Project no.:	**PIRSES-GA-2011-295261**
Project full title:	**Mobility between Europe and Argentina applying Logics to Systems**
Project Acronym:	**MEALS**
Deliverable no.:	**5.2 / 2**
Title of Deliverable:	**Automated Goal Operationalisation Based on Interpolation and SAT Solving**

Contractual Date of Delivery to the CEC:	**30-Sep-2014**
Actual Date of Delivery to the CEC:	**1-Sep-2014**
Organisation name of lead contractor for this deliverable:	**IMP**
Author(s):	**R. Degiovanni, N. Aguirre, D. Alrajeh, S. Uchitel**
Participants(s):	**IMP, UBA, UNR**
Work package contributing to the deliverable:	**WP5**
Nature:	**R**
Dissemination Level:	**Public**
Total number of pages:	**26**
Start date of project:	1 Oct. 2011 **Duration:** 48 month

Abstract:

Goal oriented methods have been successfully employed for eliciting and elaborating software requirements. When goals are assigned to an agent, they have to be *operationalised*: the agent's operations have to be refined, by equipping them with appropriate enabling and triggering conditions, so that the goals are fulfilled. Goal operationalisation generally demands a significant effort of the engineer. Although there exist approaches that tackle this problem, they are either informal or at most semi automated, requiring the engineer to assist in the process. We present an approach for goal operationalisation that *automatically* computes required preconditions and required triggering conditions for operations, so that the resulting operations establish the goals. The process is iterative, is able to deal with safety goals and particular kinds of liveness goals, and is based on the use of interpolation and SAT solving.

Note:

This deliverable is based on material that has been published in ICSE'14.

This project has received funding from the European Union Seventh Framework Programme (FP7 2007-2013) under Grant Agreement Nr. 295261.

Contents

1 Introduction

Goal oriented methods (e.g., KAOS [14] and I* [30]) have been developed and successfully applied to the problem of eliciting and elaborating software requirements. Such methods typically demand the *refinement* of high level goals that require *agent* cooperation to be achieved, into goals that can be realised by individual agents [15]. A goal assigned to an agent must be *operationalised*, i.e. mapped into operations provided and executed by agents [17].

The problem of goal operationalisation has been studied by various researchers. In the work of Letier and van Lamsweerde [17], a pattern based technique for *deriving* operational requirements from system goals is proposed. This technique provides some templates to derive, from goals expressed in linear-time temporal logic (LTL) with certain syntactic restrictions, a set of required pre/triggering conditions for operations, such that these entail the goals. More recently, Alrajeh et al. proposed an approach for semi-automa-tically learning operational requirements from a set of goals [2]. This approach uses model checking for verifying that a given set of operational requirements satisfies the goals. If verification fails, the user examines the counterexample generated by the model checker, identifies a wrongly executed operation, and provides positive scenarios illustrating "good" occurrences of this operation. These scenarios are then used by an inductive learning engine to automatically compute new required pre/triggering conditions for the selected operation. The obtained operational requirements ensure that the counterexample is avoided and the behaviour described by the positive scenarios is preserved. The approach of Alrajeh et al. is semi-automated since it requires engineers' intervention for providing positive scenarios.

In this work, we present an approach for goal operationalisation, that *automatically* computes required pre/triggering conditions for operations, in order to fulfil a set of goals. Moreover, this approach does not depend on user provided scenarios and their characteristics, e.g., "richness" and correctness, as is the case with [2]. The refinement process is based on the use of interpolation and SAT solving. As previous approaches [17, 2], our technique applies to safety and time progress goals. Moreover, we are also able to deal with a wide range of liveness goals, namely, those captured by the reactivity pattern [23]. Our approach starts with a model checking phase, for verifying whether the operational requirements specification satisfies the goals or not. If the verification is successful, then the operational specification needs no refinement. If the model checker produces a counterexample, an interpolant from the counterexample and the violated goal is computed, which is exploited to strengthen or weaken required preconditions and required triggering conditions, respectively, to remove the counterexample. The approach performs various logical checks for ensuring that the refined required conditions are consistent (in the sense of [17]) with the current operational specification. These checks are performed using SAT solving.

An interpolant for formulas A and B whose conjunction is inconsistent, is a formula that is implied by the first, is inconsistent with the second, and is expressed in the language common to A and B. Interpolation has been widely used in abstraction based verification [28], most notably for automatically refining abstract models of systems using corresponding concrete models and abstract spurious counterexamples (violations of analysed properties). In this work, we propose to use interpolation for a different, but related, purpose. Essentially, an interpolant for a formula capturing a trace and a violated goal is a property of the trace expressed in the language that the trace and the goal have in common. This interpolant can be seen as an *explanation* of why the trace violates the goal. As we will show, interpolants can be used to refine required conditions, in the process of goal operationalisation.

The remainder of this document is organised as follows: Section 2 introduces preliminary concepts. Section 3 presents a running example, used to present our approach. Section 4 describes our approach in detail, for the operationalisation of safety and time progress goals. Section 5 extends the technique to deal with liveness goals. Section 6 evaluates the approach, and compares with a closely related one, on some case studies. Finally, we discuss related work in Section 7, and present our conclusions in Section 8.

2 Background

2.1 Labelled Transition Systems, FLTL and Model Checking

Labelled Transition Systems (LTS) are typically used to model the behaviour of interacting components [22]. Formally, an LTS P is a quadruple $\langle Q, A, \delta, q_0 \rangle$, where Q is a finite set of states, A is the *alphabet* of P, $\delta \subseteq Q \times A \cup \{\tau\} \times Q$ is a transition relation, and $q_0 \in Q$ is the initial state. The semantics of an LTS P can be defined in terms of its *executions*, i.e., the set of event sequences that P can perform, starting in the initial state and following P's transitions.

Fluent Linear-time Temporal Logic (FLTL) [12] is a variant of LTL [23, 24], well suited for describing properties of event-based discrete systems (e.g., LTSs) [22]. FLTL extends LTL by incorporating the possibility of describing abstract states called *fluents*, characterised by events of the system. Formally, $Fl = \langle I_{Fl}, T_{Fl}, B \rangle$ defines a fluent Fl, where $I_{Fl}, T_{Fl} \subseteq A$ and $I_{Fl} \cap T_{Fl} = \emptyset$, and $B \in \{true, false\}$ indicates the initial value of Fl. When any event in I_{Fl} occurs, the fluent starts to be true, and it becomes false again when any event in T_{Fl} occurs. If the term B is omitted then Fl is initially *false*.

A FLTL formula is an LTL formula where propositions are fluents. Given a set Φ of fluents, a well-formed FLTL formula is defined inductively using the standard boolean operators and the temporal operators $\bigcirc$ (next) and $\mathcal{U}$ (strong until), in the following way: *(i)* every $fl \in \Phi$ is a formula, and *(ii)* if ϕ and ψ are formulas, then so are $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$, $\bigcirc\phi$, $\phi \, \mathcal{U} \, \psi$. We consider the usual definition for the operators $\Box$ (always), $\Diamond$ (eventually) and $\mathcal{W}$ (weak until) in terms of next and strong until, and boolean connectives.

Model checking [6] provides an automated method for determining whether or not a property described by a (typically temporal) formula holds in the system's state graph. In our case, we will use Labelled Transition System Analyser (LTSA), a verification tool for concurrent systems models. A system in LTSA is modelled as a set of interacting finite state machines (described by FSP processes [22]). LTSA directly supports FLTL verification by model checking.

2.2 Interpolation

Given two formulas A and B, such that $A \wedge B$ is inconsistent, an *interpolant* for A and B is a formula I with the following properties: *(i)* A implies I, *(ii)* $I \wedge B$ is unsatisfiable, and *(iii)* I is in $\mathcal{L}(A) \cap \mathcal{L}(B)$, i.e., the language that A and B have in common. For instance, consider $A \equiv p \wedge q$ and $B \equiv \neg q \wedge r$. Then, an interpolant for A and B is $I \equiv q$.

