Extracting Requirements from Scenarios with ILP

Dalal Alrajeh[1], Oliver Ray[1,2], Alessandra Russo[1], and Sebastian Uchitel[1,3]

[1] Imperial College London
{da04,or,ar3,su2}@doc.ic.ac.uk
[2] University of Cyprus
oliver@cs.ucy.ac.cy
[3] University of Buenos Aires/CONICET
s.uchitel@dc.uba.ar

Abstract. Requirements Engineering involves the *elicitation* of high-level stakeholder goals and their *refinement* into operational system requirements. A key difficulty is that stakeholders typically convey their goals indirectly through intuitive narrative-style scenarios of desirable and undesirable system behaviour, whereas goal refinement methods usually require goals to be expressed declaratively using, for instance, a temporal logic. Currently, the extraction of formal requirements from scenario-based descriptions is a tedious and error-prone process that would benefit from automated tool support. We present an ILP methodology for inferring requirements from a set of scenarios and an initial but incomplete requirements specification. The approach is based on translating the specification and scenarios into an event-based logic programming formalism and using a non-monotonic ILP system to learn a set of missing event preconditions. The contribution of this paper is a novel application of ILP to requirements engineering that also demonstrate the need for non-monotonic learning.

1 Introduction

Requirements Engineering refers to all aspects of the software development life-cycle concerned with identifying, analysing and documenting stakeholder requirements [2]. Several approaches have been developed to assist Requirements Engineers in the refinement of high-level goals into operational requirements [12,13] declaratively expressed in a temporal logic [16]. The use of a temporal formalism enables the deployment of automated analysis and refinement tools, but is not directly accessible to most stakeholders with a less technical background. In practice, stakeholders prefer to convey their goals through more intuitive narrative-style scenarios of desirable and undesirable system behaviour [30]. Because scenarios are inherently *partial* descriptions that leave requirements implicitly defined, it is necessary to synthesise a declarative requirements specification that admits the desired behaviours while rejecting the undesired ones. Currently, the extraction of declarative requirements from scenario-based descriptions is a tedious and error-prone process that relies on the manual efforts of an experienced engineer and would benefit from automated tool support.

This paper presents an ILP approach for extracting requirements from example scenarios and a partial requirements specification. Scenarios represent examples of desirable and undesirable system behaviour over time while the requirements specification captures our initial but incomplete background knowledge of the envisioned system and its environment. The task is to complete the specification by learning a set of missing requirements that cover all of the desirable scenarios, but none of the undesirable ones. We show how this task can be naturally represented as a non-monotonic ILP problem in which the partial requirements specification provides the background knowledge and the scenarios comprise the positive and negative examples. In particular, we show how the initial specification and scenarios can be translated into an ILP representation based on the Event Calculus [8,17]. Because this representation makes essential use of negation in formalising the effects and non-effects of actions, the resulting learning problem is inherently non-monotonic. We show that, under the *stable model* [4] semantics for logic programs with negation, the stable models of the transformed program correspond to the temporal models of the original specification. We show that stable models of the program correspond to the temporal models of the original specification. We then use a non-monotonic ILP system, called XHAIL [24,25], to generalise the scenarios with respect to the initial specification. For the purposes of illustration, we restrict the language bias of XHAIL so as to compute a specific form of missing requirements, called *event preconditions*, which state that a certain event may not happen under some particular conditions.

The paper is organised as follows. Section 2 presents some background material on Linear Temporal Logic (LTL) and the Event Calculus (EC). Section 3 describes the main features of our approach. Section 4 provides an illustrative case study involving a Mine Pump controller. We conclude with a summary and remarks about related and future work.

2 Background

Several logic-based formalisms have been used for representing requirements specifications [5,10,27]. Among these, the Event Calculus (EC) [8] is particularly well suited to logic programming approaches like ILP. Moreover, its explicit representation of time and domain specific axioms makes EC an ideal formalism for representing and reasoning about a wide class of event-driven systems. Although EC has been successfully used as a "back-end" computational formalism [27] it is not a mainstream representation because it necessitates familiarity with logic programming. By contrast, Linear Temporal Logic (LTL) [16] is very widely used by software engineers for specifying system goals and properties. In this paper we propose a method for translating between LTL and EC descriptions in order to enable the use of ILP techniques in Requirements Engineering. In the rest of this section we briefly recall the syntax and semantics of these formalisms.

2.1 Linear Temporal Logic

The language of LTL includes a set of propositions P, the Boolean connectives ($\neg$, $\wedge$, $\vee$ and $\rightarrow$) and the temporal operators $\bigcirc$ (*next*), $\Box$ (*always*), $\Diamond$ (*eventually*), U (*strong until*) and W (*weak until*). Well-formed formulae are constructed in the standard way. We use $(\neg)$ a to refer to either the atom a or the negation $\neg\ a$ of that atom. Also, we use $\bigcirc^i$ to denote i consecutive applications of the $\bigcirc$ operator. We assume P is partitioned into two sets P_e and P_f denoting *event* and *fluent* propositions, respectively. The truth or falsity of an LTL formulae is specified relative to a graph-based structure called a *Labelled Transition System* (LTS) [15,6].

