Deriving Non-zeno Behavior Models from Goal Models Using ILP*

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

[1] Imperial College London
{da04,ar3,su2}@doc.ic.ac.uk
[2] University of Buenos Aires
s.uchitel@dc.uba.ar

Abstract. This paper addresses the problem of automated derivation of non-zeno behaviour models from goal models. The approach uses a novel combination of model checking and machine learning. We first translate a goal model formalised in linear temporal logic into a (potentially zeno) labelled transition system. We then iteratively identify zeno traces in the model and learn operational requirements in the form of *preconditions* that prevent the traces from occurring. Identification of zeno traces is acheived by model checking the behaviour model against a time progress property expressed in linear temporal logic, while learning operational requirements is achieved using Inductive Logic Programming. As a result of the iterative process, not only a non-zeno behaviour model is produced but also a set of preconditions that, in conjunction with the known goals, ensure the non-zeno behaviour of the system.

1 Introduction

Goal oriented requirements engineering (GORE) is an increasingly popular approach to elaborating software requirements. Goals are prescriptive statements of intent whose satisfaction requires the cooperation of components in the software and its environment. One of the limitations of approaches to GORE [5,6,14] is that the declarative nature of goals hinders the application of a number of successful validation techniques based on executable models such as graphical animations, simulations, and rapid-prototyping. They do not naturally support narrative style elicitation techniques, such as those in scenario-based requirements engineering and are not suitable for down-stream analyses that focus on design and implementation issues which are of an operational nature.

To address these limitations, techniques have been developed for constructing behaviour models automatically from declarative descriptions in general [22] and goal models specifically [12]. The core of these techniques is based on temporal logic to automata transformations developed in the model checking community. For instance, in [12] Labelled Transition Systems (LTS) are built automatically from KAOS goals expressed in Fluent Linear Temporal Logic (FLTL) [8].

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

The key technical difficulty in constructing behaviour model from goal models is that the latter 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 behaviour models from a goal model may introduce deadlocks and progress violations. More specifically, the resulting behaviour model may be *zeno*, i.e exhibit traces in which time never progresses. Clearly these models do not adequately describe the intended system behaviour and thus are not suitable basis for analysis.

A solution proposed in [12] to the problem of zeno traces is to construct behavior models from a fully *operationalised* goal model rather than from a set of high-level goals. This involves identifying system operations and extracting operational requirements in the form of *pre-* and *trigger-conditions* from the high-level goals [13]. One disadvantage of this approach is that operationalisation is a manual process for which only partial support is provided. Support comes in the form of derivation patterns restricted to some common goal patterns [6].

This paper addresses the problem of non-zeno behaviour model construction using a novel 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. Briefly, the proposed method first involves translating automatically the goal model, formalised in Linear Temporal Logic (LTL), into a (potentially zeno) labelled transition system. Then, in an iterative process, zeno traces in the behaviour model are identified mechanically, elaborated into positive and negative scenarios, and used to automatically learn preconditions that prevent the traces from occurring. Identification of zeno traces is achieved by model checking the behaviour model against a time progress property expressed in LTL, while preconditions are learned using Inductive Logic Programming (ILP).

As a result of the proposed approach, not only a non-zeno behaviour model is constructed, but also a set of precondition is produced. These preconditions, in conjunction with the known goals, ensure the non-zeno behaviour of the system. Consequently, the approach also supports the operationalisation process of goal models described in [13].

The rest of the paper is organizes as follows: Section 2 provides background on goal-models, LTSs and FLTL. Section 3 describes the problem of derivation of non-zeno behaviour models and presents a formalisation of the problem resolved in this paper. Section 4 presents the details of the proposed approach. Section 5 discusses the results and observations obtained from applying the approach. Finally, Sections 6 and 7 conclude the paper by a comparison with related work and discussion on future work.

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 [10]. In this system, a pump controller is used to prevent the water in a mine sump 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. Goals are expected to be refined into sub-goals that can be assigned either to the software-to-be, to other components or to the environment as domain assumptions. Goals assigned to the software-to-be are used to derive operational requirements in the form of pre-, post- and trigger-conditions for operations provided by the software.