In this work, we use the MathSAT [4] constraint solver. This tool is behind various tasks, such as finding operations that lead to certain states, querying about operation enabledness, etc. Most notably, MathSAT is able to compute interpolants, which is a crucial mechanism underlying our approach. Although MathSAT is an SMT solver, we do not make use for any of our case studies of the theories MathSAT is equipped with. That is, we are using the tool only as a SAT solver (with interpolation computation capabilities).

3 Operational Requirements for Goal Models

Goals are properties that the system should achieve. A *goal model* is a decomposition of goals through AND/OR refinement links, which essentially capture how a goal can be achieved in terms of simpler ones. Goal refinement (the decomposition of goals in subgoals) ends when each subgoal at the bottom of the decomposition can be assigned to a single agent. Agents perform operations, whose combined behaviours must fulfil the goals. *Operations* characterise state transitions in the system. Each operation is specified by its signature (input and output variables), the *domain* pre- and post-conditions (*DomPre* and *DomPost*, respectively), and the *required* conditions. There are three different types of required conditions. A required precondition (*ReqPre*) captures *permission*: under this condition the operation *may* be executed. A triggering condition (*ReqTrig*) captures *obligation*: under this condition the operation *must* be executed. Finally, the required postcondition (*ReqPost*) captures additional effects of executing the operation. A goal is correctly operationalised by a set of operations (*operation model*), if satisfying all required conditions in the set guarantees the satisfaction of the goal [17].

It is possible to systematically obtain a behavioural model from an operation model [20]. Requirements can be expressed using FLTL, by combining the "always" and "next" operators, to capture required conditions, and introducing further FLTL constraints to cope with the synchronous semantics of KAOS. Then, an LTS that represents the system behaviour can be automatically constructed. Using this characterisation of the operation model, a model checker (e.g., LTSA) can be employed to check if the system satisfies the goals, if these are expressed as FLTL assertions.

Let us briefly describe an operation model for a *Mine Pump Controller*, introduced in [13]. We use this model as a running example throughout this document, and in particular in this section as a motivating example. The Mine Pump system controls an alarm and a water pump in a mine, depending on different levels of methane in the environment, and water in the mine. The system monitors three environmental variables: `Methane` (indicating whether methane is present in the environment), and `LowWater` and `HighWater` (indicating the level of water in the mine is low or high, respectively), and two software variables that are controlled by the system: `PumpOn` and Alarm, indicating whether the water pump and the alarm are on or not, respectively.

The above mentioned monitored variables are modified by environment events: belowLow, aboveLow, aboveHigh and belowHigh modify the water level in the mine, while signalMethane and signalNoMethane change accordingly the presence of methane in the environment. The system may actively control the state of the pump and the alarm with the events switchPumpOn, switchPumpOff, raiseAlarm and stopAlarm. Initially, the operations are enabled as long as the domain precondition is true (i.e., the required precondition is true), and are not obliged to be executed in any state (i.e., the required triggering condition is false).

```
Operation switchPumpOn    Operation switchPumpOff
 DomPre  ¬PumpOn            DomPre  PumpOn
 DomPost PumpOn            DomPost ¬PumpOn

Operation switchAlarmOn   Operation switchAlarmOff
  DomPre  ¬Alarm            DomPre  Alarm
  DomPost Alarm            DomPost ¬Alarm
```

Consider a goal PumpOffWhenMethane stating that *"when methane is present in the environment, the pump must be switched off"*. In order to check whether our operation model guarantees that this goal is achieved, the approach presented in [20] may be followed. The specification is mapped into an event-based transition system, and a model checker is used to check that the goals are satisfied. We express goals as FLTL formulas, since we will use LTSA [22]. The above goal is expressed in FLTL as follows:

$$\Box\ (\mathrm{tick} \rightarrow (\mathrm{Methane} \rightarrow \bigcirc(\neg\mathrm{tick}\ \mathcal{W}\ (\mathrm{tick} \wedge \neg\mathrm{PumpOn}))))$$

The tick event is a means for capturing KAOS' *synchronous* interpretation of an operation model under an *asynchronous* interpretation which is used in LTSA [20]. The above formula states that, if Methane is true when tick occurs, then before the next tick the pump must be switched off. Having the operation model captured as an LTS and goals as FLTL formulas, it is possible to verify whether the specification meets the goals or not. In our case, the goals can be violated, and the LTSA model checker produces the following counterexample:

```
Trace to property violation in PumpOffWhenMethane:
  tick  (s0)
  signalMethane    Methane
  tick             Methane  (s1)
  switchPumpOn     Methane ∧ PumpOn
  tick             Methane ∧ PumpOn  (s2)
```

By examining the above counterexample, one can realise that state s2 does not satisfy the goal PumpOffWhenMethane. The problem has to do with the operation switchPumpOn being able to occur when there is methane in the environment (notice that the required precondition allows for this).

3.1 Using Interpolation for Refinement

The above counterexample can be expressed as a formula A, where system states are represented by variables (one per fluent in the specification), and these variables are replicated 3 times, for the 3 different states of the trace (i.e. `s0`, `s1` and `s2`), relating each one with the next according to what the corresponding event indicates. Suppose that the goal can be expressed as a *state property* B (e.g., this is straightforward if the property is a safety property), referring to the last state of the trace. Since the trace is a violation of B, clearly $A \wedge B$ is unsatisfiable. We can then try and use interpolation, to see what the interpolant provides. In this case, the interpolant is: `Methane` ∧ `PumpOn`. This interpolant indicates that the problem is that in the last state there is methane in the environment and the pump is on, which causes a violation in the goal. Let us start going back from the state previous to the last `tick`, trying to find the first place where the violation might be avoided. We can compute the weakest precondition (WP) of the interpolant with respect to the last action, i.e., `switchPumpOn`, obtaining:
WP(`switchPumpOn`, `Methane` ∧ `PumpOn`) = `Methane`.

Notice that this last operation has the ability to *change* the truth value of the interpolant. Thus, by preventing its execution, we can get rid of the previous counterexample. We have to check, however, whether this action is obliged or not to be taken, when `Methane` is present. This can be checked using some decision procedure (e.g., SAT solving), to decide the formula `Methane` ⇒ *ReqTrig*(`switchPumpOn`). If it is the case, then the action cannot be prevented, and we have to continue searching backwards in the counterexample, for another way of removing the counterexample. If it is not the case, we can get rid of the counterexample by adding ¬`Methane` to `switchPumpOn`'s required precondition.

4 Refining Operational Requirements

Let us describe our approach for refining operational requirements in more detail. For the sake of simplicity, we assume that goals are *safety properties*. Later on we extend the approach to deal with *Time Progress* (TP) and *reactivity properties* [23]. Given a set $G = \{G_1, \ldots, G_n\}$ of goals, and an operational requirements specification *Spec* that may not satisfy G, we propose an iterative approach that refines *Spec* by constructing new required pre/triggering conditions for operations, such that the resulting refined specification satisfies G. This approach can only refine *Spec* by weakening and strengthening required preconditions and required triggering conditions of *Spec*'s operations, respectively.

