for their comments.

REFERENCES

[1] D. Alrajeh, J. Kramer, A. van Lamsweerde, A. Russo and S. Uchitel, "Generating Obstacle Conditions for Requirements Completeness", *Proc. ICSE'2012: 34th Intl. Conf. Softw. Eng.*, Zürich, 2012.

[2] D. Alrajeh, A. van Lamsweerde, J. Kramer, A. Russo, S. Uchitel, "Risk-Driven Revision of Requirements Models", *38th International Conference on Software Engineering*, Austin, TX, 2016.

[4] L. Baresi, L. Pasquale, P. Spoletini, "Fuzzy goals for requirements-driven adaptation." *18th IEEE Intl. Requirements Eng. Conf.*, 2010.

[5] H. Barringer et al., "Rule-based runtime verification." *Intl. Wkshop on Verif., Model Checking, and Abstract Interpr.* Springer, 2004.

[6] A. Bauer, M. Leucker, C. Schallhart, "Runtime verification for LTL and TLTL", *ACM Trans. Softw. Eng. (TOSEM)*, 20(4), 2011.

[7] A. Cailliau, A. van Lamsweerde, "Assessing requirements-related risks through probabilistic goals and obstacles", *Req. Eng. Journal,* 18(2), 2013.

[8] A. Cailliau, A. van Lamsweerde, "Integrating Exception Handling in Goal Models", *22th IEEE Intl. Req. Eng. Conf.,* 2014.

[9] A. Cailliau, A. van Lamsweerde, "Handling Knowledge Uncertainty in Risk-Based Requirements Engineering", *Proc. RE 2015: 23th IEEE Intl. Req. Eng. Conf.,* 2015.

[10] A. Cailliau, A. van Lamsweerde, "Runtime Monitoring and Resolution of Probabilistic Obstacles to System Goals", *UCL Report*, Dept. Ingénierie Informatique, Jan. 2017.

[11] A. Cailliau, https://www.github.com/ancailliau.

[12] B. H. C. Cheng, et al., "Software Engineering for Self-Adaptive Systems: A Research Roadmap", *Software Engineering for Self-Adaptive Systems*, LNCS 5525, 2009.

[13] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, *Software Engineering for Self-Adaptive Systems*, LNCS 5525, 2009.

[14] B.H.C. Cheng, et al, "A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty." *Model Driven Eng. Lang. and Sys.*. Springer, 2009.

[15] E. M. Clarke, O. Grumberg, D. Peled., *Model checking*, MIT, 1999.

[16] T. H. Cormen, *Introduction to algorithms*, MIT press, 2009.

[17] M. d'Amorim, G. Roşu, "Efficient monitoring of ω-languages." *Intl. Conf. on Computer Aided Verification.* Springer, 2005.

[18] I. Epifani, C. Ghezzi, R. Mirandola, G. Tamburrelli, "Model evolution by run-time parameter adaptation." *Proceedings of the 31st International Conference on Software Engineering*, IEEE, 2009.

[19] M.S. Feather, S. Fickas, A. van Lamsweerde, C. Ponsard, "Reconciling system requirements and runtime behavior." *Intl. workshop on Software specification and design*, IEEE, 1998.

[20] M.S. Feather and S.L. Cornford, "Quantitative Risk-Based Requirements Reasoning", *Req. Eng. Journal*, 8(4), 2003.

[21] A. Filieri, C. Ghezzi, A. Leva, M. Maggio, "Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements," *Intl. Conf. on Auto. Soft. Eng.* (ASE), IEEE, 2011.

[22] A. Filieri, L. Grunske, A. Leva, "Lightweight adaptive filtering for efficient learning and updating of probabilistic models." *IEEE/ACM Intl. Conf. on Software Engineering (ICSE)*, IEEE, 2015.

[23] A. Filieri, G. Tamburrelli, C. Ghezzi, "Supporting Self-adaptation via Quantitative Verification and Sensitivity Analysis at Run Time", IEEE Transaction on Software Engineering, vol. 43, no. 1, 2016.

[24] D. Garlan et al, "Rainbow: Architecture-based self-adaptation with reusable infrastructure.", *Computer* 37(10), pp 46-54, 2004.

[25] C. Ghezzi, G. Tamburrelli, "Reasoning on non-functional requirements for integrated services." *17th IEEE International Requirements Engineering Conference*, IEEE, 2009.

[26] C. Ghezzi, A. M. Sharifloo, "Dealing with non-functional requirements for adaptive systems via dynamic software product-lines." *Soft. Eng. for Self-Adapt. Syst. II*, Springer, 2013.

[27] D. Giannakopoulou, K. Havelund, "Runtime analysis of linear temporal logic specifications." *Proc. of the 16th IEEE Intl Conf. on Automated Software Engineering*, 2001.

