Elaborating Requirements Using Model Checking and Inductive Learning

Dalal Alrajeh, Jeff Kramer, *Member*, *IEEE*, Alessandra Russo, *Member*, *IEEE*, and Sebastian Uchitel, *Member*, *IEEE*

Abstract—The process of Requirements Engineering (RE) includes many activities, from goal elicitation to requirements specification. The aim is to develop an operational requirements specification that is guaranteed to satisfy the goals. In this paper, we propose a formal, systematic approach for generating a set of operational requirements that are complete with respect to given goals. We show how the integration of model checking and inductive learning can be effectively used to do this. The model checking formally verifies the satisfaction of the goals and produces counterexamples when incompleteness in the operational requirements is detected. The inductive learning process then computes operational requirements from the counterexamples and user-provided positive examples. These learned operational requirements are guaranteed to eliminate the counterexamples and be consistent with the goals. This process is performed iteratively until no goal violation is detected. The proposed framework is a rigorous, tool-supported requirements elaboration technique which is formally guided by the engineer's knowledge of the domain and the envisioned system.

Index Terms—Requirements elaboration, goal operationalization, behavior model refinement, model checking, inductive learning

1 Introduction

1.1 Motivation

Requirements Engineering (RE) is an integral part of the software development process. It is concerned with the elicitation of stake-holders goals, the elaboration of these goals into requirements on the software behavior, and the assignment of responsibilities for these requirements to agents [13]. Inadequacy in the execution of any of these RE subtasks inevitably leads to development problems which are often difficult and costly to repair. This has led researchers to seek rigorous and automated methods to support the fulfillment of these tasks.

Goal-oriented approaches have been shown to be particularly effective for formal analysis and automated validation [6], [51], [46], [39]. *Goals* are objectives the system is intended to achieve through the cooperation of agents in the envisioned software and its environment [30]. "Reverse thrust enabled when a plane is moving on the runway" is an example of a goal for a Flight Control System (FCS). Agents, such as autopilot and wheels-sensors, are active components in the software and environment whose behavior can be constrained to ensure the satisfaction of the goals. A *requirement* is a goal which has been assigned to an agent in the software being developed, while an *expectation* is a goal which has been assigned to an agent in the environment. An *operational requirement* captures the conditions under which a system component *may* or *must* perform an operation to achieve the goals (e.g., a required precondition for disabling the reverse thrust is that the wheels's pulse is on).

One of the difficulties in developing a system specification is the elaboration of operational requirements that guarantee the satisfaction of the goals. This is essentially a manual task and hence is costly and error prone. Very little systematic, rigorous support exists; among the exceptions are the informal techniques explained in [6], [46] and the formal approaches in [31]. However, such approaches lack desirable characteristics such as automation and/or generality, making them less accessible to practitioners.

This paper addresses the problem of how formal, tool-supported methods can be effectively used to identify and resolve the incompleteness of a given set of operational requirements with respect to a set of predefined goals. We propose an integrated use of model checking to detect incompleteness in a given partial operational requirements specification, and inductive learning to resolve the incompleteness. The model checking formally verifies the satisfaction of the goals and produces a counterexample when an incompleteness is detected in the partial operational requirements specification. We focus on two incompleteness problems: 1) incompleteness of the operational requirements with respect to goals expressed as safety properties, and 2) incompleteness of the operational requirements with respect to goals expressed as a particular form of liveness properties called progress properties. In this paper, we assume correctness of the initial specification, and do not address the case in which model checker detects faults caused by an erroneous specification.

The particular inductive learning technique we use, called Inductive Logic Programming (ILP) [38], automatically

- *D. Alrajeh, J. Kramer, and A. Russo are with the Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2AZ, United Kingdom.*
  E-mail: {dalal.alrajeh, j.kramer, a.russo}@imperial.ac.uk.
- *S. Uchitel is with the Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2AZ, United Kingdom and with the Departamento de Computacion, FCEyN, University of Buenos Aires, Buenos Aires, Argentina.*
  E-mail: s.uchitel@imperial.ac.uk, suchitel@dc.uba.ar.

Manuscript received 5 Oct. 2010; revised 29 Aug. 2011; accepted 25 Apr. 2012; published online 6 June 2012.
Recommended for acceptance by P. Inverardi.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2010-10-0297.
Digital Object Identifier no. 10.1109/TSE.2012.41.

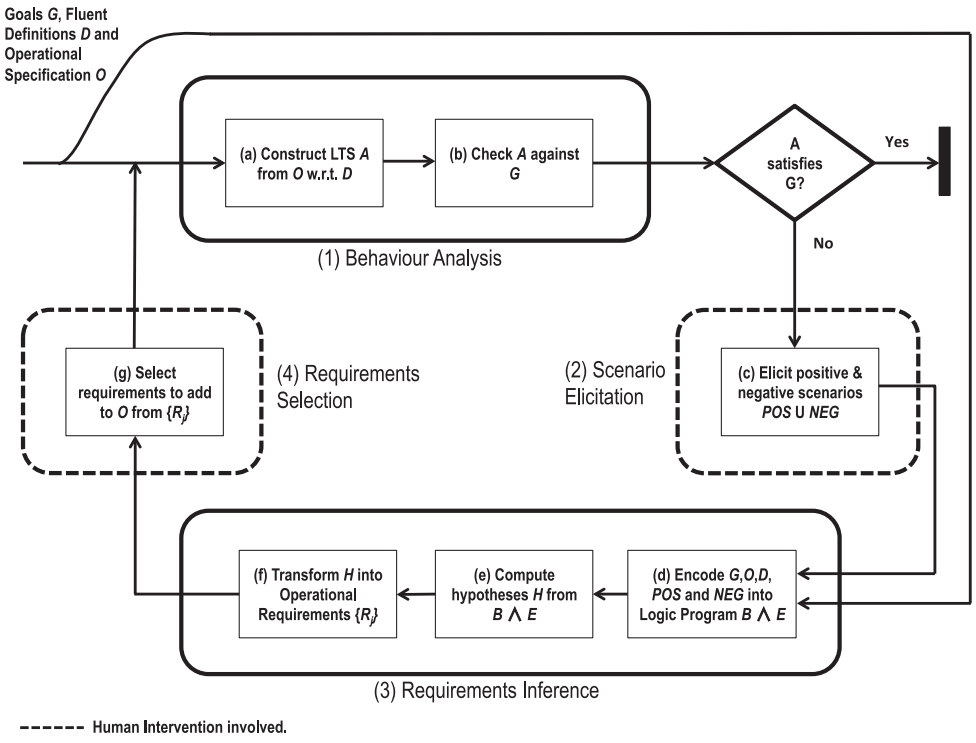

Fig. 1. Overview of the proposed approach.

generates missing operational requirements that are needed to satisfy the existing goals. One of the main advantages of using ILP is that it makes use of any existing knowledge during the inference process. This has the benefit of automatically ensuring consistency of the learned requirements with respect to the goals and the existing specification. Witnesses and counterexamples are used to suggest information about positive and negative scenarios (or examples) that the inductive learner uses to compute the operational requirements.

The proposed framework is defined as an iterative process consisting of four phases. In the *behavior analysis* phase, the existing (partial) specification of operational requirements is verified against the given goals using a model checker. If the verification is unsuccessful, the counterexample generated is used in the *scenario elicitation* phase, where an engineer extracts a set of positive and negative scenarios. In the *requirements inference* phase, goals, existing operational requirements, and scenarios are used by the ILP learning system to compute a set of new operational requirements that covers all positive scenarios and eliminates the negative ones. If the learner produces a number of alternative requirements, the engineer then selects the operational requirements to be added from the list proposed in the *requirements selection* phase. The cycle is then repeated until no goal violation is detected. The steps are illustrated in Fig. 1. The person shown in the figure indicates the points where an engineer's intervention is required.

The automated support provided by this framework is intended to reduce the manual intervention of engineers and hence avoids the introduction of errors, which are frequent in a fully manual approach, as the learner guarantees to produce correct solutions with respect to the goals and the existing specification.

1.2 Contribution and Outline

The main contribution of this paper is a systematic, rigorous, and tool-supported framework for the elaboration of operational requirements from safety and progress goals, using an integration of model checking and inductive learning.

The approach proposed here also provides more specific contributions. We provide a sound technique for transforming a specification expressed in asynchronous Fluent Linear Temporal Logic (FLTL) into a logic program, which in turn lends itself to the use of reasoning methods based on logic programming. We also prove the correctness of the ILP solution with respect to the given scenarios.

More generally, we provide automated support for the incremental development of behavior specifications that satisfy progress and safety properties. We show how the approach can be adapted and applied to other goal operationalization problems such as operationalization of goals that must be achieved within a bounded time interval. This suggests that the approach is potentially applicable to other event-based problems where a partial model exists and needs to be refined.

The rest of this paper is organized as follows: Section 2 describes background work on goals, operational requirements, and behavior modeling. Section 3 defines the problem this paper addresses. Section 4 describes in detail the different phases of the framework. Section 5 discusses the termination of the approach. Section 6 validates the proposed requirements elaboration process using the London Ambulance Service case study. A discussion and account of related work are given in Sections 7 and 8, respectively. Section 9 summarizes our contributions and discusses future work.

2 Background

In this section, we define the notions of goals, operational specifications and behavior modeling used in the approach. We also introduce some preliminaries on Event Calculus (EC) logic programs and inductive learning. To illustrate the concepts, we refer to excerpts of the flight control system example introduced in [22].

2.1 Goal and Operational Requirements Specifications

Goals are the objectives to be achieved by a system [31], [6]. They are often expressed declaratively in terms of state-based properties that should be satisfied in the envisioned system over some predefined temporal structure (e.g., "reverse thrust enabled when the plane is moving on the runway").

Operational requirements, on the other hand, express constraints on the operations to be performed by the system. They take the form of domain and required conditions. A *domain condition* captures the basic state transitions defined by the application of an operation in the domain. It is specified as a pair containing a domain precondition (*DomPre*) and domain postcondition (*DomPost*). *Required conditions*, on the other hand, capture strengthened conditions on the software-controlled operations that contribute to the satisfaction of the goals. They are expressed in the form of *required pre*, *trigger-*, and *postconditions*. *Required preconditions* (*ReqPre*) are conditions that capture a *permission* to perform an operation. *Required trigger-conditions* (*ReqTrig*) are conditions that capture an *obligation* to perform an operation. *Required postconditions* (*ReqPost*) specify the conditions that *must hold after* the execution of an operation.

A set of operational requirements is said to be *complete with respect to a goal* if satisfying the required conditions in the set guarantees the satisfaction of the goal [31]. Otherwise, it is said to be *partial*.

Goals and operational requirements can be expressed in some form of temporal logic. Such formalisms support the use of existing automated behavior model synthesis and verification techniques such as model checking. We use an asynchronous form of FLTL [21] to specify goals and operational requirements, and Labeled Transition Systems (LTS) to model the corresponding system behavior. The reason for using FLTL as the representation language is twofold. First, FLTL is a language designed for reasoning about event- and state-based properties together in event-based models. Second, it is supported by existing model checkers (e.g., LTSA [34]). However, this particular choice of formalism is not essential to our approach. The approach can be easily adapted to other formalisms such as standard linear temporal logic.

2.1.1 Fluent Linear Temporal Logic

FLTL is a special form of linear temporal logic that makes use of fluent. A fluent is a time-varying property of the system. It is defined by a pair of disjoint sets of event labels, referred to as the *initiating* (I_f) and *terminating* (T_f) sets of events, and an initial truth value *true* or *false*. Event labels in an initiating (respectively, terminating) set are those events that, when executed, cause the fluent to become true (respectively, false). For instance, assuming a fluent *WheelsTurning* is initially *false*, meaning that wheels of the plane are initially not turning, the definition for this fluent is specified as

$$WheelsTurning = \langle \{spinWheels\}, \{stopWheels\} \rangle \\ \{initially\}\ false,$$

stating that the event *spinWheels* causes the fluent *WheelsTurning* to become true, and the event *stopWheels* causes the fluent *WheelsTurning* to become false. In other words, *WheelsTurning* is a domain postcondition for the event *spinWheels*. A fluent can be state based, such as the one above, or event based. An event-based fluent f_e signals the occurrence of the event e. Its initiating set is the singleton set $\{e\}$ and its terminating set is $L - \{e\}$, where L is the universal set of event labels. We use the convention that a fluent is assumed to be initially false unless explicitly stated to be true.

Given a set of fluents F, an FLTL formula can be constructed using the classical connectives, $\neg$, $\wedge$, and $\rightarrow$, and the temporal operators $\bigcirc$ (next), meaning in the next state, $\Box$ (always), meaning always in the future, $\diamond$ (eventually), meaning some time in the future and $\bigcup$ (strong until), meaning always in the future until. In addition, bounded temporal operators [25] can be used such as $\diamond_{\leq d}$, meaning some time in the future within the next d time units, where d is a nonnegative integer. Other classical and temporal operators can be defined as combinations of the above operators (e.g., $\phi \vee \psi \equiv \neg(\neg\phi \wedge \neg\psi)$ and $\phi \mathrm{W} \psi \equiv (\Box\phi) \vee (\phi \mathrm{U} \psi)$). A well-formed FLTL formula is formulated in the standard way [35].

FLTL, as defined in [21], is said to be an asynchronous temporal logic. This means that its properties refer to sequences of system states observed after each occurrence of an event. Goals, however, are often defined over a discrete linear structure of time. When expressing goals and operational requirements in asynchronous FLTL, a *tick* fluent is introduced to mark the beginning of each time unit.

A goal can be expressed formally in four different modes: achieve, cease, maintain, and avoid [12]. The schemata for goals in the immediate achieve and cease modes are expressed respectively as follows:

$$\Box(tick \rightarrow (P \rightarrow \bigcirc(\neg tick\ \mathrm{W}\ (tick \wedge Q)))),$$
$$\Box(tick \rightarrow (P \rightarrow \bigcirc(\neg tick\ \mathrm{W}\ (tick \wedge \neg Q)))),$$

where P and Q can be any well-formed FLTL expression containing state-based fluents only. For instance, assume an FLTL language includes the state-based fluents *ThrustEnabled*, *PulseOn*, and *MovingOnRunway*, meaning "the reverse thrust is enabled," "the wheels' pulse is on," and "the plane is moving on the runway," respectively, and the event-based fluents *switchPulseOn* meaning "switch the wheels' pulse on," *disableThrust* meaning "disable the reverse thrust," and *enableThrust* which means "enable the reverse thrust." The goal "reverse thrust enabled when the plane is moving on the runway" can be formally expressed in FLTL as

$$\Box(tick \rightarrow (MovingOnRunway \\ \rightarrow \bigcirc(\neg tick\ \mathrm{W}\ (tick \wedge ThrustEnabled)))),$$

which states that it should always be the case that whenever the plane is moving on the runway, the reverse thrust should be enabled by the next time point. Hence, goals refer to what needs to be true or false at tick occurrences.

Operational requirements, on the other hand, express constraints on events occurring between the tick transitions. These are expressed in one of the following generic forms:

$$\Box(tick \rightarrow (\neg ReqPre \rightarrow \bigcirc(\neg OP \text{ W } tick))),$$
$$\Box(tick \rightarrow ((ReqTrig \wedge DomPre) \rightarrow \bigcirc(\neg tick \text{ W } OP))),$$
$$\Box(OP \rightarrow (\neg tick \text{ W } (tick \wedge ReqPost))),$$

where OP is an event-based fluent and $ReqPre$, $ReqTrig$, $DomPre$, and $ReqPost$ are FLTL expressions containing state-based fluents only.