The approach consists of two phases, that are depicted in Fig. 1. The *Model Checking* phase is concerned with verifying whether *Spec* satisfies the goals G or not. If the verification is successful, then the requirements specification needs no further refinement. If, on the other hand, the goals are not satisfied by *Spec*, then the model checker detects a violation to at least one of the goals G_i, and produces a counterexample as a witness of the violation. The *Refinement* phase automatically refines the operational requirements so as to prevent the detected violation from occurring. To achieve this, the approach uses *interpolation* and starts computing an inter-

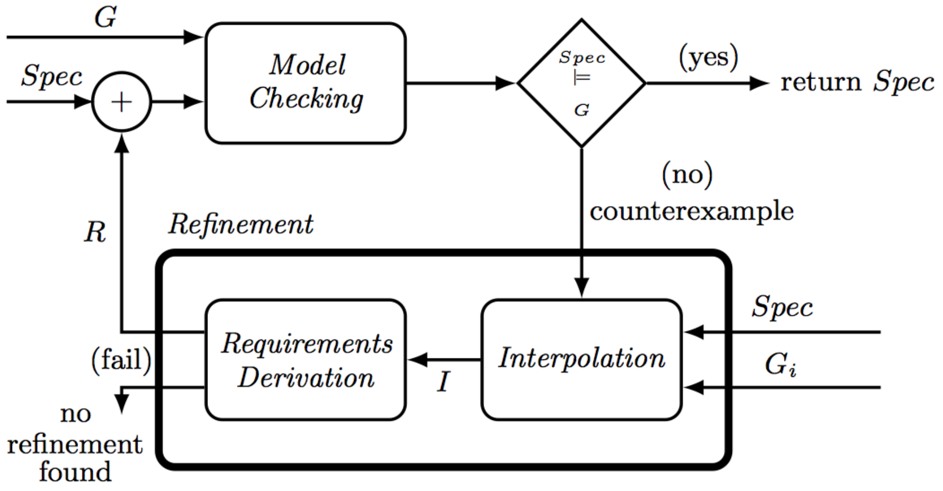

Figure 1: Overview of the iterative approach.

polant from the counterexample trace and the violated goal G_i. This interpolant is exploited, using *weakest precondition* computations, to identify operations whose guarding conditions may be altered to get rid of the obtained counterexample. More precisely, the obtained interpolant, with the aid of weakest precondition, produces the formulas to be used to strengthen or weaken required pre/triggering conditions, respectively, to remove counterexamples.

The refined specification, that incorporates new "more precise" required conditions for some operations, removes from its behaviours the violation found before. The process is repeated until no violation is detected, obtaining a valid operationalisation for G, or until we reach a point in which a violation cannot be eliminated, meaning that the found violation cannot be removed by refining required conditions of the current operational requirements specification.

4.1 Model Checking Phase

The LTSA model checker has been used for analysing operational requirements specifications against goal models expressed in FLTL [20]. The state description of the system is given with fluent definitions (cf. Section 2). We follow the approach presented in [20] for capturing KAOS operation models in LTSA, which makes use of a `tick` event to explicitly represent the start and end of time intervals in asynchronous FLTL. As an example, consider the following goal for a model in which ticks are incorporated:

```
assert PumpOnWhenHighWaterANDNoMethane =
□(tick→((HighWater∧¬Methane)→ ○(¬tick 𝒲 (tick ∧ PumpOn))))
```

This formula expresses that, if at some time-point the level of water is high and there is no methane, then the pump must be switched on at the next time-point. We can use LTSA for checking if the given *Mine Pump* specification satisfies the goals. Initially we consider the weakest required precondition (true) and the strongest required triggering condition (false) for every operation. LTSA returns the following counterexample for the goal PumpOnWhenHighWaterANDNoMethane:

```
Trace to property violation
  in PumpOnWhenHighWaterANDNoMethane:
  tick  (s0)
  aboveLow
  tick  (s1)
  aboveHigh
  tick        HighWater (s2)
  tick        HighWater (s3)
```

Because of the way the property is expressed (in relation to the tick event), the important states are those immediately following tick events. This counterexample corresponds to a case in which (s2) starts with the water pressure being high, no methane in the environment and the pump off, and by the next time-point the situation has not changed and the pump remains off.

4.2 Refinement Phase

The counterexamples generated by the model checker are used to automatically and iteratively compute refinements during the goal operationalisation process, which are guaranteed to remove the detected counterexamples.

Suppose we obtain a counterexample trace T, violating a particular goal G_i. If we build a formula F_T capturing trace T, and a formula F_{G_i} capturing the fact that G_i holds at the last state, clearly $F_T \wedge F_{G_i}$ is unsatisfiable. We can then produce an interpolant from these formulas, i.e., a statement I, in the intersection of the languages of the trace and the goal, such that $F_T \Rightarrow I$ and $I \wedge F_{G_i}$ is unsatisfiable. Intuitively, the interpolant I represents a weaker "counterexample" than T, a condition reachable from the initial state, which leads to the violation of the goal. So, as long as we do not get rid of the property I, we will not be able to stop violating the goal G_i. However, solely by removing I, we do not guarantee the satisfaction of G_i, but not removing it guarantees its violation. Notice that calculating the interpolant I is, in some sense, a form of generalisation.

A counterexample may be removed either by prohibiting the occurrence of an operation from certain states (i.e., strengthening the operation's required precondition) or by forcing an operation to occur in certain states (i.e., weakening its required triggering condition). The approach is concerned with automatically detecting which of the above cases is necessary, and using the interpolant to produce a change in the corresponding condition. In the process of refining required conditions of an operation model using interpolants, weakest preconditions play an important role.

4.2.1 Strengthening Required Preconditions

By processing the counterexample trace backwards from the last tick, we can compute the weakest precondition (WP) of the interpolant with respect to the last operation, trying to find the first place where the violation might be avoided. Let I be the interpolant and $T = a_1; a_2; \ldots; a_k$ the counterexample trace. We can have two situations with respect to $WP(a_k, I) = I'$, the weakest precondition of the last operation and the interpolant. If $\neg(I' \Rightarrow I)$, then the interpolant does not hold before executing a_k. Therefore, by forbidding a_k to occur when I' holds, by adding $\neg I'$ to the required precondition of a_k, we get rid of this counterexample. This can be done as long as it does not contradict a_k's required triggering conditions, (i.e., if the operation is not obliged to be executed when I' holds). If $I' \Rightarrow I$, then by preventing a_k from occurring we do not stop violating the goal, since I still holds. In this case, as well as when adding $\neg I'$ to the required precondition would contradict other conditions of a_k, we have to continue searching backwards in the trace, to try to find an operation whose occurrence causes the satisfaction of the interpolant, and which can be "removed" from the trace. The above modification to the required precondition of a_k does not guarantee that G_i cannot be violated, it only prevents its violation through a_k when I' holds. As an example of required precondition strengthening, Consider the following counterexample, violating PumpOffWhenLowWater:

```
Trace to property violation in PumpOffWhenLowWater:
  tick         LowWater (s0)
  switchPumpOn LowWater ∧ PumpOn
  tick         LowWater ∧ PumpOn (s1)
```

In this case switchPumpOn is the only non-tick operation executed, whose current required precondition and required triggering condition are true and false, respectively. The interpolant in this case is: LowWater ∧ PumpOn. Going backwards from the last tick, we compute the weakest precondition of this interpolant with respect to the last non-tick operation, switchPumpOn, obtaining LowWater. Notice that ¬(LowWater ⇒ LowWater ∧ PumpOn). Then, it is possible to falsify the interpolant by preventing switchPumpOn from occurring when LowWater. This is achieved simply by adding ¬LowWater to the required precondition of switchPumpOn, as long as this does not contradict the operation's required triggering condition. We then modify the required precondition of switchPumpOn, which now becomes ¬LowWater.

In the process just described, we have to perform various logical checks, namely checking whether a weakest precondition implies or not an interpolant and if a new conjunct of a required precondition does not contradict a required triggering condition. We perform these checks using SAT solving. In the particular case of checking whether a new identified conjunct for a required precondition contradicts or not a required triggering condition, we consider the formula:

$$ReqTrig.condition \;\wedge DomPre.condition \Rightarrow ReqPre.condition \tag{1}$$

which must always hold (it is a meta-rule of the KAOS language [15]). Notice that an operation is able to execute only when its domain precondition holds. Then, if by modifying a required precondition adding a new conjunct we violate this property, we conclude that the conjunct is contradictory with the current required triggering condition or the domain precondition.

4.2.2 Weakening Required Triggering Conditions

In the above described situations, a counterexample is removed by preventing an operation that *appears* in the trace from occurring. In other situations, the solution to remove the counterexample cannot be given in terms of preventing an operation from occurring, but instead in terms of forcing an operation that did not occur, to occur when it has to, i.e., by weakening the operations required triggering condition.

