Learning Operational Requirements from Goal Models

Dalal Alrajeh[‡] Jeff Kramer[‡] Alessandra Russo[‡*] Sebastin Uchitel[‡†]

[‡] Department of Computing,
Imperial College London, UK
{da04, jk, ar3, s.uchitel}@doc.ic.ac.uk

[†] Departamento de Computaciòn, FCEyN, UBA
Buenos Aires, Argentina
suchitel@dc.uba.ar

Abstract

Goal-oriented methods have increasingly been recognised as an effective means for eliciting, elaborating, analysing and specifying software requirements. A key activity in these approaches is the elaboration of a correct and complete set of opertional requirements, in the form of pre- and trigger-conditions, that guarantee the system goals. Few existing approaches provide support for this crucial task and mainly rely on significant effort and expertise of the engineer.

In this paper we propose a tool-based framework that combines model checking, inductive learning and scenarios for elaborating operational requirements from goal models. This is an iterative process that requires the engineer to identify positive and negative scenarios from counterexamples to the goals, generated using model checking, and to select operational requirements from suggestions computed by inductive learning.

Keywords: Goal-oriented requirements engineering, scenarios, inductive learning.

1. Introduction

Requirements Engineering (RE) is an integral part of the software development life-cycle, concerned with elicitation, elaboration, specification, analysis and documentation of the goals of a system-to-be. Each of these activities contributes to the development of a software requirements specification that is complete and correct with respect to the system goals [16].

A number of goal-based techniques have been developed to support the process of requirements acquisition [7, 19], but only few have focused on the elaboration of operational requirements from high-level goals. Letier and van Lamwsverde [16] have developed an approach based on operationalisation patterns which allows the derivation of operational requirements in the form of pre- and trigger-conditions from goals expressed in Linear Temporal Logic (LTL). Requirements generated by this approach are guaranteed to be correct. However, patterns are restricted to a collection of goal and requirement templates, and their application requires a fully refined goal model. Consequently, the elaboration of operational requirements from goals remains constrained to the set of templates and can be labour intensive and error-prone. The availability of a more systematic and automated approach would therefore benefit the process of operationalising goals.

In this paper, we propose a formal, tool-supported framework that combines model checking, inductive learning and scenarios to elaborate operational requirements, in the form of pre- and trigger-conditions, that are *correct* and *complete* with respect to a given set of system goals. The framework is defined as an iterative process that consists of four conceptual phases. First, in the *analysis* phase, an existing partial specification of operational requirements is verified against a given goal model using a model checker. If verification is unsuccessful, the counterexample automatically generated is used in the *scenario elaboration* phase where an engineer elaborates it into a set of positive and negative scenarios. In the *learning phase*, the partial specification of operational requirements and scenarios are used by a non-monotonic inductive learning system to compute a set of operational requirements that covers all positive scenarios and eliminates all negative ones. Finally, in the *selection* phase, the engineer selects the operational requirements to be added from the list proposed by the learning phase. The four phases are then repeated until no goal violation is detected.

In summary, we provide a tool-supported approach for learning operational requirements from goals in which the detection of the incompleteness of the requirements with respect to the goals and the generation of possible "repairs" is fully automated. The engi-

*Dr. Russo's research is continuing through participation in the International Alliance sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defense.

neer's intervention is required only for the elaboration of scenarios and the selection of operational requirements from the suggested repairs.

This paper is organized as follows. Section 2 illustrates the problem using a simplified version of the Safety Injection System described in [4]. Section 3 describes the main features of our approach. Section 4 provides an illustrative case study on a real event-driven system, the Mine Pump System [12]. Section 5 discusses some of the outcomes of the case studies and how the approach relates to other applications of learning techniques to requirements engineering. A summary and some remarks about future work conclude the paper.

2 Motivation

Consider the safety injection system described in [4, 14] where an Engineered Safety Feature Actuation System (ESFAS) is needed to prevent or minimize damage to the coolant system whenever a fault occurs such as loss of coolant. The system includes a built-in sensor which measures the pressure levels. If it detects an fall below a preset threshold then it sends a safety injection signal to the safety feature component which is responsible for dealing with the incident. Suppose the engineers identifies the goals: $G1$ = "*The safety injection signal should be on when the water pressure is below the low set point while the safety injection is not overridden*" and $G2$ = "*The safety injection signal should be on when the water pressure is below the low set point while the reactor is on*". The problem of elaborating operational requirements from such goals lies in identifying the system operations and characterizing their pre- and trigger-conditions (i.e. the conditions under which an event *can* and *must* happen respectively) such that they guarantee the satisfiability of the goals over (discrete time) event-based models [15].

The ESFAS operations are *startSignal* and *stopSignal* for turning the safety injection signal on and off, and *enableSignal* and *overrideSignal* for enabling and overriding the safety injection signal. An example of operational requirement that needs to be elaborated for the operation *stopSignal*, given the above goals, is the pre-condition "the water pressure is not low or the safety injection signal is overridden", meaning that if the water pressure is 'Low' and the safety injection signal is not 'Overridden' at the beginning of a time unit, then *stopSignal* cannot occur during that time unit. Another example of operational requirement is the trigger-condition for *startSignal* of the form "the water pressure is low and the safety injection signal is not overridden", meaning that if the water pressure is 'Low' and the safety injection signal is not 'Overridden' at the beginning of a time unit, then *startSignal* must occur during that time unit.

How can pre- and trigger-conditions over operations be derived in a systematic way and so that they guarantee the system goals? In this paper we propose a formal tool-assisted approach for supporting the elaboration of operational requirements like those described above. It is an iterative process that makes use of standard model-checking, scenarios and inductive learning to elicit new operational requirements in each cycle.

Let us assume that the initial specification includes a trigger-condition for *startSignal* and the following domain knowledge about the operations *startSignal*, *stopSignal*, *enableSignal* and *overrideSignal*: *startSignal* turns the safety Injection on, *stopSignal* turns it off, *enableSignal* enables it and *overrideSignal* overrides it.