For example, the domain precondition "the wheels' pulse switch may not be pressed if the wheels' pulse is already on" is formalized as

$$\Box(tick \rightarrow (PulseOn \rightarrow \bigcirc(\neg switchPulseOn \text{ W } tick)))$$

The required precondition "the wheels' pulse is off" for the operation *disableThrust* is specified in asynchronous FLTL as

$$\Box(tick \rightarrow (\neg PulseOn \rightarrow \bigcirc(\neg disableThrust \text{ W } tick))),$$

while the trigger-conditions "the wheels' pulse is on" for the operation *enableThrust* is expressed as

$$\Box(tick \rightarrow (PulseOn \rightarrow \bigcirc(\neg tick \text{ W } enableThrust))).$$

Goals and operational requirements can be formally expressed in a more compact way using other languages or semantics. In synchronous LTL, for instance, the above goal can be expressed without explicitly referring to ticks of the clock as $\Box(MovingOnRunway \rightarrow \bigcirc ThrustEnabled)$. It is understood implicitly that every state at which the assertion is evaluated is the state immediately proceeding the occurrence of a tick. Expressions of this form can be automatically transformed into asynchronous FLTL specification using the technique described in [29].

2.1.2 Labeled Transition Systems

A Labeled Transition Systems [23] is a behavior model that can be used to represent a system as a set of concurrent components (or agents). Each component is defined as a set of states and transitions between the states. Transitions are Labeled with events denoting the interaction that the component has with itself, the environment and other components. The definition below is adapted from [20].

Definition 1 (Labeled Transition System). *A Labeled transition system A is a tuple $(S, L, \mathcal{R}, s_0)$ where S is a finite, nonempty set of states including the error state, designated as λ, L is a finite nonempty set of event labels, called the alphabet, s_0 is an element of S, called the initial state, and $\mathcal{R} \subseteq S - \{\lambda\} \times L \times S$ is a nonempty set of transitions.*

A trace σ in A is a (possibly infinite) sequence of transitions (denoted $s_0\mathcal{R}_{e_1}s_1\mathcal{R}_{e_2}s_2\ldots$) such that for each $i \geq 0$ there is a transition $(s_i, e_{i+1}, s_{i+1}) \in \mathcal{R}$. Note that we also sometimes represent traces as sequences of event labels $(e_1, e_2, \ldots)$.

A scenario σ^* is a finite sequence of labels of the form $\langle e_1, e_2, \ldots, e_m\rangle$, where $e_i \in L$ for $1 \leq i \leq m$. It is said to be accepted by an LTS A if there is a trace $s_0\mathcal{R}_{e_1}s_1 \mathcal{R}_{e_2}s_2\ldots s_{m-1}\mathcal{R}_{e_m}s_m$ in A such that $(s_i, e_{i+1}, s_{i+1}) \in \mathcal{R}$ for all $0 \leq i < m$. We call a trace σ that accepts a scenario σ^* an *accepting trace of* σ^*.

LTSs can be generated automatically from declarative expressions specified in asynchronous FLTL. The semantics of an FLTL formula is defined with respect to traces in an LTS and a valuation function that returns the set of fluents that are true at a given position in a trace σ in A according to their fluent definition.

Given a set of fluent definitions D, a fluent f is evaluated as true at position i in a trace σ with respect to D, denoted $\sigma, i \models_D f$, if and only if either of the following conditions hold:

- f is defined ***initially*** *true* in D and $\forall j \in \mathcal{N}$ $((0 < j \leq i) \rightarrow e_j \notin T_f)$;
- $(\exists j \in \mathcal{N}.(j \leq i) \wedge (e_j \in I_f)) \wedge (\forall k \in \mathcal{N}.((j < k \leq i) \rightarrow e_k \notin T_f))$.

In other words, a fluent f is said to be true at position i if it was initially true or an initiating event for f has occurred, and no terminating event has occurred since. The semantics of Boolean operators is defined with respect to the fluent definitions over each position in a trace in a standard way. For a given set of fluent definitions D, the semantics of the temporal operators is defined inductively as follows:

- $\sigma, i \models_D \bigcirc\phi$ iff $\sigma, i+1 \models_D \phi$.
- $\sigma, i \models_D \Box\phi$ iff $\forall j \geq i.\ \sigma, j \models_D \phi$.
- $\sigma, i \models_D \Diamond\phi$ iff $\exists j \geq i.\ \sigma, j \models_D \phi$.
- $\sigma, i \models_D \phi \text{ U } \psi$ iff $\exists j \geq i.\ \ \sigma, j \models_D \psi$ and $\forall i \leq k < j.\ \sigma, k \models_D \phi$.

An FLTL formula ϕ is said to be satisfied in a trace σ with respect to D if it is satisfied at position 0. The satisfaction of FLTL expressions in a given LTS is defined below.

Definition 2 (FLTL Satisfaction and Entailment in LTSs). *Let ϕ be a single FLTL formula, Γ a set of FLTL formulae, also called a theory, and D a set of fluent definitions. The formula ϕ is said to be satisfied in an LTS A with respect to D if it is satisfied in every trace in A with respect to D. The set of formulae Γ is said to be satisfied in an LTS A with respect to D if every formula in Γ is satisfied in A with respect to D. Γ is said to be consistent if there is an LTS that satisfies it. A formula ϕ is said to be entailed by Γ with respect to D, denoted as $\Gamma \models_D \phi$, if and only if every LTS that satisfies Γ also satisfies the formula ϕ.*

Given a consistent FLTL theory, the Labeled Transition System Analyzer (LTSA) tool [34] can be used to automatically generate a least constrained LTS that satisfies it, i.e., maximal with respect to the traces it includes. This is done using an adaptation [29] of a "temporal logic to automata" algorithm used in model checking FLTL formulae [21].

2.2 Inductive Learning

We describe here the basic notions of an Event Calculus logic program, which is a formalism understood by the learning system. We also give a brief introduction to the inductive learning algorithm deployed in the approach.

TABLE 1
Definition of EC Basic Predicates

EC predicate	Definition
happens(e,p,s)	event e occurs at position p in scenario s
initiates(e,f,p,s)	the occurrence of event e at position p in scenario s causes fluent f to be true at the next time position
terminates(e,f,p,s)	the occurrence of event e at position p in scenario s causes fluent f to be false at the next time position
holdsAt(f,p,s)	fluent f is true at position p in scenario s
impossible(e,p,s)	the event e is impossible at position p in scenario s
attempt(e,p,s)	there is an attempt to execute event e at position p in scenario s

Notation and terminology used for logic programs are given in Appendix A. Readers may skip this section without loss of continuity.

2.2.1 Event Calculus Logic Programs

The Event Calculus is a formalism, first introduced by Kowalski and Sergot for reasoning about events and their effects over time [24]. A number of extensions have been introduced since. We use here the variation presented in [1].

The EC language includes the basic predicates *happens*, *initiates*, *terminates*, *holdsAt*, *impossible*, and *attempt*. Their definitions are given in Table 1.

Furthermore, additional predicates are defined to capture the notion of synchronous satisfaction in terms of asynchronous semantics [2]. For instance, the predicates *holdsAt_Tick(f,p,s)* (respectively, *not_holdsAt_Tick(f,p,s)*) means that a fluent f holds (respectively, does not hold) when a tick occurs at position p in scenario s. An atom *holdsAt_PrevTick(f,p,s)* (respectively, *not_holdsAt_PrevTick(f,p,s)*) means a fluent f holds (respectively, does not hold) at the previous tick in scenario s if it holds (respectively, does not hold) at the last preceding position where a *tick* occurred. The predicate *nextTickAt(p2,p1,s)* is an auxiliary predicate that says that *p2* is the next position where a tick occurs after position *p1* in scenario s. The predicate *occursSince_PrevTick(e,p,s)* states that event e occurs in a scenario s at some position between the current position p and the last preceding tick.

Domain-independent axioms formalize the law of inertia: A fluent that is initiated continues to hold until a terminating event occurs. They also define when an event may happen. For the definition of these axioms and the auxiliary predicates, the reader is referred to Appendix B.

Domain-dependent axioms define the predicates *initiates*, *terminates*, *impossible*, *attempt*, and *happens* used to represent the particular problem in hand. Examples of these are given in later sections.

2.2.2 Inductive Logic Programming

In general, an inductive learning task is defined as the computation of an hypothesis H that is consistent with a given background knowledge B and integrity constraints *IC*, and that together with B explains a given set E of examples [43], [37]:

$$B \wedge H \wedge \mathrm{IC} \not\models false, \tag{1}$$

$$B \wedge H \models E. \tag{2}$$

For instance, consider an EC program where the background knowledge includes the following information about the FCS example:

$$\begin{aligned} \mathrm{B} = \{&\mathrm{attempt(enableThrust, 1, s1)},\\ &\mathrm{holdsAt(thrustEnabled, 1, s1)},\\ &\mathrm{attempt(enableThrust, 0, s2)},\\ &\mathrm{not\ holdsAt(thrustEnabled, 0, s2)},\\ &\mathrm{happens(E, T, S) : - attempt(E, T, S)},\\ &\mathrm{not\ impossible(E, T, S)}\}. \end{aligned} \tag{3}$$

In this example, the background assumes at position 1 in scenario *s1* that the *thrustEnabled* fluent is true and there is an attempt to enable the reverse thrust at the same position. In scenario *s2*, *thrustEnabled* is not true at position 0 and an attempt is made to enable the reverse thrust then. The last rule states that an event E happens at position T in scenario S if it is attempted and it is not impossible to occur. From the above we can derive the facts *happens(enableThrust,1,s1)* and *happens(enableThrust,0,s2)* as neither of the events' occurrences is defined as impossible in the program. Now assume the following facts are observed:

$$\begin{aligned} \mathrm{E} = \{&\mathrm{not\ happens(enableThrust, 1, s1)},\\ &\mathrm{happens(enableThrust, 0, s2)}\}. \end{aligned} \tag{4}$$

Notice that the current background knowledge does not entail the examples E in (4) as the fact *happens(enableThrust,1,s1)* is provable from the current program. This means that to explain the observations in E, B needs to be extended with a set of rules H such that in a model of the program $B \wedge H$, the fact *happens(enableThrust,1,s1)* is no longer true.

In ILP systems, such as XHAIL [41], [42] and Progol [37], the computation of H is constrained by a mode declaration and the compression heuristic. The mode declaration specifies the syntactic form of the rules that can be learned. It includes a head declaration of the form $modeh(r, s)$ and body declarations of the form $modeb(r, s)$, where r is an integer, called the *recall*, and s is a ground literal called the *scheme*, possibly containing so-called placemarker terms of the form $+t$, $-t$, and $\#t$. These respectively denote input variables, output variables, and constants of type t. The recall is used to bound the number of atoms a mode declaration can contribute to an hypothesis. Where this is not important, an arbitrary recall is denoted by an asterisk $*$. The compression heuristic, on the other hand, favors the inference of hypotheses containing the fewest number of literals, as motivated by the scientific principle of Occam's razor.

It is also common to restrict the search space to those rules that satisfy some *integrity constraints IC*s. These are rules with an empty head. For instance, the following IC states that an impossible event cannot happen:

$$\mathrm{IC} = \{:- \mathrm{happens(E, T, S), impossible(E, T, S).}\} \tag{5}$$

TABLE 2
Fluent Definitions for the FCS Example

$ThrustEnabled = \langle enableThrust, disableThrust \rangle$
$WheelsTurning = \langle turnWheels, stopWheels \rangle$
$PulseOn = \langle switchPulseOn, switchPulseOff \rangle$
$MovingOnRunway = \langle landPlane, \{takeOff, stopPlane\} \rangle$

If the above rule is added to the program, then only those Hs in the search space that satisfy the constraint are computed.

Now, assuming the mode declaration is defined to learn rules with the predicate *impossible* in the head of the rule and the literals *holdsAt* and *not holdsAt* in the body, an ILP solution for the examples E given in (4) may be

$$\begin{aligned} \mathrm{H} = \{&\mathrm{impossible}(\mathrm{enableThrust}, \mathrm{T}, \mathrm{S}) \\ &:- \mathrm{holdsAt}(\mathrm{thrustEnabled}, \mathrm{T}, \mathrm{S}) \end{aligned} \tag{6}$$

which means that the reverse thrust cannot be enabled if it is already enabled. Adding the above rule to B ensures that the fact *happens(enableThrust,1,s1)* is no longer derivable from the new program $B \cup H$, and that $B \cup H \models E$.

3 Problem Definition

The problem in this paper addresses is how to systematically elaborate a set of operational requirements that together with an existing partial specification ensures the satisfaction of given goals.

Consider, for example, the goal [*ReverseThrustEnabledWhenMovingOnRunway*] formalized in asynchronous FLTL as the following safety property:

$$\begin{aligned} \Box(tick \rightarrow (&MovingOnRunway \\ &\rightarrow \bigcirc(\neg tick \,\mathrm{W}\, (tick \wedge ThrustEnabled)))). \end{aligned} \tag{7}$$

This states that if at the tick of a system clock, the plane is moving on the runway, then by the next tick the reverse thrust should be enabled.

Consider the definitions for the fluents *ThrustEnabled*, *WheelsTurning*, *PulseOn*, and *MovingOnRunway* given in Table 2.

Suppose that the given partial operational specification is as formalized in Table 3. It expresses domain constraints on the operations *enableThrust* and *disableThrust* stating that the reverse thrust cannot be enabled (respectively, disabled) if it is already enabled (respectively, disabled). Similarly, once the fluents *WheelsTuring*, *PulseOn*, or *MovingOnRunway* have been initiated (respectivley, terminated), they cannot be initiated (respectively, terminated) again before the next time point. The last two expressions are expectations on the domain that express necessary conditions for the wheels to be turning and the plane to be moving on the runway. Note that, initially, this partial specification includes no required conditions.

The LTS generated from the above specification using the LTSA model checker contains 65 states and 273 transitions. Due to its large size, we show a minimized LTS where only the events *tick*, *landPlane*, *enableThrust*, and *disableThrust* are observable, i.e., those event relevant to the satisfaction of the goal, and all other events are hidden or represented by the label *tau*. The problem with this LTS is that it allows behaviors that violate the goal [*ReverseThrust EnabledWhenMovingOnRunway*]. For instance, inspection of the full LTS shows that it accepts the scenario $\langle$*tick, landPlane, turnWheels, switchPulseOn, tick, tick*$\rangle$, where the plane lands on the ground, the wheels start turning and the wheels' pulse is switched on but the reverse thrust is not enabled subsequently. The violation occurs when the last tick occurs and the reverse thrust is not enabled.

TABLE 3
A Partial Operational Specification for the FCS in FLTL

$\Box(tick \rightarrow (ThrustEnabled \rightarrow \bigcirc(\neg enableThrust \,\mathrm{W}\, tick)))$
$\Box(tick \rightarrow (\neg ThrustEnabled \rightarrow \bigcirc(\neg disableThrust \,\mathrm{W}\, tick)))$
$\Box(tick \rightarrow (WheelsTurning \rightarrow \bigcirc(\neg turnWheels \,\mathrm{W}\, tick)))$
$\Box(tick \rightarrow (\neg WheelsTurning \rightarrow \bigcirc(\neg stopWheels \,\mathrm{W}\, tick)))$
$\Box(tick \rightarrow (PulseOn \rightarrow \bigcirc(\neg switchPulseOn \,\mathrm{W}\, tick)))$
$\Box(tick \rightarrow (\neg PulseOn \rightarrow \bigcirc(\neg switchPulseOff \,\mathrm{W}\, tick)))$

$\Box(tick \rightarrow (MovingOnRunway \rightarrow WheelsTurning))$
$\Box(tick \rightarrow (WheelsTurning \rightarrow PulseOn))$

This counterexample indicates that the current operational requirements are incomplete with respect to the goals. The inclusion of the required trigger-condition for the event *enableThrust*:

$$\begin{aligned} \Box(tick \rightarrow ((&PulseOn \wedge \neg \mathrm{ThrustEnabled}) \\ &\rightarrow \bigcirc(\neg tick \,\mathrm{W}\, enableThrust))), \end{aligned} \tag{8}$$

which ensures that the reverse thrust *must* be enabled when the wheels pulse is activated, would eliminate this violation. We will show in the following section how inductive learning can be used to compute such missing requirements.