Definition 1. *A labelled transition system (LTS) is a tuple $\langle S, E, \rightarrow, s_0\rangle$ where S is a non-empty set of states, E is a non-empty set of* events, $\rightarrow\ \subseteq S\times E\times S$ is a labelled transition relation, *and s_0 is* the initial state. *A transition $(s, e, s') \in \rightarrow$ from a state s to a new state s' labelled by e is denoted graphically as $s \xrightarrow{e} s'$. A* path *in an LTS is a sequence of states and transitions, from the initial state, of the form $\sigma = s_0 \xrightarrow{e_1} s_1, \ldots$ where $e_i \in E$ is said to be* at position i in σ *and s_i is said to be the i^{th} state in σ.*

As formalised in Definition 2 below, an LTL model is a pair $\langle T, V\rangle$ consisting of an LTS, T, and a valuation function, V, that assigns to each fluent proposition an arbitrary set of states in paths of T. The events are not specified in V as their truth is implicitly determined by the transitions in T. This is formalised in Definition 3, which defines the satisfaction of an LTL formula ϕ with respect to a path σ in the LTS T.

Definition 2. *Given an LTL language with propositions $P = P_e \cup P_f$ an LTL* model *is a pair $\langle T,\ V\rangle$ where T is an LTS with events P_e and V is a valuation function $V : P_f \Rightarrow 2^A$, where $A = \{(\sigma, i) \mid \sigma \text{ path in T and } i \text{ position in } \sigma\}$.*

The satisfiability of an LTL formula is defined with respect to positions (or states) in a given path σ. A formula ϕ is said to be true at position i in a path σ, denoted $\sigma, i \models \phi$ iff it is true at the s_i state in the path σ.

Definition 3. *Given an LTL language with propositions $P = P_e \cup P_f$, an LTL model $\langle T, V\rangle$ and a path σ in T, the satisfaction of an LTL formula ϕ at a position $i \geq 0$ of the path σ is defined inductively as follows:*

- $\sigma, 0 \not\models e$ *for any event proposition* $e \in P_e$
- $\sigma, i \models e$ *iff* e *is at position* i $(i \geq 1)$ *in the path* σ, *where* $e \in P_e$
- $\sigma, i \models f$ *iff* $(\sigma, i) \in V(f)$, *where* $f \in P_f$
- $\sigma, i \models \neg\phi$ *iff* $\sigma, i \not\models \phi$
- $\sigma, i \models \phi \wedge \psi$ *iff* $\sigma, i \models \phi$ *and* $\sigma, i \models \psi$
- $\sigma, i \models \phi \vee \psi$ *iff* $\sigma, i \models \phi$ *or* $\sigma, i \models \psi$
- $\sigma, i \models \bigcirc\phi$ *iff* $\sigma, i + 1 \models \phi$
- $\sigma, i \models \Box\phi$ *iff* $\forall j \geq i.\ \sigma, j \models \phi$
- $\sigma, i \models \Diamond\phi$ *iff* $\exists j \geq i.\ \sigma, j \models \phi$

– $\sigma, i \models \phi \; \mathsf{U} \; \psi$ *iff* $\exists j \geq i.\ \sigma, j \models \psi$ *and* $\forall i \leq k < j.\ \sigma, k \models \phi$
– $\sigma, i \models \phi \; \mathsf{W} \; \psi$ *iff* $\sigma, i \models \Box\phi$ *or* $\sigma, i \models \phi \; \mathsf{U} \; \psi$

An LTL formula ϕ is said to be *satisfied in a path* σ if it is satisfied at the initial position, i.e. $\sigma, 0 \models \phi$. Similarly, a set of formulae Γ is said to be satisfied in a path σ if each formula $\psi \in \Gamma$ is satisfied in the path σ.

Definition 4. *Let Γ be a set of LTL formulae and ϕ be an LTL formula. Let $M = \langle T, V \rangle$ be an LTL model. The formula ϕ is said to be* entailed *by Γ under M, written $\Gamma \models_M \phi$, iff ϕ is satisfied in each path σ of T that satisfies Γ.*

2.2 Event Calculus

The Event Calculus (EC) is a widely-used logic programming formalism for reasoning about actions and time [29]. The standard definition of an EC language includes three sorts of terms: *event* terms, *fluent* terms, and *time* terms. The latter are represented by the non-negative integers $0, 1, 2, \ldots$, while the events and fluents are chosen according to the domain being modelled. In this paper, we assume an additional sort representing *scenarios*. The EC ontology includes the basic predicates *happens*, *initiates*, *terminates* and *holdsAt*. The atomic formula $happens(e, t, s)$ indicates that event e occurs at time-point t in a given scenario s, while $initiates(e, f, t, s)$ (resp. $terminates(a, f, t, s)$) means that, in a given scenario s, if event e were to occur at time t, it would cause fluent f to be true (resp. false) immediately afterwards. The predicate $holdsAt(f, t, s)$ indicates that fluent f is true at time-point t in a given scenario s. The formalism also includes an auxiliary predicate $clipped(t_1, f, t_2, s)$ which means that, in a given scenario s, an event occurs which terminates f between times t_1 and t_2. Events correspond to actions which can be performed, while fluents correspond to time-varying Boolean properties. The interactions between the EC predicates are governed by a set of domain-independent core axioms shown below[1].

$$\begin{aligned} clipped(T_1, F, T_2, S) \leftarrow\ & happens(E, T, S), \\ & terminates(E, F, T, S), T_1 \leq T < T_2. \end{aligned} \tag{1}$$

$$\begin{aligned} holdsAt(F, T_2, S) \leftarrow\ & happens(E, T_1, S), initiates(E, F, T_1, S), \\ & T_1 < T_2, not\ clipped(T_1, F, T_2, S). \end{aligned} \tag{2}$$

$$holdsAt(F, T, S) \leftarrow initially(F, S), not\ clipped(0, F, T, S). \tag{3}$$

$$happens(E, T, S) \leftarrow attempt(E, T, S), not\ impossible(E, T, S). \tag{4}$$

