Generating Obstacle Conditions for Requirements Completeness

Dalal Alrajeh*, Jeff Kramer*, Axel van Lamsweerde†, Alessandra Russo* and Sebastian Uchitel*
*Department of Computing, Imperial College London, UK
Email: {da04,jk,ar3,su2}@doc.ic.ac.uk
†ICTEAM institute, Université catholique de Louvain, Belgium
Email: avl@info.ucl.ac.be

***Abstract*—Missing requirements are known to be among the major causes of software failure. They often result from a natural inclination to conceive over-ideal systems where the software-to-be and its environment always behave as expected. Obstacle analysis is a goal-anchored form of risk analysis whereby exceptional conditions that may obstruct system goals are identified, assessed and resolved to produce complete requirements. Various techniques have been proposed for identifying obstacle conditions systematically. Among these, the formal ones have limited applicability or are costly to automate. This paper describes a tool-supported technique for generating a set of obstacle conditions guaranteed to be complete and consistent with respect to the known domain properties. The approach relies on a novel combination of model checking and learning technologies. Obstacles are iteratively learned from counterexample and witness traces produced by model checking against a goal and converted into positive and negative examples, respectively. A comparative evaluation is provided with respect to published results on the manual derivation of obstacles in a real safety-critical system for which failures have been reported.**

***Keywords*-Requirements completeness; obstacles; risk analysis; model checking; inductive learning; goal-oriented requirements engineering.**

I. Introduction

Completeness is among the most critical and difficult challenges facing requirements engineers. Missing requirements and assumptions are reported as one of the major causes of software failure [30]. Incompleteness often arises from the lack of anticipation of exceptional conditions. The natural inclination is rather to conceive idealised systems; this prevents adverse events or conditions from being properly identified, and as a result, specifications of suitable countermeasures in such circumstances are missing.

Risk analysis is therefore at the heart of the requirements engineering process [8], [30]. A *risk* is commonly defined as an uncertain factor whose occurrence may result in some loss of satisfaction of some corresponding objective. In goal-oriented system modelling frameworks, obstacles are introduced as a natural abstraction for risk analysis when using goal models [4], [32]. An *obstacle* to a goal is a precondition for the non-satisfaction of this goal. Depending on the category of goal being obstructed, obstacles may correspond to safety hazards, security threats, inaccuracy conditions on software input/output variables with respect to their environment counterpart, etc.

Obstacle analysis roughly consists of three steps [30]: (a) identify as many obstacles as possible to every leaf goal in the systems goal refinement graph, (b) assess the likelihood and severity of each obstacle; and (c) resolve likely and severe obstacles by systematic transformations to the goal model using appropriate countermeasures.

The obstacle identification step is obviously crucial. In [32], a formal technique is described for generating obstacles by regressing goal negations through available domain properties. Although quite systematic, this technique appears costly to implement for goals formalised in a first-order real-time linear temporal logic. No tool support is available.

This paper presents an alternative, tool-supported technique for obstacle generation. A complete set of obstacles, relative to what is known about the domain, is computed by iterating the following cycle: (a) a behaviour model is synthesised from the available background properties; (b) this model is verified against the goal and against a negated form of it, in order to generate a negative trace (counterexample) and a positive trace (witness), respectively; (c) the negative trace is taken as positive example whereas the positive trace is taken as negative example input for a learning engine; (d) the learning tool generates a set of candidate obstacles that cover the positive example and exclude the negative one; (e) the user can then select from the generated obstacles those considered likely and severe, and suggest further domain properties; (f) a new cycle is applied to the background properties augmented with such properties and the negated obstacles generated at the previous cycle. The process terminates when a domain-complete set of obstacles is generated.

The synergistic use of model checking and learning technologies was already explored successfully in [1] for the generation of specifications of software operations from goals. Our contributions in this paper include the following.

- A new combination of model checking and learning for supporting the significant, challenging task of risk-driven elaboration of more complete requirements.
- Automation is increased with respect to the earlier combined use of model checking and learning (in [1], positive traces have to be elaborated manually). Moreover, the goal specification language is much more

expressive, allowing us to capture the full range of *Achieve* and *Maintain* goals (only *Immediate Achieve* goals are considered in [1]).

- Tool support is provided for formal obstacle generation, which was unavailable to date. Moreover, with respect to earlier techniques, the one here generates obstacles that are domain-consistent by construction (domain-consistency checks have to be performed separately in [32]). The technique is also guaranteed here to converge to a domain-complete set of obstacles.

The paper is organised as follows. Section II briefly provides some background on goal modelling, obstacle analysis, labelled transition systems for behaviour model analysis and synthesis, fluent linear temporal logic for goal specification and inductive logic programming for learning obstacles. Section III introduces a motivating example. The approach is detailed in Section IV. Section V discusses our experience in using our technique and tool for generating a variety of obstacles in a real safety-critical system for which failures have been reported [23]. Some related work is discussed in VI. Section VII concludes the paper.

II. Background

A. Goal-Oriented Modelling and Obstacle Analysis

A *goal* is a prescriptive statement of intent to be satisfied by the agents forming the system. The latter include the software to be developed, pre-existing software, devices such as sensors and actuators, people, etc. The word *system* thus refers to the software-to-be together with its environment. Unlike goals, *domain properties* are descriptive statements about the problem world (such as natural laws). A goal model is an AND/OR graph showing how goals contribute positively or negatively to each other [30]. Parent goals are obtained by abstraction whereas child goals are obtained by refinement. In a goal model, each leaf goal is assigned to a single system agent as a *requirement* or *assumption* depending on whether it is assigned to the software-to-be or an environment agent, respectively.

A goal may be *behavioural* or *soft* depending on whether it can be satisfied in a clear-cut sense or not. This paper focusses on behavioural goals. A behavioural goal captures intended behaviour declaratively and implicitly. It can be of type *Achieve* or *Maintain/Avoid*. The specification pattern for an *Achieve* goal is **if *C* then sooner-or-later *T***, where ***C*** denotes a *current* condition and ***T*** a *target* condition, with obvious particularisations to *Immediate Achieve*, *Bounded Achieve* and *Unbounded Achieve* goals. The pattern for a *Maintain* (resp. *Avoid*) goal is [**if *C* then**] **always *G*** (resp. [**if *C* then**] **never *B***), where ***G*** and ***B*** denote a *good* and *bad* condition, respectively.

An obstacle to a goal is a domain-satisfiable precondition for the non-satisfaction of this goal. The following definition makes this more precise [30], [32].

Definition 1. *Let Dom be a set of known domain properties and G a goal assertion. An assertion O is said to be an obstacle to G in Dom iff the following conditions hold:*

- $\{O, Dom\} \models \neg G$ (*obstruction*)
- $\{O, Dom\} \not\models false$ (*domain consistency)*

Obstacles can be AND/OR refined into sub-obstacles, resulting in a goal-anchored form of risk tree. In such tree, the root obstacle is the negation of the associated leaf goal in the goal model; an AND-refinement captures a combination of sub-obstacles entailing the parent obstacle; an OR-refinement captures alternative ways of entailing the parent obstacle —and, recursively, of obstructing the corresponding leaf goal; the leaf sub-obstacles are single, fine-grained obstacles whose likelihood can be easily estimated.