A valid implementation of such an operational specification allows the scenario depicted in Figure 1. Scenarios are represented as Message Sequence Charts [11], where the clock labels the beginning/end of time units. Figure 1 describes a situation in which initially, the water pressure is below low, the safety injection signal is off and overridden. Then *enableSignal* occurs making the safety injection signal enabled, followed by a *rise* and a *drop* in the water pressure occurs. Subsequently, *startSignal* turns the safety injection signal on and then a rise in the water pressure happens. Finally, *stopSignal* is commanded. Notice, in Figure 1, the last time unit (i.e. between the last two clock labels) starts with the water pressure being below low and the safety injection signal not overridden and ends with the safety injection signal being off. This is a violation of goal $G1$ described above.

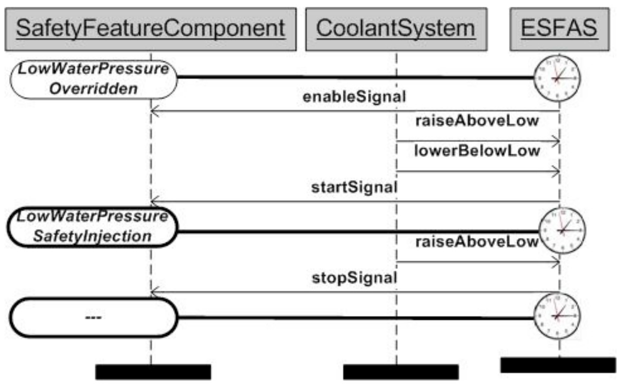

Figure 1. A violation of goal $G1$.

An engineer, faced with this example could recognize this scenario as an example of an incorrect occurrence of *stopSignal* and could easily provide an example of a correct occurrence of the same event. Alternatively, the engineer could identify the trace as an undesirable occurrence of *startSignal* since there is no point in sending the signal if the signal is initially overridden. Let us assume the latter, and that the engineer provides the scenario in Figure 2 as an example of desirable occurrence of *startSignal*.

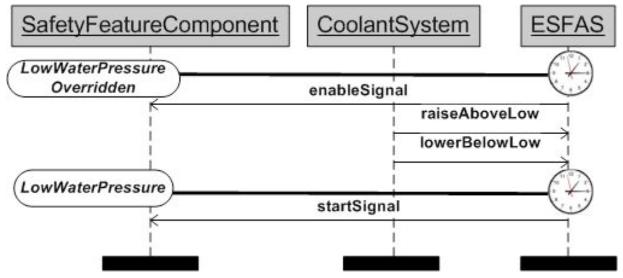

Figure 2. Desirable Occurrence of *startSignal*

From these two scenarios it is possible to learn the pre-condition for *startSignal* “the safety injection signal is not overridden”, which could be validated by the engineer and then added to the specification. The new specification would no longer allow traces of the form given in Figure 1. The process above could be repeated, presenting the engineer with other examples of goal violations (if any). For instance, checking the new specification against $G2$ would result in the counterexample shown in Figure 3. The engineer must then provide an example of how the system should have behaved (e.g., Figure 4) from which the new trigger-condition, “the water pressure is below low while the reactor is on”, for *startSignal* could be learned. Such a procedure could be continued until the complete operational specification shown in Figure 5 is produced.

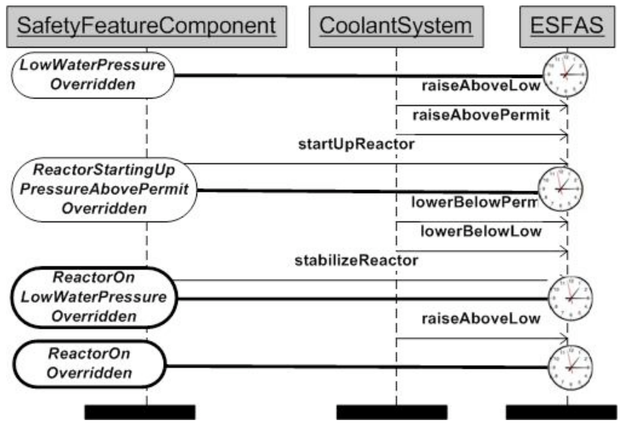

Figure 3. A violation of goal $G2$

In this paper we shall demonstrate how counterexamples to system goals are automatically generated (see Section 3.1), how an engineer can elaborate from these counterexamples positive and negative scenarios (see Section 3.2), how proposals for operational requirements are automatically induced from these scenarios (see Section 3.3) and how an engineer may pick between the various proposals (see Section 3.4).

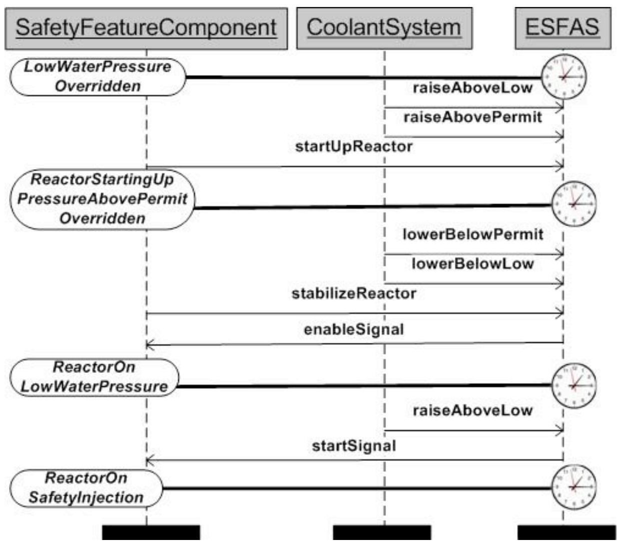

Figure 4. Desirable Occurrence of *startSignal*

```
Pre(startSignal) =¬Overridden
Pre(stopSignal)=¬LowWaterPressure ∨ (Overridden
                        ∧ ¬ReactorOn)
Trig(startSignal)=LowWaterPressure∧
                        (¬Overridden ∨ ReactorOn)
```

Figure 5. A complete set of operational requirements with respect to the ESFAS goals

3 Approach

Starting from a collection of system goals (or goal model), G, and operational requirement specification, *Spec*, that does not satisfy the system goals, our approach provides a systematic way for extending the specification with pre- and trigger-conditions for system’s events so that the derived new specification $Spec'$ satisfies the goals. The approach is composed of four phases (see Figure 6).

