Goal-Conflict Detection based on Temporal Satisfiability Checking

Renzo Degiovanni* Nicolas Ricci* Dalal Alrajeh† Pablo Castro* Nazareno Aguirre*

*Departamento de Computación, Universidad Nacional de Río Cuarto and CONICET, Argentina
†Department of Computing, Imperial College London, UK
{rdegiovanni, nricci, pcastro, naguirre}@dc.exa.unrc.edu.ar, dalal.alrajeh@ic.ac.uk

ABSTRACT

Goal-oriented requirements engineering approaches propose capturing how a system should behave through the specification of high-level goals, from which requirements can then be systematically derived. Goals may however admit subtle situations that make them diverge, i.e., not be satisfiable as a whole under specific circumstances feasible within the domain, called *boundary conditions*. While previous work allows one to identify boundary conditions for conflicting goals written in LTL, it does so through a pattern-based approach, that supports a limited set of patterns, and only produces pre-determined formulations of boundary conditions.

We present a novel automated approach to compute boundary conditions for general classes of conflicting goals expressed in LTL, using a tableaux-based LTL satisfiability procedure. A tableau for an LTL formula is a finite representation of *all* its satisfying models, which we process to produce boundary conditions that violate the formula, indicating divergence situations. We show that our technique can automatically produce boundary conditions that are more general than those obtainable through existing previous pattern-based approaches, and can also generate boundary conditions for goals that are not captured by these patterns.

CCS Concepts

•**Software and its engineering** → **Requirements analysis;** Risk management; •**Theory of computation** → *Automated reasoning;* Modal and temporal logics;

Keywords

Goal Conflicts, Satisfiability Checking, Tableaux Method

1. INTRODUCTION

The derivation of correct software requirements specifications is essential to any reliable software development process [1]. With the ever increasing complexity of software, the importance of rigorous methods in supporting the attainment of correct specifications prior to their implementation, also increases. Much research over the last decades has demonstrated the significant advantages that formal, goal-oriented approaches bring to the generation of correct software requirements specification. Goals are prescriptive statements of how the system should behave. They reflect stakeholders' understanding of what the envisioned system is intended to do, and the criteria upon which it would be evaluated. They are commonly used to: aid the elicitation and elaboration of requirements [43, 47]; guide the refinement and organisation of requirements [4]; and support the derivation of software operations [2, 8]. However, for such tasks to be successfully achieved, the goals themselves must be correct, which is not often the case. Goals are typically too ideal to start with (wavering off the exceptional conditions that may arise within its environment once implemented [43, 3]), partial and imprecise. Ensuring their correctness within the development cycle is of utmost importance.

One of the challenges in specifying correct goals is ensuring their consistency. Inconsistency occurs when two or more goals cannot be satisfied simultaneously, owing to their contradictory nature, non-conformance to standards, or because of restrictions imposed within certain domains, amongst other reasons [20]. They are typically a result of overlapping and conflicting expressions. Detecting and resolving inconsistencies in goals (a process called *inconsistency management*) early on not only helps in avoiding costly software repairs but also supports systematic requirements' elicitation and verification activities [35, 44]. Several approaches have been proposed in the literature for managing inconsistency in goals. Much work has been done on the qualitative end, e.g., [33, 15], where the general focus has been on identifying contradictory low-level requirements and computing the degree to which goals are satisficed or denied by them. On the formal side, inconsistency management has also been the focus of several studies, e.g., [44, 18, 11, 34, 10].

A weaker notion of conflict (called *divergence*) in goals expressed in Linear Temporal Logic (LTL) [37] has been addressed in [44]. This latter type of inconsistency is concerned with those goals which are not contradictory (can be simultaneously satisfied), but become inconsistent when certain conditions hold. Consider for instance the following goals from the mine pump controller example [25]: *"the pump shall be on when the water level is above the high threshold"*, and *"the pump shall be off when methane is detected in the mine"*. These goals are not logically inconsistent as they are satisfiable in cases where the water level never reaches a high level or methane is not detected in the mine. They become logically inconsistent only in the case when the water level is high and methane is present at the same time. Situations like the latter can be captured formally as assertions called *boundary conditions*, i.e., declarative formulas that characterise those particular circumstances that lead to inconsistency. Existing model synthesis approaches would not detect this type of inconsistency since there exists at least one model that satisfies such goals, in which the boundary condition never holds. So far, very limited work has been done on automatically finding boundary conditions for goal expressions, save for [44] which supposes goals expressed according to fixed templates written in LTL.

In this paper, we present a novel approach to automatically compute boundary conditions for conflicting goals expressed in LTL, using a satisfiability procedure based on tableaux. A tableau for an LTL formula is essentially a finite graph representation of all its satisfying models; it is built by first decomposing the formula whose satisfiability is being analysed, according to decomposition rules that produce, for temporal operators, constraints on the current state and future states, for their satisfaction. The resulting graph explores the possible ways of making the initial formula satisfiable, and in this process, contradictory portions are identified. The second phase of the tableau method removes contradictory portions, as well as parts of the graph that cannot satisfy eventualities, leaving a subgraph, the tableau, that captures *all* models of the formula (when it becomes empty, the formula is unsatisfiable). Intuitively, the tableau indirectly captures "conflicting situations", since any condition not included in the tableau necessarily prevents the formula from being satisfiable. Our approach consists of computing the tableau from a set G of goals, and then exploring it to identify conditions that would "escape" the tableau, thus violating the goals, to produce boundary conditions. Our approach is general, in the sense that it can automatically detect conflicts in goals expressed as any LTL formula (as opposed to [44], where goals must comply with specific patterns for conflict detection). In particular, our approach can be applied to all patterns in [44]. In fact, as we will show, our technique produces more general boundary conditions than those of the pattern-based approach, while at the same time it allows us to compute boundary conditions from goals not captured by these patterns.

The rest of the paper is organised as follow. In Section 2 we present the basic concepts that will be necessary throughout the paper. In Section 3 we present an illustrating example, together with some observations that motivate the approach. The approach itself is described in detail, in Section 4. We then perform a validation of our technique (Section 5), by comparing it with the pattern-based mechanism for conflict detection, and by applying our approach on various case studies. Finally, we discuss related work in Section 6, and draw some conclusions and further work in Section 7.

2. BACKGROUND

2.1 Goal-Oriented Modelling

Goal-Oriented Requirements Engineering [43] proposes capturing how a system should behave through the specification of a set of *high-level goals*, that will drive the requirements engineering process. Goals are *prescriptive* statements that the envisioned system is expected to achieve through the cooperation of its agents (e.g., humans, devices and software) within a given domain. Domain properties are *descriptive* statements about the problem world (such as natural laws). In this setting, a *goal model* is a decomposition of goals through refinements, which essentially capture how a goal can be achieved in terms of simpler ones. These goal refinements end when each leaf subgoal can be assigned to a single agent. Agents provide operations, whose combined behaviours must fulfill the goals. A *requirement* is a terminal goal assigned to a software agent, while an *assumption* is a goal assigned to an agent in the environment.

In this context, inconsistent goals may arise from conflicting expressions. In particular, a weak form of conflict, called *divergence*, is of relevance in goal-oriented requirements engineering. A set of goals $G_1, \ldots, G_n$ is said to be *divergent* [44, 43] with respect to a set Dom of domain properties iff there exists a *boundary condition* BC such that the following conditions hold:

$$\{Dom, BC, \bigwedge_{1\leq i\leq n} G_i\} \models \mathit{false} \qquad (\textit{logical inconsistency})$$

$$\{Dom, BC, \bigwedge_{j\neq i} G_j\} \not\models \mathit{false}, \text{for each } 1 \leq i \leq n \ (\textit{minimality})$$

$$BC \neq \neg(G_1 \wedge \ldots \wedge G_n) \qquad (\textit{non-triviality})$$

Intuitively, conditions 1 and 2 indicate that the boundary condition captures a particular combination of circumstances that makes the goals conflicting. Roughly speaking, the first condition establishes that goals $G1, \ldots, G_n$ cannot simultaneously be satisfied in Dom under any circumstances when BC holds. The second condition states that removing any of the goals no longer results in a logical inconsistency. The third condition prohibits a boundary condition to be simply the negation of goals. Notice also that the minimality condition forbids trivial boundary conditions (BC cannot be *false* nor can it be the negation of one of the goals ($\neg G_i$)), and requires it to be consistent with the domain Dom.

2.2 Linear-Time Temporal Logic

Linear-Time Temporal Logic (LTL) is a formalism that has been extensively employed to state properties of reactive systems. LTL assumes a lineal topology of time, i.e., each instant is followed by a unique future instant, and its formulas are evaluated over infinite traces that represent system executions. Given a set AP of propositional variables, LTL formulas are inductively defined using the standard logical connectives and temporal operators $\bigcirc$ and $\mathcal{U}$, as follows: *(i)* every $p \in AP$ is an LTL formula, and *(ii)* if f_1 and f_2 are LTL formulas, then so are $\neg f_1$, $f_1 \vee f_2$, $f_1 \wedge f_2$, $\bigcirc f_1$ and $f_1 \mathcal{U} f_2$. We consider the usual definition for the operators $\Box$ (always), $\Diamond$ (eventually) and $\mathcal{W}$ in terms of $\bigcirc$, $\mathcal{U}$ and logical connectives. Temporal formulas are evaluated over infinite traces of propositional valuations. Formulas with no temporal operators are evaluated in the first valuation of a trace. Given a trace σ, $\bigcirc f$ is true in σ iff f is true in $\sigma[1..]$ (the trace obtained by removing the first valuation from σ), and $f_1 \mathcal{U} f_2$ is true in σ iff there exists a position i such that f_2 holds in $\sigma[i..]$, and for all $0 \leq j < i$, f_1 holds in $\sigma[j..]$.

