Exploring Formal Verification for First-Order Linear Temporal Logic Modulo Theories

PIC2 - Master in Computer Science and Engineering
Instituto Superior Técnico, Universidade de Lisboa

José João Alves dos Santos Ferreira — 99259*
josejoaoferreira@tecnico.ulisboa.pt

Advisor: José Fragoso Santos, Alessandro Gianola

Abstract Linear Temporal Logic (LTL) is a widely used framework for specifying properties of systems that evolve over time. While it is extremely useful, it has significant limitations in expressiveness that restrict its applicability, particularly in areas such as formal verification. Specifically, it does not allow quantification over described domains or reasoning about theories related to standard structures such as integers and arrays. In recent years, several tools that perform satisfiability checking and model checking of First-Order Linear Temporal Logic (FOLTL), a more expressive variant of LTL that includes quantifiers, have emerged. However, these tools are highly heterogeneous; each is specialized in a specific fragment of FOLTL and often incorporates additional features such as support for various first-order theories. This heterogeneity, both in terms of supported fragments and the solving techniques used, complicates the understanding of how these tools relate to one another. The aim of this work is to help improve this situation by creating a tool-independent benchmark for assessing FOLTL tools and conducting an empirical study to evaluate the existing ones. In this report, we outline the most relevant related work in the field and describe our plans for executing the project.

Keywords — First-order linear temporal logic, Formal verification, Theories

*I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa (https://nape.tecnico.ulisboa.pt/en/apoio-ao-estudante/documentos-importantes/regulamentos-da-universidade-de-lisboa/).

Contents

1 Introduction

Linear Temporal Logic (LTL) [1] provides a powerful framework for reasoning about the behavior of dynamic systems over time. Extending this, First-Order Linear Temporal Logic (FOLTL) [2] adds the ability to capture relationships and properties of objects within these systems. When combined with theories, FOLTL becomes even more expressive, incorporating domain-specific knowledge to model complex systems comprehensively.

Verification of FOLTL formulas is achievable through various methodologies, including satisfiability checking [3] and model checking [4], though their decidability depends on the specific semantics of the logic and the boundedness of the domains of first-order theories in use. For certain restricted cases of FOLTL, verification is decidable, whereas in more general cases, it is only semi-decidable, meaning the procedure may not terminate for all inputs [5].

While there are various tools for verifying FOLTL formulas, these tools differ greatly in the fragments of FOLTL they support. Regarding the time horizon, some interpret formulas over finite words only, while others allow for infinite traces. Some tools support a range of first-order theories, whereas others are limited to a single theory. Similarly, some handle unbounded first-order domains, while others impose bounds. Certain tools allow quantification over domain elements, while others do not. Additionally, these tools are often tailored to specific use cases and application domains, further contributing to their diversity.

As a result, identifying the most suitable tool for a specific task or determining which tool performs best for general needs can be challenging. To address this, we will establish a comprehensive benchmark of formulas for assessing FOLTL verification tools. This process will begin with defining a canonical format for representing FOLTL formulas. Next, we will collect formulas from test suites used to evaluate existing FOLTL tools and translate them into this unified format. Using the benchmark, we will conduct an empirical study to compare the expressiveness, effectiveness, and performance of the most relevant FOLTL tools.

We expect the contributions of this work to be as follows:

1. A survey and investigation of state-of-the-art tools for first-order linear temporal logic verification, focusing on their differences and selecting a subset of these tools for detailed assessment. This assessment will evaluate their capabilities, available documentation, strengths, weaknesses, and identify potential gaps or opportunities for improvement.
2. The definition of a canonical format for first-order linear temporal logic formulas that abstracts common fragments supported by various tools. This format will serve as the foundation for an interface that translates these formulas into the different specification languages used by tools.
3. The development of a comprehensive benchmark for evaluating these tools. This benchmark will include relevant test suites, ensuring the evaluation covers a wide range of scenarios and provides meaningful insights into the performance and functionality of these tools.
4. An empirical study comparing the expressiveness, effectiveness, and performance of the most relevant tools on the established benchmark. This study will also include detailed characterization of these tools across a broad range of parameters, offering a thorough comparison of their features and capabilities.

The remainder of this report is structured as follows. Section 2 presents the logical foundations, syntax, semantics, and the verification problem for first-order linear temporal logic formulas, theories, as well as some explanations regarding key concepts of satisfiability and model checking. Section 3 presents some of the state-of-the-art tools used for the verification of temporal logic and first-order formulas, comparing them and experimenting with some. Section 4 outlines our proposed solution, which is then integrated into a work schedule detailed in Section 5. Finally, Section 6 concludes the report, followed by a list of references.

2 Background

This Section provides the context for understanding FOLTL by outlining its logical foundations and how they contribute to the extension that is FOLTL. It sets the stage for subsequent technical discussions, including FOLTL's syntax, semantics, satisfiability checking, model checking, and key concepts in formal verification of these formulas.

2.1 First-Order Linear Temporal Logic

FOLTL represents an extension of earlier logical systems, combining the strengths of classical logic, temporal reasoning, and first-order logic to address the dynamic nature of complex systems that evolve over time. To understand it, it is essential to consider its roots in these previously defined logical systems.

Classical propositional logic. PL is a branch of logic that deals with propositions — statements represented in a formal language using symbols (e.g., p, q) that are either *true* or *false*, as well as logical operators ($\wedge$ for *conjunction*, $\rightarrow$ for *implication*, $\leftrightarrow$ for *equivalence*, $\top$ for *true*, $\bot$ for *false*, among others), derived from the basic logical connectives $\neg$ for *negation* and $\vee$ for *disjunction*. Combined, they form compound propositions and provide a framework for reasoning about the truth or falsity of statements, serving as the foundation for *first-order* and *higher-order logics*.

First-order logic. FOL extends *propositional logic*, incorporating all its logical operators, by introducing predicates (e.g., $p(t_1, t_2)$, $q(u_1, u_2, u_3)$), variables (e.g., x, y) and quantifiers ($\forall$ for *universal quantification* and $\exists$ for *existential quantification*), allowing for the expression of statements about objects within a domain. FOL provides a framework for reasoning about properties and relationships, enabling the formalization of more complex statements compared to *propositional logic*.

Linear temporal logic. LTL [1] is a form of modal logic and a propositional logic framework designed to reason about the evolution of a finite set of propositional variables (e.g. p, q) over time. It achieves this by incorporating temporal operators (F for *finally*, G for *globally* and R for *release*, among others, which vary across fragments), which are defined in terms of the fundamental operators X for *next* and U for *until*, along with the logical operators inherited from *propositional logic*.

First-order linear temporal logic. FOLTL [2] is a *first-order temporal logic* that extends *linear temporal logic* with *first-order logic* by allowing the use of predicates, variables, and quantifiers in sentences containing temporal operators, enabling reasoning about properties of objects, the relationships between objects in a domain, and their behaviors over time. FOLTL has the greatest *expressive power* among all the logics previously presented.

Figure 1 illustrates the differences in expressive power between these logics across two distinct axes: higher-order features and temporal reasoning.

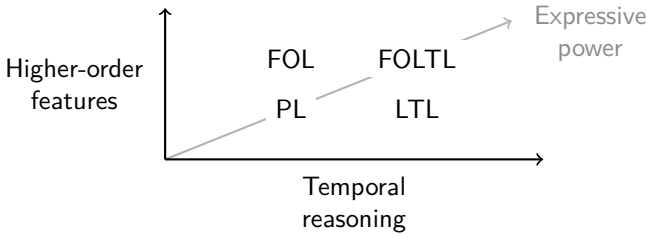


Figure 1: The increase in expressive power provided by FOLTL.