3.1 Phase 1: Analysis Phase

This phase is concerned with *automatically* checking whether a given operational requirement specification *Spec* satisfies a given goal model G.

Goals and operational requirements are expressed in asynchronous Fluent Linear Temporal Logic (FLTL) [10], a flavour of linear temporal logic for describing event- and state-based properties using fluents. Fluents are propositions whose truth value changes over time and are defined with respect to an initiating and termi-

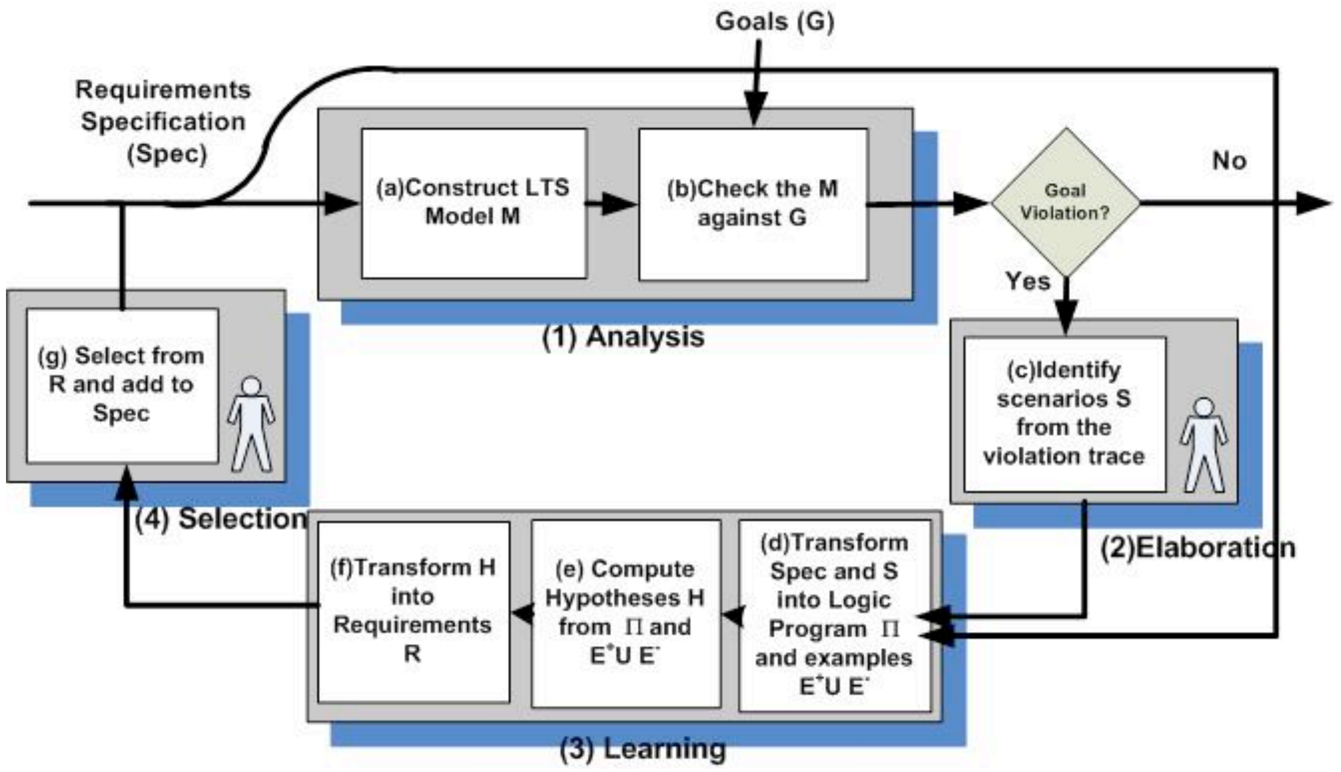
Goals (G)
Requirements Specification (Spec)
(a)Construct LTS Model M
(b)Check the M against G
Goal Violation?
No
Yes
(1) Analysis
(g) Select from R and add to Spec
(c)Identify scenarios S from the violation trace
(4) Selection
(2)Elaboration
(f)Transform H into Requirements R
(e) Compute Hypotheses H from Π and E⁺U E⁻
(d)Transform Spec and S into Logic Program Π and examples E⁺U E⁻
(3) Learning

Figure 6. Approach overview

nating set of events. A fluent becomes true (resp. false) whenever an event from its predefined *initiating* (resp. *terminating*) set of events occurs. For instance, the fluent *SafetyInjection* has the events *startSignal* and *stopSignal* in its initiating and terminating events. The fluent definitions for this system are given in Figure 8.

The asynchronous FLTL formalization of goals and operational requirements uses a *tick* event to explicitly model time. Following the approach in [15], we require goals to be true only at the occurrence of a *tick*. For example, the informal goals $G1$ and $G2$ described at the beginning of Section 2 are formalized in Figure 7. Note that these properties can be automatically generated from formulae written in synchronous FLTL [15], a logic that abstracts away tick events.

$$\begin{aligned}G1 = \Box(&tick \rightarrow ((LowWaterPressure \wedge \neg Overridden) \rightarrow \\ &\bigcirc(\neg tick\ W\ (tick \wedge SafetyInjection)))\\ G2 = \Box(&tick \rightarrow ((LowWaterPressure \wedge ReactorOn) \rightarrow \\ &\bigcirc(\neg tick\ W\ (tick \wedge SafetyInjection)))\end{aligned}$$

Figure 7. ESFAS goals expressed in asynchronous FLTL

The input to the analysis phase is a set of system goals, G, and a partial specification, *Spec*, including fluent definitions and existing operation requirement, all formalised in asynchronous FLTL. Analysis consists in using the LTSA model-checker [17] to verify if *Spec* satisfies G. This is done by first automatically synthesising from *Spec* its least constrained LTS model M (as presented in [15]), and then by checking M against G using the tool's model checking algorithm. The result of the model checker can be either that *Spec* satisfies G, in which case the process successfully terminates, or that a counterexample is computed showing that *Spec* violates G. The counterexample is a system execution that satisfies all operational requirements in *Spec* yet does not satisfy at least one of the goals in G.

```
fluent SafetyInjection =
    ⟨{startSignal}, {stopSignal}⟩
fluent Overridden =
    ⟨{overrideSignal}, {enableSignal}⟩initially True
fluent LowWaterPressure =
    ⟨{lowerBelowLow}, {raiseAboveLow}⟩initially True
fluent PressureAbovePermit =
    ⟨{raiseAbovePermit}, {lowerBelowPermit}⟩
```