Let $T = a_1; \ldots; a_i; \texttt{tick}; a_j; \ldots; a_n$ be the counterexample trace and I the interpolant computed for this counterexample. Suppose that we cannot get rid of this counterexample by strengthening some operation in $a_j; \ldots; a_n$. Then, as each operation in $\{a_j, \ldots, a_n\}$ cannot be prevented from occurring, we will try to remove the counterexample by forcing an operation to occur. Notice that the operation to be triggered, say a_t, must meet the following two conditions:

- a_t should be able to be executed when interpolant I holds:

$$I \Rightarrow DomPre(a_t) \wedge ReqPre(a_t). \tag{2}$$

- a_t's execution must falsify the interpolant I:

$$(I \wedge \bigcirc DomPost(a_t)) \Rightarrow \bigcirc\neg I. \tag{3}$$

We evaluate every *controlled* operation a_t not occurring in $a_j; \ldots; a_n$, checking whether it meets both of the above conditions or not. If we find such an operation a_t, we refine its required triggering condition, as follows:

$$ReqTrig(a_t) = ReqTrig^{pre}(a_t) \vee I,$$

where $ReqTrig^{pre}(a_t)$ is a_t's current required triggering condition. Notice that adding the disjunct I to a_t's required triggering condition does not violate the KAOS meta-rule (1), since a_t satisfies condition (2).

If no operation satisfies conditions (2) and (3), we proceed to look deeper in the counterexample trace T, going backwards from a_i, and using $I' = WP(a_j; \ldots; a_n, I)$ instead of I.

Consider, for instance, the counterexample to the goal `PumpOnWhenHighWaterANDNoMethane`, shown in Section 4.1. This time, $\neg$`Methane`$\wedge$`HighWater`$\wedge\neg$`PumpOn` is the interpolant computed. Notice that in this case, no operation is executed between the last two tick events. We have then to check if there exists an operation that satisfies conditions (2) and (3). Operation `switchPumpOn` is executable when $\neg$`Methane` $\wedge$ `HighWater` $\wedge$ $\neg$`PumpOn`, and it falsifies this interpolant. Indeed, notice that, considering that $\neg$`LowWater` and `false` are `switchPumpOn`'s current required precondition and required triggering condition, respectively, the following formulas, corresponding to conditions (2) and (3), hold:

$$\neg\texttt{Methane} \wedge \texttt{HighWater} \wedge \neg\texttt{PumpOn} \Rightarrow \neg\texttt{PumpOn} \wedge \neg\texttt{LowWater} \tag{4}$$

$$\begin{aligned}&(\neg\texttt{Methane} \wedge \texttt{HighWater} \wedge \neg\texttt{PumpOn})\wedge\\&(\texttt{PumpOn'} \wedge \texttt{Methane'=Methane} \wedge \texttt{HighWater'=HighWater})\\&\quad\Rightarrow \neg(\neg\texttt{Methane'} \wedge \texttt{HighWater'} \wedge \neg\texttt{PumpOn'})\end{aligned} \tag{5}$$

Then, switchPumpOn's required triggering condition is refined as follows:

$$\mathit{ReqTrig}(\mathtt{switchPumpOn}) = \neg\mathtt{Methane} \land \mathtt{HighWater} \land \neg\mathtt{PumpOn}$$

As for the case of strengthening required preconditions, weakening required triggering conditions also involves logical checks. We perform these logical checks using SAT solving.

4.3 Iterative Refinement

The two phases described above correspond to a single iteration of the refinement approach. These phases are iteratively applied until no further violations are detected in the model checking phase, i.e., until a specification that satisfies the goals is reached, or until we reach a point from which, by solely refining required preconditions and required triggering conditions, the goals cannot be achieved.

Let O and G be the initial operational specification and a set of goals, respectively, consistent with respect to a set of fluents D. Let us now discuss correctness, incompleteness and termination of the approach.

Correctness and Incompleteness When the approach finishes, its outcome is $Spec = O \cup Req$, where Req is the set of required pre/triggering conditions obtained. Req is guaranteed to be a *consistent* extension to O that *correctly* operationalises the goals in G. This is formalised as follows:

$$O \cup Req \cup G \not\models_D \mathit{false} \quad O \cup Req \models_D G$$

Due to space restrictions, the proof of this claim is not reported here. The justification has to do with each refinement in the process being guaranteed to be sound, thanks to the SAT checks we perform. For instance, in the case of strengthening a required precondition, the validity of condition (1) ensures that the new required precondition does not contradict the current required triggering condition. In addition, conditions (2) and (3), in the case of weakening a required triggering condition, ensure that the KAOS meta-rule (1) is not violated.

However, the proposed approach *does not* satisfy completeness. That is, if there exists a refinement (modifications to the required conditions) that can satisfy the set of goals, then the approach might fail in the process of finding that refinement. The main reason is related to the the fact that goals can be competing, i.e., trying to fulfil one goal may prevent us from later on fulfilling another goal. Then, by removing a counterexample the approach can remove edges from the system's behaviour that later could be required to be added to remove another counterexample (notice that we cannot add edges, since postconditions are not modified, nor new states are added, since the definition of variables, what defines the state space, is not altered).

Termination As we argued during the presentation of the refinement phase, a refinement step removes the counterexample from which the refinement was constructed, in the sense that the same execution cannot be obtained. However, there might be other traces violating the same goal. Thus, termination is not straightforwardly guaranteed. Let us argue about it, for the case of safety goals.

The model checking phase constructs a labelled transition system corresponding, essentially, to formula $O \land \neg G$. If this formula has satisfying traces, these are counterexamples. First, notice that the labelled transition system $O \land \neg G$ is *finite*: it has a finite number of states, and of course a finite number of edges. We have to demonstrate that each refinement of required conditions removes *at least* one transition, since there exists the risks of adding redundant conjuncts/disjuncts, which would not remove any edge.

We have explained that, when a required precondition is added, it is because an operation a has been identified, which is executable in a pre-state satisfying I', leading to a post-state in which a formula I holds (the interpolant). Moreover, this action a and condition I' come from a counterexample, indicating that I' is a reachable condition, where a can be executed. Then, when we add $\neg I'$ to the required precondition of a we remove this particular transition.

By weakening required triggering conditions, we also remove edges. When a required triggering condition is added, it is because an operation a has been identified, which is executable in a state satisfying I, the interpolant, and whose execution falsifies the interpolant. Notice then that, by weakening the triggering condition for a, we remove transitions corresponding to `tick`. Indeed, while `tick` was executable in the state satisfying I, it is no longer executable in that same state due to the obligation of executing a instead. Thus, again the refinement produces an edge removal.

Since the number of edges corresponding to $O \land \neg G$ is finite, and we alter this transition system only by deleting edges, it is guaranteed that our refinement process terminates when it deals with safety goals.

5 Dealing with Liveness

5.1 Time Progress Property

In discrete-time systems, a common desired property is the progress of time, usually specified as $\Box\Diamond\texttt{tick}$. This property is called *Time Progress* (TP) and is a particular kind of progress property. In this section, we show how the approach can be extended to operationalise this particular progress property.

In order to detect progress violations, we take into account three assumptions: *(i) KAOS' semantic preservation:* all initiating and terminating fluent events can occur only once between ticks; *(ii) Maximum Progress (MP) assumption:* a common assumption in reactive systems that gives priority to system events over all other events including ticks; and *(iii) Safety Guaranteed:* we do not want to interfere in the satisfaction of the safety goals to operationalise the time progress property. So, the checking is performed over the system that represents the composition of all domain/required conditions and the safety goals.

Time Progress violations are traces in which at some point no tick event can be executed. Then, provided that we have a finite number of events, and due to assumption *(i)*, which prohibits

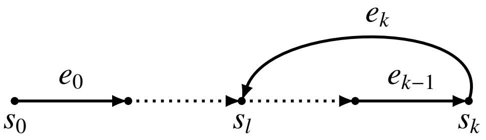

Figure 2: Counterexample for reactivity properties.

the repetition of events between ticks, these TP violations will be deadlocks. Basically, the main reason leading to these deadlocks is the way used for specifying the properties, i.e., the goals should be satisfied in the states in which tick occurs. Then, the non-progress of tick is due to the fact that its execution will violate a goal, contradicting the assumption *(iii)*.

