Deriving non-Zeno behaviour models from goal models using ILP

D. Alrajeh, J. Kramer, A. Russo, S. Uchitel

Under consideration for publication in Formal Aspects of Computing

Deriving Non-Zeno Behaviour Models from Goal Models using ILP

D. Alrajeh[1], J. Kramer[1], A. Russo[1] and S. Uchitel[1,2]

[1]Department of Computing, Imperial College London,
180 Queen's Gate London SW7 2AZ, UK
[2]Departamento de Computaciòn, FCEyN, Universidad de Buenos Aires,
Buenos Aires, Argentina

Abstract. One of the difficulties in Goal-Oriented Requirements Engineering (GORE) is the construction of behaviour models from declarative goal specifications. This paper addresses this problem using a combination of model checking and machine learning. First, a goal model is transformed into a (potentially Zeno) behaviour model. Then, via an iterative process, Zeno traces are identified by model checking the behaviour model against a time progress property, and Inductive Logic Programming (ILP) is used to learn operational requirements (*pre-conditions*) that eliminate these traces. The process terminates giving a non-Zeno behaviour model produced from the learned pre-conditions and the given goal model.

Keywords: Goal-oriented requirements engineering, zeno behaviour, operational requirements, model checking, inductive learning.

1. Introduction

Goal-Oriented Requirements Engineering (GORE) is an increasingly popular approach for elaborating software requirements. Goals are prescriptive statements of intent whose satisfaction requires the cooperation of software components and the environment. One of the limitations of GORE approaches [DvLF93, DvL96, LvL02, Ant97, GMS05] is due to the declarative nature of goals which hinders the application of a number of successful validation techniques based on executable models such as graphical animations, simulations, and rapid-prototyping. Goals are neither conceived for nor naturally support narrative style elicitation techniques, such as those in scenario-based requirements engineering. Additionally, they are not suitable for down-stream analyses that focus on design and implementation issues, which are of operational nature.

To address these limitations, techniques have been developed that construct behaviour models automatically from safety properties and scenarios [UBC07], and goal models [LKMU06]. The core of these techniques is based on "temporal logic to automata" transformation algorithms developed in the model checking community. For instance, in [LKMU06] Labelled Transition Systems (LTS) are built automatically from KAOS goals, which are safety goals expressed in Fluent Linear Temporal Logic (FLTL) [GM03].

However, the key technical difficulty in constructing behaviour models from goal models is that the latter

Correspondence and offprint requests to: D. Alrajeh, Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2AZ, UK. e-mail: dalal.alrajeh04@imperial.ac.uk

are typically expressed in a synchronous, non-interleaving semantic framework while the former have an asynchronous interleaving semantics. This mismatch relates to the fact that it is convenient to make different assumptions for modelling requirements and system goals than for modelling communicating sequential processes. One of the practical consequences of this mismatch is that the construction of a behaviour model from a goal model may introduce deadlocks and progress violations. More specifically, the resulting behaviour model may be *Zeno*, i.e exhibits traces in which time never progresses. Clearly these models do not adequately describe the intended system behaviour and thus are not suitable basis for down-stream analysis.

A solution is to construct behaviour models from fully *operationalised* goal models rather than from high-level goals [LKMU06]. This involves identifying system operations and extracting operational requirements in the form of *pre-* and *trigger-conditions* from the high-level goals [LvL02]. One disadvantage of this approach is that operationalisation is only partial supported, where such support comes in the form of pre-defined derivation patterns restricted to some common goal patterns [DvL96].

This paper proposes a new approach for (systematic) computation of non-Zeno behaviour models from high-level goal models, using a combination of model checking and machine learning. The approach starts with a goal model and produces a non-Zeno behaviour model that satisfies all goals. First, a given goal model, formalised in Linear Temporal Logic (LTL), is automatically translated into a (potentially Zeno) Labelled Transition System (LTS). Then, via an iterative process, Zeno traces in the LTS are identified mechanically, elaborated into positive and negative scenarios, and used to automatically learn pre-conditions that prevent such traces from occurring. Identification of Zeno traces is achieved by model checking the LTS model against a time progress property expressed in LTL, while pre-conditions are learned using Inductive Logic Programming (ILP). As a result, the process not only constructs a non-Zeno behaviour model, but also computes a set of pre-conditions. These pre-conditions, in conjunction with the given high-level goals, ensure the non-Zeno behaviour of the system. Consequently, the approach also supports the operationalisation process of goal models described in [LvL02].

The paper is an extended version of [ARU08]. It provides relevant background notions on goal models, LTS and FLTL (Section 2). It describes, in Section 3, the problem of deriving non-Zeno behaviour models from goal models and defines a formal characterisation of the proposed approach. It presents, in Section 4, the approach in detail, proving that it satisfies its formal characterisation. In Section 5, it illustrates the process through a case study on the Safety Injection event-driven System defined in [CP93]. The approach is then compared with other existing related work in Sections 6, and a discussion of future work in Section 7 concludes the paper.

2. Background

In this section we discuss goal and behaviour modelling. The examples we use refer to a simplified version of the mine pump control system, described in [KMS83], in which a pump controller is used to prevent the water in a mine pump from passing some threshold and flooding the mine. To avoid the risk of explosion, the pump may only be on when the level of methane gas in the mine is not critical. The pump controller monitors the water and methane levels by communicating with two sensors, and controls the pump in order to guarantee the safety properties of the system.

2.1. Goal Models

Goals focus on the objectives of the system. They are state-based assertions intended to be satisfied, over time, by the system. By structuring goals into refinement structures, GORE approaches aim to provide systematic methods that support requirements engineering activities including conflict detection, goal operationalisation and responsibility assignment. Goals are expected to be refined into sub-goals that can be assigned to the software-to-be or environment. Goals assigned to the environment form domain assumptions while those assigned software-to-be are used to derive operational requirements, in the form of pre-, post- and trigger-conditions, for the operations provided by the software.

In this paper, we define a goal model to be a collection of system goals, domain properties and operational requirements. We specify goals informally using natural language and formally using LTL [MP92]. Following the KAOS [DvLF93] approach, we assume a discrete-model of time, in which consecutive states in a trace are always separated by a single time unit. The time unit corresponds to some arbitrarily chosen time unit for

the application domain. We consider goals to be specified as safety properties. For instance, the system goal *PumpOffWhenLowWater* can be informally described as "*when the water is below the low level, the pump must be off within the next time unit*" and formally specified as

$$\Box(\neg HighWater \rightarrow \bigcirc \neg PumpOn) \quad (1)$$

where $\Box$ is the temporal operator meaning *always*, $\bigcirc$ is the *next* time point operator, and *HighWater* and *PumpOn* are propositions meaning that "the water in the pump is above the low level threshold" and "the pump is on", respectively.

LTL assertions are constructed using a set P of propositions, which refer to state-based properties, the classical connectives, $\neg, \wedge$ and $\rightarrow$, and the temporal operators $\bigcirc$ (next), $\Box$ (always), $\Diamond$ (eventually) and U (strong until). 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 $\phi \mathsf{W} \psi \equiv (\Box\phi) \vee (\phi \mathsf{U} \psi)$).

The semantics of LTL assertions is given in terms of traces (i.e. infinite sequences of states s_1, s_2, ...). A proposition p is said to be satisfied in a trace σ at position i, written $\sigma, i \models p$, if it is true at the i^{th} state in that trace (i.e. i^{th} time unit). The semantics of Boolean operators is defined in a standard way over each state in a sequence. The semantics of the temporal operators is defined as follows:

- $\sigma, i \models \bigcirc\phi$ 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 $\forall i \leq k < j.\ \sigma, k \models \phi$

Given the above semantics, a formula $\bigcirc p$ is satisfied at the i^{th} state (or time unit) of a trace σ, if p is true at the $(i+1)^{th}$ state (or time unit) of σ. An LTL assertion ϕ is said to be *satisfied in a trace* σ if and only if it is satisfied at the first state in the trace. Similarly, a set Γ of formulae is said to be satisfied in a trace σ if each formula $\psi \in \Gamma$ is satisfied in the trace σ; Γ is said to be *consistent* if there is a trace that satisfies it.

Goal models may also include domain and (partial) required conditions of operations. An *operation* causes a system to transit from one state to another. Conditions over operations can be domain *pre-condition* and *post-conditions* and/or required *pre-conditions* and *trigger-conditions*. For instance, the operation *switchPumpOn* has, as domain pre-condition and post-condition, the assertions $\neg PumpOn$ and $PumpOn$. Required pre-conditions and trigger-conditions are prescriptive conditions defining, respectively, the weakest necessary conditions and the sufficient conditions for performing an operation. The set of *required* conditions on operations to be performed by the software are called *operational requirements*

2.2. Behaviour Models

Behaviour models are event-based representations of system behaviours. Different formalisms have been proposed for modelling and analysing system behaviours (e.g. [HBGL95], [SMMM98]), among which LTS is a well known formalism for modelling systems as a set of concurrent components [MK99]. Each component is defined as a set of states and possible transitions between these states. Transitions are labelled with events denoting the interaction that the component has with its environment. The global system behaviour is captured by the parallel composition of the LTS model of each component, that interleaves their behaviour and forces synchronisation on shared events.

Definition 1. (Labelled Transition Systems) A Labelled Transition System (LTS) model is defined as a tuple $T = (S, E, s_0, \mathcal{R})$ where S is a finite non-empty set of states, E is a finite non-empty set of event labels, s_0 is the initial state, and $\mathcal{R} \subseteq S \times E \times S$ is a labelled transition relation. A transition $(s, e, s') \in \mathcal{R}$ from a state s to a new state s' labelled by e is denoted as $s \mathcal{R}_e s'$.

A (possibly infinite) *path* σ in an LTS is a sequence of states and transitions, starting from the initial state, of the form $s_0 \mathcal{R}_{e_1} s_1 \mathcal{R}_{e_2} s_2 ...$, such that for each $i \geq 0$ there is a transition relation $(s_i, e_{i+1}, s_{i+1}) \in \mathcal{R}$, with label e_{i+1}. The event $e_i \in E$ is said to be *at position i in* σ, and to be the i^{th} label in the path σ, whereas $s_i \in S$ is said to be the i^{th} state in σ. The set of all paths in T starting from the initial state is denoted as Σ.

Definition 2. (Scenarios) Given a set of event labels E, a scenario is a finite sequence of events of the form $\langle e_1, e_2, ...e_n\rangle$ where $e_i \in E$. It is said to be a *positive* scenario, and denoted as σ^+, if it represents a desirable behaviour of the system. It is said to be a *negative* scenario, and denoted as σ^-, if at least one of the events in the sequence is an undesirable occurrence. Furthermore, a scenario is said to be *accepted* by an LTS T if there is a path $\sigma = s_0\mathcal{R}_{e_1}s_1...s_{n-1}\mathcal{R}_{e_n}s_n$ in T where, for all $0 \leq i < n$, $(s_i, e_{i+1}, s_{i+1}) \in \mathcal{R}$.

Note that, in this paper, we assume that the undesirable event occurrence will always be the last event in the sequence of a negative scenario. Furthermore, we use Σ^+ and Σ^- to denote a set of positive and negative scenarios respectively

LTSA [MK99] is a tool that supports various types of automated analyses over LTSs such as model checking and animation. The logic used by LTSA is the asynchronous linear temporal logic of fluents (FLTL) [GM03]. This logic is an LTL in which propositions in P are defined as fluents. Fluents represent time varying properties of the world that are made true and false through the occurrence of events. A fluent can be either state-based or event-based. We denote the set of state-based fluents as P_f, and the set of event-based fluents as P_e. A *fluent definition* of a fluent f, denoted as $f = \langle I_f, T_f\rangle$, is a pair of disjoint sets of events, referred to as the *initiating* (I_f) and *terminating* (T_f) sets, and an initial truth value. Events of the initiating (resp. terminating) set are those events that, when executed, cause the fluent to become true (resp. false). For instance, the fluent definition for the state-based fluent *PumpOn* would be

$$PumpOn = \langle\{switchPumpOn\},\{switchPumpOff\}\rangle$$

meaning that the event *switchPumpOn* causes the fluent *PumpOn* to be true, and the event *switchPumpOff* causes the fluent *PumpOn* to be false. The fluent definition for an event-based fluent e is always defined as $e = \langle\{e\}, E - \{e\}\rangle$. For instance, the fluent definition for the event-based fluent *switchPumpOn* would be

$$switchPumpOn = \langle\{switchPumpOn\}, E - \{switchPumpOn\}\rangle$$

where E is the universe of events. An FLTL language is therefore defined in terms of a set $P = P_e \cup P_f$ of fluents and a set D of fluent definitions.