Figure 8. Fluent definitions for Safety Injection System

Returning to the safety injection system, let us assume the fluents definitions to be as in Figure 8 and the existing operational requirements to be initially composed of only the trigger-condition for *startSignal*, given by `Trig(startSignal) = (LowWaterPressure∧¬ Overridden)` and automatically formalised in asynchronous FLTL as:

$$\Box(tick \rightarrow ((LowWaterPressure \wedge \neg Overridden) \wedge \\ \neg PumpOn \rightarrow \bigcirc(\neg tick\ U\ startSignal)))$$

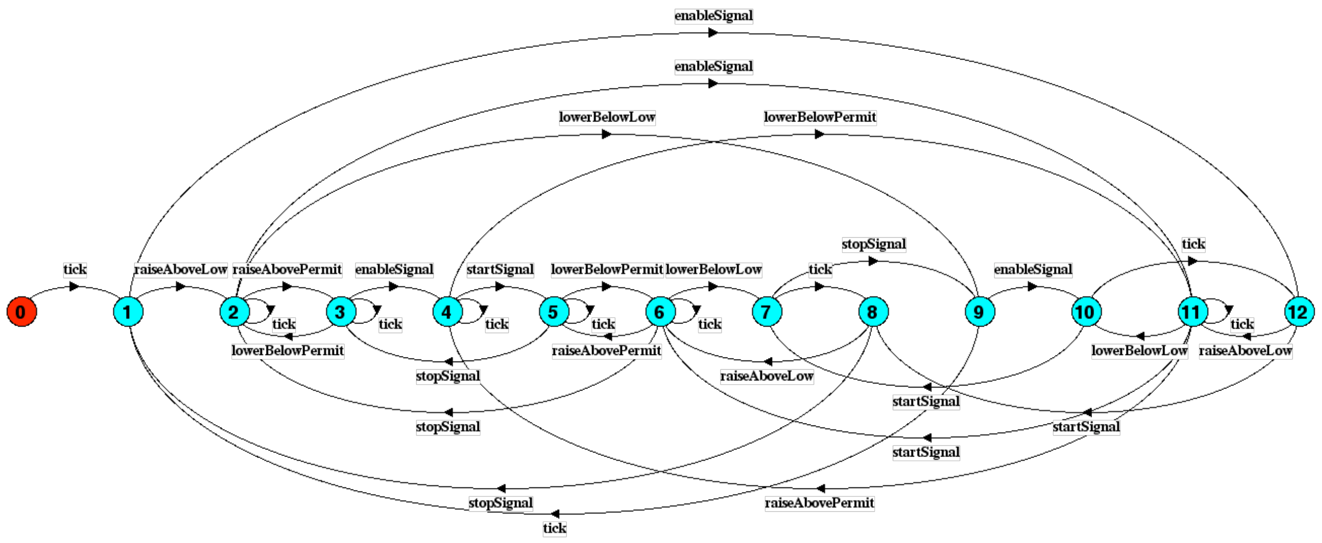

Figure 9. **LTS model of initial specification**

The first activity of this phase is to construct an LTS (Figure 9) that characterizes *Spec*. The second activity is to check the synthesised LTS against the system goals G given in Figure 7. In this case, LTSA generates the following violation trace, where the events in the left column show the shortest path leading to goal violation and the text on the right shows the fluents that are true after the execution of each event.

```
Trace to property violation in
SafetyInjectionWhenLowPressureAndNotOverridden:
       tick            LowWaterPressure && Overridden
       enableSignal    LowWaterPressure
       raiseAboveLow
       lowerBelowLow   LowWaterPressure
       startSignal     LowWaterPressure && SafetyInjection
       tick            LowWaterPressure && SafetyInjection
       raiseAboveLow   SafetyInjection
       stopSignal
       tick
Analysed in:  5ms
```

The above counterexample shows an execution that satisfies *Spec* but violates the goal $G1$. In fact, it corresponds to the scenario depicted in Figure 1. The violation indicates that the existing operational requirements are incomplete, and provides the basis for producing positive and negative scenarios from which new operational requirements can be learned.

3.2 Phase 2: Scenario Elaboration

This phase requires the involvement of an engineer to elaborate, from the counterexample generated by the analysis phase, examples of how the system should and should not behave. The aim is then to elaborate a set of positive and negative scenarios from a given violation trace. Scenarios are finite sequences, $\langle e_1, ..., e_n\rangle$, of event transitions. Such a sequence is *accepted* (resp. *rejected*) by an LTS if there is (resp. not) a trace in the LTS that has this sequence as its prefix. Positive (resp. negative) scenarios are desirable (resp. undesirable) sequences of event transitions, which may be accepted or not by an LTS. The aim of the next phases is to extend the operational requirements so that the LTS synthesised from it accepts the positive scenarios and rejects the negative scenarios identified in this phase.

In the counterexample produced by the analysis phase, there is at least one event that occurs and that should not have done so at that point in the trace. As the approach is concerned with developing operational requirements, we allow selecting only system controlled events and the *tick* event as undesirable events. Consequently, the engineer is required to elaborate from a given violation trace a negative scenario of the form $\langle e_1, ..., e_n\rangle$ where e_n is either an undesirable system event or the *tick* event. In general, if e_n is a *tick* event the negative scenario would indicate that at least one system event e should have been triggered before the *tick* event e_n for the goal to be satisfied in that trace. If the undesirable event e_n is a system event then the negative scenario would show that the event e_n should not have occurred for the goal to be satisfied in that trace. For instance, in the counterexample shown in Figure 1, according to the discussion in Section 2, *stopSignal* is an undesirable event. Once such an event is identified, the prefix of the violation trace up to and included the undesirable event constitutes a *negative scenario*. For instance, the sequence ⟨*tick, enableSignal, raiseAboveLow, lowerBelowLow, startSignal, tick, stopSignal*⟩ would be the negative scenario elaborated from the feedback given in Figure 1.