In this work we focus on *safety* properties, typically specified as $\Box f$, and a particular kind of *liveness* properties, namely those captured by the *reachability* and *response* patterns, specified as $\Diamond f$ and $\Box(f_1 \rightarrow \Diamond f_2)$, respectively. For further details on LTL and temporal patterns, see [30].

2.3 The Tableau Method for LTL

The tableau method is a well-known logical satisfiability approach, based on the decomposition of the formula being assessed according to the semantics of its logical operators, to search for satisfying valuations. In propositional logic, formulas are decomposed according to the semantics of boolean connectives: a model satisfies $f_1 \wedge f_2$ iff it satisfies *both* conjuncts at the same time; a model satisfies $f_1 \vee f_2$ iff it satisfies *at least one* of the disjuncts; and so on. Notice that if we start with the singleton containing the assessed formula and connect sets of formulas according to decomposition rules (e.g., $\{f_1 \wedge f_2\}$ will be connected to $\{f_1, f_2\}$), then this tableau process leads to a finite tree, the *tableau.* Branches originate due to disjunction; finiteness is guaranteed because sets containing contradictions are not further expanded, and atomic formulas cannot be decomposed. In a propositional tableau structure, leaves not containing contradictions characterise sets of satisfying valuations: those obtained by assigning true (resp., false) to atomic variables appearing positively (resp., negatively) in the branch, and assigning any truth value to any other atomic proposition.

The tableau method for LTL extends propositional tableau with rules to cope with temporal operators. Temporal operators will lead to requirements on the "current" state in a trace, and on the rest of the trace. For instance, $\Box f$ (globally f) will hold in a trace iff f holds in the current state of the trace, and $\Box f$ holds in the rest of the trace (or equivalently, if $\bigcirc\Box f$ holds in the current state). On the other hand, $\Diamond f$ leads to branching, as disjunction does: a trace satisfies $\Diamond f$ iff the current state satisfies f or the rest of the trace satisfies $\Diamond f$ (or equivalently, $\bigcirc\Diamond f$ is satisfied in the current state). Notice that these additional decomposition rules, if applied exhaustively, may lead to infinite branches. To avoid decomposing the same formula twice, decomposed formulas are marked, and if a decomposition leads to a previously constructed set, then the arc will lead to the already produced one (thus, LTL tableaux are *graphs*, as opposed to propositional tableaux, which are trees). Then, LTL tableaux will deal with sets of formulas and *marked* formulas (subformulas of that being assessed), connected according to decomposition rules. More precisely, LTL tableau works as follows. Initially, the tableau contains only one node, the *root*, consisting of the (unmarked) formula being queried for satisfiability. Then, each node will be decomposed according to the following rules:

$$\frac{f_1 \wedge f_2}{\begin{array}{c} f_1 \\ f_2 \end{array}}\ \alpha \qquad \frac{\Box f_1}{\begin{array}{c} f_1 \\ \bigcirc\Box f_1 \end{array}}\ \alpha$$

$$\frac{f_1 \vee f_2}{f_1 \mid f_2}\ \beta \qquad \frac{\Diamond f_1}{f_1 \mid \bigcirc\Diamond f_1}\ \beta \qquad \frac{f_1\ \mathcal{U}\ f_2}{f_2 \mid f_1 \wedge \bigcirc(f_1\ \mathcal{U}\ f_2)}\ \beta$$

Notice that we do not have rules for negation; we assume the original formula to be in negation normal form (NNF), where negations are pushed to be applied to atomic propositions. Rules labelled with α do not lead to branches, whereas those labelled with β produce branches.

Definition 2.1 (Elementary Formula). *We call* literal *to a propositional variable or its negation. A formula is* elementary *iff it is a literal, or a temporal formula whose main operator is* $\bigcirc$.

Then, the above rules are applied iteratively, taking into account marked vs unmarked formulas, and elementary formulas, as follows:

1. If n is a node labelled with a set S of formulas containing at least one unmarked non-elementary formula f, then: *(i)* if f is a formula α, and α_1, α_2 are the formulas resulting from decomposing f, then we create a node n' labelled with the set $(S - \{f\}) \cup \{\alpha_1, \alpha_2\} \cup \{f^*\}$, and then we connect n with n'; *(ii)* if f is a formula β, and β_1, β_2 are the formulas resulting from decomposing f, then we create two nodes n_1 and n_2, such that, n_1 is labelled with the set $(S - \{f\}) \cup \{f^*\} \cup \{\beta_1\}$ and n_2 is labelled with $(S - \{f\}) \cup \{f^*\} \cup \{\beta_2\}$, and finally we connect n with n_1 and n_2.
2. If n is a node labelled with a set S that contains only elementary and *marked* formulas, then we create a node n' labelled with the set S', such that, $\phi \in S'$ *iff* $\bigcirc\phi \in S$, and then we connect n with n'.

Definition 2.2 (States and Pre-States). *The set of nodes $S_N \subset N$ that contain only elementary or marked formulas are called* states, *whereas the set $P_N \subset N$ that contains the initial root node and immediate successors of states are called* pre-states.

After constructing the tableau, to decide if φ is satisfiable, the unsatisfiable nodes must be eliminated from the graph. To do so, the following deletion rules are repeatedly applied:

1. If a node contains both a proposition p and its negation $\neg p$, it is eliminated.
2. If all successors of a node have been eliminated, it is eliminated.
3. If a node is a pre-state that contains some eventuality $\Diamond f_2$ or $f_1\ \mathcal{U}\ f_2$ that is not satisfiable, then it is eliminated. A formula $\Diamond f_2$ or $f_1\ \mathcal{U}\ f_2$ is satisfiable in a pre-state iff *there exists a path* in the tableau leading from that pre-state to a node that contains formula f_2.

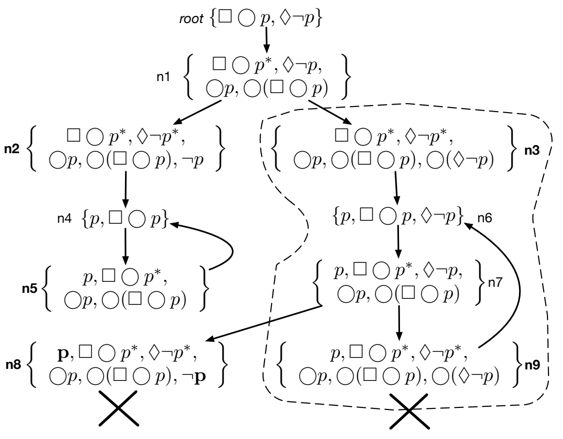
Intuitively, eventualities express "promises" that a property will eventually be fulfilled, so one must guarantee that a future state satisfies them, to consider these satisfiable.

The decision procedure ends when all unsatisfiable nodes have been removed from the tableau. If the initial node has been eliminated, the initial formula is unsatisfiable; if not, it is satisfiable. The reader is referred to [46] for further details and examples regarding the tableau construction for LTL.

As an example, consider the tableau in Figure 1, generated for the formula $\Box \bigcirc p \wedge \Diamond\neg p$. The *root* node contains $\{\Box \bigcirc p, \Diamond\neg p\}$ (we already decomposed the conjunction for space reasons). All nodes whose identifiers are in boldface ($n2$, $n3$, $n5$, $n8$ and $n9$) are *states.* Node $n8$ cannot be further expanded since it contains a propositional inconsistency (p and $\neg p$). Notice also that all nodes reachable from $n3$ contain p, and therefore they cannot fulfil the eventuality $\neg p$ associated to formula $\Diamond\neg p$. The parts of the tableau that are eliminated according to the above rules are highlighted in the figure. Since *root* belongs to the tableau after all deletion rules have been applied, $\Box \bigcirc p \wedge \Diamond\neg p$ is satisfiable; the tableau also indicates the only way to make it satisfiable is in a trace in which the eventuality $\neg p$ is fulfilled in the first state, while in the rest of the trace p is satisfied.