To give a broad and general idea of what can be expressed in FOLTL, some examples and their respective explanations are given below:

$$\mathsf{G}(\forall country\,.\,\exists person\,.\,HeadOfState(person, country))$$

It is always the case that for every country, there exists a person who is the head of state of that country.

$$\forall person\,.\,((ItIsRaining() \wedge \neg\exists umbrella\,.\,Holding(person, umbrella)) \rightarrow \mathsf{F}(\mathsf{IsWet}(person)))$$

For all people, if it is raining and there does not exist an umbrella that the person is holding, then that person will eventually become wet.

$$\mathsf{G}(IsEmail(e) \wedge IsPerson(p)) \wedge (BeingWritten(e, p)\ \mathsf{U}\ (SentBy(e, p) \wedge \mathsf{X}(Delivered(e) \vee Failed(e))))$$

There exists an email and a person, both of which remain valid entities, such that the person continues writing the email until it is sent, after which it is either successfully delivered or fails to send.

2.1.1 Syntax

Let $\mathcal{S}$ be a set of sort symbols, $\mathcal{C}$ a set of constant symbols, $\mathcal{V}$ a set of free variable symbols, $\mathcal{W}$ a set of quantified variable symbols, $\mathcal{F}$ a set of function symbols, and $\mathcal{P}$ a set of predicate symbols, where these sets are all mutually disjoint. Each constant in $\mathcal{C}$, and each variable in $\mathcal{V}$ and $\mathcal{W}$, is assigned a sort s, which is a symbol in $\mathcal{S}$ representing its type or category. Similarly, the domain of each predicate in $\mathcal{P}$, and the domain and codomain of each function in $\mathcal{F}$, are specified by sorts in $\mathcal{S}$.

Given a first-order signature $\Sigma = \mathcal{S} \cup \mathcal{C} \cup \mathcal{V} \cup \mathcal{W} \cup \mathcal{F} \cup \mathcal{P}$, the syntax of Σ-terms is defined as follows:

$$\begin{aligned} t \coloneqq\ & c \mid x \mid y \mid f(t_1, \ldots, t_m) \\ & \mid x' \end{aligned}$$

where $c \in \mathcal{C}$ is a constant symbol, $x \in \mathcal{V}$ is a free variable symbol, $y \in \mathcal{W}$ is a quantified variable symbol, $f \in \mathcal{F}$ is an m-ary function symbol, each t_i (for any $i \in \{1, \ldots, m\}$) is a Σ-term, and x' is the *next* x. When $m = 0$, the function has arity 0, effectively being equivalent to a constant. It is important to distinguish between variables in $\mathcal{V}$ and $\mathcal{W}$, as it does not make sense to apply x', which denotes the value of x in the *next* state, to a quantified variable.

The syntax of Σ-formulas is defined as follows:

$$\begin{aligned} \phi \coloneqq\ & p(t_1, \ldots, t_n) \mid t_1 = t_2 \mid \neg\phi \mid \phi \vee \phi \mid \exists y\,.\,\phi \\ & \mid \mathsf{X}\,\phi \mid \phi\ \mathsf{U}\ \phi \end{aligned}$$

where $p \in \mathcal{P}$ is an n-ary predicate symbol, each t_i (for any $i \in \{1, \ldots, n\}$) is a Σ-term, $\exists$ is the existential quantifier symbol, $y \in \mathcal{W}$ is a quantified variable symbol, and the temporal operators X and U are called *next* and *until*, respectively. One may also consider the *past operators* Y (*yesterday*) and S (*since*), but these have been omitted here for the sake of simplicity. When $n = 0$, the predicate has arity 0, effectively being equivalent to a proposition.

Derived constructs ($\wedge, \rightarrow, \leftrightarrow, \top, \bot, \forall, \mathsf{F}, \mathsf{G}, \mathsf{R}$) are formally defined in Section 2.1.3.

2.1.2 Semantics

FOLTL supports both bounded and unbounded semantics in the context of time (see Section 2.3.1). The following semantics of the FOLTL fragment are defined in an unbounded context and are based on those presented by Geatti et al. [6]. Bounded semantics can be derived using the standard technique of Biere et al. [7].

Let Σ-word be the sequence $\bar{\sigma} = \langle \sigma_0, \sigma_1, \cdots \rangle$, where each σ_i is a first-order *structure* over the signature Σ, referred to as a Σ-structure — think of it as a "snapshot". An *environment* is a mapping ξ that associates each first-order variable, function, or constant symbol of sort s with a value from its corresponding domain $\mathcal{D}(s)$.

Let t be a Σ-term and σ_i a Σ-structure. The evaluation of t on σ_i with environment ξ, denoted as ${t}^{\sigma_i,\xi}$, is the following:

1. ${c}^{\sigma_i,\xi} = c^{\sigma_i}$
2. ${x}^{\sigma_i,\xi} = \xi_i(x)$
3. ${y}^{\sigma_i,\xi} = \xi_i(y)$
4. ${f(t_1, \ldots, t_m)}^{\sigma_i,\xi} = f^{\sigma_i}({t_1}^{\sigma_i,\xi}, \ldots, {t_m}^{\sigma_i,\xi})$
5. ${x'}^{\sigma_i,\xi} = x^{\sigma_{i+1},\xi}$

where $c \in \mathcal{C}$ is a constant symbol, $x \in \mathcal{V}$ is a free variable symbol, $y \in \mathcal{W}$ is a quantified variable symbol, $f \in \mathcal{F}$ is an m-ary function symbol, and each t_i (for any $i \in \{1, \ldots, m\}$) is a Σ-term. Given a free variable x, the interpretation of x' in a non-empty Σ-word $\bar{\sigma}$ at time point i is equal to the value of x assigned over $\bar{\sigma}$ at time point $i+1$ in the environment ξ.

Let ϕ be a FOLTL formula and σ_i a Σ-structure. The interpretation of ϕ on σ_i is represented as ϕ^{σ_i}. The *satisfaction* of ϕ over a non-empty Σ-word $\bar{\sigma}$, with environment ξ at a time point $i \in \mathbb{N}$, denoted as $\bar{\sigma}, \xi, i \models \phi$, is defined inductively as follows:

1. $\bar{\sigma}, \xi, i \models p(t_1, \ldots, t_n)$ iff $({t_1}^{\sigma_i,\xi}, \ldots, {t_n}^{\sigma_i,\xi}) \in p^{\sigma_i}$
2. $\bar{\sigma}, \xi, i \models t_1 = t_2$ iff ${t_1}^{\sigma_i,\xi} = {t_2}^{\sigma_i,\xi}$
3. $\bar{\sigma}, \xi, i \models \neg\phi$ iff $\bar{\sigma}, \xi, i \not\models \phi$
4. $\bar{\sigma}, \xi, i \models \phi_1 \vee \phi_2$ iff $\bar{\sigma}, \xi, i \models \phi_1$ or $\bar{\sigma}, \xi, i \models \phi_2$
5. $\bar{\sigma}, \xi, i \models \exists y \,.\, \phi$ iff there exists a $v \in \mathcal{D}(s)$, where s is the sort of y, such that $\bar{\sigma}, \xi[y \mapsto v], i \models \phi$
6. $\bar{\sigma}, \xi, i \models \mathsf{X}\,\phi$ iff $\bar{\sigma}, \xi, i+1 \models \phi$
7. $\bar{\sigma}, \xi, i \models \phi_1 \;\mathsf{U}\; \phi_2$ iff there exists a $k \geq i$ such that $\bar{\sigma}, \xi, k \models \phi_2$ and $\bar{\sigma}, \xi, j \models \phi_1$ for all j with $i \leq j < k$

Item 5 states that the formula $\exists y \,.\, \phi$ is satisfied at time point i if and only if there exists a value v within the domain of the quantified variable y, as defined by the interpretation, such that when y is assigned the value v in the environment ξ, ϕ is satisfied under this modified environment $\xi[y \mapsto v]$.