In this paper, we define a goal model to be a collection of system goals and operation descriptions. We specify goals informally using natural language and formally using LTL [16]. We follow the KAOS [5] approach assuming 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 smallest possible time unit for the application domain. Hence, the 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 $\Box(\neg HighWater \rightarrow \bigcirc\neg PumpOn)$, 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 sump is above the low level threshold" and "the pump is on", respectively.

LTL assertions are constructed using a set P of propositions that 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, ...). Each state defines the set of propositions true at that state. In addition, each state is classified as an *observable* or a *non-observable* state.

LTL assertions are evaluated only on observable states. In the case of goal models, an observable state corresponds to a time point where a time unit ends and another one starts. 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} observable state in that trace. Note this state is not necessarily the i^{th} state of the trace, as non-observable states may appear between observable ones. The semantics of Boolean operators is defined in a standard way over each observable 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} observable state in σ if p is true at the $(i+1)^{th}$ observable state in σ. An LTL assertion ϕ is said to be *satisfied in a trace* σ if and only if it is satisfied at the first observable 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 also include domain and required conditions of operations. An *operation* causes the system to transit from one state to another. Conditions over operations can be domain *pre-* and *post-conditions* and required *pre-* and *trigger-conditions*. Domain pre- and post-conditions describe elementary state transitions defined by operation applications in the domain. For instance, the operation *switchPumpOn* has as domain pre- and post-condition the assertions $\neg PumpOn$ and *PumpOn*. Required pre- and trigger-conditions are prescriptive conditions defining respectively the weakest necessary conditions and the sufficient conditions for performing an operation. Conditions on operations are evaluated at the beginning/end of time units. For instance, if a required precondition holds at the beginning of a time unit (i.e. at an observable state in a trace), then the operation *may* occur within the time unit (i.e. before the next observable state in that trace). It is expected that the required pre- and trigger-conditions guarantee the satisfaction of system goals.

2.2 Behaviour Models

Behaviour models are event-based representations of system behaviors. Different formalisms have been proposed for modelling and analyzing system behaviors (e.g. [9]), among which LTS is a well known formalism for modelling systems as a set of concurrent components. Each component is defined as a set of states and possible transitions between these states [15]. Transitions are labelled with events denoting the interaction that the component has with its environment. The global system behavior is captured by the parallel composition of the LTS model of each component, by interleaving their behavior but forcing synchronization on shared events.

LTSA [15] 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) [8]. 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 fluents as P_f whereas event fluents as P_e. A *fluent definition* is a pair of disjoint sets of events, referred to as the *initiating* and *terminating* 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 fluent *PumpOn* would be

PumpOn= $\langle\{switchPumpOn\},\{switchPumpOff\}\rangle$, meaning that the event *switchPumpOn* (resp. *switchPumpOff*) causes the fluent *PumpOn* to be true (resp. false). Given an event e, the event fluent also denoted as e is always defined as $\langle\{e\}, L - \{e\}\rangle$ where L is the universe of events.

Asynchronous FLTL assertions are evaluated over traces of states and events (i.e. $s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} s_2 ...$). A position i in a trace σ, for $i \geq 0$, denotes the i^{th} state in σ. The satisfiability of fluents over an LTS is defined with respect to positions in a given traces. A fluent f is said to be true at position i in a trace σ if and only if either of the following conditions hold: (i) 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. As the satisfiability depends on the fluent definition, we use $\sigma, i \models_D f$ to denote that f is satisfied in σ at i with respect to a given set of fluent definitions D.

LTS models are *untimed*. To support the derivation of behavior models from goal models, which are timed models, time must be represented explicitly in the LTSs. We adopt the standard approach to discrete timed behaviour models by introducing an event *tick* to model the global clock with which each timed-process synchronizes [15]. In the context of this paper, a *tick* signals the end of a time unit (as assumed by the goal model) and the beginning of the next.

