Inferring Operational Requirements from Scenarios and Goal Models Using Inductive Learning

Dalal Alrajeh, Alessandra Russo, Sebastian Uchitel
Department of Computing, Imperial College London
180 Queen's Gate, SW7 2RH, London, UK
{da04, ar3, s.uchitel}@doc.ic.ac.uk

ABSTRACT

Goal orientation is an increasingly recognised Requirements Engineering paradigm. However, integration of goal modelling with operational models remains an open area for which the few techniques that exist are cumbersome and impractical. In particular, the derivation of operational models and operational requirements from goals is a manual and tedious task which is, currently, only partially supported by operationalisation patterns. In this position paper we propose a framework for supporting such tasks by combining model checking and machine learning. As a proof of concept we instantiate the framework to show that progress checks and inductive learning can be used to infer preconditions and hence to support derivation of operational models.

Categories and Subject Descriptors: D.2.1 Software EngineeringRequirement/Specifications [Elicitation methods]; I.2.6 Artificial IntelligenceLearning[Induction]

General Terms: Theory, Verification

Keywords: Goal-oriented requirements engineering, scenarios, inductive inference, Event Calculus, FLTL

1. INTRODUCTION

Goal orientation is an increasingly recognised paradigm for eliciting, elaborating, structuring, specifying, analysing, negotiating, documenting and modifying software requirements [7, 8]. Goals are prescriptive statements of intent whose satisfaction requires the cooperation of agents (or active components) in the software and its environment. Goals are organised in refinement structures; high-level goals are recursively refined into sub-goals until each terminal goal is realisable by some individual agent, in the sense that it is defined in terms of phenomena that are monitored and controlled by the agent [12].

Although goal models naturally support a number of analyses such as obstacle and conflict detection, their declarative nature is not the best fit for validation techniques based on executable models such as graphical animations, simulations, and rapid-prototyping. In addition, they do not naturally support narrative style elicitation techniques, such as those in scenario-based requirements engineering [20]. Moreover, they are not suitable for down-stream analyses that focus on design and implementation issues which are of an operational nature.

To address these issues, the construction of operational event-based models from goal models has been investigated. Such models can be derived automatically from goals expressed in temporal logic. However, because goals are typically expressed in a synchronous, non-interleaving semantic framework while operational models are essentially asynchronous, interleaved semantic frameworks, such derivation can lead to operational models with deadlocks and livelocks [10]. The essence of the problem lies in the fact that goals under-specify system behaviour, requiring certain properties to hold at fixed time points (in this sense they are synchronous [11]). However, it does not specify the sequence of interleaved events that make these properties true.

It is possible to derive an operational model without deadlocks and livelocks if the goal model has been fully refined into operational requirements, i.e. preconditions and triggering conditions have been identified for all events [10]. However, refinement of operational requirements from goals is a tedious manual task that is only partially supported by operationalisation patterns. Additionally, these patterns are only defined for certain types of goals [13].

The aim of the work described in this position paper is to support the elicitation of operational requirements such as preconditions and triggers and, more generally, to aid the construction of operational models from goal models. Our approach combines scenarios, model checking and machine learning into a framework that supports iterative elaboration of goal models towards fully operationalised ones. The overall approach (Figure 1) consists of three conceptual phases; *scenario generation* where a negative scenario leading to deadlock or livelock is generated fully automatically by the LTSA model checker, *scenario elicitation* where an engineer produces positive scenarios that exemplify the correct behaviour of the system under similar circumstances and *inductive learning* where the inductive learning tool, Progol5, is used to compute preconditions and triggers that would avoid the negative scenario and be consistent with the positive ones.

This paper is organised as follows: Section 2 briefly presents Labelled Transition Systems, Fluent Linear Temporal Logic, Event Calculus and Learning Event Calculus programs, as well as a running example, a Mine Pump Control System.

Section 3 presents our approach and Section 4 presents a discussion and related work. Finally, we conclude in Section 5 with future work.

2. BACKGROUND

We use a simplified version of the mine pump control system described in [6] throughout this paper as a running example. The system concerns a mine that tends to fill up with water. The water must not rise above a specified level otherwise the miners would drown. When the water level starts to rise the pump can be turned on to lower it. However, in order to prevent explosions, the pump must not be turned on if there is a high level of methane in the mine.

2.1 Labelled Transition Systems

Labelled Transition Systems (LTS) are used to model a system as a set of concurrent components [14]. Each component is characterised by a set of states and transitions between these states. Each transition is labelled with an event describing an observable behaviour of the component. The global system behaviour is given by the parallel composition of LTSs, which results in a LTS that interleaves the behaviour of all components but forces synchronisation on shared events.

Modelling discrete-time systems in LTS requires explicit reference to an event *tick*. This event signals the regular ticks of the system clock in which each timed-process is assumed to synchronise. When modelling discrete-time systems in this way, it is important to check that the defined LTS model does not indefinitely prevent time from progressing. This can be checked automatically by the tool LTSA (Labelled Transition Systems Analyser) through a progress check. This ensures that the LTS model does not deadlock and that a tick event occurs infinitely often in any infinite execution trace [14]. We refer to this property as the time progress property (TP).