Whereas the selection of the negative scenario is naturally implied by the violation trace and the goal it violates, the elaboration of positive scenarios is *less rigid*. The main underlying principle is that the positive scenario should be: i) a sequence of events exemplifying a desirable occurrence of the event e_n in question, ii) accepted by the LTS model of the given operational specification, and iii) consistent with all the goal.

In cases where the negative scenario ends with *tick*, the positive scenario should include, before this *tick* event, the occurrence of an event that will make it permissible. If the negative scenario has a system event as the last event e_n, then a positive scenario could, in principle, be any finite sequence of events starting from the initial state and ending with e_n, that is accepted by the LTS model and satisfies all goals. The engineer could use the animation feature of the LTSA system to animate the LTS model synthesised in the previous phase through the same prefix of the violation trace up to but excluded the event e_n and then follow alternative event transitions until the event e_n is subsequently encountered and the goal is satisfied. An heuristic for elaborating positive scenarios as counterparts to a negative scenario $\langle e_1, ..., e_n \rangle$ is then to detect sequences of the form $\langle e_1, \ldots, e_{n-1}, e_{k1}, \ldots, e_{km}, e_n \rangle$ where $\langle e_1, \ldots, e_{n-1} \rangle$ is a prefix of the negative scenario, and the sequence $e_{k1}, \ldots, e_{km}, e_n$ is an alternative, but this time desirable, sequence of transitions ending with the event e_n.
The question that naturally arises here is *how different should the positive scenarios be from an elaborated negative scenario?* A possible heuristic is the *richness* of positive scenarios with respect to a negative scenario. This can be measured by how many state-based fluents share the same truth value in the state where the conditions of an operational requirement (either pre- or trigger-conditions) are evaluated, over the collection of positive and negative scenarios elaborated by the engineer. The *fewer* these fluents are the *richer* the collection of positive scenarios is with respect to the negative scenario. Our case study indicates that there is a linear dependancy between the richness of the scenarios and that of pre- and/or trigger-conditions computed during the learning phase.

3.3 Phase 3: Learning

The objective of this third phase is to compute suggestions of operational requirements, pre- and trigger-conditions, that added to the specification resolve the violation detected during the analysis phase. The input to this phase includes a (partial) specification of operational requirements, and the positive and negative scenarios elaborated by the engineer. By construction, the previous phase guarantees that both the positive and negative scenarios are accepted by the LTS synthesised in the analysis phase, so the outcome of the learning phase has to be a set of operational requirements that together with the current specification, will yield an LTS that still accepts the positive scenarios but none of the negative ones.

In non-monotonic inductive learning terms [20], this problem is expressed as the task of learning a set of hypothesis that, added to a given specification, removes the negative examples E^- from the current set of consequences, while preserving the positive examples E^+. Our learning phase uses the non-monotonic learning system XHAIL [20]. This takes in input a set of Prolog rules, Π, and a set of positive and negative examples, E^+ and E^-, also expressed in Prolog, and computes a set of new (Prolog) rules, H, that added to Π derives E^+ but not E^-. The computation of H is performed within a search space defined by a *language bias*, also given to the XHAIL system in input to define the syntactic form of the rules that can be learned.

The first step of our learning phase is to encode the FLTL specification and the scenarios into Prolog. An automated encoding function has been developed [1] that provides a sound implementation of an FLTL specification into an Event Calculus [18] Prolog program (see Theorem 1 in [1]) to reason about time and effect of events. The positive and negative scenarios are also encoded into appropriate Prolog facts (E^+ and E^-). The language bias must be pre-set for the system. For validation purposes, in our case studies, we have mainly focused on the syntactic form of pre- and trigger-conditions considered by the KAOS approach [16], and corresponding Prolog-based language bias has been defined [1]. But, as discussed in Section 5, the approach is general enough to support the computation of more elaborate temporal rules. The XHAIL system is then used as a "back-end" tool on the automatically generated Prolog program and examples. The outcome is then also automatically translated back into the FLTL syntactic form of pre- and/or trigger-conditions and returned to the engineer. As the encoding function from FLTL and scenario into Prolog and vice-versa are automatically done, the learning phase is essentially a "black-box" process of our approach. No expertise is required by the engineer of Prolog and of the usage of the learning system. The formal correctness of this "black box" process has been shown in [1] for the case of learning pre-conditions. The correctness of the process of learning trigger-conditions is a natural extension of these results. In simple terms, it can be shown that the pre- and trigger rules learned by the XHAIL system to cover given positive examples and none of the negative ones, translated back into the FLTL representation provide the operational requirements needed to be added to the current specification in order to remove the negative scenario while preserving the positive one.

Within the context of the safety injection system, given the specification in Subsection 3.1 and the elaborated negative and positive scenarios illustrated in Subsection 3.2, the learning phase would return `Pre(startSignal)= ¬Overridden` indicating "safety injection signal not overridden" as a pre-condition for *startSignal* formally expressed as:

$$\Box(tick \rightarrow (Overridden \rightarrow \bigcirc(\neg startSignal\ U\ tick))$$

In the next *analysis phase*, the new specification obtained after adding these new requirements, produces an LTS model that still accepts the positive scenarios but now rejects the negative ones and therefore rejects the trace identified in the previous analysis phase.

3.4 Phase 4: Selection

The outcome of the learning phase consists of a set of operational requirements. The engineer is required to select from the proposed operational requirements and add his/her selection to the current specification. Any of the produced solutions are formally correct, meaning that any choice will remove the violation detected in the analysis phase, cover the positive and not cover the negative scenarios[1].

The choice of operational requirement to include in the specification has an impact on the overall elaboration process. For instance, pre-conditions that are too strong (e.g. a pre-condition that restricts the occurrence of events more than necessary) may constrain the new specification too much and impair the learning process in subsequent iterations. On the other hand, pre-conditions that are too weak may only 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. Providing automated guidance for this phase is discussed in Section 5.

