



STLSAT—An Improved Tableau for Satisfiability Checking of Signal Temporal Logic Formulas

Marco Zamponi¹ , Florian Lammel², Ezio Bartocci² , and Michele Chiari^{2,3} 

¹ IMT Lucca, Lucca, Italy, name.surname@imtlucca.it

² TU Wien, Vienna, Austria, name.surname@tuwien.ac.at

³ AIT Austrian Institute of Technology, Vienna, Austria, name.surname@ait.ac.at

Abstract

Signal Temporal Logic (STL) is a formalism used to describe temporal properties of real-valued signals in cyber-physical systems. In mission- and safety-critical domains, specifications often consist of large collections of STL formulas, making consistency checking and requirement analysis a major engineering bottleneck. Tableau-based satisfiability procedures are a natural way to address this problem. Two of the authors of this paper contributed to the only existing tree-shaped tableau for bounded discrete-time STL, but we have recently found out that the procedure can return incorrect verdicts for some STL formulas. In this paper, we pinpoint the flaw in that procedure and present a new tree-shaped tableau that we prove to be **sound** and **complete** for bounded discrete-time STL.

On top of this theoretical foundation, we introduce STLSAT, an open-source Rust tool that decides the satisfiability of STL formulas, synthesizes concrete witness signals, and extracts unsatisfiable cores with its tableau engine, allowing users to identify inconsistent subsets of requirements for more effective specification debugging. STLSAT also implements enhanced first-order logic and satisfiability modulo theories encodings for STL, which allow it to act as a portfolio solver.

We evaluate STLSAT on an extended benchmark suite (including STL and Mission-time Linear Temporal Logic formulas) that we release publicly. Across the whole benchmark, the portfolio solver matches or outperforms state-of-the-art tools while preserving correctness.

1 Introduction

Signal Temporal Logic (STL) [30] is a well-established specification language for describing the temporal properties of real-valued signals in mission- and safety-critical cyber-physical systems (CPS). STL underpins a broad spectrum of verification tasks, ranging from runtime monitoring [25] and mutation testing [9] to falsification [38], parameter synthesis [4], fault localization [6] and debugging [7]. During the design of a CPS, engineers typically collect a large number of STL requirements. In this context, the following two situations can be problematic:

Inconsistency. A set of STL formulas $\Phi = \{\varphi_1, \dots, \varphi_n\}$ is *consistent* iff there exists at least one signal s such that $s \models \bigwedge_{i=1}^n \varphi_i$. Equivalently, the conjunction $\bigwedge_{i=1}^n \varphi_i$ must be **satisfiable**; if it is not, the requirements are mutually contradictory.

Redundancy. A formula $\psi \in \Phi$ is *redundant* w.r.t. the remaining requirements iff those requirements already entail ψ , i.e., $\bigwedge_{\varphi \in \Phi \setminus \{\psi\}} \models \psi$. By definition, this holds exactly when the formula $(\bigwedge_{\varphi \in \Phi \setminus \{\psi\}}) \wedge \neg\psi$ is **unsatisfiable**.

Moreover, manual analysis of large CPS requirement sets is error-prone and demanding in terms of time, expertise, and cognitive effort, especially when expressed in a formal language such as STL. Thus, automated requirement engineering tools are highly desirable to support engineers in these tasks. Luckily, both notions of consistency and redundancy can be reduced to a single STL-satisfiability query: for any pair of formulas φ, ψ , the implication $\varphi \rightarrow \psi$ holds iff $\varphi \wedge \neg\psi$ is unsatisfiable, and equivalence follows from mutual implication. Hence, in this paper we tackle the problem of checking STL satisfiability.

These reductions are especially valuable in *specification mining* [8], where automatically extracted temporal-logic specifications may contain noise, redundancies, or inconsistencies. Satisfiability checking can detect inconsistent subsets, eliminate redundancies, and cluster semantically equivalent specifications, yielding a more concise and traceable requirement model.

Example. Consider an autonomous crop-irrigation controller whose water-flow signal f is measured along the irrigation line. The following three STL requirements have been elicited:

$$\begin{aligned}\varphi_1 &:= G_{[0,100]} F_{[10,40]}(f \geq 50) \\ \varphi_2 &:= G_{[0,120]}(f \geq 50 \Rightarrow F_{[0,10]} G_{[0,30]}(f < 5)) \\ \varphi_3 &:= G_{[0,120]}(f < 100)\end{aligned}$$

- φ_1 requires that, at every time in $[0, 100]$, the flow reaches at least 50 l/min within the following 1040 time units.
- φ_2 states that whenever the flow reaches this irrigation value, it must be reduced below 5 l/min within the next 10 time units and stay low for another 30 time units, in order to avoid over-watering.
- φ_3 is a safety bound that forbids flows equal to or above 100 l/min at any time in the interval 0-120 units, preventing flooding of the field.

The three formulas together form the specification $\Phi_w = \{\varphi_1, \varphi_2, \varphi_3\}$. Before Φ_w can be used for model checking or falsification, we must verify that it is **consistent**, i.e. that the conjunction $\bigwedge_{i=1}^3 \varphi_i$ is *satisfiable*. Intuitively, φ_2 forces a drop from $f \geq 50$ to $f < 5$, which might conflict with φ_1 ; however, turning this intuition into a rigorous proof requires exploring all possible signal behaviors over $[0, 140]$. Manual reasoning is error-prone, whereas automated STL satisfiability tools decide consistency in seconds; if the set is satisfiable they may synthesize a concrete witness signal, and if it is unsatisfiable they may extract an unsatisfiable core that pinpoints the conflicting requirements, enabling their immediate refinement.

Although STL has become the *de facto* standard specification language for CPS requirements, it is still lacking mature, high-performance, and dedicated satisfiability solvers. Existing approaches either target different logics—e.g., MLTLSAT [28]—or exist only as research prototypes such as the tableau implementation described in Melani et al. [31]. Moreover, these tools do not integrate smoothly into verification pipelines, which limits their usefulness for practical requirement-engineering workflows that require automated debugging and traceability.

The most prominent tableau-based solver for bounded discrete-time STL is STLTREE [31], also co-developed by two of the authors of this paper. Its construction accelerates reasoning by “jumping” over intervals, but a careful analysis reveals a soundness flaw: in formulas with nested temporal operators, the old jump rules may skip instants where conflicts would be introduced, causing the tableau to miss rejecting branches. Consider the following unsatisfiable formula:

$$\varphi_S := G_{[0,5]}(p \vee G_{[2,2]} p) \wedge G_{[3,5]} q,$$

where p and q are inconsistent atomic predicates, for example $p = x \geq 0$ and $q = x < 0$. At time 3, the second conjunct requires q ; hence, the first conjunct cannot be satisfied by choosing p and instead generates the obligation $G_{[5,5]} p$. The second conjunct also requires q at time 5, making φ_S unsatisfiable. However, STLTREE jumps from time 3 to time 5 and incorrectly shifts the obligation from $G_{[5,5]} p$ to $G_{[7,7]} p$. The resulting tableau avoids the conflict between p and q at time 5 and accepts φ_S with a signal that is not actually a model of φ_S .

This motivates our first contribution: a corrected tableau algorithm with formally proved soundness and completeness. Our approach inspects proposition-appearance intervals to guarantee that no conflicts can be introduced in skipped time steps.

Building on this foundation we introduce STLSAT, an open-source Rust tool implementing three complementary decision procedures that can be executed in parallel to speed up the solving process: our correct tableau, an optimized first-order-logic encoding, and a qualitative-semantics SMT encoding. The tableau supports unsatisfiable-core extraction for debugging, and can synthesize witness signals for satisfiable formulas by extracting them from the tableau. To foster reproducibility, we also release an extensive STL benchmark suite [40].

STLSAT determines in under one second that the φ_1 – φ_3 irrigation requirements are mutually inconsistent and the tableau engine also returns the subset of constraints $\{\varphi_1, \varphi_2\}$ responsible for the conflict as an *unsatisfiable core*, thus providing immediate debugging support.

We conduct an extensive empirical evaluation on the released benchmark set of STL formulas, as well as its close relative MLTL coming from related works, comparing each STLSAT engine against state-of-the-art STL solvers and against a dedicated MLTL satisfiability checker [28]. The results show that no single technique dominates across all instances; however, the portfolio approach combining the three STLSAT solving engines achieves the best overall performance in terms of runtime and scalability.

In summary, our contributions are:

1. a sound and complete tree-shaped tableau algorithm with a novel set of rules;
2. STLSAT, a high-performance Rust portfolio solver with tableau, FOL, and SMT engines;
3. a novel unsat-core extraction method for the tableau engine;
4. an open benchmark suite covering STL formulas; and
5. a thorough experimental study demonstrating superior performance over existing tools.

The remainder of the paper is organized as follows. Section 2 surveys related work on temporal logic satisfiability. Section 3 recalls the syntax and semantics of STL and MLTL. Section 4 details the flaws in the existing tableau [31] and presents our corrected tableau algorithm together with its soundness and completeness proofs. Section 5 details the implementation of STLSAT’s three solving engines. Section 6 describes our experimental methodology and reports the results. Finally, Section 7 draws conclusions and outlines future research directions.

2 Related Work

Originally proposed for CPS monitoring and testing [30], STL has primarily been studied for runtime verification [5], while consistency checking, redundancy removal, and implication analysis remain less explored.

Validity, vacuity, and redundancy. These issues were firstly addressed by Dokhanchi et al. [19], who reduced validity, vacuity, and redundancy questions for STL to satisfiability problems in Metric Interval Temporal Logic (MITL) [1]. Their approach leverages existing MITL solvers [10] to obtain counterexamples or proofs of equivalence. While effective, this pipeline incurs the overhead of translating STL into a different logic and does not directly support the extraction of minimal conflicting subsets.