The above example shows an instance of an incompleteness with respect to a goal expressed as a safety property. Incompleteness of an operational requirements can also be with respect to a goal expressed as a progress property. For instance, consider the LTS shown in Fig. 2, which is generated from the operational specification in Table 3. It includes the trace (*tick, enableThrust, enableThrust, enableThrust,...*), in which after the first *tick* event, no ticks occur. This is an example of a behavior that vacuously satisfies the goal [*ReverseThrustEnabledWhenMovingOnRunway*] but violates the expectation that time progresses, commonly referred to as the *Time Progress* (TP) property. This is a common property check in discrete-time LTSs and is formalized as $\Box \diamond tick$. Though such violations cannot occur in the real world, violations to this progress property indicate a problem in the specification and may give rise to additional operational requirements. In our example, this trace exists because the safety goals require certain fluents

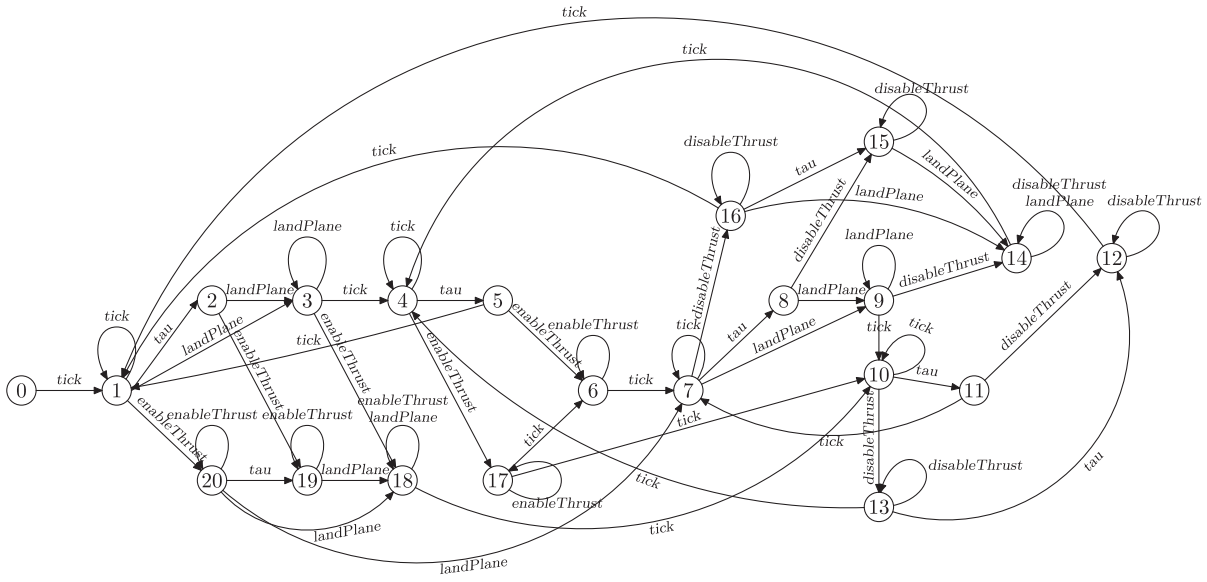

Fig. 2. A minimized LTS that violates the goal [*ReverseThrustEnabledWhenMovingOnRunway*].

to hold at fixed time points, but do not sufficiently constrain the sequence of interleaved events that ensure that these properties hold. These violations thus give rise to additional operational requirements or expectations missing from the specification in order to satisfy the goals. For instance, by including an additional required precondition ($\neg PulseOn$) for the event *enableThrust* expressed as

$$\begin{aligned}\Box(tick \rightarrow (\neg PulseOn \\ \rightarrow \bigcirc(\neg enableThrust \ \mathrm{W} \ tick))),\end{aligned} \tag{9}$$

the behavior described above would be eliminated.

In summary, systems implemented from partial operational requirements specifications may exhibit behavior that violates goals expressed as either safety or progress properties. Existing model checking techniques and tools allow the automatic detection of behaviors that violate such properties. However, completing the specification with respect to these properties remains an open problem with no rigorous, automated support.

In the following section, we show how an existing partial operational specifications can be completed by computing operational requirements such as those given in 8 and 9 through an integration of goal-based model checking and inductive learning.

4 Learning Operational Requirements

The input to the requirements elaboration process is a set of goals, fluent definitions, and a partial set of operational specifications expressed in asynchronous FLTL that are consistent with the goals. We consider that a partial operational specification consists of domain preconditions, required conditions (if any), and any domain specific expectations. The task is to complete this partial specification with new operational requirements to guarantee the given goals are satisfied.

The proposed framework, as depicted in Fig. 1, is defined as an incremental process composed of four phases:

1. ***Behavior analysis phase.*** The model checker is used first to construct an LTS A from a partial operational specification O with respect to fluent definitions D. It is then used to verify the LTS A against the goals G. The result of the analysis is either a notification that no violation traces have been detected, in which case the process successfully terminates, or that a counterexample has been detected, in which case it is displayed.
2. ***Scenario elicitation phase.*** If a counterexample is found, an engineer elicits a set of positive and negative scenarios ($POS \cup NEG$) from the counterexample and the LTS model. A negative scenario represents a violation while a positive scenario exemplifies an instance of some desirable behavior.
3. ***Requirements inference phase.*** The goals, partial operational specification, fluent definitions, and scenarios are translated into a logic program used by an ILP system. The output of this phase is sets of alternative required conditions $\{\mathrm{Req}_j\}$, each of which permits all the positive scenarios, forbids all the negative ones, and is consistent with the goals and the existing operational specification.
4. ***Selection phase.*** From the list of alternative sets of computed operational requirements proposed by the learning phase, the engineer selects the set Req_j, which is then added to the current operational specification. This selection is domain dependent.

The four phases are then repeated until no violation is detected during the analysis phase. The steps are explained in what follows: Note that, without loss of generality, we focus on learning two types of operational requirements: required pre and trigger-conditions. Our approach can be adapted to learn required postcondition. Further details on this are given in Section 4.3.1.

4.1 Behavior Analysis

The analysis phase of the approach is concerned with automatically checking whether a given partial operational

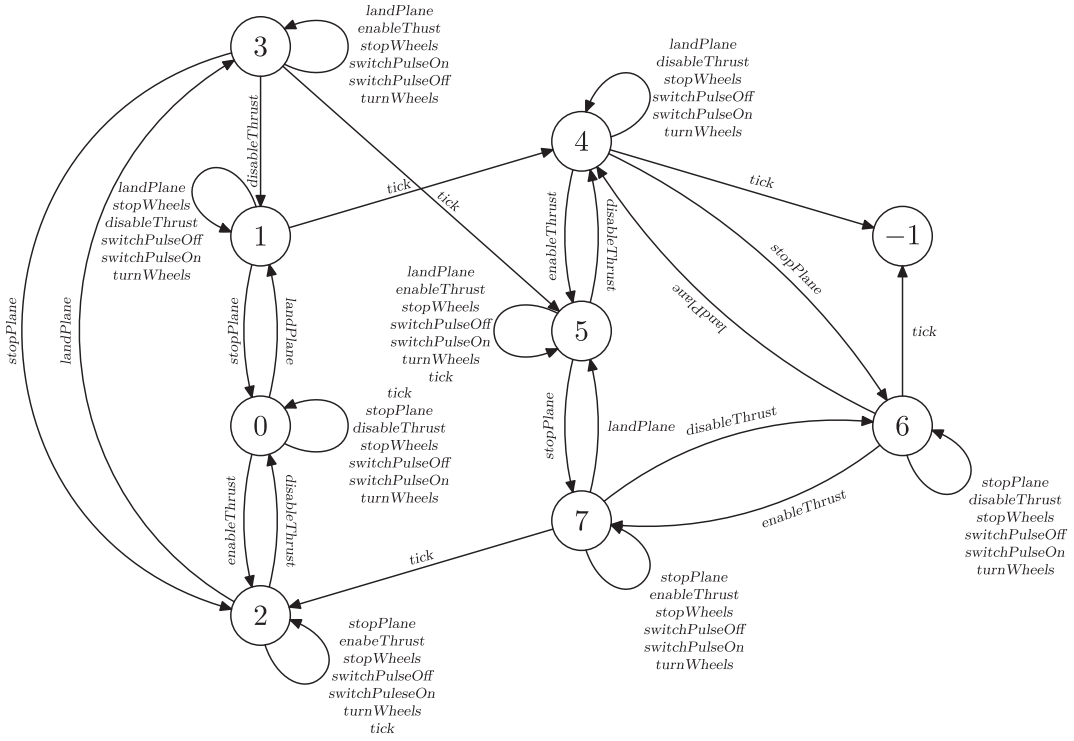

Fig. 3. Tester automaton for the goal [*ReverseThrustEnabledWhenMovingOnRunway*].

specification entails a given FLTL property ϕ, where ϕ maybe a safety or progress property. In the case of a progress property, ϕ is a conjunction of events.

Verifying an LTS model A against an FLTL *safety property* ϕ with fluents P, using the LTSA, requires generating a tester automaton for ϕ [21]. Fig. 3, for example, shows the tester automaton for the goal [*ReverseThrustEnabledWhenMovingOnRunway*]. Any trace from the initial state to the error state exemplifies one that violates the property ϕ. All traces that do not reach the error state are said to satisfy ϕ. In the case of the LTSA model checker, it produces the shortest sequence of events from the initial state to the error state.

When checking an LTS A for the satisfaction of a progress property ϕ, the LTSA searches for all terminal sets in A [34]. If there is at least one terminal set in which an event e in ϕ does not appear, then A is said to violate the progress property with respect to that event. The violation trace produced by the LTSA is the shortest sequence of events from the initial state to the root of a terminal set where the progress property is violated.

Consider the LTS model generated from the given FCS partial operational specification. When checking against the safety property [*ReverseThrustEnabledWhenMovingOnRunway*], the LTSA generates the violation trace shown below.

```
Trace to property violation in
ReverseThrustEnabledWhenMovingOnRunway:
    tick
    landPlane           MovingOnRunway
    turnWheels          MovingOnRunway
    switchPulseOn       MovingOnRunway
    tick                MovingOnRunway
    tick                MovingOnRunway
Analyzed in: 4 ms
```

The column on the left represents the sequence of consecutive transitions that occur starting from the initial state and ending at the state 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. The violation shown above, for instance, is caused by the fact that at the last observable state in the prefix $\langle$*tick*, *landPlane*, *turnWheels*, *switchPulseOn*, *tick*$\rangle$, i.e., at position 5 of the trace, the fluent *MovingOnRunway* is true and the fluent *ThrustEnabled* is false, and no initiating event occurs afterward. Hence the fluent *ThrustEnabled* is evaluated to false at the tick occurring at position 6 when it should have been true.

In the context of this work, the detection of a violation trace signifies a missing operational requirement for some software-controlled event occurring or not occurring within the last time unit and hence an incompleteness in the current operational specification.

4.2 Scenario Elicitation

Our aim is to identify required conditions on those events whose presence or absence in the counterexample leads to a goal violation.

From a logical standpoint, a number of correct solutions could be proposed to eliminate the counterexample. Our objective is to produce a solution that is relevant to the particular problem domain at hand while relieving the engineer from having to manually derive the full formal operational specification. Hence, the engineer is asked to provide some intuition on the cause of the violation in the form of scenarios that exemplify good and bad system behavior. The engineer is expected to identify, from the trace, events which may have contributed to the violation, along with other examples which are consistent with the property being verified.

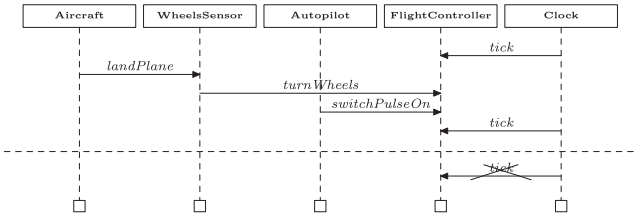

Fig. 4. Negative scenario from goal violation trace.

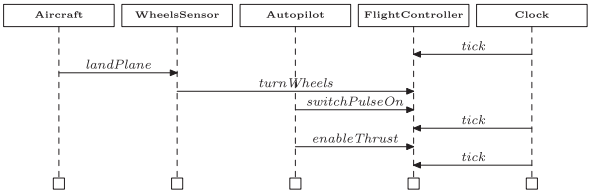

Fig. 5. Positive scenario satisfying goal [*ReverseThrustEnabledWhenMovingOnRunway*].

As the approach is concerned with developing required conditions, we only consider software-controlled events and the *tick* event as potential causes of the violation. In the following, we illustrate general guidelines for eliciting negative and positive scenarios from violation traces.

4.2.1 Eliciting Negative Scenarios

In eliciting a negative scenario from a violation trace, the engineer is required to identify an event in the violation trace that should not have occurred at a particular position in the trace. The sequence of event labels starting from the initial state of that trace up to and including the violating event is identified as a negative scenario. In general, given a trace of the form $(e_1, e_2, \ldots, e_k, \ldots)$, where e_k is the violating event, a negative scenario *neg* is the prefix $\langle e_1, e_2, \ldots, e_k \rangle$. The intended meaning of the negative scenario $\langle e_1, e_2, \ldots, e_k \rangle$ is that if the system exhibits the sequence $(e_1, e_2, \ldots, e_{k-1})$, then e_k should not happen immediately afterward.

In our models, the violating event is either a *software-controlled* event appearing in the last time unit or the last *tick* event of the trace produced. The former indicates the identified software-controlled event should not have occurred to avoid a goal be violated, while the latter suggests that a software-controlled event should have occurred before that tick for a goal to be satisfied. In general, in the LTSA the position of the violating event can often be predicted, depending on the pattern of the goal being checked.

Returning to our running example, the violation trace given in Section 4.1 terminates with two consecutive *tick* transitions with no software-controlled event occurring in between. The engineer could reason from this that some event e that initiates the fluent *ThrustEnabled* must occur within the last time unit for the goal [*ReverseThrustEnabledWhenMovingOnRunway*] to be satisfied. The last *tick* event is marked as the violating event since it should not have occurred. The sequence from the initial state until this last *tick* event in the violation trace constitutes a negative scenario (see Fig. 4 where the event occurring below the dotted line is the violating event).

4.2.2 Eliciting Positive Scenarios

Once the engineer has elicited a negative scenario terminating with a violating event e, he must provide at least one scenario which shows a positive occurrence of the same event e. The positive scenario should be a sequence of events:

- exemplifying when the event e in question *may* or *must* occur,
- accepted by the LTS of the given asynchronous operational specification,
- consistent with all the goals,
- terminating with a *tick* event,
- extendable with at least one subsequent infinite extension that satisfies the goals.

The first condition is to ensure that any requirement inferred from the scenarios would preserve good behavior in the model. The second and third are to ensure that any requirement learned is consistent with the current specification and goals. We require positive scenarios to terminate with *tick* to ensure that the occurrence of the satisfying event in the last time unit of the sequence does not violate the goals in that sequence. The last condition is needed to avoid the engineer providing a finite trace that satisfies the goals but where all possible extensions to this trace would violate them, and hence no completion to the initial set of requirements would exist.

To ensure the specified characteristics are satisfied, the engineer can use the built-in animation and deadlock features of the LTSA system to generate a positive scenario by first generating an LTS from the composition of the goals and operational specification and then walking through the LTS model by following a trace until the event e is executed.