So, our approach to remove progress violations consists of extending the obtained counterexample trace with a `tick` event at the end, and proceeding to compute an interpolant for the extended counterexample and the safety goals (that will not hold in the last state due to tick is now being executed). Intuitively, this interpolant explains the reasons why tick does not progress, and gives us a reachable property of the system that should be removed in order to contribute with the the satisfaction of the safety goals and the time progress property. To remove this counterexample, we follow the technique just presented in Section 4.

5.2 Reactivity Properties

Liveness properties have been used for reactive systems extensively, e.g., in the work of Manna and Pnueli [23]. In the context of goal oriented methods, liveness properties are typically restricted to *bounded liveness*. For instance, Letier argues in [18] that responsiveness properties for systems $\Box(trigger \rightarrow \Diamond response)$ (where *response* is controlled by the software-to-be and *trigger* is monitored by the system-to-be) should be bounded, since otherwise agents may postpone their duties indefinitely, without a finite observable violation. However, it is sometimes convenient for these kinds of properties to abstract away the bound (because it is *unknown*, or because the bound, if large, may make the system state space *explode* as the time units must be explicitly counted in the model). When properties are further away from the "machine/world" interface or encompass many interactions between the world and the machine (think of *chained responsiveness patterns*), properties such as $(\Box\Diamond As \rightarrow \Box\Diamond G)$ allow abstracting away from the bounded behaviour that the system will have to achieve the goal.

It is then, in our opinion, worthwhile to deal with liveness in goal operationalisation. Our approach considers liveness properties that match the *reactivity* pattern $(\Box\Diamond As \rightarrow \Box\Diamond G)$, where As and G are non-temporal fluent expressions. This pattern is general enough to embrace many liveness properties [23]. Moreover, it gives us information about the shape of the counterexample, which gives us the opportunity to use interpolation (see Fig. 2).

Liveness properties corresponding to the reactivity pattern have two parts: the antecedent or assumptions (As), and the consequent or goals (G). A violation of a property of this kind consists of a prefix (finite part) leading to a loop in which the antecedent is satisfied, but not the consequent. That is, at least one state $s_l \dots s_k$ in the loop satisfies the assumptions As, but no state satisfies the goal G. In order to compute an interpolant for this counterexample, we encode the reactivity goal in the following propositional formula:

$$P = (\bigvee_{i=l}^{k} As^i) \Rightarrow (\bigvee_{j=l}^{k} G^j)$$

where the expression F^i means that F holds in s_i. Let F_T be the formula that characterises the counterexample trace. Clearly, $F_T \wedge P$ is unsatisfiable. Then, we can calculate an interpolant I for these formulas. This interpolant is a weaker representation of the loop part that explains what is wrong in the loop. To remove this counterexample, we search for an operation a_t that can be executed at some point in the loop, such that its execution reaches a states that does not satisfy the interpolant (i.e., it “breaks” the loop):

$$s_i \Rightarrow DomPre(a_t) \wedge ReqPre(a_t) \quad (6)$$

$$(s_i \wedge \bigcirc DomPost(a_t)) \Rightarrow \bigcirc\neg I \quad (7)$$

If we find such an operation that satisfies both (6) and (7), then we refine a_t’s triggering condition with the conjunction of its required precondition and the negation of the goal:

$$ReqTrig(a_t) = ReqTrig^{pre}(a_t) \vee (ReqPre(a_t) \wedge \neg G)$$

Notice that we do not refine a new triggering condition based on the interpolant. Worse, we may produce triggering conditions that are *weaker* than needed. Still, we can guarantee that the approach is correct and consistent with respect to the operations’s preconditions (including their previous refinements). So, the overall refinement process would first remove time-progress violations, second operationalise safety goals, and finally deal with liveness goals.

6 Demonstrating Examples

In this section, we report the experimental results of applying our approach to two models, namely the Mine Pump Controller [13] and the Engineered Safety Feature Actuation System (ESFAS) [19]. For the case of safety goals and the time progress property, our approach is compared with the framework based on Inductive Logic Programming (ILP) introduced in [2], for evaluation and validation purposes. One of the authors provided the positive scenarios needed for ILP-based framework. These were produced manually following the guidelines provided in [1]. This human intervention, however, was not required in our proposed approach since it is fully automated. On the other hand, previous approaches to goal operationalisation do not deal with liveness goals, so we do not have previous results to compare with, to validate the technique. We argue about this problem later on in this section.

Each model is accompanied by an informal description of the system-to-be, a partial operational requirements specification, and a set of goals specified in FLTL. The experiments can be reproduced by downloading:
`http://dc.exa.unrc.edu.ar/staff/rdegiovanni/icse2014.zip`
and following the instructions therein.

6.1 Mine Pump Controller

This first model was used as the running example in this article (refer to Section 3 for details). In addition to the already specified goals (Sections 3 and 4), we consider additional objectives that should be achieved by the system:

```
assert PumpOffWhenLowWater =
  □ (tick → (LowWater → ◯(¬tick 𝒲 (tick ∧¬PumpOn))))
assert AlarmWhenMethane =
  □ (tick → (Methane → ◯(¬tick 𝒲 (tick ∧ Alarm))))
```

We have already discussed earlier in the article, in the form of examples, various refinements performed to the mine pump's operational model. Let us see now how we remove time progress violations. By performing a TP progress check on the Mine Pump system, considering the needed assumptions, LTSA produces the following counterexample.

```
TP violation. Trace to DEADLOCK:
  tick
  switchPumpOn
  raiseAlarm
  aboveLow
  signalMethane
```

Extending the counterexample with a tick event at the end, the interpolant computed is: $\texttt{PumpOn} \wedge \texttt{LowWater}$. It exhibits a similar violation to the goal PumpOffWhenLowWater shown in subsection 4.2, where we get rid of this counterexample prohibiting the execution of switchPumpOn when LowWater (refinement (T1)). We follow the approach put forward in [1], which proposes removing first time progress violations and then safety violations.

We now present a summary of the iterations performed by our refinement process, in which each iteration indicates the required condition identified to be added to the operational specification. Required preconditions (T1)-(T4) remove time progress violations, and the required triggering conditions (T5)-(T8) guarantee the satisfaction of the safety goals.

$$\mathit{ReqPre}(\texttt{switchPumpOn}) = \neg\texttt{LowWater} \tag{T1}$$

$$\mathit{ReqPre}(\texttt{stopAlarm}) = \neg\texttt{Methane} \tag{T2}$$

$$\mathit{ReqPre}(\texttt{switchPumpOn}) = \neg\texttt{Methane} \tag{T3}$$

$$\mathit{ReqPre}(\texttt{switchPumpOff}) = \neg(\neg\texttt{Methane} \wedge \texttt{HighWater}) \tag{T4}$$

$$\mathit{ReqTrig}(\texttt{raiseAlarm}) = \neg\texttt{Alarm} \wedge \texttt{Methane} \tag{T5}$$

$$\mathit{ReqTrig}(\texttt{switchPumpOn}) = \neg\texttt{PumpOn} \wedge \neg\texttt{Methane} \wedge \texttt{HighWater} \tag{T6}$$

$$\mathit{ReqTrig}(\texttt{switchPumpOff}) = \texttt{PumpOn} \wedge \texttt{LowWater} \tag{T7}$$

$$\mathit{ReqTrig}(\texttt{switchPumpOff}) = \texttt{PumpOn} \wedge \texttt{Methane} \tag{T8}$$

When compared with the requirements learned by the approach presented in [2], for this same model, we observe that both approaches iterate exactly the same number of times. Both approaches produce the same required conditions, except for the required triggering condition produced in the sixth iteration. The ILP-based framework learns a weaker triggering condition for `switchPumpOn`, namely `HighWater`. Due to this overgeneralisation problem of ILP, the learned condition leads to the following deadlock in the system:

```
Trace to DEADLOCK:
  tick (s0)
  aboveLow
  tick (s1)
  aboveHigh
  signalMethane
  tick (s2)
  belowHigh
  signalNoMethane
```

The deadlock is produced because in the state `(s2)`, both `HighWater` and `Methane` are true. Then, the required precondition from (T3) indicates that `switchPumpOn` cannot occur when `Methane`, but the learned triggering condition obliges `switchPumpOn` to occur when `HighWater`. In our case, the required triggering condition refined in (T6) is stronger, requiring methane to be false when the water level is above high. On the other hand, to remove the deadlock, the ILP-based framework is forced to backtrack to previous iterations and requires the engineer to provide further scenarios or manually produce the refinement.