Given an FLTL language, an FLTL model is a pair $\langle T, V_D\rangle$ where T is a LTS and V_D a *valuation* function. The set of events in T is isomorphic to the set P_e of event fluents in the FLTL language, and the valuation function V_D assigns truth value to fluents over paths in T according to their fluent definition. Specifically, a fluent f is said to be true at position i in a path σ if and only if either of the following conditions hold: (i) a state-based fluent f is initially true and no terminating event has occurred since; and (ii) some initiating event has occurred before position i and no terminating event has occurred since. Note that event-based fluents are always initially false. Hence, given a path σ and a position i, the valuation function V_D returns the set of fluents that are true at i in σ with respect to D. As the satisfiability of fluents depends on their fluent definition, we use $\sigma, i \models_D f$ to denote that f is satisfied in σ at position i with respect to a given set D of fluent definitions. As formalised in the following definition, the satisfiability of asynchronous FLTL assertions over an FLTL model is also defined with respect to positions in a given path and given fluent definitions.

Definition 3. (Satisfiability in FLTL) Given an FLTL language with propositions P and set D of fluent definitions, an FLTL model $\langle T, V_D\rangle$ and a path σ in T, the satisfiability of an FLTL formula ϕ at a position $i \geq 0$ of the path σ with respect to D, denoted $\sigma, i \models_D \phi$, is defined inductively as follows:

- $\sigma, i \models_D f$ iff $f \in V_D(\sigma, i)$, where $f \in P$
- $\sigma, i \models_D \neg\phi$ iff $\sigma, i \not\models \phi$
- $\sigma, i \models_D \phi \wedge \psi$ iff $\sigma, i \models \phi$ and $\sigma, i \models \psi$
- $\sigma, i \models_D \bigcirc\phi$ iff $\sigma, i+1 \models \phi$
- $\sigma, i \models_D \Box\phi$ iff $\forall j \geq i.\ \sigma, j \models \phi$
- $\sigma, i \models_D \phi\ \mathsf{U}\ \psi$ iff $\exists j \geq i.\ \sigma, j \models \psi$ and $\forall i \leq k < j.\ \sigma, k \models \phi$

Furthermore, an FLTL formula ϕ is said to be *satisfied in a path* σ, denoted $\sigma \models_D \phi$ if and only if $\sigma, 0 \models_D \phi$. Given a set Γ of FLTL assertions, Γ is said to be s*satisfied in a path* σ, if and only if every formula $\phi \in \Gamma$ is satisfied in the path σ. An FLTL model $\langle T, V_D\rangle$ is said to *satisfy* a formula ϕ, if ϕ is

satisfied in every path σ in T. Similarly, the model is said to satisfy a set Γ of FLTL formulae, if it satisfies every FLTL formula $\phi \in \Gamma$.

Definition 4. (Entailment in FLTL) Given an FLTL language with propositions P and set D of fluent definitions, and an FLTL model $\langle T, V_D \rangle$, let Γ be a set of FLTL assertions, called also theory, and ϕ an FLTL formula. The formula ϕ is said to be entailed by the theory Γ, and denoted as $\Gamma \models_D \phi$, if and only if in every path σ where Γ is satisfied, the formula ϕ is also satisfied.

FLTL language can also be used to formalise the notion of scenarios. Note that we use symbol $\bigcirc^i$ to denote i number of the temporal operator $\bigcirc$.

Definition 5 (Scenario properties). A positive scenario $\sigma^+ = \langle e_1, ..., e_m \rangle^+$ can be formalised as the FLTL positive scenario property $\bigwedge_{1 \leq i \leq m-1} \bigcirc^i e_i \wedge \bigcirc^m e_m$, whereas a negative scenario $\sigma^- = \langle e_1, ..., e_n \rangle^-$ can be formalised as the FLTL negative scenario property $\bigwedge_{1 \leq i \leq n-1} \bigcirc^i e_i \rightarrow \bigcirc^n \neg e_n$.

Note that, by slight abuse of notation, σ^+ and σ^- will also be used to denote the FLTL properties for positive and negative scenarios respectively, and Σ^+ and Σ^- to denote a set of positive and negative scenario properties respectively.

3. Problem Formulation

In this section we discuss and exemplify why the construction of behaviour models from goal models can result in models with Zeno executions. We then formally characterise the problem that this paper addresses in Section 4.

3.1. From Goal Models to Behaviour Models

LTS models are *untimed*. To support the derivation of behaviour models from goal models, which are timed models, time must be represented explicitly in the LTSs [MK99]. We adopt the approach to discrete timed behaviour models proposed in [LKMU05]. An event *tick* is used to model the global clock with which each timed-process synchronises. The occurrence of this event signals the end of a time unit (as assumed by the goal model) and the beginning of the next. This event does not initiate or terminate any state-based fluent. A state-based fluent *Occurs_e* is also used, where needed, to denote that the event e has occurred within the last time unit. Its fluent definition is given by the tuple *Occurs_e*= $\langle \{e\}, \{tock\} \rangle$, where *tock* is an event that always immediately follows a *tick* event to terminate the *Occurs_e* fluent.

In a timed behaviour model states are classified as either *observable* or *non-observable*. The former are states with an in-going *tick* transition, and the latter are any other state. LTL assertions are assumed to be evaluated only at observable states. All timed behaviour models have a *tick* transition as first transition. This forces the first observable state of the model to have the same state-based fluents as those at the initial state. In this way, the evaluation of LTL assertions at the first observable state captures the synchronous LTL satisfiability at an initial state. To "simulate" synchronous LTL semantics within the context of timed behaviour models, LTL assertions are translated into asynchronous FLTL formulae, adopting the methodology given in [LKMU06] and briefly described below.

The transformation of a goal model, G_s, into a set, G_a, of asynchronous FLTL assertions, requires translating the LTL goals and operational requirements that are in G_s into FLTL formulae. Goal assertions, written in LTL, are translated into semantically equivalent asynchronous FLTL expressions using the event fluent *tick*. The translation $Tr : LTL \rightarrow FLTL_{Async}$ is defined as follows (where ϕ and ψ are state-based LTL assertions):

$$
\begin{array}{lcl}
Tr(\Box\phi) & = & \Box(tick \rightarrow Tr(\phi)) \\
Tr(\phi \mathsf{U} \psi) & = & tick \rightarrow Tr(\phi)\ \mathsf{U}\ (tick \wedge Tr(\psi)) \\
Tr(\Diamond\phi) & = & \Diamond(tick \wedge Tr(\phi)) \\
Tr(\bigcirc\phi) & = & \bigcirc(\neg tick\ \mathsf{W}\ (tick \wedge Tr(\phi)))
\end{array}
$$

The translation of the synchronous (LTL) next operator (see $Tr(\bigcirc\phi)$) exemplifies well the difference between

synchronous and asynchronous semantics. The synchronous formula $\bigcirc\phi$ asserts that at the next time point ϕ is true. The translation assumes that the formula $Tr(\bigcirc\phi)$ is evaluated at the start of a time unit, in other words at the occurrence of a *tick*, and requires that no *tick* occurs from that point onwards until the asynchronous translation of ϕ holds and the *tick* event can occur. Consider the synchronous system goal *PumpOffWhenLowWater* formalised in (1) as $\Box(\neg HighWater \rightarrow \bigcirc\neg PumpOn)$. Its translation gives the asynchronous FLTL assertion

$$\Box(tick \rightarrow (\neg HighWater \rightarrow \bigcirc(\neg tick \ \mathsf{W} \ (tick \wedge \neg PumpOn))))$$

Operations specified in the goal model, G_s, correspond to the events in the timed behaviour model; domain properties and operational requirements are then translated into asynchronous FLTL assertions using the associated event-based fluents. A domain pre-condition *DomPre*, for an operation e, in G_s, is represented in G_a by the following asynchronous FLTL assertion:

$$\Box(tick \rightarrow ((\neg DomPre) \rightarrow \bigcirc(\neg e \ \mathsf{W} \ tick))) \tag{2}$$

For instance, the domain pre-condition $\neg PumpOn$ for the event *switchPumpOn* is expressed as

$$\Box(tick \rightarrow (\neg(\neg PumpOn) \rightarrow \bigcirc(\neg switchPumpOn \ \mathsf{W} \ tick)))$$

The domain post-condition *DomPost* for an operation e, captured in G_s by the fluent definitions, is represented in asynchronous FLTL as $\Box(e \rightarrow (DomPost))$. The FLTL formalisation of required pre-conditions, *ReqPre*, in G_a is analogous to that of domain pre-conditions, namely

$$\Box(tick \rightarrow ((\neg ReqPre) \rightarrow \bigcirc(\neg e \ \mathsf{W} \ tick))) \tag{3}$$

where the FLTL representation of required trigger-condition, *ReqTrig*, for an operation e is given by

$$\Box(tick \rightarrow ((ReqTrig \ \wedge \ DomPre) \rightarrow \bigcirc(\neg tick \ \mathsf{W} \ e))) \tag{4}$$

An LTS model can then be computed from the asynchronous FLTL representation of a given goal model. This is done using an adaptation [LKMU06] of a "temporal logic to automata" algorithm used in model checking of FLTL [GM03]. The technique for model-checking an asynchronous FLTL property ϕ over an LTS T involves constructing a Büchi automaton $B(\neg\phi)$ that recognises all infinite traces, over the alphabet L, that violate ϕ and checking that the synchronous product of $B(\neg\phi)$ with T is empty [GM03]. When ϕ is a safety property, which is the case for goal models, $B(\neg\phi)$ has only one accepting state that only has self-loop transitions as outgoing transitions. Thus, $B(\neg\phi)$ can be viewed as an observer for ϕ, i.e., an LTS with an error state in which reaching the error state corresponds violating ϕ. Removing the error state and all transitions that lead to it yields an LTS that is *as least constrained as possible*, yet that is guaranteed to satisfy ϕ. Consider the model in Figure 1. This is the Büchi automata resulting from the formula

$$\phi = \Box(tick \rightarrow (\neg HighWater \rightarrow \bigcirc(\neg tick \ \mathsf{W} \ (tick \wedge \neg PumpOn))))$$

which is the FLTL translation of the LTL goal $\Box(\neg HighWater \rightarrow \bigcirc\neg PumpOn)$. Removing the accepting state labelled -1, yields an LTS that is guaranteed to satisfy the formula ϕ.

The above-mentioned procedure for constructing LTS models from asynchronous FLTL representation of goal models can be applied to each assertion in the goal model individually. The resulting LTS models can then be composed in parallel, so capturing logical conjunction [LKMU06]. However, in so doing, it is possible that the resulting LTS model may exhibit problematic behaviours in the form of traces in which a finite number of ticks occur. These traces, called *Zeno traces* or *Zeno-executions*, represent behaviours in which time does not progress. We refer to LTS models with Zeno traces as *Zeno models*.

3.2. The Problem with Zeno Models

To summarise, given a goal model, G_s, as described in Section 2.1, it is possible to construct a timed LTS model that satisfies G_s by transforming the goal model into a semantically equivalent set of asynchronous FLTL assertions, G_a, and then computing an LTS model, using an adaptation [LKMU06] of the temporal logic to automata algorithms in [GM03], that satisfies G_a.