We give an account of some general guidelines for eliciting positive scenarios that support learning the missing requirements. Consider a negative scenario of the general form $\langle e_1, e_2, \ldots, e_{k-1}, e \rangle$ where the violating event e is a software-controlled event. An associated positive scenario has the general form $\langle e_1, \ldots, e'_l, e, tick \rangle$, where the sequence $\langle e_1, \ldots, e'_l \rangle$ is either different from $\langle e_1, \ldots, e_{k-1} \rangle$ or a prefix of it. On the other hand, if the negative scenario terminates with a *tick* event as its violating event, $\langle e_1, \ldots, e_{k-1}, tick \rangle$, then an associated positive scenario would be a sequence with $\langle e_1, \ldots, e_j, \rangle$ as its prefix, where e_j, is a tick-event occurring at $1 \leq j < k$, showing that the occurrence of event e before the last tick is necessary for the satisfaction of the goal.

For example, a positive scenario associated to the negative scenario shown in Fig. 4 that satisfies the goal [*ReverseThrustEnabledWhenMovingOnRunway*] enforces the event *enableThrust* to occur before the last tick (see Fig. 5).

Note that because scenarios are finite traces, positive scenarios do not exemplify traces that satisfy progress properties. They merely capture desirable system behaviors which are consistent with the given goals, and could be subsequently extended with a sequence that satisfies a given progress property. Furthermore, all elaborated scenarios are assumed to include no hidden events in their sequence. The main reason for this is that, as discussed below, when computing an operational requirements specification, the learning phase uses the fluents that are true and false at the tick that precedes the violating event in the given sequence to compute hypotheses. Hidden events may obviously affect

the truth values of these fluents and hence would affect the correctness of the requirements with respect to scenarios elaborated in subsequent iterations. Note also that although we only show examples for elaborating a single positive scenario for each negative scenario, the engineer may elicit a number of positive scenarios from a violation trace.

The question that naturally arises here is how different should the positive scenarios be from the elaborated negative scenario? A possible heuristic is the *richness* of the positive scenarios with respect to a negative scenario. The richness of a set of positive scenarios with respect to a negative scenario is determined by the set of fluent literals evaluated at the tick event proceeding the occurrence of the satisfying event with respect to the fluent literals at the last tick proceeding the violating event in the negative scenario. This is expressed formally below.

Definition 3 (Rich set of Positive Scenarios). *Given a set of fluents F, a set D of fluent definitions, and an LTS model A, let* $\text{neg} = \langle e_1, e_2, \ldots, e_j, \ldots, g_k \rangle$ *be a negative scenario accepted in A, where $j < k$ is the position of the tick that precedes the violating event g. Let* $\sigma = (s_0 \mathcal{R}_{e_1} s_1 \ldots s_{k-1} \mathcal{R}_{g_k} s_k)$ *be an accepting trace of neg in A. Let* $\text{POS} = \{pos_1, \ldots, pos_n\}$ *and* $\text{POS}' = \{pos'_1, \ldots, pos'_m\}$ *be sets of positive scenarios accepted in A. Let $\{p_1, \ldots, p_n\}$ and $\{p'_1, \ldots, p'_m\}$ be the sets of the set of fluents that are true at the last tick preceding the occurrence of the event g in each positive scenario in POS and* POS'*, respectively. Then* POS' *is said to be richer than* POS *with respect to neg, denoted* $\text{POS} <_{neg} \text{POS}'$ *iff*

$$|(2^{p_1} \cup \cdots \cup 2^{p_n})) - 2^{s_j}| < |(2^{p'_1} \cup \cdots \cup 2^{p'_m}) - 2^{s_j}|,$$

where the notation 2^x denotes the power set of the fluent valuations in set x.

This property characterizes a set of positive scenarios that ensures that the learned requirements do not overconstrain the behavior of the final system, i.e., eliminate some behavior that may be considered in later iterations as positive. We illustrate the concept of richness through an extended example of the FCS.

Assume the fluent definitions include, in addition to those in Table 2, a definition for a fluent *EmergencyDeceleration*, meaning the pilot has requested a sudden drop in the speed of the aircraft as follows:

$$EmergencyDeceleration = \langle dropSpeed, fixSpeed \rangle,$$

where the event *dropSpeed* means drop the speed of the aircraft and the event *fixSpeed* means fix the speed of the aircraft.

Assume the negative scenario $neg = \langle tick, enableThrust \rangle$, stating that enabling the reverse thrust after the first tick is not permissible, is elicited where the event *enableThrust* is the violating event. Consider two sets of positive scenarios which show acceptable occurrences of the event *enableThrust*. Let the first set include the single scenario

$$pos_1 = \langle tick,\ landPlane,\ turnWheels,\ switchPulseOn,\ tick,\ enableThrust,\ tick \rangle,$$

which states it is acceptable to enable the reverse thrust when the plane is moving on the runway, the wheels are turning, and the wheels' pulses are switched on. Suppose the second set includes the two scenarios

$$pos'_1 = \langle tick,\ landPlane,\ turnWheels,\ switchPulseOn,\ tick,\ enableThrust,\ tick \rangle$$

and

$$pos'_2 = \langle tick,\ dropSpeed,tick,\ enableThrust,\ tick \rangle,$$

which together state that it is acceptable to enable the reverse thrust if the plane is moving on the runway, the wheels' pulse has been switched on, and the wheels are turning, or when there is a request for the plane to drop in speed. The sets of fluent literals evaluated to true at the last tick preceding the occurrence of the event *enableThrust* in all three positive scenarios are

$$p_1 = \{MovingOnRunway,\ WheelsTurning,\ PulseOn\},$$
$$p'_1 = \{MovingOnRunway,\ WheelsTurning,\ PulseOn\},$$
$$p'_2 = \{EmergencyDeceleration\}.$$

The power set for each of the above is computed. From Definition 3, we check the value of $|(2^{p_1}) - 2^{s_j}| = |9 - 1| = 8$ and $|(2^{p'_1} \cup 2^{p'_2}) - 2^{s_j}| = |10 - 1| = 9$. The number of elements after removing the power set of s_j from the union of the power set of p'_1 and p'_2 is greater than those as result of removing the elements appearing in the power set of s_j from the power set of p_1. This means that the positive scenarios in the latter set consider different courses of events before executing the desirable event and hence cover more acceptable behavior. Hence, the second set is said to be a richer set of positive scenarios with respect to the given negative scenario.

Identifying a rich set of scenarios ensures that the learning does not overgeneralize the learned hypotheses and hence produces operational requirements that are too restrictive, e.g., a required precondition that prevents the occurrence of the *enableThrust* whenever the plane is not on the runway.

4.3 Requirements Inference

The input to the learning phase is a set of goals G, a partial operational specification O, a set of fluent definitions D, and a set of elaborated positive and negative scenarios ($POS \cup NEG$) accepted in an LTS of O. The output of this phase is a set of operational requirements $\{Req_j\}$ that, together with the given partial specification, defines an LTS that accepts all positive scenarios in *POS* but none of the negative ones in *NEG*, and is consistent with all goals in G.

We present here the main steps involved in this phase, which are: 1) encoding the input into a language understandable by the learning system, 2) computing the new operational requirements, and 3) translating the learned requirements back into FLTL. It is important to note that these steps are done automatically and hidden from the engineer.

4.3.1 Translating FLTL Specifications into EC

The corresponding EC logic program uses variables of four sorts: events, fluents, time positions, and scenarios. Event and fluent constants are, respectively, the event-based fluents F_e and state-based fluents F_s of the FLTL language, time positions are represented by the nonnegative integers

0, 1, 2, ..., corresponding to the positions in the traces of an LTS, and scenario terms are constants introduced for each elicited positive and negative scenario. Definition 4 below is an adaptation of the encoding detailed in [1] which is relevant to the examples given in this paper.

Definition 4 (Encoding Operational Specifications into EC Programs). *Let O be an operational specification expressed in an asynchronous FLTL language. Let D be a set of fluent definitions and A be an LTS model generated from O with respect to D. Let $NEG \cup POS$ be a set of negative and positive scenarios. The corresponding logic program $\Pi = \tau(O) \wedge \tau(D) \wedge \tau(\mathrm{NEG} \cup \mathrm{POS})$ is the EC program containing the following clauses:*

- `initially(f,S) :- scenario(S).`
  for each fluent f defined to be initially true in D
- `initiates(e,f,P,S):- position(P), scenario(S).`
  for each event $e \in I_f$ of fluent f in D
- `terminates(e,f,P,S):- position(P), scenario(S).`
  for each event $e \in T_f$ of fluent f in D
- `impossible(tick,P,S):- position(P), scenario(S), (not_)holdsAt_PrevTick(f_1,P,S), ..., (not_)holdsAt_PrevTick(f_n,P,S), not occursSince_LastTick(e,P,S).`
  for each trigger-condition assertion $\Box(tick \rightarrow ((\bigwedge_{1 \leq i \leq n}(\neg) f_i) \rightarrow \bigcirc(\neg tick \,\mathrm{W}\, \mathrm{e})))$ in O
- `impossible(e,P,S):- position(P), position(P), scenario(S), (not_)holdsAt_PrevTick(f_1,P,S), ..., (not_)holdsAt_PrevTick(f_n,P,S).`
  for each precondition assertion $\Box(tick \rightarrow ((\bigwedge_{1 \leq i \leq n}(\neg) f_i) \rightarrow \bigcirc(\neg e \,\mathrm{W}\, tick)))$ in O
- `:- position(P), scenario(S), (not_)holdsAt_Tick(f_1,P,S), ..., (not_)holdsAt_Tick(f_n-1,P,S), not_holdsAt_Tick(f_n,P,S).`
  for each expectation assertion $\Box(tick \rightarrow ((\bigwedge_{1 \leq i \leq n-1}(\neg) f_i \rightarrow f_n)$ in O
- `:- position(P), scenario(S), (not_)holdsAt_Tick(f_1,P,S), ..., (not_)holdsAt_Tick(f_n-1,P,S), holdsAt_Tick(f_n,P,S).`
  for each expectation assertion $\Box(tick \rightarrow ((\bigwedge_{1 \leq i \leq n-1}(\neg) f_i \rightarrow \neg f_n)$ in O
- `attempt(`e_j`, j-1, `sc_i`)` *for each e_j in sc_i where $sc_i \in NEG \cup POS$,*

We assume that the program Π, resulting from the encoding above, is always extended with a set of EC domain-independent axioms (see Appendix B). For proof of soundness of the encoding, the reader is referred to [1]. Note that the above translation considers pre and trigger-conditions that prevent and require, respectively, the occurrence of an event e before the next tick of the system clock. The encoding may be adapted for other forms of required conditions, e.g., those that require the occurrence of an event after a number of tick occurrences in the past. This is done by using auxiliary predicates in the body of the rules such as *holdsAt_ith_Prevtick(f,p,i,s)*, which means that f holds at the ith tick occurring before position p in scenario s. The definition may also be extended to encode required postcondition as EC rules with the predicate *happens* in the body and *holdsAt_tick* in the head. Both negative and positive scenarios are translated into *attempt* facts. This is to state that both scenarios are permitted by the current specification. Note that, in the translation, one is subtracted from the time position argument of the *attempt* facts. This is because the EC program assumes that the effect of initiating and terminating events on fluents is only observable at the next time position in a scenario, as opposed to asynchronous FLTL, which assumes events have immediate effects on fluents in a given trace.

Table 4 shows an excerpt of the EC program for the FCS example, including the rules generated by applying τ to the fluent definitions in 2, the operational specification described in Table 3, and the scenarios in Figs. 4 and 5. For instance, the clauses shown in lines 5-6 are generated from the definition of the fluent *ThrustEnabled*, and the clause shown in lines 20-21 is constructed from the domain precondition of the event *switchPulseOn*.

4.3.2 Learning Requirements

To fully define the inductive learning task, a corresponding translation of goals into EC integrity constraints and scenarios into examples E must be specified. The former is to ensure that all learned operational requirement do not violate any of the goals. This encoding is given below.

Definition 5 (Encoding goals into integrity constraints). *Let G be a set of goals formulated as immediate achieve expressions in FLTL. The corresponding set of integrity constraints IC includes:*

- `:- position(P1), scenario(S), (not_)holdsAt_Tick(f_1,P1,S),..., (not_)holdsAt_Tick(f_n,P1,S), position(P2), nextTickAt(P2,P1,S), not_holdsAt_Tick(g,P2,S).`
  for each immediate cease goal $\Box(tick \rightarrow ((\neg) f_1 \wedge \cdots \wedge (\neg) f_n \rightarrow \bigcirc(\neg tick \,\mathrm{W}(tick \wedge \neg g))))$ in G,
- `:- position(P1), scenario(S), (not_)holdsAt_Tick(f_1,P1,S),..., (not_)holdsAt_Tick(f_n,P1,S), position(P2), nextTickAt(P2,P1,S), not_holdsAt_Tick(g,P2,S).`
  for each immediate achieve goal $\Box(tick \rightarrow ((\neg) f_1 \wedge \cdots \wedge (\neg) f_n \rightarrow \bigcirc(\neg tick \,\mathrm{W}(tick \wedge g))))$ in G.

Note that the corresponding EC integrity constraints capture the negation of the FLTL goals. For instance, the encoding of the asynchronous FLTL goal [*ReverseThrustEnabledWhenMovingOnRunway*], shown in lines 52-53, states

TABLE 4
An Excerpt of the EC Encoding for the Flight Control System

```
% *** BACKGROUND KNOWLEDGE ***
% ***********************************************
% DOMAIN-DEPENDENT AXIOMS
% — Fluent Definitions —
initiates(enableThrust,thrustEnabled,P,S) :- position(P), scenario(S).
terminate(disableThrust,thrustEnabled,P,S) :- position(P), scenario(S).
initiates(turnWheels, wheelsTurning,P,S) :- position(P), scenario(S).
terminates(stopWheels, wheelsTurning,P,S) :- position(P), scenario(S).
initiates(switchPulseOn, pulseOn,P,S) :- position(P), scenario(S).
terminates(switchPulseOff, pulseOn,P,S) :- position(P), scenario(S).
initiates(landPlane, movingOnRunway,P,S) :- position(P), scenario(S).
terminates(stopPlane, movingOnRunway,P,S) :- position(P), scenario(S).
terminates(takeOff movingOnRunway,P,S) :- position(P), scenario(S).

% — Domain Conditions —
impossible(enableThrust,P,S) :- position(P), scenario(S),
                                holdsAt_Tick(thrustEnabled,P,S).
impossible(disableThrust,P,S) :- position(P), scenario(S),
                                 not_holdsAt_Tick(thrustEnabled,P,S).
impossible(switchPulseOn,P,S) :- position(P), scenario(S),
                                  holdsAt_Tick(pulseOn,P,S).
impossible(switchPulseOff,P,S) :- position(P), scenario(S),
                                  not_holdsAt_Tick(pulseOn,P,S).
impossible(turnWheels,P,S) :- position(P), scenario(S),
                              holdsAt_Tick(wheelsTurning,P,S).
impossible(stopWheels,P,S) :- position(P), scenario(S),
                              not_holdsAt_Tick(wheelsTurning,P,S).

% — Assumptions —
   :- position(P1), scenarios(S), holdsAt_Tick(wheelsTurning,P1,S),
      not_holdsAt_Tick(pulseOn,P1,S).
   :- position(P1), scenarios(S), holdsAt_Tick(movingOnRunway,P1,S),
      not_holdsAt_Tick(wheelsTurning,P1,S).

% — Scenarios —
attempt(tick,0,neg_1).
attempt(landPlane,1,neg_1).
attempt(turnWheels,2,neg_1).
attempt(switchPulseOn,3,neg_1).
attempt(tick,4,neg_1).
attempt(tick,5,neg_1).

attempt(tick,0,pos_1).
attempt(landPlane,1,pos_1).
attempt(turnWheels,2,pos_1).
attempt(switchPulseOn,3,pos_1).
attempt(tick,4,pos_1).
attempt(enableThrust,5,pos_1).
attempt(tick,6,pos_1).

% — Goals —
   :- position(P1), scenarios(S), holdsAt_Tick(movingOnRunway,P1,S),
      position(P2), nextTickAt(P2,P1,S), not_holdsAt_Tick(thrustEnabled,P2,S).

% *** EXAMPLES ***
% **********************
examples :-
       happens(tick,0,neg_1), happens(tick,4,neg_1),
       not happens(tick,5,neg_1),
       happens(tick,0,pos_1), happens(tick,4,pos_1),
       happens(enableThrust,5,pos_1), happens(tick,6,pos_1).
```

that the constraint is violated if the fluent *movingOnRunway* is true at a tick occurrence and the fluent *thrustEnabled* is false at the next tick. The encoding of all the given goals is assumed to be included in the EC program.

