Leviathan: A New LTL Satisfiability Checking Tool Based on a One-Pass Tree-Shaped Tableau

Matteo Bertello, Nicola Gigante, Angelo Montanari
University of Udine, Italy
bertello.matteo@spes.uniud.it
gigante.nicola@spes.uniud.it
angelo.montanari@uniud.it

Mark Reynolds
University of Western Australia, Australia
mark.reynolds@uwa.edu.au

Abstract

The paper presents Leviathan, an LTL satisfiability checking tool based on a novel one-pass, tree-like tableau system, which is way simpler than existing solutions. Despite the simplicity of the algorithm, the tool has performance comparable in speed and memory consumption with other tools on a number of standard benchmark sets, and, in various cases, it outperforms the other tableau-based tools.

1 Introduction

Linear Temporal Logic (LTL) is a modal logic useful to express properties of systems that evolve over time [Pnueli, 1977]. Prominently, in the field of automatic verification of computer programs and circuit designs, LTL has become a de-facto standard language to express desired or forbidden properties of safety-critical systems [Clarke *et al.*, 1999]. In this context, the *model checking* problem is the primary challenge: to decide whether a system, represented through the set of its possible computation paths, satisfies or not a given property expressed by an LTL formula. However, correctly stating the desired properties as logical formulae is not a trivial task, and it is important to perform sanity checks to at least avoid common pitfalls. For this reason, the *satisfiability checking* problem is important as well, and it has recently regained attention, *e.g.*, [Goranko *et al.*, 2010; Rozier and Vardi, 2010; Vardi *et al.*, 2013]. Interesting applications of LTL satisfiability checking arise also in the area of AI. As an example, in automated planning problems, LTL has long since been used as a modeling language to express temporal goals [Bacchus and Kabanza, 1998], and some authors have investigated its applicability as a direct solving strategy [Kress-Gazit *et al.*, 2009; Cialdea Mayer *et al.*, 2007].

The LTL satisfiability problem has been shown to be PSPACE-complete in [Sistla and Clarke, 1985]. Despite the high complexity, the problem can be solved sufficiently fast in practice by a number of different techniques. The first proposed approach was a graph-shaped, two-pass tableau-based procedure by Wolper [1985], later refined by various authors (see, *e.g.*, [Kesten *et al.*, 1993; Manna and Pnueli, 1995]). An alternative tableau system was proposed by Schwendimann [1998], which differs from previous systems by having a tree-like shape and requiring only a one-pass construction, thus being faster than previous proposals [Goranko *et al.*, 2010].

As a matter of fact, LTL satisfiability checking tools based on different techniques have been recently developed. Tools that can solve LTL satisfiability by reducing it to model checking, such as NuSMV [Cimatti *et al.*, 2002], have been surveyed in [Rozier and Vardi, 2010; Vardi *et al.*, 2013]. The satisfiability of an LTL formula can also be reduced to the emptiness of a corresponding *Büchi automaton*, an approach adopted by the recently developed tool Aalta [Li *et al.*, 2014]. Another important approach to LTL satisfiability is *temporal resolution*, which was pioneered by [A. R. Cavalli and L. Fariñas del Cerro, 1984; Venkatesh, 1985], more recently employed by Fisher *et al.* [2001], and which is at the core of the *labeled superposition* technique proposed by Suda and Weidenbach [2012]. The performance of most of these different techniques have been carefully compared in [Schuppan and Darmawan, 2011].

In this paper, we present Leviathan, an LTL satisfiability checking and model building tool, whose full source code is available online under a liberal open source license. It is based on a one-pass tree-shaped tableau, which is much simpler to state, to understand, to explain, and to implement than previous LTL tableaux systems, including Schwendimann's one. The tableau has been implemented in the C++ language with speed and memory usage in mind, and extended tests have been done to compare its performance with other tools from a broad portfolio of different techniques. The results show that, despite the greater simplicity of the underlying tableau system, Leviathan's performance is often comparable, both regarding speed and memory usage, with existing tools. Although a naive implementation would have reflected the algorithm's simplicity, the question of how to obtain the most performant implementation resulted to be far from trivial. For this reason, we include a detailed description of the design choices that underlie our implementation.

The paper is structured as follows. After a short account of LTL syntax and semantics in Sect. 2, a succinct presentation of the tableau algorithm is given in Sect. 3. Then, Sect. 4 describes the implementation of the tool, while Sect. 5 provides a detailed account of the experimental results. In Sect. 6, we recap our results and discuss possible improvements.

2 Linear Temporal Logic

In this section, we provide syntax and semantics of LTL. An LTL formula is obtained from a set Σ of proposition letters by possibly applying the usual Boolean connectives and the two temporal operators X (*tomorrow*) and $\mathcal{U}$ (*until*). Formally, LTL formulae ϕ are generated by the following syntax:

$$\phi := p \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \mathsf{X}\,\phi_1 \mid \phi_1\,\mathcal{U}\,\phi_2,$$

where ϕ_1 and ϕ_2 are LTL formulae and $p \in \Sigma$. Standard derived boolean connectives are also defined, together with logical constants $\bot \equiv p \wedge \neg p$, for $p \in \Sigma$, and $\top \equiv \neg\bot$. Moreover, two derived temporal operators $\mathsf{F}\,\phi \equiv \top\,\mathcal{U}\,\phi$ (*eventually*) and $\mathsf{G}\,\phi \equiv \neg\,\mathsf{F}\,\neg\phi$ (*always*) are also defined.