However, the LTS models constructed from asynchronous FLTL goal assertions are not *good models* of a

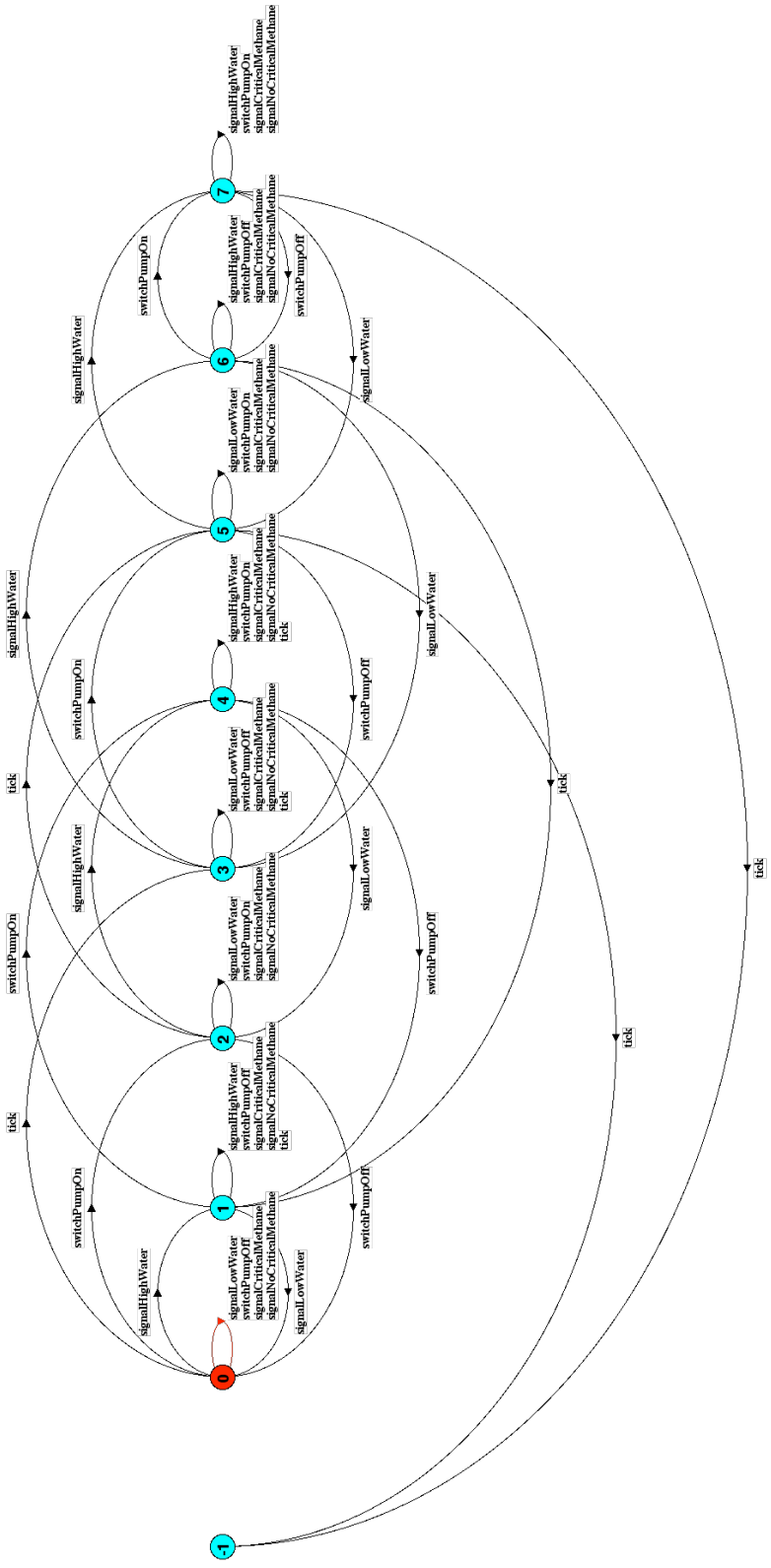

Fig. 1. Büchi automata constructed from the goal *PumpOffWhenLowWater*

system behaviour. They constrain the event *tick*, which should not be constrained as it cannot be controlled by the system, and do not impose sufficient constraints on the system events. Such constraints are those introduced by the domain and required conditions for operations that are included in the initial goal model. So, if the considered goal model has insufficient conditions on the system events, as it is often the case in practice, then spurious executions may be exhibited by the LTS model generated from the goal model. For instance, the LTS model for the goal *PumpOffWhenLowWater*, given in Figure 1, includes the infinite trace ⟨`tick, switchPumpOn, switchPumpOff, switchPumpOn, switchPumpOff,...`⟩. This trace does not exhibit a second tick events so violating the expectation that time progresses (referred to as *time progress property*). Such trace occurs because there is no restriction as to when the pump may be switched on or off (i.e. required pre-condition for *switchPumpOn* and/or *switchPumpOff* is missing). Therefore, although the LTS model constructed automatically from an asynchronous FLTL encoding of a goal model may satisfy all goals, it may contain Zeno executions due to missing conditions over system operations. Note that this is not a problem caused by the translation but rather caused by the under-specification in the synchronous description of what happens *within* the time units [LKMU06].

The problem addressed in this paper is how to provide automated support for extending a goal model with required pre-conditions over system's operations in order to guarantee the construction of a non-Zeno behaviour model from a given goal model. In other words, given the asynchronous transformation G_a of a synchronous goal model G_s, we want to compute a set, $\{Pre_i\}$, of required pre-conditions that is consistent with G_a and such that the LTS model constructed from $G_a \cup \{Pre_i\}$ satisfies the *time progress* (TP) property (i.e. $\Box\Diamond tick$). This is formally defined below.

Definition 6. (Correct Operational Extension) Let G_a be an asynchronous goal model and TP be the time progress property defined as the FLTL assertion $\Box\Diamond tick$. Then a set $\{Pre_i\}$ of required pre-conditions is said to be a *correct operational extension* of G_a iff the following conditions hold:

- $G_a \cup \{Pre_i\} \models_D TP$
- $G_a \cup \{Pre_i\} \cup TP \not\models_D false$

4. The Approach

This section presents a novel approach for extending a goal model with the set of required pre-conditions necessary to generate a non-Zeno behaviour model that satisfies the given goal model. The approach uses model checking to analyse the LTS of the given goal model with respect to the time progress property and to generate automatically any existing Zeno traces, and uses ILP to compute the required pre-conditions that eliminate such traces. A brief overview of the framework is first given, followed by a detailed description of each of its phases.

4.1. Overview of the Approach

An overview of the approach is depicted in Figure 2. A goal model G_s is initially transformed into an asynchronous model G_a and a set of fluent definitions D. The computation of the correct operational extension of G_a is then done by iterating over four phases. (1) The *Analysis* phase uses the LTSA model checking to construct an LTS model of G_a with respect to D and then to check the LTS against the time progress property. If the property does not hold, a violation trace is generated. (2) The *Scenario Elaboration* phase requires the engineer to elaborate the violation trace into a set of positive and negative scenarios. (3) The *Learning* phase transforms the asynchronous goal model, fluent definitions and scenarios into a logic program and then uses an ILP system to find a set of required pre-conditions that cover the positive but none of the negative scenarios. (4) The *Selection* phase then requires the engineers to select the pre-condition(s) to be added to G_a from those computed during the learning phase. These four phases are repeated until no more violation traces are detected. The final output is an extended FLTL goal model from which a non-Zeno behaviour model can be constructed.

The first step, as depicted in Figure 2, involves translating the synchronous goal model into an FLTL asynchronous goal model using the translation process presented in Section 3. In our running example, the full set G_s includes the system goals given by the following LTL assertions:

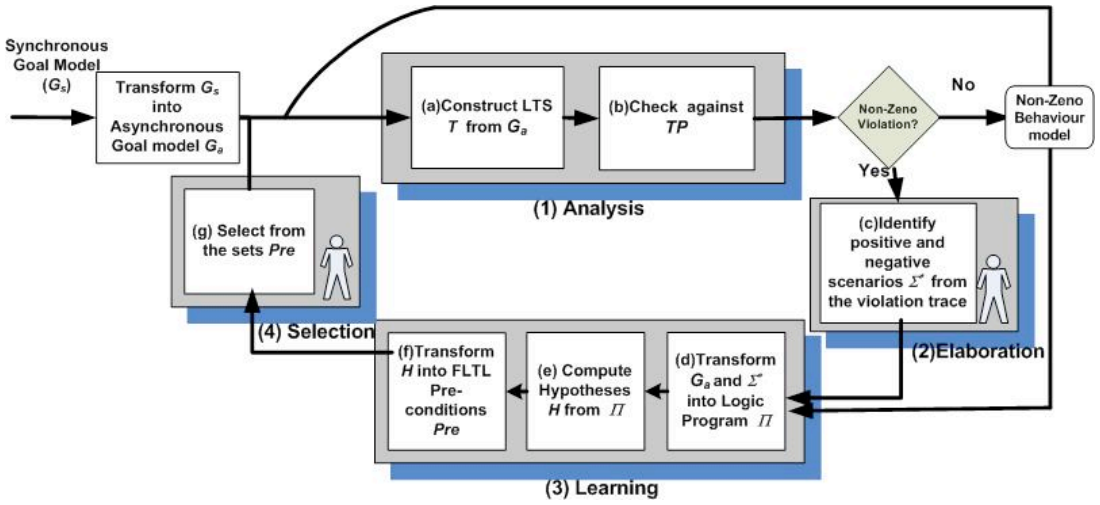

Fig. 2. Overview of the approach

$$g[PumpOffWhenLowWater]=$$
$$\Box(\neg HighWater \rightarrow \bigcirc\neg PumpOn) \tag{5}$$

$$g[PumpOffWhenMethane]=$$
$$\Box(CriticalMethane \rightarrow \bigcirc\neg PumpOn) \tag{6}$$

$$g[PumpOnWhenHighWaterAndNoMethane]=$$
$$\Box(\neg CriticalMethane \wedge HighWater \rightarrow \bigcirc PumpOn) \tag{7}$$

and the required pre-conditions ($HighWater$) and ($CriticalMethane \vee \neg HighWater$) for the events *switchPumpOn* and *switchPumpOff* respectively. The goal assertions are translated into the following asynchronous FLTL formulae:

$$g_a[PumpOffWhenLowWater]=$$
$$\Box(tick \rightarrow (\neg HighWater \rightarrow \bigcirc(\neg tick \,\mathsf{W}\, (tick \wedge \neg PumpOn)))) \tag{8}$$

$$g_a[PumpOffWhenMethane]=$$
$$\Box(tick \rightarrow (CriticalMethane \rightarrow \bigcirc(\neg tick \,\mathsf{W}\, (tick \wedge PumpOn)))) \tag{9}$$

$$g_a[PumpOnWhenHighWaterAndNoMethane] =$$
$$\Box(tick \rightarrow ((\neg CriticalMethane \wedge HighWater) \rightarrow \bigcirc(\neg tick \,\mathsf{W}\, (tick \wedge PumpOn)))) \tag{10}$$

The required pre-condition are expressed asynchronously as follows:

$$\Box(tick \rightarrow (\neg HighWater \rightarrow \bigcirc(\neg switchPumpOn \,\mathsf{W}\, tick))) \tag{11}$$

$$\Box(tick \rightarrow ((HighWater \wedge \neg CriticalMethane) \rightarrow \bigcirc(\neg switchPumpOff \,\mathsf{W}\, tick))) \tag{12}$$

4.2. Analysis Phase

This phase takes as input an asynchronous FLTL encoding of a goal model and produces a Zeno trace if the goal model does not guarantee time progress.

The LTSA model checker is used to build automatically the least constrained LTS model from the asynchronous FLTL assertions [GM03]. LTSA is then used to verify that time progresses by checking the property $\Box\Diamond tick$ against the model. The output of LTSA, in the case of a Zeno model, is an infinite trace in which, from one position onwards, no *tick* event occurs. The check is performed by assuming maximal progress of the system with respect to the environment (a standard assumptions of reactive systems), and weak fairness [MK99]. Fairness ensures that the environment will eventually perform *tick* instead of other environment controlled events (assuming the environment itself is consistent with time progress).