2.2 Fluent Linear Temporal Logic

Fluent Linear Temporal Logic (FLTL) is a Linear Temporal Logic for expressing state-based temporal logic properties for an event-based operational model [5]. As in Linear Temporal Logic, FLTL assertions are constructed from a set of atomic propositions, the standard Boolean operators $\neg, \vee, \wedge, \rightarrow$ and $\leftrightarrow$ and the temporal operators X (next), $\Box$ (always), $\Diamond$ (eventually), U (until), and W (awaits). The set of atomic propositions in FLTL is the set of *fluents*, a proposition whose truth value changes over time. Given an FLTL language, each fluent Fl in the language is defined in terms of an initiating (I_{Fl}) and terminating (T_{Fl}) set of events and an initial value, $initially_{Fl}$; we assume that fluents are initially false unless stated otherwise. The occurrence of an event in I_{Fl} (resp. T_{Fl}) makes the fluent Fl true (resp. false). We define a fluent as follows: fluent Fl = $<I_{Fl}, T_{Fl}>$ initially $initially_{Fl}$. In addition, every event e in the alphabet A of a LTS model defines an implicit fluent: fluent e = <{e},A-{e}> initially false.

Let Φ denote the set of fluents in a given FLTL language. An interpretation is an infinite history $h : Nat \rightarrow 2^{\Phi}$ which defines for each position $i \in Nat$ the set of fluents that are true at that position. A fluent Fl is said to be true at a given position i if and only if either of the following conditions holds:

- Fl initially holds and no terminating events has occurred since ($Initially_{Fl} \wedge (\forall k \in Nat. 0 \leq k \leq i, e_k \notin T_{Fl})$).
- Some initiating event has occurred before position i and no terminating event has occurred since ($\exists j \in Nat : ((j \leq i) \wedge (e_j \in I_{Fl}) \wedge (\forall k \in Nat\ j < k \leq i, e_k \notin T_{Fl})$).

Note that from the above definition, events have an immediate effect on the values of fluents.

The notation (h,i) $\models$ P is used to denote that the expression P is true at position i in the history h. Assuming P and Q are two FLTL expressions, h is a given history, and i a position in the history, the truth of an expression P is based on the standard semantic meaning of the Boolean operators and on the following semantic definitions of the temporal logic operators [15][1]:

- (h,i) $\models$ X P *iff* (h,i+1) $\models$ P.
- (h,i) $\models$ $\Box$ P *iff* (h,j) $\models$ P *for all* j $\geq$ i
- (h,i) $\models$ P U Q *iff* (h,j) $\models$ $\Diamond$ Q *for any* j $\geq$ i *and* (h,k) $\models$ P *for all* k *s.t.* i $\leq$ k $<$ j .

Note that the definition of position i in history h, hence the semantics of the temporal operators, can have different interpretations. In an asynchronous interpretation of FLTL, position i is the one after the ith event, while in a synchronous interpretation, i would be interpreted as after the ith tick of the global clock. For instance, an asynchronous interpretation makes the expression '$\Box$P' mean P is true after the occurrence of every event in a history, while in a synchronous interpretation '$\Box$P' would mean P is true after the occurrence of every tick event (assuming tick models the global clock). A technique for translating FLTL expressions interpreted synchronously into equivalent asynchronous FLTL expressions has been presented in [11]. According to this translation, the synchronous expression '$\Box$P' becomes '$\Box$(tick $\rightarrow$ P)'.

Returning to the mine pump example, its goals can be formulated with the following fluents and synchronous FLTL expressions:

fluent PumpOn = <switchPumpOn,switchPumpOff>,
fluent Methane = <methaneAppears,methaneLeaves>,
PumpOnWhenHighWaterAndNoMethane =
$\Box$ (HighWater && $\neg$Methane $\rightarrow$ X PumpOn),
PumpOffWhenLowWater = $\Box$ (LowWater $\rightarrow$ X $\neg$PumpOn),
PumpOffWhenMethane = $\Box$ (Methane $\rightarrow$ X $\neg$PumpOn),
RaiseWaterLevelPre[i] = $\Box$ (PumpOn $\rightarrow$ X$\neg$raiseWaterLevel[i])[2].

2.3 Event Calculus

Event Calculus (EC) is a logic-based formalism for representing and reasoning about event-based systems. As in FLTL, the basic idea in standard Event Calculus is that a fluent, a property whose value varies over time, is true (resp. false) at a particular time-point if an event has occurred at an earlier time-point that initiates (resp. terminates) it, and no terminating (resp. initiating) events have occurred in the meantime [16].