The encoding of scenarios into examples E is driven by the violating event for which the required condition is to be learned, i.e., the last event in the negative scenario. The translation is given below.

Definition 6 (Encoding scenarios into EC examples). *Let NEG and POS be a set of negative and positive scenarios. The corresponding set of examples is the program E given by*

- *for each negative scenario* $neg_j = \langle e_1, \ldots, e_n \rangle$ *in NEG:*
  - E *includes* $n-1$ *facts of the form* `happens(e_i,i-1,neg_j)` *with* $1 \leq i \leq n-1$,
  - E *includes a fact of the form* `not happens(e_n,n-1, neg_j)`,
- *for each positive scenario* $pos_j = \langle e_1, \ldots, e_m \rangle$ *in POS:*
  - E *includes* m *facts of the form* `happens(e_i,i-1,pos_j)` *with* $1 \leq i \leq m$.

Note that to guarantee consistency of the learned hypothesis with the positive scenarios elicited in previous iterations, the translation of the latter is preserved

throughout all iterations. The encoding of the scenarios shown in Figs. 4 and 5 into examples is given in lines 57-61 of Table 4.

The task of learning required conditions is defined, within the context of EC programs, as learning rules that define an impossibility of a software and/or a tick event from an EC encoding of goals G, operational specification O, fluent definitions D, and scenarios (NEG $\cup$ POS). In our running example, the EC encoding of the partial operational specification and scenarios currently entails that both positive and negative scenarios happen. The learning task is hence required to find hypotheses that would prevent the violating event in the negative scenario from happening. The search space is defined by the following mode declaration:

$$\begin{aligned}
&\texttt{modeh}(*, \texttt{impossible}(\text{``}\sharp\texttt{tick\_event}\text{''}, \text{``}+\texttt{position}\text{''}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{``}+\texttt{scenario}\text{''})).\\
&\texttt{modeh}(*, \texttt{impossible}(\text{``}\sharp\texttt{sw\_event}\text{''}, \text{``}+\texttt{position}\text{''}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{``}+\texttt{scenario}\text{''})).\\
&\texttt{modeb}(*, \texttt{holdsAt_PrevTick}(\text{`` }\sharp\texttt{fluent}\text{''}, \text{`` }+\texttt{position}\text{''}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{``}+\texttt{scenario}\text{''})).\\
&\texttt{modeb}(*, \texttt{not_holdsAt_PrevTick}(\text{`` }\sharp\texttt{fluent}\text{''}, \\
&\qquad\qquad\qquad\qquad \text{`` }+\texttt{position}\text{''}, +\texttt{scenario}\text{''})).\\
&\texttt{modeb}(*, \texttt{not occursSince_PrevTick}(\text{`` }\sharp\texttt{sw_event}\text{''}, \\
&\qquad\qquad\qquad\qquad\quad \text{``}+\texttt{position}\text{''}, \text{``}+\texttt{scenario}\text{''})).
\end{aligned}$$

Given a corresponding EC program and mode declaration, the actual computation is performed by the XHAIL system, which is one of the few ILP systems designed for nonmonotonic ILP. It is based on a three-phase Hybrid Abductive Inductive Learning (HAIL) approach, introduced by Ray et al. [43], which operates by constructing and generalizing a preliminary ground hypothesis K, called a *Kernel Set* of B and E.

For our running example, XHAIL computes the minimal explanation as follows:

$$\Delta = \{impossible(tick, 5, neg\_1)\}, \tag{10}$$

The following ground rule is generated from the above explanation

$$\begin{aligned}
&\texttt{impossible}(\texttt{tick}, 5, \texttt{neg_1}) :- \\
&\quad \texttt{position}(5), \texttt{scenario}(\texttt{neg_1}),\\
&\quad \texttt{not_holdsAt_PrevTick}(\texttt{movingOnRunway}, 5, \texttt{neg_1}),\\
&\quad \texttt{not_holdsAt_PrevTick}(\texttt{wheelsTurning}, 5, \texttt{neg_1}),\\
&\quad \texttt{holdsAt_PrevTick}(\texttt{pulseOn}, 5, \texttt{neg_1}),\\
&\quad \texttt{not_holdsAt_PrevTick}(\texttt{thrustEnabled}, 7, \texttt{neg_1}),\\
&\quad \texttt{not occursSince_PrevTick}(\texttt{enableThrust}, 5, \texttt{neg_1}).,
\end{aligned} \tag{11}$$

and generalized to the following alternative solutions, each with a minimal number of body literals that are needed to cover the examples.

$$\begin{aligned}
&\texttt{impossible}(\texttt{tick}, \texttt{X1}, \texttt{X2}) :- \\
&\quad \texttt{position}(\texttt{X1}), \texttt{scenario}(\texttt{X2}),\\
&\quad \texttt{holdsAt_PrevTick}(\texttt{pulseOn}, \texttt{X1}, \texttt{X2}),\\
&\quad \texttt{not_holdsAt_PrevTick}(\texttt{thrustEnabled}, \texttt{X1}, \texttt{X2}),\\
&\quad \texttt{not occursSince_PrevTick}(\texttt{enableThrust}, \texttt{X1}, \texttt{X2}),
\end{aligned} \tag{12}$$

$$\begin{aligned}
&\texttt{impossible}(\texttt{tick}, \texttt{X1}, \texttt{X2}) :- \\
&\quad \texttt{position}(\texttt{X1}), \texttt{scenario}(\texttt{X2}),\\
&\quad \texttt{holdsAt_PrevTick}(\texttt{wheelsTurning}, \texttt{X1}, \texttt{X2}),\\
&\quad \texttt{not_holdsAt_PrevTick}(\texttt{thrustEnabled}, \texttt{X1}, \texttt{X2}),\\
&\quad \texttt{not occursSince_PrevTick}(\texttt{enableThrust}, \texttt{X1}, \texttt{X2}),
\end{aligned} \tag{13}$$

$$\begin{aligned}
&\texttt{impossible}(\texttt{tick}, \texttt{X1}, \texttt{X2}) :- \\
&\quad \texttt{position}(\texttt{X1}), \texttt{scenario}(\texttt{X2}),\\
&\quad \texttt{holdsAt_PrevTick}(\texttt{movingOnRunway}, \texttt{X1}, \texttt{X2}),\\
&\quad \texttt{not_holdsAt_PrevTick}(\texttt{thrustEnabled}, \texttt{X1}, \texttt{X2}),\\
&\quad \texttt{not occursSince_PrevTick}(\texttt{enableThrust}, \texttt{X1}, \texttt{X2}).
\end{aligned} \tag{14}$$

Once computed, our approach automatically transforms the learned hypotheses into asynchronous FLTL assertions. For instance, hypotheses 12, 13, and 14 are mapped back into FLTL, respectively as

$$\begin{aligned}
\Box(tick \rightarrow ((&PulseOn \wedge \neg ThrustEnabled)\\
&\rightarrow \bigcirc(\neg tick \ \mathrm{W}\ enableThrust))),
\end{aligned} \tag{15}$$

$$\begin{aligned}
\Box(tick \rightarrow ((&\mathrm{WheelsTurning}) \wedge \neg ThrustEnabled)\\
&\rightarrow \bigcirc(\neg tick \ \mathrm{W}\ enableThrust))),
\end{aligned} \tag{16}$$

$$\begin{aligned}
\Box(tick \rightarrow ((&MovingOnRunway \wedge \neg ThrustEnabled)\\
&\rightarrow \bigcirc(\neg tick \ \mathrm{W}\ enableThrust))),
\end{aligned} \tag{17}$$

stating that the reverse thrust should be enabled whenever the wheels' pulse is on, the wheels are turning, and the plane is moving on the runway, respectively.

The effect of the compression mechanism deployed by the learning system on the computed hypotheses is that our approach learns operational requirements that eliminate additional behavior sharing characteristics with the negative scenarios while keeping those sharing characteristics with the positive scenarios. However, the degree of the compression may be controlled by the richness of scenarios heuristic discussed in Section 4.2.2.

4.4 Requirements Selection

When the learning phase produces alternative sets of requirements, the engineer is required to select, from among these, the best requirements that fit his understanding of the system's intended functionality. The reason why only one set is selected is that, though each set of learned hypotheses is consistent with the background and explains the examples, the learning does not guarantee consistency among the alternative solutions computed in a single iteration, and hence selecting several solutions at once may invalidate integrity constraints given in the program.

For instance, the learning phase produced three alternative required trigger-conditions for the event *enableThrust*.

The former triggers the reverse thrust to be enabled when the wheels' pulse is on, the second when the wheels are turning, and the last when the plane is moving on the runway. Any would eliminate the violation trace for the goal [*ReverseThrustEnabledWhenMovingOnRunway*] shown in Section 4.1. The selection is subject to the engineer's understanding of the system and the goals the system must satisfy. We select the first as it is the only one realizable by the autopilot agent. Note that, although both are consistent with the current specification and the goals, only one is selected to remove the prescribed behavior and not both. Once the engineer makes his selection, the operational requirement is added to the current operational specification and a single iteration in the overall framework is completed.

Any LTS model generated from the newly extended specification does not accept the negative scenario elaborated in the previous phase and accepts all the positive scenarios. This fact, and the soundness of the learning step, is proven in Theorem 1. The theorem states that, given a partial operational specification O, fluent definitions D, goals G, and a consistent set (NEG $\cup$ POS) of negative and positive scenarios, any nonmonotonic ILP solution computed by XHAIL can be translated back into a set of operational requirements when added to the initial O; it would allow traces that accept all positive behaviors in POS but none of the negative ones in NEG.

Theorem 1 (Correctness of Learned Requirements). *Let O be an operational specification and G a set of goals both expressed in asynchronous FLTL. Let D be a set of fluent definitions, A an LTS model of O with respect to D, and $NEG \cup POS$ a set of negative and positive scenarios. Let $\Pi = \tau(O) \wedge \tau(D) \wedge \tau(\mathrm{NEG} \cup \mathrm{POS})$, IC be the encoding of the goals into integrity constraints, and E be the encoding of the scenarios into examples. Let H be an inductive solution for E, computed by the XHAIL algorithm, with respect to Π and IC, under the mode declaration MD, such that $\Pi \cup H \models E$. Then the corresponding set Req of asynchronous FLTL pre and trigger-conditions such that $\tau(Req) = H$, is a correct operational extension of O with respect to $NEG \cup POS$.*

The proof is given in [1]. Consequently, the newly generated LTS is also guaranteed not to exhibit the violation trace detected by the LTSA in the first phase (since it is a trace that accepts a negative scenario). In addition to the removal of all traces accepting the negative scenario (including the detected violation trace), other traces sharing common properties with the violation traces are eliminated. This is a consequence of the generalization feature of the learning system, which identifies common undesirable properties from the negative scenarios.

5 Termination

Within a given application of our framework, there may be a number of iterations of the cycle depicted in Fig. 1. The process is repeated until all the necessary required pre and trigger-conditions have been learned: Those which, together with the initial specification, allow only those behaviors that satisfy the goals. The process terminates when no further violations are detected in the analysis phase.

In the case of safety properties, the termination of the cycle is reached once the error state is no longer reachable in the composition of the LTS generated from the specification and the negation of the property in question. If, in each iteration, no new event labels are introduced, it can be shown that a finite number of iterations is needed to make the error state unreachable. The argument is based on the idea that each state in the LTS generated in the analysis phase can be characterized (through bisimulation) by a unique fluent valuation. Given that there are a finite number of states and fluents, then there is a finite number of fluent valuations to characterize this LTS. As each rule that is learned removes at least one transition from the generated LTS in a trace leading to the error state, a finite number of iterations is required to make the error state unreachable.

The argument as to why the states of an LTS resulting from the composition of the LTSs for the operational requirements specification and the property can be characterized with a unique fluent valuation is as follows: LTSA synthesizes the least constrained (with respect to trace inclusion) minimal (with respect to bisimulation) LTS that satisfies the operational requirements specification. By least constrained we mean that the generated LTS includes all possible traces that satisfy the operational specification, while by minimal we mean the LTS is one which contains the least number of states in comparison with any other LTS satisfying the specification. This is also the case of the LTS generated from the negation of the property. The value of a fluent in a single state in the generated LTS may be true or false, depending on the trace executed to reach that state. States that have these characteristics are referred to as *top* states [10]. However, a bisimilar finite state LTS can be built that does not have any top states, i.e., each state in the LTS has a particular fluent valuation that holds for all traces leading to that state. An algorithm showing this computation is given in [10]. Furthermore, this valuation is unique as the operational requirements specification language can only express behavior in terms of fluent valuations. Hence, two states with the same valuation must have the same behavior.

When checking a specification against a progress property, the termination of the cycles is reached if in every terminal set of the LTS, there is a reachable state with an outgoing transition of each event in the progress. Similar to the argument of termination when checking against safety properties, since each learned rule removes at least one transition, a finite number of iterations is required to make a terminal set in which the transitions in the progress set no longer appear unreachable.

The above cases consider what happens when the elaboration process successfully terminates. In other cases, the process may come to a premature termination point, i.e., before learning all requirements that would guarantee the satisfaction of the desired property. Such termination often occurs when the engineer discovers a positive scenario that is inconsistent with the current specification during the scenario elaboration. This case occurs if the requirements learned in previous iterations are too strong (i.e., there is a precondition that constrains the occurrence of events more than is necessary or a trigger-condition that forces the occurrence of events more than required).

The choice of requirements to include in the specification also has an impact on the overall elaboration process. For

instance, a precondition that is too strong may prevent the system being modeled from behaving desirably and hence affects the conditions of a requirement learned in subsequent iterations. On the other hand, conditions that are too weak may marginally constrain the specification and lead to a larger number of iteration steps before termination.

Having identified a premature termination point, the engineer will have to backtrack to specific decision points in previous iterations. The proposed approach has two main junctures at which a choice is made: the elaboration of scenarios and the selection of requirements. Backtracking to the previous iteration will go to the immediately preceding decision point. The engineer thus first backtracks to the selection phase of the previous iteration in order to select alternative requirements. Failing this, he will then backtrack to the scenario elicitation phase of the previous iteration and so on—if necessary, to previous iterations.

6 Validation

Our general strategy for validation is to apply the proposed framework to a number of case studies for which a formal operationalization of the goals exists and compare the outcome of our application with those obtained by other existing techniques.

6.1 Methodology

For each case study, we consider a set of goals expressed in LTL and a complete set of operational requirements with respect to the goals. These are automatically translated into an asynchronous FLTL specification using the techniques described in [29].