Debugging tools. In the same spirit, Roehm et al. [37] proposed STLINSPECTOR, a debugging aid that generates concrete witness signals by mutating candidate models and querying an SMT encoding of the STL formula. The tool is useful for pinpointing why a specification holds or fails on a given trace, but it does not provide a systematic analysis of the logical relationships among multiple requirements.

Satisfiability/Model checking. The close relationship between satisfiability and bounded model checking has motivated several encodings of STL into first-order logic (FOL) or Satisfiability Modulo Theories (SMT). Bae and Lee [3] presented a refutationally complete SMT encoding for bounded model checking of continuous-time STL against hybrid system models; an improved variant was later shown to be suitable for satisfiability checking [27]. For discrete-time settings, Raman et al. [34] introduced an SMT formulation aimed at model predictive control that can also serve as a decision procedure for STL satisfiability [31].

Tableau-based approaches. The only dedicated tableau method for bounded discrete-time STL prior to our work is the one-pass tree-shaped construction proposed by Melani et al. [31]. They proved that STL satisfiability is EXPSPACE-complete and implemented their algorithm, inspired by tableau techniques for other temporal logics [36, 21], in a prototype Python tool. In the course of our investigation we discovered a soundness flaw in their original jump optimization; we therefore revisited the mathematical definition, and proved both soundness and completeness for the corrected rule.

Related temporal logics. Several works have studied satisfiability for logics closely related to STL. Bersani et al. [11, 12] offered SMT encodings for MITL and Quantified Temporal Logic (QTL) [23], but these frameworks admit only Boolean signals, limiting their applicability to the quantitative nature of STL. Mission-time Linear Temporal Logic (MLTL) [35] shares many syntactic constructs with STL; a systematic comparison of encodings for MLTL including translations to LTL on finite traces, SMV specifications, and FOL was carried out by Li et al. [29], who identified the FOL encoding as the best performing. Our tool incorporates an improved version of this FOL translation and demonstrates better empirical performance on both STL and MLTL benchmarks.

Unsatisfiable-core extraction. The extraction of minimal unsatisfiable cores has been extensively investigated for propositional SAT and SMT solving [14]. Within temporal logics, similar techniques have been explored mainly for Linear Temporal Logic (LTL) [39, 22], but to the best of our knowledge no existing framework provides unsat-core extraction for STL formulas. By integrating a dedicated core extraction mechanism into our tableau engine we fill this gap and enable direct debugging of inconsistent requirement sets.

3 Background

Signal Temporal Logic (STL). We define bounded discrete-time STL [30] on the temporal domain of natural numbers $\mathbb{N}$. A *signal* is a function $w : \mathbb{N} \rightarrow \mathbb{R}^n$ that associates time instants with valuations of a finite set of signal variables $S = \{x_1, \dots, x_n\}$. We define the projection w_R of a signal w on a set $R \subseteq S$ in the usual way. STL formulas have the following syntax:

$$\varphi := \top \mid f(R) \bowtie k \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathbf{U}_I \varphi_2$$

where $\bowtie \in \{<, =, >\}$, $R \subseteq S$ is a set of signal variables, $f : \mathbb{R}^{|R|} \rightarrow \mathbb{R}$ is a linear function of signal values, $k \in \mathbb{Q}$ is a rational constant, and $I = [a, b]$ with $a, b \in \mathbb{N}$ and $a \leq b$.

We define the semantics on a signal w and a time instant $t \in \mathbb{N}$ as follows:

$$\begin{aligned}
(w, t) &\models \top && \text{(always true)} \\
(w, t) &\models f(R) \bowtie k && \text{iff } f(w_R(t)) \bowtie k, \text{ with } \bowtie \in \{<, =, >\} \\
(w, t) &\models \neg \varphi && \text{iff } (w, t) \not\models \varphi \\
(w, t) &\models \varphi_1 \vee \varphi_2 && \text{iff } (w, t) \models \varphi_1 \text{ or } (w, t) \models \varphi_2 \\
(w, t) &\models \varphi_1 \mathbf{U}_{[a,b]} \varphi_2 && \text{iff } \exists t' \in [t+a, t+b] : (w, t') \models \varphi_2 \text{ and } \forall t'' \in [t, t'] : (w, t'') \models \varphi_1
\end{aligned}$$

Boolean signals can be encoded by defining a real variable x_p for each *atomic proposition* p , and asserting p as $x_p = 1$ and $\neg p$ as $x_p \neq 1$.

We also use propositional operators ($\wedge, \rightarrow$), and temporal operators *eventually* (or *finally*) F_I and *always* (or *globally*) G_I as shortcuts for $F_I \varphi \equiv \top \mathbf{U}_I \varphi$ and $G_I \varphi \equiv \neg F_I \neg \varphi$. We also employ the *release* operator R , defined as the dual of *until* under negation, i.e., $\varphi_1 R_I \varphi_2 \equiv \neg(\neg \varphi_1 \mathbf{U}_I \neg \varphi_2)$. Given $\varphi_1 \mathbf{U}_I \varphi_2$ and $\varphi_2 R_I \varphi_1$, we refer to φ_1 as the *invariant*, and to φ_2 as the *target*.

Mission-Time Linear Temporal Logic (MLTL). MLTL [35] can be seen as a fragment of bounded discrete-time STL that only allows for Boolean signals. It also differs from STL in the semantics of the until operator, which we denote as strict until ($\mathbf{sU}$) for MLTL:

$$(w, t) \models \varphi_1 \mathbf{sU}_{[a,b]} \varphi_2 \text{ iff } \exists t' \in [t+a, t+b] : (w, t') \models \varphi_2 \text{ and } \forall t'' \in [t+a, t'-1] : (w, t'') \models \varphi_1$$

The strict release operator is defined as $\varphi_1 \mathbf{sR}_I \varphi_2 \equiv \neg(\neg \varphi_1 \mathbf{sU}_I \neg \varphi_2)$.

Indeed, MLTL's until operator is expressible in terms of STL's, and *vice versa*:

$$\begin{aligned}
\varphi_1 \mathbf{sU}_{[a,b]} \varphi_2 &\equiv F_{[a,a]} ((\varphi_1 \vee \varphi_2) \mathbf{U}_{[0,b-a]} \varphi_2) & \varphi_1 \mathbf{sR}_{[a,b]} \varphi_2 &\equiv F_{[a,a]} ((\varphi_1 \wedge \varphi_2) R_{[0,b-a]} \varphi_2) \\
\varphi_1 \mathbf{U}_{[a,b]} \varphi_2 &\equiv G_{[0,a]} \varphi_1 \wedge \varphi_1 \mathbf{sU}_{[a,b]} (\varphi_1 \wedge \varphi_2) & \varphi_1 R_{[a,b]} \varphi_2 &\equiv F_{[0,a]} \varphi_1 \vee \varphi_2 \mathbf{sU}_{[a,b]} \varphi_1 \vee G_{[a,b]} \varphi_2
\end{aligned}$$

The equivalences in the second row, together with common propositional and temporal logic equivalences, allow for rewriting all STL formulas in *strict normal form*, i.e., by using only strict operators in positive form, with negation only applied to atomic propositions.

4 Tree-Shaped Tableau

Building upon the tree-shaped tableau introduced in [31], we propose an alternative definition of the JUMP rule, an optimization that skips sequences of redundant time steps during tableau exploration while fixing the soundness and completeness problems of the original rule.

We begin by reviewing the original *basic* tableau, i.e., without the JUMP rule, whose soundness and completeness results still hold. We then introduce the new JUMP rule and, finally, present its correctness theorems.

4.1 Basic Tableau

A tableau is a tree in which each node u is labeled with a set of formula occurrences $\Gamma(u)$ required to hold conjunctively, and a time counter $t(u) \in \mathbb{N}$. The root u_0 is labeled with the formula ϕ to be checked, assumed to be in strict normal form. Besides a formula, elements of $\Gamma(u)$ may carry auxiliary metadata recording the temporal formula occurrence that generated them—i.e., their *parent*. We write $\langle \varphi \rangle_\psi$ for the occurrence whose underlying formula is φ and whose parent occurrence is ψ , and $\langle \varphi \rangle_\perp$ when no parent is present (e.g., for operators in the

Table 1: Expansion rules. A node u with $\phi \in \Gamma(u)$ that satisfies the condition is expanded into one child u_p or two children u_s and u_p such that $t(u_s) = t(u_p) = t(u)$, $\Gamma(u_s) = (\Gamma(u) \setminus \{\phi\}) \cup \Gamma_\phi(u_s)$ and $\Gamma(u_p) = (\Gamma(u) \setminus \{\phi\}) \cup \Gamma_\phi(u_p)$.