EC is a specialised three-sorted first-order logic including a sort A of events (e_1,e_2,..), a sort F of fluents (f_1,f_2..), and a sort T of time-points (t_1,t_2,..) isomorphic to the set of

[1]W is defined in terms of U and $\Box$; $\Diamond$ is equivalent to $\neg\Box\neg$.
[2]This is a precondition describing how the water level changes using the following indexed set of asynchronous FLTL expression

non-negative integers. The basic predicates in EC are: *HoldsAt* $\subseteq F \times T$, *Happens* $\subseteq A \times T$, *Initiates* $\subseteq A \times F \times T$ and *Terminates* $\subseteq A \times F \times T$. The predicate $HoldsAt(f,t)$ indicates that a fluent f is true at a time-point t. The predicate $Happens(e,t)$ means that the event e occurs at time-point t. The predicates *Initiates(e,f,t)* (resp.*Terminates(e,f,t)*) denotes that if an event e occurs at time-point t, then e causes the fluent f to be true (resp. false) immediately afterwards. For example, the statement: 'The pump is initially off, the switch is turned on at time 2, and turning the switch on activates the pump' can be expressed in EC as follows[3]: $\neg$*HoldsAt(PumpOn,0), Happens(switchPumpOn,2),* and *Initiates(switchPumpOn,PumpOn,t).*

To capture the reasoning process underlying the effects of events on fluents, an EC program is assumed to include a set of core axioms. These are domain-independent axioms that describe the default-persistence of the truth values of fluents. For a full definition of these axioms, the reader is referred to [16]. In addition to the core axioms, an EC program consists of a set of domain-dependent axioms that describe casual relations between events and fluents in terms of $Initiates$ and $Terminates$ rules. The EC core axioms and the domain-dependent axioms are used to derive information about the truth value of fluents at future time-points. For instance from the above example it is possible to derive that Pump is on at time-point 4, i.e. $HoldsAt(PumpOn, 4)$.

2.4 Learning Event Calculus Programs

An inductive learning task is the process of deriving a hypothesis H from given partial background knowledge B and evidence E which is consistent with B and, together with B, explains E[4]. Formally, given a partial background knowledge B and evidence E, the process tries to find a theory H consistent with B such that $B \cup H \models E$ and $B \cup H \not\models False$.

Inductive logic programs are built from Horn clauses. A Horn clause is a disjunction of literals with at most one positive. In standard inductive learning programming (ILP) problems, the background knowledge B is a conjunction of Horn clauses and the evidence E is a set of ground facts. Integrity constraints I can also be considered as part of an inductive learning task and are used to further refine the learning process. They are Horn clauses with no positive literals.

Within the context of leaning EC programs, the background knowledge, denoted by T_{EC}, consists of the EC core axioms (T_{DIA}), domain-dependent axioms (T_{DDA}) and static axioms (T_{Static}), i.e. $T_{EC} = T_{DIA} \cup T_{DDA} \cup T_{Static}$ [17]. The static axioms include a narrative (T_{Nar}) given by a set of ground facts describing event occurrences, the initial state also given as ground facts and any auxiliary axiom. The evidence is a set of positive ground facts describing the truth value of fluents at specific time-points.

In the above context, learning EC programs means computing additional domain-dependent axioms that are consistent with the given background knowledge T_{EC} and that together with T_{EC} explain the evidence E. In formal terms, the learning problem can be defined as: given a set of integrity constraints I, background knowledge T_{EC} and evidence E, find a theory H such that $T_{EC} \cup H \cup I \models E$ and $T_{EC} \cup H \cup I \not\models False$.

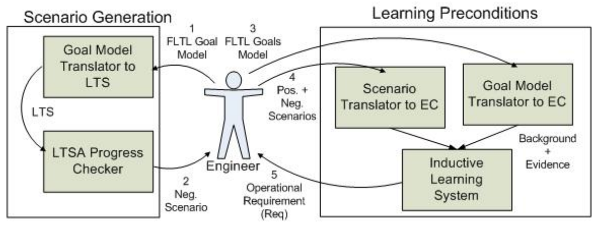

Figure 1: Inference of operational requirements from goals and scenarios

3. NEW APPROACH

This section describes a semi-automated approach for inferring operational requirements that refine a given goal model. Figure 1 illustrates this approach. A goal model is first transformed into a LTS model which is then checked for time progress. If progress is violated, an example of such violation is produced for the engineer. The engineer elaborates this example of negative behaviour and produces a number of alternative acceptable scenarios. The resulting scenarios are integrated and an inductive learning technique is then used to infer new operational requirements.

3.1 Problem Statement

Let G_s be a consistent goal model composed of synchronous FLTL assertions modelling high-level goals and operational requirements. From G_s, it is possible to construct a semantically equivalent set of asynchronous FLTL assertions G_a [11]. In addition, these asynchronous assertions can be used to derive a LTS model M such that M satisfies G_a:

$$M \models G_a.$$

However, M may not be a suitable operational model as it may allow time not to progress. In other words, M may allow traces in which tick never occurs because the environment or the system is continuously performing other events. For example, consider the synchronous goal g_s, $\Box$(Methane $\rightarrow$ X $\neg$PumpOn). Its asynchronous version g_a, $\Box$(tick $\rightarrow$ (Methane $\rightarrow$ X ($\neg$tick W (tick && $\neg$ PumpOn)))) only constrains the occurrence of ticks. Hence, it permits traces such as (switchPumpOn, switchPumpOff, switchPumpOn, switchPumpOff...) and (switchPumpOn, switchPumpOn...) which do not allow time to progress, violating the TP property. These traces vacuously satisfy g_a as tick never occurs.