Definition 2. *Let Dom be a set of known domain properties and G a goal assertion. A set of obstacles $O_1, O_2, \ldots, O_n$ to G is said to be domain-complete for G in Dom if the following condition holds:*

$\neg O_1, \neg O_2, \ldots, \neg O_n, Dom \models G$ (*domain-completeness*)

Note that completeness, while highly desirable, is bounded by what we know about the domain. Obstacle generation techniques should therefore support the incremental elicitation of relevant domain properties as well. Pattern-based heuristics can be used for more focussed elicitation [30]. In particular, for the above patterns defining *Achieve* and *Maintain/Avoid* goals, domain properties of the form **if *T* then *N*** or **if *G* then *N*** are worth eliciting, where ***N*** denotes a *necessary* condition for the target condition ***T*** or good condition ***G***; they result in obstacles of form [***C* and**] **never *N*** or [***C* and**] **sooner-or-later not *N***, respectively.

B. Model Checking

Model checking is an automated technique for verifying that a model M satisfies property ϕ, written $M \models \phi$. In this paper we are interested in declarative models and properties both expressed in a temporal logic called Fluent Linear Temporal Logic (FLTL) [11] with labelled transition systems as the semantic structure.

In FLTL, a fluent is a propositional atom defined by a set I_f of initiating events, a set T_f of terminating events and an initial truth value either *true* or *false*. Given a set of event labels *Act*, we write $f = \langle I_f, T_f, Init \rangle$ as a shorthand for a fluent definition, where $I_f \subseteq Act$, $T_f \subseteq Act$ and $I_f \cap T_f = \emptyset$, and $Init \in \{true, false\}$. FLTL assertions use standard operators for temporal referencing such as: $\bigcirc$ (at the next time-point), $\bullet$ (at the previous time-point), $\Diamond$ (some time in the future), $\Box$ (always in the future), U (always in the future until), W (always in the future unless) and $\Rightarrow$ (always implies) [21]. We will also use real-time operators $\Box_{\sim x}, \Diamond_{\sim x}$ and $\mathsf{U}_{\sim x}$ where $\sim \in \{<, \leq, >, \geq\}$ and x denotes a number of time-points [13].

The semantics of FLTL assertions is defined in terms of behaviour models called Labeled Transition Systems (LTSs). An LTS is an automaton defined by a structure

$(\mathcal{Q}, \mathit{Act}, \delta, q_0)$, where $\mathcal{Q}$ is a finite set of states, *Act* is a set of event labels, $\delta \subseteq \mathcal{Q} \times \mathit{Act} \times \mathcal{Q}$ is a labeled transition relation, and q_0 is an initial state. The global system behaviour is captured by the parallel composition, denoted as $\|$, of LTSs (one for each component) by interleaving their behaviour but forcing synchronisation on shared events [20]. A trace in an LTS is a sequence of events from the initial state.

Given a trace *tr* over *Act* and fluent definitions D, a fluent is said to be *true* (resp. *false*) in a trace *tr* at position i, denoted $\mathit{tr}, i \models f$, *iff* either of the following conditions hold: (a) the fluent is initially *true* and no terminating event has occurred since, or (b) some initiating event has occurred with no terminating event occurring since then.

An LTS can be synthesised from a safety assertion P, written in FLTL, using the algorithm in [11]. The resulting LTS, denoted $L(P)$, is an LTS that captures all infinite traces over the alphabet *Act* that satisfy P. Also, an FLTL property ϕ can be verified against an LTS $L(P)$ using the model checking algorithm described in [11] and implemented in the LTSA tool [20]. This procedure involves constructing an automaton $\mathit{Büchi}(\neg\phi)$ that recognises all infinite traces over the alphabet *Act* violating ϕ and checking that the synchronous product $L(P) \| \mathit{Büchi}(\neg\phi)$ is empty. Any trace leading to the error state in this product is a *counterexample* to the property ϕ, while any trace not leading to the error state is a *witness* to the property ϕ.

C. Inductive Logic Programming

Inductive Logic Programming (ILP) [22] is a machine learning technique for generating a generalisation H that, together with a given knowledge base B, covers a given set of positive examples $\{e^+\}$ and excludes a set of negative examples $\{e^-\}$, where B, H, e^+ and e^- are expressed as logic programs [18]. A *Mode Declaration* (*MD*) is a form of language bias that defines the space of plausible generalisations (solutions) for a given ILP task, by specifying the syntactic form of the generalisations that can be learned. We use $s(\mathit{MD})$ to denote the set of all plausible generalisations rules that can be constructed from *MD*. For the rest of this paper, we define an inductive task to be a tuple $\langle B, \{e^+\}, \{e^-\}, \mathit{MD}\rangle$ and an inductive solution as follows.

Definition 3. *Let $\langle B, \{e^+\}, \{e^-\}, \mathit{MD}\rangle$ be an inductive task. The logic program H, where $H \subseteq s(\mathit{MD})$, is an inductive solution iff, $\forall e^+.\ B \wedge H \models e^+$ and $\forall e^-.\ B \wedge H \not\models e^-$, where $\models$ is entailment under stable model semantics [10].*

We consider here logic programs that use the following predicates from [12]: *fluent*(f) to assert that f is a fluent, *event*(e) for an event e, *timepoint*(i) for time-point i, *trace*(s) for trace s, *holdsAt*(f, i, s) to express that the fluent f holds at time-point i in trace s, *happens*(e, i, s) to express that event e happens at time-point i in trace s, and *initiates*(e, f) (resp. *terminates*(e, f)) to capture fluents that are initiated (resp. terminated) by event e. The knowledge base also comprises a set of frame axioms for determining fluents' truth values and semantic constraints (e.g., events cannot happen concurrently), denoted *Ax*.

III. Motivating Example

In this section we give an overview of the approach and introduce our running example. We use a simplified version of the train control system introduced in [30]. The example is propositional, rather than first order, for simplicity; it comprises a train, a signal and a driver. A train may *start* or *stop* moving on the track. The signal may be *set to stop* or *go*. The view of the signal may be *clear* or *obstructed*. The driver may *respond* to or *ignore* the stop signal. We define the fluents for the train control system as follows.

$$\begin{array}{lcl} \mathit{TrainStopped} & = & \langle \mathit{stop_train}, \mathit{start_train}, \mathit{false}\rangle \\ \mathit{StopSignal} & = & \langle \mathit{set_to_stop}, \mathit{set_to_go}, \mathit{false}\rangle \\ \mathit{SignalVisible} & = & \langle \mathit{clear_signal}, \mathit{obstruct_signal}, \mathit{true}\rangle \\ \mathit{DriverResponsive} & = & \langle \mathit{driver_responds}, \mathit{driver_ignores}, \mathit{true}\rangle \end{array}$$

Consider the goal stating that the train shall stop when the signal is set to stop:

Goal *Achieve* [*TrainStoppedAtBlockSignalIfStopSignal*]

$$\mathit{StopSignal} \Rightarrow \bigcirc\ \mathit{TrainStopped} \quad (1)$$

Suppose the domain properties include the following necessary conditions for the train to stop:

$$\bigcirc \mathit{TrainStopped} \Rightarrow \mathit{DriverResponsive} \quad (2)$$

which states that if a train stops, the train driver was responsive to the stop command, and

$$\bigcirc \mathit{TrainStopped} \Rightarrow \mathit{SignalVisible} \quad (3)$$

meaning that if a train stops, the stop signal was visible.

Consider a system that satisfies the domain properties (2) and (3). A simple goal violation scenario of such a system is one in which the signal is set to stop and then although the stop signal is visible initially, the driver ignores the signal and the train does not stop. Similarly, another scenario possible in the domain is when the signal is set to stop and the signal is visible initially, the signal is then obstructed (e.g. because of foggy weather) and consequently the train does not stop. Such scenarios represent situations where the goal satisfaction is at risk.