Informally, item 6 states that, in order for the formula $\mathsf{X}\,\phi$ (read "next ϕ") to be satisfied at time point i, ϕ must be true at the following time point in the Σ-word, i.e., at $i+1$. On the other hand, item 7 states that, in order for the formula $\phi_1 \;\mathsf{U}\; \phi_2$ (read "ϕ_1 until ϕ_2") to be satisfied at time point i, ϕ_1 must hold continuously from time i until time k, at which point ϕ_2 becomes true.

2.1.3 Derived constructs

The following standard abbreviations are adopted:

$$\begin{aligned}
\phi_1 \wedge \phi_2 &\coloneqq \neg(\neg\phi_1 \vee \neg\phi_2) \\
\phi_1 \rightarrow \phi_2 &\coloneqq \neg\phi_1 \vee \phi_2 \\
\phi_1 \leftrightarrow \phi_2 &\coloneqq (\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1) \\
\top &\coloneqq p \vee \neg p \\
\bot &\coloneqq \neg\top \\
\forall y \,.\, \phi &\coloneqq \neg(\exists y \,.\, \neg\phi) \\
\mathsf{F}\,\phi &\coloneqq \top\ \mathsf{U}\ \phi \\
\mathsf{G}\,\phi &\coloneqq \neg(\mathsf{F}\,\neg\phi) \\
\phi_1\ \mathsf{R}\ \phi_2 &\coloneqq \neg(\neg\phi_1\ \mathsf{U}\ \neg\phi_2)
\end{aligned}$$

The formula $\forall y \,.\, \phi$ is satisfied at time point i if and only if for every value v within the domain of the quantified variable y, as defined by the interpretation, ϕ is satisfied under the modified environment $\xi[y \mapsto v]$, where y is assigned the value v.

Informally, in order for the formula $\mathsf{F}\,\phi$ (read "eventually ϕ") to be satisfied at time point i, ϕ must hold at time point i or at some future time point $k > i$ in the Σ-word. For the formula $\mathsf{G}\,\phi$ (read "globally ϕ") to be satisfied at time point i, ϕ must be true at every time point from i onward in the Σ-word. Finally, in order for the formula $\phi_1\ \mathsf{R}\ \phi_2$ (read "ϕ_1 release ϕ_2") to be satisfied at time point i, ϕ_2 must hold at every time point from i onward in the Σ-word, either until (and including) some time k when ϕ_1 holds, or indefinitely if ϕ_1 never holds.

2.2 Formal verification of FOLTL formulas

It is crucial to distinguish between the often-confused concepts of validation and verification.

Validation involves assessing whether a system or formula is meaningful and aligns with its intended purpose. It ensures that it accurately reflects real-world requirements or expectations, focusing on its relevance and appropriateness rather than its correctness.

Verification, on the other hand, involves ensuring the correctness of a system and/or formula within a specific context. In the case of FOLTL formulas, this includes approaches like *satisfiability checking*, where the formula's truth is confirmed in a concrete run, and *model checking*, where the system model is demonstrated to behave as expected according to the defined rules.

The concepts of satisfiability checking and model checking will now be explored in more detail.

2.2.1 Satisfiability checking

Satisfiability checking [3] is the process of determining whether a given logical formula is consistent, meaning that there exists at least one model or interpretation in which the formula evaluates to true. Techniques for satisfiability checking leverage boolean satisfiability problem (SAT) [8] solvers for propositional logic formulas and sophisticated SAT-modulo-theories (SMT) [9] solvers to handle first-order formulas over a wide range of theories (see Section 2.3.2). This general framework extends naturally to logics such as FOLTL.

Given a FOLTL formula ϕ, ϕ is *satisfiable* if and only if there exists a non-empty Σ-word $\bar{\sigma}$ and an environment ξ such that $\bar{\sigma}, \xi, 0 \models \phi$, i.e., if ϕ holds at the initial time point of the word. In the case of a purely first-order formula ϕ, which contains neither temporal operators nor terms involving x', satisfiability checking reduces to determining whether there exists a single Σ-structure σ and an environment ξ such that $\sigma, \xi \models \phi$.

Satisfiability checking does not require a given Σ-word or any other specific system to check against, and it is foundational for model checking, automated theorem proving, and other formal verification tasks.

The problem of determining whether a FOLTL formula is satisfiable is known to be highly undecidable, i.e., not even recursively enumerable [5], as it combines first-order logic, which is undecidable due to its quantification over infinite structures, with temporal logic, resulting in an expressive system that defies a universal decision procedure.

2.2.2 Model checking

Model checking [4] is an automated technique that, unlike satisfiability checking, relies on the existence of a finite-state model of a system as a basis for verifying whether a given property or specification holds. If a property is violated, a counterexample is generated in the form of a trace, which is very useful for debugging.

Properties [10] can be categorized as:

- *Safety*: asserts that something bad should never occur; it can describe system invariants and is expressed using the *globally* operator, in the form $\mathsf{G}(\phi)$.
- *Liveness*: ensures that something desirable will eventually occur and is often represented using the *globally* and the *finally* operator, in the form $\mathsf{G}(\phi_1 \rightarrow \mathsf{F}\phi_2)$.
- *Fairness*: ensures that all relevant actions or events have a chance to occur and no action is perpetually ignored or prevented in an infinite execution.

Let $\mathcal{M}$ be a finite transition system of some kind. In the case of FOLTL formulas, given a model $\mathcal{M}$ and a FOLTL formula ϕ, ϕ is *model checked*, i.e., $\mathcal{M} \models \phi$, if and only if there does not exist, for a time point $i \in \mathbb{N}$, a Σ-word $\bar{\sigma}$ and an environment ξ such that, $\mathcal{M}, \bar{\sigma}, \xi, i \models \neg\phi$, i.e., if ϕ holds for every execution of the specific $\mathcal{M}$ at all time points in the $\bar{\sigma}$ word.

Model checking typically involves verifying whether every possible execution of a system satisfies a given property ϕ, referred to as *universal model checking*. However, in some cases, the objective is to verify whether at least one execution satisfies ϕ (*existential model checking*), essentially becoming a satisfiability problem. Negation can be applied to prove that the property always holds, as successfully demonstrating that no run satisfies $\neg\phi$ is equivalent to proving that every run satisfies ϕ.

Verification through model checking is partial, i.e., it involves verifying individual properties independently without requiring a comprehensive specification. It verifies a system model rather than the actual system, so results depend on the model's accuracy and cover only stated requirements, with no guarantee of completeness. Its applicability is limited by decidability issues, and for reasoning about abstract data types with undecidable or semi-decidable logics, model checking is generally not computable.

Model checkers often take a long time to verify properties, or may even fail to terminate due to the "state-space explosion" problem, where the number of generated states to explore the model exceeds available memory. For these reasons, model checking is typically used to identify errors rather than prove their absence.

The model, which addresses how the system behaves, and the specification, which prescribes what the system should and should not do, are made using model description/specification languages of the model checker at hand. When a property is violated, the counterexample describes an execution path that leads from the initial system state to a state that violates the property being verified. With the help of a simulator, the user can replay the violating scenario, in this way obtaining useful debugging information, and adapt the model or the property accordingly.

Figure 2 depicts the model checking process.