When modelling discrete-timed systems using LTS with *tick* events, it is important to check the system does not exhibit traces in which a finite number of ticks occur. These traces, called *zeno traces*, represent executions in which time does not progress. We refer to an LTS with zeno traces as a *zeno model*.

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 describe formally the problem that this paper addresses in Section 4.

3.1 From Goal Models to Behaviour Models

Let G_s be a consistent goal model as described in Section 2.1. It is possible to construct an LTS satisfying G_s by transforming G_s into a semantically equivalent set of asynchronous FLTL assertions G_a and then computing an LTS model using an adaptation [12] of the temporal logic to automata algorithms in [8].

The transformation of a goal model G_s into asynchronous FLTL assertions G_a requires (i) translating the LTL goals in G_s to FLTL assertions using the technique described in [12] and (ii) coding the operations descriptions in G_s into asynchronous FLTL.

Goal assertions in G_s are translated into semantically equivalent asynchronous FLTL using an event fluent *tick* to model observable states where LTL formulae are evaluated. The translation $Tr : LTL \rightarrow FLTL_{Async}$ is defined as follows (where ϕ and ψ are LTL assertions):

$$Tr(\Box\phi) = \Box(tick \rightarrow Tr(\phi)) \qquad Tr(\phi \mathsf{U} \psi) = tick \rightarrow Tr(\phi)\mathsf{U}(tick \wedge Tr(\psi))$$
$$Tr(\Diamond\phi) = \Diamond(tick \wedge Tr(\phi)) \qquad Tr(\bigcirc\phi) = \bigcirc(\neg tick \mathsf{W}(tick \wedge Tr(\phi)))$$

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 $Tr(\bigcirc\phi)$ will be evaluated at the start of a time unit, in other words at the occurrence of a *tick*, and requires that no *ticks* shall occur from that point onwards until the asynchronous translation of ϕ holds and *tick* occurs. Consider the synchronous goal *PumpOffWhenLowWater* formalised in Section2.1. Its translation gives the asynchronous assertion $\Box(tick \rightarrow (\neg HighWater \rightarrow \bigcirc\ (\neg\ tick\mathsf{W}(tick \wedge \neg PumpOn))))$.

The operations defined in G_s correspond to events in the behaviour model constructed, and the operation descriptions are expressed as FLTL assertions using the associated event fluents. For instance, if *DomPre* is the domain precondition for an operation e, then its asynchronous FLTL assertion is $\Box((tick \rightarrow (\neg DomPre) \rightarrow \bigcirc\neg e\ W\ tick))$ (intuitively, if *DomPre* is false at the start of a time unit, then e may not occur until the next tick). The coding of required preconditions *ReqPre* in G_a is analogous to domain preconditions. The FLTL assertion coding of required trigger-condition *ReqTrig* for an operation e is $\Box(tick \rightarrow ((ReqTrig\ \wedge\ DomPre) \rightarrow \bigcirc\ \neg tick\mathsf{W}e))$. Finally, the domain postcondition *DomPost* for an operation e in G_s is coded within the fluent definition associated with it. If f is the fluent appearing positively (resp. negatively) in the domain postcondition for e, then it is added to the fluent f's initiating (resp. terminating) set of events.

Computing an LTS model from the asynchronous FLTL representation of a goal model requires using an adaptation [12] of a temporal logic to automata algorithm used in model checking of FLTL [8]. This adaptation consists of applying the technique, described in [8], to each FLTL assertion and then composing in parallel the individual LTS models, which amounts to logical conjunction. We shall see in the next subsection that this is not sufficient and that the resulting LTS may exhibit problematic behaviour in the form of zeno executions.

3.2 The Problem with Zeno Models

The LTS models constructed from asynchronous FLTL goal assertions are not good models of a system behavior, as they constrain the event *tick* which cannot be controlled by the system and do not impose any constraints on the controllable events. The latter are only introduced by conditions on operations (i.e. domain and required conditions). If the goal model has insufficient conditions on operations then spurious executions may be exhibited by the goal model. For instance, the LTS model for the goal *PumpOffWhenLowWater* includes the infinite trace $\langle$`tick, switchPumpOn, switchPumpOff, switchPumpOn, switchPumpOff,...`$\rangle$ which does not exhibit a second tick events, so violating the expectation that time progresses (also referred 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. preconditions for *switchPumpOn* or *switchPumpOff* is missing). In conclusion, 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 operations.