We start from a set of goals and a partial operational specification, and iterate through the four phases described in this paper. We assume that the set of goals and initial partial operational specification are consistent and that the goals may vary from high-level to low-level ones. The final set of operational requirements learned is compared to the ones provided using the operationalization patterns in [29], [28], and [27].

Whenever human intervention is required, we play the role of the engineer. As this intervention is only required in two activities, 1) scenario elicitation and 2) requirements selection, we believe that our influence on the validation of the approach is minimal. For 1), no understanding of the underlying approach is needed, only understanding of the problem domain. It may be argued that the choice of the elicited scenario is biased by the understanding of the underlying approach; however, this selection is done following the heuristics set out in this paper regarding richness of scenarios (see Section 4.2.2). For 2), human intervention is limited to checking if one of the proposed requirements automatically learned is part of the original specification developed by a third party. All case studies are executed on a standard desktop computer (Core2Duo, 1 GB RAM).

In what follows, we show an excerpt of the London Ambulance Service System case study. Our aim for this case study is to show that the approach is capable of handling various forms of goal specifications; in this case the one specified using the *bounded* achieve/cease pattern $\Box(tick \rightarrow (P \rightarrow \diamond_{\leq d}(\neg tick \text{ W } (tick \wedge (\neg)Q))))$.

6.2 London Ambulance Service System

The LAS was implemented in 1992 for dispatching ambulances to emergency incidents in London. The following description is taken from [16].

> *The LAS despatch system is responsible for: receiving calls; despatching ambulances based on an understanding of the nature of the calls and the availability of resources; and monitoring the progress of the response to each call. A computer-aided despatching system was to be developed, and would include an automatic vehicle locating system (AVLS) and mobile data terminals (MDTs) to support automatic communication with ambulances. This system was to supplant the existing manual system.*

The initial set of goals is taken from the specification provided in [27], where they are specified in first-order logic LTL. For the purpose of applying our approach, we transpose the formalization of goals and operational requirements into propositional FLTL.

We assume the given goals are correct and show how operational requirements can be derived for them. Only a subset of the operational specification in [27] constitutes our initial partial operational specification. We consider definitions for the fluents *Allocated*, *Mobilized*, *Available*, and *Encoded*, as well as the fluents *Occurs_encode* and *Occurs_alloc*, which signal the occurrences of the encode and allocate events, respectively, since the last clock tick. The last two fluents are terminated by an event *tock*, which proceeds every occurrence of a *tick* event. These are given below.

$$
\begin{array}{lcl}
\mathit{Allocated} & = & \langle \{\mathit{allocate}\}, \{\mathit{deallocate}\} \rangle, \\
\mathit{Available} & = & \langle \{\mathit{free}\}, \{\mathit{assign}\} \rangle \ \textbf{\textit{initially}}\ \mathit{true}, \\
\mathit{Occurs\_alloc} & = & \langle \{\mathit{allocate}\}, \{\mathit{tock}\} \rangle, \\
\mathit{Mobilized} & = & \langle \{\mathit{mobilize}\}, \{\mathit{demobilize}\} \rangle, \\
\mathit{Intervention} & = & \langle \{\mathit{intervene}\}, \{\mathit{withdraw}\} \rangle, \\
\mathit{Encoded} & = & \langle \{\mathit{encode}\}, \{\mathit{decode}\} \rangle, \\
\mathit{Occurs\_encode} & = & \langle \{\mathit{encode}\}, \{\mathit{tock}\} \rangle.
\end{array}
$$

The partial operational specification includes the following required precondition:

$$\Box(tick \rightarrow (\neg Allocated \rightarrow \bigcirc(\neg mobilize \text{ W } tick))),$$

meaning that an ambulance may not mobilize if it has not been allocated, and

$$\Box(tick \rightarrow ((Allocated \wedge \neg Intervention) \rightarrow \bigcirc(\neg demobilize \text{ W } tick))),$$

stating that an ambulance may not demobilize if it is allocated and has not intervened, as well as the domain preconditions:

$$\Box(tick \rightarrow (Available \rightarrow \bigcirc(\neg free \text{ W } tick))),$$
$$\Box(tick \rightarrow (\neg Available \rightarrow \bigcirc(\neg assign \text{ W } tick))),$$
$$\Box(tick \rightarrow (Allocated \rightarrow \bigcirc(\neg allocate \text{ W } tick))),$$
$$\Box(tick \rightarrow (\neg Allocated \rightarrow \bigcirc(\neg deallocate \text{ W } tick))),$$
$$\Box(tick \rightarrow (Mobilized \rightarrow \bigcirc(\neg moblize \text{ W } tick))),$$
$$\Box(tick \rightarrow (\neg Mobilized \rightarrow \bigcirc(\neg demoblize \ W \ tick))),$$
$$\Box(tick \rightarrow (Intervention \rightarrow \bigcirc(\neg intervene \text{ W } tick))),$$
$$\Box(tick \rightarrow (\neg Intervention \rightarrow \bigcirc(\neg withdraw \text{ W } tick))),$$

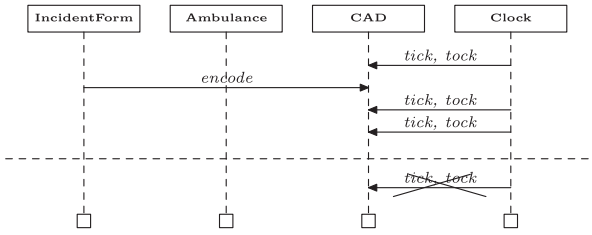

Fig. 6. Negative scenario from goal violation trace.

and the expectation

$$\Box(tick \rightarrow (Allocated \leftrightarrow \neg Available)),$$

stating that whenever an ambulance is allocated, it is not available. The operational specification also contains domain preconditions that restrict repeated occurrences of events between two consecutive ticks. Note that it is possible to consider an empty operational specification with fluent definitions only and iteratively generate all the operational requirements needed to satisfy the goal.

We check this specification against two forms of goals: safety and time progress. Our focus is on elaborating required conditions for events of the CAD component which is responsible for sending signals as well as allocating, assigning, and mobilizing an ambulance. We assume the CAD component monitors all information about the incident and controls all ambulance events.

6.2.1 Safety Check

We consider the goal [*AllocationBasedOnIncidentFormWhenAmbAvailable*], formalized as the following safety property in asynchronous FLTL:

$$\Box(tick \rightarrow ((Occurs\_encode \wedge Available) \rightarrow \bigcirc((\neg tick \vee \bigcirc(\neg tick \,\mathrm{W}\, (tick \wedge Allocated))) \,\mathrm{W}\, (tick \wedge Allocated)))).$$

The above states that once an incident is encoded, an available ambulance must be allocated within the next two time units, i.e., before the occurrence of a second tick.[1]

First, we use the LTSA to generate an LTS model from the initial partial operational specification. The synthesized LTS contains 3,394 states and 12,994 transitions. Running an LTL property check on this LTS results in the following violation trace:

```
Trace to property violation in
 AllocationBasedOnIncidentFormWhenAmb-
 Available:
  tick       Available
  tock       Available
  encode     Occurs_encode && Available
  tick       Occurs_encode && Available
  tock       Available
  tick       Available
  tock       Available
  tick       Available
 Analyzed in: 38 ms
```

1. This is a simplified version of the one specified in [27] which also includes a condition that the time it takes the ambulance to reach the incident location is be less than 11 time units.

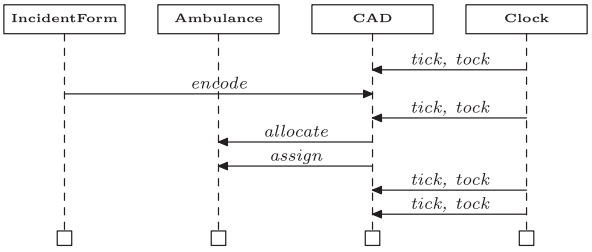

Fig. 7. First positive scenario satisfying the goal.

We play the role of the engineer here and try to identify the violating event. As the violation trace is quite short, its inspection is feasible. The truth value of the fluents appearing in the goal is indicated in the right column of the trace. On observation, we notice that at the second tick, an incident form is encoded and an ambulance is available but has not been allocated by the second proceeding tick. As the *Allocated* fluent is false and not made true by an occurrence of an initiating event before the last tick, we conclude that the last tick is the violating event. Fig. 6 shows the negative scenario extracted from the above violation trace.

To elicit positive scenarios, we regenerate a new LTS from the composition of the partial operational specification and the goal [*AllocationBasedOnIncidentFormWhenAmbAvailable*]. We then use the LTSA run facility to produce a positive scenario that is consistent with the partial specification and satisfies the goal. In this instance, we elicit two positive scenarios showing that allocating an ambulance within the first or second time units after an incident is encoded are both desirable behaviors. These are shown in Figs. 7 and 8.

The elicited scenarios, goal, and partial operational specification are systematically transformed into a corresponding logic program and given to the learning system as input. The XHAIL system produces a number of plausible solutions for eliminating the negative scenario. The FLTL formulation of two learned required trigger-condition variants is given below:

1.

$$\mathrm{ReqTrig}_1(allocate) = \Box(tick \rightarrow (((Occurs\_encode \wedge Available \wedge \neg Allocated) \wedge (\bigcirc(\neg tick \,\mathrm{W}\, (tick \wedge Encoded)))) \rightarrow \bigcirc((\neg tick \vee \bigcirc(\neg tick \,\mathrm{W}\, allocate)) \,\mathrm{W}\, (allocate))),$$

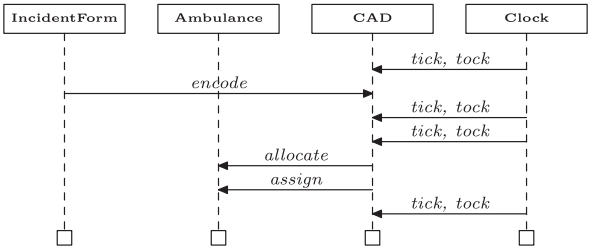

Fig. 8. Second positive scenario satisfying the goal.

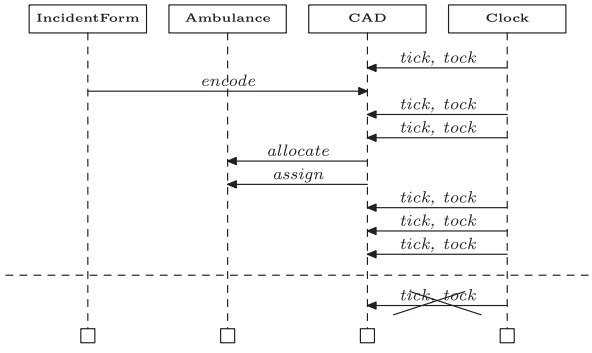

Fig. 9. A negative scenario leading to a progress violation.

which says that the event *allocate* should occur by the third tick if the incident has just been encoded, an ambulance is available, and the incident remains encoded at the next tick.

2.

$$\begin{aligned} ReqTrig_2(allocate) = \Box(tick \rightarrow (((&Encoded \\ &\wedge Available \wedge \neg Allocated) \\ &\wedge (\bigcirc(\neg tick \ \mathrm{W} \ (tick \wedge \neg Allocated)))) \\ &\rightarrow \bigcirc((\neg tick \vee \bigcirc(\neg tick \ \mathrm{W} \ allocate)) \\ &\quad \mathrm{W} \ (allocate))), \end{aligned}$$

which states that the event *allocate* must occur by the third tick if an incident has just been encoded, an ambulance is available, and an ambulance is not allocated at the next tick.

By comparing the learning outcome with the trigger-condition obtained from applying the operationalization patterns in [29], we select the second learned operational requirement and add this to the initial operational specification. This marks the end of the first iteration.

Running the analysis on the extended specification, which now includes the selected $ReqTrig_2(\mathrm{allocate})$, against the same goal shows that the refined LTS contains no further violations. Hence, a complete set of operational requirements with respect to the goal [*AllocationBasedOnIncidentFormWhenAmbAvailable*] has been identified.

6.2.2 Progress Check

The above shows an excerpt of the validation procedure for checking against safety properties. In this section, we show how the framework is also used to verify the satisfaibility of progress properties. Our focus is on checking the time progress property. To illustrate the problem of progress violations in LTSs, we deploy a common assumption in reactive systems called the Maximum Progress (MP) assumption [14]. Effectively, this gives priority to system events over all other events including ticks.

By performing a TP progress check on the LTS generated from the given FLTL specification, under the MP assumption, the LTSA produces the following counterexample:

```
Progress violation: TimeProgress
Trace to terminal set of states:
    tick
    tock
```

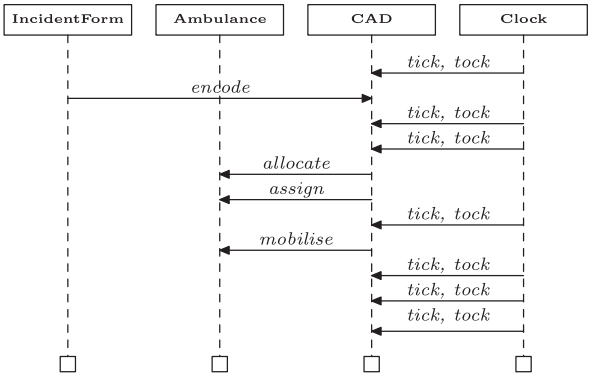

Fig. 10. A positive scenario satisfying the goals and operational specification.

```
    encode
    tick
    tock
    allocate
    assign
    tick
    tock
    tick
    tock
    tick
    tock
    tick
Cycle in terminal set:
Actions in terminal set:
 {}
Progress Check in: 3 ms
```

The violation shows a case in which an incident form is encoded, an ambulance is allocated to resolve the incident by the third tick, after which time progresses until the fifth time unit, where a deadlock state is reached.

From the violation trace and our understanding of the system, we identify a missing occurrence for the event mobilize since no ambulance was mobilized by the last time point in the trace. Therefore we consider the violating event to be the last *tick* event. The procedure for eliminating the violation is similar to that for safety violations. The negative scenario becomes the whole sequence of events presented in the trace as shown in Fig. 9. The positive scenarios are given in Figs. 10, 11, 12 showing positive occurrences of the event *mobilize*.

The positive scenarios collectively state that if an incident form is encoded and an ambulance is allocated to resolve the incident, then the ambulance may mobilize to the incident location within the next three time units.

The learning produced a number of required trigger-condition for the event *mobilize* expressed in FLTL, including

$$\begin{aligned} \mathrm{ReqTrig}_1(mobilize) = \Box(tick \rightarrow ((&Allocated \wedge \neg Mobilized) \\ &\rightarrow \bigcirc((\neg tick \vee \bigcirc(\neg tick \ \mathrm{W} \ mobilize)) \\ &\quad \mathrm{W} \ mobilize))) \end{aligned}$$

Informally, the above ensures that an ambulance is mobilized within the first two time units after an ambulance has been allocated to the incident site.

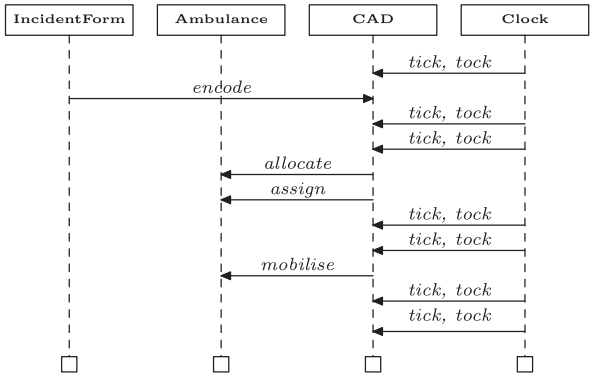

Fig. 11. A positive scenario satisfying the goals and operational specification.