With respect to running times, it is worth mentioning that, for this model, our approach is significantly more efficient. While the ILP-based framework requires aprox. 29 seconds per iteration, our approach takes less than 1 sec. per iteration.

6.2 ESFAS

The Engineered Safety Feature Actuation System (ESFAS) was originally introduced by Courtois and Parnas in [5]. Later on, Letier reported a KAOS specification of this system in [19]. The ESFAS system of a nuclear power plant prevents or mitigates damage to the core and coolant system of the plant, on the occurrence of a fault such as a loss of coolant. ESFAS monitors the water pressure, and a couple of switches for blocking and resetting, and it controls a single boolean variable, indicating whether the safety injection system is on or off, with the events `sendSignal` and `stopSignal`. Basically, the system must start safety injection when the pressure becomes too low. In addition, the system can be "disengaged" via the switches, indicating that its actions are overridden (`overrideSignal`, `enableSignal`). The goals that ESFAS has to satisfy are formalised in FLTL as follows:

```
assert SafetyInjectionWhenLowWaterPressureAndNotOverridden =
  □ (tick → ( (PressureBelowLow ∧ ¬Overridden)
                           → ○(¬tick 𝒲 (tick ∧ SafetyInjection))))
assert SIEnabledWhenPressureAbovePermitOrManualReset =
  □ (tick → ( (Occurs_reset ∨ PressureAbovePermit)
                    → ○(¬tick 𝒲 (tick ∧ ¬Overridden))))
assert SIOverriddenWhenBlockSwOnAndPressureLessThanPermit =
  □ (tick → ( (Occurs_block ∧ ¬PressureAbovePermit)
                    → ○(¬tick 𝒲 (tick ∧ Overridden))))
```

Intuitively, the first goal indicates that the safety injection signal should be on when the water pressure is below ‘Low’ and the safety injection is not overridden (this is the main objective of the ESFAS). The second goal expresses that the safety injection should become enabled when the water pressure raises above ‘Permit’ or when the reset button is pushed. And the third goal establishes that the safety injection should become overridden when the block switch is set on and the water pressure is lower than ‘Permit’. The following fluent definitions are considered for the ESFAS system:

```
fluent Overridden = < overrideSignal, enableSignal, True>
fluent SafetyInjection = < sendSignal,stopSignal >
fluent PressureBelowLow =
   < lowerPressureBelowLow, raisePressureAboveLow, True>
fluent PressureAbovePermit =
   <raisePressureAbovePermit, lowerPressureBelowPermit>
fluent Occurs_block = < block,tock >
fluent Occurs_reset = < reset,tock >
```

Event tock is executed immediately after tick. This auxiliary event is introduced as a terminating action for the operators’ press-button actions (the actions corresponding to the system operator pressing the reset and block buttons). We assume true and false to be the initial required preconditions and the required triggering conditions for the operations, respectively. In addition, we consider the next domain preconditions for the controlled operations and two environmental assumptions to indicate that the operator cannot press the reset and block buttons at the same time:

$$\mathit{DomPre}(\texttt{overrideSignal}) = \neg\texttt{Overridden}$$
$$\mathit{DomPre}(\texttt{enableSignal}) = \texttt{Overridden}$$
$$\mathit{DomPre}(\texttt{sendSignal}) = \neg\texttt{SafetyInjection}$$
$$\mathit{DomPre}(\texttt{stopSignal}) = \texttt{SafetyInjection}$$
$$\mathit{Assumption\_1} = \Box(\texttt{tick} \wedge \texttt{Occurs\_block} \rightarrow \neg\texttt{Occurs\_reset})$$
$$\mathit{Assumption\_2} = \Box(\texttt{tick} \wedge \texttt{Occurs\_reset} \rightarrow \neg\texttt{Occurs\_block})$$

The process checks the validity of the TP property, and refines 4 conditions to remove the violations.

$$\begin{aligned}
&\mathit{ReqPre}(\texttt{stopSignal}) = \neg(\neg\texttt{Overridden} \wedge \texttt{PressureBelowLow}) && \text{(R1)}\\
&\mathit{ReqPre}_1(\texttt{overrideSignal}) = \neg\texttt{Occurs\_reset} && \text{(R2)}\\
&\mathit{ReqPre}_2(\texttt{overrideSignal}) = \neg\texttt{PressureAbovePermit} && \text{(R3)}\\
&\mathit{ReqPre}(\texttt{enableSignal}) = && \text{(R4)}\\
&\qquad\qquad \neg(\neg\texttt{PressureAbovePermit} \wedge \texttt{Occurs\_block})
\end{aligned}$$

Now, the approach starts the refinement process of Section 4 in order to fulfil the safety goals. The first counterexample that LTSA reports is:

```
Trace to property violation in
SafetyInjectionWhenLowWaterPressureAndNotOverridden:
  tick  PressureBelowLow  (s0)
  tock  PressureBelowLow
  tick  PressureBelowLow  (s1)
```

This counterexample corresponds to a case in which the water pressure is below low and the system is not overridden, and in the next time-unit the safety injection signal is off. The process continues by computing an interpolant for the counterexample and the violated goal, obtaining $\neg\texttt{SafetyInjection} \wedge \neg\texttt{Overridden} \wedge \texttt{PressureBelowLow}$. The weakest precondition of this interpolant with respect to the last non-tick operation (i.e., tock), leads us to the same formula (the interpolant). So, by preventing tock from occurring we do not stop violating the goal. Thus, our approach attempts to remove the counterexample by forcing the occurrence of an operation such that the operation's execution avoids the interpolant. Operation sendSignal meets the two conditions for required triggering condition refinement, namely:

$$\begin{aligned}
&I \Rightarrow \neg\texttt{SafetyInjection} \wedge \texttt{true} && \text{(A)}\\
&(I \wedge \texttt{SafetyInjection}' \wedge (\texttt{Overridden}' = \texttt{Overridden}) \wedge && \text{(B)}\\
&(\texttt{PressureBelowLow}' = \texttt{PressureBelowLow})) \Rightarrow \neg(I')
\end{aligned}$$

where I represents the above mentioned interpolant. Condition (A) ensures that sendSignal can be executed when $\neg\texttt{SafetyInjection} \wedge \neg\texttt{Overridden} \wedge \texttt{PressureBelowLow}$, while condition (B) ensures that sendSignal's execution avoids the interpolant. Then, by forcing sendSignal to occur when $\neg\texttt{SafetyInjection} \wedge \neg\texttt{Overridden} \wedge \texttt{PressureBelowLow}$, the counterexample is removed. The fifth iteration finishes refining sendSignal's required triggering condition (R5):

$$\mathit{ReqTrig}(\mathtt{sendSignal}) = \tag{R5}$$
$$\neg\mathtt{SafetyInjection} \wedge \neg\mathtt{Overridden} \wedge \mathtt{PressureBelowLow}$$
$$\mathit{ReqTrig}_1(\mathtt{enableSignal}) = \mathtt{Overridden} \wedge \mathtt{Occurs\_reset} \tag{R6}$$
$$\mathit{ReqTrig}_2(\mathtt{enableSignal}) = \mathtt{Overridden} \wedge \mathtt{PressureAbovePermit} \tag{R7}$$
$$\mathit{ReqTrig}(\mathtt{overrideSignal}) = \tag{R8}$$
$$\neg\mathtt{Overridden} \wedge \mathtt{Occurs\_block} \wedge \neg\mathtt{PressureAbovePermit}$$

The refined required conditions of subsequent iterations (R5)-(R8) of our process guarantee the satisfaction of the safety goals. Notice that, for example in (R5), $\neg$SafetyInjection is redundant in sendSignal's required triggering condition, since $\neg$SafetyInjection is the domain precondition of sendSignal. The approach guarantees that overrideSignal's required triggering condition (R8) does not contradict overrideSignal's required preconditions (R2) and (R3), thanks to the first environment assumption, which expresses that Occurs_block $\Rightarrow$ $\neg$Occurs_reset. The second assumption is used for justifying that enableSignal's required triggering conditions (R6) and (R7) do not contradict enableSignal's required precondition (R4).