3.3 Problem Formulation

The problem the remainder of this paper addresses is providing automated support to extending a goal model with conditions over its operations in order to guarantee the construction of a non-zeno behaviour model. In other words, given the asynchronous transformation G_a of a synchronous goal model G_s, the aim is to find a set of required preconditions Pre, referred to as the *correct operational extension of* G_a, such that the LTS model constructed from $G_a \cup Pre$ satisfies the *time progress* (TP) property .

4 The Approach

This section presents a novel approach for extending a goal model with the necessary set of required preconditions for deriving a non-zeno behavior model that satisfies a given goal model. The approach uses model checking to provide automated support for generating zeno traces and ILP to produce the preconditions.

4.1 Overview of the Approach

Figure 4.1 depicts an overview of the approach. A goal model G_s is initially transformed into an asynchronous model G_a and a set of fluent definitions D. The actual computation of the correct operational extension of G_a is then iteratively done in three phases: (i) the *analysis phase* in which the LTSA model checking tool is used to construct an LTS model satisfying G_a with respect to D and then checks the LTS against the time progress property. If the property does not hold, a counter example trace is generated, (ii) the *scenario elaboration phase* in which the violation trace is elaborated into a set of positive and negative scenarios and (iii) the *learning phase* which first transforms asynchronous goal model, fluent definitions and scenarios into a logic program and then uses an ILP system to find a set of required preconditions that cover the positive but none of the negative scenarios. Once computed, these preconditions are added to the initial goal model and the steps above repeated. The final output is an extended goal model from which a non-zeno behavior model can be constructed.

4.2 Analysis Phase

This phase considers an asynchronous FLTL encoding of a goal model as input 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 [8]. 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.

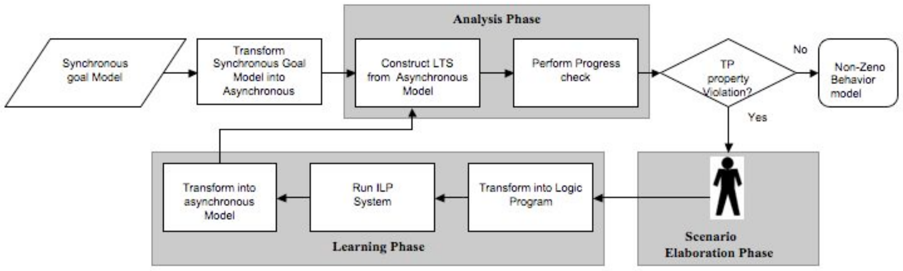
Analysis Phase
Synchronous goal Model
Transform Synchronous Goal Model into Asynchronous
Construct LTS from Asynchronous Model
Perform Progress check
TP property Violation?
No
Yes
Non-Zeno Behavior model
Transform into asynchronous Model
Run ILP System
Transform into Logic Program
Learning Phase
Scenario Elaboration Phase

Fig. 1. Approach overview

The check is performed by assuming maximal progress of the system with respect to the environment (a standard assumptions for reactive systems), and weak fairness [15]. 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 full set G_s of initial goals includes, in addition to the *PumpOffWhenLowWater*, the following LTL goal assertions: $PumpOffWhenMethane = \Box(CriticalMethane \rightarrow \bigcirc\neg PumpOn)$, and $PumpOnWhenHighWaterAndNoMethane = \Box(\neg CriticalMethane \wedge HighWater \rightarrow \bigcirc PumpOn)$.