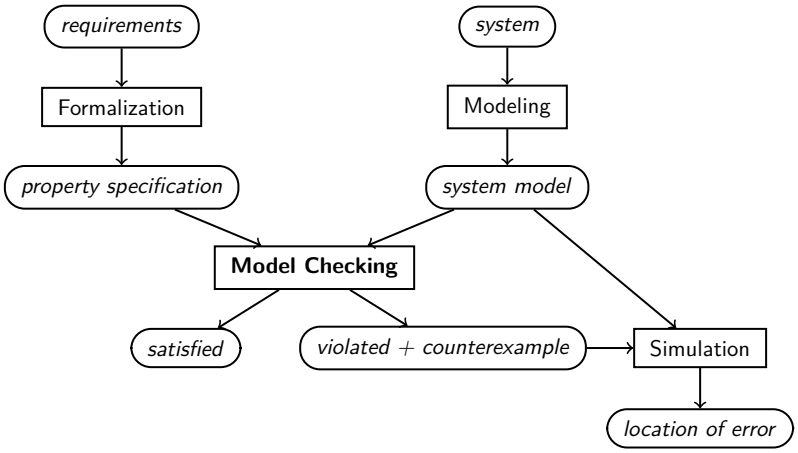


Figure 2: Schematic view of the model checking approach to verification [10].

2.3 Key concepts for FOLTL verification

In order to be able to express formulas such as the following example:

$$x = 0 \wedge \mathsf{G}(x' = x + 1) \wedge \mathsf{F}(x = 25)$$

The value of x starts at 0, it is always the case that it increases by 1 at each subsequent time step, and eventually, it will reach 25.

the presented FOLTL fragment would need to be extended to support arithmetic, specifically the *linear integer arithmetic* (LIA) theory, and possibly be adapted to finite time semantics, as verifying this for infinite words may be impractical. The following subsections introduce some key concepts that have not been discussed yet and common extensions to FOLTL fragments.

2.3.1 Time

For the given cases and interests, time is considered to be *linear*, *qualitative*, and *discrete*, i.e., interpreted over the natural numbers ($\mathbb{N}$), which is a common approach in tools for first-order linear temporal logic verification. A critical distinction is the treatment of infinite versus finite time, with growing interest in the latter [11]. As a result, when evaluating a formula, it is essential to determine whether verification is to be performed over infinite or finite sequences of time points.

Formally, in the context of FOLTL formulas, a formula ϕ can be verified over a finite word $\bar{\sigma} = \langle \sigma_0, \sigma_1, \cdots \rangle$ or an infinite word $\bar{\sigma} = \langle \sigma_0, \cdots, \sigma_o \rangle$, where $o \in \mathbb{N}$, with the semantics for infinite words provided in Section 2.1.2.

Finite temporal reasoning is particularly useful in contexts like bounded model checking [12], where properties are verified over a fixed time horizon, making the process computationally more tractable. In contrast, reasoning about infinite time requires techniques capable of handling infinite traces, which often introduce additional complexity.

2.3.2 First-order theories

Extensions are added on top of first-order logics, and similarly, they can be integrated with FOLTL. Some of the most commonly used theories include *linear integer arithmetic* (LIA), *linear real arithmetic* (LRA), *equality with uninterpreted functions* (EUF), arrays, bit-vectors, strings, and many other forms of arithmetic, such as floating-point and Presburger's [13], which is fully decidable despite being less expressive than full arithmetic. Many tools provide built-in support for these first-order theories to enhance reasoning capabilities.

A careful reader might question the introduction of theories in FOLTL, as everything expressible by theories can also be handled with sets of first-order axioms in a *top-down* manner. However, in SMT literature, the norm is a *bottom-up* approach: SAT solvers are extended with theory solvers and specialized decision procedures tailored to specific theories and use cases. Top-down approaches that operate directly on pure FOLTL often result in more complex and less efficient solutions, whereas the most successful methods leverage the bottom-up approach.

2.3.3 First-order domains

Domains can be unbounded, allowing variables to take on unlimited values. This enables more expressive reasoning but can complicate verification, potentially leading to undecidability in some cases. In contrast, bounded domains are finite, with variables restricted to a limited set of values. This simplification makes reasoning more manageable and decidable, as only a finite number of possibilities need to be evaluated.

In the context of the FOLTL formalism, this distinction is reflected in the size of the domains for each sort $s \in \mathcal{S}$. For example, a bounded s might be "State" with a finite state space, while an unbounded s could be "Natural" with an infinite domain of natural numbers ($\mathbb{N}$) or "Real" for the real numbers ($\mathbb{R}$).

The distinction between bounded and unbounded domains is crucial for determining the computational complexity and decidability of first-order theories integrated with temporal reasoning. Tools often employ different techniques for each case, with some focusing on bounded domains for practical reasons. Understanding these distinctions is crucial when verifying first-order temporal logic properties.

2.3.4 Support for primed evaluation

Primed evaluation refers to the ability to explicitly reason about the "next" state of free first-order variables in temporal logic by using primed versions of variables (e.g., x') to represent their values in the subsequent state. This allows for direct expressions of state transitions, such as $x' = x + 1$, enabling a structured and intuitive way to model dynamic systems. This is encompassed within the FOLTL fragment described in Section 2.1.

2.3.5 Support for quantifiers

Quantifier expressiveness refers to a logic's ability to handle existential ($\exists$) and universal ($\forall$) quantifiers over variable symbols. This enables reasoning about properties that hold for some or all elements in a domain. In FOLTL, quantifiers allow for statements about groups of objects or states while maintaining compatibility with temporal operators. However, formulas with temporal operators inside quantifier constructs are generally undecidable.

2.3.6 Algorithms and language-theoretic methods

There is a wide variety of algorithms used for formal verification, such as CDCL [14], and DPLL(T) [15] for satisfiability checking, and BDDs [16], BMC [12], and CEGAR [17] for model checking, among many others. Language-theoretic techniques, like *tableau methods* [18] and *automata-based approaches* [19], can also be integrated with specific algorithms to verify FOLTL formulas. Different tools employ various and different combinations of language-theoretic methods, decision procedures and algorithms.

3 Related Work

This Section presents examples and descriptions of tools that perform FOLTL verification. It then compares them in a concise table and provides an example encoded in two of these tools to illustrate their differences.

3.1 Tools for FOLTL verification

Several tools have been developed for satisfiability checking, model checking, and formal verification of formulas that combine temporal logic with first-order logic. The following section provides an overview of some of these tools.

3.1.1 BLACK

BLACK [20], short for *Bounded LTL sAtisfiability ChecKer*, is a software tool that stands out as a fast, stable, flexible, and portable alternative to satisfiability checking of LTL formulas. BLACK supports both future and past temporal operators (LTL+P) [21], propositional linear temporal logic formulas interpreted both on infinite and finite words (LTL$_f$) and first-order linear temporal logic formulas with theories LIA, LRA, and EUF, interpreted on finite words (LTL$_f{}^{MT}$) [22], while infinite-trace semantics are not supported for these formulas. It allows the use of quantifiers, with the trade-off that temporal operators cannot appear in their scope. Model checking is still not explicitly supported in BLACK, even though it can be reduced to a satisfiability problem.

BLACK uses an incremental encoding approach based on SAT for LTL formulas and SMT for LTL$_f{}^{MT}$ formulas, inspired by the one-pass and tree-shaped tableau method proposed by Reynolds [23]. Given a formula ϕ, the tool encodes the branches of the tableau for ϕ into SAT or SMT formulas, depending on the logic, up to depth k, for increasing values of k,until an accepted branch is found or a *witness of unsatisfiability* is detected. When it comes to the supported propositional linear temporal logics (LTL, LTL+P, and LTL$_f$), the algorithm underlying BLACK is *complete*, i.e., it is guaranteed to provide a definitive result for any given formula, and the extraction of a model for satisfiable formulas is easily supported, as models of the formula are trivially derived from the tableau branches. The option to print the model of the formula, when satisfiable, is not (yet) supported for LTL$_f{}^{MT}$ formulas.

A variety of backends are supported, such as MathSAT [24], Z3 [25], cvc5 [26], MiniSAT [27] and CryptoMiniSAT [28]. It is implemented in C++17 as a shared library over the available SAT solvers, with a well-defined API that can be invoked in client applications, with the tool itself being a thin command-line wrapper over the library.