LTL formulae are interpreted over temporal structures. A *temporal structure* is a triple $M = (S, R, g)$, where S is a finite set of states, $R \subseteq S \times S$ is a binary relation, and, for each $s \in S$, $g(s) \subseteq \Sigma$. R is the transition relation, which is assumed to be total, and g is a labeling function that tells us which proposition letters are true at each state. Given a structure M, we say that an ω-sequence of states $\langle s_0, s_1, s_2, \ldots\rangle$ from S is a *full-path* if and only if, for all $i \geq 0$, $(s_i, s_{i+1}) \in R$. If $\sigma = \langle s_0, s_1, s_2, \ldots\rangle$ is a full-path, then we write σ_i for s_i and $\sigma_{\geq i}$ for the infinite suffix $\langle s_i, s_{i+1}, \ldots\rangle$ (also a full-path). We write $M, \sigma \models \varphi$ if and only if the LTL formula φ is true on the full-path σ in the structure M, which is defined by induction on the structural complexity of the formula:

- $M, \sigma \models p$ iff $p \in g(\sigma_0)$, for $p \in \Sigma$,
- $M, \sigma \models \neg\varphi$ iff $M, \sigma \not\models \varphi$
- $M, \sigma \models \varphi_1 \vee \varphi_2$ iff $M, \sigma \models \varphi_1$ or $M, \sigma \models \varphi_2$
- $M, \sigma \models \mathsf{X}\,\varphi$ iff $M, \sigma_{\geq 1} \models \varphi$
- $M, \sigma \models \varphi_1\,\mathcal{U}\,\varphi_2$ iff there is some $i \geq 0$ such that $M, \sigma_{\geq i} \models \varphi_2$ and for all j, with $0 \leq j < i$, it holds that $M, \sigma_{\geq j} \models \varphi_1$

We conclude the section with an example of a meaningful LTL formula that will be useful later:

$$p \wedge \mathsf{G}(p \leftrightarrow \mathsf{X}\,\neg p) \wedge \mathsf{G}\,\mathsf{F}\,q_1 \wedge \mathsf{G}\,\mathsf{F}\,q_2 \wedge$$
$$\mathsf{G}\,\neg(q_1 \wedge q_2) \wedge \mathsf{G}(q_1 \rightarrow \neg p) \wedge \mathsf{G}(q_2 \rightarrow \neg p)$$

The $\mathsf{G}(p \leftrightarrow \mathsf{X}\,\neg p)$ clause forces the truth value of p to alternate at every step, while $\mathsf{G}\,\mathsf{F}\,q_1$ and $\mathsf{G}\,\mathsf{F}\,q_2$ require q_1 and q_2 to be true infinitely many times, although they can never be true at the same time due to $\mathsf{G}\,\neg(q_1 \wedge q_2)$. Moreover, $\mathsf{G}(q_1 \rightarrow \neg p)$ forbids q_1 to be true at the same time of p ($\mathsf{G}(q_2 \rightarrow \neg p)$ forces the same for q_2).

3 Description of the tableau system

Here we will succinctly describe the tableau system implemented in our tool. An extended description is available online in [Reynolds, 2016]. W.l.o.g., hereafter we assume the input formula to be in Negation Normal Form, with the exception of (sub-)formulae of the form $\neg(\phi\,\mathcal{U}\,\psi)$, which are handled explicitly[1]. The tableau can be thought of as an extension to the classical tableau for propositional logic with the addition of a number of specific rules to handle temporal operators. As in the propositional analogue, the tableau for an LTL formula ϕ is a tree where each node is labeled by a set of subformulae of ϕ, with the root labeled by $\{\phi\}$. The algorithm proceeds building the tree by applying a set of rules to the node label which will create one or two children nodes, or will decide to close the current branch. In the former case, the algorithm will recursively proceed on each children. In the latter case, the resulting leaf will be marked either as *ticked* (✓), in which case the corresponding branch is called *successful*, or *crossed* (✗), meaning that the corresponding branch, called a *failed* branch, is contradictory. In both cases, the branch is said to be *closed*. A leaf that is neither ticked nor crossed identifies an *open* branch, *i.e.*, a branch that needs further exploration. The formula is reported as *satisfiable* if the corresponding tableau contains at least a successful branch, which can be used to extract a class of satisfying models. Otherwise (the completely built tableau contains no successful branches), the formula is *unsatisfiable*. We further define a partial ordering relation between tableau nodes, where $u < v$ if u is a proper ancestor of v (*i.e.*, $v \neq v$).

Tableau rules can be partitioned into two sets: *static* and *non-static* rules. Static rules can be applied depending only on the contents of the current node's label; non-static rules need to take into account also predecessor nodes. Both kinds of rules, however, deal with propositional and/or temporal reasoning regarding the current state of the model that is being built. Once all the rules have been applied, and the branch has not been closed, a last rule called the STEP rule is applied to make a *temporal step* through the model, starting to reason about the next state.