The application of the analysis phase to the asynchronous FLTL translation of G_s and asynchronous preconditions $\Box(tick \wedge \neg HighWater \rightarrow \bigcirc\neg switchPumpOn)$ and $\Box(tick \wedge HighWater \wedge \neg CriticalMethane \rightarrow \bigcirc\neg switchPumpOff)$, gives the following output:

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

The output represents an infinite trace compactly displayed as a (finite) trace with the prefix ⟨`tick,signalCriticalMethane,signalHighWater,tick, switchPumpOn, switchPumpOff`⟩ followed by a cycle in which *tick* does not occur ⟨`switchPumpOn, switchPumpOff, switchPumpOn,`...⟩. The capitalized text on the right column indicates the fluents that are true after the occurrence of each event of the trace prefix on the left. The trace indicates that a precondition for at least one of the 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 preconditions 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 precondition for *switchPumpOn* needs strengthening to prevent the pump being switched on unnecessarily.

4.3 Scenario Elaboration Phase

This phase assumes the engineer will elaborate the violation trace generated by the LTSA and produce a set of positive and negative scenarios. The engineer is assumed to identify an event in the trace returned by the LTSA that should not have occurred at a particular position in the trace. The prefix starting from the initial state of that trace up to and including the undesirable event is identified as a negative scenario. Hence, given a trace of the form $\langle w_1, e, w_2\rangle$, a negative scenario is $\langle w_1, e\rangle$. The intended meaning of $\langle w_1, e\rangle$ is that if the system exhibits w_1 then 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 a scenario which shows a positive occurrence of e. This is a scenario that starts from the same initial state, is consistent with the goal model and terminates with a tick (i,e, $\langle x_1, e, x_2\rangle$ where *tick* is the last event in x_2). Note that because scenarios are finite traces, positive scenarios are not meant to exemplify non-zeno traces. They merely capture desirable system behavior which is consistent with the given model. Note that the same model generated by LTSA can be walked through or animated by the engineer to generate positive scenarios that are consistent with existing goals and operation conditions.

Returning to the zeno trace described above, the engineer may identify the first occurrence of `switchPumpOn` as incorrect. A negative scenario becomes ⟨`tick, signalCriticalMethane, signalHighWater, tick, switchPumpOn`⟩ 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* must be provided. A positive scenario could be ⟨ `tick, signalHighWater, tick, switchPumpOn, tick`⟩. The completion of this phase is noted by the identification of at least one positive and one negative scenario.

4.4 Learning Phase

This phase is concerned with the inductive learning computation of missing event preconditions with respect to a given set of positive and negative scenarios. It makes use of an ILP framework, called XHAIL [19].

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 [20,18]. Intuitively, within the context of learning preconditions, the background knowledge is the fluent definitions D and the goal model G_a, currently considered by the analysis phase. The set of positive and negative scenarios generated during the scenario elaboration phase form the examples. The learned (set of) asynchronous preconditions, Pre, is the hypothesis that added to G_a generates an LTS model that accepts the positive scenarios but none of the negative ones. We refer to Pre as the *correct extension of a goal model with respect to scenarios.*

To apply XHAIL to the task of learning preconditions, the asynchronous FLTL goal model G_a, with preconditions computed in previous iterations, fluent definitions D, and the elaborated positive and negative scenarios $\Sigma_P \cup \Sigma_N$ are encoded into a, semantically equivalent, Event Calculus (EC)[17] logic program Π.

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. The two sorts P and T denote, respectively, positions (of (non-)observable states) 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 [17].

To relate positions and time units within a given scenario, our EC programs use the predicate `posInTime(p,t,s)`. For example, the scenario ⟨`tick, signalHighWater, tick, switchPumpOn, tick`⟩ is encoded using the ground atoms `happens(tick,0,0,s)`, `happens(signalHighWater,1,0,s)`, `happens(tick,2,1,s)`,... for the event transitions, `posInTime(0,0,s)`, `posInTime(1,0,s)`, `posInTime(2,1,s)`, `posInTime(3,1,s)`, `posInTime(4,2,s)`, ... for the relation between position and time units. Finally, to capture the notion of synchronous satisfiability in terms of asynchronous semantics our programs make use of the predicates `holdsAtTick` and `notHoldsAtTick`, which are defined as follows:

```
holdsAtTick(f,t,s):- attempt(tick,p,t,s),holdsAt(f,p,s),
                     posInTime(p,t,s)                        (1)
```

```
notHoldsAtTick(f,t,s) :- attempt(tick,p,t,s), not holdsAt(f,p,s)
                         posInTime(p,t,s)                    (2)
```

where `p`, `t` and `s` are respectively position, time unit and scenarios[1]. Axioms (1) and (2) 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 its starting `tick` is attempted.