These axioms formalise the commonsense law of inertia which states that, in any scenario S, a fluent that has been initiated by an event occurrence continues

[1] The EC axioms used here are identical to those in [17] apart from the extra argument S for representing scenarios and the predicate *impossible* for capturing preconditions.

to hold until a terminating event occurs and vice versa. To allow the representation of preconditions, we say that an event E happens at a time point T if it is attempted and is not impossible[2]. Information about which events affect which fluents is provided by domain-dependent axioms for the predicates *initiates* and *terminates*, together with information about which fluents are initially true and which events are attempted in given system behaviours.

EC theories are *normal logic programs* - i.e. a set of clauses of the form $A \leftarrow B_1, \ldots, B_n,\ not\ C_1, \ldots, not\ C_m$ where A is the *head atom*, B_i are *positive body literals*, and $not\ C_j$ are *negative body literals*. Their semantics is given by the standard *stable model* semantics [4]. In general, a *model* I of a program Π is a set of ground atoms such that, for each ground instance G of a clause in Π, I satisfies the head of G whenever it satisfies the body. A model I is *minimal* if it does not strictly include any other model. Definite programs (i.e. programs with no negative body literals) always have a unique minimal model. Normal programs may have instead one, none, or several minimal models. It is usual to identify a certain subset of these models, called *stable models*, as the possible meanings of the program. Given a normal program Π, the definite program Π^I is the program obtained from the ground instances of Π by removing all clauses with a negative literal that is not satisfied in Π and removing negative literals from the remaining clauses. Clearly Π^I is a definite logic program and as such has a unique minimal (Herbrand) model M_{Π^I}. A model I of a program Π is *stable* if it is equal to M_{Π^I}.

Definition 5. *A model I of Π is a* stable model *if $I = M_{\Pi^I}$ where Π^I is the definite program $\Pi^I = \{A \leftarrow B_1, \ldots, B_n \mid A \leftarrow B_1, \ldots, B_n,\ not\ C_1, \ldots, not\ C_m$ is the ground instance of a clause in Π and I does not satisfy any of the $C_j\}$.*

3 The Approach

In this section we show how ILP can be used to extend an incomplete requirements specification using information from given scenarios. We formalise the learning problem in terms of LTL specifications and scenarios, and show how these can be soundly translated into a non-monotonic ILP problem using an EC formalisation to extend the specification in order to cover the given scenarios.

3.1 Problem Description

Our aim is to develop an approach for extending an incomplete requirements specification with a particular type of requirement called event preconditions by using information inferred from desirable and undesirable user scenarios. In order to formalise this task we need to state precisely what we mean by a requirements specification and by a desirable or undesirable scenario and we need

[2] Alternative formalisations of event preconditions have been proposed in EC [17]. The one adopted here captures the intuition that *impossible*(E, T, S) means the event E could not actually occur at time point T.

to define what it means for a specification to cover a set of such scenarios. To do this, we assume an LTL language with fluents P_f and events P_e in which each fluent $f \in P_f$ is associated with two disjoint sets I_f and T_f of initiating and terminating events $e \in P_e$. For convenience we use the notation E_I^f to represent the disjunction $\bigvee_{e \in I_f} e$ of f-initiating events, and E_T^f for the disjunction $\bigvee_{e \in T_f} e$ of f-terminating events. We also use the notation S_0 to represent the set of fluents $f \in P_f$ that are true in the initial system state s_0. We now define requirements specifications and scenarios as LTL theories containing formulae of the forms defined below.

As formalised in Definition 6, a requirements specification consists of a set of *initial state axioms* (5,6) stating which fluents are initially true and false; *persistence axioms* (7,8) formalising the commonsense law of inertia that any fluent will remain true (resp. false) until a terminating (resp. initiating) event occurs that causes it to flip state; *change axioms* (9,10), stating that, for any fluent $f \in P_f$, the occurrence of any initiating (resp. terminating) event will cause f to become true (resp. false); and a set of *event precondition axioms* (11) which disallow any models that include transitions of the form $s_k \xrightarrow{e} s_{k+1}$ for any state s_k that satisfies a certain conjunction of fluent literals $\bigwedge_{0 \leq i \leq n} (\neg) f_i$.

Definition 6. *A* requirements specification *is an LTL theory consisting of*

– *two* initial state axioms

$$\bigwedge_{f_i \in S_0} f_i \tag{5}$$

$$\bigwedge_{f_j \in P_f - S_0} \neg f_j \tag{6}$$

– *two* persistence axioms *for each fluent* $f \in P_f$

$$\Box(f \rightarrow f \; \mathcal{W} E_T^f) \tag{7}$$

$$\Box(\neg f \rightarrow \neg f \; \mathcal{W} E_I^f) \tag{8}$$

– *two* change axioms *for each fluent* $f \in P_f$

$$\Box(E_I^f \rightarrow \; f) \tag{9}$$

$$\Box(E_T^f \rightarrow \; \neg f) \tag{10}$$

– *a set of* event precondition axioms *of the form*

$$\Box(\bigwedge_{0 \leq i \leq n} (\neg) f_i \rightarrow \bigcirc \neg e) \tag{11}$$

As formalised below, a scenario is a formula stating a sequence of occurrences of events $\langle e_1, \ldots, e_m \rangle$.