The operational requirements specification obtained by the above requirements achieves the ESFAS goals, so our process successfully terminates.

When comparing our approach with the ILP-based framework in [2], the first difference shows up in the fifth iteration. The ILP-based framework removes the same counterexample, but due to the problem of overgeneralisation of this approach, it produces a weaker required triggering condition for sendSignal: PressureBelowLow (i.e., rather than $\neg$Overridden $\wedge$ PressureBelowLow). This condition forces sendSignal to occur when the water pressure is below the low threshold regardless of whether it is overridden or not.

In the sixth and seventh, both approaches produce the same required conditions. In the eighth iteration however, a second difference is detected. The ILP-based framework learns a weaker triggering condition for overrideSignal, namely Occurs_block. This learned required condition produces a deadlock in the system, for a reason similar to the case of the Mine Pump. To remove the deadlock, the ILP-based framework requires the engineer to backtrack to previous iterations and either manually refine the required conditions or provide further positive and negative scenarios, and rerun the operationalisation process.

Let us compare the running times of our approach with those of the ILP-framework. Since both approaches use LTSA, the critical part in time is interpolation plus SAT vs. inductive learning. In the case of ILP, the time increases from 6 seconds in the first iteration to 18 seconds in the eighth, because the set of examples gets bigger (the engineer accumulates the positive scenarios from each iteration). Our approach takes less than 1 second per iteration.

6.3 Analysing Reactivity Properties

Previous approaches to goal operationalisation do not deal with liveness goals, so we do not have previous results or case studies to compare with, to validate our technique. In particular, Letier's patterns based approach classifies these goals as unrealisable [18]. So, the specifications that we use for evaluation (MinePump, ESFAS) do not have liveness goals assigned to agents, to be operationalised.

We then developed some liveness goals that, based on the knowledge we have on the models, should be satisfied, although these do not appear explicitly in the specifications. In addition, we do not use fairness assumptions, and we have to assume certain liveness constraints on the environment (e.g. if the pump is on, eventually the water level is not high), and to remove some safety goals that imply our developed properties, so that liveness counterexamples emerge.

For the Mine Pump Controller, the introduced property is: *"If infinitely often there is no methane (so the pump can be turned on), then infinitely often the water level is not high."* We specify this property as:

$$\Box(\mathrm{PumpOn} \rightarrow \Diamond\neg\mathrm{HighWater}) \wedge (\Box\Diamond\neg\mathrm{Methane} \rightarrow \Box\Diamond\neg\mathrm{HighWater})$$

In order to obtain a counterexample for this goal, we remove safety goals that lead to required conditions (T4) and (T6). Then, a counterexample for the goal is obtained, which has the following loop:

```
Violation of LTL property: @LivenessGoal
Trace to terminal set of states:
  ...
  signalMethane  HighWater ∧ Methane
Cycle in terminal set:
  tick  HighWater ∧ Methane  (s3)
  signalNoMethane  HighWater
  tick  HighWater  (s4)
  signalMethane  HighWater ∧ Methane
```

Notice that `(s4)` is the state in the loop that satisfies the assumption `¬Methane`, but neither `(s3)` nor `(s4)` satisfy the goal `¬HighWater`. Then, the interpolant computed for this counterexample is the following:

```
   (Methane3 ∧ HighWater3 ∧ ¬PumpOn3)
∧ (¬Methane4 ∧ HighWater4 ∧ ¬PumpOn4)
```

This interpolant explains what is wrong in the loop. To remove this counterexample, we search for an operation that can be executed at some point in the loop, such that its execution reaches a states that does not satisfy the interpolant. `switchPumpOn` meets these two conditions. Then, the refinement weakens the `switchPumpOn`'s required triggering condition with the conjunction of its required precondition and the negation of the goal, which correctly operationalises the goal:

$$\mathit{ReqTrig}(\texttt{switchPumpOn}) = \neg\texttt{Methane} \wedge \neg\texttt{LowWater} \wedge \texttt{HighWater} \tag{L1}$$

For the ESFAS system, the developed property is: *"If infinitely often the user does not override the system pressing the block button, then infinitely often the level of coolant won't be low"*. The property to operationalise is:

$$\begin{aligned}&\Box(\mathrm{SafetyInjection} \rightarrow \neg \mathrm{PressureBelowLow}) \wedge\\ &\qquad(\Box\Diamond\neg \mathrm{Occurs\_block} \rightarrow \Box\Diamond\neg \mathrm{PressureBelowLow})\end{aligned}$$

The counterexample found for this goal is:

```
Violation of LTL property: @LivenessGoal
Trace to terminal set of states:
  ...
  tick  PressureBelowLow
Cycle in terminal set:
  tock  PressureBelowLow
  tick  PressureBelowLow
```

Then, the refinement process computes an interpolant and weakens the required triggering condition for `sendSignal` operation (in this case, weaker than (R5)), which correctly operationalises the goal:

$$\mathit{ReqTrig}(\texttt{sendSignal}) = \neg\neg\texttt{PressureBelowLow} \tag{L2}$$

7 Discussion and Related Work

Goal-oriented requirements engineering (e.g., KAOS [14] and I* [30]) has been the focus of much research in the requirements engineering community. An important aspect of this approach to requirements engineering is the notion of relating high-level goals achievable only through agent cooperation with lower level goals that can be assigned to specific agents, some of which may be software to be built. Support for refining goals has been studied extensively too (e.g. [3]).

Goal operationalisation aims to produce requirements on a per-operation basis that will be provided by a specific agent to guarantee it achieves a goal that has been assigned to it. Approaches concerned with goal operationalisation are for instance the NFR framework [25] and CREWS [29]. However, these either focus on non-functional requirements or are informal and hence cannot be fully verified. More formal approaches such as [10, 11] allow checking the correctness of operationalisation rather than supporting the elaboration of such operational requirements.

The use of generalisation techniques in the context of goal models is not novel. For instance, [16] presents a method for inferring declarative assertions from scenarios. It elicits goals from tailored scenarios provided by stake-holders using an inductive inference process based on Explanation-Based Learning (EBL) [26]. Apart from not being used specifically for operationalisation, the learning in [16] cannot consider existing knowledge (e.g., existing goals or operational requirements) during the inference process. Hence it is unsound and can produce inconsistent specifications.

Interpolation has been used for software analysis purposes, notably by McMillan [28], in combination with SAT-based model checking, for circuit verification [27]. It has also been employed for automated counterexample guided abstraction refinement [7]. Essentially, interpolation is used in the context of the verification of abstract (imprecise) models. When a counterexample is obtained, it must be checked whether it is an actual counterexample or it arised due to the imprecision of the model. If it is spurious, then when building the conjunction of the model and the counterexample, one arrives to an unsatisfiable formula. Interpolation *explains* what is the difference between the abstract and concrete models of the system, that led to the spurious counterexample. The interpolants obtained can be made part of the abstract model to make it more concrete, and thus get rid of the spurious counterexample. This process is iterated until no further counterexamples are obtained or a real (non spurious) counterexample is produced. In this work, we proposed using interpolation for a different, but related, purpose. Basically, from concrete counterexamples showing goal violations we produce unsatisfiable formulas so that interpolants can be computed. These interpolants are used to refine the concrete operation model. As opposed to abstraction refinement, in our setting we do not have a model of reference to be used for refinement (the concrete model in the context of abstraction refinement); instead, we have an *objective*, namely to correctly and completely operationalise the goals. The deviation from the objective is what guides our process in the use of interpolation, for refining the operation model.