EC programs are equipped with a set of domain-independent core axioms suitable for reasoning about effects of events over fluents. A full definition of these axioms is given in [21]. These include, in particular, an axiom for differentiating the possibility of an event occurring from it actually occurring. This is defined as follows: `happens(e,p,t,s):-attempt(e,p,t,s), not impossible(e,p,t,s).` It is one of the key axioms in our learning process, as it relates the occurrence of an event (`happens(e,p,t,s)`) with the notion of its preconditions, captured by rules defining the predicate `impossible`.

From Goal Models to EC Programs. Given a goal model G_a, written in asynchronous FLTL, and a set of fluent definitions D, a mapping τ has been defined that automatically generates from G_a and D an EC program of the type described above. In brief, the mapping assigns to each FLTL fluent definition $f \equiv \langle\{a_i\},\{b_i\}\rangle$ *initially* I the set of atomic literals comprising of `initially(f,S)`, for those fluents f where I is set to true, `initiates(a`$_i$`,f,P,S)`, for each a_i in the initiating set of f and `terminates(b`$_i$`,f,P,S)`, for each b_i in the terminating set of f. 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)` and `terminates(switchPumpOff, pumpOn,P,S).`

Asynchronous FLTL goal assertions are, instead, encoded into integrity constraints[2], using only `holdsAtTick` and `notHoldsAtTick` predicates. For instance, applying the function τ to the asynchronous formalisation of the goal *PumpOffWhenLowWater* gives the EC integrity constraint

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

where `next(T2,T)` means *T2* is the next time point. Asynchronous FLTL preconditions are encoded into rules for the predicate `impossible`. So a precondition $\Box(\bigwedge_{1\leq i\leq n}(\neg)f_i) \rightarrow \bigcirc\neg e\ \mathsf{W}\ tick)$ would be expressed by the EC rule:

```
impossible(e,P,T,S):-(not)HoldsAtTick(f_1,T,S),..,
                     (not)HoldsAtTick(f_n,T,S)          (3)
```

In [1] the authors have shown that the above translation function is sound with respect to the stable model semantics [7].

Learning Preconditions. To learn event preconditions, positive and negative scenarios generated during the scenario elaboration phase have to be translated into our EC program. The translation depends on the event for which the precondition axiom is to be learnt. Without loss of generality, we assume that

[1] The operators $:-$ and , denote implication and conjunction operators respectively in logic programming.

[2] An integrity constraint (IC) is a disjunction of literals with no positive literal.

preconditions are to be learnt always for the last event of each negative scenario. The encoding of negative and positive scenarios contribute to both the background knowledge and the examples of our learning task. For a positive scenario $\sigma_P = \langle e_1, e_2, ..., e_k \rangle$, the facts `attempt(e_i,i-1,t,`$\sigma_P$`)` and `posInTime(i-1,t,`$\sigma_P$`)` are added to the background knowledge, and the facts `happens(e_i ,i-1,t,`σ_P`)` are added to the example. For each negative scenario of the form $\sigma_N = \langle e_1, e_2, ..., e_l \rangle$ in Σ_N, the facts `attempt(e_j,j-1,t,`σ_N`)` and `posInTime(j-1,t,`σ_N`)` are added to the background knowledge, and the facts `happens(e_j ,j-1,t,`σ_N`)` together with the atom `not happens(e_l ,l-1,t,`σ_N`)` are added to the examples.