Cat.	$\phi \in \Gamma(u)$	condition	$\Gamma_\phi(u_s)$	$\Gamma_\phi(u_p)$
Prop.	$\langle \varphi_1 \vee \varphi_2 \rangle_\psi$ $\langle \varphi_1 \wedge \varphi_2 \rangle_\psi$		$\langle \varphi_1 \rangle_\psi$ $\langle \varphi_1 \rangle_\psi, \langle \varphi_2 \rangle_\psi$	$\langle \varphi_2 \rangle_\psi$
Temporal	$\langle F_{[a,b]} \varphi \rangle_\psi$	$a \leq t(u) < b$ $t(u) = b$	$\langle \text{tex}^{t(u)}(\varphi) \rangle_\perp$ $\langle \text{tex}^{t(u)}(\varphi) \rangle_\perp$	$\langle \bar{F}_{[a,b]} \varphi \rangle_\psi$
	$\langle G_{[a,b]} \varphi \rangle_\psi$	$a \leq t(u) < b$ $t(u) = b$		$\langle \bar{G}_{[a,b]} \varphi \rangle_\psi, \langle \text{tex}^{t(u)}(\varphi) \rangle_\phi$ $\langle \text{tex}^{t(u)}(\varphi) \rangle_\phi$
	$\langle \varphi_1 \text{sU}_{[a,b]} \varphi_2 \rangle_\psi$	$a \leq t(u) < b$ $t(u) = b$	$\langle \text{tex}^{t(u)}(\varphi_2) \rangle_\perp$ $\langle \text{tex}^{t(u)}(\varphi_2) \rangle_\perp$	$\langle \varphi_1 \text{s}\bar{U}_{[a,b]} \varphi_2 \rangle_\psi, \langle \text{tex}^{t(u)}(\varphi_1) \rangle_\phi$
	$\langle \varphi_1 \text{sR}_{[a,b]} \varphi_2 \rangle_\psi$	$a \leq t(u) < b$ $t(u) = b$	$\langle \text{tex}^{t(u)}(\varphi_1) \rangle_\perp, \langle \text{tex}^{t(u)}(\varphi_2) \rangle_\perp$	$\langle \varphi_1 \text{s}\bar{R}_{[a,b]} \varphi_2 \rangle_\psi, \langle \text{tex}^{t(u)}(\varphi_2) \rangle_\phi$ $\langle \text{tex}^{t(u)}(\varphi_2) \rangle_\phi$

input formula). For instance, in $\langle a \rangle_{G_I a}$, the occurrence $G_I a$ is the parent of a , meaning that unfolding it generated that occurrence of a . Parent annotations are only needed by the JUMP rule, hence we omit them whenever not strictly relevant, and we identify occurrences in $\Gamma(u)$ by their underlying formula.

The tableau is built incrementally by applying the *expansion* rules in Table 1, which decompose formulas in a node's label into simpler ones according to their semantics generating child nodes. Decomposing temporal operators introduces marked operators denoted by a bar (e.g., $\bar{F}$), which represent operators whose satisfaction is postponed to the next time step. Furthermore, temporal operator decomposition employs a temporal expansion function tex^t mapping formulas inside nested temporal operators to their realizations when unfolded at time t .

which updates intervals when nested temporal operators are unfolded:

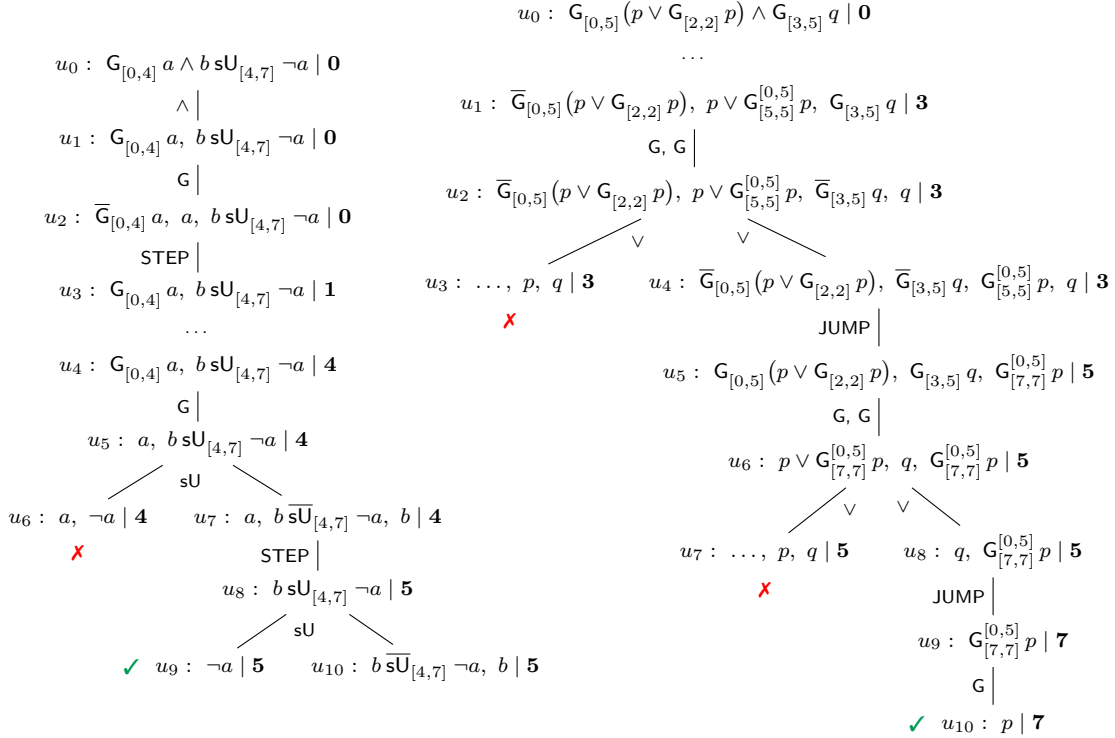
$$\begin{aligned}
\text{tex}^t(\top) &= \top & \text{tex}^t(f(R) \bowtie k) &= f(R) \bowtie k & \text{tex}^t(\neg \varphi_1) &= \neg \text{tex}^t(\varphi_1) \\
\text{tex}^t(\varphi_1 \circ \varphi_2) &= \text{tex}^t(\varphi_1) \circ \text{tex}^t(\varphi_2) & & & \text{with } \circ \in \{\wedge, \vee\} \\
\text{tex}^t(A_{[a,b]} \varphi) &= A_{[a+t, b+t]} \varphi & & & \text{with } A \in \{F, G\} \\
\text{tex}^t(\varphi_1 B_{[a,b]} \varphi_2) &= \varphi_1 B_{[a+t, b+t]} \varphi_2 & & & \text{with } B \in \{\text{sU}, \text{sR}\}.
\end{aligned}$$

During tableau construction, nodes are checked for local inconsistencies. Formally, a node u is rejected (and closed) if either $\neg \top \in \Gamma(u)$ or the set of propositions $\{f(R) \bowtie k \in \Gamma(u)\} \cup \{\neg(f(R) \bowtie k) \in \Gamma(u)\}$ is inconsistent. Rejected nodes are leaves.

When no further decomposition rule is applicable to a node, i.e., all temporal operators in its label are marked or not active, the node is said to be *poised*. In this case, the tableau advances time through the STEP rule:

STEP Let u be a poised node. If $\Gamma(u)$ contains any temporal operators, then u has one child u' with $t(u') = t(u) + 1$, and

$$\begin{aligned}
\Gamma(u') &= \{A_I \varphi \in \Gamma(u) \mid A \in \{F, G\}\} \cup \{\varphi_1 B_I \varphi_2 \in \Gamma(u) \mid B \in \{\text{sU}, \text{sR}\}\} \\
&\cup \{A_{[a,b]} \varphi \mid \bar{A}_{[a,b]} \varphi \in \Gamma(u) \wedge A \in \{F, G\} \wedge t(u) < b\} \\
&\cup \{\varphi_1 B_{[a,b]} \varphi_2 \mid \varphi_1 \bar{B}_{[a,b]} \varphi_2 \in \Gamma(u) \wedge B \in \{\text{sU}, \text{sR}\} \wedge t(u) < b\}
\end{aligned}$$

Figure 1: Basic tableau for $G_{[0,4]} a \wedge b sU_{[4,7]} \neg a$.Figure 2: STLTREE tableau for φ_S .

If u is poised, not rejected, and its **STEP** successor label is empty, then the corresponding branch is accepted and witnesses the satisfiability of the input formula. If the tableau frontier consists of only rejected nodes, the input formula is unsatisfiable.

Fig. 1 shows an example of the basic tableau for formula $G_{[0,4]} a \wedge b sU_{[4,7]} \neg a$. The tableau starts with the expansion rule for the $\wedge$ operator, and proceeds with the one for the G operator, which creates a marked instance of $G_{[0,4]} a$ and extracts its argument a . The interval of the sU operator starts at 4, so no more expansion rules can be applied and u_2 is *poised*, and the **STEP** rule advances time by creating node u_3 . This process goes on until time 4, creating many nodes identical to u_1 and u_2 , which we omit from the figure. At time 4, the sU operator is expanded, creating two nodes $u_s = u_6$ and $u_p = u_7$. In u_6 , we try to *satisfy* it by extracting $\neg a$, which conflicts with a extracted by the G operator and leads to a rejected node. In u_7 , we *postpone* the satisfaction of the sU operator by marking it and extracting b , which does not conflict with any other proposition. Since $G_{[0,4]} a$ ends at time 4, the step rule does not propagate it to u_8 . Thus, the sU can finally be satisfied in u_9 . Since u_9 has no temporal operators and no conflicts between atomic propositions, it is accepted, witnessing the satisfiability of the input formula.

Soundness and completeness of the basic tableau were established in [31]:

Theorem 1 ([31, Theorem 4.4 and Lemma 3]). *A formula φ is satisfiable iff the basic tableau rooted in it has an accepting branch.*

4.2 The JUMP_{Old} rule and its unsoundness

As an optimization to the basic tableau, Melani et al. [31] introduced the JUMP_{Old} rule to skip repetitive parts of the tableau by jumping directly to a later point in time. For instance, for formula $G_{[0,20]} x > 0 \wedge F_{[0,5]} x > 5$, the tableau need not explore every time instant, as checking consistency only at interval bounds 0, 5, and 20 is sufficient to determine satisfiability. Formulas with nested operators are trickier: the tableau by [31] waits until the end of the time interval of the first extracted instance of the inner operator. For instance, with formula $G_{[5,20]} F_{[0,5]} x > 0$, the tableau proceeds with the **STEP** rule until time $5 + 5 = 10$, after which it may jump to time 20.