The explanation of the system's rules will start from the static rules. The following two static rules can close a branch:

EMPTY If the node's label is empty, the branch is ticked.

CONTRADICTION If the node's label contains both a propositional letter p and its negation $\neg p$ then the branch is crossed.

To explain the remaining static rules, the following notation will be used. Let ϕ be a formula, Γ be a set of formulae, and Δ be the current node's label. A rule of the form $\phi \rightarrow \Gamma$ fires when $\phi \in \Delta$ and creates a child node,[2] whose label is $\Delta \setminus \{\phi\} \cup \Gamma$, *i.e.*, substitutes Γ for ϕ. A rule of the form $\phi \rightarrow \Gamma \mid \Gamma'$ does the same, but creating two different children using the two different Γ and Γ' sets. The rules are the following:

CONJUNCTION	$\alpha \wedge \beta$	$\rightarrow$	$\{\alpha, \beta\}$
DISJUNCTION	$\alpha \vee \beta$	$\rightarrow$	$\{\alpha\} \mid \{\beta\}$
UNTIL	$\alpha\,\mathcal{U}\,\beta$	$\rightarrow$	$\{\beta\} \mid \{\alpha, \mathsf{X}(\alpha\,\mathcal{U}\,\beta)\}$
NOT-UNTIL	$\neg(\alpha\,\mathcal{U}\,\beta)$	$\rightarrow$	$\{\neg\alpha, \neg\beta\} \mid$ $\{\neg\beta, \mathsf{X}\neg(\alpha\,\mathcal{U}\,\beta)\}$
FUTURE	$\mathsf{F}\,\alpha$	$\rightarrow$	$\{\alpha\} \mid \{\mathsf{X}\,\mathsf{F}\,\alpha\}$
ALWAYS	$\mathsf{G}\,\alpha$	$\rightarrow$	$\{\alpha, \mathsf{X}\,\mathsf{G}\,\alpha\}$

Static rules are recursively applied to the children until either the branch is closed or no more static rules can be applied to the label. When this is the case the label is called *poised*. The order of application of the static rules does not matter, as long

[1] NNF of not-until formulae would require the *release* operator, which we decided not to support.

[2] Rules that add a single child can be equivalently thought of as changing the node's label itself.

as a poised label is reached. Order can impact performance, though (see Sect. 4). At this point, non-static rules have to be applied. As the reader may have noticed, the UNTIL and FUTURE rules generate, in one of the two branches, formulae of the form $\mathsf{X}(\alpha\,\mathcal{U}\,\beta)$ or $\mathsf{X}\,\mathsf{F}\,\alpha$, that re-propose the same formula that triggered the rule, but inside a *tomorrow* operator. A formula of one of those kinds is called X-*eventuality*, and represents some requirement that has still to be fulfilled by the partial model built so far by the current branch. Note that the ALWAYS[3] rule behaves in a similar way for a formula $\mathsf{G}\,\alpha$, by adding $\{\alpha, \mathsf{X}\,\mathsf{G}\,\alpha\}$ to the node's label. However, the rule does not introduce an eventuality because the requirement has been *already* fulfilled, relatively to the partial model built so far, by adding α to the current state. Eventualities are different in that the requirement could be postponed indefinitely without being ever able to fulfill it. The next two rules address these issues. The LOOP rule ticks a branch when the current branch is repeating the same actions over again and there are no unfulfilled eventualities. The PRUNE rule handles the opposite case, crossing a branch that is indefinitely pushing forward an unfulfilled eventuality that will not be satisfied.

LOOP Let v and u be nodes, with $u < v$ and poised labels Γ_v and Γ_u, respectively, such that $\Gamma_u \supseteq \Gamma_v$. If for each X-eventuality $\mathsf{X}(\alpha\mathcal{U}\beta)$ or $\mathsf{X}\,\mathsf{F}\,\beta$ in Γ_u there exists a node w such that $u < w \leq v$ and $\beta \in \Gamma_w$, then the branch terminating with v is ticked.

PRUNE Let u, v, w be nodes, with $u < v < w$ and the same poised label Γ. W.l.o.g., we can suppose no other nodes with the same label exist between u and w. If for each X-eventuality $\mathsf{X}(\alpha\,\mathcal{U}\,\beta)$ or $\mathsf{X}\,\mathsf{F}\,\beta$ in Γ, each time β was fulfilled between v and w (*i.e.*, there exists x, with $v < x \leq w$, such that $\beta \in \Gamma_x$) it was also fulfilled between u and v (*i.e.*, there exists y, with $u < y \leq v$, such that $\beta \in \Gamma_y$), then the branch terminating in w is crossed.