The syntax, semantics and use cases of the LTL$_f{}^{MT}$ logic, along with its SMT encoding and implementation of the semi-decision procedure in BLACK, are well documented [22]. The fragment of LTL$_f{}^{MT}$ closely resembles the one presented in Section 2.1, with the key differences being the inclusion of weak temporal operators and a weak next constructor for free variables, tailored for finite traces, and past temporal operators.

Brief examples of BLACK's specification language and command-line interface for propositional and first-order logic are given below, respectively:

```
$ black solve -m -f '!p & X !p & F p'
```

```
SAT
Model:
- t = 0: {¬p}
- t = 1: {¬p}
- t = 2: {p} ← loops here
```

```
$ black solve -s -d Int -f 'x = 0 & (wnext(x) = x + 1) & X(x = 0)'
```

```
UNSAT
```

3.1.2 nuXmv

nuXmv [29], an evolution of nuSMV [30], is a symbolic bounded and unbounded model checker widely used as a back-end verification engine across various domains. It employs the SMV specification language, which provides a symbolic representation of transition systems, and supports a comprehensive suite of model checking algorithms.

For finite-state systems, such as purely propositional configurations or configurations with bounded first-order domains, two main model checking techniques are employed: BDD-based algorithms [16], which allow users to specify hints inspired by guided reachability [31]; and SAT-based model checking, using the MiniSAT solver [27] extended with interpolation-based invariant checking [32] and incorporating additional algorithms based on IC3 [33] and k-liveness [34].

For infinite-state systems, which involve configurations with unbounded first-order domains, nuXmv extends the capabilities of nuSMV by supporting theories such as LIA, LRA, and EUF. It provides several techniques for verifying these systems, including abstraction refinement methods like CEGAR [17], leveraging k-induction and IC3. Additionally, nuXmv employs algorithms grounded in satisfiability modulo theories (SMT) [9], using the MathSAT [24] solver. These methods integrate bounded model checking (BMC) with k-induction [35] and combine k-liveness with IC3 [36]. While these approaches are generally incomplete, they guarantee the detection of counterexamples with a loop structure when such counterexamples exist.

The representation of values of free variables at the next time point, for instance x', is supported, but the representation of existential and universal quantifiers using the LTL fragment is not supported. The temporal logic fragment supported by nuXmv is described in [37]. For every LTL specification, the tool constructs a tableau representing behaviors that falsify the property, which is then synchronously composed with the model.

nuXmv has been implemented in C and C++ and is distributed as binary code, available free of charge for academic research and non-commercial purposes. It is possible to generate an explicit state representation of the model under verification in XMI format, which can be visualized using any UML-based viewer that supports it.

nuXmv has been used for requirements analysis, contract-based design, safety assessments, model checking of hybrid systems and software [29]. Applications span diverse areas, such as railway interlocking systems and projects conducted by the European Space Agency.

3.1.3 Alloy Analyzer

Alloy [38] is a lightweight formal specification language based on what its authors call relational logic, an extension of first-order logic, excelling at expressing structural properties. Its latest version integrates LTL constraints and combines first-order with temporal logics, an extension previously implemented in Electrum [39], making it suitable for modeling systems with both structural and dynamic aspects.

The Alloy Analyzer is a model finder and bounded model checker that functions as a compiler to SAT solvers. It generates graphical representations of sets and relations, the tool's internal representations, from signature declarations. Alloy users often take an agile approach, incrementally building the model and verifying its properties during development, rather than constructing the entire model upfront.

The Alloy specification language and the FOLTL fragment it employs are available at [40].The `run` command invokes the analyzer to find a matching model (similar to existential model checking in a way), while the `check` command seeks a counterexample to a given constraint, performing model checking.

Alloy supports the representation of values at the next time point, enabling variable temporal reasoning. To enforce bounded model checking, a finite, adjustable numerical range is imposed, i.e., the support for integers, the only theory Alloy natively provides, is limited to a bound. This is also the case for the total number of model signatures under consideration. While Alloy supports the expression of quantifiers, which are highly useful for property verification, they are inherently limited as they quantify only over bounded first-order domains, making them less general than true first-order quantification.

Alloy traces are infinite yet periodic, represented as traces where the final state loops back to itself for finite time or as finite lasso traces for infinite time. For a finite time horizon (e.g., `M..N steps`),

only lasso traces with at least `M` and at most `N` transitions are explored using bounded model checking. For an infinite time horizon (e.g., `1..steps`), complete model checking may be employed, i.e., model checking over all possible traces without preemptively bounding them, requiring the solver to explore the entire state space. While complete model checking is theoretically guaranteed to terminate, it may fail in practice due to memory limitations or long execution times, as the state spaces of Alloy models can grow quite large. Consequently, this option is best suited for small models or when bounded time horizon checks can no longer uncover counterexamples.

Currently, Alloy supports NuSMV and nuXmv for both complete and bounded model checking, though these tools need to be installed separately. Additionally, it is compatible with various solvers, including MiniSat [27], SAT4J, PLingeling, and a JNI-based SAT solver, among others.

3.1.4 linDMT

linDMT [41] is a model checker designed for verifying so-called DMTs, *data-aware processes modulo theories*, against certain LTL properties over finite words. The FOLTL fragment it uses is a first-order extension of LTL where the atoms are constraints in some first-order theory. The tool interfaces with cvc5 [26] for SMT solving and is implemented in Python.

linDMT employs a semi-decision procedure that combines automata-theoretic techniques for linear temporal properties with quantifier elimination methods to handle datatypes. A crucial aspect of this approach is the presence of an abstract semantic property that ensures a faithful finite-state abstraction of the original model. If this property holds, the method effectively becomes a decision procedure.

To use the linDMT web interface, one must provide a DMT, denoted as $\mathcal{M}$, which can be thought of as the model. This is specified in a simple JSON format that includes states, transitions, variables, relations, and functions. Additionally, a temporal logic property in the form of a formula, denoted as ϕ, must be provided.

To perform model checking, linDMT constructs a constraint graph that represents a faithful finite-state abstraction of $\mathcal{M}$, an NFA (non-deterministic finite automaton) construction for the formula ϕ (denoted as $\mathcal{N}_\phi$), and the product automaton $\mathcal{N}^\phi_\mathcal{M}$, which combines the two. If the product automaton contains a final state, the tool then extracts a witness for the given formula ϕ from the automaton.

If the tool succeeds, the results are displayed across multiple tabs. The first tab shows a graphical representation of the DMT $\mathcal{M}$. The second tab presents $\mathcal{N}_\phi$, while the third tab displays $\mathcal{N}^\phi_\mathcal{M}$. The final tab contains the output, including a run that witnesses the property ϕ, if such a witness exists.

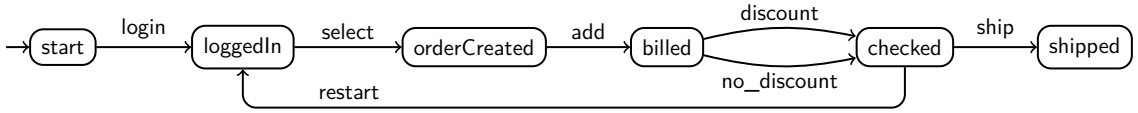
3.2 Comparison of FOLTL tools

The following table presents a comparison among the tools discussed in Section 3.1.