[28] H. J. Goldsby et al, "Goal-based modeling of dynamically adaptive system requirements." *15th Annual IEEE Intl. Conf. and Workshop on the Engineering of Computer Based Systems*, 2008.

[29] K. Havelund, G. Roşu, "Efficient monitoring of safety properties" *Intl Jour. on Soft. Tools for Tech. Transfer*, 6(2), 158-173, 2004.

[30] D. Hughes et al, "GridStix: Supporting Flood Prediction using Embedded Hardware and Next Generation Grid Middleware", *Intl. Symp. on a World of Wireless, Mob. and Multim. Netw.*, IEEE, 2006.

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

[32] A. van Lamsweerde and Emmanuel Letier, "Handling Obstacles in Goal-Oriented Requirements Engineering", *IEEE Trans. Softw. Eng.*, 26(10), October 2000, 978-1005.

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

[34] R. de Lemos, H. Giese, H. A. Müller, M. Shaw, *Software Engineering for Self-Adaptive Systems II*, LNCS 7475, 2010.

[35] E. Letier, A. van Lamsweerde, "Reasoning about Partial Goal Satisfaction for Requirements and Design Engineering", *Proc. FSE 2004: 12th ACM Symp. on Found. of Software Engineering*, 2004.

[36] E. Letier, J. Kramer, J. Magee, and S. Uchitel, "Deriving event-based transition systems from goal-oriented requirements models", *Automated Software Engineering* 15(2), 2008, 175-206.

[37] R. Lutz et al, "Using Obstacle Analysis to Identify Contingency Requirements on an Unpiloted Aerial Vehicle", *Requirements Engineering Journal* Vol. 12 No. 1, 2007, 42-54.

[38] T. Menzies, E. Chiang, M. Feather, Y. Hu, J. Kiper, "Condensing uncertainty via incremental treatment learning." *Soft. Eng. with Computational Intelligence*, Springer, 2003, 319-361.

[39] M. Morandini, L. Penserini, A. Perini, "Towards goal-oriented development of self-adaptive systems." *Proc of the 2008 Intl. Wksh. on Soft. Eng. for adaptive and self-managing systems*, ACM, 2008.

[40] M. Oriol et al, "Requirements monitoring for adaptive service-based applications." *Requirements Engineering: Foundation for Software Quality*, Springer-Verlag, 2012.

[41] L. Pasquale, P. Spoletini, M. Salehie, L. Cavallaro, B. Bashar Nuseibeh, "Automating trade-off analysis of security requirements", *Requirement Engineering Journal*, Vol. 20, 2015.

[42] N. A. Qureshi, A. Perini, "Engineering adaptive requirements." ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS'09, IEEE, 2009.

[43] N. A. Qureshi, A. Perini, "Requirements engineering for adaptive service based applications." *18th Intl. Req. Eng. Conf.,* IEEE, 2010.

[44] N. A. Qureshi, I. J. Jureta, A. Perini, "Requirements engineering for self-adaptive systems: Core ontology and problem statement." *Adv. Information Systems Engineering*, Springer-Verlag, 2011.

[45] A. J. Ramirez et al, "Relaxing claims: Coping with uncertainty while evaluating assumptions at run time." *Intl. Conf. on Model Driven Eng. Lang. and Sys*, Springer-Verlag, 2012, 53-69.

[46] U. Sammapun et al, "RT-MaC: runtime monitoring and checking of quantitative and probabilistic properties." *11th IEEE Intl. Conf. on Embedded and Real-Time Computing Sys. and App.,* IEEE, 2005.

[47] U. Sammapun, L., Insup, O. Sokolsky, J. Regehr, "Statistical runtime checking of probabilistic properties." *Intl. Workshop on Runtime Verification*, Springer-Verlag, 2007, pp. 164-175.

[48] V. E. S. Souza et al, "Awareness requirements for adaptive systems." *Proc. 6th Intl. Symp. on Software Engineering for Adaptive and Self-managing systems*, ACM, 2011.

[49] V. E. S. Souza, A. Lapouchnian, J. Mylopoulos, "(Requirement) evolution requirements for adaptive systems." *Proc. 7th Intl Symp. on Soft Eng for Adaptive and Self-Managing Systems*, IEEE, 2012.

[50] P. Thati, G. Roşu, "Monitoring algorithms for metric temporal logic specifications". *Electronic Notes in Theoretical Computer Science*, 113, pp.145-162, 2005.

[52] P. Zhang, W. Li, D. Wan, L. Grunske, "Monitoring of probabilistic timed property sequence charts", *Software: Practice and Experience*, 41(7), 2011, pp.841-866.