The LOOP and PRUNE rules must be tested and potentially applied in this order, because a node satisfying the LOOP rule could also satisfy the PRUNE rule, leading to incomplete results if the order is swapped. The reader might wonder why the PRUNE rules needs to go through three different nodes with the same label before crossing the branch, as at first it would seem sufficient to stop at the first repetition of the same label. However, there are formulae where this is not the case, one of them being the example formula from Sect. 2. Intuitively, the $\mathsf{G}\,\mathsf{F}\,q_1$ and $\mathsf{G}\,\mathsf{F}\,q_2$ subformulae repeatedly generate two X-eventualities that cannot be satisfied at the same time because of the $\mathsf{G}\,\neg(q_1 \wedge q_2)$ clause. This means that the two eventualities are repeatedly requested but alternatively fulfilled, so the label is repeated but only one of them is satisfied between a given pair of nodes with equal labels. A naive PRUNE rule would cross the branch where instead it would suffice to wait for another repetition and then tick the branch with the LOOP rule. Instead, the correct PRUNE rule waits to see two sequences of steps where the *same* eventualities, but not all of them, have been fulfilled between two pairs of nodes with the same label, meaning that nothing more can be fulfilled continuing on this branch.

[3]The same applies to the NOT-UNTIL rule.

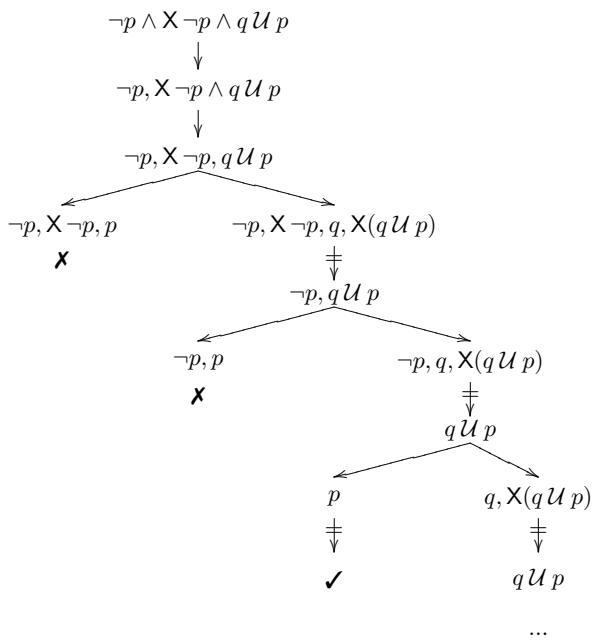

Figure 1: Example tableau for $\neg p \wedge \mathsf{X}\,\neg p \wedge (q\,\mathcal{U}\,p)$

To complete the picture, we now state the last rule, which is responsible to advance to the next temporal state:

STEP Let v be a node, with a poised label Γ, where all the previous rules have already been applied and cannot be applied anymore. Then, a child node is added to v with the label:

$$\Delta = \{\alpha \mid \mathsf{X}\,\alpha \in \Gamma\}$$

In other words, the branch steps to the next temporal state by pushing forward all the *tomorrow*s. This is the point where, if no tomorrow operators were present in the label, an empty label can be formed, thus allowing the EMPTY rule to tick the branch. As an example, a tableau built from the formula $\neg p \wedge \mathsf{X}\,\neg p \wedge (q\,\mathcal{U}\,p)$ is shown in Fig. 1. In the picture, double striked edges represent applications of the STEP rule.

The rules explained so far define a complete and correct satisfiability testing procedure for LTL formulae.

Theorem 1 *Let ϕ be an* LTL *formula. The tableau built from ϕ contains at least a ticked branch iff ϕ is satisfiable.*

In addition, a ticked branch can be used to extract a satisfying model for the formula, actually more than one as will be clear later. The satisfying full-path is built from the ticked branch by extracting all the literals contained in the label of each node on which a STEP rule has been applied, starting from the root down to the ticked leaf. Then, if the branch was ticked by applying the EMPTY rule, the full-path is complete and the remaining infinite suffix does not affect the formula's satisfaction. This is the case, for instance, with a simple formula like $p \wedge \mathsf{X}\,\neg p$, which is satisfied by any full-path $\langle s_1, s_2, \ldots\rangle$ where p holds in s_1, and $\neg p$ holds in s_2. If instead the branch was ticked by a LOOP rule, the full-path continues by infinitely repeating the sequence of states between the

ticked leaf v and the node u that triggered the LOOP rule. In this case, we say that the model *loops* from v to u. Note that this does not mean that the full-path itself contains a loop, as the full-path is an infinite linear sequence of states and cannot loop. It means instead that the labeling of the full-path states repeat. Note that from a single ticked branch we can extract a number of different (sometimes infinite) full-paths that satisfy the formula, which differ only in some states where the value of some proposition letter does not matter. The tableau system is correct and complete also as a model building tool, as formally stated by the following theorem, which implies Theorem 1:

Theorem 2 *Let ϕ be a* LTL *formula. For each ticked branch in the tableau for ϕ, there is a satisfying model of ϕ. Vice versa, for each satisfying full-path σ of ϕ there is a ticked branch from which σ can be extracted.*

As an example, Fig. 2 shows in a compact way the model for the example formula shown in Sect. 2.