Definition 7. *A* scenario *is an LTL formula of the form*

$$\bigwedge_{1 \leq i \leq m} \bigcirc^i e_i \tag{12}$$

Note that the definition above assumes one event to be true per point position. A *desirable scenario* is a scenario that may occur while an *undesirable scenario* is a sequence of events that should never occur.

Using Definition 7, we can now formalise our learning task. Given an initial specification *Spec* together with a set of undesirable scenarios *Und* and desirable scenarios *Des*, our aim is to learn a set of event precondition axioms *Pre* that, when added to *Spec*, entails the negation of each undesirable scenario and is consistent with each desirable scenario. As formalised in Definition 8, the first condition states that, in any model of $Spec \cup Pre$, there is no path which produces any undesirable scenario in *Und*, while the second condition states that, in any model of $Spec \cup Pre$, there is always a path corresponding to each desirable scenario in *Des*. Any set of event precondition axioms that satisfy these two properties is said be a *correct extension* of a requirements specification with respect to the given scenarios.

Definition 8. *Let* Spec *be a requirements specification,* Des *be a set of desirable scenarios, and* Und *be a set of undesirable scenarios. A set* Pre *of event precondition axioms is a* correct extension *of* Spec *with respect to* Des *and* Und *iff*

- $Spec \cup Pre \models_M \neg P_u$, *for each undesirable scenario* $P_u \in Und$
- $Spec \cup Pre \not\models_M \neg P_d$, *for each desirable scenario* $P_d \in Des$

3.2 Translating LTL into EC

To apply ILP to the task of learning correct extensions, a methodology is now defined for translating LTL specifications and scenarios of the form defined above into EC normal logic programs. The EC language is obtained very simply from the LTL formulae: one fluent (resp. event) term is introduced to represent each fluent f (resp. event e) in P_f (resp. P_e); time points are represented by the non-negative integers 0, 1, 2, ...; one scenario term is introduced to represent each desirable (resp. undesirable) scenario $P_d \in Des$ (resp. $P_u \in Und$). Relative to this language, the EC translation of a requirements specification is defined as follows.

Definition 9. *Let* Spec *be a requirements specification. The* EC translation $\tau(Spec)$ *of* Spec *is the EC program* Π *constructed as follows:*

- *add to* Π *one fact* $initially(f_i, S)$ *for each fluent* f_i *in an initial state axiom of the form* $\bigwedge_{f_i \in S_0} f_i$.
- *add to* Π *one fact* $initiates(e, f, T, S)$ *for each* f*-initiating event* $e \in E_I^f$ *in a change axiom of the form* $\Box(E_I^f \rightarrow f)$.
- *add to* Π *one fact* $terminates(e, f, T, S)$ *for each* f*-terminating event* $e \in E_T^f$ *in a change axiom of the form* $\Box(E_T^f \rightarrow \neg f)$.

– *add to Π one rule $impossible(e, T, S) \leftarrow \bigwedge_{0 \leq i \leq k}(not)holdsAt(f_i, T, S)$ for each event precondition axiom of the form $\Box(\bigwedge_{0 \leq i \leq k}(\neg)f_i \rightarrow \bigcirc\neg e)$.*

Note that the negative initial state axiom (6) and the persistence axioms (7) and (8) are all implicitly captured by stable model interpretation of the EC core axioms (which are incorporated into the translation of scenarios in Definition 10 below). Note also that the effect of the temporal operator $\Box$ is captured by implicit universal quantification on the time variable T appearing in the *initiates* and *terminates* facts. As shown in Theorem 1, the translation τ is sound in the sense that for any path σ in any model of *Spec* there is a corresponding narrative of events Nar such that the program $\Pi = \tau(Spec) \cup Nar$ has a stable model that satisfies the same fluent and event formulae as σ.

Theorem 1. *Let Spec be a requirements specification with LTL model $\langle T, V\rangle$ such that any path in T satisfies Spec at position 0. Let σ be a path in T of the form $s_0 \xrightarrow{e_1} s_1, \ldots, s_{n-1} \xrightarrow{e_n} s_n$, and let Nar be the set of facts of the form $attempt(e_i, i-1, \sigma)$ for each event e_i in σ. Let Π be the EC logic program $\Pi = \tau(Spec) \cup Nar$ with stable model I. Then, for any fluent f and position i, we have $\sigma, i \models f$ iff $holdsAt(f, i, \sigma)$ is true in I; and, for any event e and position i, we have $\sigma, i \models e$ iff $happens(e, i-1, \sigma)$ is true in I.*

The function τ translates an LTL requirements specification into an ILP theory. It now remains to specify a corresponding translation from scenarios to ILP examples. As formalised in Definition 10 below, scenarios contribute facts to the background theory as well as to the examples. Specifically, each scenario produces a set of example literals of the form $(not)happens(e, t, s)$ and a set of background facts of the form $attempt(e, t, s)$. The translation of the undesirable scenarios depends on the event for which the precondition axiom is to be learned. In what follows, it is assumed that preconditions are to be learned for the last event of each undesirable scenario. Consequently, each undesirable scenario produces a sequence of facts stating that certain events do happen followed by one fact stating that some particular event does not happen immediately afterward. Each desirable scenario simply states that a certain sequence of events does happen.