We found out, however, that this precaution is not enough to preserve the soundness of the tableau. As we mentioned in §1, formula φ_S witnesses this flaw. Fig. 2 shows part of the tableau rooted in φ_S that uses the JUMP_{Old} rule. The figure omits the prefix up to time 3, in which the first conjunct is satisfied by choosing p and the **STEP** rule is applied. At time 3, expanding the two outer G operators produces q and the disjunction $p \vee G_{[5,5]}^{[0,5]} p$ (here, the superscript $[0,5]$ indicates that the formula is extracted from an operator with interval $[0,5]$). The branch that chooses p is locally inconsistent with q and is rejected. The other branch chooses $G_{[5,5]}^{[0,5]} p$.

Since its current time is greater than $0 + 2 = 2$, the poised node u_4 satisfies the conditions of the JUMP_{Old} rule, which jumps from time 3 to time 5. Since the parent of $G_{[5,5]}^{[0,5]} p$ is still active at time 3, the rule treats this formula as a derived obligation and shifts both interval bounds by the jump length 2. Consequently, u_6 contains $G_{[7,7]}^{[0,5]} p$ instead of the required $G_{[5,5]}^{[0,5]} p$. At time 5, the first conjunct can again select its nested G disjunct, which yields the same obligation at time 7; after q expires, the tableau satisfies p at time 7 and accepts the branch. The branch therefore never checks the inconsistent pair p, q at time 5.

The JUMP_{Old} rule compromises both soundness and completeness of the tableau. We describe this flaw in more detail in [42, Appendix A].

4.3 A new, correct **JUMP** rule

To speed up tableau construction, we introduce the **JUMP** rule, which can be used to compress a sequence of **STEP** rule applications. Intuitively, a jump from a poised node u to a node u' has the same effect as postponing all active temporal operators until time $t(u')$ by always choosing the expansion rule $\Gamma_\phi(u_p)$.

To describe the **JUMP** rule, we introduce new notation, then discuss the jump-size computation, and finally describe the resulting node. To simplify notation, we omit the G and F operators, which we replace by the standard equivalences $G_I \varphi \equiv \neg \top \text{ sR}_I \varphi$ and $F_I \varphi \equiv \top \text{ sU}_I \varphi$.

The **JUMP** rule needs parent annotations, which are created by expansion rules. When the **JUMP** rule is enabled, the **STEP** rule propagates parent annotations on all formulas that it inserts in the new node. They are accessed through the *parent-active* predicate P , which we define over a formula occurrence and a node as follows:

$$P(\phi, u) \quad \text{iff} \quad \phi = \langle \varphi \rangle_\psi \text{ for some } \varphi, \psi \in \Gamma(u) \text{ s.t. } \psi \neq \perp \text{ and } \psi \in \Gamma(u).$$

Proposition Validity Intervals. We define the proposition validity intervals $\mathcal{T}(\varphi)$ of a formula φ as the time intervals in which its atomic propositions may appear, allowing us to

identify potential conflicts. Formally:

$$\begin{aligned}
\mathcal{T}(f(R) \bowtie k) &= \mathcal{T}(\top) = \{(0, 0)\} & \mathcal{T}(\neg\varphi) &= \mathcal{T}(\varphi) \\
\mathcal{T}(\varphi_1 \circ \varphi_2) &= \mathcal{T}(\varphi_1) \cup \mathcal{T}(\varphi_2) & \text{with } \circ &\in \{\wedge, \vee\} \\
\mathcal{T}(\varphi_1 \text{ sU}_{[a,b]} \varphi_2) &= \mathcal{T}(\varphi_1 \text{ sR}_{[a,b]} \varphi_2) = \{(l + a, r + b - 1) \mid (l, r) \in \mathcal{T}(\varphi_1), a < b\} \\
&\quad \cup \{(l + a, r + b) \mid (l, r) \in \mathcal{T}(\varphi_2)\}
\end{aligned}$$

To distinguish different occurrences of the same proposition, we associate a unique identifier to every occurrence. For instance, in $\varphi = p \text{ sU}_{[0,5]} q \wedge p \text{ sU}_{[0,10]} r$, the two occurrences of p are treated as distinct because they are governed by different temporal constraints. Formally, the formula is viewed as a syntax tree whose leaves carry unique identifiers. Two propositions are considered the same only if they share the same syntactic structure and identifier. Identifiers are preserved by expansion rules and by the temporal expansion operator tex . Such identifiers are used only for validity intervals: for local consistency checking, propositions are compared solely based on their syntactic structure.

Function $\mathcal{T}$ is defined so that each occurrence of an atomic proposition ρ in a formula contributes exactly one pair (l, r) . Hence, we may unambiguously refer to a pair $i = (l, r) \in \mathcal{T}(\varphi)$ deriving from ρ , meaning the specific occurrence identified via its identifier rather than merely its syntactic structure. We denote as $\text{id}(i)$ such an identifier.

The following lemma, whose proof is in [42, Appendix B], relates validity intervals computed from a formula φ to the time instants at which occurrences of a proposition produced by decompositions of φ may appear in the tableau:

Lemma 2. *Let u be a node of the basic tableau, $\varphi \in \Gamma(u)$ be a formula in the label of the node, and ρ be an atomic proposition appearing in φ . Let $(l, r) \in \mathcal{T}(\varphi)$ be the proposition validity interval deriving from ρ . Then, in the subtree rooted at u , any occurrence of ρ derived from the expansion of φ can only appear in nodes v such that $l \leq t(v) \leq r$ if ρ is nested within a temporal operator in φ , and such that $t(v) = t(u)$ otherwise.*

4.3.1 Jump Size Calculation

To guarantee that the JUMP rule preserves the correctness of the algorithm, we compute the jump size k^* as the minimum of two bounds.

Bounds Limit (k) The first limit ensures that the jump does not cross a time instant at which an operator becomes active or expires. It is defined as

$$k(u) = \min\{t \in K(u) \mid t > t(u)\} - t(u),$$

where the set $K(u)$ collects bounds of temporal operators in u :

$$K(u) = \{a, b \mid \varphi_1 \text{ B}_{[a,b]} \varphi_2 \in \Gamma(u), \text{ B} \in \{\text{sU}, \overline{\text{sU}}, \text{sR}, \overline{\text{sR}}\}\}$$

We illustrate the computation of the bounds on node u_2 from the tableau in Fig. 1. We have $K(u_2) = \{0, 4, 7\}$, and thus $k(u_2) = 4$.

Soundness Limit (k_{sound}^*) Consider a temporal operator $\varphi = \varphi_1 \overline{\text{sU}}_{[a,b]} \varphi_2$ or $\varphi = \varphi_2 \overline{\text{sR}}_{[a,b]} \varphi_1$ in a poised node u . The basic tableau would emit the invariant part φ_1 at every intermediate time instant $t(u) + j$, with $1 \leq j < k$, before reaching the jump destination $t(u) + k$. To guarantee

the soundness of the approach, we need to be sure that such invariants do not generate conflicts with other formulas in the node.

We first collect the validity intervals of atomic propositions that may be generated by the invariant part of active postponed temporal operators:

$$N(u) = \bigcup_{\varphi \in \Gamma(u)} \{(t(u) + l, t(u) + r) \mid (l, r) \in \mathcal{T}(\varphi_1), \varphi = \varphi_1 \overline{sU}_{[a,b]} \varphi_2 \vee \varphi = \varphi_2 \overline{sR}_{[a,b]} \varphi_1\}.$$

An element $(l_n, r_n) \in N(u)$ describes the validity interval of an invariant emitted at time instant $t(u)$ by an active temporal operator φ , i.e., $\text{tex}^{t(u)}(\varphi_1)$. If the same invariant is emitted at a skipped time $t(u) + j$, its interval is shifted to $(l_n + j, r_n + j)$.

We then collect the validity intervals of atomic propositions that may be generated by temporal operators in $\Gamma(u)$ whose parent is not active:

$$O(u) = \bigcup_{\varphi \in \Gamma(u)} \{(l, r) \mid (l, r) \in \mathcal{T}(\varphi), \varphi = \varphi_1 B_{[a,b]} \varphi_2, B \in \{sU, \overline{sU}, sR, \overline{sR}\}, \neg P(\varphi, u)\}$$

If an interval in $N(u)$ intersects one in $O(u)$, we cannot exclude that the skipped invariant conflicts with other formulas. Since we do not know which propositions will eventually form an inconsistent core, we perform a **STEP** as a safe choice: the **JUMP** rule is disabled whenever

$$\exists(n, o) \in N(u) \times O(u) \text{ s.t. } \text{id}(n) \neq \text{id}(o) \wedge n \cap o \neq \emptyset \quad (1)$$

If no intersection exists, for every $n = (l_n, r_n) \in N(u)$ and $o = (l_o, r_o) \in O(u)$, either $r_n < l_o$ or $r_o < l_n$. The case $r_o < l_n$ is always safe: delaying the emitted invariants moves the interval to the future, so other proposition intervals will never intersect with it. In case $r_n < l_o$ instead, shifting the invariant to the future might cause the intersection of the intervals. We define the sound jump limit as the earliest distance at which propositions may appear together:

$$k_{\text{sound}}^*(u) = \min(\{l_o - r_n \mid (n, o) \in N(u) \times O(u) \wedge \text{id}(n) \neq \text{id}(o) \wedge r_n < l_o\} \cup \{+\infty\})$$

Let us evaluate the soundness bounds on node u_2 in Fig. 1: we have $N(u_2) = \{(0, 0)\}$ and $O(u_2) = \{(0, 4), (4, 6), (4, 7)\}$, hence $k_{\text{sound}}^*(u_2) = 4$.