The problem with the example given above is that there is no information as to when the pump *must* be switched on, i.e. (the triggering condition for the event switchPumpOn), and when it *should not* be turned on (i.e. the precondition for the event switchPumpOn). The goal g_a only provides information as to when a tick of the clock is reasonable. In fact, the model does not constrain the occurrence of any event except tick. Hence, although $M \models G_a$, M may have progress problems if suitable preconditions and triggering conditions are not captured in G_a. *The aim of this approach is to support eliciting the missing operational requirements.*

In this section we instantiate the framework described in Figure 1 to learn preconditions. This involves, in particular,

[3]Free variables are assumed to be universally quantified unless specified.

[4]We can restrict ourselves to consider E to be just a set of positive examples, since negative examples can be defined as integrity constraints

the introduction of a standard assumption in reactive system modelling which is that of maximal progress (MP) [2]. It is assumed that the reactive system is fast enough to perform all necessary computation before the next input from the environment. In this case, we are interested in assuming that the system will perform all it needs to do before the next global clock event. In a sense, the assumption is that the system is greedy and will try to perform all the operations it can while not violating any goals. If in doing so time does not go forward, this implies that a precondition to some action is missing. For instance, in the example above, setting ¬PumpOn && ¬Methane as a precondition for switchPumpOn would avoid both traces mentioned above. Note that our approach also takes into account the existence of additional assumptions, A, that may be relevant to the goal oriented modelling framework being used. For instance, the KAOS framework assumes a fluent can only change value once per time unit. This assumption, as we show in Section 3.2, can introduce deadlocks which are a particular kind of progress violation.

Let G_s be a goal model containing synchronous FLTL goals and operational requirements, and let G_a be its translation into asynchronous FLTL. Let F be a set of fluent definitions that define the fluents in G_a. We can now formulate the learning problem for preconditions as follows: Given,

$$G_a \wedge F \wedge MP \wedge A \not\models TP. \tag{1}$$

our aim is to derive a set of required preconditions Req such that:

$$Req \wedge G_a \wedge F \wedge MP \wedge A \models TP, \tag{2}$$

by means of an iterative process of learning from positive and negative scenarios. Learnt required preconditions, *Req*, when added to the initial goal assertions will have the effect of disallowing sequences of events that previously led to the violation of the TP property. The final set of requirements *Req* together with the original goals G_a should then allow time to progress while still satisfying the MP property.

Note that the above problem formulation can be adapted to learn other operational requirements such as trigger conditions. However, for these cases other assumptions instead of maximum progress may be needed. In particular, for learning triggers a "maximal laziness" assumption would be needed to highlight scenarios in which if the system is not forced (triggered) to perform an action, then an undesirable situation, such as a goal violation, will occur.

3.2 Scenario Generation

The first phase of our approach consists of generating scenarios that exemplify the problems associated with lack of operational requirements. More specifically, this phase generates traces that exemplify Equation 1.

The input to this phase is a set of goals, a set of operational requirements (if available) and assumptions of the goal modelling framework. Goals are described in synchronous FLTL and include the definition of the fluents in terms of initiating and terminating events. Operational requirements are described by associating fluent expressions with operations in the style of KAOS [13]. For instance, the operational requirements for event switchPumpOn might be: **Operation** switchPumpOn **Trigger:** HighWater **Precondition:** ¬Methane ∧ ¬PumpOn. Finally, assumptions are given as synchronous or asynchronous FLTL assertions depending on the modelling framework used.

The construction of an operational model requires a set of asynchronous FLTL formulae. Goals can be translated to asynchronous FLTL by following the translation described in [11]. The asynchronous FLTL assertion for the trigger $Trig$ of an event e can be constructed according to the pattern $\square(\mathsf{Trig}\rightarrow \mathsf{Xe})$, while the asynchronous FLTL assertion for the precondition Pre of an event e can be constructed using $\square(\neg\ \mathsf{Pre} \rightarrow \mathsf{X}\neg\mathsf{e})$. The resulting asynchronous FLTL theory can be used by the LTSA tool to produce a LTS model that satisfies the goal model.