3.5 The Cycle

The above describes the steps applied in a single iteration of the approach. The process is assumed to be repeated until all the necessary pre-conditions and trigger-conditions have been learned, which, together with the initial specification allow only those behaviors that satisfy the goals. Consider again the example. Running the analysis again on the model of the extended specification results in the following violation trace.

```
Trace to property violation in
SafetyInjectionWhenReactorOnAndLowPressure:
   tick                 PressureBelowLow
   raiseAboveLow
   raiseAbovePermit
   startUpReactor
   tick
   lowerBelowPermit
   lowerBelowLow        PressureBelowLow
   stabilizeReactor     ReactorOn && PressureBelowLow
   tick                 ReactorOn && PressureBelowLow
   raiseAboveLow        ReactorOn
   tick                 ReactorOn
Analysed in:  7ms
```

This counterexample is the basis for the positive and negative scenarios depicted in Figures 3 and 4, which in turn lead to the learning two alternative trigger-conditions for *startSignal*.

The termination of this cycle is based upon the invariant property that *from one iteration to another, the number of violation traces for the violated goal progressively reduces*. This means that assuming that two iterations are needed to satisfy the goal model G, the traces in the LTS model of $Spec_i$, that violate the goals G *strictly includes* the set of traces in the LTS model of $Spec_{i+1}$ that violate the goals. This is because the addition of a pre-condition and/or trigger-condition to the current specification reduces the number of violation traces, provided that the initial set of violation traces accepted by the LTS model generated from $Spec$ is *finite* and no further goals are added to G. The application of our approach to two case studies, the Mine Pump System [12] and Injection System [4], has so far successfully confirmed the termination of the cycle and its convergence to the computation of a complete set of requirements that are complete with respect to the given goal model.

4 Case Study

This section reports on a case study we conducted to validate our approach. We report here in detail on a Mine Pump Controller case study taken from [12]. , and briefly discuss our experience of applying the approach to the London Ambulance System case study defined in [9].

For each of the systems studied, we had an informal description of the system-to-be, a linear temporal logic representation of its high level goals and a formal operationalisation of the goals, that is, a set of operational requirements that is complete with respect to the goals. It is important to note that all these elements, informal description, goals, and operational requirements, were produced by a third-party. The case studies consisted

[1]In case the engineer provides scenarios that are inconsistent with the given specification, the learning phase will fail to find requirements that are consistent with the given specification and scenario, and return a warning message

in starting from the high-level goals and applying the iterative method described in this paper. Human interventions required by the approach were performed by one of the authors based on her understanding of the informal description of the system and the high-level goals. Having completed all iterations, the operational requirements learned were compared to the ones provided. In all cases, we were able to learn the provided operational requirements, however, in some cases we were also able to identify alternative operationalisations of the high-level goals.

In addition to the LTSA and the XHAIL systems, the case studies were conducted using a number of automated translation tools that support sound conversion of the various formalisms used in our approach. Note that the case studies were executed on a standard desktop computer (Core2Duo, 1 GB RAM) and that each iteration within a single case studies required less than 11 seconds of computation time.

4.1 Mine Pump Controller

The controller of a mine pump [12] is expected to monitor and control water levels in a mine, to prevent water overflow. It is composed of a pump for pumping mine-water up to the surface and sensors for monitoring the water levels and methane percentage. The pump must be activated once the water has reached pre-set high water level and deactivated once it reaches low water level. Moreover, the pump must be switched off if the percentage of methane in the mine exceeds a certain critical limit. The goals are expressed in asynchronous FLTL as:

$$G[PumpOnWhenHighWaterANDNoMethane] = \Box(tick \rightarrow ((HighWater \bigcirc \neg Methane) \rightarrow \bigcirc(\neg tick W(tick \wedge PumpOn)))) \quad (1)$$

$$G[PumpOffWhenLowWater] = \Box(tick \rightarrow (LowWater \rightarrow \bigcirc(\neg tick W(tick \wedge \neg PumpOn)))) \quad (2)$$

$$G[PumpOffWhenMethane] = \Box(tick \rightarrow (Methane \rightarrow \bigcirc(\neg tick W(tick \wedge \neg PumpOn)))) \quad (3)$$

$$G[AlarmWhenMethane] = \Box(tick \rightarrow (Methane \rightarrow \bigcirc(\neg tick W(tick \wedge Alarm)))) \quad (4)$$

The fluents appearing in the goal formalisation are defined as follows:

```
fluent PumpOn=< {switchPumpOn},  {switchPumpOff} >
fluent HighWater=< {raiseWaterLevel[High-1]},
                          {lowerWaterLevel[High]} >
fluent LowWater=< {lowerWaterLevel[Low+1]},
                          {raiseWaterLevel[Low]} >
                                        initially True
fluent Methane=< {signalMethane},
                          {signalNoMethane } >
fluent Alarm =< {raiseAlarm}, {stopAlarm} >
```

where $High$ and Low are preset thresholds equal to 10 and 3 respectively.

We start the case study assuming no pre- and trigger-conditions are known. What follows is a summary of some of the iterations resulting from the application of our approach.

Iteration 1

Analysis: The analysis phase resulted in the following violation trace:

```
Trace to property violation in PumpOffWhenLowWater:
tick                            LowWater
switchPumpOn                    LowWater && PumpOn
tick                            LowWater && PumpOn
Analysed in:  0ms
```

This trace exemplifies a possible system behaviour which violates the goal *PumpOffWhenLowWater* since the goal requires the pump to be off at the last tick.

Scenario Elaboration: The *switchPumpOn* is identified as the undesirable event. The negative scenario ⟨*tick, switchPumpOn*⟩ is elaborated. A possible positive scenario is ⟨*tick, raiseWaterLevel.0, tick, raiseWaterLevel.1, tick,..., raiseWaterLevel.10, tick switchPumpOn*⟩.

Learning: The learning phase produces the two alternative pre-conditions for *switchPumpOn*:

Pre(switchPumpOn) = ¬LowWater

$$\Box(tick \rightarrow ((LowWater) \rightarrow \bigcirc\neg switchPumpOn\ U\ tick)) \quad (5)$$

Pre(switchPumpOn) = HighWater

$$\Box(tick \rightarrow ((\neg HighWater)) \rightarrow \bigcirc\neg switchPumpOn\ U\ tick)) \quad (6)$$

Selection: Pre-condition (6) is added to *Spec*.