Completeness Guard Consider a temporal operator $\varphi = \varphi_1 \overline{sU}_{[a,b]} \varphi_2$ or $\varphi = \varphi_2 \overline{sR}_{[a,b]} \varphi_1$ in a poised node u . The **JUMP** rule reconstructs the branch in which this temporal obligation is postponed until the jump destination $t(u) + k$. In the basic tableau, an accepting branch may satisfy the temporal obligation (i.e., select the $\Gamma_\varphi(u_s)$ expansion rule) at some intermediate time $t(u) + j$, with $1 \leq j < k$. If time step $t(u) + j$ is skipped, the **JUMP** rule may reach a node rejected due to the postponed obligation, even though the basic tableau had an accepting branch rooted at $t(u) + j$.

We first collect the validity intervals of atomic propositions that may be generated by the target (i.e., φ_2) part of active temporal operators:

$$M(u) = \bigcup_{\varphi \in \Gamma(u)} \{(t(u) + l, t(u) + r) \mid (l, r) \in \mathcal{T}(\varphi_2), \varphi = \varphi_1 \overline{sU}_{[a,b]} \varphi_2 \vee \varphi = \varphi_2 \overline{sR}_{[a,b]} \varphi_1\}.$$

An element $(l_m, r_m) \in M(u)$ describes the validity interval of a proposition generated by the formula emitted at time $t(u)$, namely $\text{tex}^{t(u)}(\varphi_2)$. If the same target is emitted at some time $t(u) + j$ with $1 \leq j < k$, its interval is shifted to $(l_m + j, r_m + j)$.

We then collect the validity intervals of propositions that may generate conflicts with a target extraction. These consist of propositions already present in the node and the ones generated by non-expired temporal operators without an active parent:

$$S(u) = O(u) \cup \{(t(u), t(u)) \mid \varphi \in \Gamma(u), \varphi = f(R) \bowtie k \vee \varphi = \neg(f(R) \bowtie k), \neg P(\varphi, u)\}.$$

If an interval in $M(u)$ intersects one in $S(u)$, then we cannot determine when the invariant stops conflicting with the target. Since the extraction might be satisfiable at the next instant, we perform a STEP, and disable the JUMP rule whenever:

$$\exists m \in M(u), \exists s \in S(u) \text{ s.t. } \text{id}(m) \neq \text{id}(s) \wedge m \cap s \neq \emptyset \quad (2)$$

If no such intersection exists, then for every $m = (l_m, r_m) \in M(u)$ and $s = (l_s, r_s) \in S(u)$, either $r_m < l_s$ or $r_s < l_m$. Case $r_s < l_m$ is safe, as the potentially conflicting interval is before the target one, and delaying extraction would move the latter to the future. Otherwise, when $r_m < l_s$, extracting the target at a later time can cause the two intervals to intersect, leading to a possible conflict. However, if the JUMP rule is enabled, then no two intervals m and s intersect, and all currently active temporal formulas can be satisfied by extracting their targets at the current time. Thus, there is no need for a jump limit to ensure completeness.

In the running example of Fig. 1, we have $M(u_2) = \{(0, 0)\}$ and $S(u_2) = \{(0, 4), (4, 7)\}$. Note that proposition a has been extracted from the active operator $G_{[0,4]} a$, so it is marked with $\langle a \rangle_{G_{[0,4]} a}$, and its validity interval is excluded from $S(u_2)$.

4.3.2 Rule Application

Let u be a poised node in which (1) and (2) do *not* hold. The JUMP rule creates one child u' with $t(u') = t(u) + k^*(u)$, where the jump size $k^*(u)$ and node label $\Gamma(u')$ are defined as

$$\begin{aligned} k^*(u) &= \min(k(u), k_{\text{sound}}^*(u)) \\ \Gamma(u') &= \{ \langle \varphi_1 \text{ sU}_{[a,b]} \varphi_2 \rangle_\psi \mid \langle \varphi_1 \text{ B}_{[a,b]} \varphi_2 \rangle_\psi \in \Gamma(u), \text{ B} \in \{\text{sU}, \overline{\text{sU}}\}, t(u') \leq b \} \\ &\quad \cup \{ \langle \varphi_1 \text{ sR}_{[a,b]} \varphi_2 \rangle_\psi \mid \langle \varphi_1 \text{ B}_{[a,b]} \varphi_2 \rangle_\psi \in \Gamma(u), \text{ B} \in \{\text{sR}, \overline{\text{sR}}\}, t(u') \leq b \} \end{aligned}$$

The label $\Gamma(u')$ omits real-valued constraints and unmarks operators, so that expansion rules can build nodes for the next time instants. The bound $k_{\text{sound}}^*(u)$ guarantees that the JUMP rule preserves remaining temporal obligations while avoiding redundant checks.

Finally, we can evaluate the JUMP bounds on node u_2 of Fig. 1. We have $k(u_2) = 4$ and $k_{\text{sound}}^*(u_2) = 4$; hence, the JUMP rule can be applied with a jump size of 4, skipping time instants 1 to 3. With the JUMP rule, node u_4 is the (only) child of u_2 .

Shared-Variable Optimization For a set of atomic constraints to be inconsistent, at least two of them have to share a variable. Moreover, two occurrences of a Boolean variable cannot conflict if they have the same polarity. From this observation we can derive a simple optimization that allows for applying the JUMP rule more often: in (1) and (2) we consider only pairs of intervals that come either from two opposite occurrences of the same Boolean variable (such as a and $\neg a$) or from two atomic constraints such that at least one variable occurs in both.

4.3.3 Correctness

We prove soundness and completeness of the JUMP rule by showing, respectively, that an accepted branch in the JUMP tableau implies one in the basic tableau, and *vice versa*.

Lemma 3. *Let u be a poised node. Let u' be the node obtained by applying the *JUMP* rule to u , or the *STEP* rule if *JUMP* conditions are not satisfied. If the subtree rooted at u' contains an accepted branch, then the subtree of the basic tableau rooted at u also contains one.*

Theorem 4 (Soundness). *If the tableau built by using the *JUMP* rule has an accepted branch, then the basic tableau for a formula ϕ also has an accepted branch*

Theorem 5 (Completeness). *If the basic tableau for a formula ϕ has an accepted branch, then the tableau built by using the *JUMP* rule also has an accepted branch.*

We report manually written proofs of these results in [42, Appendix B]. Moreover, we have formalized the above claims and their proofs with the Lean 4 proof assistant [18], with the help of GPT 5.6 Sol [33]. The Lean formalization is publicly available in [40].

5 STLSat Tool

STLSAT [41] is a modular tool for satisfiability checking of bounded discrete-time STL formulas. The full implementation, including source code, benchmarks, and reproducibility scripts, is publicly available¹. Given a formula φ , STLSAT determines whether there exists a discrete-time signal $w : \mathbb{N} \rightarrow \mathbb{R}^n$ such that $(w, 0) \models \varphi$. STLSAT implements three different solving engines working over the same formula representation:

- the tableau-based method described in §4;
- a FOL engine for STL inspired by [29];
- a quantifier-free encoding inspired by MPC [34] and bounded model checking [15] methods.

5.1 Tableau Method

STLSAT features an efficient implementation of the tableau methodology described in §4.

It introduces a lightweight syntactic simplification phase before tableau construction. This pre-processing step applies common rewriting rules, including nested conjunction and disjunction flattening, redundant operand removal (e.g., $\varphi \wedge \top \equiv \varphi$), and merging of temporal operator intervals (e.g., $G_{[0,3]} \varphi \wedge G_{[2,5]} \varphi \equiv G_{[0,5]} \varphi$). These simplifications reduce formula complexity and, consequently, the depth of tableau expansion without affecting satisfiability.

Furthermore, STLSAT replaces the tableau expansion of [31], which relies on the program call stack for node exploration, with an explicit stack-based implementation. This design enables finer control over memory allocation and improves performance when exploring deep tableaux.

STLSAT performs local consistency checks of atomic propositions by invoking the Z3 solver [17]. We also implemented a custom solver for systems of difference constraints by reducing them to negative-cycle detection in graphs [16, 2, 20]. This optimization improves performance by eliminating the external Z3 calls for supported formulas.

Witness Signals STLSAT is able to extract witness signals for satisfiable STL formulas. The witness signal is constructed by traversing the accepting branch of the basic tableau and collecting the atomic propositions extracted at each time instant. The resulting signal is then represented as a piecewise function over the time domain, with values determined by the extracted propositions.

¹<https://github.com/ZamponiMarco/stlsat/>

Unsat-Core Extraction Whenever a formula is determined to be unsatisfiable, STLSAT performs a lightweight unsatisfiable-core extraction to identify the source of inconsistency. We represent the input formula φ as a conjunction of subformulas after repeatedly applying the $\wedge$ -expansion rule, so that at the root node u_0 we have $\Gamma(u_0) = \{\varphi_1, \dots, \varphi_m\}$, and $\varphi = \bigwedge_{i=1}^m \varphi_i$. Let $\mathcal{A}$ be the set of atomic predicates $f(R) \bowtie k$ occurring in φ . We associate each atomic predicate $a \in \mathcal{A}$ to its originating subformula φ_i through a mapping $M : \mathcal{A} \rightarrow \{1, \dots, m\}$.