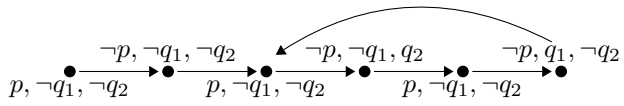

Figure 2: Model for the example formula of Sect. 2

As shown in [Sistla and Clarke, 1985, Theorem 4.6], for every model of an LTL formula ϕ, there is a ultimately periodic model that satisfies ϕ, which is in fact found by the above procedure. The same theorem also shows an exponential upper bound on the size of the prefix and periodic part of those models. The bound is strict, since for example the counters formulae shown in [Rozier and Vardi, 2010] have models of exponential size. It can be shown that this is also the upper bound on the space required for the search of a successful tableau branch. A single branch having exponential size means the entire tree is at worst doubly exponential in size, and this gives us the worst case time complexity. In summary, the following result can be proved:

Theorem 3 *The search for a successful tableau branch has an exponential and doubly exponential space and time computational complexity, respectively.*

Although not theoretically optimal for the problem, which is PSPACE-complete, note that this result is in line with the other tree-like tableau systems like Schwendimann's one.

4 Implementation

The tableau system described in the previous section has been implemented in a C++ software tool called `Leviathan`. The tool has been designed to be as fast and space-efficient as possible, and portable across a wide number of operating systems and compilers[4]. Despite the simplicity of the system's rules, finding the most efficient way to implement them was not trivial. The result is an optimized implementation which uses a remarkably low amount of memory. This section describes the design choices that underlie `Leviathan`'s implementation in view of its speed and low memory consumption requirements.

Each input formula, before being given as input to the main algorithm, passes through several preprocessing steps which syntactically simplify the formula while maintaining logical equivalence. The preprocessing phase is used to desugar derived logical syntax and to turn the formula into Negation Normal Form, as assumed in Sect. 3, but also to remove or transform a few kinds of trivial subformulae with common propositional and temporal equivalences (most of which can be found in [Giannakopoulou and Lerda, 2002]).

To achieve the promised space efficiency, formulae are represented in a compact way during the search. In the preprocessing phase, all the subformulae that will be needed for the expansion of static rules are extracted. The resulting set of formulae is then ordered in such a way that, for each formula φ, if φ is at position i, then formulae $\neg\varphi$ and $\mathsf{X}\,\varphi$, if present, are at positions $i+1$ and $i+2$, respectively. The ordered set is not represented explicitly. Instead, a few *bitset* data structures are created, one for each syntactic type of formula, such that the i-th bit in the bitset T is set to 1 if and only if the i-th formula in our ordering is of type T. To complete the picture, two vectors, respectively called `lhs` and `rhs`, are used to get the index of the left and right subformulae of each formula.

This compact representation also provides an efficient way to test the conditions of the tableau rules. As an example, consider the CONTRADICTION rule, which crosses a branch if occurrences of both p and $\neg p$, for some p, are detected in a node's label. Such an operation can be efficiently implemented as follows. Let `formulae` be the bitset corresponding to the current label and `neg_lits` be the bitset that specifies which subformulae are of type *negative literal*. Then, consider the following expression of bitwise operations:

```
((formulae & neg_lits) << 1) & formulae
```

The first *bitwise and* operation intersects the current label with the set of all the negative literals. The *shift* moves of one position all those bits, and in the result of the second *bitwise and* there will be a bit set to 1 only if both were set, *i.e.*, only if both positive and negative literals of the same atom were present. This exploits the fact that ϕ and $\neg\phi$ are consecutive in the order[5]. Another example is a static rule like CONJUNCTION, whose condition can be tested by a *bitwise and* operation between the bitsets representing the current node's label, the set of formulae in the label that still have to be processed, and the set of all the subformulae of *conjunction* type.

Finally, during the preprocessing step all the subformulae corresponding to X-eventualities are discovered and saved in a vector for later uses, together with a lookup table that links each eventuality to the corresponding index in the bitset representation.

[4]The tool has been continuously tested on the following operating systems: Windows 7/8/Server 2012, Ubuntu 14.04, CentOS 6.7, and Mac OS X from 10.9 to 10.11; and the following compilers: VC++ 2015, G++ 4.8/4.9/5.1, and Clang 3.5/3.6/3.7

[5]Contradictions due to the occurrences of two formulae of the forms $\alpha\,\mathcal{U}\,\beta$ and $\neg(\alpha\,\mathcal{U}\,\beta)$ cannot be detected in this way, and thus they are dealt with separately.

The algorithm is a tree-shaped tableau system, allowing completely independent descents through every branch. This relieves us from the need to maintain in memory the complete tree, and it allows for a linear and more compact representation of the tree itself. Since a run of the algorithm resembles a pre-order depth-first visit in the tree, a stack data structure is sufficient to maintain the state of the search.

Two different types of frame are interleaved into the stack: *choice* and *step* frames. The former are pushed when a static rule has been applied and a new branch has been created. Additional information is held by the frame to make it possible to *rollback* the choice and to descend through the other branch. The latter are pushed when a transition rule has been applied and thus a *temporal step* has been made. These are the frames corresponding to the nodes which have a *poised label* in the tableau and they bring with them information about the satisfied X-eventuality during this temporal step. Note that only the static rules that create a branch in the tree have a corresponding choice frame in the stack. The others are expanded in-place in the current frame. The rules are currently applied in a fixed order. Some early tests have shown that a wrong order can have a huge impact on performance, but an extensive search for the better order still needs to be done.

Each frame of the stack records the set of formulae belonging to the corresponding tableau node, and it keeps track of those formulae that have been already expanded by static rules. Both these pieces of information are stored in two bitsets similar to those described previously. Moreover, each frame keeps track of which eventualities have been fulfilled.