Definition 10. *Let* Spec *be a requirements specification, and* Des *and* Und *be sets of desirable and undesirable scenarios respectively. The* EC translation *$\tau(Spec, Des, Und)$ is the pair (B, E) of EC programs constructed as follows:*

– *for each undesirable scenario $P_u = \bigwedge_{1 \leq i \leq n} \bigcirc^i e_i$ in* Und
  - *add to E $n-1$ facts $happens(e_i, i-1, u)$ with $1 \leq i < n$*
  - *add to E 1 fact not $happens(e_n, n-1, u)$*
  - *add to B n facts $attempts(e_i, i-1, u)$ with $1 \leq i \leq n$*

– *for each desirable scenario $P_d = \bigwedge_{1 \leq i \leq m} \bigcirc^i e_i$ in* Des
  - *add to E m facts $happens(e_i, i-1, d)$ with $1 \leq i \leq m$*
  - *add to B m facts $attempts(e_i, i-1, u)$ with $1 \leq i \leq m$*

– *add to B all of the facts and rules in $\tau(Spec)$*

– *add to B the 4 EC core axioms (1)-(4).*

3.3 Computation of Event Precondition Axioms Using XHAIL

Given an initial specification *Spec* and sets of desirable and undesirable scenarios *Des* and *Und*, the translation τ defined above can be used to generate a normal ILP theory B and examples E (such that $\tau(Spec, Des, Und) = (B, E)$). For any set Pre of event precondition axioms, τ can also be used to generate a set H of normal clauses of the form (13) below (such that $\tau(Pre) = H$).

$$impossible(e, T, S) \leftarrow \bigwedge_{0 \leq j \leq n} (not)\ holdsAt(f_j, T, S) \tag{13}$$

Moreover, it follows from Theorem 1 that Pre is a correct extension of *Spec* with respect to *Des* and *Und* iff $B \cup H \models E$ under the stable model view of $\models$. Hence, the task of computing correct extensions can be reduced to a non-monotonic ILP problem in the sense of [28] where the hypothesis space is the set of all clauses of the form (13) above.

The computation of such preconditions is performed by the non-monotonic ILP system XHAIL [25], which uses an abductive engine to implement a three-phase Hybrid Abductive Inductive Learning (HAIL) approach [24]. This approach is based on constructing and generalising a preliminary ground hypothesis K, called a *Kernel Set* of B and E, which can be regarded as a non-monotonic multi-clause generalisation of the well-known *Bottom Set* concept used in several Progol-based ILP systems [20]. As in these monotonic ILP systems, the construction of the Kernel Set is heavily guided by language and search bias, and its main purpose is to bound the ILP hypothesis space.

The XHAIL language and search bias mechanisms are based upon the tried-and-tested notions of mode declarations and compression as used for example in Progol [20]. Intuitively, the compression heuristic favours the inference of theories containing the fewest number of literals and is motivated by the scientific principle of Ocam's razer (which roughly speaking, means choose the simplest hypothesis that fits the data). Mode declarations on the other hand provide a convenient mechanism for specifying which predicates may appear in the heads and bodies of hypothesis clauses and for controlling the placement and linking of constants and variables within those clauses [20].

As formalised in [20] mode declarations are of two types *head* and *body* declarations. To learn formulae of the form (13) above, one head mode declaration is needed $modeh(*, impossible(\#event, +time, +scenario))$ to allow atoms of the form impossible(e,T,S) to appear in the heads of H. Two body mode declarations are also needed, $modeb(*, holdsAt(\#fluent, +time, +scenario))$ and $modeb(*, not\ holdsAt(\#fluent, +time, +scenario))$, to allow literals of the form $holdsAt(f, T, S)$ and $not\ holdsAt(f, T, S)$ to appear in the bodies of H. The symbols $\#, +, -$ are called *placemarkers* and are replaced by constants, input and output variables, respectively.

As explained in [25], the hypothesis H is computed in three stages: first the head atoms Δ of the Kernel Set K are obtained abductively, then the body literals of K are obtained by deduction, and finally K is inductively generalised to give H. To exploit a close correspondence between negation and abduction, XHAIL performs all three phases by translating them into an Abductive Logic Programming (ALP) [7] formalism and using an efficient extension [26] of the Kakas-Mancarella proof procedure [7] to solve each subproblem in turn.

The first phase of the XHAIL proof procedure returns a minimal set of ground atoms Δ that entail all of the examples E when added to the theory B. This done by simply querying the examples E against the theory B. The abducible atoms are defined as the well-typed ground instances of any head declarations. To avoid any unsoundness caused by the non-monotonicity of the EC axioms, an incremental cover set approach is not used; instead XHAIL generalises all of the examples at once.

The second phase of the procedure computes a ground Kernel Set K of B and E by making each abduced atom $\alpha \in \Delta$ into the head of a clause and saturating it with a set of ground body literals entailed by B. This is done using a non-monotonic generalisation of the Progol saturation procedure [20]. In order to compute the deductive consequences of B, XHAIL employs the Eshgi-Kowalski transformation for implementing negation through abduction [3]. In effect, negative literals $not(a)$ are treated as positive abducibles a^* subject to the implicit integrity constraints $a \rightarrow \neg a^*$ and $a^* \rightarrow \neg a$.