Other related approaches are the works on synthesis of behaviour models (in form of LTS) from goals [8, 9, 21]. In contrast with our approach, which produces declarative required pre/triggering conditions for controlled operations, these works based on synthesis produce operational refinements for controlled operations.

There exist other approaches that deal with the problem of goal operationalisation systematically. As mentioned in Section 1, the approach introduced in this article is more closely related to the framework presented in [2], which provides a semi-automated method that uses model checking for analysing goal achievement and Inductive Logic Programming (ILP) for learning operational requirements. There are some important differences between the two approaches. [2] requires user intervention to provide positive scenarios for the learning phase and hence its results are dependent on the correctness and ‘richness’ [1] of the scenarios given, whereas this is not required by our presented approach. [2] uses ILP which searches for the ‘most compressed’ conditions (i.e., fewest number of fluent literals appearing in the required conditions) and hence is prone to generating over-generalised conditions. The approach introduced in this article, on the other hand, uses interpolation, which produces more precise conditions, as the interpolant is necessarily implied by the counterexample trace and it necessarily leads to a violation of a goal. Exactly because of this, interpolation based refinement may require more iterations to reach a valid operationalisation, compared to the ILP approach. Finally, our approach is able to deal with a wide range of liveness properties, which are not handled by previous approaches to goal operationalisation.

8 Conclusion and Future Work

We presented an approach for goal operationalisation, that *automatically* computes required pre/triggering conditions for operations, in order to fulfil a set of goals. Moreover, this approach does not depend on user provided scenarios and their characteristics, e.g., “richness” and correctness, as is the case with [2]. This approach is based on interpolation and SAT solving, and applies to safety goals and particular kinds of liveness goals, namely reactivity properties (a general class that embraces many liveness properties). We have evaluated our technique on some models taken from the literature, and compared our approach with that of [2], showing that in these cases our approach is able to produce goal operatinalisations effectively and more efficiently.

There are various lines for future work. We plan to carry out case studies to validate our technique. This may, in particular, enable us to evaluate possible scalability issues with our approach to goal operationalisation. In addition, we plan to investigate what is the precise relationship between operationalisations obtained by interpolation and those obtained by inductive logic programming. In particular, we are interested in analysing a possible notion of “most general” operationalisation, and to assess whether interpolation based refinement enable us to reach such operationalisations. We also plan to investigate a potential complementation between interpolation and inductive logic programming, for goal operationalisation. Finally, since our approach heavily relies on the calculation of interpolants, we plan to evaluate alternative mechanisms for interpolant computation, to analyse whether a particular interpolant computation approach better fits our purposes.

Bibliography

[1] D. Alrajeh, J. Kramer, A. Russo and S. Uchitel, *E*laborating Requirements Using Model Checking and Inductive Learning. IEEE Transactions on Software Engineering 39(3): 361–383, 2013.

[2] D. Alrajeh, J. Kramer, A. Russo and S. Uchitel, *Learning Operational Requirements from Goal Models*, in Proc. of ICSE 2009, IEEE, 2009.

[3] A. I. Anton, *Goal Identification and Refinement in the Specification of Software-based Information Systems*, PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1997.

[4] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio and R. Sebastiani, *The MathSAT 4 SMT Solver*, in Proc. of CAV 2008, LNCS 5123, Springer, 2008.

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

[6] E. Clarke, *Automatic verification of hardware and software systems*, ACM SIGSOFT Software Engineering Notes 25(1): 41-42, 2000.

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. *Counterexample-guided abstraction refinement for symbolic Model Checking*. J. ACM, 50(5):752–794, 2003.

[8] N. D’Ippolito, V. Braberman, N. Piterman, and S. Uchitel, *Synthesis of live behaviour models*, in FSE 2010, ACM, 2010.

[9] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel, *Synthesis of live behaviour models for fallible domains*, in ICSE 2011, ACM, 2011.

[10] A. Fuxman, J. Mylopoulos, M. Pistore, and P. Traverso, *Model checking early requirements specifications in TROPOS*, in Proceedings of the 5th IEEE International Symposium on Requirements Engineering, pages 174181, 2001.

[11] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso, *Specifying and analyzing early requirements in TROPOS*, Requirements Engineering, 9(2):132–150, 2004.

[12] D. Giannakopoulou and J. Magee, *Fluent Model Checking for Event-based Systems*, in Proc. of ESEC/FSE '03, ACM 1-58113-743-5/03/0009, 2003.

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

[14] A. van Lamsweerde, A. Dardeene, D. Delcourt and F. Dubisy, *The KAOS Project: Knowledge Acquisition in Automated Specification of Software*, in Proc. of AAAI Spring Symposium Series, Track: "Design of Composite Systems", Stanford University, March 1991, 59–62.

[15] A. van Lamsweerde, A. Dardeene and S. Fickas, *Goal-directed Requirements Acquisition*, in Proc. of Science in Computer Programming, Vol.20, 3–50, 1993.

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

[17] E. Letier and A. van Lamsweerde, *Deriving Operational Software Specifications from System Goals*, in Proc. of 10th ACM SIGSOFT International Symposium on Foundations of software engineering, 2002.

[18] E. Letier, *Reasoning about Agents in Goal-Oriented Requirements Engineering*, PdD Thesis, D'partement d'Ing'nierie Informatique, UCL, 2001.

[19] E. Letier, *Goal-oriented elaboration of requirements for a safety injection control system*, technical report, D'partement d'Ing'nierie Informatique, UCL, 2002.

[20] E. Letier, J. Kramer, J. Magee and S. Uchitel, *Deriving Event-Based Transition Systems from Goal-Oriented Requirements Models*, Journal of ASE, 15:175–206, 2008.

[21] E. Letier and W. Heaven, *Requirements Modelling by Synthesis of Deontic Input-Output Automata*, in Proc. of ICSE 2013, IEEE, 2013.

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

[23] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems specification*, Springer, 1992.

[24] Z. Manna and A. Pnueli, *Temporal verification of reactive systems - safety*, Springer, 1995.

[25] J. Mylopoulos, L. Chung, and B. A. Nixon, *Representing and using non-functional requirements: A process-oriented approach*, IEEE Transactions on Software Engineering, 18:483497, 1992.

[26] T. Mitchell, *Machine Learning*, McGraw Hill, 1997.

[27] K. McMillan, *Interpolation and SAT-Based Model Checking*, in Proc. of CAV 2003, LNCS 2725, Springer, 2003.

[28] K. McMillan, *Applications of Craig Interpolants in Model Checking*, in Proc. of TACAS 2005, LNCS 3440, 2005.

[29] C. Rolland, C. Souveyet, and C. B. Achour, *Guiding goal modelling using scenarios*, IEEE Transaction on Software Engineering, 24(12):10551071, 1998.

[30] E. S. K. Yu, *Towards Modelling and Reasoning Support for Early-Phase Requirement Engineering*, IEEE Int. Symp. on Requirements Eng. pp. 226-235, 1997.

MEALS Partner Abbreviations

SAU: Saarland University, D

RWT: RWTH Aachen University, D

TUD: Technische Universität Dresden, D

INR: Institut National de Recherche en Informatique et en Automatique, FR

IMP: Imperial College of Science, Technology and Medicine, UK

ULEIC: University of Leicester, UK

TUE: Technische Universiteit Eindhoven, NL

UNC: Universidad Nacional de Córdoba, AR

UBA: Universidad de Buenos Aires, AR

UNR: Universidad Nacional de Río Cuarto, AR

ITBA: Instituto Técnológico Buenos Aires, AR