3. MOTIVATING EXAMPLE

To illustrate both the problem addressed and our proposed solution, let us consider a simplified version of the Mine Pump Controller (MPC) [25]. The MPC has a sensor that detects



root $\{\Box\bigcirc p, \Diamond\neg p\}$
n1 $\{\Box\bigcirc p^*, \Diamond\neg p, \bigcirc p, \bigcirc(\Box\bigcirc p)\}$
n2 $\{\Box\bigcirc p^*, \Diamond\neg p^*, \bigcirc p, \bigcirc(\Box\bigcirc p), \neg p\}$
$\{\Box\bigcirc p^*, \Diamond\neg p^*, \bigcirc p, \bigcirc(\Box\bigcirc p), \bigcirc(\Diamond\neg p)\}$ n3
n4 $\{p, \Box\bigcirc p\}$
$\{p, \Box\bigcirc p, \Diamond\neg p\}$ n6
n5 $\{p, \Box\bigcirc p^*, \bigcirc p, \bigcirc(\Box\bigcirc p)\}$
$\{p, \Box\bigcirc p^*, \Diamond\neg p, \bigcirc p, \bigcirc(\Box\bigcirc p)\}$ n7
n8 $\{\mathbf{p}, \Box\bigcirc p^*, \Diamond\neg p^*, \bigcirc p, \bigcirc(\Box\bigcirc p), \neg\mathbf{p}\}$
$\{p, \Box\bigcirc p^*, \Diamond\neg p^*, \bigcirc p, \bigcirc(\Box\bigcirc p), \bigcirc(\Diamond\neg p)\}$ n9

Figure 1: Tableau Example.

when the water level is high, and a sensor to detect the presence of methane in the environment. The propositional variables hw, m and po are employed to represent the facts that a high water level is reached, that methane is present in the environment, and that the pump is on, respectively. In this context, the following goals are relevant:

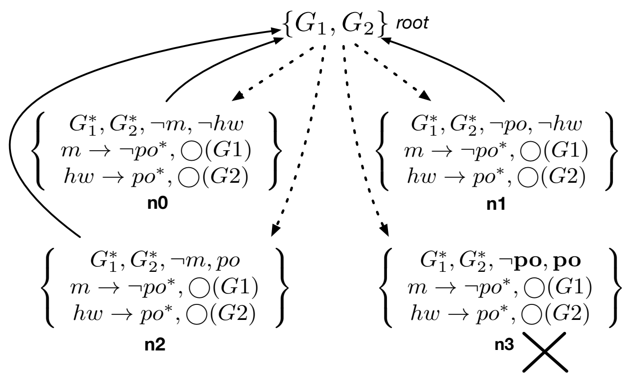
Goal: Maintain[*PumpOffWhenMethane*]
InformalDef: The pump should be off when methane is detected in the mine.
FormalDef: $\Box(m \rightarrow \bigcirc(\neg po))$

Goal: Maintain[*PumpOnWhenHighWater*]
InformalDef: The pump should be on when the water level is above the high threshold.
FormalDef: $\Box(hw \rightarrow \bigcirc(po))$

While these goals can be simultaneously satisfied (e.g., when the water level is never high or methane is never present in the environment), they become logically inconsistent when, at the same time, the water level is high and methane is present. Such conflicting situations, in this case characterised by the boundary condition $\Diamond(hw \wedge m)$, are not obvious to identify, and are very important to detect. We propose automatically detecting such boundary conditions by constructing and processing a tableau from the goals' temporal formalisations. Let us provide some intuition on how the process works, in the particular case of safety goals.

From the above safety goals (that we denote here as G_1 and G_2 for space reasons), the tableau method previously introduced produces the tableau shown in Figure 2. The *root* node is the only pre-state, and nodes $n0$, $n1$, $n2$ and $n3$ are states. Also, since $n3$ contains both p and $\neg p$, it is not further expanded (will be removed from the tableau). The dashed lines in the figure indicate that target nodes are reached through intermediate nodes (that are neither states nor pre-states).

States in the tableau capture sets of valuations in particular instants of a trace, through the literals they contain. For instance, state $n0$ characterises valuations (or points during a trace execution) in which there is no methane and water level is not high ($\neg m \wedge \neg hw$). Moreover, if the "consistent" states in a tableau are identified (basically, the states of the tableau that remain after the removal phase), since these characterise *all* consistent situations, the negation of their disjunction indirectly captures how the goals would be violated (would



$\{G_1, G_2\}$ root
$\{G_1^*, G_2^*, \neg m, \neg hw, m \rightarrow \neg po^*, \bigcirc(G1), hw \rightarrow po^*, \bigcirc(G2)\}$ n0
$\{G_1^*, G_2^*, \neg po, \neg hw, m \rightarrow \neg po^*, \bigcirc(G1), hw \rightarrow po^*, \bigcirc(G2)\}$ n1
$\{G_1^*, G_2^*, \neg m, po, m \rightarrow \neg po^*, \bigcirc(G1), hw \rightarrow po^*, \bigcirc(G2)\}$ n2
$\{G_1^*, G_2^*, \neg\mathbf{po}, \mathbf{po}, m \rightarrow \neg po^*, \bigcirc(G1), hw \rightarrow po^*, \bigcirc(G2)\}$ n3

Figure 2: Tableau for the Mine Pump Controller.

"fall" into the unsatisfiable part of the tableau). In our particular example, this formula is $\neg((\neg m \wedge \neg hw) \vee (\neg m \wedge po) \vee (\neg po \wedge \neg hw))$, and reaching it leads to a violation of the goals. Thus, $\Diamond\neg((\neg m \wedge \neg hw) \vee (\neg m \wedge po) \vee (\neg po \wedge \neg hw))$ is a boundary condition (or equivalently $\Diamond((\neg hw \wedge m \wedge po) \vee (hw \wedge (m \vee \neg po)))$, as our tool produces it after simplification), evidencing a weak conflict between the goals. Actually, as we will explain later on, this is in fact a *potential* boundary condition, since it may not satisfy minimality (an additional checking must be performed on our produced violations to guarantee minimality).

It is worth noting that our computed boundary condition is more general than that manually identified above ($\Diamond(hw \wedge m)$). By more general we mean that the formula $\Diamond(hw \wedge m)$ *implies* the boundary condition computed by our approach. As we will show in our experimental assessment, this tableau-based approach produces more general conflicts than those obtained using alternative pattern-based techniques. In the next section, we describe in detail the approach, including how other kinds of goals (besides safety) are handled.

4. THE APPROACH

Our tableau-based approach to automatically detect goal conflicts receives a goal-oriented requirements specification composed of LTL formulas capturing the domain assumptions Dom, as well as goals $G = \{G_1, \ldots, G_n\}$. The process may determine that there are no conflicts, or that there exist either *strong* or *weak* conflicts. The most relevant case, upon which we concentrate in this work, is when *weak* conflicts are detected. If this is the case, our process produces a set $BC = \{BC_1, \ldots, BC_k\}$ of *boundary conditions* capturing different divergent situations between the goals in the domain.

Our approach is able to deal with *safety* goals and a particular kind of *liveness* goals, namely those that can be expressed following the *reachability* or *response* (progress) patterns [30]. Reachability goals have the form $\Diamond f$, while response goals have the form $\Box(f_1 \rightarrow \Diamond f_2)$. Progress goals, on the other hand, are expressed as $\Box\Diamond f_2$, and are a particular case of response goals (those where $f_1 = true$) [30].

The overall approach is summarised in the following steps. (1) **Tableau generation and refinement:** A tableau structure T for $Dom \wedge G$ is automatically generated using the LTL decision procedure presented in Section 2.3. As mentioned before, structure T encodes *all* models of $Dom \wedge G$. If after the deletion rules are applied all the nodes are removed from

Algorithm 1 takes as input an LTL specification of the domain *Dom* and of a set of goals *G*. The output indicates if the goals have *no* conflict, or if they are *strongly* or *weakly* conflicting. When goals are weakly conflicting, a set *BC* of boundary conditions capturing divergences is produced.

```
1: function DetectDivergences($Dom$, $G$): $BC$
2:   $T = \langle N, R, root \rangle \leftarrow$ LTL-TableauMethod($Dom \wedge G$)
3:   if $N = \emptyset$ then
4:     ▷ $Dom \wedge G$ is unsatisfiable.
5:     return UNSAT. Goals and domain specifications are contradictory.
6:   else
7:     ▷ $Dom \wedge G$ is satisfiable. Look for weak conflicts.
8:     ▷ compute potential boundary conditions.
9:     $pBC \leftarrow$ SafetyBC($T$)                        ▷ safety case.
10:    $G_{live} \leftarrow$ LivenessGoals($G$)
11:    for all $G_i \in G_{live}$ do                       ▷ liveness case.
12:      $pBC \leftarrow pBC \cup \{$LivenessBC($T$, $G_i$)$\}$
13:    end for
14:    ▷ Check divergence conditions.
15:    $BC \leftarrow$ FilterDivergences($Dom$, $G$, $pBC$)
16:    if $BC \neq \emptyset$ then
17:      return Weak conflict detected: $BC$.
18:    else
19:      return No conflict detected: $\emptyset$.
20:    end if
21:  end if
22: end function
```

T, then goals and domain specifications are contradictory, and we do not look at it any further; otherwise, we proceed to look for weak conflicts.

(2) **Potential *BC* identification:** At this point, tableau *T* produced in the previous phase is non-empty, and characterises all the models for $Dom \wedge G$, in the sense that every sequence that satisfies $Dom \wedge G$ is a path in the tableau *T*, and every *finite* path obtained from *T* is the prefix of some model of $Dom \wedge G$ [31]. However, there may exist some *infinite* paths in *T* that leave some eventuality formula unsatisfied, and thus, such paths do not satisfy $Dom \wedge G$.