Finally, each frame stores three pointers to previous frames in the stack: to the last step frame, to the last occurrence of its label, and to its first, earliest occurrence. These pointers are used to check the PRUNE rule in the following way. As stated in Sect. 3, the rule would require to check the fulfilled eventualities between every pair of nodes u, v which share the same poised label Γ as the current node w, but this is not really needed. It is actually sufficient to set v as the last previous occurrence of Γ and u as the first, earliest one, which are exactly the nodes pointed by the aforementioned pointers. To see that it is sufficient, suppose by contradiction that there are some other u', v' that together with w trigger the PRUNE rule, while u, v, and w do not. Since u is the earliest occurrence of Γ, and v is the last, u' and v' lie between u and v. Since u, v, and w do not trigger the rule, there is an eventuality α, fulfilled between v and w, which is not already fulfilled between u and v. But then α has to be also fulfilled between v' and w, because $v' \leq v < w$. Moreover, α will not be fulfilled between u' and v', because $u \leq u' < v' \leq v$. This is a contradiction and thus u, v, and w must also trigger the rule.

A similar argument shows that, to test the LOOP rule, it suffices to consider nodes v and w, instead of scanning through the entire branch. Moreover, the rule can be implemented by testing the equality of the labels of v and w, instead of looking for subsets. It is easy to see that this change can at worst make the branch loop later, but cannot compromise completeness. On the other hand, the time required to expand a branch longer than needed is overcome by the greater efficiency in checking the rule, as only a check against the first appearance of the label is needed instead of each superset.

5 Experimental results

This section outlines the experimental evaluation of the tool against a number of already existing satisfiability checkers. In order to obtain significant data and to reduce chances of misinterpretation, we relied on *StarExec* [Stump *et al.*, 2014], an online testing and benchmarking infrastructure specifically designed to measure performance of tools for logic-related problems like SAT, SMT, CLP, *etc*. The use of a common infrastructure increases the reproducibility of the experimental results, and minimizes the risk of configuration errors of the tools that could lead to misleading results.

Complete and detailed surveys of the performance of available LTL satisfiability checkers appeared in the last few years [Goranko *et al.*, 2010; Rozier and Vardi, 2010; Schuppan and Darmawan, 2011; Vardi *et al.*, 2013]. The following analysis used the reference set of testing formulae available on StarExec, which came from the survey by Schuppan and Darmawan, including a total of 3723 formulae collected from a number of different sources. The following tools were available in StarExec and were included in the comparison: Aalta [Li *et al.*, 2014], based on Büchi automata, TRP++ [Hustadt and Konev, 2003] and LS4 [Suda and Weidenbach, 2012], which are based on temporal resolution, the NuSMV state-of-the-art symbolic model checker [Cimatti *et al.*, 2002], and PLTL, another tableau-based tool. NuSMV, configured in BDD-based symbolic model checking mode, has been chosen among other model checkers, as a single proponent of this class of tools, mainly because it was the one used in the aforementioned survey by Schuppan and Darmawan. The PLTL tool implements two different kinds of tableau-based algorithms for LTL satisfiability, a recent graph-shaped tableau system [Abate *et al.*, 2009], and the tree-like tableau system by Schwendimann.

A plot of the comparison results can be seen in Fig. 3, where test formulae are grouped by type and displayed horizontally, and vertical bars of different shades represent the relative performance of different tools. The top and bottom plots show time and memory usage, respectively, obtained in two different runs with 500MB maximum memory limit in the second case, and a 30 minutes timeout in both (which is the maximum timeout allowed by StarExec). Experiments are basically limited to solvers already available in the StarExec infrastructure, to ensure repeatability and availability of results. We plan to extend the comparison to other tools, *e.g.*, [Bradley, 2012; Hassan *et al.*, 2013].

In summary, the results show that `Leviathan`'s performance is comparable in most cases with other tools, both regarding time and memory usage, with both dark and bright corners. Diving deeper, a case-by-case analysis has to be done looking at different kinds of test formulae. While data confirm that LS4, Aalta and NuSMV are likely the best tools available, our tool is competitive in a number of cases. Notable examples are the `anzu`, `forobots`, and `rozier` (excluding counters) datasets, where `Leviathan`'s running time is one of the lowest, and the `schuppan` and `rozier-counter` datasets, where the memory usage was very low. The `rozier` dataset is also notable for being much easier for all the tested tableau-based approaches, including