The problem is that the goal *Achieve* [*TrainStoppedAtBlockSignalIfStopSignal*] is too ideal; it presupposes that nothing in the environment will prevent the goal from being achieved. The goal can only be achieved if it is assumed that the driver is always responsive and the stop signal is always visible, among others; these two assumptions are likely to be violated from time to time.

In both scenarios, we can see that some exceptional circumstance has prevented the goal from being satisfied. In the first case, the driver not being responsive is an obstacle preventing the goal *TrainStoppedAtBlockSignalIfStopSignal* from being achieved; this obstacle can be formally expressed as $\diamond(\mathit{StopSignal} \wedge \bigcirc\neg \mathit{DriverResponsive})$. In the second case,

the signal being not visible, $\diamond(StopSignal \wedge \bigcirc\neg SignalVisible)$, is another obstacle to the goal.

Detecting obstacle conditions under which goals may be violated is essential for not missing requirements that would prescribe what the software should do in order to properly handle such unexpected situations. In this simple example, detecting obstacles is straightforward; however, for complex systems, performing such a task manually is likely to be error-prone and miss important obstacles. The technique in this paper can automatically detect all obstacles to a goal for a set of known domain properties.

IV. Generating Obstacles to Goal Satisfaction

The approach proposed here considers a set of goals and background properties as input and iteratively computes obstacles to the goals. Each iteration can be subdivided into three main phases (see Figure 1):

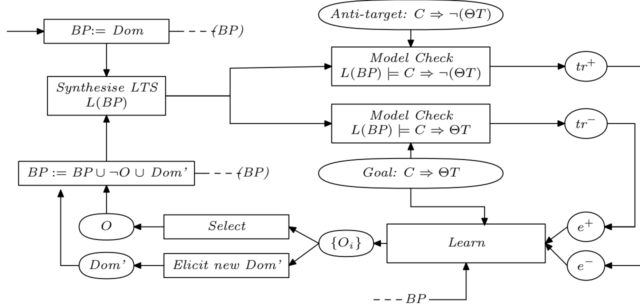

Figure 1. Overview of the proposed obstacle generation approach.

- The first phase generates domain-consistent counterexample and witness traces for a goal. This is achieved by synthesising a behaviour model $L(BP)$ from the background properties *BP* currently available and performing two model checks. To obtain a counterexample trace tr^- to a goal, taking the form $C \Rightarrow \Theta T$ where Θ is a temporal operator, the model is checked against it. To obtain a witness tr^+ to the goal, the model is checked against a perturbation of the property that asserts that the goal target is violated: $C \Rightarrow \neg\, \Theta T$.
- The second phase is concerned with learning obstacles. A learning system is given the background properties and the goal in addition to the counterexample and witness traces generated in the first phase. Critically, it is the counterexample trace to the goal that is provided to the learning system as a positive example e^+, i.e., an example in which an obstacle is satisfied. The witness trace to the goal is provided as a negative example e^- to the learning system, i.e., an example in which the obstacle is not satisfied. The learning system generates a set of candidate obstacles $\{O_i\}$ that cover the positive example while excluding the negative one.
- The third phase, contrary to the others, requires human intervention. An engineer makes a selection O from the set of proposed obstacles based on those considered likely and severe. The selected obstacles are negated and added to the background properties. This is to allow for further identification of obstacles in the next iteration. In this phase, the engineer can also suggest further domain properties, based on the learned obstacles and provided they are consistent with the existing background properties to which they are added, to allow for identification of further obstacles or finer-grained sub-obstacles.

The process terminates when a domain-complete set of obstacles is generated (see Definition 2). In what follows we structure the presentation according to these phases.

A. Generating goal counterexamples and witnesses

Consider the LTS shown in Figure 2 which is synthesised from the background properties including (2) and (3) of Section III. Notice that some of the traces the LTS exhibits satisfy the system goal *Achieve* [*TrainStoppedAtBlockSignalIfStopSignal*] while others do not. Traces which satisfy the goal are examples in which no obstacles are present (e.g., the signal is set to stop and the train stops). On the other hand, traces which violate the goal are examples of obstacles that are present (e.g., the view of the stop signal is not clear). To automatically find such examples, we perform two model checks which are explained below.

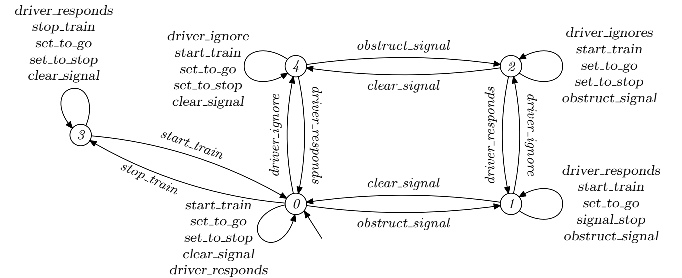

Figure 2. The LTS $L(BP)$ for the train control system.

1) Goal counterexamples: For a given set of background properties *BP*, the computation of a counterexample trace to a goal G is done in a straightforward manner by checking the LTS $L(BP)$ against G. If the model checker finds that $L(BP)$ contains a trace that violates G, it is produced automatically. For example, model checking the LTS in Figure 2 against the goal *Achieve* [*TrainStoppedAtBlockSignalIfStopSignal*] using the LTSA model checker gives the following violation.

```
Violation in TrainStoppedAtBlockSignalIfStopSignal:
  set_to_stop          StopSignal
  driver_ignores       StopSignal
```

The left column represents a sequence of events in which the goal is violated. The column on the right indicates the fluents that are true immediately after the occurrence of the event to their left. This trace fragment exemplifies a case in which the train does not stop after the signal is set to stop. This behaviour thus satisfies an obstacle which prevents the goal from being achieved.

If checking the LTS $L(BP)$ against G does not yield a counterexample, this means that $L(BP)$ satisfies G and hence there are no more obstacles to be generated; the three-phase cycle then comes to an end.

2) Goals witnesses: Conceptually, a witness trace to a goal is obtained by checking the LTS $L(BP)$ against a property stating that the goal is violated. However, this might lead to traces that vacuously satisfy the goal. If, for example, the goal is of the form $C \Rightarrow \Theta T$ the property asserting that the goal is violated is $\neg(C \Rightarrow \Theta T)$, that is, $\diamond(C \wedge \neg\Theta T)$. Checking $L(BP)$ against the latter property could generate a counterexample that never satisfies C. Such counterexample is a witness that vacuously satisfies the goal $C \Rightarrow \Theta T$ as the antecedent is never true. We are interested in obtaining witnesses where both C and ΘT hold. Hence we need to check $L(BP)$ against a variant property $C \Rightarrow \neg\Theta T$. We refer to this property as the *anti-target* for the goal $C \Rightarrow \Theta T$.

Table I shows the corresponding anti-target assertions for goals expressed in *Achieve* and *Maintain* modes. Anti-target assertions for goals expressed in *Avoid* modes can be constructed by simply removing the negation preceding the target condition in the anti-target assertion of the corresponding *Maintain* assertions in Table I.

If checking $L(BP)$ against G's anti-target yields no counterexamples, this means that $L(BP)$ satisfies the anti-target, i.e., there are no traces that satisfy the goal non-vacuously. This is an indication of problem in the formulation of the goal itself, which needs to be weakened, or in the background properties which need to be revised. This situation is beyond the scope of this paper.