The search space of all possible preconditions is defined by a language bias, which specifies the predicates that can appear in the hypothesis. In our learning task, the language bias defines the predicate `impossible` to appear in the head of the H rule, and the predicates `holdsAtTick` and `notHoldsAtTick` to appear in the body. For a detailed description of the XHAIL learning algorithm the reader is refer to [20]. To describe an example of learned precondition, consider again our example of the Mine Pump system, where the set of goals are as states in Subsection 4.2, with associated fluent definitions, and the positive and negative scenarios in Subsection 4.3. The XHAIL is applied to the EC programs B and E generated from these inputs. The system computes the ground rule:

```
impossible(switchPumpOn,4,2,σN) :- holdsAtTick(highWater,2, σN),
                                   notHoldsAtTick(pumpOn,2,σN),
                                   holdsAtTick(criticalMethane,2,σN),
                                   posInTime(4,2,σN).
```

And then generalizes it into the hypothesis: `impossible(switchPumpOn,X,Y,Z) :- holdsAtTick(criticalMethane,Y,Z)`. This output is then translated back into the FLTL assertion $\Box($ *tick* $\wedge$ *criticalMethane* $\rightarrow \bigcirc\neg$*switchPumpOn W tick*$)$.

4.5 The Cycle

At the end of each iteration, the learned preconditions 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 property is formally captured by the following theorem and constitutes the main invariant of the approach.

Theorem 1. *Let G_a be an asynchronous goal model, D a set of fluent definitions, and $\Sigma_P \cup \Sigma_N$ a set of positive and negative scenarios. Let $(B, E) = \tau(G_a, D, \Sigma_P, \Sigma_N)$ be the EC programs generated by the translation function τ. Let H be the set of preconditions computed by the XHAIL system as inductive solution for the programs (B, E) such that $B \cup H \models E$. Then the corresponding set Pre of asynchronous FLTL preconditions, such that $\tau(Pre) = H$, is a correct extension of G_a with respect to $\Sigma_P \cup \Sigma_N$.*

This process is expected to be repeated until all the preconditions necessary to guarantee, together with the initial goal model, the construction of a non-zeno

LTS model. Although the convergence of this process has not yet been studied fully, experiments have shown so far that to avoid the derivation of a zeno-model, one only needs to learn a sufficient set of preconditions, i.e. triggering conditions are not necessary to avoid non-zeno traces.

5 Validation of the Approach

We validated our approach with two case studies, the Mine Pump Controller [10], parts of which have been used as a running example in this paper, and the Injection System [2]. The methodology used was to start with goal models formalised in the KAOS goal oriented approach (the mine pump [13] and the safety injection [12] systems), to apply the approach iteratively using informal existing documentation on the case studies to inform the elaboration of zeno traces into positive and negative scenarios and to compare the preconditions learned against the manually operationalised models of the provided goal models.

In the mine pump case study, it was necessary to learn three preconditions where as the safety injection system required only two. Moreover, in the safety injection system the set of preconditions needed for the process to converge, and reach a goal model from which a non-zeno behaviour model can be derived, was a subset of the preconditions of the fully operationalised goal model[12]. This indicates that the operationalisation process presented in [13] and required in [12] to build non-zeno models introduces unnecessary (and labour intensive) work. This also indicates that the process of resolving time progress violation in behavior models synthesized from goal models is a first step towards an automated procedure to produce a complete operationalisation of goal models.

6 Related Work

Automated reasoning techniques are increasingly being used in requirements engineering [11,3,4]. Among these, the work most related to our approach is [11], 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 [11] 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 [3] 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 derive an LTS that covers all positive scenarios but none of the negative ones.

The generated LTS can then be used for formal event-based analysis techniques (e.g. check against the goals expressed as safety properties). Our approach, on the other hand, uses the LTSA to generate the LTS models directly from goal models, so our LTS models are always guaranteed to satisfy the given goals.

The technique in [12] 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 [12] but does not require a fully operationalised model. Rather it provides automated support for completing an operational model with respect to the given goals.