Taking these observations into account, from *T*, there are two potential sources of divergences that must be analysed. On one hand, all the unsatisfiable nodes removed by the application of the tableau deletion rules characterise a particular kind of violation to $Dom \wedge G$. A condition that would force one to reach the inconsistent portion removed from *T*, may represent a divergence for *Dom* and *G*. We call this the *safety case.* On the other hand, given a *liveness* goal $G_i \in G$ whose eventuality is *f* (e.g., a reachability goal $G_i = \Diamond f$), an (*infinite*) path in the tableau *T* that does not pass through a state in which *f* holds, is a violation for G_i and a potential source of divergence between the goals. We call this the *liveness case.* Why these cases are *potential* divergences and not actual divergences is further discussed later on, and analysed in the next phase. This current phase will produce a set of LTL formulas characterising potential divergences for both the safety and liveness cases.

This phase is achieved by extracting from tableau *T* a set of path conditions that characterise the two kinds of inconsistencies just described. A *path condition* is an LTL formula characterising a path in the tableau (see Section 4.2). To produce path conditions, *T* is traversed starting from the root node, collecting the information provided by the states to build LTL formulas that capture the following situations. For the safety case, the produced LTL formula captures a path that escapes from *T*, directing us into the unsatisfiable nodes (those removed from the tableau during node deletion). For the liveness case, the produced LTL formula characterises a path in *T* that avoids the states in which the eventuality of some liveness goal is fulfilled.

Path conditions are used to produce the potential boundary conditions $pBC = \{pBC_1, \ldots, pBC_m\}$, whose concrete formulation depends on whether they are safety or liveness cases. As previously mentioned, formulas in *pBC* are *potential* conflicts, because although they characterise violations for $Dom \wedge G$, they do not necessarily represent a divergence case between the goals with respect to the domain. More precisely, according to the divergence definition given in Section 2.1, we have to check if the logical inconsistency and minimality conditions are satisfied, to guarantee that these are divergences.

(3) ***BC* extraction:** This final phase consists of removing from *pBC* those conditions that do not satisfy the divergence properties in Section 2.1. The remaining boundary conditions $BC = \{BC_1, \ldots, BC_k\}$ capture weak conflicts between the goals in *G* and the domain *Dom*.

Algorithm 1 describes how the above steps are combined. We provide further details of each step below.

4.1 Generating and Refining the Tableau

LTL-TableauMethod (line 2 in Algorithm 1) implements the LTL decision procedure introduced in Section 2.3. It generates a tableau structure $T = \langle N, R, root \rangle$, which encodes *all* potential models for $Dom \wedge G$. Recall that the formula to which the tableau construction is applied (in our case $Dom \wedge G$) is satisfiable iff the root node is not removed during the application of the deletion rules. Notice that in Algorithm 1 we simply indicate that the goals and domain constraints are contradictory, when the tableau is empty (root node removed). When, on the other hand, *T* maintains some nodes after the application of the deletion rules, then $Dom \wedge G$ is *satisfiable.* In this case, the approach proceeds to generate potential boundary conditions, as described below.

4.2 Generating Potential Boundary Conditions

After generating the tableau *T*, the approach proceeds to extract from *T* a set of of LTL formulas that characterise potential boundary conditions. More precisely, functions SafetyBC and LivenessBC in Algorithm 1 generate these LTL formulas for the safety and liveness cases, respectively. In both cases, the LTL formulas are generated in two steps. First, SafetyBC computes a set of frontier path conditions that escape from *T* directing us into the unsatisfiable nodes, previously removed by the tableau method. Similarly, LivenessBC first computes a set of path conditions in *T* that avoid passing through the states in which the eventuality of some liveness goal is fulfilled. Then, given these path conditions, SafetyBC and LivenessBC generate the set of potential boundary conditions from these computed path conditions. In the remainder of this subsection, we introduce some terminology necessary to present the process for extracting potential boundary conditions from a tableau, and then we further describe the above mentioned functions.

State Constraints, Successors and Path Conditions.

Let $T = \langle N, R, root \rangle$ be a tableau structure. The state nodes of *T* (see definition 2.2) contain literals that represent the constraints that the states should satisfy to be part of a model for the formula that was queried for satisfiability.

Definition 4.1 (State Constraints). *Given a state node $s \in S_N$, the* state constraints *imposed by s, denoted by*

$CONS(s)$, are composed of the conjunction of literal formulas that s contains. Given a set $S \subseteq S_N$ of state nodes, $CONS(S)$ is defined as $\bigvee_{s \in S} CONS(s)$.

$CONS(s)$ then returns a propositional formula that characterises all states that satisfy the conditions imposed by state s. For instance, if $s = \{\Box(p \vee \neg q)^*, \Diamond r^*, p \vee \neg q^*, \neg q, r\}$, then $CONS(s) = \neg q \wedge r$, characterising the states in which the proposition q is false and r is true.

Notice that, according to definition 2.2, some nodes in N are neither states nor pre-states. For the sake of simplicity, let us ignore all these intermediate nodes, preserving the transition relation between states and pre-states in R.

DEFINITION 4.2 (SUCCESSORS). *Given two pre-state nodes $d_1, d_2 \in P_N$, $succs(d_1, d_2) = \{s \in S_N | (d_1, s) \in R \wedge (s, d_2) \in R\}$, i.e., the set of state nodes that transit from d_1 to d_2. We will use $succs(d)$ to denote all state nodes that are successors of pre-state d, i.e., $succs(d) = \{s \in S_N | (d, s) \in R\}$.*

DEFINITION 4.3 (PATH). *A* path *$\rho = (d_0, S_0); (d_1, S_1); \ldots; (d_k, S_k)$ in T is a sequence of pairs of a pre-state node and a set of state nodes such that: 1. $d_0 = root$; 2. $\forall i : 0 \leq i < k, S_i \subseteq succs(d_i, d_{i+1})$, and $S_k \subseteq succs(d_k)$; 3. $\forall i : 0 \leq i \leq k, S_i \neq \emptyset$.*
A loop-free path *is a path where $d_i \neq d_j$, for $0 \leq i < j \leq k$.*

DEFINITION 4.4 (PATH CONDITION). *A* path condition *$[\varphi_0, \ldots, \varphi_k]$ is a* sequence of propositional formulas *such that there exists a path $\rho = (d_0, S_0); (d_1, S_1); \ldots; (d_k, S_k)$ for a given tableau T, where $\varphi_i = CONS(S_i)$, for all $0 \leq i \leq k$.*

Intuitively, a path and its corresponding path condition characterise consistent ways of traversing the tableau T. Our approach will consider loop-free paths to compute path conditions for producing potential boundary conditions. The frontier path condition defined below is used for describing the paths that "cross" the frontier of the consistent part of the tableau, and lead to an inconsistent portion of it.

DEFINITION 4.5 (FRONTIER PATH CONDITION). *A* frontier path condition *$[\varphi_0, \ldots, \varphi_{k-1}, \psi]$ is a* sequence of propositional formulas *such that there exists in T a path condition $\rho = [\varphi_0, \ldots, \varphi_k]$, and $\psi = \neg\varphi_k$.*

4.2.1 Potential Safety Boundary Conditions

Function SAFETYBC given in Algorithm 2 receives as input the tableau structure $T = \langle N, R, root\rangle$ and returns a set of potential boundary conditions. First, it calls function SAFETYFPCS (line 3) to compute a set of frontier path conditions, capturing paths that "escape" from T reaching the unsatisfiable nodes removed during the tableau construction. SAFETYFPCS explores T starting from the *root* node (invocation in line 3), and traverses T using the successors definition given above (set S_i in lines 19–21), recovering path conditions (line 22) required to finally reach the inconsistent portion of the tableau (set S_k and formula ψ in lines 14–15). The algorithm keeps track of the already visited pre-states nodes, so loops are not considered (lines 11–13). Intuitively, SAFETYFPCS computes all the shortest loop-free frontier path conditions in T, that start in the *root* node and escape from T, ending in the unsatisfiable portion removed previously from T. Then, function SAFETYBC encodes each frontier path condition as an LTL formula f, by nesting each condition in the path using the next temporal operator $\bigcirc$ (line 5). The resulting formula f is used to generate a potential boundary condition $\Diamond(f)$ (line 6). Intuitively, this formula indicates that there might be a conflict in our model if condition f is eventually reached, since it would lead to reaching the inconsistent part of the tableau. Notice that by applying $\Diamond$ to f we are somehow "generalising" the inconsistency case f. This is necessary because otherwise, and due to our analysis being limited to loop-free paths, the obtained divergences would be too strong (representing divergent cases of a fixed length). This generalisation step may however make the candidate boundary condition *consistent* with the goals and domain, so we will have to later on check for inconsistency with respect to these constraints (see Section 4.4).