Table I
GOALS AND THEIR CORRESPONDING ANTI-TARGET PATTERNS

Mode	Goal Pattern	Anti-target Pattern
Achieve	$C \Rightarrow \bigcirc T$	$C \Rightarrow \bigcirc \neg T$
	$C \Rightarrow \diamond_{\leq X} T$	$C \Rightarrow \square_{\leq X} \neg T$
	$C \Rightarrow \diamond T$	$C \Rightarrow \square \neg T$
Maintain	$C \Rightarrow T$	$C \Rightarrow \neg T$
	$C \Rightarrow \square T$	$C \Rightarrow \diamond \neg T$
	$C \Rightarrow G \mathsf{W} T$	$C \Rightarrow \neg T \mathsf{U} \neg G$
	$C \Rightarrow \bullet T$	$C \Rightarrow \bullet \neg T$
	$C \Rightarrow \square_{\leq X} T$	$C \Rightarrow \diamond_{\leq X} \neg T$

Returning to the example of the previous section, to generate a witness to the goal *Achieve* [*TrainStoppedAtBlockSignalIfStopSignal*], the LTS in Figure 2 is checked against the anti-target assertion:

$$\mathit{StopSignal} \Rightarrow \bigcirc \neg \mathit{TrainStopped} \tag{4}$$

This results in the violation trace below where the signal is set to stop and the train stops immediately afterwards.

```
Violation in NotTrainStoppedAtBlockSignalIfStopSignal:
  signal_stop        StopSignal
  stop_train         StopSignal && TrainStopped
```

B. Learning Obstacles from Traces

The next step involves the generation of obstacles from the produced counterexample and witness traces to the goal. To perform this computation, ILP is deployed. The application of ILP to the obstacle generation task comprises three main steps: (1) definition of an ILP task through appropriate encoding of background properties, goals, counterexample and witness, (2) computation of an inductive solution, and (3) decoding of the inductive solutions into obstacles. These steps are done automatically and are hidden from users who are only shown the final set of possible obstacles.

1) ILP task for learning obstacles: As explained in Section II-C, ILP computes generalised rules from given examples and knowledge base, expressed as logic programs. Hence, to learn obstacles, the background properties, goals, and counterexample and witness traces have to be mapped into a logic program; as in any learning task, a suitable language bias has also to be chosen. The encoding uses, in addition to the predicates described in Section II, a collection $\mathcal{L}_O$ of anti-target predicates, specific to the anti-target patterns in Table II, and a collection $\mathcal{L}_T$ of target predicates specific to the target patterns in Table III. The correspondence illustrated in these tables considers the target condition to be a conjunction or disjunction of fluents and that the predicates are introduced for each fluent in T. Examples of anti-target predicates include *obstructed_next*(f, i, s), which means that a fluent f is prevented from being true at the next time-point from i in trace s, and *obstructed_always_by*(f, i, x, s), which states that a fluent f is obstructed from being true for x time-points after time-point i in trace s. Examples of target predicates include *holdsAt_eventually*(f, i, s), meaning that fluent f is true sometime in the future after time-point i.

Table II
ILP ENCODING OF ANTI-TARGET

Anti-target Pattern	Corresponding obstacle predicates
$\bigcirc \neg T$	`obstructed_next(t,I,S).`
$\square_{\leq x} \neg T$	`obstructed_always_by(t,I,x,S).`
$\square \neg T$	`obstructed_always(t,I,S).`
$\neg T$	`obstructed(t,I,S).`
$\diamond \neg T$	`obstructed_eventually(t,I,S).`
$\neg T \mathsf{U} \neg G$	`obstructed_always_until(t,I1,I2,S).` `obstructed_eventually(g,I1,S).`
$\bullet \neg T$	`obstructed_before(t,I,S).`
$\diamond_{\leq x} \neg T$	`obstructed_eventually_by(t,I,x,S).`

Table III
ILP ENCODING OF TARGET

Target Pattern	Corresponding goal predicates
$\bigcirc T$	`holdsAt_next(t,I,S).`
$\diamond_{\leq x} T$	`holdsAt_eventually_by(t,I,x,S).`
$\diamond T$	`holdsAt_eventually(t,I,S).`
T	`holdsAt(t,I,S).`
$\square T$	`holdsAt_always(t,I,S).`
$G \mathsf{W} T$	`holdsAt_always_until(g,I1,I2,S);` `holdsAt_always(g,I1,S).`
$\bullet T$	`holdsAt_before(t,I,S).`
$\square_{\leq x} T$	`holdsAt_always_by(t,I,x,S).`

The knowledge base B of the ILP task includes an encoding of the background properties (fluent definitions and domain properties), the goals and a set of frame axioms *Ax* that specify conditions under which an initiating or terminating changes the truth value of fluents.

Fluent definitions are expressed as *initiates* and *terminates* rules. For example the fluent definition for *TrainStopped* is expressed as `initiates(stop_train, trainStopped)` and `terminates(start_train, trainStopped)`. Domain properties are encoded as integrity constraints, which force the learning tool to compute only obstacles that are consistent with the domain. Intuitively, every domain property is of the form $T \Rightarrow N$, where N is either a conjunction of literals or a *weak until* formula, defining a necessary condition for T. In the former case, a domain property is translated into a set of rules of the form `false :- holdsAt(t,I,S),...,Lit(n_j,I,S)`, one for each conjunct literal n_j in N, where *Lit* is an `holdsAt` literal if n_j is negated in N, and a `not holdsAt` literal otherwise. For example, the domain property $\bigcirc$ *TrainStopped*$\Rightarrow$ *DriverResponsive* in (2) is represented as

```
false:- holdsAt_next(trainStopped,I,S),
        not holdsAt(driverResponsive,I,S).
```

This means that in any stable model of B where *holdsAt_next(trainStopped,I,S)* is true, *holdsAt(driverResponsive, I,S)* must also be true. Domain properties of the form $T \Rightarrow N \mathsf{W} P$ are encoded into rules of the form

```
false:- holdsAt(t,I1,S), holdsAt_eventually(p,I1,I2,S),
      not holdsAt_always_until(n,I1,I2,S).
false:- holdsAt(t,I1,S), not holdsAt_eventually(p,I1,I2,S),
      not holdsAt_always(n,I1,S).
```

Goals, on the other hand, are encoded as rules of the form `holdAt(target,I,S):- τ(current,I,S), not O`. In this rule, O is a predicate in $\mathcal{L}_\mathcal{O}$. The general interpretation of such a rule is that the goal's target condition holds if its current condition holds and no obstacles exist. For instance, the *Achieve* goal [*TrainStoppedAtBlockSignalIfStopSignal*], expressed as ($StopSignal \Rightarrow \bigcirc TrainStopped$) is encoded as

```
holdsAt_next(trainStopped,I,S):-
      holdsAt(stopSignal,I,S),
      not obstructed_next(trainStopped,I,S).
```

This rule informally states that if a signal is set to stop and no obstacles to stopping the train at the next time-point are satisfied, then the train stops. This encoding, as shown later, plays a key role in the computation of obstacles. The encoding is proven to be sound in terms of preservation of the FLTL model consequences in the encoded logic program.