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 precondition from scenarios. These preconditions can incrementally be added to the initial goal model so to generate at the end of the cycle a non-zeno behavior model. The precondition learned at each iteration has the effect of removing zeno traces identified by the LTS 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 elaboration process by using other forms of learning techniques (e.g. abduction) is being considered. Future work includes learning other forms of requirements such as trigger conditions, learning preconditions with bounded temporal operators that refer to the past such as B and S in [13], and to integrate the approach within a framework for generating a set of required pre- and trigger-conditions that is complete [13] with respect to a given goal model.

References

1. Alrajeh, D., Ray, O., Russo, A., Uchitel, S.: Extracting requirements from scenarios with ILP. In: Muggleton, S., Otero, R., Tamaddoni-Nezhad, A. (eds.) ILP 2006. LNCS (LNAI), vol. 4455, pp. 64–78. Springer, Heidelberg (2007)
2. Courtois, P.J., Parnas, D.L.: Documentation for safety critical software. In: Proc. 15th Int. Conf. on software engineering (1993)
3. Damas, C., Dupont, P., Lambeau, B., van Lamsweerde, A.: Generating annotated behavior models from end-user scenarios. IEEE Transactions on Software Engineering 31(12), 1056–1073 (2005)

4. Damas, C., Lambeau, B., van Lamsweerde, A.: 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)
5. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Science of Computer Programming 20 (1), 3–50 (1993)
6. Darimont, R., van Lamsweerde, A.: Formal refinement patterns for goal-driven requirements elaboration. In: Proc. of the 4th ACM Symp. on the Foundations of Software Engineering (1996)
7. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R.A., Bowen, K. (eds.) Proc. of the 5th Intl. Conf. on Logic Programming, MIT Press, Cambridge (1988)
8. Giannakopoulou, D., Magee, J.: Fluent model checking for event-based systems. In: Proc. 11th ACM SIGSOFT Symp. on Foundations Software Engineering (2003)
9. Heitmeyer, C., Bull, A., Gasarch, C., Labaw, B.: Scr*: A toolset for specifying and analyzing requirements. In: Proc. of the 10th Annual Conf. on Computer Assurance (1995)
10. Kramer, J., Magee, J., Sloman, M.: Conic: An integrated approach to distributed computer control systems. In: IEE Proc., Part E, vol. 130 (January 1983)
11. Van Lamsweerde, A., Willemet, L.: Inferring declarative requirements specifications from operational scenarios. IEEE Transactions on Software Engineering 24(12), 1089–1114 (1998)
12. Letier, E., Kramer, J., Magee, J., Uchitel, S.: Deriving event-based transitions systems from goal-oriented requirements models. Technical Report 02/2006, Imperial College London (2006)
13. Letier, E., Van Lamsweerde, A.: Deriving operational software specifications from system goals. In: Proc. 10th ACM SIGSOFT Symp. on Foundations of Software Engineering (2002)
14. Letier, E., van Lamsweerde, A.: Agent-based tactics for goal-oriented requirements elaboration. In: Proc. of the 24th Intl. Conf. on Software Engineering (2002)
15. Magee, J., Kramer, J.: Concurrency: State Models and Java Programs. John Wiley and Sons, Chichester (1999)
16. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, Heidelberg (1992)
17. Miller, R., Shanahan, M.: Some alternative formulation of event calculus. In: Computer Science; Computational Logic; Logic programming and Beyond, vol. 2408 (2002)
18. Muggleton, S.H.: Inverse Entailment and Progol. New Generation Computing, Special issue on Inductive Logic Programming 13(3-4), 245–286 (1995)
19. Ray, O.: Using abduction for induction of normal logic programs. In: Proc. ECAI 2006 Workshop on Abduction and Induction in AI and Scientific Modelling (2006)
20. Ray, O., Broda, K., Russo, A.: A hybrid abductive inductive proof procedure. Logic Journal of the IGPL 12(5), 371–397 (2004)
21. Shanahan, M.P.: Solving the Frame Problem. MIT Press, Cambridge (1997)
22. Uchitel, S., Brunet, G., Chechik, M.: Behaviour model synthesis from properties and scenarios. In: Proc. of the 29th IEEE/ACM Intl. Conf. on Software Engineering (2007)