	Typology	Time horizon	Theories	Domains	Primed variables	Quantifiers	Language-theoretic methods	Decision procedures	Applications
BLACK	satisfiability checker	finite/ infinite	LIA, LRA, EUF	unbounded	✓	✓	tableau	SAT, SMT-based	formal verification, artificial intelligence
nuXmv	model checker	finite/ infinite	LIA, LRA, EUF	bounded/ unbounded	✓	×	tableau	BDD, SAT, SMT-based	safety assessments, hybrid and railway interlocking systems
Alloy Analyzer	model finder/ checker	finite/ infinite	LIA	bounded	✓	×	tableau	SAT-based	software and network design exploration
linDMT	model checker	finite	LIRA, EUF	unbounded	✓	×	automata	SMT-based	data-aware processes, database systems

3.3 Detailed example

Consider a *webshop* model, directly extracted from [41]. A customer logs in using credentials stored in a customer database, and then selects 5 products from a product database, creating an order. After billing, the system checks if the customer qualifies for a 20% discount based on their status, applying it if eligible. The order is then shipped if the account balance is sufficient; otherwise, the process restarts from the beginning. The system is modeled as a guarded transition system:



$\mathsf{login}\colon\ \mathsf{cust}(c^w, a^w, v^w)$ $\quad \mathsf{select}\colon\ \bigwedge_{i=1}^{5} \mathsf{id}(p_i^w)$ $\quad \mathsf{add}\colon\ t^w = \Sigma_{i=1}^{5} f_{\mathsf{price}}(p_i^r)$
$\mathsf{discount}\colon\ v^r \wedge t^w = t^r - \frac{1}{5}t^r$ $\quad \mathsf{no\_discount}\colon\ \neg v^r \wedge t^w = t^r$ $\quad \mathsf{ship}\colon\ t^r \leq a^r$
$\mathsf{restart}\colon\ \top$

Variables c, a, and v represent the customer ID, account balance, and eligibility for discounts, respectively, while p_i denotes different product IDs, and t represents the total order cost. These variables are either read (^{r}) or written (^{w}) based on the specified transition guards. The relations $\mathsf{cust}(customer\_id, account, is\_vip)$ and $\mathsf{id}(product\_id)$ and the function $f_{\text{price}}(product\_id)$ are backed by an underlying database structure that stores and manages customer and product information. Key verification properties include, for example:

(a) ensuring that discounts are only applied to customers who are eligible (e.g., those marked as VIPs) — a safety property
(b) guaranteeing that every order, when created, is eventually shipped or restarted — a liveness property
(c) guaranteeing that no order is ever shipped to a non-VIP customer if their account balance is insufficient to cover the order cost — a safety property

3.3.1 Implementation

To highlight the difference between verifying propositional LTL and first-order LTL formulas, we present the webshop example detailed above, modeled in each case using BLACK (see Section 3.1.1).

First, a simplified version of the webshop is modeled in Figure 3 using propositional LTL. The formula captures only the finite-state machine and the property being checked, without incorporating relations or transition guards. This simplification was used to test control flow and model visualization. BLACK searches for one infinite word that satisfies the given property. In this case, the property is a reachability condition, verifying whether the final state can eventually be reached based on the provided model.

Figure 4 illustrates the webshop modeled using first-order LTL, the logic that is the focus of our work. The formula now incorporates the first-order concepts missing in the previous example. The term `wnext(x)` is essentially the same as x', except that it may evaluate to true if the next time point does not exist, making it particularly useful for finite words. We are verifying the safety property described in (c), which ensures that no order is shipped to a non-VIP customer if their account balance is less than the order cost. Because BLACK is a satisfiability checker, in order to prove that a property ϕ holds for all model states, we must confirm that no witnesses for its negation ($\neg\phi$) are found. Since BLACK is semi-decidable for first-order logic and no witness was found within a reasonable time for the given model, we conclude that the property holds.

```
G(start -> !(loggedIn | orderCreated | billed | checked | shipped)) &
G(loggedIn -> !(orderCreated | billed | checked | shipped | start)) &
G(orderCreated -> !(billed | checked | shipped | start | loggedIn)) &
G(billed -> !(checked | shipped | start | loggedIn | orderCreated)) &
G(checked -> !(shipped | start | loggedIn | orderCreated | billed)) &
G(shipped -> !(start | loggedIn | orderCreated | billed | checked)) &

G(start -> X loggedIn) & G(loggedIn -> X orderCreated) & G(orderCreated -> X billed) &
G(billed -> X checked) & G(checked -> X((shipped | loggedIn) & !(shipped & loggedIn))) &

start & F(shipped)
```

```
$ black solve -m -c webshop.pltl
```

```
SAT
Model:
- t = 0: {¬shipped, start, ¬loggedIn, ¬orderCreated, ¬billed, ¬checked}
- t = 1: {¬shipped, ¬start, loggedIn, ¬orderCreated, ¬billed, ¬checked}
- t = 2: {¬shipped, ¬start, ¬loggedIn, orderCreated, ¬billed, ¬checked}
- t = 3: {¬shipped, ¬start, ¬loggedIn, ¬orderCreated, billed, ¬checked}
- t = 4: {¬shipped, ¬start, ¬loggedIn, ¬orderCreated, ¬billed, checked}
- t = 5: {shipped, ¬start, ¬loggedIn, ¬orderCreated, ¬billed, ¬checked} ← loops here
```

Figure 3: `webshop.pltl` — The webshop modeled in BLACK using propositional LTL and its execution result.

```
G(wnext(a) = a) &
G((wnext(p1) = p1) & (wnext(p2) = p2) & (wnext(p3) = p3) & (wnext(p4) = p4) & (wnext(p5) = p5)) &

...

G((start & cust(next(c), next(a), next(v))) -> (X loggedIn & (next(t) = t))) &
G((loggedIn & id(next(p1)) & id(next(p2)) & id(next(p3))) -> (X orderCreated) & (next(t) = t)) &
G(orderCreated -> (X billed & (next(t) = price(p1) + price(p2) + price(p3) + price(p4) + price(p5))))
G((billed & v) -> (X checked & (next(t) = t * 0.80))) &
G((billed & !v) -> (X checked & (next(t) = t))) &
G((checked & (t > a)) -> (X loggedIn & (next(t) = 0.00))) &
G((checked & (t <= a)) -> (X shipped & (next(t) = t))) &

(t = 0.00) &
start & F(shipped & !v & (a < (price(p1) + price(p2) + price(p3) + price(p4) + price(p5))))
```

```
$ black solve -d Real -semi-decision webshop.foltl
```

Figure 4: `webshop.foltl` —The webshop example modeled in BLACK using first-order LTL and its execution.

Additionally, the webshop was modeled in Alloy (see Section 3.1.3), which uses nuXmv as the underlying solver. Figure 5 illustrates an excerpt of the model and property described using the specification language, along with the results of this experiment. The liveness property described in (b) was selected for verification, which states that every order, when created, is eventually either shipped or restarted.

```
enum State { Start, LoggedIn, OrderCreated, Billed, Checked, Shipped }
one sig Webshop { var state: one State, var customer: lone Customer }
sig Customer { account: one Account, var order: lone Order }
sig VIPCustomer in Customer { }
sig Account { var balance: Int }
sig Order { var products: set Product, var cost: Int }
sig Product { price: Int }

fact init {
    always transition
    Webshop.state = Start and no customer and no order and no products and Order.cost = 0
    all a: Account | a.balance >= 0 }

pred login[c: Customer, a: Account] {
    Webshop.state = Start and Webshop.state' = LoggedIn
    Webshop.customer = none and Webshop.customer' = c
    c.account = a and balance' = balance }

pred select[c: Customer, o: Order, p1: Product, p2: Product, p3: Product, p4: Product, p5: Product] {
    Webshop.state = LoggedIn and Webshop.state' = OrderCreated
    Webshop.customer = c and c.order = none and c.order' = o
    o.products = none and o.products' = p1 + p2 + p3 + p4 + p5 and o.cost = 0
    customer' = customer and balance' = balance and cost' = cost }

...

pred transition {
    (some c: Customer, a: Account                                    | login[c, a]) or
    (some c: Customer, o: Order, disj p1, p2, p3, p4, p5: Product | select[c, o, p1, p2, p3, p4, p5])
    (some c: Customer, o: Order, disj p1, p2, p3, p4, p5: Product | add[c, o, p1, p2, p3, p4, p5]) or
    (some c: Customer, o: Order                                      | discount[c, o]) or
    (some c: Customer, o: Order                                      | no_discount[c, o]) or
    (some c: Customer, o: Order                                      | restart[c, o]) or
    (some c: Customer, o: Order                                      | ship[c, o]) }

pred orderEventuallyShippedOrRestarted {
    always (Webshop.state = OrderCreated implies
            eventually (Webshop.state = Shipped or Webshop.state = LoggedIn)) }
run orderEventuallyShippedOrRestarted for 1 Customer, 1 Account, 1 Order, 5 Product, 8 Int
```