Theorem 1. *Let BP be a set of background properties expressed in FLTL let $L(BP)$ be the LTS synthesised from BP and tr a trace in $L(BP)$. Let ϕ be a property expressed in FLTL. Let τ be the encoding of BP, ϕ and tr into a corresponding knowledge base B under stable model semantics. Then the following condition holds:*
$tr \models \phi$ if and only if $B \models \tau(\phi)$

2) Inferring obstacle: Positive and negative examples need to be provided to the ILP task. In the problem at hand, the counterexample trace tr^- to a goal is taken as a positive example for learning obstacles to the goal while the witness trace tr^+ to the goal is taken as a negative example. Each event occurrence in the counterexample (resp. witness) trace is represented as a *happens* fact and added to the positive (resp. negative) examples; every fluent valuation in the counterexample (resp. witness) trace is represented as a *holdsAt* literal and also added to the positive (resp. negative) examples. For instance, the goal counterexample trace for our toy train system shown in Section IV-A can be systematically expressed as the following positive examples,

```
happens(set_to_stop,1,c).       holdsAt(stopSignal,1,c).
happens(driver_ignores,2,c).    holdsAt(stopSignal,2,c).
```

and the witness trace as the following negative examples.

```
happens(set_to_stop,1,w).    holdsAt(stopSignal,1,w).
happens(stop_train,2,w).     holdsAt(stopSignal,2,w).
                             holdsAt(trainStopped,2,w).
```

Once B, e^+ and e^- are generated, with $B = \tau(BP) \cup \tau(G) \cup Ax$, $e^+ = \tau(tr^-)$ and $e^- = \tau(tr^+)$, they are given to a non-monotonic ILP system to compute a generalisation H that covers e^+ and excludes e^-. The encoding of the goal and traces plays a crucial role in the computation of the obstacles. Since the consequent of $\tau(G)$ is true in the stable model of B when no obstacles are present whilst the positive examples indicate the contrary, the ILP task is concerned with finding rules that would prevent the derivation of the goal's target. To define the solution space, we define the mode declaration to include rules with $\mathcal{L}_\mathcal{O}$ predicates in the head, and *holdsAt* and $\mathcal{L}_\mathcal{T}$ predicates in the body. In our running example, the tool generates the following rule stating that an obstacle to stopping the train at the next time-point is the driver not being responsive.

```
obstructed_next(trainStopped, I,S):-
        holdsAt(stopSignal, I,S),
        not holdsAt_next(driverResponsive, I,S).
```

The computed explanation is then mapped back into FLTL and presented to the engineer:

$$\Diamond(StopSignal \wedge \bigcirc\neg DriverResponsive) \tag{5}$$

In many situations, the learning tool may find several solutions as obstacles to a goal. When this is the case, these alternatives are presented to the engineer to select the obstacles considered likely and severe. Theorem 2 below states that any computed solution is an obstacle to a goal.

Theorem 2. *Let BP and G be a set of background properties and a goal, respectively, expressed in FLTL. Let tr^- and tr^+ be a counterexample and witness traces to G in $L(BP)$. Let Ax be a set of frame axioms expressed in the logic programming formalism. Given the inductive task $\langle \tau(BP) \cup \tau(G) \cup Ax, \tau(tr^+), \tau(tr^-), MD\rangle$, a set of rules $H \in s(MD)$ is an inductive solution under stable model semantics iff the FLTL expression O given by $H = \tau(O)$ is an obstacle to G satisfied by tr^- and not by tr^+.*

C. Completing the Obstacles

Once the analyst selects those obstacles, their negation is added to the background properties. As the LTS synthesised

from the extended background properties may contain other violation traces to the considered goal, the process of model checking and learning is repeated again.

To generate a domain-complete set of obstacles, in every new iteration $i+1$, we look for obstacles different from those already generated and selected. The negation of each one is added to the background properties *BP* and the verification process is repeated again. If a counterexample trace is found, then the steps in Sections IV-A and IV-B are repeated until no further counterexamples to the goal are found.

In some cases, the computed obstacles $\{O_i\}$ may highlight some domain properties that are missing from the current background properties, typically, properties of the form $T \Rightarrow N$ (other necessary condition for the goal's target) or $S \Rightarrow O$ (sufficient condition for the obstacle under consideration). When this is the case, the engineer may add such properties to *BP* before the start of the new iteration.

Returning to our running example, the analysis is performed on the LTS synthesised from the background properties including the domain properties (2) and (3), fluent definitions and the negation of the obstacle (5), i.e., $\Box(\neg \mathit{StopSignal} \vee \bigcirc \mathit{DriverResponsive})$. The counterexample trace $(\mathit{set_to_stop}, \mathit{obstruct_signal})$ is generated. This trace along with the witness $(\mathit{set_to_stop}, \mathit{stop_train})$ are given to the learner tool which in turn computes the obstacle

$$\Diamond(\mathit{StopSignal} \wedge \bigcirc \neg \mathit{VisibleSignal}) \tag{6}$$

Once this is added to the current set of obstacles, a new cycle results in no further violations to the goal. Hence the set comprising the obstacles (5) and (6) is the domain-complete set of obstacles for the goal *Achieve* [*TrainStoppedAtBlockSignalIfStopSignal*]. If sufficient conditions for the learned obstacles are added to the background properties as additional domain properties, the process will go on to generate finer-grained sub-obstacles. Theorem 3 states that for a given set of background properties *BP* and goal G that is consistent with *BP*, the proposed approach will converge to a domain-complete set of obstacles to goal G.

Theorem 3. *Let BP be a set of background properties, and G a goal expressed in FLTL, consistent with BP. Let* Tr^- *be the set of shortest finite counterexample traces in the composition* $L(BP) \| \mathit{Buchi}(\neg G)$. *Let* Tr^+ *be a set of finite witness traces to the goal G in the composition* $L(BP) \| L(G)$. *Let Ax be a set of frame axioms expressed in the logic programming formalism. Given the inductive task* $\langle \tau(BP) \cup \tau(G) \cup Ax, \tau(Tr^+), \tau(Tr^-), MD \rangle$, *there exists a set* $\{H_i\}$ *of inductive solutions under stable model semantics, such that* $BP \cup \neg O_1 \cup .... \cup \neg O_n \models G$, *where* $H_i = \tau(O_i)$ *for* $1 < i < n$.

V. Validation

We show here an application of the proposed approach to the London Ambulance Service (LAS) [9]. We report on our findings and illustrate the various steps in our approach with respect to two goal patterns: *Bounded* and *Unbounded Achieve*. The application to goals under *Maintain* and *Avoid* modes was similar. For the purpose of validating our approach, we considered a set of obstacles which were formally derived manually in [14], [32].

The LAS specification, as presented in [14], [32], is formalised in a synchronous form of first-order Linear Temporal Logic (LTL) where the goals are expected to hold at fixed time-points but maybe violated in between, and several events may occur between two consecutive time-points. LTSs, on the other hand, assume an asynchronous interpretation of LTL expressions. Therefore, to preserve these semantics when applying model checking, the available synchronous specification was systematically translated into asynchronous FLTL using the algorithm described in [15]. This translation makes use of a special event *tick* which is introduced in the alphabet of the LTS to mark the time-points at which goals are expected to hold. Similarly, the language of our logic programs is extended with this notion by defining auxiliary predicates such as *holdsAt_tick* and *holdsAt_eventually_tick*. Fluent definitions were also extracted from the specification using the technique in [15].

A. Experimental Results

Our experiments generated all the obstacles which were manually obtained described in [32] as well as other obstacles that were not found there. The outcomes also confirmed that our approach provides support for eliciting new domain properties. For instance, when checking the background properties against one of the LAS goals reported later in this section, the model checker produced the counterexample (*tick, a1.assign, a1.inc1.allocate, inc1.resolve, ...*) where an incident is stated to be resolved before the ambulance has been mobilised and has intervened in the incident. Such traces help engineers in detecting goals or domain properties that are too weak, e.g., the domain property $\forall inc$: *Incident inc.Resolved* $\Rightarrow \exists a$: *Ambulance Intervention(a,inc)*.