In our Pump example, we assume G_s to be the set of goals defined in Section 2.2 and A to restrict, as in KAOS, fluents from changing value more than once per tick. The output of LTSA is as follows:

```
Trace to terminal set of states
tick
methaneAppears                METHANE
switchPumpOn                  METHANE && PUMPON
raiseWaterLevel.0             METHANE && PUMPON
```

This trace shows a sequence of events that leads to deadlock and represents an example of why Equation 2 does not hold. The capitalised text on the right column indicates the fluents that are true after the occurrence of the event on the left. The cause of deadlock is that all events are constrained from occurring due to some goal or assumption; the event tick is constrained from occurring because the pump is on but PumpOffWhenLowWater requires the pump to be off at the next tick. Nonetheless, the assumption on changes of fluent values in one time unit restricts the pump from being turned off and methane from leaving. Preconditions on switchPumpOn and methaneAppears prevent these actions from occurring if the pump is on and there is methane present respectively. Finally, the water level is prevented from rising when the pump is on.

On inspecting the above trace, the engineer may see as anomalous the fact that the pump is switched on when methane is present, situation which is not prevented by the synchronous goal PumpOnWhenHighWaterNoMethane. Alternatively, the engineer may identify that the pump should not have been switched on because at the end of the time unit, the goal PumpOffWhenLowWater requires the pump to be off. In any case, engineers can use the trace produced in a fully automated way to locate such specific negative scenarios. We believe this is an intuitive task that can to be performed manually, in particular because the negative scenario will always be a prefix of the trace fed back by the progress check.

We define a negative scenario as a sequence of events we, where w is a sequence of events and e is an *undesired* event. The intended meaning of ωe is that if the system exhibits w, then e should not occur next . In the following we assume that the negative scenario identified by the engineer is as follows: tick, methaneAppears, switchPumpOn.

Once a negative scenario has been identified, a scenario that exemplifies the correct behaviour of the system under similar circumstances needs to be given. More precisely, if we is the negative scenario identified, the positive scenario is required to be of the form ww', where w' is a sequence of events that contains e. Furthermore, this positive scenario must be consistent with all existing goals and assumptions.

In our example, a positive scenario showing the pump being switched on once the water level has surpassed the low water threshold (e.g. level 2) may be: tick, mathaneAppears, tick, raiseWaterLevel.0, methaneLeaves, tick, raiseWaterLevel.1, tick, raiseWaterlevel.2, tick, switchPumpOn.

Note to guarantee that the positive scenario is consistent with existing goals and assumptions, the engineer can walk through or animate the LTS model on which the progress check was performed. This model is guaranteed to satisfy all goals and assumptions, hence any trace generated by it will be adequate. Finally, note that although in this case the progress violation is a deadlock, the same procedure applies to time locks.

3.3 Learning Preconditions

This section describes the learning stage of our approach in which operational requirements are learnt from scenarios generated in the previous phase. As before, we consider the case of learning preconditions, although the approach is applicable to event triggers as discussed in Section 3.1.

3.3.1 Overview

As described in Section 2.3, EC is a framework for reasoning about action and change. The framework assumes that events occur at certain time points and, consequently, fluents may change truth value. When learning EC programs using ILP, the aim is to learn a set of axioms, H, that describes (or prescribes) the rules that govern the relation between events occurring and changes to the truth values of fluents. The static axioms, particularly the narrative T_{Nar} and the evidence E, play a crucial role in the learning process. The learning attempts to explain the relation between the narrative and the evidence.

We can instantiate the inductive learning task described in Subsection 2.4 for the problem statement given in Section 3.1 by having the set of goals G_a, MP , and A as integrity constraints I, the fluent definitions F as T_{DDA}, the narrative derived from positive and negative scenarios as T_{Nar} and the fluent trace derived from positive and negative scenarios as evidence E.

A key to the correctness and utility of the result provided by this learning setting is how the scenarios generated as in the previous section are used to construct the narrative and evidence. Furthermore, the formalisation of the narrative and evidence depends on the form of assertions that are to be learnt(e.g triggers or preconditions).

In this paper, we aim to learn preconditions for an event e of the asynchronous FLTL form $\Box(\neg \textsf{Pre} \rightarrow \textsf{X} \neg \textsf{e})$ where $\textsf{Pre} = \bigwedge_i (\neg)(\textsf{f}_i)$. These preconditions are formalised in EC as $Impossible(e,t) \leftarrow \bigwedge_i (\neg)HoldsAt(f_i,t)$. Consequently, the narrative and evidence must be formalised such that a relation between the impossibility of an event occurring and the values of fluents can be explained. The EC core axioms given in [16] provide an axiomatization which essentially defines *Happens* as follows: *Happens(e,t)* $\leftrightarrow$ *Performs(e,t)* $\wedge\neg$ *Impossible(e,t)*, where the predicate *Performs* is interpreted as "e is attempted to be performed". The above definition of *Happens(e,t)* can therefore be understood as: if an attempt to perform e is made and it is actually possible (or permitted) for it to occur then e happens.

In the next subsection we show how this axiomatization is exploited to learn assertions with *Impossible* predicates by formalising the narrative using *Performs*, fluent traces with *HoldsAt*, and fluent definitions in terms of the predicates *Initiates/Terminates*.