The third phase, returns a hypothesis H that subsumes K and entails E with respect to B. Two transforms prepare the ALP system for this task. First, all input and output terms in K are replaced by variables. Then, each body literal λ_i^j at position i in the j-th clause of K is replaced by the atom $try(i, j, [X_1, \ldots, X_k])$, where $X_1, \ldots, X_k$ are the variables added to that clause, and the two clauses $try(i, j, [X_1, \ldots, X_k]) \leftarrow not(use(i, j))$ and $try(i, j, [X_1, \ldots, X_k]) \leftarrow use(i, j), \lambda_i^j$ are added to K. Applying an ALP procedure to the resulting theory $B \cup K$ with goal E and abducible $use/2$ gives a set of atoms $S = \bigwedge use(i, j)$ indicating which literals λ_i^j should be kept in H.

Soundness of XHAIL with respect to the stable model semantics follows from the soundness of the Kakas-Mancarella ALP procedure and the fact that H is equivalent to the theory $K \cup S$ computed in the inductive phase of the XHAIL procedure and which, by definition, entails the examples. Strictly speaking, XHAIL implements the partial stable model semantics, but since the EC programs generated by τ are categorical in the sense of [28], the two semantics coincide in this particular application. As illustrated by the case study in the next section, XHAIL can therefore be used to compute correct extensions of a partial specification and scenarios via the translation function τ.

4 Case Study: A Mine Pump Control System

This section shows an application of the learning approach proposed in this paper on a real event-driven system, namely the Mine Pump Control System fully described in [9]. This is a system that is supposed to monitor and control water levels in a mine, so to avoid the risk of flood. It is composed of a pump for pumping mine-water up to the surface. The pump works automatically, controlled by water-level sensors: detection of a high-level water causes the pump to run until low-level is indicated. For safety reasons, the pump must not run if the percentage of methane in the mine exceeds a certain critical limit.

An initial partial requirement specification *Spec* is given, written in an LTL language with fluent propositions $P_f = \{pumpOn, criticalMethane, highWater\}$ and event propositions $P_e = \{turnPumpOn, turnPumpOff, signalCriticalMethane, signalNotCriticalMethane, signalHighWater, signalNotHighWater\}$. The specifications includes information about the initial state of the system, persistence axioms, and change axioms formalised as follows:

$$(\neg criticalMethane \wedge \neg pumpOn \wedge \neg highWater) \tag{14}$$

$$\Box(criticalMethane \rightarrow (criticalMethane\ \mathsf{W}\ signalNotCriticalMethane)) \tag{15}$$

$$\Box(\neg criticalMethane \rightarrow (\neg criticalMethane\ \mathsf{W}\ signalCriticalMethane\ )) \tag{16}$$

$$\Box(pumpOn \rightarrow\ (pumpOn\ \mathsf{W}\ turnPumpOff)) \tag{17}$$

$$\Box(\neg pumpOn \rightarrow\ (\neg pumpOn\ \mathsf{W}\ turnPumpOn)) \tag{18}$$

$$\Box(highWater \rightarrow (highWater\ \ \mathsf{W}\ signalNotHighWater)) \tag{19}$$

$$\Box(\neg highWater \rightarrow (\neg highWater\ \mathsf{W}\ signalHighWater)) \tag{20}$$

$$\Box(signalCriticalMethane\ \rightarrow criticalMethane) \tag{21}$$

$$\Box(signalNotCriticalMethane \rightarrow\ \neg criticalMethane) \tag{22}$$

$$\Box(signalHighWater \rightarrow highWater) \tag{23}$$

$$\Box(signalNotHighWater \rightarrow \neg highWater\ ) \tag{24}$$

$$\Box(turnPumpOn \rightarrow\ pumpOn) \tag{25}$$

$$\Box(turnPumpOff \rightarrow \neg pumpOn) \tag{26}$$

Equation (14) defines the initial state of the system, equations (15)–(20) specify the persistence axioms, and equations (21)–(26) define the change axioms. Together with the informal description the case study includes undesirable and desirable scenarios which have been formalised as follows:

$$\begin{aligned} P_u = (&\bigcirc signalCriticalMethane \wedge \bigcirc^2 signalNotCriticalMethane \wedge \\ &\bigcirc^3\ signalCriticalMethane \wedge \bigcirc^4 turnPumpOn) \end{aligned} \tag{27}$$

$$\begin{aligned} P_{d1}\ = (&\ \bigcirc signalCriticalMethane\ \ \wedge \bigcirc^2 signalHighWater \wedge \\ &\bigcirc^3\ signalNotCriticalMethane\ \ \wedge \bigcirc^4 turnPumpOn \wedge \\ &\bigcirc^5\ signalCriticalMethane \wedge \bigcirc^6 turnPumpOff) \end{aligned} \tag{28}$$

$$\begin{aligned} P_{d2}\ =\ (&\bigcirc signalHighWater \wedge \bigcirc^2 turnPumpOn \wedge \\ &\bigcirc^3\ signalNotHighWater \wedge \bigcirc^4 turnPumpOff \wedge \\ &\bigcirc^5\ signalHighWater \wedge \bigcirc^6 turnPumpOn) \end{aligned} \tag{29}$$

Applying the translation τ to the specification and scenarios above results in an ILP theory B composed of the EC core axioms and the following clauses:

initiates(signalCriticalMethane,criticalMethane,T,S).
terminates(signalNotCriticalMethane,criticalMethane,T,S).
initiates(signalHighWater,highWater,T,S).
terminates(signalNotHighWater,highWater,T,S).
initiates(turnPumpOn,pumpOn,T,S).
terminates(turnPumpOff,pumpOn,T,S).

attempt(signalCriticalMethane,0,u).
attempt(signalNotCriticalMethane,1,u).
attempt(signalCriticalMethane,2,u).
attempt(turnPumpOn,3,u).

attempt(signalHighWater,0,dp1).
attempt(turnPumpOn,1,dp1).
attempt(signalNotHighWater,2,dp1).
attempt(turnPumpOff,3,dp1).

attempt(signalCriticalMethane,0,dp2).
attempt(signalHighWater,1,dp2).
attempt(signalNotCriticalMethane,2,dp2).
attempt(turnPumpOn,3,dp2).
attempt(signalCriticalMethane,4,dp2).
attempt(turnPumpOff,5,dp2).