During tableau expansion, every node contains a set of atomic predicates $A(u) \subseteq \mathcal{A}$ extracted from the input formula φ . When a node is rejected by the underlying theory solver, the solver provides the subset of conflicting atomic predicates responsible for the inconsistency:

$$C_u = \{a_{u,1}, a_{u,2}, \dots, a_{u,k}\} \subseteq A(u).$$

When all branches of the tableau are rejected, the local unsatisfiable cores are aggregated into a single set $C = \bigcup_{u \text{ rejected}} C_u$ and mapped back to their originating formulas in the root node, producing a high-level unsatisfiable core $U \subseteq \Gamma(u_0)$ that identifies a subset of subformulas responsible for the inconsistency, with $U = \{\varphi_i \mid \exists a \in C : M(a) = i\}$. The returned core is unsatisfiable, but not necessarily irreducible or minimal. Its further evaluation and minimization, as well as solver-native core extraction for the remaining engines, are left to future work.

5.2 First-Order Logic Encoding

An almost out-of-the-box solution for STL satisfiability is to translate a formula φ into First-Order Logic (FOL) by recursively replacing each operator with its semantics definition, and then check the resulting formula with a theorem prover or an SMT solver with quantifier support.

Li et al. [29] applies this idea to MLTL. They encode signals as functions from non-negative integers (representing time) to Boolean values: for each atomic proposition p , they define a function $p : \mathbb{N} \rightarrow \{\top, \perp\}$, such that $p(t) = \top$ iff p holds at time t . Moreover, they define a Boolean function f taking two integers k and len , which returns $\top$ iff the FOL translation of φ is true at time k for a signal of length len .

We implement an optimized STL encoding where signals are integer-to-real functions (for real values) or integer-to-Boolean functions (for Boolean signals). The translation follows the semantics in §3. To reduce formula size, we employ the usual *ad hoc* FOL translations for the F and G operators, rather than translating them to U.

5.3 Quantifier-Free SMT Encoding

Closely related to the FOL approach, STL satisfiability can also be encoded as a quantifier-free SMT problem over linear real arithmetic, similarly to SMT encodings used for optimal control [34] and bounded model checking [15].

The encoding first computes the formula horizon and creates symbolic variables for each signal variable at every time instant. The encoding recursively unfolds the semantics of STL operators. Boolean operators are translated directly into their SMT counterparts, while temporal operators are expanded over their intervals. The resulting quantifier-free linear real arithmetic formula can be solved by an SMT solver.

Table 2: Descriptive statistics of the MLTL and STL benchmark suites. Values show median (95th percentile) for depth, temporal depth and horizon, and mean for branching factor (i.e., number of node children). The last four columns show the frequency (%) of each operator.

	Bench. suite	Depth	T. Depth	Horizon	B. F.	F	G	U	R
MLTL	NASA-Boeing	8.0 (11.0)	1.0 (1.9)	100000.0 (100001.8)	1.80	12.8	87.2	0.0	0.0
	Random	10.0 (20.0)	6.0 (18.0)	461.0 (1374.0)	1.46	36.8	37.1	15.9	10.3
	Random0	10.0 (20.0)	6.0 (18.0)	316.0 (943.0)	1.46	37.0	36.8	15.6	10.6
STL	Random-STL	10.0 (21.0)	4.0 (8.0)	1000.0 (10000.0)	1.65	33.4	33.4	16.6	16.6
	Random0-STL	11.0 (25.0)	5.0 (12.0)	205.5 (4234.1)	1.54	33.3	33.0	16.9	16.7

6 Experiments

We compare the three STLSAT engines against state-of-the-art STL and MLTL satisfiability tools. For both logics, we include STL_{TREE} [32], the original Python implementation of [31]; for MLTL, we also compare with the best-performing technique for MLTL satisfiability analyzed by Li et al. [29], namely their FOL encoding which we call ML_{TLSAT}, obtained by running the tool made available by the authors [28]. ML_{TLSAT} translates MLTL formulas into SMTLIB files, which we feed to Z3 4.15.8 [17]. We ran all experiments on a server equipped with an AMD EPYC 9654 CPU and 755 GB of RAM running Ubuntu 24.04. The parallel configuration runs all three engines concurrently and reports the wall-clock time until the first engine terminates, thus representing a portfolio result. A replication package can be found in [40].

6.1 Benchmarks

We evaluate STLSAT on MLTL and STL benchmark suites, whose descriptive statistics are reported in Table 2.

MLTL. The first benchmark suite in Li et al. [29] contains three subsets of MLTL formulas:

- *NASA-Boeing*: 63 satisfiable formulas extracted from real-world avionics systems;
- *Random*: 4,000 randomly generated formulas with realistic patterns and random integer time intervals;
- *Random0*: 4,000 randomly generated formulas whose time intervals all start at 0.

All MLTL formulas use only Boolean variables, stressing temporal operators over large horizons without linear real arithmetic.

STL. The second benchmark suite comprises three groups of STL formulas:

- nine requirements collected by Melani et al. [31] plus the running example Φ_w ;
- *Random-STL*: 2,000 randomly generated formulas of varying length;
- *Random0-STL*: 2,000 random formulas of varying length with all intervals starting at 0.

Note that the satisfiability of all generated formulas is not controlled beforehand and is ultimately determined by the solving engines. The generation process for the random STL benchmark suites is detailed in [42, Appendix C].

6.2 Results

MLTL. We compare STLSAT with STL_{TREE} and the FOL encoding of [29] (ML_{TLSAT}). We show the results as survival plots in Fig. 3, with a 120 s timeout. In Table 3 we present the number of instances in which each STLSAT solving engine was strictly the fastest, grouped by benchmark and formula satisfiability for both MLTL and STL suites. On the NASA-Boeing

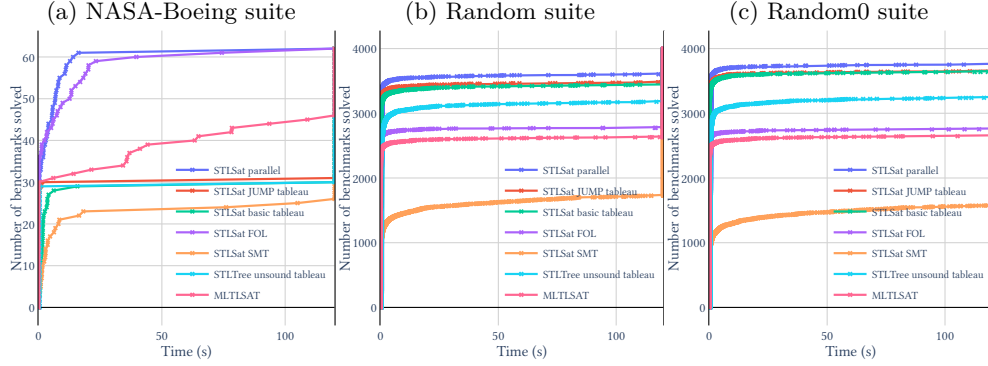


Figure 3: Survival plots for MLTL benchmarks [29]. Higher is better.

Table 3: Fastest STLSAT engine by benchmark and satisfiability class. Upper rows report counts and lower rows percentages among instances solved by at least one engine (n); ties are counted separately.

	benchmark	sat					unsat				
		tableau	FOL	SMT	tie	n	tableau	FOL	SMT	tie	n
MLTL	nasa-boeing	27 43.5%	35 56.5%	0 0.0%	0 0.0%	62	— —	— —	— —	— —	0
	random	3166 92.5%	226 6.6%	14 0.4%	17 0.5%	3423	153 85.5%	23 12.8%	3 1.7%	0 0.0%	179
	random0	3278 94.0%	190 5.4%	3 0.1%	16 0.5%	3487	217 83.1%	41 15.7%	0 0.0%	3 1.2%	261
STL	random	702 69.1%	143 14.1%	146 14.4%	25 2.4%	1016	227 55.0%	121 29.3%	22 5.3%	43 10.4%	413
	random0	682 83.4%	36 4.4%	81 9.9%	19 2.3%	818	284 62.6%	108 23.8%	40 8.8%	22 4.8%	454

suite, the FOL encoding (§ 5.2) dominates: it solved 62 benchmarks within 80 s, reporting a parsing error on one. MLTLSAT solved 46 benchmarks within 108 s and timed out on 14 launching parsing errors on 3. This advantage comes from a smaller generated encoding with fewer quantifiers: these formulas—derived from manually written requirement sets—contain only F and G rather than U operators, and can benefit from our *ad hoc* encoding of F and G.

On the other hand, tableau-based tools perform better on randomly generated formulas, with STLSAT outperforming STLTree despite using stricter jumping rules, thanks to the efficient algorithm implementation. In fact, formulas in these suites contain many temporal operators with wide time intervals, which can greatly benefit from the tableau’s JUMP rule. The SMT encoding (§ 5.3) performed worst on average, consistent with the findings of [31], although it was the only STLSAT engine able to solve the *Aerospace* benchmark within the time limit.

Executing STLSAT’s three solving engines in parallel brings a slight benefit in random benchmarks, because there are some formulas for which one particular engine performs better than the others. In the NASA-Boeing set, instead, the FOL encoding terminates first on the large majority of instances (although by a small margin), and parallel execution offsets the overhead of running all engines.