3.3.2 Narratives and Evidence

In an EC learning program, a narrative is a single sequence of events. Hence, positive and negative scenarios need to be integrated. However, one needs to ensure that no information is lost on what is considered good and bad behaviour. This is captured in the way the fluent trace is constructed.

Suppose we have a negative scenario $S_N = \omega e$, and a positive scenario $S_P = \omega\omega'$ where ω and ω' are sequences of events and e is an undesired event. We define the result of integrating S_N and S_P as the sequence of events $N = \omega e \omega'$. For instance, from our running example, the negative and positive scenarios result in the following narrative: tick, methaneAppears, switchPumpOn, tick, raiseWaterLevel.0, methaneLeaves, tick, raiseWaterLevel.1, tick, raiseWaterLevel.2, tick, switchPumpOn where $\omega =$\{tick, methaneAppears\}, $e =$\{switchPumpOn\} and $\omega' =$ \{tick,raiseWaterLevel.0, methaneLeaves, tick, raiseWaterLevel.1, tick, raiseWaterLevel.2, tick, switchPumpOn\}.

The fluent trace is built based on the narrative and the definition of fluents. The first set of fluent values of the fluent trace, i.e. those corresponding to time-point $t = 0$, is given by the initial value of fluents as defined in F. Hence, for the mine pump, the initial values of fluents PumpOn, Methane, HighWater and LowWater are false, false, false and true respectively. The set of fluent values for a time-point $t > 0$ is then computed based on the changes provoked by the event occurring at $t-1$ and on the set of fluent values of time-point $t-1$. In other words, the truth value of the fluent f at t will be true, false or the same as its value at $t-1$ depending on whether the event occurring at $t-1$ in the narrative is, according to the fluent definition of f, initiating, terminating or neither. The exception to this rule is for the fluent values at time point $t = |S_N|$ where $|S_N|$ is the number of events in the negative scenario, i.e. for the event in the narrative that represents undesired behaviour. For this time-point, the set of fluents at t is defined to be exactly as the set at $|S_N| - 1$.

The above construction effectively builds a fluent trace in which the occurrence of the undesired event is ignored in terms of its impact in fluent values. In our running example it is the switchPumpOn after the first tick which is considered as undesirable and consequently ignored for the computation of fluent values at $t = 3$. This leads to the following narrative and fluent trace:

t	Narrative	Fluent trace
0	tick	LowWater && ¬Methane && ¬PumpOn...
1	methaneAppears	LowWater && Methane && ¬PumpOn...
2	switchPumpOn	LowWater && Methane && ¬PumpOn...
3	tick	LowWater && Methane && ¬PumpOn...
4	raiseWaterLevel.0	LowWater && Methane && ¬PumpOn...
.	.	.
.	.	.
.	.	.
11	switchPumpOn	¬LowWater && ¬Methane && ¬PumpOn...
12		¬LowWater && ¬Methane && PumpOn...

In conclusion, the construction of fluent traces is not the trace derived merely from the narratives and fluent definitions, but is a variation to encode undesirable events.

Returning to our main objective, learning preconditions of the form $Impossible(a,t) \leftarrow \bigwedge_i (\neg)HoldsAt(f_i,t)$ and given the axiomatization of *Happens(e,t)* $\leftrightarrow$ *Performs(e,t)* $\wedge$ $\neg$ *Impossible(e,t)*, we formalise the narrative and fluent trace in terms of *Performs* and *HoldsAt* respectively. In our running example, the narrative is formalised with the following set of axioms: \{*Performs(tick, 0)*, *Performs(methaneAppears, 1)*, *Performs(switchPumpOn, 2)*, *Performs(tick, 3)*, ..., *Per-*

forms (switchPumpOn, 11)}, while the fluent trace is formalised with the following set of axioms: {*HoldsAt(LowWater, 0)* $\wedge$ $\neg$ *HoldsAt(Methane, 0)* $\wedge$ $\neg$ *HoldsAt(PumpOn, 0), ..., HoldsAt(LowWater, 1)* $\wedge$ *HoldsAt(Methane, 1)* $\wedge$ $\neg$ *HoldsAt(PumpOn,1), ...,* $\neg$ *HoldsAt(LowWater, 12)* $\wedge$ $\neg$*HoldsAt (Methane, 12),* $\wedge$ *HoldsAt(PumpOn, 12), ...*}.

3.3.3 Computing preconditions

Given the above formalisation, the learning algorithm finds an event e corresponding to the undesired behaviour that is (attempted to be) performed (*Performs(e,*$|S_N|$*−1)*) and has no effect on the fluent trace at time point $|S_N|$. Moreover, it finds another occurrence *Performs(e,t)* that changes the fluent value at the consecutive time-point. From this, the learning algorithm infers that it cannot be the case that $Happens(e, |S_N| - 1)$. Consequently, under conditions holding at time-point $|S_N| - 1$, it is impossible for e to happen, i.e. $Impossible(e, |S_N| - 1)$. However, it does not infer $Impossible(e, t)$. In our running example, the learning algorithm tries to infer an explanation as to why *Impossible(switchPumpOn,3)* and not *Impossible(swithPumpOn,11)*.