In our running example, the application of the analysis phase to the asynchronous FLTL goal model (i.e. equations (8)–(12)) gives the following output, where the capitalised text on the right column indicates the fluents that are true after the occurrence of each event of the trace prefix on the left.

```
Violation of LTL property: Non_Zeno
Trace to terminal set of states:
tick
tock
signalCriticalMethane           CRITICALMETHANE
signalHighWater                 HIGHWATER && CRITICALMETHANE
tick                            HIGHWATER && CRITICALMETHANE
tock                            HIGHWATER && CRITICALMETHANE
switchPumpOn                    HIGHWATER && CRITICALMETHANE && PUMPON
switchPumpOff                   HIGHWATER && CRITICALMETHANE
Cycle in terminal set:
switchPumpOn
switchPumpOff
LTL Property Check in: 8ms
```

The above is an infinite trace compactly displayed as a (finite) trace with the prefix

⟨tick,tock, signalCriticalMethane, signalHighWater, tick, tock, switchPumpOn, switchPumpOff⟩

followed by the cycle ⟨switchPumpOn, switchPumpOff, switchPumpOn,...⟩, in which *tick* does not occur.

The trace indicates that a pre-condition for at least one of two system controlled events, *switchPumpOn* and *switchPumpOff*, is missing or requires strengthening. Indeed, consider the second *tick* of the trace, where *HighWater* and *CriticalMethane* are true. At this point the goals proscribe *tick* from occurring while *PumpOn* is still true. Hence, the occurrence of *switchPumpOff* is desirable. However, as soon as *switchPumpOff* happens, there are no pre-conditions preventing the pump being switched on again. Note that switching the pump back on does not violate any goals as the requirement is that the pump be off at the next *tick* and nothing is stated about the number of times the pump may be switched on during the time unit. A reasonable outcome of this analysis is to conclude that the pre-condition for *switchPumpOn* needs strengthening to prevent the pump being switched on unnecessarily.

4.3. Scenario Elaboration Phase

During this phase, the engineer elaborates the violation trace generated by the LTSA, and produces a set of positive and negative scenarios. The engineer is assumed to identify, within the violation trace returned by the LTSA, the event in the trace that should not have occurred at a particular position in that trace. The prefix starting from the initial state of that trace up to and including the undesirable event is denoted as a *negative scenario*. So, given a violation trace of the form $\langle w_1, e, w_2\rangle$, where e is the undesirable event, a negative scenario would be $\langle w_1, e\rangle$, denoting that whenever the system exhibits w_1 then event e should not happen. The task of producing a negative scenario from the trace returned by the LTSA is believed to be an intuitive task that can be performed manually, in particular because the negative scenarios will always be sub-traces of the trace produced by the LTSA.

The engineer is also assumed to provide at least one scenario which shows a positive occurrence of e. This

should be a scenario that starts from the same initial state, is consistent with the goal model and terminates with a tick (i.e. $\langle w_1^{'}, e, w_2^{'}, tick \rangle$), where $w_1^{'}$ and/or $w_2^{'}$ may well be empty. The same model generated by LTSA can be walked through or animated by the engineer to generate such positive scenarios, guaranteeing their consistency with the given goals and operational requirements. Note that because scenarios are finite traces, positive scenarios are not meant to exemplify non-Zeno traces. They merely capture desirable system behaviours which are consistent with the given goal model.

Returning to the example of Zeno trace described above, the engineer may identify the first occurrence of `switchPumpOn` as incorrect. A negative scenario could therefore be

$\sigma_1^- = \langle$`tick, tock, signalCriticalMethane, signalHighWater, tick, tock, switchPumpOn`$\rangle$

stating that the pump should not have been switched on after high water and methane have been signalled. In addition, a positive scenario exemplifying a correct occurrence of `switchPumpOn` could be:

$\sigma_1^+ = \langle$`tick, tock, signalHighWater, tick, tock, switchPumpOn, tick`$\rangle$.

This phase finishes when at least one positive and one negative scenarios have been identified. Note that more than one positive and/or negative scenarios can be provided during this phase (e.g. positive examples from previous iterations).

4.4. Learning Phase

This phase is concerned with the inductive learning computation of missing required pre-conditions with respect to a given collection $\Sigma_P \cup \Sigma_N$ of positive and negative scenarios. It makes use of an Inductive Logic Programming (ILP) framework, called non-monotonic Hybrid Abductive and Inductive Learning (XHAIL) [Ray09]. In general, an inductive learning task is defined as the computation of an hypothesis H that explains a given set E of examples with respect to a given background knowledge B [RBR04, Mug95]. The hypothesis H is called, in this case, *an inductive solution* for E with respect to B. Intuitively, within the context of learning pre-conditions, the background knowledge is the fluent definitions D and the goal model G_a, generated during the analysis phase. The set of positive and negative scenarios constructed during the scenario elaboration phase form the examples. The learned (set of) asynchronous pre-conditions, Pre, is the hypothesis that added to G_a generates an LTS model that accepts the positive scenarios but none of the negative ones. This can be formally defined below with respect to the scenario properties.

Definition 7 (Correct Operational Extension with respect to Scenarios). Let G_a be an asynchronous goal model and let Σ^+ and Σ^- be, respectively, sets of positive and negative scenario properties. The (set of) asynchronous pre-conditions Pre is said to be a *correct extension of a goal model with respect to scenarios* Σ^+ and Σ^-, if the following conditions holds:

- $G_a \cup Pre \models_D \sigma^-$ for each $\sigma^- \in \Sigma^-$
- $G_a \cup Pre \not\models_D \neg\sigma^+$ for each $\sigma^+ \in \Sigma^+$.

To apply XHAIL to the task of learning pre-conditions, the current asynchronous FLTL goal model G_a (including pre-conditions computed in previous iterations), fluent definitions D, and the collection of positive and negative scenarios, $\Sigma^+ \cup \Sigma^-$, are encoded into a semantically equivalent Event Calculus (EC)[MS02] logic program Π and set of examples E.

Event Calculus Programs. Our EC programs include a sort A of events (e_1,e_2,...), a sort F of fluents (f_1,f_2,...), a sort S of scenarios (s_1,s_2,...), and two sorts $P = (p_1,p_2,..)$ and $T = (t_1,t_2,..)$ both isomorphic to the set of non-negative integers. Each of the sorts P and T represents, respectively, positions and time units along a trace. EC programs make use of the basic predicates `happens`, `initiates`, `terminates`, `holdsAt`, `impossible` and `attempt`. The atom `happens(e,p,t,s)` indicates that event `e` occurs at position `p`, within time unit `t` in scenario `s`, the atom `initiates(e, f, p, s)` (resp. `terminates(e,f,p,s)`) means that if, in a scenario `s`, event `e` were to occur at position `p`, it would cause fluent `f` to be true (resp. false) immediately afterwards. The predicate `holdsAt(f,p,s)` denotes, instead, that in a scenario `s`, fluent `f` is true at position `p`. The atoms `impossible(e,p,t,s)` and `attempt(e,p,t,s)` are used, respectively, to state that in a scenario `s`, at position `p` within a time unit `t`, the event `e` is impossible, and that an attempt has

been made to perform `e`. The first four predicates are standard, whereas the last two are adapted from the EC extension presented in [MS02].

To relate positions and time units within a given scenario, the predicate `posInTime(p,t,s)` is used, which denotes that, in a given scenario `s`, a position `p` is within a time unit `t`. Specifically, the time unit of a position $p \geq 0$ in a scenario is the number of tick occurrences between position 0 and p. A additional predicate `next(`x_2`, `x_1`)` is also used to denote that x_2 is the next time (resp. position) unit after x_1. For example, for the following scenario

⟨`tick, tock, signalHighWater, tick, tock, switchPumpOn, tick`⟩

The EC program would include the following relations between position and time:

```
posInTime(0,1,s).
posInTime(1,1,s).
posInTime(2,1,s).
posInTime(3,2,s).
posInTime(4,2,s).
posInTime(5,2,s).
posInTime(6,2,s).
posInTime(7,3,s).
```

where s is a constant that uniquely represents the above sequence in the EC program, and the facts `next(i+1,i)`, for every $0 \geq i \leq 6$, which defines the ordering relation between time units and positions.

EC programs are equipped with a set of domain-independent core axioms suitable for reasoning about effects of events over fluents.

```
clipped(P1,F,P2,S) :- happens(E,P,T,S),
                      terminates(E,F,P,S), P1 < P < P2.          (13)

holdsAt(F,P2,S) :- happens(E,P1,T,S), initiates(E,F,P1,S),
                   P1 < P2, not clipped(P1,F,P2, S).             (14)

holdsAt(F,P,S) :- initially(F,S), not clipped(0,F,P,S).          (15)

happens(E,P,T,S) :- attempt(E,P,T,S),
                    not impossible(E,P,T,S),
                    posInTime(P,T,S).                            (16)
```

The three axioms (13)-(15) describe general principles for deciding when fluents hold or do not hold at particular time-points.[1] They formalise the common-sense law of inertia: a fluent that is true continues to hold until a terminating event occurs, and vice versa. The last axiom (16) captures the semantics of event pre-conditions. It states that an event E *cannot* happen if its pre-conditions are not satisfied (i.e. *impossible* is true)[2].

Furthermore, to capture the notion of synchronous satisfiability in terms of asynchronous semantics our programs make use of two new predicates `holdsAtTick` and `notholdsAtTick`. These are defined by the following additional domain-independent core axioms.

```
holdsAtTick(F,T,S):- attempt(tick,P,T,S), next(P2,P1), holdsAt(F,P2,S).      (17)

notholdsAtTick(F,T,S) :- attempt(tick,P,T,S), next(P2,P1), not holdsAt(F,P2,S).   (18)
```

Axioms (17) and (18) state that a fluent `F` holds (resp. does not hold) at the beginning of a time unit `T` in a scenario `S` if it holds (resp. does not hold) at the position `P` where the `tick` event is attempted.

In addition to the domain-independent core axioms, EC programs are equipped with *domain-dependent axioms* and *integrity constraints*. The former define the predicates `initiates`, `terminates`, `impossible`,

[1] Axioms (13)–(15) are identical to those presented in [Sha97] apart from the extra argument T and S, for representing time units and scenarios. Axiom (16) extends the formalism in [Sha97] to support event pre-conditions.

[2] The symbol `:-` is used in Prolog to denote the implication ←.

`attempt`, and `posInTime`, depending on the particular problem in hand, and are automatically generated from the fluent definition D of the FLTL goal model as described below. The *integrity constraints* are instead used to express constraints over the `holdsAtTick` and `notholdsAtTick` predicates, which capture the FLTL goal models. These constraints are represented as *denial* rules – i.e. rules of the form `:-` $\phi_1, \ldots, \phi_n$, where, ϕ_i can in principle be any of the EC predicates.

From Goal Models to EC Programs. Given a goal model, G_a, written in asynchronous FLTL, and a set D of fluent definitions, a mapping τ has been defined that automatically generates from G_a and D an EC program. The definition of this mapping is given below.