6.3 Case Study Conclusion

We have demonstrated how the approach can be used for the operationalization of different goal patterns. Though we consider a partial set of operational requirements to be given as input, the process may be applied to the case where no operational requirements are known.

In addition to the FCS and LAS case studies, we have applied our approach to other systems, including the Mine Pump [26] and the Engineered Safety Feature Actuation System (ESFAS) described in [28]. Table 5 gives an overview of these applications.

The case studies indicate that the number of iterations is mainly affected by the number of goals that are violated in the generated model. Furthermore, they show that there is an increase in the computation time with respect to the number of fluents and events in the language, the length of elicited scenarios, the number of formulae that form the background, and the number of possible solutions. This is particularly noticeable in the learning phase, as opposed to the analysis phase. This is due to a number of factors. First, the learning system computes the truth value of each fluent according to each scenario to produce an explanation for the uncovered example. Then it tries to find all possible minimal explanations which may cause the search space to become very large and hence the complexity of the search increased.

In the mine pump example, the average time for computing a solution is less than those for the other case

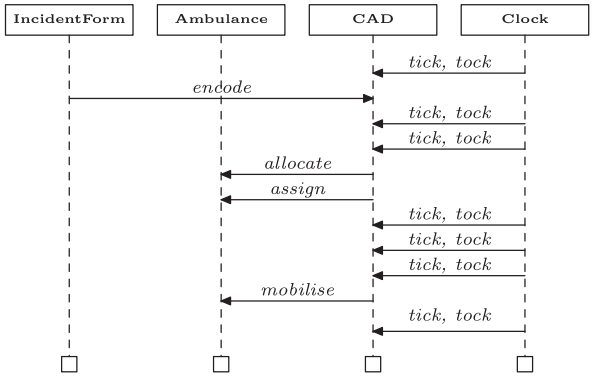

Fig. 12. A positive scenario satisfying the goals and operational specification.

TABLE 5
Summary of Case Studies

Case Study	Events	Fluents	Goals	cycles	sec/cycle
Mine Pump	10	5	4	8	∼29 sec
ESFAS	17	12	5	11	∼88 sec

studies. This is influenced by the fact that the scenarios elicited from the violation traces are shorter than those produced for the other case studies. In the ESFAS, on the other hand, no required conditions are initially captured. Also the number of fluents is higher than that in the other case studies and hence the computation of their truth value is more computationally demanding. In the last case study, the increase in average time is mainly caused by the number and length of the scenarios included in the program.

Our case studies indicate that there is a dependency between the richness of the scenario and the weakness of pre and trigger-conditions computed by the learning phase. The more different the course of events leading to a satisfying occurrence of an event in the positive scenario with respect to the negative scenario, the weaker the conditions computed by the learning system are and hence the less likely that it will cause a premature termination.

In comparison with standard manual approaches, we believe that the proposed approach is simpler and requires less work for the engineer in many cases, providing consistent elaborations of the requirements specification and not simply potential corrections for rechecking. This is because understanding the exact cause of the violation trace, which is a sequence of event labels in the given specification, which is a set of temporal formulae, is hard. Any edits to such a specification may introduce inconsistency and new errors into the specification. Requiring feedback from the engineer in the form of example scenarios, which are more intuitive [49], rather than direct manipulation of the temporal specification is also intended to relieve the engineer of this nontrivial task.

7 Discussion

The applicability of the learning algorithm to a requirements elaboration problem is dependent on the ability to provide a sound encoding into a logic program. Hence, any formal language for which such an encoding can be constructed may be used. Also, it is possible to use the EC language or other logic programming formalisms directly as a specification language.

The scalability of the approach is dependent on the scalability of the model checking and ILP techniques used independently. As our current analysis is on the system LTS, the scalability of the model checking may be improved by deploying compositional verification methods as well as by using abstraction techniques in our elaboration. Additional constraints in the learning process are needed, however, to ensure the soundness of the encoding and correctness of the solutions. Also, recent advances have helped to improve the scalability of ILP systems significantly through the use of search heuristics, search ordering, and pruning [15].

As the successful completion of the approach relies, to an extent, on the engineer's ability to elicit positive and negative scenarios that contribute to learning correct requirements, providing further support for this process is

therefore desirable. In addition to the conditions considered in Section 4.2.2 specific to identifying positive scenarios, our experience has shown that there are a number of considerations which often reduce the risk of backtracking. These include the following:

- Verifying the model against the progress properties first and then the safety properties. Our method for eliminating traces that violate progress properties is based on the *Maximal Progress* assumption, a common assumption for reactive systems, which means that the software will do all it *can* before the next tick. Hence, to prevent the software from executing events that would lead to a progress violation, preconditions are often learned and added to the specification. Because the learning system is guaranteed to only produce requirements that are consistent with given specification, then enriching the specification with preconditions for an event guarantees that any learned trigger-condition (in later iterations) for that same event will be consistent with any precondition for that event.
- Giving preference to learning preconditions over trigger-conditions, for the same reason discussed above.
- Checking the satisfaction of a set of goals in a single iteration. Often checks against goals individually may facilitate the identification of the violating event in violation trace. However, checking the specification against the conjunction of given goals may lead to learning requirements that remove more violations in a cycle than if performed individually.

Note that the learning system automatically checks that the sets of positive and negative scenarios are disjoint in the sense that a single scenario cannot be deemed to be both positive and negative. If the engineer makes such a mistake, then the learning system produces a message that no operational requirement can be computed for this set of examples. However, the current approach does not address the case in which an engineer elicits a scenario and then realizes in a later iteration that this scenario is not valid within the domain being described. To handle such a situation, the approach needs to be able to revise the operational specification to cope with such changes. This is discussed further in Section 9.

Furthermore, the learning step in our approach is restricted to the vocabulary introduced by the engineer. This means that any solution the learning system computes will refer to the set of events and fluents the engineer has already provided. Nonetheless, it possible for an engineer at the beginning of an iteration to introduce additional relevant events or fluents to the vocabulary. Once these have been determined, the learning process will produce solutions based on the extended vocabulary.

8 Related Work

Among the few approaches concerned with goal operationalization are the NFR framework [39], GBRAM [6], and CREWS [46], [45]. However, these either focus on nonfunctional requirements or are informal and hence cannot be fully verified.

Fuxman et al. present the TROPOS methodology [18], [17] for formally specifying and analyzing goals and requirements. They also use a symbolic model checking tool [7] to perform the analyses. However, unlike our approach, they do not provide support for correcting and completing the specification with respect to goals.

In [31], a formal framework is presented for incrementally constructing an operational requirements specification from goals. This is done by applying goal refinement patterns to high-level goal assertions expressed in Real-Time Linear Temporal Logic (RT-LTL) [35] to generate a set of terminal goals realizable by some agent in the software. The terminal goals are then used to derive the list of operations to be performed by the system and their domain pre and postconditions. The domain conditions are then strengthened with required pre, trigger-, and postconditions using the operationalization pattern catalogue provided. One main advantage of this approach is that the final product, i.e., the operational requirements, is guaranteed to be complete, consistent, and minimal with respect to the goals [31]. Nonetheless, the case studies suggest that our framework can produce alternative and acceptable operationalization for the same goal that are complete and consistent. As in our work, operational requirements derived using the operationalization approach in [31] can be checked for consistency and satisfaction of the goals using the operationalization checker FAUST [44]. However, if a counterexample is detected, the engineer is assumed to *manually* revise the declarative specification to resolve the violation [29]. As consistency checks are applied to the goals only and are assumed to precede the operationalization process, any changes to the specification will have to be made to the goals first to guarantee consistency. Using our methodology, on the other hand, the consistency of the requirements is automatically guaranteed by the learning system without the need for further analysis. One limitation in our approach as opposed to that described in [31] is that the minimality of the generated set of required condition with respect to the goals is scenario-dependent and therefore not guaranteed. By minimality of the requirements, we mean that the requirements should not restrict the behavior of the system more than is required to satisfy the goals. Though we have not paid particular attention to this during the elaboration process, we believe that by including richer (positive and negative) scenarios, the requirements will be less restrictive and so minimality is more likely to be preserved.

The work in [52] presents a method for inferring declarative assertions from scenarios. It elicits goals, as temporal formulae, from tailored scenarios provided by stake-holders using an inductive inference process based on Explanation-Based Learning (EBL) [36]. The tailored scenarios are then used to elicit new declarative goals that explain the given scenario. These new goals are added to the given initial (partial) goal model for "nonoperational" analysis (i.e., goal decomposition, conflict management, and obstacle detection). Our approach differs from that in [52] in numerous aspects. First, the inductive inference of declarative goals in [52] is based on EBL, which does not take into account current knowledge of the system (e.g.,

existing goals or operational requirements) during the inference process. It is therefore (potentially) an unsound inference process in that it can generate declarative goals that are inconsistent with the given (partial) goal model. ILP, on the other hand, which is the paradigm we use, makes substantial use of the knowledge provided and only produces rules that are consistent with it. Hence, goals, operational requirements, and any available knowledge are used as constraints on possible acceptable solutions. As a result, any inferred rules can therefore be directly added to the current specification. Moreover, the generalization process in [52] is from the most specific case (i.e., the scenarios) to a more general description (i.e., the goals). Negative scenarios are used to avoid overgeneralization of the permissible behaviors. Although ILP generalizes from specific examples to general rules, these rules are used to constrain an overgeneralized behavior specification which allows both desirable and undesirable behavior. Scenarios are used to avoid overconstraining the behavior in the final set of permitted behaviors. The approach [52] is, in addition, sequential, considering one scenario at a time during the inference procedure. The method used in our work accommodates multiple positive and negative scenarios *collectively* in a single inference process.

Other approaches, such as [11], [50], are synthetic, in the sense that they incrementally extend the set of permissible behavior. Although the outcomes of these approaches are LTSs that may capture the same set of behavior as that in the LTS generated by the final set of operational requirements, the approach itself does not support the elaboration of a declarative specification which can then be used to guide the system implementation.

The framework presented in this paper builds upon our preliminary work on learning requirements from goal models in [3], [2]. This paper mainly differs in that: 1) It presents a framework that supports the operationalization of goals expressed as both safety and progress properties, 2) it discusses the termination of the approach, and 3) it provides a more comprehensive validation with the application to the London Ambulance Service System. It also relates to the work in [5], [4], where we presented an approach for learning pre and trigger-conditions for operations from scenarios only. The approaches described in both works do not consider goals in the process nor do they use model checking to guide the learning process. Furthermore, the operational requirements are represented using synchronous LTL and hence the methods are not capable of reasoning about the two levels of granularity discussed in this paper. This is also reflected in the EC programs, which do not refer to what is true at *tick* points.

9 Conclusion and Future Work

The overall aim of this work is to develop a systematic approach for elaborating operational requirements that satisfy the stake-holders' goals. In particular, we focus on providing formal, automated support for *analyzing* a given partial operational specification and *completing* it with respect to the given goals.

This paper provides automated formal methods for analyzing a given consistent and correct operational specification, expressed in propositional FLTL, and suggesting ways to fix detected problems—based on a fixed vocabulary provided by the engineer. It specifically focuses on problems that are caused by incompleteness of the operational specification with respect to goals (expressed in the achieve/cease mode) rather than its incorrectness. We have deployed two well-founded techniques, model checking and ILP, to perform the elaboration task. Any requirement that is computed is automatically guaranteed to be correct and consistent with the goals and existing operational specification. At points where users' input is believed necessary, we simplify the representation of the problem by using scenarios to express positive and negative behavior and the selection of proposed requirements by guaranteeing their correctness. We also present a sound method for encoding specifications expressed in FLTL into EC logic programs. Such encoding has provided us with insight on ways of integrating temporal-based analysis methods which are widely used in software development, such as goal decomposition, agent assignment, and scenario elaboration, and the various techniques offered by logic programming, such as learning [33] and planning [48].

Our approach can be adapted to learning fluent definitions from scenarios. This would correspond to learning *initiates* and *terminates* EC rules, as shown in [41]. In this way, we would be able to provide support for the computation of domain conditions from scenarios, as well as required conditions, allowing the stake holders to convey such descriptions purely in terms of narrative-style scenarios of system behaviors, rather than temporal assertions. Additionally, the learning phase can be performed in isolation to learn requirements directly from scenarios. See [5] for further details.

In future work, we envision extensions specific to the current framework as well as extensions to more general related themes within software engineering and closely related disciplines.

One of the main areas we intend to investigate is handling incorrect and/or inconsistent operational specifications. For this purpose, we will consider the use of revision-based ILP techniques such as in [9]. This would provide support for correcting requirements previously elaborated with respect to newly elicited goals and scenarios, which in effect would avoid the need for backtracking.

We plan to incorporate information about the software architecture in the learning phase to restrict the solution space so that only those that satisfy constraints imposed by the architecture are computed. For instance, predicates that capture the monitorability and controllability of events by different components will be included in the background knowledge. Furthermore, soft goals will be used as constraints to select among alternative hypotheses.

A further useful extension would be to consider the scenarios used for learning requirements to be incomplete, i.e., contain hidden or unobservable transitions in their sequences. This would hence simplify the scenario elaboration, which currently requires sequences to be complete. One way is to explore the use of triggered scenarios [47]. Such consideration requires the requirements to be learned with respect to the scope of the given scenarios.

Additionally, more research on ways of supporting the generation of rich scenarios that contribute to learning

requirements that would preserve a larger number of desirable traces is a key interest. We believe this to be possible since the LTS generated from the specifications is the least constrained LTS which already contains traces that satisfy the goals. For this, we will consider adopting automated technique such as planning as described in [40].

We also intend to investigate the applicability of the framework to the problem of elaborating requirements expressed in first-order logic. Though the learning phase uses a first-order logic, other analysis techniques will need to be deployed, such as theorem proving, that are capable of handling analysis of first-order logic properties.

Appendix A

Notation and Terminology in Logic Programs

A *term* is either a variable X or a compound term $f(t_1, \ldots, t_n)$, where f is a function symbol and the t_i are terms. A constant is a 0-ary function symbol [32]. Variables are represented by alphanumeric strings beginning with upper case letters. Constants are represented by alphanumeric strings beginning with lower case letters.

An *atom* is an atomic formula $p(t_1, \ldots, t_n)$, where p is a predicate symbol of n-ary and t_i are terms. Predicates are represented by strings beginning with lower case letters. A *literal* is an atom, called a positive literal, or an atom preceded by *not*, called a negative literal, where *not* is the negation as failure operator [8]. We use $(not)\ p$ to refer to either the positive literal p or the negative literal *not* p of that atom.

A *clause* is an expression of the form $h{:-}\ b_1, \ldots, b_n$, where h is an atom (called the *head* atom) and the b_i are literals (called *body* literals). The symbol :- is used to denote material implication in logic programs so that p:-q means $q \rightarrow p$. A clause is *ground* if it contains no variables. A clause is *definite* if all of its body literals are positive. The *empty clause* is denoted $\square$ and represents the truth value *false*. A *goal clause* is a clause (:-$b_1, \ldots, b_n$) with an empty head. A *normal logic program* is a set of clauses. A *definite logic* program is a program in which all clauses are definite.

A Herbrand *model* I of a logic program Π is a set of ground atoms such that, for each ground instance C of a clause in Π, I satisfies the head of C whenever it satisfies the body of C. A program is consistent if it has at least one model. A model I is *minimal* if it does not strictly include any other model. Definite programs always have a unique minimal model. Normal programs may have one, none, or several minimal models. When there is no unique minimal model, alternative semantics are often provided to single out specific models as the intended model. One widely used semantics is that of stable model semantics [19]. Given a normal logic program Π, the *reduct* of Π with respect to I, denoted Π^I, is the program obtained from the set of all ground clauses in Π by 1) removing all clauses with a negative literal $not\ a$ in its body where $a \in I$ and 2) removing all negative literals from the bodies of the remaining clauses. Π^I is therefore definite and has a single unique definite model. If I is the least Herbrand model of Π^I, then I is said to be a stable model of Π.