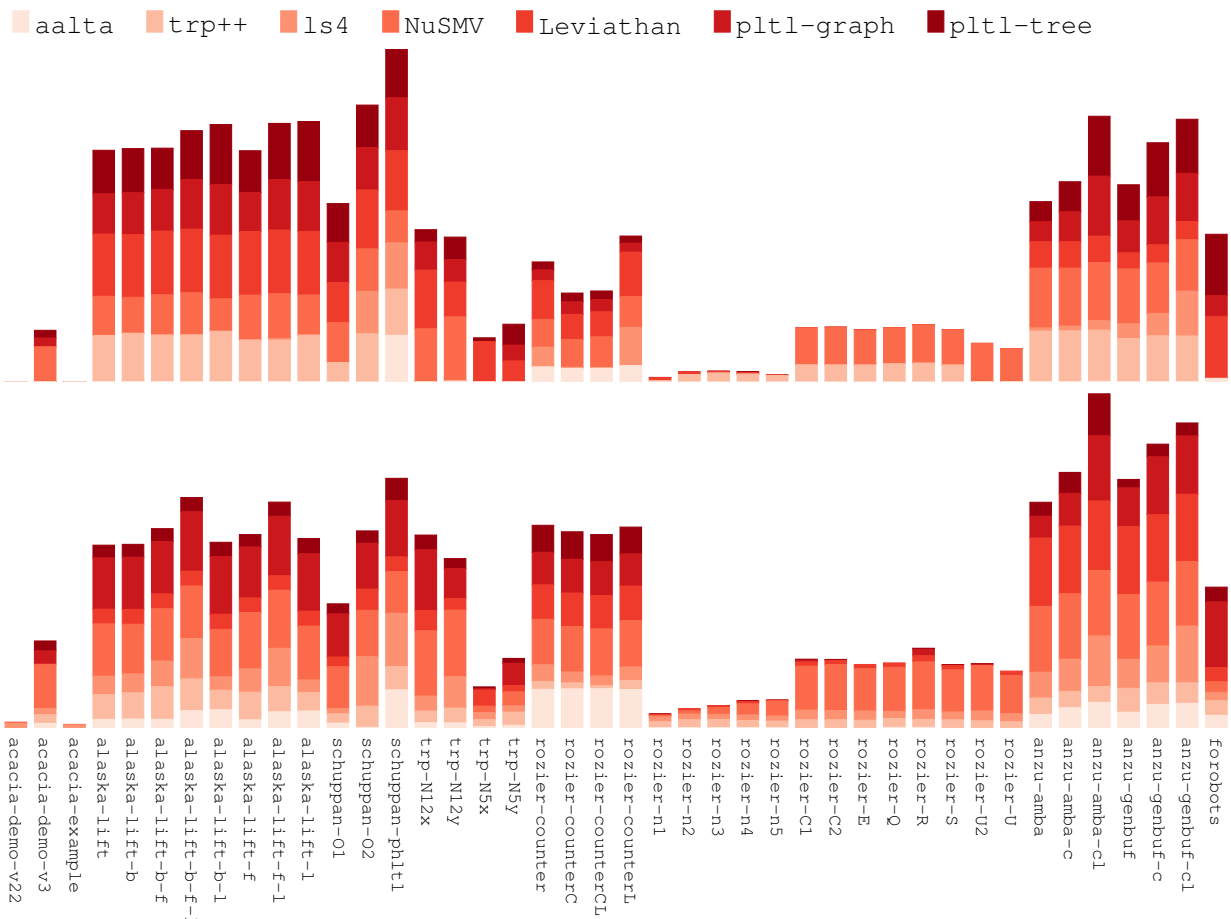

Figure 3: Benchmark results about time (above) and memory consumption (below)

Leviathan, than for other tools. The alaska dataset is very difficult for most of the tested tools, and Leviathan times out on all of it. However, it is a curious fact that, for the time it has been running before the timeout, its memory usage in this dataset was very low compared to other tools.

Another interesting point of view is the comparison of Leviathan with PLTL, as both are tableau-based tools. Leviathan performs better on the anzu dataset, uses less memory in the trp and shuppan datasets, and performance is comparable in other datasets.

6 Conclusions and future work

This paper presents Leviathan, an LTL satisfiability tool based on a simple yet effective one-pass tree-shaped tableau system. Despite the simplicity of the algorithm, experimental tests has shown that a careful implementation can achieve performance comparable with other tools, both in time and memory usage.

It is worth to point out that, although the tool has been efficiently implemented with speed and memory usage in mind, the implementation completely follows the theoretical description of Sect. 3. It is thus interesting to note that this approach can achieve good performance without applying any kind of search heuristics or other improvements, and still perform fairly well against existing tools.

Future work will explore the space of possible improvements to the basic algorithm. For example, the tableau construction is an embarrassingly parallel procedure, and a parallel implementation may easily explore multiple branches at the same time, thus exploiting modern multi-core architectures. Some work can be done on the application of heuristics to the search process. Previous work has been done on the use of unit propagation techniques, borrowed from propositional SAT solvers, to tableau systems (see [Stenz, 2005]). Another approach to improve search performance would be to embed a SAT solver into the search procedure itself, to deal with formulae with strong propositional components for which tableau systems are notoriously weak, which would likely help to deal with some of the more pathological test cases. Additional care can still be devoted to the implementation, especially to optimize the preprocessing phase that is currently a bottleneck for performance for some large formulae. Thanks to the simplicity of the proposed algorithm, we can expect that all these improvements could be implemented in a relatively easy way.

References

[A. R. Cavalli and L. Fariñas del Cerro, 1984] A. R. Cavalli and L. Fariñas del Cerro. A Decision Method for Linear Temporal Logic. In *Proc. of the 7th International Conference on Automated Deduction*, pages 113–127, 1984.