Definition 8. (Encoding Goal Models into EC Programs) Given an asynchronous goal model G_a and a set D of fluent definitions, the *corresponding* logic program $\Pi = \tau(G_a, D)$ is the EC program containing the following (atomic) clauses:

- `initially(`$\mathtt{f}_i$`,S)`, for each fluent defined to be initially true in D
- `initiates(`$\mathtt{e}_i$`,f,P,S)`, for each event $\mathtt{e}_i \in I_f$ in D
- `terminates(`$\mathtt{e}_i$`,f,P,S)`, for each event $\mathtt{e}_i \in T_f$ in D
- `impossible(e,P,T,S):-(not)holdsAtTick(`$\mathtt{f}_1$`,T,S),..,`
  `(not)holdsAtTick(`$\mathtt{f}_n$`,T,S), posInTime(P,T,S)`
  for each operational pre-condition $\Box(tick \rightarrow ((\bigwedge_{1 \leq i \leq n}(\neg)f_i) \rightarrow (\bigcirc \neg e \ \mathsf{W} \ tick)))$ in G_a
- `:- (not)holdsAtTick(`$\mathtt{f}_1$`,T,S),..., (not)holdsAtTick(`$\mathtt{f}_n$`,T,S),`
  `next(T2,T), notholdsAtTick(g,T2,S)`
  for each goal $\Box(tick \rightarrow ((\neg)f_1 \wedge \ldots \wedge (\neg)f_n \rightarrow \bigcirc \neg tick \ \mathsf{W} \ (tick \wedge \neg g)))$ in G_a
- `:-(not)holdsAtTick(`$\mathtt{f}_1$`,T,S),...,(not)holdsAtTick(`$\mathtt{f}_n$`,T,S),`
  `next(T2,T), holdsAtTick(g,T2,S)`
  for each goal $\Box(tick \rightarrow ((\neg)f_1 \wedge \ldots \wedge (\neg)f_n \rightarrow \bigcirc \neg tick \ \mathsf{W} \ (tick \wedge g)))$ in G_a
- EC domain-independent axioms (13)-(18)

For example, the mapping function τ would generate from the fluent definition

$$pumpOn \equiv \langle \{switchPumpOn\}, \{switchPumpOff\} \rangle$$

the facts

```
initiates(switchPumpOn,pumpOn,P,S).
terminates(switchPumpOff,pumpOn,P,S).
```

from the required pre-condition

$$\Box(tick \rightarrow (\neg HighWater \rightarrow \bigcirc(\neg switchPumpOn \ \mathsf{W} \ tick)))$$

the clause

```
impossible(switchPumpOn,P,T,S):- notholdsAtTick(highWater,T,S),
posInTime(P,T,S).
```

referred to as an EC pre-condition, and from the asynchronous FLTL goal *PumpOffWhenLowWater*

$$\Box(tick \rightarrow (\neg HighWater \rightarrow \bigcirc(\neg tick \mathsf{W}(tick \wedge \neg PumpOn))))$$

the integrity constraint

```
:- notholdsAtTick(highWater,T,S),next(T2,T),holdsAtTick(pumpOn,T2,S).
```

The learning phase provides also an automatic encoding of positive (Σ^+) and negative (Σ^-) scenarios into two sets of EC facts. The first set, called Nar for *narrative*, includes the encoding of positions and time units ordering as facts of the form `posInTime(p,t,s)` and `next(t+1,t)`, and the encoding of each event transition as a fact of the form `attempt(e,p,t,s)`. The second set constitutes the set E of positive and

negative examples that the learning system uses to compute new operational requirements, and it is defined later in the section.

A formal definition of the transformation of scenarios into narratives is given below.

Definition 9. (Encoding of scenarios into Narrative) Let $\sigma = \langle e_1, e_2, ..., e_n \rangle$ be a scenario. A corresponding EC *narrative* is the program *Nar* composed of the facts `attempt(e`$_i$`, i-1, t, s)` and `posInTime (i-1,t,s)` for each `e`$_i$ in `s`, where $1 \leq i \leq n$, and `t` is the number of *tick* transitions in `s` from position 0 until position i inclusive in σ.

For example, consider the positive scenario σ_1^+:

$\langle$`tick, tock, signalHighWater, tick, tock, switchPumpOn, tick`$\rangle^+$

The EC encoding gives the narrative Nar:

```
attempt(tick,0,1,spos_1).
attempt(tock,1,1,spos_1).
attempt(signalHighWater,2,1,spos_1).
attempt(tick,3,2,spos_1).
attempt(tock,4,2,spos_1).
attempt(switchPumpOn,5,2,spos_1).
attempt(tick,6,3,spos_1).
posInTime(0,1,spos_1).
posInTime(1,1,spos_1).
posInTime(2,1,spos_1).
posInTime(3,2,spos_1).
posInTime(4,2,spos_1).
posInTime(5,2,spos_1).
posInTime(6,3,spos_1).
next(1,0).
next(2,1).
next(3,2).
next(4,3).
next(5,4).
next(6,5).
```

Note that from now on, the constants *sneg_h* and *spos_h* are used to refer to the positive and negative scenarios σ_h^- and σ_h^+ where $h \in \mathcal{N}$.
The following theorem proves that the encoding of FLTL goal models and scenarios into EC programs, given in Definitions 8 and 9 respectively, is sound.

Theorem 1. Let G_a be a goal model expressed in asynchronous FLTL and T an LTS model that satisfies G_a. Let $\langle e_1, e_2, \ldots, e_n \rangle$ be a scenario accepted by T, and σ its associated path in T. Let Nar be the corresponding encoding in EC of the scenario, and Π be the EC logic program $\Pi = \tau(G_a, D) \cup Nar$. This program has a unique stable model I. Then, for each fluent f in the FLTL language and position p in the scenario, $\sigma, p \models_D f$ iff $holdsAt(f, p, s)$ is true in I; for every event-based fluent e in the FLTL language, position p and associated time unit t in the scenario, $\sigma, p \models_D e$ iff $happens(e, p-1, t, s)$, is true in I.

Proof:
The proof is by induction of the position i in time unit j in σ using the fact that Π is a locally stratified program and, as such, has a unique stable model. Let Π^I be the reduct of the program Π, so that I is its minimal (Herbrand) model of Π^I. We consider three base cases for positions $i = 0$, $i = 1$ and $i = 2$, respectively. The first base case is to prove the soundness with respect to state-based fluent propositions at the initial state (i.e. at position 0), the second is with respect to the *tick* event-based proposition at position 1 (as this is the only event-based fluent at position 1 different from those at position 0), and the third case is the actual based case for any arbitrary event-based and fluent-based proposition.

Base cases

- $i = 0$:
  For any $e \in P_e$, $\sigma, 0 \not\models e$ by Definition 3, and no ground atom $happens(e, -1, 0, s)$ is in I, since the time constant -1 is outside the scope of the EC sort P of positions. Hence, $\sigma, 0 \models e$ iff $happens(e, -1, 0, s) \in I$. As for the case $\sigma, 0 \models f$ iff $holdsAt(f, 0, s) \in I$, we consider the "if case" first and assume that $\sigma, 0 \models f$ for a given state-based fluent f. Since our D includes complete information on the initial state, we have that $f \in S_0$. Hence the program Π, and therefore I, will contain the ground atom $initially(f, s)$. Moreover, $clipped(0, f, 0, s) \notin I$, as there is no t, $0 < t < 0$. Then, Π^I will contain the ground clause $holdsAt(f, 0, s) \leftarrow initially(f, s)$ so giving $holdsAt(f, 0, s) \in I$. For the "only if" case, we know that $holdsAt(f, 0, s) \in I$ and assume, reasoning by contradiction, that $\sigma, 0 \not\models f$. Therefore, f is assumed to be initially false in D. In this case, Π will contain the ground atom $initially(f, s)$, as well as any fact $happens(e, -1, 0, s)$. Hence, the program Π^I will have no ground definition of $holdsAt(f, 0, s)$, which implies that $holdsAt(f, 0, s) \notin I$, giving a contradiction.
- $i = 1$:
  In this case, it is sufficient to show $\sigma, 1 \models tick$ iff $happens(tick, 0, 1, s) \in I$. Let's consider the "if case". Given that the scenario is accepted by T and that $\sigma, 1 \models tick$, then there is a transition $s_0 \mathcal{R}_{tick} s_1$ in σ and *tick* is the first event in the scenario s. Therefore $attempt(tick, 0, 1, s)$ and $posInTime(0, 1, s)$ are in I. Moreover, since G_a does not contain any pre-condition for the event *tick*[3], the program Π^I does not contain ground definitions of *impossible(tick,0,1,s)*. So, by the domain-independent core axiom (16) in Π^I, we have that $happens(tick, 0, 1, s) \in I$. Consider now the "only if" case and let us assume that $happens(tick, 0, 1, s) \in I$ and, reasoning by contradiction, that $\sigma, 1 \not\models tick$. So the first transition in σ will be different from *tick* which contradicts the fact that every path in an asynchronous LTS T always starts with a *tick* transition.

  We need now to show that $\sigma, 1, \models f$ iff *holdsAt(f,1,s)* for any state-based fluent f different from *tick*. Assume that $\sigma, 1 \models f$, for an arbitrary fluent f different from *tick*. Since the first transition in σ is a *tick*, which does not initiate or terminate any state-based fluent different from *tick*, $\sigma, 1, \models f$ iff $\sigma, 0 \models f$. Therefore, *initially(f,σ)* $\in I$. Moreover, since *tick* does not terminates f, *terminates(tick,f,0,s)* $\notin I$ and $clipped(0, f, 1, s) \notin I$. Hence, $holdsAt(f, 1, s) \leftarrow initially(f, s), 0 < 1$ is in Π^I, and then $holdsAt(f, 1, s) \in I$. The proof for the "only if" case is similar.
- $i = 2$:
  We show that $\sigma, 2 \models e$ iff *happens(e,1,j,s)* $\in I$ for any event e. We consider the "if case" first and assume that $\sigma, 2 \models e$ for an arbitrary event e. Then, there will be a transition $s_1 \mathcal{R}_e s_2$ in σ. We distinguish two cases, namely e is a tick event and e is a *non-tick* event. The proof for the first case is similar to the base case for $(i = 1)$. Let us assume then that $e \in P_e - \{tick\}$. This means that the antecedents of all pre-conditions in G_a for the event e are false at position 1 (i.e. at the previous tick) in σ. Then, by the previous base case, $impossible(e, 1, 1, s) \notin I$. Hence, by the domain-independent axiom (16) in Π^I and the fact that $attempt(e, 1, 1, s)$ and $posInTime(1, 1, s)$ are in I, we have that $happens(e, 1, 1, \sigma) \in I$.

  We now show that $\sigma, 2 \models f$ iff *holdsAt(f,2,s)* $\in I$. We assume that $\sigma, 2 \models f$ for an arbitrary fluent f. This implies that f is either (a) initially true (i.e $\sigma, 0 \models f$) and no event in T_f occurs at 2, or (b) the fluent f has been initiated by an event in I_f at 2. Consider case (a). The program Π^I contains then the ground atom $initially(f, s)$. Moreover, since $\sigma, 2 \not\models e_{T_f}$ for any terminating event $e_{T_f} \in T_f$ then, by the second base case, $happens(e_{T_f}, 1, j, s) \notin I$ (where $j = 1$ if $\sigma, 2 \not\models tick$ and $j = 2$ otherwise). Thus, $clipped(0, f, 2, s) \notin I$, and Π^I contains the ground clause $holdsAt(f, 2, s) \leftarrow initially(f, s)$. Hence, $holdsAt(f, 2, s) \in I$. We now consider case (b). We have that $\sigma, 2 \models e_{I_f}$. Therefore, $happens(e_{I_f}, i, 1, s) \in I$[4]. We also know that $\sigma, 2 \not\models e'$ for any $e' \in P_e - \{e_{I_f}\}$, including any event that terminates f. Hence, $happens(e_{T_f}, 1, 1, s) \notin I$ for any terminating event in T_f. This implies that $clipped(0, f, 2, s) \notin I$ and $holdsAt(f, 2, s) \leftarrow initiates(e_{I_f}, f, 1, s), happens(e_{I_f}, 1, 1, s), 1 < 2$ is in Π^I. Since $initiates(e_{I_f}, f, 1, s) \in I$ also $holdsAt(f, 2, s) \in I$. The proof of the "only if" case is similar.

Inductive Hypothesis (IH):
We assume that for any position k, $0 \leq k \leq i - 1$ and for any event e, we have $\sigma, k \models e$ iff $happens(e, k - 1, h, s) \in I$ where $k - 1$ is position in time unit h, and for any fluent f $\sigma, k \models f$ iff $holdsAt(f, k, s) \in I$. We

[3] This is because G_a is assumed to include only pre-conditions for system events.
[4] Note that we can assume that non-tick event occurs at 1 and at 2 since *tick* does not initiates fluent f.

want to show this is true for position $k = i$.

Inductive Step:

We want to show that $\sigma, i \models e$ iff $happens(e, i-1, h, s) \in I$ where $i-1$ is a position in time unit h. We consider the "if case" first and assume that $\sigma, i \models e$. This means that a transition $s_{i-1}\mathcal{R}_e s_i$ exists in σ. Then the antecedent of any pre-condition in G_a for the event e is false at the previous *tick* in σ. Given that the program Π contains pre-condition clauses on e, by (IH) their respective antecedents are also not satisfied at the last *tick* and hence $impossible(e, i-1, h, s) \notin I$ for all pre-condition clauses on e. Now given that $attempt(e, i-1, h, s)$ and $posInTime(i-1, h, s)$ are true in I, we can conclude that $happens(e, i-1, h, s) \in I$. For the "only if" case, we assume that $happens(e, i-1, h, s) \in I$ and that, reasoning by contradiction, $\sigma, i \not\models e$. Since e is not satisfied at position i of σ, Π does not contain the fact $attempt(e, i-1, h, s)$. So $happens(e, i-1, h, s) \notin I$ which contradicts the initial assumption.