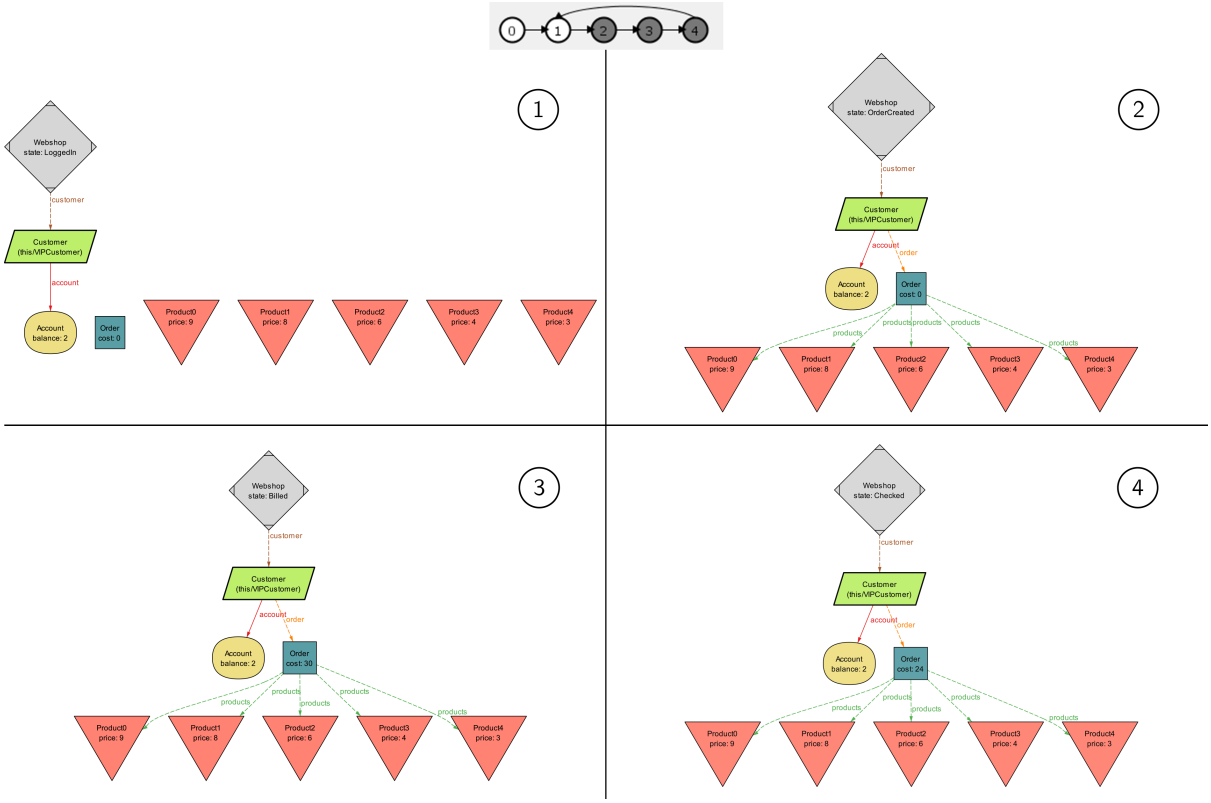


Figure 5: The webshop example modeled in Alloy. Some parts of the code and the state 0 representation have been omitted for simplicity.

4 Solution

In this section, we present an overview of the proposed solution, which aims to provide a comprehensive and standardized evaluation of formal verification tools for FOLTL. The outcomes of this study are expected to contribute with valuable insights to the community and guide future advancements in this domain. The criteria for selecting the FOLTL tools are based primarily on their capability to support formal verification of temporal logic combined with first-order theories. Thus far, we have chosen BLACK and nuXmv, but it is possible that, as our work progresses, we may select additional tools or make adjustments to our current selection. Figure 6 illustrates the general architecture.

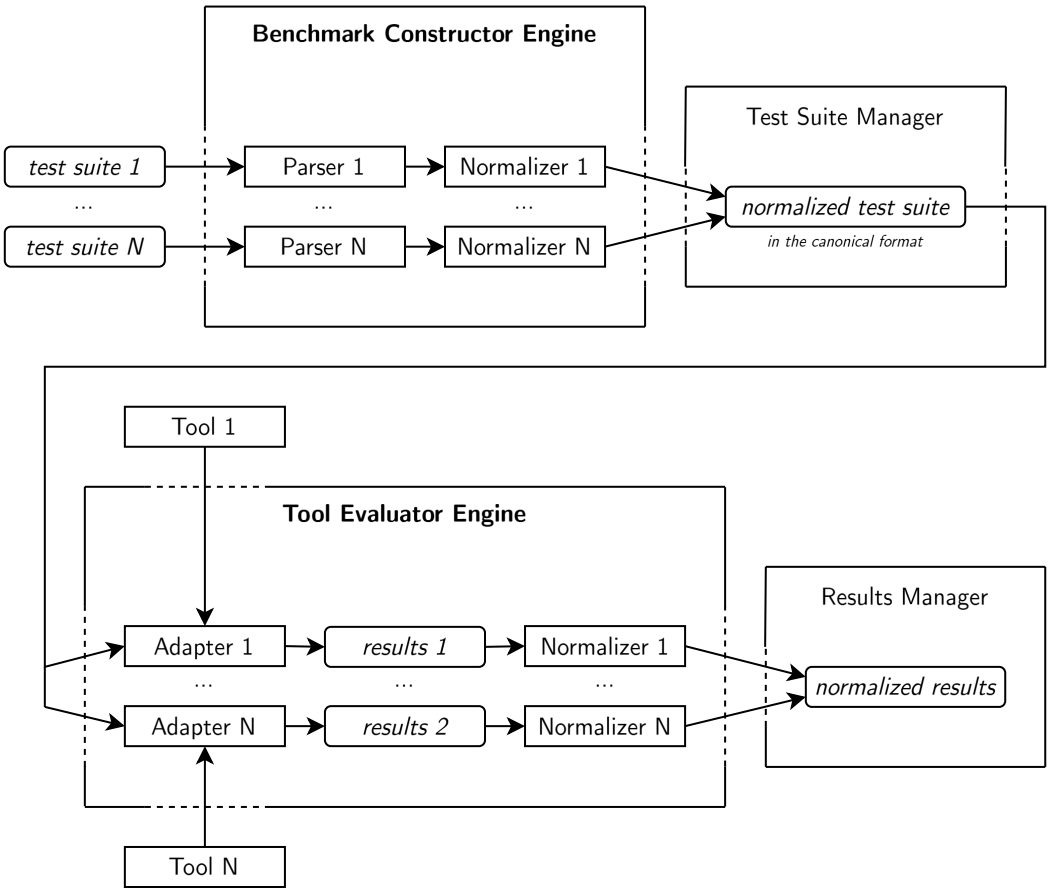


Figure 6: A sketch of the architecture of the solution.

As illustrated in Figure 6, our evaluation pipeline consists of two main engines:

- the **Benchmark Constructor Engine**, which is responsible for merging the test suites of the selected FOLTL tools into a single coherent benchmark that can be used to test all tools.
- the **Tool Evaluator Engine** that is responsible for running the given tools on the generated benchmark and collecting their results.