Algorithm 2 Receives as input a tableau $T = \langle N, R, root\rangle$ and returns a set pBC of LTL formulas, characterising potential boundary conditions.

```
 1: function SAFETYBC($T$):$pBC$
 2:     $pBC \leftarrow \emptyset$
 3:     $frontierPCs \leftarrow$ SAFETYFPCS($T, \emptyset, [\,], root$)
 4:     for all $[\varphi_1, \ldots, \varphi_k, \psi] \in frontierPCs$ do
 5:         $f \leftarrow \varphi_1 \wedge \bigcirc(\varphi_2 \wedge \ldots \wedge \bigcirc(\varphi_k \wedge \bigcirc\psi) \ldots)$
 6:         $pBC \leftarrow pBC \cup \{\Diamond(f)\}$
 7:     end for
 8:     return $pBC$
 9: end function

10: function SAFETYFPCS($T$,$V$,$\rho$,$d_\rho$):$FPC$
11:     if $V = P_N$ then return $\emptyset$
12:     else
13:         $V' \leftarrow V \cup \{d_\rho\}$
14:         $S_k \leftarrow succs(d_\rho)$
15:         $\psi \leftarrow \neg CONS(S_k)$
16:         $currFPC \leftarrow \rho + +[\psi]$
17:         $FPC \leftarrow \{\}$
18:         for all $d \in P_N - V'$ do
19:             if $succs(d_\rho, d) \neq \emptyset$ then
20:                 $S_i \leftarrow succs(d_\rho, d)$
21:                 $\varphi \leftarrow CONS(S_i)$
22:                 $\rho' \leftarrow \rho + +[\varphi]$
23:                 $FPC \leftarrow FPC \cup$ SAFETYFPCS($T, V', \rho', d$)
24:             end if
25:         end for
26:         return $FPC \cup \{currFPC\}$
27:     end if
28: end function
```

4.2.2 Potential Liveness Boundary Conditions

Function LIVENESSBC in Algorithm 3 takes as input the tableau T and a liveness goal $G_i \in G_{live}$, and returns an LTL formula that characterises different paths in T that do not guarantee the eventuality corresponding to the goal G_i. This LTL formula represents a potential divergence between goal G_i and the other goals and domain constraints. LIVENESSBC starts by identifying the nodes in T in which the eventuality required by G_i is fulfilled. That is, if G_i is a reachability goal $\Diamond f$ or a response goal $\Box(g \to \Diamond f)$, then $\Diamond f$ is the eventuality that G_i should fulfil. Then, sentence $E \leftarrow Eventualities(T, G_i)$ in line 2 will store in E the set of all nodes from T that contain formula f (i.e., each node in E satisfies the eventuality demanded by goal G_i). For instance, if $G_i = \Diamond p$, then all nodes from T that contain proposition p, will be in the set E.

After that, LIVENESSFPCS computes all the loop-free path conditions in a way similar to that for the safety case. However, for the liveness case, the path conditions computed avoid those paths that pass through the nodes in E. Then, the main difference with the safety case is the composition of the sets S_i and S_k in Algorithm 3. In particular, set S_i only considers state nodes that transition to the next pre-state node, but that are not included in the set E (line 21). The set S_k only contains the nodes that are in E (line 18), thus ψ (i.e.,$\neg CONS(S_k) \wedge CONS(succs(d_\rho) - E)$) characterises the nodes that do not satisfy the eventuality (line 19).

LivenessFPCs then returns the shortest loop-free path conditions that avoid reaching nodes in which the eventuality required by G_i holds. In line 6, a formula f characterising each path condition, is generated. The disjunction of all these formulas captures all the different ways of traversing the tableau without hitting a state in which the eventuality of goal G_i is satisfied (line 7). If the goal is a reachability one, of the form $G_i = \Diamond f$, and the computed path condition is FC, then $\Box(FC)$ is the potential divergence (line 12). This LTL formula indicates that there is a conflict if the condition FC always holds, preventing the eventuality f required by G_i from being fulfilled. If, on the other hand, the goal is a response one, of the form $G_i = \Box(f_1 \rightarrow \Diamond f_2)$, and the computed path condition is FC, then $\Diamond(f_1 \wedge \Box(FC))$ is the potential divergence (line 10). This LTL formula indicates that there is a conflict if f_1 holds at some point, but the eventuality f_2 is never fulfilled. Notice that f_2 may be satisfied before the condition f_1 holds, but it will not be satisfied an infinite number of times. The case of a progress goal $\Box\Diamond f_2$ is the same as for response, simply taking $f_1 = true$. Notice again that, in both cases, by putting FC in the scope of $\Box$ we are generalising the divergent case. Intuitively, formula $\Box FC$ tries to characterise the *infinite* paths in the tableau that do not satisfy the G_i's eventuality.

Despite the fact that the generated LTL formulas capture conflicts with $Dom \wedge G$, these may be discarded in the following phase because they may not meet all the requirements to be considered divergences, e.g., the minimality condition.

Algorithm 3 Takes a tableau $T = \langle N, R, root\rangle$ and a liveness goal G_i, and returns a potential boundary condition pBC.

```
1:  function LivenessBC($T$,$G_i$):$pBC$
2:      $E \leftarrow Eventualities(T, G_i)$
3:      $FC \leftarrow false$
4:      $PCs \leftarrow$ LivenessFPCs($T, E, \emptyset, [\,], root$)
5:      for all $[\varphi_1, \ldots, \varphi_k, \psi] \in frontierPCs$ do
6:          $f \leftarrow \varphi_1 \wedge \bigcirc(\varphi_2 \wedge \ldots \wedge \bigcirc(\varphi_k \wedge \bigcirc\psi)\ldots)$
7:          $FC \leftarrow FC \vee f$
8:      end for
9:      if $G_i = \Box(\psi \rightarrow \Diamond\varphi)$ then
10:         return $\Diamond(\psi \wedge \Box(FC))$          ▷ $G_i$ is a response goal.
11:     else
12:         return $\Box(FC)$          ▷ $G_i$ is a reachability goal.
13:     end if
14: end function

15: function LivenessFPCs($T$,$E$,$V$,$\rho$,$d_\rho$):$PC$
16:     ▷ same algorithm as SafetyFPCs, except the computation of
        sets $S_i$ and $S_k$.
17:     . . .
18:     $S_k \leftarrow succs(d_\rho) \cap E$
19:     $\psi \leftarrow \neg CONS(S_k) \wedge CONS(succs(d_\rho) - E)$
20:     . . .
21:     $S_i \leftarrow succs(d_\rho, d) - E$
22:     $\varphi \leftarrow CONS(S_i)$
23:     . . .
24: end function
```

4.3 Filtering Boundary Conditions

Continuing with Algorithm 1, after computing the set pBC of potential boundary conditions for the safety and liveness cases, function *FilterDivergences*(*Dom*, G, pBC) is concerned with checking which ones represent actual divergences. The previous phase produces potential boundary conditions pBC, that may fail to be actual boundary conditions because of various reasons: *(i)* they might be consistent with the goals and domain constraints; *(ii)* they might not be minimal; or *(iii)* they might be the negation of a goal. Situation *(i)* can arise because, despite the fact that we compute path conditions that indeed contradict the goals and domain specification, since such path conditions are loop-free we "generalise" them (weaken them) by applying a temporal operator; such generalisation may make some potential boundary conditions consistent with the goals and domain constraints. Situations *(ii)* and *(iii)* are more clear: these impose conditions on our potential boundary conditions that the process to generate them does not take into account, and thus they must be checked afterwards. We then have to go through an additional process, represented by function *FilterDivergences* that for each LTL formula $bc \in pBC$, checks whether bc meets the conditions described in Section 2.1, that define divergences (i.e., inconsistency with goals and domain specification, minimality with respect to the set G of goals, and that bc is not a trivial boundary condition). *FilterDivergences* checks inconsistency with goals and domain specification by assessing the satisfiability of $bc \wedge G \wedge Dom$ using an LTL satisfiability procedure; minimality, i.e., that for each goal $G_i \in G$, $Dom \wedge \bigwedge_{j \neq i} G_j \wedge bc$ is satisfiable, is also checked using LTL satisfiability; finally, checking that $bc \neq \neg G$ is just a syntactical check. *FilterDivergences* discards those LTL formulas in pBC that do not meet the above conditions, and returns a set $BC \subseteq pBC$ of boundary conditions for the goals G in the domain *Dom*.

FilterDivergences needs, for each potential boundary condition, to perform as many calls to the LTL decision procedure as goals are in G (to check the minimality condition). To efficiently perform all these checks, we use *Aalta* [27], an efficient LTL satisfiability checker, recently developed. The experimental evaluation in Section 5 shows that the time required by *FilterDivergences* is small, compared to the time required to build the tableau.

4.4 Correctness and (In)completeness

Let us discuss now termination, correctness and (in)completeness of our approach. Regarding termination, recall that in Section 2.3 we explained that the tableau generation process is guaranteed to terminate, and the structure generated is a finite graph. So, the tableau contains a finite number of pre-state and state nodes. Since the functions to compute the path conditions, SafetyFPCs and LivenessFPCs, consider only loop-free paths over a finite graph, these are finitely many, and thus SafetyFPCs and LivenessFPCs terminate. All the checks performed in the last phase of the approach, *FilterDivergences*, are made using the LTL satisfiability checker *Aalta* [27] (that is guaranteed to terminate), for a finite number of potential boundary conditions. Thus, the whole process is guaranteed to terminate.

Regarding the *correctness* of the approach, i.e., that if the process produces a formula bc this is indeed a boundary condition, notice that the last phase of our approach, *FilterDivergences*, checks that bc satisfies all the conditions of the definition of boundary condition (see Section 2.1), thus guaranteeing correctness.