To allow the learning algorithm to relate *Happens* predicates with *HoldAt* predicates, and hence perform the inference described above, we need to provide as part of the EC program the relation between fluents and events as given by the fluent definitions for the FLTL goal model. The translation can be described as: given an FLTL fluent definition $Fl = \langle I_{Fl}, T_{Fl} \rangle$, sentences of the form *Initiates*(e_I*,Fl,t)* and *Terminates* (e_T*,Fl,t)* are generated for every event $e_I \in I_{Fl}$ and event $e_T \in T_{Fl}$. Implicit FLTL fluents defined by events are also translated in a similar way. The set of these *Initiates* and *Terminates* ground predicates generated constitutes the domain dependent theory T_{DDA} used by the learning system with the core axioms to generate H.

The actual learning is provided by the Progol5 system [18] which, in addition to the integrity constraints I, the background knowledge T_{EC}, and the evidence E, requires a language bias that guides Progol5's search through the hypotheses space for assertions of the form $Impossible(e, t) \leftarrow \bigwedge_i (\neg) HoldsAt(f_i, t)$ that explain E. Although the inductive learning algorithm allows us to learn assertions of that exact form, a weaker form of preconditions is computed. This is due to limitations introduced by Progol5 when dealing with negation as failure [18]. Instead of learning assertions that describe when it is impossible for an event to occur, the system learns when it is possible for an event to occur , where *Possible* means an event e is possible to occur at time-point t. An example hypothesis that is learnt in the above case is:

Possible(switchPumpOn,t)$\leftarrow$ $\neg$*HoldsAt(Methane,t).*

However, other tools have been introduced to handle negation as failure. For instance, the system HAIL (Hybrid Abductive Inductive Learning) is capable of learning hypotheses where the head of a clause appears negated in the background [19]. It would thereby learn hypotheses of the form:

Impossible(switchPumpOn,t)$\leftarrow$ *HoldsAt(Methane,t),*

which can be translated back into the asynchronous FLTL formula $\Box$ (Methane $\rightarrow$ X $\neg$ switchPumpOn)[5].

Once a precondition has been learnt, it can be added to the goal model and the learning process can be restarted generating new scenarios.

[5]This corresponds to the FLTL syntax for defining preconditions by assuming the condition as $\neg$($\neg$ Methane).

4. DISCUSSION AND RELATED WORK

This paper advocates an approach for deriving operational requirements from goal models based on scenario generation through model checking and inductive learning. As a proof of concept, the paper describes in detail the case of learning preconditions for KAOS models. We believe that other operational requirements could also be computed by suitable tailoring of the same approach. For instance, trigger conditions could be learnt by providing the learning system with appropriate language bias that expresses the performance of an event as conditioned to (a conjunction of) fluents. In the mine pump case study, a trigger condition of the form: 'switch the pump on when the water has reached a high level' could be learnt by considering a language bias where the predicate *Perform* is conditioned by (a conjunction of) predicates *HoldsAt* so allowing the computation of rules of the form: *Performs(switchPumpOn,t)* $\leftarrow$ $\neg$ *HoldsAt(PumpOn,t)* $\wedge$ *HoldsAt(HighWater,t).*

A formal underpinning of the 'termination' of the iterative learning cycle proposed needs to be formulated. It is still unclear what guarantees exist to ensure that by iteratively adding learnt preconditions (or operational requirements in general) to the goal model the engineer will eventually obtain a fully operationalised model with no progress violations. We believe that to guarantee some notion of termination, the approach will need to be extended to learn triggers and also to learn more general forms of preconditions, since at the moment preconditions are restricted to conjunctive normal form.

Key to the approach is the use of properties that can be automatically verified and for which violations can prompt the identification of incomplete aspects of a requirements specification. Our approach exploits the semantic gap between synchronous non-interleaved goal models and asynchronous operational models and also the maximal progress assumption to generate traces that violate progress and that can drive forward the refinement of a goal model.

Note that although the methodology uses different formalisms as FLTL, EC, and ILP, the engineer only deals with FLTL assertions and scenarios. Nonetheless, the approach makes use of these formalisms to exploit existing tools.

Various examples of application of learning techniques to software engineering have been proposed in the literature [1, 21, 4, 3]. Among these, the most relevant is the work described in [9], which proposes an approach for extracting goal requirements, as temporal formulae, from single scenarios using explanation-based learning. The scenarios are single positive scenarios provided by the stakeholder and tailored manually to improve the output of the learning; the explanation-based learning is used to support the generation of new goals that explain the given scenario. These goals are then added to the given KAOS goal structure to which the full KOAS methodology of goal decomposition and obstacle detection is applied.