[Abate *et al.*, 2009] P. Abate, R. Goré, and F. Widmann. An On-the-fly Tableau-based Decision Procedure for PDL-satisfiability. *Electronic Notes in Theoretical Computer Science*, 231:191–209, 2009.

[Bacchus and Kabanza, 1998] F. Bacchus and F. Kabanza. Planning for Temporally Extended Goals. *Annals of Mathematics and Artificial Intelligence*, 22(1–2):5–27, 1998.

[Bradley, 2012] A. R. Bradley. Understanding IC3. In *15th International Conference on Theory and Applications of Satisfiability Testing*, pages 1–14, 2012.

[Cialdea Mayer *et al.*, 2007] M. Cialdea Mayer, C. Limongelli, A. Orlandini, and V. Poggioni. Linear Temporal Logic as an Executable Semantics for Planning Languages. *Journal of Logic, Language and Information*, 16(1):63–89, 2007.

[Cimatti *et al.*, 2002] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An Open-Source Tool for Symbolic Model Checking. In *Proc. of the 14th International Conference on Computer Aided Verification*, pages 359–364, 2002.

[Clarke *et al.*, 1999] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[Fisher *et al.*, 2001] M. Fisher, C. Dixon, and M. Peim. Clausal temporal resolution. *ACM Transactions on Computational Logic*, 2(1):12–56, 2001.

[Giannakopoulou and Lerda, 2002] D. Giannakopoulou and F. Lerda. From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata. In *Formal Techniques for Networked and Distributed Systems*, pages 308–326, 2002.

[Goranko *et al.*, 2010] V. Goranko, A. Kyrilov, and D. Shkatov. Tableau Tool for Testing Satisfiability in LTL: Implementation and Experimental Analysis. *Electronic Notes in Theoretical Computer Science*, 262:113–125, 2010.

[Hassan *et al.*, 2013] Z. Hassan, A. R. Bradley, and F. Somenzi. Better generalization in IC3. In *Formal Methods in Computer-Aided Design*, pages 157–164, 2013.

[Hustadt and Konev, 2003] U. Hustadt and B. Konev. TRP++2.0: A Temporal Resolution Prover. In *Proc. of the 19th International Conference on Automated Deduction*, pages 274–278, 2003.

[Kesten *et al.*, 1993] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A Decision Algorithm for Full Propositional Temporal Logic. In *Proc. of the 5th International Conference on Computer Aided Verification*, volume 697 of *LNCS*, pages 97–109, 1993.

[Kress-Gazit *et al.*, 2009] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. Temporal-Logic-Based Reactive Mission and Motion Planning. *IEEE Transactions on Robotics*, 25(6):1370–1381, 2009.

[Li *et al.*, 2014] J. Li, Y. Yao, G. Pu, L. Zhang, and J. He. Aalta: an LTL Satisfiability Checker over Infinite/Finite traces. In *Proc. of the 22nd ACM International Symposium on Foundations of Software Engineering*, pages 731–734, 2014.

[Manna and Pnueli, 1995] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems - Safety*. Springer, 1995.

[Pnueli, 1977] A. Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[Reynolds, 2016] Mark Reynolds. A Traditional Tree-Style Tableau for LTL. *arXiv:1604.03962*, 2016.

[Rozier and Vardi, 2010] K. Y. Rozier and M. Y. Vardi. LTL Satisfiability Checking. *International Journal on Software Tools for Technology Transfer*, 12(2):123–137, 2010.

[Schuppan and Darmawan, 2011] V. Schuppan and L. Darmawan. Evaluating LTL Satisfiability Solvers. In *Proc. of the 9th International Symposium Automated Technology for Verification and Analysis*, pages 397–413, 2011.

[Schwendimann, 1998] S. Schwendimann. A New One-Pass Tableau Calculus for PLTL. In *Proc. of the 4th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 277–292, 1998.

[Sistla and Clarke, 1985] A. P. Sistla and E. M. Clarke. The Complexity of Propositional Linear Temporal Logics. *Journal of the ACM*, 32(3):733–749, 1985.

[Stenz, 2005] G. Stenz. Unit Propagation in a Tableau Framework. In *Proc. of the 11st International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 338–342, 2005.

[Stump *et al.*, 2014] A. Stump, G. Sutcliffe, and C. Tinelli. StarExec: A Cross-Community Infrastructure for Logic Solving. In *Proc. of the 7th International Joint Conference on Automated Reasoning*, pages 367–373, 2014.

[Suda and Weidenbach, 2012] M. Suda and C. Weidenbach. A PLTL-Prover Based on Labelled Superposition with Partial Model Guidance. In *Proc. of the 6th International Joint Conference on Automated Reasoning*, pages 537–543, 2012.

[Vardi *et al.*, 2013] M. Y. Vardi, Li J, L. Zhang, G. Pu, and J. He. LTL Satisfiability Checking Revisited. In *Proc. of the 20th International Symposium on Temporal Representation and Reasoning*, pages 91–98, 2013.

[Venkatesh, 1985] G. Venkatesh. A Decision Method for Temporal Logic Based on Resolution. In *Proc. of the 5th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 272–289, 1985.

[Wolper, 1985] P. Wolper. The Tableau Method for Temporal Logic: An Overview. *Logique et Analyse*, 28, 1985.