Iteration 2

The second iteration starts from the extended specification $Spec_2 = Spec \cup (6)$.

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

```
Trace to property violation in PumpOffWhenLowWater:
tick                            LowWater
raiseWaterLevel.0               LowWater
tick                            LowWater
raiseWaterLevel.1               LowWater
tick                            LowWater
.
.
.
raiseWaterLevel.10
tick
switchPumpOn                    PumpOn
tick                            PumpOn
```

```
lowerWaterLevel.11          PumpOn
.
.
.
lowerWaterLevel.4           LowWater && PumpOn
tick                        LowWater && PumpOn
tick                        LowWater && PumpOn
Analysed in:  0ms
```

This trace exemplifies another system behaviour which violates the goal *PumpOffWhenLowWater*.

Scenario Elaboration: The last tick is identified as the undesirable event. The negative scenario hence becomes ⟨*tick, raiseWaterLevel.0, tick, raiseWaterLevel.1, tick,..., lowerWaterLevel.4, tick, tick*⟩. A possible positive scenario is ⟨*tick, raiseWaterLevel.0, tick, raiseWaterLevel.1, tick,..., raiseWaterLevel.4, tick, switchPumpOff, tick*⟩.

Learning: The learning phase in this case results in the following alternative solutions:

`Trig(switchPumpOff) = LowWater`

$$\Box(tick \rightarrow ((LowWater \wedge PumpOn) \rightarrow \bigcirc\neg tick\ U\ switchPumpOff)) \tag{7}$$

and `Trig(switchPumpOff) =` ¬ `HighWater`

$$\Box(tick \rightarrow ((\neg HighWater \wedge PumpOn) \rightarrow \bigcirc\neg tick\ U\ switch.PumpOff)) \tag{8}$$

Selection: The trigger-condition (7) is selected and added to the specification after which the approach is applied again.

Iterations 3 to 7

We omit these iterations due to lack of space. The operational requirements learned in these iterations are the following:

$$\begin{array}{l}
\texttt{Trig}_2\texttt{(switchPumpOff) = Methane} \\
\Box(tick \rightarrow ((\neg Methane \wedge \neg PumpOn) \rightarrow \bigcirc\neg tick\ U\ switchPumpOff)) \\
\texttt{Pre(switchPumpOn) = }\neg\texttt{Methane} \\
\Box(tick \rightarrow ((Methane) \rightarrow \bigcirc\neg switchPumpOn\ U\ tick)) \\
\texttt{Trig(switchPumpOn) = HighWater}\wedge\neg\texttt{Methane} \\
\Box(tick \rightarrow ((HighWater \wedge \neg Methane) \rightarrow \bigcirc\neg tick\ U\ switchPumpOn)) \\
\texttt{Trig(raiseAlarm) = Methane} \\
\Box(tick \rightarrow ((Methane \wedge \neg Alarm) \rightarrow \bigcirc\neg tick\ U\ raiseAlarm)) \\
\texttt{Pre(stopAlarm) = }\neg\texttt{Methane} \\
\Box(tick \rightarrow ((Methane) \rightarrow \bigcirc\neg stopAlarm\ U\ tick))
\end{array} \tag{9}$$

Iteration 8

This iteration starts from the extended specification $Spec_8 = Spec_2 \cup$ (9).

Analysis: The following violation trace is produced

```
Trace to property violation in
PumpOnWhenHighWaterANDNoMethane:
tick                        LowWater
raiseWaterLevel.0           LowWater
tick                        LowWater
.
.
.
tick
raiseWaterLevel.10          HighWater
tick                        HighWater
switchPumpOn                HighWater && PumpOn
tick                        HighWater && PumpOn
switchPumpOff               HighWater
tick                        HighWater
Analysed in:  16ms
```

This trace indicates a violation to the goal *PumpOnWhenHighWaterANDNoMethane*.

Scenario Elaboration: *switchPumpOff* is indicated as the undesirable one. The negative scenario here becomes ⟨ *tick, raiseWaterLevel.0, tick, ..., tick, switchPumpOn, tick, switchPumpOff*⟩. Two possible positive scenarios are ⟨ *tick, raiseWaterLevel.0, tick, ..., tick, switchPumpOn, tick, signalMethane,tick, switchPumpOff*⟩ as well as ⟨ *tick, raiseWaterLevel.0, tick, ..., tick, switchPumpOn, tick, lowerWaterLevel.10, tick,...., lowerWaterLevel.4, tick, switchPumpOff*⟩

Learning: The learning phase in this case results in the following alternative solutions:

$\texttt{Pre}_2\texttt{(switchPumpOff) = LowWater} \vee \texttt{Methane}$

$$\Box(tick \rightarrow ((\neg LowWater \wedge \neg Methane) \rightarrow \bigcirc(\neg\ switchPumpOff\ U\ tick))) \tag{10}$$

and $\texttt{Pre}_2\texttt{(switchPumpOff) =}\ \neg\ \texttt{HighWater} \vee \texttt{Methane}$

$$\Box(tick \rightarrow ((HighWater \wedge \neg Methane) \rightarrow \bigcirc(\neg\ switchPumpOff\ U\ tick))) \tag{11}$$

Selection: The precondition (10) is added to $Spec_8$. Running the analysis again on $Spec_9 = Spec_8 \cup$ (10) gives the following output:

```
No deadlocks/errors
Analysed in:  125ms
```

which indicates that a complete set of requirements has been computed.

Due to space restrictions, we have shown only a subset of the alternative proposals that the learning phase computes within a single iteration. It is important to note that not only did the learning phase propose all the operational requirements that could be generated manually in [16] but also, in some cases together with the operational requirements chosen in [16], the tool proposed stronger requirements that raised interesting issues regarding the behaviour of the controller. For instance, in addition to proposing $\neg LowWater$ as the precondition for $switchPumpOn$, the learning tool proposed the pre-condition $HighWater$ which is perfectly sensible: the goals under specify the behaviour of the pump when water is neither high or low.