Our work differs from that proposed in [9] not only in the type of learning technique adopted, which is inductive logic programming instead of explanation-based learning but also in the overall objective of the approach. In [9] the learning process is to support the elicitation of arbitrary requirements from user provided scenarios, whereas our approach is much more targeted. We aim to learn a specific kind of requirements, those on operations that the system provides. In addition, in our approach the starting point for learning is an automatically-generated scenario which exemplifies the consequences of having a partially-refined goal model. We believe that, although the more focused application of machine learning is less general than that of [9], it will allow for increased automation in learning requirements as well as less and simpler forms of user intervention.

5. CONCLUSION

This paper presents an approach for learning operational requirements from positive and negative scenarios in order to support the operationalisation process of goal models. Up to now this has been only partially supported by operationalisation patterns [13], which must be applied manually. This represents a tedious task that may lead to over-specification.

The approach deploys a well established inductive logic programming system that facilitates the computation of hypotheses that together with a given background knowledge can explain the positive but not the negative observations.

The overall operationalisation process advocated in this paper builds upon existing work by some of the authors on automated generation of LTS models from goal models [10]. These LTS models, although consistent with the given goal model, can exhibit progress violations due to lack of operational requirements. By identifying these violations as negative scenarios and eliciting positive scenarios that avoid such progress problems, new operational requirements can be inferred automatically.

We believe the approach is promising and we show that it can be applied specifically to learning preconditions for operations described in goal models. We have validated precondition learning through the Mine Pump case study by comparing learnt preconditions with the preconditions that would have been computed manually by means of the operationalisation patterns.

Ongoing and future work involves developing the formal underpinning to characterise the termination of the iterative learning cycle, extending the work to learning triggers and generalised forms of preconditions, automating the process of the formalisation presented and validating the approach with other case studies.

6. ACKNOWLEDGEMENTS

We gratefully acknowledge the support of the EPSRC grants GR/S03270/01 and EP/C541133/1

7. REFERENCES

[1] D. Barstow. Artificial intelligence and software engineering. In *Proceeding of the 9th International Conference on Software Engineering*, Oct. 1987.

[2] G. Berry and L. Cosserat. The esterel synchronous programming language and its mathematical semantics. *Lecture Notes in Computer Science*, 197:389–448, 1984.

[3] J. F. Bowring, J. M. Rehg, and M. J. Harold. Active learning for automatic classification of software behaviour. In *Proceeding of the Int. Symp. on Software Testing and Analysis*, July 2004.

[4] A. D. Garcez, A. Russo, B. Nuseibeh, and J. Kramer. Combining abductive reasoning and inductive learning to evolve requirements specifications. *IEE Proceedings-Software*, 150(1):25–38, Feb. 2003.

[5] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *Proceeding of the 9th ACM European Software Engineering Conference*, pages 257–266, Sept. 2003.

[6] J. Kramer, J. Magee, and M. Sloman. Conic: An integrated approach to distributed computer control systems. In *IEE Proceedings, Part E 130*, pages 1–10, Jan. 1983.

[7] A. V. Lamsweerde. Requirements engineering in the year 00: A research perspective. In *Proceedings of the 22nd Int. Conference on Software Engineering*, pages 5–19. ACM press, June 2000.

[8] A. V. Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of the 5th IEEE Int. Symposium on Requirements Engineering*, pages 249–263, Aug. 2001.

[9] A. V. Lamsweerde and L. Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE Transactions on Software Engineering*, 24(12):1089–1114, 1998.

[10] E. Letier, J. Kramer, J. Magee, and S. Uchitel. Deriving event-based transitions systems from goal-oriented requirements models.

[11] E. Letier, J. Kramer, J. Magee, and S. Uchitel. Fluent temporal logic for discrete-time in event-based models. In *Proceedings of the 10th European Software Engineering Conference*, pages 70–79, Sept. 2005.

[12] E. Letier and A. V. Lamsweerde. Agent-based tactics for goal-oriented requirements elaboration. In *Proceedings of the 24th Int. Conference on Software Engineering*, pages 83–93, May 2002.

[13] E. Letier and A. V. Lamsweerde. Deriving operation software specifications from system goals. In *Proceeding of the 10th ACM SIGSOFT Symp. on Foundation software engineering*, pages 119–128, Charleston, November 2002.

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

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

[16] R. Miller and M. Shanahan. Some alternative formulation of event calculus. *Computer Science; Computational Logic; Logic programming and Beyond*, 2408, 2002.

[17] S. Moyle. Learning about action and change: An inductive logic programming approach. MSc thesis, Computing Laboratory, University of Oxford, 1998.

[18] S. Muggleton. Inverse entailment and progol5. *New Generation Computing*, 13:245–286, 1995.

[19] O. Ray, K. Broda, and A. Russo. A hybrid abductive inductive learning proof procedure. *Journal of the IGPL*, 12(5):371–397, 2004.

[20] A. Sutcliffe, N. A. M. Maiden, S. Minocha, and D. Manuel. Supporting scenario-based requirements engineering. *IEEE Transactions on Software Engineering*, 24:1072–1088, 1998.

[21] D. Zhang and J. Tsai. Machine learning and software engineering. *Software Quality Journal*, 11:87–119, 2003.