We need now to show that $\sigma, i \models f$ iff $holdsAt(f, i, s) \in I$. We consider the "if case" first and assume that $\sigma, i \models f$. Then either (a) $\sigma, 0 \models f$ and $\sigma, j \not\models e_{T_f}$ for all $1 \leq j \leq i$ and events $e_{T_f} \in T_f$, or (b) $\sigma, j \models e_{I_f}$, for some j such that $1 \leq j \leq i$ and some $e_{I_f} \in I_f$, and for all l, with $j < l \leq i$, $\sigma, l \not\models e_{T_f}$ for all events $e_{T_f} \in T_f$. In case (a) we know that $initially(f, s) \in I$ and for all $0 \leq j \leq i$, $\sigma, j \not\models e_{T_f}$. Then, by (IH), $happens(e_{T_f}, j-1, m, s) \notin I$, for all $0 \leq j \leq i$ where $j-1$ is a position in time unit m. Therefore $clipped(0, f, i, s) \notin I$. The transformed program Π^I will therefore include $holdsAt(f, i, s) \leftarrow initially(f, s)$ and hence $holdsAt(f, i, s) \in I$. We consider case (b) and assume that an f-Initiating event has occurred at some j, with $1 \leq j \leq i$ (i.e. $\sigma, j \models e_{I_f}$). Then, by (IH), $happens(e_{I_f}, j-1, m, s) \in I$ where $j-1$ is in time unit m. Moreover, since no f-Terminating event occurs at any l where $j < l \leq i$, Π^I does not contain any atom of the form $attempt(e_{T_f}, l-1, n, s)$ where $l-1$ is a position in time unit n and $n \leq h$. Therefore, $happens(e_{T_f}, l-1, n, s) \notin I$. This means that the ground atom $clipped(j-1, f, i, s) \notin I$. The program Π^I then contains the ground rule $holdsAt(f, i, s) \leftarrow initiates(e_{I_f}, f, j-1, s), happens(e_{I_f}, j\text{-}1, m, s), j-1 < i$. Hence, $holdsAt(f, i, s) \in I$.

We consider now the "only if" case and assume that $holdsAt(f, i, s) \in I$. We want to show that $\sigma, i \models f$. Reasoning by contradiction, we assume that $\sigma, i \not\models f$. Then, either (a) f is initially false and has not been initiated at any position j, with $0 \leq j \leq i$, or (b) for every position j, $0 \leq j \leq i-1$, where f is true there is a later position l, with $j < l \leq i$ such that $\sigma, l \models e_{T_f}$. We first consider the case (a). Since $\sigma, 0 \not\models f$, $initially(f, s) \notin I$. Furthermore, because $\sigma, k \not\models e_{I_f}$ for any event in I_f, $happens(e_{I_f}, j-1, m, s) \notin I$ where $j-1$ is associated with time unit m. Thus, under the stable model semantics $holdsAt(f, i, s) \notin I$ which contradicts our initial assumption. Consider now case (b) and let k be the last position in σ such that $\sigma, k \models f$ where $0 < k < i-1$ and for which there is a later position l where $j < l \leq i$ where $\sigma, l \models e_{T_f}$ for some event $e_{T_f} \in T_f$. Then, by (IH), $happens(e_{T_f}, l-1, n, s) \in I$, where position $l-1$ is associated with time unit n, and $clipped(j-1, f, i, s) \in I$. Thus, under the stable model semantics, the following ground rules are not included in Π^I:

$$holdsAt(f, i, s) \leftarrow initially(f, s), not\ clipped(0, f, i, s)$$
$$holdsAt(f, i, s) \leftarrow initiates(e_{I_f}, f, j-1, s), happens(e_{I_f}, j-1, m, s), not\ clipped(j-1, f, i, s)$$

Hence, $holdsAt(f, i, s) \notin I$ which contradicts our initial assumption. □

The above theorem can be generalised to a set Σ of finite paths, as shown in the following corollary.

Corollary 1. Let G_a be a goal model in asynchronous FLTL language, T an LTS model that satisfies G_a. Let Σ be a set of scenarios accepted by T with associated paths $\{\sigma_h | 1 \leq h \leq m\}$. Let Nar be the encoding in EC of Σ and let Π be the EC logic program $\Pi = \tau(G_a, D) \cup Nar$. This program has a unique stable model I. Then for each σ_h and for any event-based fluent $e \in P_e$ and position $p \geq 0$, we have $\sigma_h, j \models_D e$ iff $happens(e, j-1, t_j, s_h) \in I$; for any fluent $f \in P_f$, we have $\sigma_h, j \models_D f$ iff $holdsAt(f, j, s_h) \in I$.

Learning Pre-conditions. As mentioned before, the translation of the positive and negative scenarios also contributes to the set E of examples that the learning algorithm uses to compute pre-conditions. This part of the translation, as shown below, depends on the event for which the pre-condition has to be learned.

Without loss of generality we assume that pre-conditions are to be learned always for the last event of each negative scenario.

Definition 10 (Encoding of scenarios into Examples). Given a set $\Sigma^+ \cup \Sigma^-$ of scenarios, the corresponding set of *examples* is the program E given by:

- for each negative scenario $\sigma_h^- = \langle e_1, ..., e_n \rangle^-$ in Σ^-
  - $n-1$ facts of the form `happens(e`$_i$`,i-1,sneg_h)` with $1 \leq i \leq n-1$
  - a fact of the form `not happens(e`$_n$`,n-1, sneg_h)`
- for each positive scenario $\sigma_h^+ = \langle e_1, ..., e_m \rangle^+$ in Σ^+
  - m facts of the form `happens(e`$_i$`,i-1,spos_h)` with $1 \leq i \leq m$

For instance, a translation of the positive scenario:

$\sigma_1^+ = \langle$`tick, tock, signalHighWater, tick, tock, switchPumpOn, tick`$\rangle^+$

contributes the following facts to E.

```
happens(tick,0,1,spos_1).
happens(tock,1,1,spos_1).
happens(signalHighWater,2,1, spos_1).
happens(tick,3,2,spos_1).
happens(tock,4,2,spos_1).
happens(switchPumpOn,5,2,spos_1).
happens(tick,6,3,spos_1).
```

while a translation of the negative scenario

$\sigma_1^- = \langle$`tick, tock, signalCriticalMethane, signalHighWater, tick, tock, switchPumpOn`$\rangle^-$

gives the following facts to E.

```
happens(tick,0,1,sneg_1).
happens(tock,1,1,sneg_1).
happens(signalCriticalMethane,2,1, sneg_1).
happens(signalHighWater,3,1, sneg_1).
happens(tick,4,2, sneg_1).
happens(tock,5,2, sneg_1).
not happens(switchPumpOn,6,2, sneg_1).
```

The search space, also referred to as Hypothesis Space (*HS*), of all possible pre-conditions is constrained by a language bias, which specifies the predicates that can appear in the hypothesis, H. In our learning task, the language bias defines the predicate `impossible` to appear in the head of the H rule, and the predicates `holdsAtTick`, `notholdsAtTick` and `posInTime` to appear in the body.

As shown in the definition below, an inductive solution is a hypothesis, of the form given by the language bias, that is consistent with the given background theory B and together with B entails a given set of examples E.

Definition 11. Let G_a be a goal model expressed in asynchronous FLTL and T be an LTS model of G_a. Let $\Sigma^+ \cup \Sigma^-$ be a set of positive and negative scenarios accepted by T. Let Nar be the encoding in EC of $\Sigma^+ \cup \Sigma^-$, B be the normal logic program $\tau(G_a, D) \cup Nar$ and HS be the hypothesis space containing the set of clauses with the predicate `impossible` in the head, and any of the predicates `holdsAtTick`, `notholdsAtTick` and `posInTime` in the body. An *inductive solution* $H \subseteq HS$ of the set E, of examples corresponding to $\Sigma^+ \cup \Sigma^-$,

with respect to the background knowledge B, is a set of EC pre-conditions, for every event e_n appearing as the last event in $\sigma_h^- \in \Sigma^-$, of the form

```
impossible(e_n,X,Y,Z) :- position(X), time(Y), scenario(Z),
                         holdsAtTick(f_1,Y,Z),
                         holdsAtTick(f_r,Y,Z),
                         posInTime(X,Y,Z).
```

such that $B, H \models E$ and $B \cup H \not\models$ false.

Consider again our running example of the Mine Pump system, where the the goal model G_a is given by (8)–(12) with associated fluent definitions, and the positive and negative scenarios are as described in Section 4.3. Our learning system, XHAIL, would generate the ground rule:

```
impossible(switchPumpOn,6,2,sneg_1) :- holdsAtTick(highWater,2,sneg_1),
                                       notholdsAtTick(pumpOn,2,sneg_1),
                                       holdsAtTick(criticalMethane,2,sneg_1),
                                       posInTime(6,2,sneg_1).
```

and generalises it into the hypothesis:

```
impossible(switchPumpOn,X,Y,Z) :- holdsAtTick(criticalMethane,Y,Z).
```

The generalisation process is based on a compression heuristic which favours hypotheses containing the fewest number of literals and is motivated by the principle of Occam's razor (which, roughly speaking, means choose the simplest hypothesis that fits the data). For a detailed description of the XHAIL learning algorithm the reader is refer to [RBR04]. This output is then translated back into the FLTL assertion

$\Box(tick \rightarrow (CriticalMethane \rightarrow \bigcirc(\neg switchPumpOn \ \mathsf{W} \ tick)))$

4.5. Selection Phase

The outcome of the learning phase consists of a set of required pre-conditions. Any of these solutions is formally correct, meaning that it removes the violation detected in the analysis phase, covering the positive but not the negative scenarios detected during the scenario elaboration phase. However, the choice of pre-condition to include in the extended goal model does impact on the overall elaboration process. For instance, a "too strong" pre-conditions (i.e.. a pre-condition that restricts the occurrence of events more than necessary) may constrain the new goal model too much and impair the learning process in subsequent iterations. On the other hand, "too weak" pre-conditions may just marginally constraint the specification and lead to a larger number of iteration steps before termination.

The role of the engineer during this phase is therefore crucial. The engineer is prompted with the alternative collections of operational requirements and required to select among these the one that is considered to be more plausible. The selected set of pre-conditions is then added to the current goal model. If the engineer selects a weak pre-condition, more iterations will be required to obtain a non-Zeno model. On the other hand, if an excessively strong requirement is selected then in some future iteration, the engineer will find that the positive example he or she desires to propose as part of the scenario-elaboration phase is inconsistent with the erroneously introduced strong requirements. Having identified this situation, the engineer will have to "backtrack" to this iteration and select a weaker pre-condition.

4.6. The Cycle

At the end of each iteration, the learned pre-conditions are translated back into asynchronous FLTL and added to the goal model. The LTS resulting from the extended goal model is guaranteed not to exhibit the

Zeno trace detected by the LTSA in that iteration and captured by the elaborated negative scenario. In addition, the LTS is guaranteed to accept the positive scenarios identified by the engineer and, of course, all previously elicited goals and operational requirements. The soundness of the learning step is formally captured by the following theorem and constitutes the main invariant of the approach.

Theorem 2. Let G_a be an asynchronous goal model, D a set of fluent definitions, (T, V_D) an FLTL model of G_a and $\Sigma^+ \cup \Sigma^-$ a set of positive and negative scenarios. Let $B = \tau(G_a, D) \cup Nar$ where Nar and is the encoding of scenarios $\Sigma^+ \cup \Sigma^-$ into a narrative. Let E be the set of examples corresponding to the scenarios $\Sigma^+ \cup \Sigma^-$, and HS be the hypothesis space containing the set of clauses with predicate `impossible` in the head, and any of the predicates `holdsAtTick`, `notholdsAtTick` and `posInTime` in the body. Let $H \subset HS$ be an inductive solution for E with respect to B, such that $B \cup H \models E$ and $B \cup H \not\models$ false. Then the corresponding set Pre of asynchronous FLTL pre-conditions, such that $\tau(Pre) = H$, is a correct extension of G_a with respect to $\Sigma^+ \cup \Sigma^-$.