In addition, the translation produces the following set of ILP examples E:

happens(signalCriticalMethane,0,u).
happens(signalNotCriticalMethane,1,u).
happens(signalCriticalMethane,2,u).
not happens(turnPumpOn,3,u).

happens(signalHighWater,0,dp1).
happens(turnPumpOn,1,dp1).
happens(signalNotHighWater,2,dp1).
happens(turnPumpOff,3,dp1).

happens(signalCriticalMethane,0,dp2).
happens(signalHighWater,1,dp2).
happens(signalNotCriticalMethane,2,dp2).
happens(turnPumpOn,3,dp2).
happens(signalCriticalMethane,4,dp2).
happens(turnPumpOff,5,dp2).

Applying XHAIL to B and E yields a single abductive explanation

$$\Delta = \{impossible(turnPumpOn, 3, u)\} \tag{30}$$

This results in a single Kernel Set containing one clause

$$K = \{\mathit{impossible(turnPumpOn,3,u)} \leftarrow \mathit{holdsAt}(\mathit{criticalMethane}, 3, u), \\ \mathit{not\ holdsAt}(\mathit{pumpOn}, 3, u), \mathit{not\ holdsAt}(\mathit{highWater}, 3, u)\} \quad (31)$$

which gives two maximally compressive inductive generalisations

$$H_1 = \{\mathit{impossible}(\mathit{turnPumpOn}, T, S) \leftarrow \mathit{holdsAt}(\mathit{criticalMethane}, T, S)\} \quad (32)$$

$$H_2 = \{\mathit{impossible}(\mathit{turnPumpOn}, T, S) \leftarrow \mathit{not\ holdsAt}(\mathit{highWater}, T, S)\} \quad (33)$$

that correspond to the two correct LTL event precondition axioms

$$\mathit{Pre}_1 = \Box(\mathit{criticalMethane} \rightarrow \bigcirc\neg \mathit{turnPumpOn}) \quad (34)$$

$$\mathit{Pre}_2 = \Box(\neg \mathit{highWater} \rightarrow \bigcirc\neg \mathit{turnPumpOn}) \quad (35)$$

5 Conclusion, Related and Future Work

This paper describes a methodology for using ILP to extend a partial requirements specification with event preconditions extracted from user scenarios. The proposed approach works in two stages whereby the initial specification and scenarios are first translated from an LTL model into an EC representation so that a nonmonotonic ILP system can then be used to learn the missing requirements. By exploiting the semantic relationship between the LTL and EC, we thereby provide a sound ILP computational "back-end" to a temporal formalism familiar to Requirements Engineers.

Our approach is closely related to that of [11], where an inductive method is proposed for inferring high-level goal assertion from positive and negative scenarios provided by stakeholders. Scenarios are incrementally generalised by (a) conjoining new assertions with those obtained from previous scenarios and (b) merging assertions through pattern matching on common antecedent prefixes. Compared to [11], our ILP-based approach has the advantage of incorporating background knowledge into the learning process and producing more compact and comprehensible hypotheses. Moreover, by making *happens* abducible, our approach can be applied to scenarios missing events while [11] cannot.

The method proposed in this paper builds upon earlier work in [19] and [18] in which the ILP systems Progol5 and Alecto were applied to the learning of domain specific EC axioms. Like XHAIL, these procedures employ an abductive reasoning module to enable the learning of predicates distinct from those in the examples — an ability that is clearly required in this application. However, unlike XHAIL, they do not have a well-defined semantics for non-definite programs and their handling of negation is rather limited [25]. In fact, the inability of Progol5 and Alecto to reason abductively through nested negations means that neither of these systems can solve the case study presented in this paper.

Some related approaches for inferring action theories from examples are presented in [14], [21] and more recently in [23], which reduce learning in the

Situation Calculus to a monotonic ILP framework. These approaches work by pre- and post- processing the inputs and outputs of a conventional Horn Clause ILP system. This technique is very efficient, but is not as general as our own approach. An alternative method for nonmonotonic ILP under the stable model semantics is proposed in [28], but cannot be used in our case study because it assumes the target predicate is the same as the examples. [28] also includes a thorough review of previous work on nonmonotonic ILP. A more recent technique is proposed in [22] that uses a combination of SAT solvers and Horn ILP to perform induction under the stable model semantics.

Although the approach presented in this paper has been tailored for the learning of event preconditions, it can also learn other types of requirements such as triggers and post-conditions of the form $\Box(\bigwedge_{0\leq i\leq k} f_i \rightarrow \bigcirc e)$ and $\Box(e \rightarrow \bigwedge_{0\leq j\leq h} f_j)$ respectively. In principle, this can be achieved by changing the language bias appropriately; but it remains to test the efficiency of the approach when learning more general forms of requirements and when processing larger case studies. In this paper we also assumed that scenarios are provided by stakeholders. However, scenarios could also be automatically generated from desirable system properties via model-checking [1]. We therefore intend to investigate the integration of ILP and model checking techniques in order to find new ways of increasing the flexibility and efficiency of the approach.