Regarding *completeness*, i.e., that if there exists a divergence situation between the goals and the domain our process is able to produce it, the situation is different. Unfortunately our approach is not complete. Let us provide an example. Consider again goals $\Box \bigcirc p$ and $\Diamond \neg p$, used in Section 2.3 to illustrate a tableau. As we mentioned, these goals can be satisfied only when the eventuality $\neg p$ is fulfilled in the first state, and in the rest of the trace p holds. So, if p holds in the initial state, the goals diverge. In fact, p is indeed a boundary condition, it meets the three properties defining it in Section 2.1. However, our approach cannot compute it, since we only produce formulas that contain at least one temporal operator. In fact, our approach will produce formulas $\Box p$, $\Diamond p$ and $\Diamond(\neg p \wedge \bigcirc \neg p)$ as potential boundary conditions, none of which satisfy the boundary condition definition and thus are removed by *FilterDivergences*.

5. EVALUATION

In this section we evaluate our proposal, addressing the following research questions:

RQ1 *Is our approach well suited to detect divergences in goal specifications?*

RQ2 *Are the boundary conditions computed by our approach more general than those derived by related techniques?*

RQ3 *Does our approach apply to specifications that cannot be handled by related techniques?*

To answer RQ1, we take various case studies from the literature on formal requirements specifications, that feature both safety and liveness goals, and evaluate our technique for computing divergences. Section 5.1 reports the results of analysing two case studies: the Elevator Controller [9] and the Rail Road Crossing System [5]. To answer RQ2, in Section 5.2 we briefly introduce the pattern-based approach to goal conflict detection from [44], and compare divergences computed using our approach against those obtained using the pattern-based one. To answer RQ3, in Section 5.3 we study a simplified version of the TCP protocol, whose goals do not correspond to any of the patterns of the previous approach from [44], to assess how our approach deals with them. Finally, in Section 5.4 we provide further examples, and discuss about the scalability of our approach and the succinctness of the computed boundary conditions.

These research questions are answered using a tool that we developed, that implements our tableau based goal conflict detection approach. The tableau generation and potential boundary conditions computation is implemented in Haskell, and employs a BDD library [6] to simplify expressions when path conditions are collected, while computing boundary conditions. Moreover, the tool integrates the LTL satisfiability checker *Aalta* [27], to efficiently perform all the SAT checks required by the *FilterDivergences* phase.

The tool, the specifications for all case studies, and a description of how to reproduce the experiments can be found in http://dc.exa.unrc.edu.ar/staff/rdegiovanni/ase2016. All the experiments were run on an Intel Core i5 4460 processor, 3.2Ghz, with 8Gb of RAM, running GNU/Linux (Ubuntu 15.04). For each case study we report the size of the constructed tableau (number of nodes and transitions), the number of computed potential boundary conditions, the number of resulting divergences, and the analysis time required for constructing the tableau and for the entire process. This information is summarised in Section 5.4.

5.1 Case Studies

We evaluate two case studies taken from the literature, the Elevator Controller introduced in [9], and a simplified version of the Rail Road Crossing System taken from [5].

5.1.1 Elevator Controller

The Elevator Controller [9] has one sensor *call* to detect when a user has called the elevator, and one sensor *atfloor* that is set to true when the elevator reaches the floor where the user is waiting. In addition, the controller sends an *open* signal indicating the door must open. For simplification, we assume there is only one user that may call the lift at a time. The following goal and domain properties are elicited:

Goal: Achieve[*OpenWhenCall*]
FormalDef: $\Box(call \rightarrow \Diamond(open))$

Domain: Maintain[*DoorOpensWhenAtFloor*]
FormalDef: $\Box(\bigcirc(open) \rightarrow atfloor)$

The goal *OpenWhenCall* indicates that the controller should respond to the user's calling by opening the door. The domain property *DoorOpensWhenAtFloor* captures the door's behaviour, indicating the door opens only when the elevator reaches the floor where the user is waiting. Our approach computes the following 2 potential boundary conditions:

1. $\Diamond(call \wedge \neg atfloor \wedge \neg open \wedge \bigcirc(open))$
2. $\Diamond(call \wedge \Box(\neg open \vee (call \wedge \neg atfloor \wedge \neg open \wedge \bigcirc(\neg open))))$

Condition 1 is computed for the safety case, while condition 2 is for the liveness goal. Only condition 2 meets the definition of divergence (condition 1 is inconsistent with the domain because the door will open in the next state, but the elevator is not at the floor where the user called it). This formula captures the scenarios in which the elevator has been called but the elevator will not be at the floor where it was called, and it will never open the door. The whole process to compute this boundary condition takes 0.80 seconds.

5.1.2 The Rail Road Crossing System

Consider now a simplified Rail Road Crossing System [5]. In this model, a train can approach and enter a crossing, captured by *ta* and *tc*, respectively. A car may also approach and enter the crossing, captured by *ca* and *cc*, respectively. The crossing gate may be opened (go) or closed ($\neg go$). The following goals and domain properties are elicited:

Goal: Avoid[*Collision*]
FormalDef: $\Box\neg(tc \wedge cc)$

Goal: Maintain[*ClosedGateWhenTrainApproaching*]
FormalDef: $\Box(ta \rightarrow \neg go)$

Domain property: *TrainsNotStop*
FormalDef: $\Box(\bigcirc(tc) \leftrightarrow ta)$

Domain property: *CarsCrossWhenGateIsOpened*
FormalDef: $\Box(\bigcirc(cc) \rightarrow ca \wedge go)$

Our tool computes 5 potential boundary conditions, with only one being a divergence, in 0.5 seconds. The identified divergence is the following:

1. $\Diamond((\neg cc \wedge go \wedge ta) \vee (cc \wedge (\neg go \wedge tc \vee go \wedge (ta \vee tc))))$

This boundary condition reveals a few dangerous situations. A conflict arises if the gate is open when the train is approaching, and the car has not crossed yet. Other similar conflicting situation arise when the car is crossing at the same time as the train is approaching or crossing.

5.2 Comparison with Divergence Patterns

We now compare our technique with the only previous formal approach to derive boundary conditions, presented in [44]. This previous approach requires matching goals against a set of pre-defined divergence patterns, for which divergence expressions are provided. To answer RQ2, we compare these divergences with those computed by our approach, when fed with the pre-defined patterns from [44]. Figure 3 summarises the three divergence patterns presented in [44], in which the goals and domain are specified in LTL.

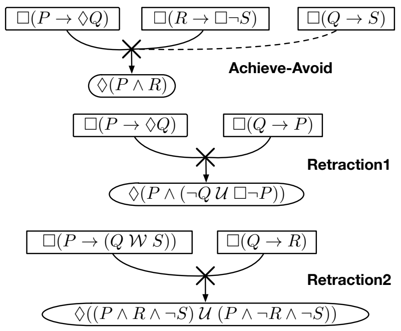

$\Box(P \rightarrow \Diamond Q)$
$\Box(R \rightarrow \Box\neg S)$
$\Box(Q \rightarrow S)$
Achieve-Avoid
$\Diamond(P \wedge R)$
$\Box(P \rightarrow \Diamond Q)$
$\Box(Q \rightarrow P)$
Retraction1
$\Diamond(P \wedge (\neg Q\, \mathcal{U}\, \Box\neg P))$
$\Box(P \rightarrow (Q\, \mathcal{W}\, S))$
$\Box(Q \rightarrow R)$
Retraction2
$\Diamond((P \wedge R \wedge \neg S)\, \mathcal{U}\, (P \wedge \neg R \wedge \neg S))$

Figure 3: Divergence Patterns from [44].

In the case of the Achieve-Avoid pattern, the following potential boundary conditions are produced, resulting all in divergences:

1. $\Diamond(\neg P \wedge (\neg Q \wedge R \wedge S \vee Q \wedge (R \vee \neg S)) \vee P \wedge (\neg Q \wedge R \vee Q \wedge (R \vee \neg S)))$
2. $\Diamond(\neg P \wedge \neg Q \wedge R \wedge \neg S \wedge \bigcirc(P \vee Q \vee S))$
3. $\Diamond(P \wedge \neg Q \wedge \neg R \wedge \bigcirc(\neg Q \wedge R \vee Q \wedge (R \vee \neg S)))$
4. $\Diamond(P \wedge \Box((P \wedge \neg Q \wedge \neg R) \vee (\neg P \wedge \neg Q \wedge (\neg R \vee \neg S)) \vee (P \wedge \neg Q \wedge \neg R \wedge \bigcirc(\neg Q \wedge \neg R)) \vee (\neg P \wedge \neg Q \wedge R \wedge \neg S \wedge \bigcirc(\neg P \wedge \neg Q \wedge \neg S))))$

The entire process for computing and filtering the boundary conditions takes 0.43 seconds. Boundary conditions 1–3 are computed for the safety case, while the last condition is for the liveness goal. When we compare the pattern-based derived boundary condition $\Diamond(P \wedge R)$ with those computed with our technique, we observe that the former *implies* our boundary condition 1 (but not vice versa). That is, boundary condition 1 is *more general* than that derived by the pattern. The mentioned implication has been verified using a simple LTL satisfiability check (f_1 implies f_2 iff $f_1 \wedge \neg f_2$ is unsatisfiable). From the point of view of succinctness, clearly our boundary conditions, computed from loop-free path conditions in the tableau, are more complex and less readable than that derived by the pattern.