Table 4: Results on STL benchmarks from [31]. Columns 2 to 4 contain, respectively, number of requirements, maximum time Horizon, and temporal operators. Column R gives the result, shared by all approaches ($\top$ = “satisfiable”, $\perp$ = “unsatisfiable”). The remaining columns give the solution time (s) for each approach, where TO denotes a timeout at 120s. Bold marks the fastest solver per row.

Benchmark	#	H	Op.	R	STLTree tableau	STLSat SMT	STLSat FOL	STLSat basic t.	STLSat JUMP t.
Car [3]	4	100	G,F,U	$\top$	0.5328	0.0238	0.0212	0.0114	0.0050
Thermostat [3]	4	40	G,F,U,R	$\top$	0.4930	0.0186	0.0310	0.0060	0.0044
Watertank [3]	4	50	G,F	$\top$	0.5044	0.0174	0.0194	0.0116	0.0146
Railroad [3]	4	100	G,F	$\perp$	5.4144	0.0186	0.0582	0.0538	0.0258
Battery [3]	4	64	G,F,U	$\top$	0.5940	0.0240	0.0852	0.0158	0.0088
PCV [26]	9	1,250	G,F	$\top$	2.2950	1.0698	0.0190	22.1564	25.0328
MTL [24]	10	2,020	G,F	$\perp$	1.4636	0.1914	0.0178	0.2798	0.2834
CPS [13]	37	13,799	G,F,U	$\perp$	3.1226	TO	0.0198	TO	TO
Aerospace [31]	31	1,010	G,F	$\top$	8.7816	37.5802	TO	TO	TO
Irrigation Φ_w , §1	3	160	G,F	$\perp$	0.6774	0.0306	TO	0.0698	0.0404

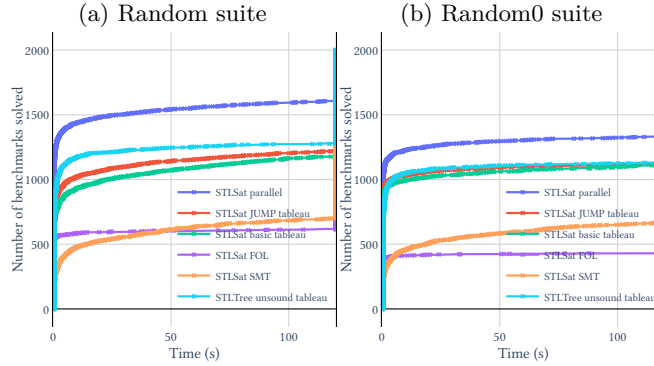


Figure 4: Survival plots for random STL benchmarks. Higher is better.

STL. Table 4 shows our comparison on benchmarks from [31], plus the running example Φ_w . STLSAT outperforms STLTree in some benchmarks, but times out in the largest ones; on the other hand, the FOL encoding quickly solves CPS, and the SMT encoding solves Aerospace without timing out.

Fig. 4 shows survival plots for randomly generated STL formulas. STLSAT’s tableau performs slightly worse than STLTree, but much better than the FOL and SMT encodings. Table 3 suggests that the tableau is the fastest engine on most instances, while FOL and SMT are each fastest on smaller subsets of formulas. In particular, FOL wins on most of the NASA-Boeing benchmarks, and wins more frequently in unsatisfiable instances than satisfiable ones. Consequently, parallel execution of the STLSAT engines yields further computational gains.

Ablation Study Our JUMP rule is a sound and complete alternative to the JUMP_{Old} rule [31], although it is generally applicable in fewer cases. To assess its practical impact, we compare the STLSAT tableau against the basic tableau with JUMP disabled (where only STEP rules execute) across all benchmarks from §6.1.

Fig. 5 compares their solving times: most points lie below the bisector, showing that JUMP

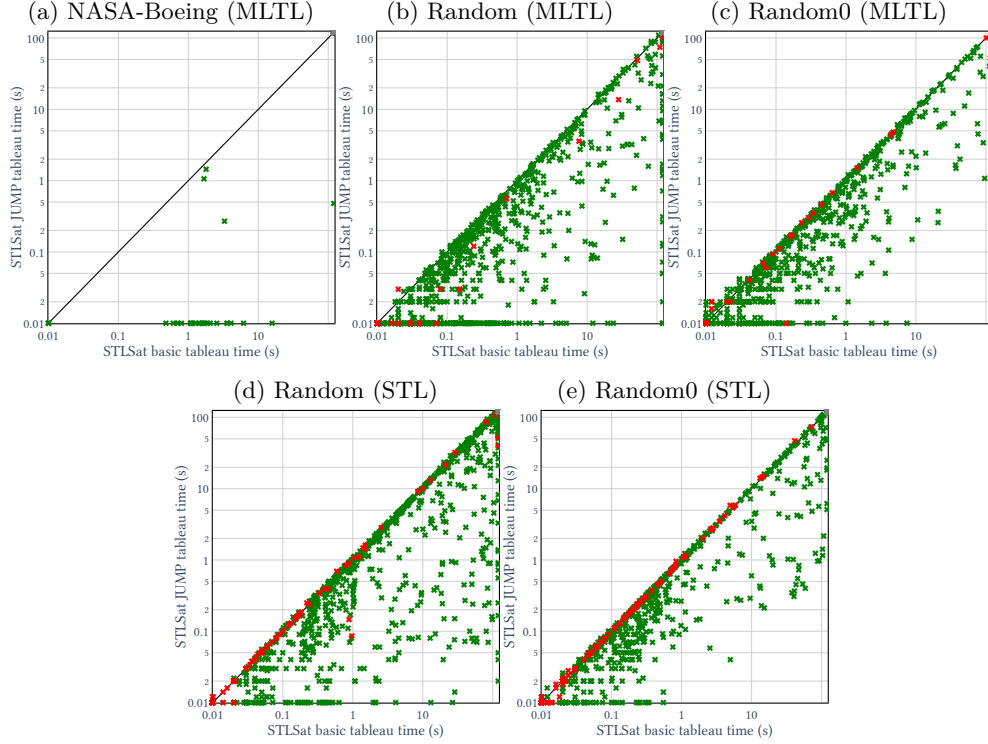


Figure 5: Scatter plots for the STLSAT ablation study on MLTL and STL benchmarks.

generally improves performance. Furthermore, instances where jumping provides no benefit from the optimization remain close to the bisector, indicating that computing and checking the jump conditions introduces negligible overhead.

7 Conclusions

We presented STLSAT, an open-source Rust tool for satisfiability checking of bounded discrete-time STL. We identified an unsound optimization in the tableau of Melani et al. [31] and replaced it with a sound and complete rule. STLSAT combines this tableau with two alternative encodings: a FOL translation and a quantifier-free SMT encoding, which can be executed in parallel as a portfolio solver.

Our extensive benchmark suite (covering both STL and MLTL, with real-world and randomly generated instances) shows that STLSAT performs comparably to STLTREE, and often outperforms state-of-the-art SMT encodings such as MLTLSAT. Furthermore, running all STLSAT solving engines in parallel yields the best overall performance. Moreover, we argued how *unsat* core extraction facilitates specification mining, redundancy elimination, and traceability in CPS development.

Future work will focus on temporal analysis for *unsat cores*, support for unbounded STL operators and continuous-time semantics, tighter integration with model-checking, falsification, and specification mining, and the development of visual debugging interfaces that display counterexample signals and core constraints.

Acknowledgements

Partially funded by the European Union. Views and opinions expressed are those of the authors only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency (HADEA). Neither the European Union nor the granting authority can be held responsible for them. RobustifAI project, ID 101212818; VASSAL project, ID 101160022; MoSaiC project, ID 101103010. Partially funded by WWTF project ICT22-023 (TAIGER).

References

- [1] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43(1):116–146, 1996. doi:[10.1145/227595.227602](https://doi.org/10.1145/227595.227602).
- [2] Alessandro Armando, Claudio Castellini, Enrico Giunchiglia, and Marco Maratea. A SAT-based decision procedure for the Boolean combination of difference constraints. In *SAT 2004*, 2004. URL <http://www.satisfiability.org/SAT04/programme/71.pdf>.
- [3] Kyungmin Bae and Jia Lee. Bounded model checking of signal temporal logic properties using syntactic separation. *Proc. ACM Program. Lang.*, 3:1–30, 2019. doi:[10.1145/3290364](https://doi.org/10.1145/3290364).
- [4] Ezio Bartocci, Luca Bortolussi, Laura Nenzi, and Guido Sanguinetti. System design of stochastic models using robustness of temporal properties. *Theor. Comput. Sci.*, 587:3–25, 2015. doi:[10.1016/J.TCS.2015.02.046](https://doi.org/10.1016/J.TCS.2015.02.046).
- [5] Ezio Bartocci, Jyotirmoy V. Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Nickovic, and Sriram Sankaranarayanan. Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 135–175. Springer, 2018. doi:[10.1007/978-3-319-75632-5_5](https://doi.org/10.1007/978-3-319-75632-5_5).
- [6] Ezio Bartocci, Thomas Ferrère, Niveditha Manjunath, and Dejan Nickovic. Localizing faults in Simulink/Stateflow models with STL. In *HSCC '18*, pages 197–206. ACM, 2018. doi:[10.1145/3178126.3178131](https://doi.org/10.1145/3178126.3178131).
- [7] Ezio Bartocci, Niveditha Manjunath, Leonardo Mariani, Cristinel Mateis, and Dejan Nickovic. Cpsdebug: Automatic failure explanation in CPS models. *Int. J. Softw. Tools Technol. Transf.*, 23(5):783–796, 2021. doi:[10.1007/S10009-020-00599-4](https://doi.org/10.1007/S10009-020-00599-4).
- [8] Ezio Bartocci, Cristinel Mateis, Eleonora Nesterini, and Dejan Nickovic. Survey on mining signal temporal logic specifications. *Inf. Comput.*, 289(Part):104957, 2022. doi:[10.1016/J.IC.2022.104957](https://doi.org/10.1016/J.IC.2022.104957).
- [9] Ezio Bartocci, Leonardo Mariani, Dejan Nickovic, and Drishti Yadav. Property-based mutation testing. In *ICST '23*, pages 222–233. IEEE, 2023. doi:[10.1109/ICST57152.2023.00029](https://doi.org/10.1109/ICST57152.2023.00029).
- [10] Marcello M. Bersani, Matteo Rossi, and Pierluigi San Pietro. A tool for deciding the satisfiability of continuous-time metric temporal logic. In *TIME'13*, pages 99–106. IEEE Computer Society, 2013. doi:[10.1109/TIME.2013.20](https://doi.org/10.1109/TIME.2013.20).