Acknowledgments. This work is funded by the Philip Leverhulme Trust and the Saudi Arabian Ministry of Higher Education.

References

1. Alrajeh, D., Russo, A., Uchitel, S.: Inferring operational requirements from goal models and scenarios using inductive systems. In: Proc. 5th Int. Workshop on Scenarios and State Machines (2006)
2. Dardenne, A., Lamsweerde, A.v., Fickas, S.: Goal-directed requirements acquisition. Science of Computer Programming 20 (1), 3–50 (1993)
3. Eshghi, K., Kowalski, R.A.: Abduction compared with negation by failure. In: Levi, G., Martelli, M. (eds.) Proc. of the 6th Int. Conf. on Logic Programming, pp. 234–254 (1989)
4. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R.A., Bowen, K. (eds.) Proc. of the 5th Int. Conf. on Logic Programming, pp. 1070–1080. MIT Press, Cambridge (1988)
5. Giannakopoulou, D., Magee, J.: Fluent model checking for event-based systems. In: Proc. 11th ACM SIGSOFT Symp. on Foundations Software Engineering, ACM Press, New York (2003)
6. Huth, M., Ryan, M.D.: Logic in Computer Science: Modelling and Reasoning about systems. Cambridge University Press, Cambridge (2000)
7. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive Logic Programming. Journal of Logic and Computation 2(6), 719–770 (1992)
8. Kowalski, R.A., Sergot, M.: A logic-based calculus of events. New generation computing 4(1), 67–95 (1986)
9. Kramer, J., Magee, J., Sloman, M.: Conic: An integrated approach to distributed computer control systems. In: IEE Proc., Part E 130, pp. 1–10 (January 1983)

10. Lamsweerde, A.V.: Goal-oriented requirements engineering: A guided tour. In: Proc. 5th IEEE Int. Symp. on Requirements Engineering, pp. 249–263. IEEE Computer Society Press, Los Alamitos (2001)
11. Lamsweerde, A.V., Willemet, L.: Inferring declarative requirements specifications from operational scenarios. IEEE Trans. on Software Engineering 24(12), 1089–1114 (1998)
12. Letier, E., Kramer, J., Magee, J., Uchitel, S.: Deriving event-based transitions systems from goal-oriented requirements models. Technical Report 2006/2, Imperial College London (2005)
13. Letier, E., Lamsweerde, A.V.: Deriving operational software specifications from system goals. In: Proc. 10th ACM SIGSOFT Symp. on Foundations of Software Engineering, pp. 119–128. ACM Press, New York (2002)
14. Lorenzo, D.: Learning non-monotonic Logic Programs to Reason about Actions and Change. PhD thesis, University of Coruna (2001)
15. Magee, J., Kramer, J.: Concurrency: State Models and Java Programs. John Wiley and Sons, Chichester (1999)
16. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, Heidelberg (1992)
17. Miller, R., Shanahan, M.: Some alternative formulation of event calculus. Computer Science: Computational Logic: Logic programming and Beyond 2408 (2002)
18. Moyle, S.: An investigation into Theory Completion Techniques in ILP. PhD thesis, University of Oxford (2000)
19. Moyle, S., Muggleton, S.: Learning programs in the event calculus. In: Proc. 7th Int. Workshop on ILP (1997)
20. Muggleton, S.H.: Inverse Entailment and Progol. New Generation Computing, Special issue on Inductive Logic Programming 13(3-4), 245–286 (1995)
21. Otero, R.: Embracing causality in inducing the effects of actions. In: Proc. 10th Conf. of the Spanish Assoc. for AI (2004)
22. Otero, R., Gonzalez, J.: Iaction: a system for induction under non-horn programs with stable models. In: Dumke, R.R., Abran, A. (eds.) IWSM 2000. LNCS, vol. 2006, Springer, Heidelberg (2001)
23. Otero, R., Varela, M.: Iaction: a system for learning action descriptions for planning. In: Dumke, R.R., Abran, A. (eds.) IWSM 2000. LNCS, vol. 2006, Springer, Heidelberg (2001)
24. Ray, O.: Hybrid Abductive-Inductive Learning. PhD thesis, Imperial College London (2005)
25. Ray, O.: Using abduction for induction of normal logic programs. In: Proc. of the ECAI'06 Workshop on Abduction and Induction in AI and Scientific Modelling, pp. 28–31 (2006)
26. Ray, O., Kakas, A.: Prologica: a practical system for abductive logic programming. In: Dix, J., Hunter, A. (eds.) 11th International Workshop on Non-monotonic Reasoning. IFL Technical Report Series, pp. 304–312 (2006)
27. Russo, A., Miller, R., Nuseibeh, B., Kramer, J.: An abductive approach for analysing event-based requirements specifications. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, pp. 22–37. Springer, Heidelberg (2002)
28. Sakama, C.: Induction from answer sets in non-monotonic logic programs. ACM Trans. on Computational Logic 6(2), 203–231 (2005)
29. Shanahan, M.P.: Solving the Frame Problem. MIT Press, Cambridge (1997)
30. Sutcliffe, A., Maiden, N.A.M., Minocha, S., Manuel, D.: Supporting scenario-based requirements engineering. IEEE Trans. on Software Engineering 24, 1072–1088 (1998)