When applying our technique to the Retraction1 pattern, the approach computes the following potential boundary conditions, where 1 and 2 correspond to the safety case, and 3 to the liveness case:

1. $\Diamond(\neg P \wedge Q)$ *2.* $\Diamond(P \wedge \neg Q \wedge \bigcirc(\neg P \wedge Q))$
3. $\Diamond(P \wedge \Box((\neg P \vee \neg Q) \vee (P \wedge \neg Q \wedge \bigcirc(\neg P \vee \neg Q))))$

The first two are discarded because they do not satisfy the minimality condition, while formula 3 is identified as a divergence. The whole boundary condition computation takes just 0.28 seconds. Again, our computed boundary condition is not as succinct as the pattern-based one. And also as in the previous pattern, our computed boundary condition is *more general* than the pattern-based derived one (the pattern based boundary condition implies formula 3).

When applying our technique to the Retraction2 pattern, the tool computes the following potential boundary condition, which is a divergence:

1. $\Diamond((\neg P \wedge Q \wedge \neg R) \vee (P \wedge (\neg Q \wedge \neg S \vee Q \wedge \neg R)))$

The whole divergence computation takes 0.48 seconds. Since this pattern does not consider liveness goals, the boundary conditions were computed for the safety case only. As with the previous patterns, formula 1 is implied by that derived by the Retraction2 pattern, and consequently, our approach is again *more general*.

Notice that for the Achieve-Avoid pattern our approach is able to produce boundary conditions that are not identified by the pattern. Thus, these characterise additional divergent cases, that can be very useful to engineers when analysing conflicting situations in goal specifications.

5.3 An example not captured by patterns

Consider the TCP network protocol, which provides reliable in-order delivery of packets in packet based data transmission. For simplification, let us assume that the protocol can send one packet at a time, i.e., it waits for an acknowledgement (ack) before sending the next packet. Briefly, the following liveness goals are elicited for this protocol:

Goals: Achieve[$\mathit{DeliveredWhenSent}$]
FormalDef: $\Box(\mathit{send} \rightarrow (\neg \mathit{ack}\ \mathcal{U}\ \mathit{delivered}))$

Goals: Achieve[$\mathit{WaitACKBeforeSendAgain}$]
FormalDef: $\Box(\mathit{delivered} \rightarrow (\neg \mathit{send}\ \mathcal{U}\ \mathit{ack}))$

Notice that this example cannot be matched to any of the above patterns. Our technique is able to analyse this specification, and computes, in 1.31 seconds, the following divergences:

1. $\Diamond(\mathit{send} \wedge \Box((\neg \mathit{ack} \wedge (\neg \mathit{delivered} \vee \mathit{send}) \vee \mathit{ack} \wedge \neg \mathit{delivered}) \vee (\neg \mathit{ack} \wedge \neg \mathit{delivered} \wedge \mathit{send} \wedge \bigcirc(\neg \mathit{ack} \wedge (\neg \mathit{delivered} \vee \mathit{send}) \vee (\mathit{ack} \wedge \neg \mathit{delivered})))))$
2. $\Diamond(\mathit{delivered} \wedge \Box(\neg \mathit{ack} \vee (\neg \mathit{delivered} \wedge \mathit{send}) \vee (\neg \mathit{ack} \wedge \neg \mathit{delivered} \wedge \mathit{send} \wedge \bigcirc(\neg \mathit{ack} \vee \neg \mathit{delivered}))))$

The first boundary condition evidences a divergence if, from a certain point onwards, after sending a packet, either the ack signal is never received, or it is received before the packet has been delivered. The second boundary condition indicates that the goals are divergent when a packet was delivered, but its corresponding ack is never received, or a new packet is sent before receiving the ack.

5.4 Discussion

Let us briefly discuss about the scalability of our approach and the readability of our computed boundary conditions. Table 1 summarises for each case study the size of the constructed tableau (number of nodes and transitions), the number of computed potential boundary conditions (pBCs) and the resulting divergences (BCs), the analysis time required for constructing the tableau and for the entire process (in seconds). Moreover, to assess the readability of the BCs computed, we evaluate the succinctness of the boundary condition computed for each case study (we choose the biggest one, when multiple pBCs are computed). We measure the number of literals (Lit.), and the number of logical and temporal operators involved in the formula (L.Op. and T.Op., resp.). We also consider additional, more complex, specifications, namely, the ATM [42], Telephone [13], and London Ambulance Service (LAS) [14] (3, 5 and 5 formulas, resp.).

Table 1: Scalability and Succinctness Summary

Case Study	Tableau nodes/trans	pBCs /BCs	BC Succinctness			Tableau Time	Total Time
			Lit.	L.Op.	T.Op.		
Elevator	24 / 48	2 / 1	6	8	3	0.64	0.80
RRCS	53 / 120	5 / 1	9	10	1	0.31	0.50
TCP	34 / 86	2 / 2	14	20	3	0.92	1.31
ATM	72 / 206	4 / 3	13	16	3	2.08	2.71
Telephone	163 / 508	4 / 1	13	19	2	8.84	11.43
LAS	93 / 184	1 / 1	42	58	1	11.58	11.68
Achieve-Avoid	34 / 86	4 / 4	17	28	4	0.27	0.43
Retraction1	12 / 24	3 / 1	7	11	3	0.02	0.20
Reatraction2	30 / 66	1 / 1	8	12	1	0.45	0.48

Table 1 shows that almost all the analysis time is spent in constructing the tableau. We observe that this is more evident in the specifications that contain more liveness goals, as the TCP, ATM and Telephone case studies. This is due to the fact that removing the nodes that do not satisfy eventualities requires multiple visits to the tableau. Although we need to perform a more thorough scalability evaluation, the efficiency of our approach is promising, at least when analysing specifications that involve only safety goals, or a restricted number of liveness goals.

Table 1 also shows that the computed boundary conditions involve a considerable number of literals, logical and temporal operators, hindering their readability. This is, evidently, one of the most important issues we need to overcome to help the engineer in understanding the divergences computed and resolving the goal models.

6. RELATED WORK

Various informal and semi-formal approaches have been proposed for detecting conflicts in requirements models [19, 23, 24]. In addition, formal methods for detecting inconsistency have also been proposed in [41, 21, 18, 11, 34, 10]. These approaches focus on logical inconsistencies only, or ontology mismatch. Other techniques for reasoning about inconsistencies based on abduction such as [36] consider generating explanations (in the form of conjunctions of ground literals) for strong inconsistencies in requirements expressed in quasi-classical logic. Our approach focuses on detecting weak inconsistencies and automatically generating general LTL expressions that characterise situations in which inconsistencies may arise. Techniques such as [18], perform consistency checks for requirements expressed as conditional scenarios as a precursor for model synthesis. These consistency checks would identify inconsistency between two enabled charts. Our work on the other hand focuses on finding a characterisation for enabling conditions that could lead to an inconsistency. Conflicts in non-functional requirements have also been considered in [28, 29, 22]. For instance, [29] proposes a catalogue of conflicts, categorising these (pairwise) according to the frequency in which they occur together.

In [13], an approach for detecting conflicts between feature descriptions expressible in LTL is presented. It focuses on strong conflicts (i.e., mutually inconsistent features), applies to features of the form $p \rightarrow q\,\mathcal{U}\,(r \vee d)$, and uses a model checker for the conflict check. Our approach focuses on identifying weak conflicts, characterising them by boundary conditions, and is applicable to a wider class of LTL formulas.

The technique presented in [3] combines model checking and machine learning to automatically generate a set of obstacle conditions with respect to goals and domain properties expressed in LTL. It supposes however that each goal is satisfiable within its domain, and focuses on identifying the conditions under which the goal and domain may be inconsistent. It does not handle situations that arise because the goals themselves are inconsistent with one another.

The resolution of conflicts has been the subject of recent work, e.g., [12, 32]. The technique proposed in [32] makes use of argumentation patterns to elicit, compose and relate stakeholders beliefs. The technique assumes conflicts have already been elicited, and calculates an inconsistency score between beliefs and goals (described informally). These scores are then used as a guide for engineers as to which goals to resolve. Approaches like [40] propose methods that generate consistency specifications by construction, eliminating the need for detection. The problem of detecting inconsistency in specifications is related to that of vacuity detection [26, 17]. The latter may be a result of the former though inconsistency detection cannot be reduced to a vacuity check. It is also somewhat related to that of detecting overconstrained specifications. For instance, [39] proposes an algorithm for extracting a core set of assertions that cause an inconsistency in an Alloy model. The method assumes that inconsistency is already known to exist. Our method however attempts to find the assertion (i.e., BC) that would lead to the inconsistency. It also relates to that of realisability of specifications, e.g., [38, 7], in that conflicting goals may lead to unrealisable requirements.