- [11] Marcello M. Bersani, Matteo Rossi, and Pierluigi San Pietro. An SMT-based approach to satisfiability checking of MITL. *Inf. Comput.*, 245:72–97, 2015. doi:[10.1016/J.IC.2015.06.007](https://doi.org/10.1016/J.IC.2015.06.007).
- [12] Marcello M. Bersani, Matteo Rossi, and Pierluigi San Pietro. A tool for deciding the satisfiability of continuous-time metric temporal logic. *Acta Informatica*, 53(2):171–206, 2016. doi:[10.1007/S00236-015-0229-Y](https://doi.org/10.1007/S00236-015-0229-Y).
- [13] Chaima Boufaied, Maris Jukss, Domenico Bianculli, Lionel C. Briand, and Yago Isasi Parache. Signal-based properties of cyber-physical systems: Taxonomy and logic-based characterization. *J. Syst. Softw.*, 174:110881, 2021. doi:[10.1016/J.JSS.2020.110881](https://doi.org/10.1016/J.JSS.2020.110881).
- [14] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. A simple and flexible way of computing small unsatisfiable cores in SAT modulo theories. In *SAT'07*, volume 4501 of *LNCS*, pages 334–339. Springer, 2007. doi:[10.1007/978-3-540-72788-0_32](https://doi.org/10.1007/978-3-540-72788-0_32).
- [15] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods Syst. Des.*, 19(1):7–34, 2001. doi:[10.1023/A:1011276507260](https://doi.org/10.1023/A:1011276507260).
- [16] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.
- [17] Leonardo de Moura and Nicolaj Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*. Springer, 2008. doi:https://doi.org/10.1007/978-3-540-78800-3_24.
- [18] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *CADE'21*, LNCS, page 625635, Berlin, Heidelberg, 2021. Springer-Verlag. doi:[10.1007/978-3-030-79876-5_37](https://doi.org/10.1007/978-3-030-79876-5_37).
- [19] Adel Dokhanchi, Bardh Hoxha, and Georgios Fainekos. Formal requirement debugging for testing and verification of cyber-physical systems. *ACM Trans. Embed. Comput. Syst.*, 17(2), 2017. ISSN 1539-9087. doi:[10.1145/3147451](https://doi.org/10.1145/3147451).
- [20] Bruno Dutertre and Leonardo M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV 2006*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006. doi:[10.1007/11817963_11](https://doi.org/10.1007/11817963_11).
- [21] Luca Geatti, Nicola Gigante, Angelo Montanari, and Mark Reynolds. One-pass and tree-shaped tableau systems for TPTL and TPTL_b+past. *Inf. Comput.*, 278:104599, 2021. doi:[10.1016/J.IC.2020.104599](https://doi.org/10.1016/J.IC.2020.104599).
- [22] Luca Geatti, Nicola Gigante, Angelo Montanari, and Gabriele Venturato. SAT meets tableaux for linear temporal logic satisfiability. *J. Autom. Reason.*, 68(2):6, 2024. doi:[10.1007/S10817-023-09691-1](https://doi.org/10.1007/S10817-023-09691-1).
- [23] Yoram Hirshfeld and Alexander Moshe Rabinovich. Timer formulas and decidable metric temporal logic. *Inf. Comput.*, 198(2):148–178, 2005. doi:[10.1016/J.IC.2004.12.002](https://doi.org/10.1016/J.IC.2004.12.002).
- [24] Bardh Hoxha, Houssam Abbas, and Georgios Fainekos. Benchmarks for temporal logic requirements for automotive systems. In *ARCH14-15*, volume 34 of *EPiC Series in Computing*, pages 25–30. EasyChair, 2015. doi:[10.29007/xwrs](https://doi.org/10.29007/xwrs).

- [25] Stefan Jaksic, Ezio Bartocci, Radu Grosu, Reinhard Kloibhofer, Thang Nguyen, and Dejan Nickovic. From signal temporal logic to FPGA monitors. In *MEMOCODE '15*, pages 218–227. IEEE, 2015. doi:[10.1109/MEMCOD.2015.7340489](https://doi.org/10.1109/MEMCOD.2015.7340489).
- [26] Xiaoqing Jin, Jyotirmoy V. Deshmukh, James Kapinski, Koichi Ueda, and Ken Butts. Powertrain control verification benchmark. In *HSCC'14*, page 253262, New York, USA, 2014. ACM. ISBN 9781450327329. doi:[10.1145/2562059.2562140](https://doi.org/10.1145/2562059.2562140).
- [27] Jia Lee, Geunyeol Yu, and Kyungmin Bae. Efficient SMT-based model checking for signal temporal logic. In *ASE'21*, pages 343–354. IEEE, 2021. doi:[10.1109/ASE51524.2021.9678719](https://doi.org/10.1109/ASE51524.2021.9678719).
- [28] Jianwen Li. mltlsat. GitHub.com, 2019. URL <https://github.com/lijwen2748/mltlsat>.
- [29] Jianwen Li, Moshe Y. Vardi, and Kristin Y. Rozier. Satisfiability checking for mission-time LTL (MLTL). *Inf. Comput.*, 289(Part):104923, 2022. doi:[10.1016/J.IC.2022.104923](https://doi.org/10.1016/J.IC.2022.104923).
- [30] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *FORMATS/FTRTFT'04*, volume 3253 of *LNCS*, pages 152–166. Springer, 2004. doi:[10.1007/978-3-540-30206-3_12](https://doi.org/10.1007/978-3-540-30206-3_12).
- [31] Beatrice Melani, Ezio Bartocci, and Michele Chiari. A tree-shaped tableau for checking the satisfiability of signal temporal logic with bounded temporal operators. *ACM Trans. Embed. Comput. Syst.*, 24(5s):1–26, 2025. doi:[10.1145/3759917](https://doi.org/10.1145/3759917).
- [32] Beatrice Melani, Ezio Bartocci, and Michele Chiari. Stltree, 2025. URL https://github.com/beamelani/Consistency_Check.
- [33] OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition. <https://openai.com/index/gpt-5-6/>, 2026. Accessed: 2026-08-28.
- [34] Vasumathi Raman, Alexandre Donzé, Mehdi Maasoumy, Richard M. Murray, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. Model predictive control with signal temporal logic specifications. In *CDC'14*, pages 81–87. IEEE, 2014. doi:[10.1109/CDC.2014.7039363](https://doi.org/10.1109/CDC.2014.7039363).
- [35] Thomas Reinbacher, Kristin Y. Rozier, and Johann Schumann. Temporal-logic based runtime observer pairs for system health management of real-time systems. In *TACAS'14*, volume 8413 of *LNCS*, pages 357–372. Springer, 2014. doi:[10.1007/978-3-642-54862-8_24](https://doi.org/10.1007/978-3-642-54862-8_24).
- [36] Mark Reynolds. A new rule for LTL tableaux. In *GandALF'16*, volume 26 of *EPCTS*, pages 287–301, 2016. doi:[10.4204/EPTCS.226.20](https://doi.org/10.4204/EPTCS.226.20).
- [37] Hendrik Roehm, Thomas Heinz, and Eva Charlotte Mayer. Stlinspector: STL validation with guarantees. In *CAV'17*, volume 10426 of *LNCS*, pages 225–232. Springer, 2017. doi:[10.1007/978-3-319-63387-9_11](https://doi.org/10.1007/978-3-319-63387-9_11).
- [38] Sriram Sankaranarayanan and Georgios Fainekos. Falsification of temporal properties of hybrid systems using the cross-entropy method. In *HSCC'12*, pages 125–134. ACM, 2012. doi:[10.1145/2185632.2185653](https://doi.org/10.1145/2185632.2185653).
- [39] Viktor Schuppan. Extracting unsatisfiable cores for LTL via temporal resolution. *Acta Informatica*, 53(3):247–299, 2016. doi:[10.1007/S00236-015-0242-1](https://doi.org/10.1007/S00236-015-0242-1).

- [40] Marco Zamponi, Michele Chiari, and Ezio Bartocci. Replication package for the paper “STLSat—an improved tableau for satisfiability checking of Signal Temporal Logic formulas”, 2026. URL <https://doi.org/10.5281/zenodo.22746464>.
- [41] Marco Zamponi, Michele Chiari, and Florian Lammel. STLSat. GitHub.com, 2026. URL <https://github.com/ZamponiMarco/stlsat>.
- [42] Marco Zamponi, Florian Lammel, Ezio Bartocci, and Michele Chiari. STLSat—An improved tableau for satisfiability checking of Signal Temporal Logic formulas. *CoRR*, abs/2607.21081, 2026. doi:10.48550/ARXIV.2607.21081.