Furthermore, our approach produced finer sub-obstacles depending on the granularity of the provided domain properties. This case is exemplified in Section V-C where the learning tool produced the obstacle *ResourceNotUsedOnPatient* which is a refinement of the obstacle *PatientNotTreatedAtIncidentLocation*.

We tested our approach using two ILP systems, XHAIL [25] and TAL [6]. These mainly differ in directions in which they explore the generalisation search space and in the degree of control they provide to users in defining further syntactic constraints over acceptable generalisations. In our experience, we observed that the approach may compute obstacles that appear too general, e.g. an obstacle to *TrainStopped* being that the driver is not responsive even when the signal is not set to stop. However, further experiments suggest that producing several witnesses to a

goal, including traces in which the target of goals is true (with the condition being false), results in weaker obstacles. In what follows we illustrate these findings where applicable.

B. Obstacles to Bounded Achieve Goals

Consider the goal *Achieve* [*MobilisedAmbulanceIntervention*] stating that a mobilised ambulance shall intervene at the incident within 11 minutes.

Goal *Achieve* [*MobilisedAmbulanceIntervention*]:

$$(\bullet Available \wedge a.Mobilised) \Rightarrow \Diamond_{\leq 11} Intervention(a, inc) \quad (7)$$

The background properties capture the following necessary conditions for the target condition of the above goal to hold.

$$\forall a : Ambulance, inc : Incident$$
$$(\bullet\ a.Available \ \wedge \ a.Mobilised \ \wedge \ \Diamond\ (Intervention(a, inc))) \Rightarrow \bigcirc(a.Mobilised \ \mathsf{W}\ (a.Mobilised \wedge Intervention(a, inc))) \quad (8)$$

$$\forall a : Ambulance, inc : Incident$$
$$(\bullet\ a.Available \ \wedge \ a.Mobilised \ \wedge \ \Diamond\ (Intervention(a, inc))) \Rightarrow \bigcirc(a.InService \ \mathsf{W}\ (a.InService \wedge Intervention(a, inc))) \quad (9)$$

These properties state that if an available ambulance is mobilised and eventually intervenes in an incident, then it must remain mobilised and in service, respectively, until it intervenes. The definitions of the fluents appearing in the goal and background properties are given below.

Resolved[inc:Incident] = ⟨*[inc].resolve, [inc].happens, false*⟩
Intervention[a:Ambulance][inc:Incident]=
⟨*[a][inc].intervene,[a][inc].stop_intervention, false*⟩
Allocated[a:Ambulance].[inc:Incident]=
⟨*[a][inc].allocate,[a][inc].deallocate, false*⟩
Mobilised[a:Ambulance]= ⟨*[a].mobilise, [a].demobilise, false*⟩
Available[a:Ambulance] = ⟨*[a].assign, [a].free, true*⟩
InService[a:Ambulance]= ⟨*[a].onCall, [a].offCall, true*⟩

Model checking an LTS synthesised from those background properties against the goal Achieve [*MobilisedAmbulanceIntervention*] results in the following counterexample.

```
 tick                      Available.a1
 a1.inc1.allocate          Available.a1
 a1.assign                 Available.a1
 tick                      Available.a1
 a1.mobilise               Mobilised.a1
 tick                      Mobilised.a1
 a1.offCall
 a1.demobilise
 tick
 a1.inc1.deallocate
 ...
 tick
Cycle in terminal set:
 tick
 tick
```

The events appearing in the terminal set are the only ones that can occur thereafter. In the above violation trace, the goal antecedent is satisfied by the third *tick* after which the ambulance does not intervene, i.e. the goal's target is not satisfied. A witness is generated by checking the synthesised LTS against the anti-target assertion $(\bullet\ a.Available \wedge a.Mobilised) \Rightarrow \Box_{\leq 11} \neg Intervention(a, inc)$. In this case, the model checker produces the following witness.

```
 tick                   Available.a1
 a1.inc1.allocate       Available.a1
 a1.assign
 tick
 a1.mobilise            Mobilised.a1
 tick                   Mobilised.a1
 ...
 a1.inc1.intervene      Mobilised.a1 && Intervention.a1.inc1
 tick                   Mobilised.a1 && Intervention.a1.inc1
Cycle in terminal set:
 tick
 tick
```

The ambulance *a1* intervenes by the fourth *tick*. Please note that the goal *Achieve* [*MobilisedAmbulanceIntervention*], the background properties including assertions (8) and (9) and the counterexample and witness traces are automatically encoded as a logic program. For instance, the goal Achieve [*MobilisedAmbulanceIntervention*] is encoded as the rule

```
holdsAt_eventually_by_tick(intervention(A,Inc),I2,11,S):-
 holdsAt_tick(available(A),I1,S),
 next_tick_at(I1,I2,S),
 holdsAt_tick(mobilised(A),I2,S),
 not obstructed_always_by_tick(intervention(A,Inc),I2,11,S).
```

The learning task aims to generate conditions explaining why `holdsAt_eventually_by_tick(intervention(a1,inc1),5,11,c)` does not hold in the positive example but is true in the negative one. The learning outcome is guided by the mode declaration provided to the learning system. In this example, the mode declaration is restricted to learn rules with the predicate `obstructed_always_by_tick` in the head. The ILP tool produces a number of possible obstacles for the goal *Achieve* [*MobilisedAmbulanceIntervention*] which include:

Obstacle *MobilisedAmbulanceStopsService*:
$\exists a : Ambulance, inc : Incident$

$$\Diamond(\bullet\ a.Available \wedge a.Mobilised \ \wedge \Box_{\leq 11} \neg\ a.InService) \quad (10)$$

Obstacle *MobilisedAmbulanceDeallocated*:
$\exists a : Ambulance, inc : Incident$

$$\Diamond(\bullet\ a.Available \wedge a.Mobilised \ \wedge \Diamond_{\leq 11} \neg Allocated(a, inc)) \quad (11)$$

Obstacle *AmbulanceMobilisationRetracted*:
$\exists a : Ambulance, inc : Incident$

$$\Diamond(\bullet\ a.Available \wedge a.Mobilised \ \wedge \bigcirc \neg a.Mobilised) \quad (12)$$

A comparison of the above outcome with the obstacles generated manually in [32] revealed that our method is capable of computing obstacles produced in [32] by instantiation of the non-persistence pattern, e.g., the first and last obstacles above cover the obstacles *MobilizedAmbulanceStopsServiceBeforeIntervention* and *AmbulanceMobilizationRetracted*, respectively. It also showed that we were able to generate obstacles not obtained through pattern instantiations but related to real obstacles reported in the inquiry report [23] where ambulances did not intervene in incidents because of incorrect allocation of ambulances.

C. Obstacles to Unbounded Achieve Goals

We now consider the goal *Achieve* [*IncidentResolvedByIntervention*] specified as follows.