Proof:
The proof uses the following notation. For every position i in a given scenario, t_i will be used to denote the number of *tick* transitions from position 1 until position i inclusive. Let $\sigma_1^+ = \langle e_1, ..., e_m \rangle^+$ and $\sigma_1^- = \langle e_1, ...e_n \rangle^-$ be a positive and negative scenario respectively accepted in T, where $e_m = e_n$ and such that $\sigma_1^+ \in \Sigma^+$ and $\sigma_1^- \in \Sigma^-$. Consider now the program $B \cup E$. From the translation of the negative scenario, we know that E contains the ground facts `happens(`e_i`,i-1,`t_i`,sneg_1)` for all $1 \leq i \leq n-1$, and the literal `not happens(`e_n`,n-1,`t_n`,sneg_1)`, and from the translation of the positive scenario the literals `happens(`e_j`,j-1,`t_j`,spos_1)`, for all $1 \leq j \leq m$. We also know that B contains the ground literals `attempt(`e_i`,i-1,`t_i`,sneg_1)` for all $1 \leq i \leq n$, and the ground literals `attempts(`e_j`,j-1,`t_j`,spos_1)` for all $1 \leq j \leq m$. Hence, given the stable model I of B, `happens(`e_i`,i-1,`t_i`,sneg_1)`$\in I$, for $1 \leq i \leq n$, `happens(`e_j`,j-1,`t_j`,spos_1)`$\in I$, for $1 \leq j \leq m$. Therefore, $B \not\models$ `not happens(`e_n`,n-1,`t_n`,sneg_1)` and $B \not\models E$.

H is defined as a set of EC pre-condition clauses. Hence given the mode declaration of H, then an inductive solution of E with respect to B is the single ground clause

```
impossible(en,n-1,tn,sneg_1):- (not)holdsAtTick(f1,tn,sneg_1), ...,
                               (not)holdsAtTick(fr,tn,sneg_1).
```

such that `holdsAtTick(`f_i`,`t_n`,sneg_1)`, $0 \leq i \leq r$, appears in the body if the literals `holdsAt(`f_i`,k,sneg_1)`, `attempt(tick, k,`t_n`,sneg_1)` and `posInTime(k,`t_n`,sneg_1)` for some $1 \leq k \leq n$, are true in I. Similarly, the ground literal `(not)holdsAtTick(`f_i`,`t_n`,sneg_1)` appears in the body of the rule above, if the literal `holdsAt(`f_i`,k,sneg_1)` is not true in I and both the literals `attempt(tick,k,`t_n`,sneg_1)` and `posInTime(k,`t_n`,sneg_1)` are true in I.

The above rule is then generalised giving a (minimally compressed) hypothesis H that subsumes the rule

```
impossible(en,X,Y,Z) :- (not)holdsAtTick(f1,Y,Z),..., (not)holdsAtTick(fl,Y,Z).
```

obtained by replacing the input ground terms with distinct variables.

Now, given that Pre is the FLTL pre-condition such that $\tau(Pre) = H$, we need to show that the LTS model of $G_a \cup Pre$ accepts σ_1^+ but does not accept σ_1^- . The proof is by contradiction.

- First suppose that the LTS model of $G_a \cup Pre$ accepts σ_1^-. This means that there is a path in T of the form $s_0 \mathcal{R}_{e_1} s_1 ... s_{n-1} \mathcal{R}_{e_n} s_n$. Consider the program $B \cup H = \tau(G_a \cup Pre)$. From Theorem 1 we have $B \cup H \models$ `happens(`e_i`,i-1,`t_i`,sneg_1)` including the last event e_n of the negative scenario σ_1^-. But we know that $B \cup H \models$ `not happens(`e_n`,n-1,`t_n`,sneg_1)` which is a contradiction. Hence $G_a \cup Pre$ does not accept σ_1^-.
- Now we suppose that $G_a \cup Pre$ does not accept the positive scenario σ_1^+. So $G_a \cup Pre$ must include a pre-condition for an event e_h, with $1 \leq h \leq m$, in σ_1^+. Assume $G_a \cup Pre$ includes a pre-condition for an event e_h where $1 \leq h \leq m-1$. Since the positive scenario σ_1^+ is assumed to be accepted in G_a, then this

means that the pre-condition for event e_h must be included in Pre, as otherwise this would contradict the fact that the path $\sigma = s_0\mathcal{R}_{e_1}s_1...s_{m-1}\mathcal{R}_{e_m}s_m$ already exists in the LTS model of G_a . Assuming now that Pre includes a pre-condition for e_h where the antecedent is satisfied at state s_{h-1} of a path accepting σ_1^+ and there is no transition e_h to state s_h. Now, the program $B \cup H$ is equal $\tau(G_a \cup Pre) \cup Nar$ where Nar is the encoding of the scenarios $\Sigma^+ \cup \Sigma^-$. So by Corollary 1 we have that $B \cup H \models$ `happens(e_i,i-1,t_i, spos_1)` where $1 \leq i \leq h-1$, $B \cup H \models$ `happens(e_i,i-1,t_i,spos_1)` where $h+1 \leq i \leq m$ and $B \cup H \models$ `not happens(e_h,h-1,t_h,spos_1)`, which is in contradiction with the fact that $B \cup H \models E$.
The remaining case to consider is that $G_a \cup Pre$ includes a pre-condition for e_m. Let's assume that G_a includes such a pre-condition. The path $s_0\mathcal{R}_{e_1}s_1...s_{m-1}\mathcal{R}_{e_m}s_m$ would then not exist already in the LTS model of G_a, which is inconsistent with our initial assumption. Let's assume then that Pre includes a pre-condition on e_m. Consider the sub-path $s_0\mathcal{R}_{e_1}s_1, ..., s_{m-2}\mathcal{R}_{e_{m-1}}s_{m-1}$ in which e_m does not occur. Now, the program $B \cup H$ is equal to $\tau(G_a \cup Pre, D) \cup Nar$. So by Corollary 1 we have that $B \cup H \models$ `happens(e_j,j-1,t_j,spos_1)` where $1 \leq j \leq m-1$ and $B \cup H \models$ `not happens(e_m,m-1,t_m,spos_1)`, which is in contradiction with the fact that $\Pi \cup H \models E$.

The four phases described in this section are then repeated with respect to the extended G_a. Assuming the initial set of violation traces in the LTS model generated from given G_a is *finite* and that no further information (other than the learned event pre-conditions) is added to G_a, then this cycle is repeated until all the pre-conditions necessary to guarantee, together with the initial goal model, the construction of a non-Zeno LTS model are computed. The termination of this cycle is based upon the invariant property that *from one iteration to another, the number of violation traces for the Zeno property (TP) progressively reduces.* This means that assuming that two iterations are needed to satisfy the time progress property TP, the set of traces in the LTS model of G_{a_i}, that violate TP *strictly includes* the set of traces in the LTS model of $G_{a_{i+1}}$ that violate TP. This is because the addition of a learned pre-condition to the current goal model is guaranteed (as proved by the above theorem) to reduce the number of violation traces in the extended goal model. The application of our approach to two case studies, the Mine Pump System [KMS83] and Injection System [CP93], have so far successfully confirmed the termination of the cycle and its convergence to the computation of a correct extension of G_a that accepts non-Zeno LTS models.

5. Case Study

This section presents the approach applied on a simplified version of the Safety Injection system originally described in [CP93, Let02]. This is a system for Nuclear Power Plant that prevents or mitigates damage to the core and coolant system on the occurrence of a fault such as a loss of coolant. The system monitors the water pressure level: If it drops below a predetermined set point "Low", the system sends a safety injection signal to the safety feature components which are responsible for dealing with the incident. A manual block (push-button) is provided in order to override the safety injection signal and to avoid actuation of the protection system during a normal start-up or cool down phase. A manual block is permitted if and only if the steam pressure is below a specified value (permissive). The manual block must be automatically reset by the system. A manual block is effective if and only if it is executed before the protection signal is present.

Assume the language consists of the state-based fluents {*SafetyInjection, Overridden, PressureBelowLow, PressureAbovePermit, Occurs_Block, Occurs_Reset*}, the event-based fluents {*sendSafetySignal, stopSafetySignal, overrideSafetySignal, enableSafetySignal, lowerPressureBelowLow, raisePressureAboveLow, raisePressureAbovePermit, lowerPressureBelowPermit, block. reset*} as well as the following fluent definitions D:

```
fluent SafetyInjection = <{sendSafetySignal},{stopSafetySignal}>
fluent Overridden = <{overrideSafetySignal},{enableSafetySignal}> initially True
fluent PressureBelowLow = <{lowerPressureBelowLow},{raisePressureAboveLow}>
fluent PressureAbovePermit = <{raisePressureAbovePermit}, {lowerPressureBelowPermit}>
fluent Occurs_Block = <{block},{tock}>
fluent Occurs_Reset = <{reset},{tock}>
```

Given the language above, the asynchronous goal model, G_{a_1}, generated from the synchronous goal model

in [CP93], is composed of the following FLTL formulae:

$$g_a[SafetyInjectionWhenLowPressureAndNotOverridden] = \Box(tick \rightarrow ((PressureBelowLow \wedge \neg Overridden) \rightarrow \bigcirc(\neg tick \,\mathsf{W}\, (tick \wedge SafetyInjection)))) \quad (19)$$

$$g_a[SafetyInjectionOverriddenWhenBlockOccursAndPressureBelowPermit] = \Box(tick \rightarrow ((Occurs\_Block \wedge \neg PressureAbovePermit) \rightarrow \bigcirc(\neg tick \,\mathsf{W}\, (tick \wedge Overridden)))) \quad (20)$$

$$g_a[SafetyInjectionNotOverriddenWhenPressureBelowPermit] = \Box(tick \rightarrow (PressureAbovePermit \rightarrow \bigcirc(\neg tick \,\mathsf{W}\, (tick \wedge \neg Overridden)))) \quad (21)$$

$$DomPre(sendSafetySignal) = \Box(tick \rightarrow (PumpOn \rightarrow \bigcirc(\neg sendSafetySignal \,\mathsf{W}\, tick))) \quad (22)$$

$$DomPre(stopSafetySignal) = \Box(tick \rightarrow (\neg PumpOn \rightarrow \bigcirc(\neg stopSafetySignal \,\mathsf{W}\, tick))) \quad (23)$$

$$DomPre(overrideSafetySignal) = \Box(tick \rightarrow (Overridden \rightarrow \bigcirc(\neg overrideSafetySignal \,\mathsf{W}\, tick))) \quad (24)$$

$$DomPre(enableSafetySignal) = \Box((tick \rightarrow (\neg Overridden) \rightarrow \bigcirc(\neg enableSafetySignal \,\mathsf{W}\, tick))) \quad (25)$$

where (19)-(21) specify the (asynchronous) goals and (22)-(25) define the domain pre-conditions for the operations. Furthermore, we start our case study assuming the following required pre-condition is known.

$$ReqPre(sendSafetySignal) = \Box(tick \rightarrow ((\neg PressureBelowLow \vee Overridden) \rightarrow \bigcirc(\neg sendSafetySignal \,\mathsf{W}\, tick))) \quad (26)$$

What follows is a summary of some of the iterations resulting from the application of our approach.

Iteration 1

Analysis: Applying the analysis phase to G_{a_1} results in the following violation trace.

```
Violation of LTL property: @TICK
Trace to terminal set of states:
tick                            Overridden
tock                            Overridden
enableSafetySignal
reset                           Occurs_Reset
raisePressureAbovePermit        PressureAbovePermit && Occurs_Reset
tick                            PressureAbovePermit && Occurs_Reset
tock                            PressureAbovePermit
overrideSafetySignal            Overridden && PressureAbovePermit
lowerPressureBelowPermit        Overridden
lowerPressureBelowLow           PressureBelowLow && Overridden
Cycle in terminal set:
block
LTL Property Check in: 47ms
```

This trace exemplifies a possible system behaviour which violates the time progress property. In the above example a tick is not allowed to occur as its occurrence would result in a goal violation, specifically the goal g_a [*SafetyInjectionNotOverriddenWhenPressureBelowPermit*] which requires the safety injection signal to be

enabled by the next tick.