Definition 7 (Stable model). *A model I of Π is a stable model if I is the least Herbrand model of Π^I, where Π^I is the definite program $\{A{:-}\ B_1, \ldots, B_n \mid A{:-}\ B_1, \ldots, B_n,\ not\ C_1, \ldots,\ not\ C_n$ is the ground instance of a clause in Π and I does not satisfy any of the $C_i\}$.*

The definition of entailment under stable model semantics is given below.

Definition 8 (Entailment in stable model semantics). *A program Π entails an expression E (under the credulous stable model semantics), denoted $\Pi \models E$, iff E is satisfied in at least one stable model of Π.*

Appendix B

Here, we give the definitions of the EC domain-independent axioms used in our programs.

```
clipped(P1,F,P2,S):-fluent(F), scenario(S),
  position(P1), position(P2),position(P),
  P1 < P,
  P < P2, event(E), happens(E,P,S),
  terminates(E,F,P,S).

holdsAt(F,P2,S) :-fluent(F), scenario(S),
  position(P2), position(P1), P1< P2,
  event(E),
  happens(E,P1,S), initiates(E,F,P1,S),
  not clipped(P1,F,P2,S).

holdsAt(F,P,S) :- fluent(F), scenario(S),
  position(P), initially(F,S), not
  clipped(0,F,P,S).

happens(E,P,S) :-event(E), scenario(S),
  position(P), attempt(E,P,S), not
  impossible(E,P,S).
```

Below are the definitions of the auxiliary predicates used to construct EC pre- and trigger-conditions.

```
holdsAt_Tick(F,P,S) :-
  fluent(F), scenario(S), position(P),
  happens(tick,P,S), holdsAt(F,P,S).

not_holdsAt_Tick(F,P,S) :-
  fluent(F), scenario(S), position(P),
  happens(tick,P,S), not holdsAt(F,P,S).

holdsAt_PrevTick(F,P2,S) :- fluent(F),
  scenario(S),
  position(P2), position(P1), P1> P2,
  happens(tick,P1,S), holdsAt(F,P1,S),
  not occursInBetween(tick,P1,P2,S).

not_holdsAt_PrevTick(F,P2,S) :-
  fluent(F), scenario(S),
  position(P2), position(P1), P1< P2,
  happens(tick,P1,S), not holdsAt(F,P1,S)
  not occursInBetween(tick,P1,P2,S).

nextTickAt(P2,P1,S) :- scenario(S),
  position(P1),
```

```
    position(P2), P1<P2, happens(tick,P2,S),
    not occursInBetween(tick,P1,P2,S).

occursInBetween(E,P1,P2,S):- event(E),
    scenario(S),
    position(P1), position(P2),
    position(P), P1<P, P<P2,
    happens(E,P,S).

occursSince_LastTick(E,P2,S) :-
    event(E), scenario(S),
    position(P1), position(P2), P1<P2,
    happens(tick,P1,S),
    not occursInBetween(tick,P1,P2,S),
    position(P), P1<P, P<P2, happens(E,P,S).
```

ACKNOWLEDGMENTS

The authors acknowledge financial support for this work from the ERC project PBM—FIMBSE (No. 204853).

REFERENCES

[1] D. Alrajeh, "Requirements Elaboration Using Model Checking and Inductive Learning," PhD thesis, Imperial College London, 2010.

[2] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel, "Deriving Non-Zeno Behavior Models from Goal Models Using ILP," *J. Formal Aspects of Computing,* vol. 22, pp. 217-241, 2010.

[3] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel, "Learning Operational Requirements from Goal Models," *Proc. 31st Int'l Conf. Software Eng.,* pp. 265-275, 2009.

[4] D. Alrajeh, O. Ray, A. Russo, and S. Uchitel, "Extracting Requirements from Scenarios with ILP," *Proc. 16th Int'l Conf. Inductive Logic Programming,* pp. 63-77, 2006.

[5] D. Alrajeh, O. Ray, A. Russo, and S. Uchitel, "Using Abduction and Induction for Operational Requirements Elaboration," *J. Applied Logic,* vol. 7, no. 3, pp. 275-288, 2009.

[6] A.I. Anton, "Goal Identification and Refinement in the Specification of Software-Based Information Systems," PhD thesis, Georgia Inst. of Technology, 1997.

[7] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tachella, "NuSMV 2: An Opensource Tool for Symbolic Model Checking," *Proc. 14th Int'l Conf. Computer Aided Verification,* pp. 241-268, 2002.

[8] K. Clark, "Negation as Failure," *Readings in Nonmonotonic Reasoning,* pp. 311-325, Morgan Kaufmann Publishers, 1987.

[9] D. Corapi, A. Russo, and E. Lupu, "Inductive Logic Programming as Abductive Search," *Proc. Technical Comm. 26th Int'l Conf. Logic Programming,* pp. 54-63, 2010.

[10] C. Damas, P. Dupont, B. Lambeau, and A. van Lamsweerde, "Generating Annotated Behavior Models from End-User Scenarios," *IEEE Trans. Software Eng.,* special issue on interaction and state-based modeling, vol. 31, no. 12, pp. 1056-1073, Dec. 2005.

[11] C. Damas, B. Lambeau, and A. van Lamsweerde, "Scenarios, Goals, and State Machines: A Win-Win Partnership for Model Synthesis," *Proc. ACM Int'l Symp. Foundations of Software Eng.,* pp. 197-207, 2006.

[12] A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-Directed Requirements Acquisition," *Science of Computer Programming,* vol. 20, no. 1, pp. 3-50, 1993.

[13] R. Darimont and A. van Lamsweerde, "Formal Refinement Patterns for Goal-Driven Requirements Elaboration," *Proc. Fourth ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 179-190, 1996.

[14] W.P. de Roever and J. Hooman, "Design and Verification in Real-Time Distributed Computing: An Introduction to Compositional Methods," *Proc. IFIP WG6.1 Ninth Int'l Symp. Protocol Specification, Testing and Verification IX,* pp. 37-56, 1990.

[15] A. Fidjeland, W. Luk, and S. Muggleton, "Scalable Acceleration of Inductive Logic Programs," *Proc. IEEE Int'l Conf. Field-Programmable Technology,* pp. 252-259, 2002.

[16] A. Finkelstein and J. Dowell, "A Comedy of Errors: The London Ambulance Service Case Study," *Proc. Eighth Int'l Workshop Software Specification and Design,* pp. 2-4, 1996.

[17] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso, "Specifying and Analyzing Early Requirements in TROPOS," *Requirements Eng.,* vol. 9, no. 2, pp. 132-150, 2004.

[18] A. Fuxman, J. Mylopoulos n, M. Pistore, and P. Traverso, "Model Checking Early Requirements Specifications in Tropos," *Proc. IEEE Fifth Int'l Symp. Requirements Eng.,* pp. 174-181, 2001.

[19] M. Gelfond and V. Lifschitz, "The Stable Model Semantics for Logic Programming," *Proc. Fifth Int'l Conf. Logic Programming,* pp. 1070-1080, 1988.

[20] D. Giannakopoulou, "Model Checking for Concurrent Software Architectures," PhD thesis, Imperial College London, 1999.

[21] D. Giannakopoulou and J. Magee, "Fluent Model Checking for Event-Based Systems," *Proc. 11th ACM SIGSOFT Symp. Foundations Software Eng.,* pp. 257-266, 2003.

[22] M. Jackson, "The World and the Machine," *Proc. 17th Int'l Conf. Software Eng.,* pp. 283-292, 1995.

[23] R.M. Keller, "Formal Verification of Parallel Programs," *Comm. ACM,* vol. 19, no. 7, pp. 371-384, 1976.

[24] R.A. Kowalski and M. Sergot, "A Logic-Based Calculus of Events," *New Generation Computing,* vol. 4, no. 1, pp. 67-95, 1986.

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

[26] J. Kramer, J. Magee, and M. Sloman, "Conic: An Integrated Approach to Distributed Computer Control Systems," *Proc. IEE Computers and Digital Techniques,* vol. 30, pp. 1-10, 1983.

[27] E. Letier, "Reasoning About Agents in Goal-Oriented Requirements Engineering," PhD thesis, Dépt. Ingénierie Informatique, Université Catholique de Louvain, 2001.

[28] E. Letier, "Goal-Oriented Elaboration of Requirements for a Safety Injection Control System," technical report, Département d'Ingénierie Informatique, Université Catholique de Louvain, 2002.

[29] E. Letier, J. Kramer, J. Magee, and S. Uchitel, "Deriving Event-Based Transitions Systems from Goal-Oriented Requirements Models," *Automated Software Eng.,* vol. 15, pp. 175-206, 2008.

[30] E. Letier and A. van Lamsweerde, "Agent-Based Tactics for Goal-Oriented Requirements Elaboration," *Proc. 24th Int'l Conf. Software Eng.,* pp. 83-93, 2002.

[31] E. Letier and A. van Lamsweerde, "Deriving Operational Software Specifications from System Goals," *Proc. 10th ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 119-128, 2002.

[32] J.W. Lloyd, *Foundations of Logic Programming.* Springer, 1984.

[33] J. Ma, A. Russo, K. Broda, and K. Clark, "DARE: A System for Distributed Abductive Reasoning," *Autonomous Agents and Multi-Agent Systems,* vol. 16, no. 3, pp. 271-297, 2008.

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

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

[36] T. Mitchell, *Machine Learning.* McGraw Hill, 1997.

[37] S.H. Muggleton, "Inverse Entailment and Progol," *New Generation Computing,* special issue on inductive logic programming, vol. 13, nos. 3/4, pp. 245-286, 1995.

[38] S.H. Muggleton and L. De Raedt, "Inductive Logic Programming: Theory and Methods," *J. Logic Programming,* vol. 19, no. 20, pp. 629-679, 1994.

[39] J. Mylopoulos, L. Chung, and B.A. Nixon, "Representing and Using Non-Functional Requirements: A Process-Oriented Approach," *IEEE Trans. Software Eng.,* vol. 18, pp. 483-497, 1992.

[40] F. Patrizi, N. Lipoveztky, G. De Giacomo, and H. Geffner, "Computing Infinite Plans for LTL Goals Using a Classical Planner," *Proc. 22nd Int'l Joint Conf. Artificial Intelligence,* pp. 2003-2009, 2011.

[41] O. Ray, "Using Abduction for Induction of Normal Logic Programs," *Proc. Workshop Abduction and Induction in AI and Scientific Modelling,* 2006.

[42] O. Ray, "Nonmonotonic Abductive Inductive Learning," *J. Applied Logic,* vol. 7, no. 3, pp. 329-340, 2009.

[43] O. Ray, K. Broda, and A. Russo, "A Hybrid Abductive Inductive Proof Procedure," *Logic J. Interest Group in Pure and Applied Logic,* vol. 12, no. 5, pp. 371-397, 2004.

[44] A. Rifaut, P. Massonet, J. Molderez, C. Ponsard, P. Stadnik, A. van Lamsweerde, and H. Tran Van, "FAUST: Formal Analysis Using Specification Tools," *Proc. 11th IEEE Int'l Conf. Requirements Eng.*, p. 350, 2003.
[45] C. Rolland, G. Grosz, and R. Kla, "Experience with Goal-Scenario Coupling in Requirements Engineering," *Proc. IEEE Int'l Symp. Requirements Eng.*, pp. 74-81, 1999.
[46] C. Rolland, C. Souveyet, and C.B. Achour, "Guiding Goal Modeling Using Scenarios," *IEEE Trans. Software Eng.*, vol. 24, no. 12, pp. 1055-1071, Dec. 1998.
[47] G. Sibay, S. Uchitel, and V. Braberman, "Existential Live Sequence Charts Revisited," *Proc. 30th Int'l Conf. Software Eng.*, pp. 41-50, 2008.
[48] V.S. Subrahmanian and C. Zaniolo, "Relating Stable Models and AI Planning Domains," *Proc. 12th Int'l Conf. Logic Programming*, pp. 233-247, 1995.
[49] A. Sutcliffe, N.A.M. Maiden, S. Minocha, and D. Manuel, "Supporting Scenario-Based Requirements Engineering," *IEEE Trans. Software Eng.*, vol. 24, no. 12, pp. 1072-1088, Dec. 1998.
[50] S. Uchitel, G. Brunet, and M. Chechik, "Synthesis of Partial Behavior Models from Properties and Scenarios," *IEEE Trans. Software Eng.*, vol. 35, no. 3, pp. 384-406, May/June 2009.
[51] A. van Lamsweerde, "Goal-Oriented Requirements Eng.: A Guided Tour," *Proc. Fifth IEEE Int'l Symp. Requirements Eng.*, pp. 249-262, 2001.
[52] A. van Lamsweerde and L. Willemet, "Inferring Declarative Requirements Specifications from Operational Scenarios," *IEEE Trans. Software Eng.*, vol. 24, no. 12, pp. 1089-1114, Dec. 1998.

Dalal Alrajeh received the PhD degree from Imperial College London in February 2010. She is a research associate on the ERC project "Partial Behavior Modeling: A Foundation for Incremental and Iterative Model-Based Software Engineering" in the Department of Computing, Imperial College London. She has published papers in software engineering and artificial intelligence journals and conferences, including ICSE, FASE, JAL, ILP, and JFAC. She was appointed as the early research career officer for the BCS's Requirements Engineering Specialist Group. For further details, please see http://www.doc.ic.ac.uk/~da04.

Jeff Kramer is the senior dean of Imperial College London and a professor of distributed computing within the Department of Computing. His research interests include behavior analysis, the use of models in requirements elaboration, and architectural approaches to self-organizing software systems. He is a fellow of the Royal Academy of Engineering. In addition, he is a chartered engineer, fellow of the IET, fellow of the BCS, and fellow of the ACM. He is the corecipient of the 2005 ACM SIGSOFT Outstanding Research Award for significant and lasting research contributions to software engineering. He is a coauthor of a recent book on concurrency, coauthor of a previous book on distributed systems and computer networks, and the author of more than 200 journal and conference publications. He has also worked with many industries, including BP, BT, NATS, Fujitsu, Barclays Capital, QinetiQ, Kodak, and Philips, in research collaboration and/or as a consultant. See http://www.doc.ic.ac.uk/~jk for further details and selected publications. He is a member of the IEEE.

Alessandra Russo is a reader in applied computational logic within the Department of Computing at Imperial College London. Her research interests include logic, artificial intelligence, and their applications in software engineering, with a focus on the development of formal frameworks, techniques, and tools, based on abductive and inductive reasoning, for the analysis and management of evolving specifications. She is an author of more than 90 publications in international conferences and journals, one of which was awarded a prize for best application paper. She has served on the program committees of many international conferences. She has been an investigator on various United Kingdom funded research projects, and ITA funded projects in collaboration with the IBM T.J. Watson Research Center. She is also editor-in-chief of the *IET Software* journal. See http://www.doc.ic.ac.uk/~ar3 for further details and a list of publications. She is a member of the IEEE

Sebastian Uchitel is a professor in the Department of Computing, FCEN, University of Buenos Aires, holds a readership at Imperial College London, and is a visiting professor at the Japanese National Institute of Informatics. He was an associate editor of the *IEEE Transactions on Software Engineering* and is currently an associate editor of the *Requirements Engineering Journal*. He was program cochair of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006) and of the 32nd IEEE/ACM International Conference on Software Engineering (ICSE 2010). He has been awarded the Philip Leverhulme Prize, the ERC Starting Grant Award, and an Argentine National Academy of Exact, Physical and Natural Sciences Award. He is a member of the IEEE.