4.2 London Ambulance System

This case study is based on a system for dispatching ambulances to incidents in London. We applied our approach to a simplified version of the case study originally described in [9] considering the goal "*If an incident form has been encoded an available ambulance should be allocated with in allocation_delay time units*". The case study showed that our approach is capable of learning requirements for complex goals such as the one which need to be satisfied within a bounded time period. In addition, the case study demonstrates certain flexibility of the approach to learn operational requirements of different syntactic structure. For instance, the approach computed a trigger-condition *allocateAmbulance* of the form: "*the incident form has been encoded and the ambulance has not been allocated within allocation_delay time units*" formally expressed as ($\mathit{Encoded}\ S_{\leq \mathit{allocation\_delay}} \neg\ \mathit{Allocated}$).

5 Discussion and Related Work

The approach described in this paper generalises that proposed in [2], which also integrates model checking and inductive learning. In [2], sets of pre-conditions are learned to guarantee a time progress property. The work described herein extends this approach, as it supports learning trigger-conditions and does not restrict goals to a specific kind of liveness property.

In Section 3.5, we have presented an argument concerning the termination of the cycle. Successful termination clearly holds in the case when the engineer is successful in elaborating those negative and positive scenario that directly contribute to learning complete and correct set of pre- and trigger-conditions with respect to the goals. Common to any technique that involves user intervention, backtracking maybe required in situations where the engineer fails to elaborate the scenarios correctly.

The work which is most related to ours is [16]. In [16], a pattern-based approach is proposed. From a temporal logic representation of the goals, it is possible to derive operational requirements by applying the appropriate pattern to terminal goals [8]. For instance given the goals $G1$ and $G2$ it possible to derive the trigger-conditions for *startSignal* and preconditions for *stopSignal* by applying the immediate achieve pattern to the goals[16]. However, because our approach relies on analyses of behaviour traces rather than temporal formulae, the processes such as goal refinement, obstacle analysis and operationalisation [8] can be bypassed provided the engineer gives the appropriate scenarios.

Other related work on the use of machine learning techniques as automated support in various aspects of requirements engineering include [13, 5, 3, 21]. Most relevant to our work is the approach in [13], where goal assertions, expressed in LTL, are iteratively inferred from user-given scenarios. Each iteration takes a single user scenario and generates a set of goal expressions that satisfy the scenario under certain considerations. These are added to the existing goal model, and the extended model checked for obstacles. The inductive learning process adopted in [13] does not take into account the current goal model as background whereas one of the main advantages in using the XHAIL system is the use of the current specification. Consequently the goals learned in [13] may be inconsistent, obstacles may be detected and a process of obstacle resolution may be needed before a subsequent iteration is performed. In our approach, however, the operational requirements that are learned during one iteration are guaranteed to be consistent with the current partial specification.

The work in [5, 6] uses grammar inference to support automated behavior model synthesis from user-defined scenarios. This involves generating an LTS that satisfies all given positive scenarios and none of the negative ones. Starting from this initial LTS model, the inference procedure attempts to generalise 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 the number of scenarios to be classified, a goal specification is assumed and only paths satisfying the goals are presented 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, and it *refines* it adding pre- and trigger-conditions on system events that eliminate undesirable behaviors. In contrast to [5, 6], this allows the addition of information such as goals and even scenarios that extend the alphabet of the description, during the process without requiring to start the process from scratch.

6 Conclusion

This paper proposes a tool-supported framework for requirements operationalisation from high-level goals that integrates model checking and inductive learning. The process supports an important activity in goal-based requirements engineering and complements existing approaches, by learning requirements that cannot be derived from operationalisation patterns.

The approach complements standard model checking techniques by providing automated refinement via inductive learning. We believe the combination of model-checking for automated counterexample generation and inductive learning for automated generation of declarative expressions can be applied to a number of software engineering activities. In particular, an interesting extension of this approach would be to, support a component-based view of the software-to-be. This would mean learning operational requirements for individual components by taking into account the underlying architecture which may also increase the approach's scalability to large systems.

Acknowledgement The work reported herein was partially supported by EP/F023294, CONICET, ERC 204853/PBM, UBACyT X021, and ANPCyT PICT 32440 and King Saud University.

References

[1] D. Alrajeh, O. Ray, A. Russo, and S. Uchitel. Extracting requirements from scenarios with ILP. In *Proc. of 16th ILP Conf.*, pages 63–77, 2006.

[2] D. Alrajeh, A. Russo, and S. Uchitel. Deriving non-zeno behavior models from goal models using ILP. In *Proc. of 10th FASE Conf.*, pages 1–15, 2008.

[3] E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Proc. of CAV*, 2002.

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

[5] C. Damas, P. Dupont B. Lambeau, and A. van Lamsweerde. Generating annotated behavior models from end-user scenarios. *IEEE TSE Journal, Special Issue on Interaction and State-based Modeling*, 31(12):1056–1073, 2005.

[6] C. Damas, B. Lambeau, and A. van Lamsweerde. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *Proc. of FSE Symp*, 2006.

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

[8] R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. *SIGSOFT Softw. Eng. Notes*, 21(6):179–190, 1996.

[9] A. Finkelstein. The london ambulance system case study. In *Proc. of 8th Intl. Workshop on Software Specification and Design*, pages 5–19, 196.

[10] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *Proc. of ACM ESEC/FSE-11*, pages 257–266, 2003.

[11] ITU. *Message Sequence Charts.* International Telecommunications Union, Telecommunication Standardisation Sector, 1996.

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

[13] A. Van Lamsweerde and L. Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE TSE Journal*, 24(12):1089–1114, 1998.

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

[15] E. Letier, J. Kramer, J. Magee, and S. Uchitel. Deriving event-based transition systems from goal-oriented requirements models. *ASE Journal*, 15:175–206, 2008.

[16] E. Letier and A. Van Lamsweerde. Deriving operational software specifications from system goals. In *Proc. of 10th ACM FSE Symp.*, pages 119–128, 2002.

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

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

[19] J. Mylopoulos, L. Chung, and B. A. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE TSE*, 18:483–497, 1992.

[20] O. Ray. Using abduction for induction of normal logic programs. In *Proc. of 2nd Workshop on AIAI and Scientific Modelling*, pages 28–31, 2006.

[21] A. Russo, R. Miller, B. Nuseibeh, and J. Kramer. An abductive approach for analysing event-based requirements specifications. In *Proc. of 18th ICLP*, volume 2401 of *LNCS*, pages 22–37, 2002.