7. CONCLUSION AND FUTURE WORK

Detecting inconsistencies early on in the requirements engineering process helps avoiding costly software repairs, and also supports systematic requirements' elicitation and verification activities. This paper presents a novel approach for detecting divergences in goal specifications, and generating boundary conditions that characterise the cases under which a strong inconsistency could potentially arise. To the best of our knowledge, our approach is the first fully automated technique for computing goal conflicts that applies both to safety and to a wide range of liveness goals, specified in LTL. Our tableau-based method is guaranteed to produce correct results and to terminate. Our approach computes more general formulas than previous, related techniques. However, as our boundary conditions are automatically produced from tableau paths that lead to inconsistencies, they are often longer and less comprehensible in some cases than the more compact ones derived from patterns.

The presented work opens various lines for future work. For instance, a source of incompleteness of our approach is related to the way we interpret divergent tableau paths, applying a kind of generalisation through the application of temporal operators. We are studying alternative, stronger ways of dealing with these divergent paths, to be able to identify more boundary conditions. We are also considering the use of inductive learning techniques to support the inference of more generalised expressions.

Our approach relies on a tableaux construction; we plan to study more efficient tableau procedures, e.g., [16], to increase our technique's efficiency. We also plan to exploit tableaux structures for other problems related to requirements specifications, such as obstacle condition identification [45].

8. REFERENCES

[1] IEEE recommended practice for software requirements specifications. *IEEE Std 830-1998*, pages 1–40, 1998.

[2] Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastián Uchitel. Learning operational requirements from goal models. In *Proceedings of the 31st Intl. Conf. on Software Engineering*, ICSE '09, pages 265–275, Washington, DC, USA, 2009. IEEE Computer Society.

[3] Dalal Alrajeh, Jeff Kramer, Axel van Lamsweerde, Alessandra Russo, and Sebastián Uchitel. Generating obstacle conditions for requirements completeness. In *ICSE*, pages 705–715, 2012.

[4] Ana I. Anton. *Goal Identification and Refinement in the Specification of Software-based Information Systems*. PhD thesis, Atlanta, GA, USA, 1997. UMI Order No. GAX97-35409.

[5] Adrian Beer, Stephan Heidinger, Uwe Kühne, Florian Leitner-Fischer, and Stefan Leue. Symbolic causality checking using bounded model checking. In *Proc. of the 22nd Intl. Sym. on Model Checking Software*, pages 203–221, 2015.

[6] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[7] A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev. Diagnostic information for realizability. In *Proc. of the 9th Intl. Conf. on Verification, Model Checking, and Abstract Interpretation*, pages 52–67, 2008.

[8] Renzo Degiovanni, Dalal Alrajeh, Nazareno Aguirre, and Sebastián Uchitel. Automated goal operationalisation based on interpolation and sat solving. In *ICSE*, pages 129–139, 2014.

[9] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.

[10] Christian Ellen, Sven Sieverding, and Hardi Hungar. Detecting consistencies and inconsistencies of pattern-based functional requirements. In *Proc. of the 19th Intl. Conf. on Formal Methods for Industrial Critical Systems*, pages 155–169, 2014.

[11] Neil A. Ernst, Alexander Borgida, John Mylopoulos, and Ivan J. Jureta. Agile requirements evolution via paraconsistent reasoning. In *Proc. of the 24th Intl. Conf. on Advanced Information Systems Engineering*, pages 382–397, 2012.

[12] Alexander Felfernig, Gerhard Friedrich, Monika Schubert, Monika Mandl, Markus Mairitsch, and Erich Teppan. Plausible repairs for inconsistent requirements. In *IJCAI*, pages 791–796, 2009.

[13] Amy P. Felty and Kedar S. Namjoshi. Feature specification and automated conflict detection. *ACM TOSEM*, 12(1):3–27, 2003.

[14] A. Finkelstein and J. Dowell. A comedy of errors: The london ambulance service case study. In *Proceedings of the 8th International Workshop on Software Specification and Design*, IWSSD '96, pages 2–, Washington, DC, USA, 1996. IEEE Computer Society.

[15] Paolo Giorgini, John Mylopoulos, and Roberto Sebastiani. Goal-oriented requirements analysis and reasoning in the tropos methodology. *Engineering Applications of Artificial Intelligence*, 18(2):159 – 171, 2005.

[16] Valentin Goranko, Angelo Kyrilov, and Dmitry Shkatov. Tableau tool for testing satisfiability in ltl: Implementation and experimental analysis. *Electronic Notes in Theoretical Computer Science*, 262:113–125, 2010.

[17] Arie Gurfinkel and Marsha Chechik. Robust vacuity for branching temporal logic. *ACM Trans. on Computational Logic*, 13(1):1–32, 2012.

[18] David Harel, Hillel Kugler, and Amir Pnueli. Synthesis revisited: Generating statechart models from scenario-based requirements. In *Formal Methods in Software and Systems Modeling: Essays Dedicated to Hartmut Ehrig on the Occasion of His 60th Birthday*, pages 309–324, 2005.

[19] J.H. Hausmann, R. Heckel, and G. Taentzer. Detection of conflicting functional requirements in a use case-driven approach. In *ICSE*, pages 105–115, 2002.

[20] Sebastian J.I. Herzig and Christiaan J.J. Paredis. A conceptual basis for inconsistency management in model-based systems engineering. *Procedia CIRP*, 21:52 – 57, 2014.

[21] Anthony Hunter and Bashar Nuseibeh. Managing inconsistent specifications: Reasoning, analysis, and action. *ACM TOSEM*, 7(4):335–367, 1998.

[22] I. J. Jureta, A. Borgida, N. A. Ernst, and J. Mylopoulos. Techne: Towards a new generation of requirements modeling languages with goals, preferences, and inconsistency handling. In *Proc. of the 18th IEEE International Requirements Engineering Conference*, pages 115–124, 2010.

[23] M. Kamalrudin. Automated software tool support for checking the inconsistency of requirements. In *ASE*, pages 693–697, 2009.

[24] Massila Kamalrudin, John Hosking, and John Grundy. Improving requirements quality using essential use case interaction patterns. In *ICSE*, pages 531–540, 2011.

[25] 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.

[26] Orna Kupferman and Moshe Y. Vardi. Vacuity detection in temporal model checking. In *Proc. of the 10th IFIP WG10.5 Advanced Research Working Conf. on Correct Hardware Design and Verification Methods*, pages 82–98, 1999.

[27] Jianwen Li, Yinbo Yao, Geguang Pu, Lijun Zhang, and Jifeng He. Aalta: an LTL satisfiability checker over infinite/finite traces. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 731–734, 2014.

[28] C. L. Liu. Ontology-based conflict analysis method in non-functional requirements. In *Proc. of the 9th IEEE/ACIS Intl. Conf. on Computer and Information Science*, pages 491–496, 2010.

[29] Dewi Mairiza and Didar Zowghi. Constructing a catalogue of conflicts among non-functional requirements. In *Proc. of the Intl. Conf. Evaluation of Novel Approaches to Software Engineering*, pages 31–44, 2011.

[30] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[31] Zohar Manna and Pierre Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 6(1):68–93, January 1984.

[32] P.K. Murukannaiah, A.K. Kalia, P.R. Telangy, and M.P. Singh. Resolving goal conflicts via argumentation-based analysis of competing hypotheses. In *Proc. 23rd IEEE Int. Requirements Engineering Conf.*, pages 156–165, 2015.

[33] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Trans. Softw. Eng.*, 18(6):483–497, June 1992.

[34] Tuong Huan Nguyen, Bao Quoc Vo, Markus Lumpe, and John Grundy. KBRE: a framework for knowledge-based requirements engineering. *Software Quality Journal*, 22(1):87–119, 2013.

[35] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Leveraging inconsistency in software development. *Computer*, 33(4):24–29, 2000.

[36] Bashar Nuseibeh and Alessandra Russo. Using abduction to evolve inconsistent requirements specification. *Australasian Journal of Information Systems*, 6(2), 1999.

[37] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[38] Suchismita Roy, Sayantan Das, Prasenjit Basu, Pallab Dasgupta, and P. P. Chakrabarti. Sat based solutions for consistency problems in formal property specifications for open systems. In *CAD*, pages 885–888, 2005.

[39] Ilya Shlyakhter, Robert Seater, Daniel Jackson, Manu Sridharan, and Mana Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. In *ASE*, pages 94–105, 2003.

[40] Monique Snoeck, Cindy Michiels, and Guido Dedene. Consistency by construction: The case of merode. In *Proc. of the ER Workshop on Conceptual Modeling for Novel Application Domains*, pages 105–117, 2003.

[41] George Spanoudakis and Anthony Finkelstein. Reconciling requirements: a method for managing interference, inconsistency and conflict. *Annals of Software Engineering*, 3(1):433–457, 1997.

[42] Sebastián Uchitel, Jeff Kramer, and Jeff Magee. Synthesis of behavioral models from scenarios. *IEEE Trans. Software Eng.*, 29(2):99–115, 2003.

[43] Axel van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications.* Wiley, 2009.

[44] Axel van Lamsweerde, Emmanual Letier, and Robert Darimont. Managing conflicts in goal-driven requirements engineering. *IEEE Trans. Softw. Eng.*, 24(11):908–926, November 1998.

[45] Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Trans. Softw. Eng.*, 26(10):978–1005, October 2000.

[46] Pierre Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, (110–111):119–136, 1985.

[47] Eric S. K. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, RE '97, pages 226–, Washington, DC, USA, 1997. IEEE Computer Society.