Goal *Achieve* [*IncidentResolvedByIntervention*]

$$\forall a : Ambulance, inc : Incident \quad Intervention(a, inc) \Rightarrow \Diamond\ inc.Resolved \quad (13)$$

Among the background properties *BP*, the following provide necessary conditions for the target *inc.Resolved* and for treating a patient at an incident location.

$$\forall p : Patient, inc : Incident \quad inc.Resolved \Rightarrow (Injured(p, inc) \rightarrow TreatedAtLocation(p, inc)) \tag{14}$$

$$\forall p : Patient, inc : Incident \quad inc.Resolved \Rightarrow (Injured(p, inc) \rightarrow p.AdmittedToHospital) \tag{15}$$

$$\forall p : Patients, inc : Incident.\ TreatedAtLocation(p, inc) \Rightarrow (\forall r : Resource \ \ CriticallyNeeds(p, r) \rightarrow UsedOn(r, p)) \tag{16}$$

BP also includes the fluent definitions:

AdmittedToHospital[p:Patient]= $\langle$*[p].admit, [p].discharge, false*$\rangle$
Injured[p:Patient][i:Incident] = $\langle$*[p][i].isInjured,[p][i].isRecovered, false*$\rangle$
TreatedAtLocation[p:Patient][i:Incident]= $\langle$*[p][i].isTreated,[p][i].finishTreatment, false*$\rangle$
CriticallyNeeds[p:Patient][r:Resource]= $\langle$*[p][r].needs,[p][r].use, false*$\rangle$
UsedOn[r:Resource][p:Patient]= $\langle$*[r][p].use,[r][p].not_use, false*$\rangle$

Model checking $L(BP)$ against the goal results in the following counterexample trace.

```
 tick
 p1.inc1.isInjured
 a1.inc1.allocate
 a1.assign
 tick
 a1.mobilise
 tick
 a1.inc1.intervene              Intervention.a1.inc1
 tick                           Intervention.a1.inc1
 a1.inc1.stop_intervention
 a1.inc1.deallocate
 a1.free
 tick
Cycle in terminal set:
 tick
 tick
```

In this violation trace, an ambulance intervenes but the incident is never resolved.The background properties are then verified against the anti-target assertion $Intervention(a, inc) \Rightarrow \Box\neg inc.Resolved$ which yields the following witness trace.

```
 tick
 p1.inc1.isInjured
 p1.r1.needs
 a1.inc1.allocate
 assign.a1
 tick
 a1.mobilise
 tick
 a1.inc1.intervene              Intervention.a1.inc1
 tick                           Intervention.a1.inc1
 r1.p1.use
 p1.inc1.isTreated
 tick                           Intervention.a1.inc1
 admit.p1
 inc1.resolve                   Resolved.inc1
 tick                           Resolved.inc1
Cycle in terminal set:
 tick                           Resolved.inc1
 tick                           Resolved.inc1
```

This witness illustrates an incident resolved eventually after an ambulance's intervention. The background properties including domain properties (14), (15) and (16), the goal *Achieve* [*IncidentResolvedByIntervention*] and the counterexample and witness traces are automatically encoded as a logic program. For example, the goal [*IncidentResolvedByIntervention*] is represented by the following rule.

```
holdsAt_eventually_tick(resolved(Inc),I,S):-
  holdsAt_tick(intervention(A,Inc),I,S),
  not obstructed_always_tick(resolved(Inc),I,S).
```

The predicate `holdsAt_eventually_tick(f,i,s)` means that fluent f is true at some tick in the future after time-point i in trace s unless an obstacle that always prevents it from being true holds. The ILP tool suggests the following possible obstacles.

Obstacle *IncidentNotResolvedByIntervention*
$$\Diamond\exists a : Ambulance, inc : Incident \quad (Intervention(a, inc) \wedge \Box\neg Resolved(inc)) \tag{17}$$

Obstacle *PatientNotAdmittedToHospital*
$$\Diamond\exists a : Ambulance, inc : Incident \quad (Intervention(a, inc) \wedge \exists p : Patient \ \ Injured(p, inc) \wedge \Box\neg p.AdmittedToHospital) \tag{18}$$

Obstacle *PatientNotTreatedAtIncidentLocation*
$$\Diamond\exists a : Ambulance, inc : Incident \quad (Intervention(a, inc) \ \wedge (\exists p : Patient \ \ Injured(p, inc) \wedge \Box\neg TreatedAtLocation(p, inc))) \tag{19}$$

Obstacle *CriticalCareNotGivenToPatient*
$$\Diamond\exists a : Ambulance, inc : Incident \quad (Intervention(a, inc) \ \wedge (\exists p : Patient, r : Resource \ \ Injured(p, inc) \wedge \Box\neg CriticallyNeeds(p, r))) \tag{20}$$

Obstacle *ResourceNotUsedOnPatient*
$$\Diamond\exists a : Ambulance, inc : Incident \quad (Intervention(a, inc) \ \wedge (\exists p : Patient, r : Resource \ \ Injured(p, inc) \wedge \Box\neg UsedOn(r, p))) \tag{21}$$

Obstacle *AmbulanceNotAvailableAfterIntervention*
$$\Diamond\exists a : Ambulance, inc : Incident \quad (Intervention(a, inc) \wedge \Diamond \ \neg a.available) \tag{22}$$

By comparing the output of the learning system with those identified in [32], we find that the first obstacle corresponds to the high-level obstacle *IncidentNotResolvedByIntervention* obtained through manual regression in [32]. The obstacles (18) and (19) are sub-obstacles for *IncidentNotResolvedByIntervention* obtained from the domain properties. The tool also produced obstacles at a finer level such as *ResourceNotUsedOnPatient*. Note that this obstacle suggests that resources may be used on patients who do not need it. This may trigger the addition of a missing domain property that states that a necessary condition for using a resource on a patient is that the patient critically needs it.

VI. Discussion and Related Work

Our applications to medium-sized case studies indicate that the techniques used are scalable. In general, the scalability of the approach is determined by the scalability of the model checker and learning tool used. The scalability of model checker is affected by the number of goals and composed LTSs, one for each domain property, per iteration. Since the approach is incremental, it takes one goal at a time and finds all its obstacles. Hence we do not discuss an upper bound on the number of goals handled. The number of domain properties provided per iteration is dependent on the necessary conditions for the goal's target condition to be

satisfied and is assumed to be small. As for the ILP system, it uses an Answer Set Programming solver [29] which grounds the logic programs before searching for solutions. It scales for finite domains and is comparable to SAT-solvers. The complexity of translating each assertion to an automata or a logic program is exponential in the size of the FLTL formula. As these correspond to individual goals or domain properties for goals' target conditions, they remain small enough to be handled by the tool.

Obstacles were originally introduced as informal goal obstruction scenarios in [24], used in [4] for requirements elaboration. Heuristics for informally identifying possible exceptions and errors in such scenarios are proposed in [28]. Obstacles as goal obstruction preconditions were introduced in [31] where various techniques for obstacle analysis are outlined. Lutz and colleagues discuss a convincing application of obstacle analysis to identify anomaly-handling requirements for a NASA unmanned aerial vehicle [19]. The DDP approach to risk analysis also integrates a lightweight form of obstacle analysis [8]. The integration of hazard analysis in the early stages of the RE process is advocated in [17]. Iterative approaches for defect-driven modification of requirements specifications in the context of system faults and human errors are proposed in [3], [7]. Fault trees and threat trees are discussed as special types of risk trees in [16] and [27], respectively.

The approach presented here is much related to that described in [32] where techniques are proposed for identifying and resolving obstacles to goals. In [32] obstacles are generated from goal negations and domain properties by applying a formal regression calculus or by using obstruction patterns. Though their techniques are sound, the processes are manual; the pattern-based technique is restricted to patterns available in the catalogues. Our approach complements the obstacle identification process by (a) providing automated support for generating obstacles from given goals and domain properties and (b) computing a wider class of obstacles that cannot be identified through pattern-based techniques. However, our approach does not address the various obstacle resolution options studied there.