Scenario Elaboration: The *overrideSafetySignal* is identified as the undesirable event. The negative scenario hence becomes:

$\sigma_1^- = \langle$`tick, tock, enableSafetySignal, reset, raisePressureAbovePermit, tick, tock, overrideSafetySignal`$\rangle$

A possible positive scenario is:

$\sigma_1^+ = \langle$`tick, tock, reset, enableSafetySignal, raisePressureAbovePermit, tick, tock, lowerPressureBelowPermit, tick, tock, block, tick, tock, overrideSafetySignal`$\rangle$

Learning: The learning phase produces the following three alternative pre-conditions for the system event *overrideSafetySignal*:

$\texttt{Pre}_1\texttt{(overrideSafetySignal)} =$

$$\Box(tick \rightarrow (PressureAbovePermit \rightarrow \bigcirc\neg overrideSafetySignal \,\mathsf{W}\, tick)) \tag{27}$$

$\texttt{Pre}_2\texttt{(overrideSafetySignal)} =$

$$\Box(tick \rightarrow (\neg Occurs\_Block \rightarrow \bigcirc\neg overrideSafetySignal \,\mathsf{W}\, tick)) \tag{28}$$

$\texttt{Pre}_3\texttt{(overrideSafetySignal)} =$

$$\Box(tick \rightarrow (Occurs\_Reset \rightarrow \bigcirc\neg overrideSafetySignal \,\mathsf{W}\, tick)) \tag{29}$$

Selection: The required pre-condition (27) is chosen an added to G_{a_1}

Iteration 2

The second iteration starts from the extended goal model $G_{a_2} = G_{a_1} \cup$ (27).

Analysis: The following violation trace is identified in the second iteration.

```
Violation of LTL property: @TICK
Trace to terminal set of states:
tick                            Overridden
tock                            Overridden
reset                           Occurs_Reset && Overridden
enableSafetyInjection           Occurs_Reset
lowerPressureBelowLow           PressureBelowLow && Occurs_Reset
tick                            PressureBelowLow&& Occurs_Reset
tock                            PressureBelowLow
sendSafetySignal                SafetyInection && PressureBelowLow
tick                            SafetyInjection && PressureBelowLow
tock                            SafetyInjection && PressureBelowLow
stopSafetySignal                PressureBelowLow
raisePressureAboveLow
raisePressureAbovePermit        PressureAbovePermit
Cycle in terminal set:
lowerPressureBelowPermit
raisePressureAbovePermit
LTL Property Check in: 16ms
```

The above exemplifies another violation of the non-Zeno property caused this time by the occurrence of the event *stopSafetySignal*.

Scenario Elaboration: *stopSafetySignal* is indicated as the undesirable one. The negative scenario here

becomes:

$\sigma_1^- = \langle$`tick, tock, enableSafetyInjection, lowerPressureBelowLow, tick, tock, sendSafetySignal, reset, tick, tock, stopSafetySignal`$\rangle$

Two possible positive scenarios are:

$\sigma_1^+ = \langle$`tick, tock, enableSafetyInjection, lowerPressureBelowLow, tick, tock, sendSafetySignal, raisePressureAboveLow, tick, tock, stopSafetyInjection`$\rangle$

$\sigma_2^+ = \langle$`tick, tock, enableSafetyInjection, lowerPressureBelowLow, tick, tock, sendSafetySignal, block, tick, tock, overrideSafetySignal, tick, tock, stopSafetyInjection`$\rangle$

Learning: The learning phase in this case results in the single pre-condition formally expressed as:

$\texttt{Pre}_1\texttt{(stopSafetySignal)} =$

$$\Box(\mathit{tick} \rightarrow ((\neg \mathit{Overridden} \wedge \mathit{PressureBelowLow}) \rightarrow \bigcirc \neg \mathit{stopSafetySignal}\, \mathsf{W}\, \mathit{tick})) \quad (30)$$

Selection: The pre-condition (30) is then added to the goal model G_{a_2}.

Iteration 3

Running the analysis phase again on $G_{a_3} = G_{a_2} \cup$ (30) gives the following output:

```
-- States: 420 Transitions: 1370 Memory used: 3247K
No LTL Property violations detected.
LTL Property Check in: 0ms
```

6. Related Work

Automated reasoning techniques are increasingly being used in requirements engineering [LW98, DBLvL05, DLvL06]. Among these, the work most related to our approach is [LW98], where an *ad-hoc* inductive inference process is used to derive high-level goals, expressed as temporal formulae, from manually attuned scenarios provided by stake-holders. Each scenario is used to infer a set of goal assertions that explains it. Then each goal is added to the initial goal model, which is then analyzed using state-based analysis techniques (i.e. goal decomposition, conflict management and obstacle detection). The inductive inference procedure used in [LW98] is mainly based on pure generalization of the given scenarios and does not take into account the given (partial) goal model. It is therefore a potentially unsound inference process by the fact that the generated goals may well be inconsistent with the given (partial) goal model. In our approach learned requirements are guaranteed to be consistent with the given goals.

The work in [DBLvL05, DLvL06] also proposes the use of inductive inference to generate behavior models. It provides an automated technique for constructing LTSs from a set of user-defined scenarios. The synthesis procedure uses a grammar induction to generate an LTS that satisfies all given positive scenarios and none of the negative ones. Starting from this initial LTS model, the inference procedure attempts "generalizing" this model by merging states of the LTS and still preserving the initial set of scenarios. After each merge, the user is requested to categorize specific paths of the new LTS as positive or negative. To reduce number of scenarios to be classified, a goal specification is assumed and only paths satisfying the goals are queried to the user. The *generalisation* process is based on a *bottom-up* search. It starts with the most constrained LTS (i.e. the LTS that only contains paths that cover the initial set of scenarios) and progressively generalises it by merging states and therefore including more behaviors. This generalization process, however, depends on the order in which the states are considered for merging. On the other hand, the approach proposed in this paper considers a *top-down* search. It starts from the least constrained LTS model that satisfies the given specification, but that can exhibits Zeno traces behaviors, and it *refines* it adding pre-conditions on system events so to eliminate Zeno undesirable behaviors.

The technique in [LKMU06] describes the steps for transforming a given KAOS goal and operational

model into an FLTL theory that is used later by the LTSA to construct an LTS. Deadlock analysis reveals inconsistency problems in the KAOS model. However, the technique assumes these are resolved by manually reconstructing the operational model. Our approach builds on the goal to LTS transformation of [LKMU06] but does not require a fully operationalised model. Rather it provides automated support for completing an operational model with respect to the given goals, that does satisfy the non-Zeno property.

7. Conclusion and Future Work

The paper presents an approach for deriving non-Zeno behavior model from goal models. It deploys established model checking and learning techniques for the computation of required pre-condition from scenarios. These pre-conditions can incrementally be added to the initial goal model so to generate at the end of the cycle a non-Zeno behavior model. The pre-condition learned at each iteration has the effect of removing Zeno traces identified by the LTSA-based analysis of the goal model at the beginning of that iteration. The cycle terminates when no more Zeno traces are generated from the LTSA on the current (extended) goal model. A formal characterization of termination of the cycle is currently under investigation. But our experiments and case study results have so far confirmed the convergence of our process. Furthermore, the approach assumes, in the second phase, that the engineer will manually elaborate the violation trace into a set of scenarios. The possibility of automating the process of scenario elaboration process by using other forms of reasoning techniques (e.g. abduction) is being considered. Future work includes learning other types of operational requirements such as trigger-conditions, learning pre-conditions with time-bounded operators such as $\Diamond_{\leq h}$ meaning some time in the future within time h and $\Box_{\leq h}$ meaning always in the future within time h in [LL02], as well as past operators (e.g. *Back-to*, B, and *Since*, S, operators) in [MP92], and to integrate the approach within a framework for generating a set of required pre- and trigger-conditions that is complete with respect to a given goal model.

Acknowledgements

We acknowledge EPSRC EP/CS541133/1, ANPCyT PICT 11738, the Levehulme Trust and King Saud University for partially funding this work.

References

[Ant97] A.I Anton. *Goal identification and refinement in the specification of software-based information systems*. PhD thesis, Atlanta, GA, USA, 1997.

[ARU08] D. Alrajeh, A. Russo, and S. Uchitel. Deriving non-zeno behavior models from goal models using ilp. In *Proc. ETAPS/FASE08 Conference on Foundation Aspects of Software Engineering*, 2008.

[CP93] P.J. Courtois and D. L. Parnas. Documentation for safety critical software. In *Proc. of 15th ICSE Conference*, pages 315–323, 1993.

[DBLvL05] C. Damas, P. Dupont B. Lambeau, and A. van Lamsweerde. Generating annotated behavior models from end-user scenarios. *IEEE Transactions on Software Engineering*, 31(12):1056–1073, 2005.

[DLvL06] C. Damas, B. Lambeau, and A. van Lamsweerde. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *Proc. of the Intl. ACM Symp. on the Foundations of Software Engineering*, 2006.

[DvL96] R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *Proc. of the 4th ACM Symp. on the Foundations of Software Engineering*, 1996.

[DvLF93] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1):3–50, 1993.

[GM03] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *Proc. 11th ACM SIGSOFT Symp. on Foundations Software Engineering*, 2003.

[GMS05] P. Giorgini, J. Mylopoulos, and R. Sebastiani. Goal-oriented requirements analysis and reasoning in the tropos methodology. *Engineering Applications of Artificial Intelligence*, 18:159–171, 2005.

[HBGL95] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. Scr*: A toolset for specifying and analyzing requirements. In *Proc. of the 10th Annual Conf. on Computer Assurance*, 1995.

[KMS83] J. Kramer, J. Magee, and M. Sloman. Conic: An integrated approach to distributed computer control systems. In *IEE Proc., Part E 130*, 1983.

[Let02] E. Letier. Goal-oriented elaboration of requirements for a safety injection control system. Technical report, Dèpartement d'Ingènierie Informatique, UCL, 2002.

[LKMU05] E. Letier, J. Kramer, J. Magee, and S. Uchitel. Fluent temporal logic for discrete-time in event-based models. In *Proc. of the 10th European Software Engineering Conf.*, 2005.

[LKMU06] E. Letier, J. Kramer, J. Magee, and S. Uchitel. Deriving event-based transitions systems from goal-oriented requirements models. Technical Report 02/2006, Imperial College London, 2006.

[LL02] E. Letier and A. Van Lamsweerde. Deriving operational software specifications from system goals. In *Proc. 10th ACM SIGSOFT Symp. on Foundations of Software Engineering*, 2002.

[LvL02] E. Letier and A. van Lamsweerde. Agent-based tactics for goal-oriented requirements elaboration. In *Proc. of the 24th Intl. Conf. on Software Engineering*, 2002.

[LW98] A. Van Lamsweerde and L. Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE Transactions on Software Engineering*, 24(12):1089–1114, 1998.

[MK99] J. Magee and J. Kramer. *Concurrency : State Models and Java Programs*. John Wiley and Sons, 1999.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 1992.

[MS02] R. Miller and M. Shanahan. Some alternative formulation of event calculus. *Computer Science; Computational Logic; Logic programming and Beyond*, 2408, 2002.

[Mug95] S.H. Muggleton. Inverse Entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

[Ray09] O. Ray. Nonmonotonic abductive inductive learning. *Journal of Applied Logic*, 7(3):329–340, 2009.

[RBR04] O. Ray, K. Broda, and A. Russo. A hybrid abductive inductive proof procedure. *Logic Journal of the IGPL*, 12(5):371–397, 2004.

[Sha97] M.P. Shanahan. *Solving the Frame Problem*. MIT Press, 1997.

[SMMM98] 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.

[UBC07] S. Uchitel, G. Brunet, and M. Chechik. Behaviour model synthesis from properties and scenarios. In *Proc. of the 29th IEEE/ACM Intl. Conf. on Software Engineering*, 2007.