Our approach builds upon the work presented in [1], [2] where model checking and ILP are used for the elaboration of specifications operationalising goals and the derivation of non-zeno behaviour models, respectively. However the technique here involves several challenges that were not addressed in [1], [2]. The witnesses are here generated without human intervention. The role of counterexamples and witnesses is inverted as counterexamples are considered to be positive examples for learning obstacles while witnesses are considered as negative examples. The goal language is much more expressive as it covers any *Achieve* or *Maintain/Avoid* goal which were not covered in [1].

The approach presented here is also somewhat related to the technique described in [26]. The authors in [26] use a reasoning technique called abduction by refutation to detect a complete set of counterexamples for invariants expected to hold for a given system description. The output of the approach is a set of transition relations that show where an invariant does not hold. Such transitions, however, are not guaranteed to be reachable from the initial state of the description and hence it is assumed that reachability checks are performed separately. Also, the invariant structure is limited to the form $(C \rightarrow T)$. Our approach has several advantages: (a) it guarantees that any detected obstacle is indeed reachable from the initial state, (b) the obstacles are produced in the goal language and (c) this language is richer and more expressive than the one applicable in [26].

Our mechanism for encoding goals to learn obstacles is related to the formalisation of abnormality-relations in logic programming [5]. In the latter, abnormality-relations are used to revise an agent's default assumptions. Similarly, our use of obstruction predicates in the body of goal rules provides the means for revising the engineer's default assumption about when the goal's target holds in the domain.

VII. Conclusion and Future Work

The framework described in this paper is a formal, tool-supported approach for incrementally generating obstacles that may prevent goals from being achieved within a given domain. It uses model checking for generating traces that violate and satisfy the goals from a given system description, and an inductive learning tool for computing obstacles from these traces. This iterative process ends once a domain-complete set of obstacles is generated. We applied the proposed approach to a real safety-critical system, the London Ambulance Service where a domain-complete set of obstacles was computed. Our application has shown an improvement over existing methods through automation and detection of a wider class of obstacles.

We envision a number of extensions to the work presented here. First, we will investigate enriching the framework with incremental techniques for resolving the detected obstacles. For this purpose, we plan to revisit the learning phase of the approach and apply ILP algorithms for theory revision. We further plan to extend the framework to handle goals and obstacles which are associated with probabilities. To do so, the use of tools such as probabilistic model checking and probabilistic ILP will be explored. As part of our future work, we intend to adapt our approach to detect conflicts among goals. One way we envision this being possible is by learning boundary conditions for goal conflict from traces generated through deadlock checks.

Acknowledgment

We are grateful to Bernard Lambeau and the reviewers for their careful inspection and feedback on an earlier version of the paper. This work is financially supported by ERC project PBM - FIMBSE (No. 204853).

REFERENCES

[1] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Learning operational requirements from goal models. In *Proc. of 31st Intl. Conf. on Softw. Eng.*, pp. 265–275, 2009.

[2] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Deriving non-zeno behaviour models from goal models using ILP. *J. Form. Asp. Comput.*, 22(3–4):217–241, 2010.

[3] T. Anderson, R. de Lemos, and A. Saeed. Analysis of safety requirements for process control systems. In *Predictably Dependable Computing Systems*, pp. 42–53, 1995.

[4] A.I. Antón and C. Potts. The use of goals to surface requirements for evolving systems. In *Proc. of Intl. Conf. on Softw. Eng.*, pp. 157–166, 1998.

[5] K.R. Apt and R.N. Bol. Logic programming and negation: A survey. *J. of Log. Prog.*, 19/20:9–71, 1994.

[6] D. Corapi, A. Russo, and E. Lupu. Inductive logic programming as abductive search. In *Tech. Comm. of 26th Intl. Conf. on Log. Prog.*, vol. 7 of *LIPIcs*, pp. 54–63, 2010.

[7] R. de Lemos, A. Saeed, and T. Anderson. Analysis of safety requirements in the context of system faults and human errors. In *Proc. of IEEE Intl. Symp. and Work. on Systems Engineering of Computer Based Systems*, pp. 374–381, 1995.

[8] M.S. Feather and S.L. Cornford. Quantitative risk-based requirements reasoning. *J. Req. Eng.*, 8(4):248–265, 2003.

[9] A. Finkelstein and J. Dowell. A comedy of errors: the london ambulance service case study. In *Proc. of 8th Intl. Work. on Software Specification and Design*, pp. 2–4, 1996.

[10] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of 5th Intl. Conf. on Log. Prog.*, pp. 1070–1080, 1988.

[11] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *Proc. 11th Symp. on Foundations Softw. Eng.*, pp. 257–266, 2003.

[12] R.A. Kowalski and M. Sergot. A logic-based calculus of events. *New generation computing*, 4(1):67–95, 1986.

[13] R. Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. Springer, 1992.

[14] E. Letier. Reasoning about Agents in Goal-Oriented Requirements Engineering. Ph.D. dissertation, UCL, Louvain-la-Neuve, Belgium, 2001.

[15] E. Letier, J. Kramer, J. Magee, and S. Uchitel. Deriving event-based transitions systems from goal-oriented requirements models. *J. Auto. Softw. Eng.*, 15(2):175–206, 2008.

[16] N. Leveson. *Safeware - System Safety and Computers*. Addison-Wesley, 1995.

[17] N. Leveson. An approach to designing safe embedded software. In *Proc. 2nd Intl. Conf. on Embedded Software*, pp. 15–29, 2002.

[18] J.W. Lloyd. *Foundations of logic programming*. Springer, 1984.

[19] R. Lutz, A. Patterson-Hine, S. Nelson, C.R. Frost, D. Tal, and R. Harris. Using obstacle analysis to identify contingency requirements on an unpiloted aerial vehicle. *J. Req. Eng.*, 12(1):41–54, 2006.

[20] J. Magee and J. Kramer. *Concurrency : State Models and Java Programs*. John Wiley and Sons, 1999.

[21] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 1992.

[22] S.H. Muggleton and L. De Raedt. Inductive Logic Programming: Theory and Methods. *J. of Log. Prog.*, 19/20(1):629–679, 1994.

[23] Report on the Inquiry Into the London Ambulance Service. The communications directorate, south west thames regional authority, 1993.

[24] C. Potts. Using schematic scenarios to understand user needs. In *Proc. of 1st conf. on Designing interactive systems: processes, practices, methods & techniques*, pp. 247–256, 1995.

[25] O. Ray. Nonmonotonic abductive inductive learning. *J. of Applied Log.*, 7(3):329–340, 2009.

[26] A. Russo, R. Miller, B. Nuseibeh, and J. Kramer. An abductive approach for analysing event-based requirements specifications. In *Proc. of 18th Intl. Conf. Log. Prog.*, pp. 69–105. 2002.

[27] B. Schneier. *Secrets and Lies: Digital Security in a Networked World*. Wiley, 2000.

[28] A.G. Sutcliffe, N.A. Maiden, S. Minocha, and D. Manuel. Supporting scenario-based requirements engineering. *IEEE Trans. on Softw. Eng.*, 24(12):1072–1088, 1998.

[29] T. Syrjänen and I. Niemelä. The Smodels System. *Log. Prog. and Nonmotonic Reasoning*, vol. 2173 of *LNCS*, 2001.

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

[31] A. van Lamsweerde and E. Letier. Integrating obstacles in goal-driven requirements engineering. In *Proc. of 20th Intl. Conf. on Softw. Eng.*, pp. 53 –62, 1998.

[32] A. van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirement engineering. *IEEE Trans. on Softw. Eng.*, 26(10):978–1005, 2000.