We discuss the inner workings of these two engines in the following two sections.

4.1 Benchmark Constructor Engine

To conduct a comprehensive evaluation of the selected tools, we plan to create an extensive benchmark consisting of standardized test cases. The methodology for constructing this benchmark involves the following steps:

(a) **Collection of formulas:** We will systematically analyze the repositories of the tools to gather the test cases utilized for their internal evaluations.

(b) **Definition of a canonical format:** Given the significant variations in notations across different tools, we aim to define a standardized canonical format for representing the formulas. This will ensure consistency and interoperability.

(c) **Benchmark construction:** The benchmark will be constructed and serialized in *JSON* format to facilitate easy parsing and integration. Once finalized, it will be made openly accessible to the research community via a GitHub repository.

Let us now take a closer look at the Benchmark Construction Engine. As illustrated in Figure 6, this engine receives as input the test suites of the selected tools. The engine first parses each test suite using the appropriate Parser Module and then converts its formulas to the established canonical format using the appropriate Normalizer Module. This format will have a tool-independent syntax. Then, all normalized test suites are given to the Test Suite Manager Module, which organizes and labels each test with the metadata required for running it successfully; for instance: expected output, required first-order theories, required temporal operators, among other features required for its verification.

4.2 Tool Evaluator Engine

Upon completion of the normalized test suite, we will conduct an empirical evaluation to benchmark the selected tools. This study will involve the following steps:

(a) **Development of a generic executor:** We will implement the Tool Evaluator Engine, which will process the formulas in the canonical format and execute them using the selected tools. The engine will leverage the libraries or APIs provided by the tools to ensure accurate execution.

(b) **Metrics measurement:** The tools will be evaluated based on their effectiveness and computational performance, also taking into account their differences. Specific metrics to be measured include execution time, accuracy, solvers chosen by the tool to verify the formulas, among others.

(c) **Data analysis:** The results will be analyzed to identify trends, strengths, and limitations of each tool, providing valuable insights into their comparative performance.

Let us now take a closer look at the Tool Evaluator Engine. As illustrated in Figure 6, this engine receives the labeled test suite and the tools to be evaluated as inputs. The engine then uses the appropriate Adapter Module to transform the test suite into the format supported by each tool and run the corresponding tool on the obtained formulas. The results of all tools are then normalized and collected for analysis by the Results Manager Module.

4.3 Portfolio

In the future, the evaluator could be extended into an intelligent interface capable of selecting the most appropriate tool for each formula based on insights gathered from prior evaluations. This would effectively serve as a front-end system for the tools. By integrating this functionality, the evaluator would not only facilitate tool selection but also provide a unified interface for these tools that themselves abstract various lower-level processes, like solvers and algorithms for satisfiability and model checking.

5 Work Schedule

The following section, in particular Figure 7, outlines the planned schedule for the development of the Dissertation.

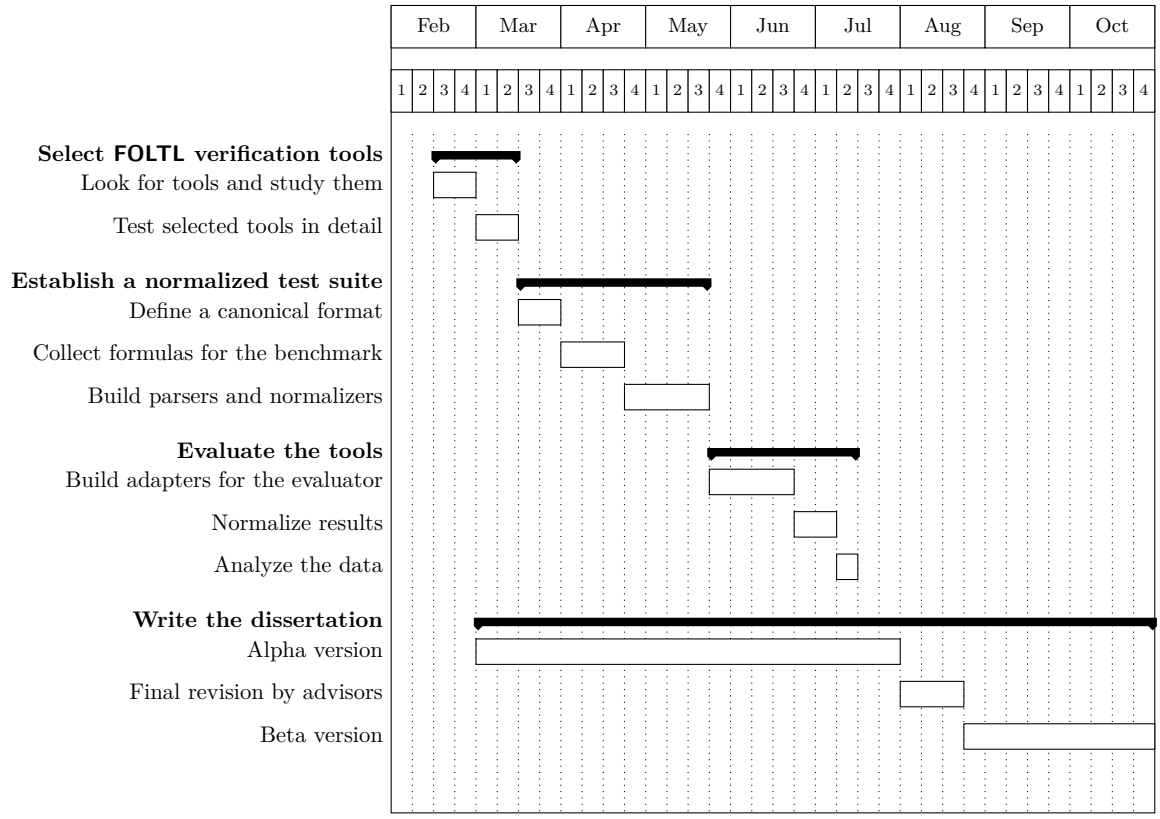


Figure 7: Planned schedule.

6 Conclusion

This report presented the logical foundations, syntax, semantics, common extensions and the satisfiability checking and model checking problems for first-order linear temporal logic (FOLTL) formulas.

We also provided an overview of state-of-the-art tools for FOLTL verification, comparing their features and conducting preliminary experiments.

The conclusion included an outline of our plans to develop a benchmark for evaluating these tools in terms of expressiveness, effectiveness, and performance, as well as to conduct an empirical study for comparison.

Bibliography

[1] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 1977, pp. 46–57.

[2] I. M. Hodkinson, F. Wolter, and M. Zakharyaschev, "Decidable fragments of first-order temporal logics," *Ann. Pure Appl. Log.*, vol. 106, no. 1-3, pp. 85–134, 2000.

[3] J. Franco and J. Martin, "A history of satisfiability," in *Handbook of Satisfiability - 2nd Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 3–74.

[4] E. M. Clarke, O. Grumberg, D. Kroening, D. A. Peled, and H. Veith, *Model checking - 2nd Edition*. MIT Press, 2018.

[5] D. Gabbay, A. Kurucz, F. Wolter, and M. Zakharyaschev, "Fragments of first-order temporal logics," in *Many-Dimensional Modal Logics: Theory and Applications*, ser. Studies in Logic and the Foundations of Mathematics. Elsevier, 2003, vol. 148, pp. 465–545.

[6] L. Geatti, A. Gianola, and N. Gigante, "A general automata model for first-order temporal logics (extended version)," *CoRR*, vol. abs/2405.20057, 2024.

[7] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems - TACAS 1999, Held as Part of the European Joint Conferences on the Theory and Practice of Software - ETAPS 1999*, ser. Lecture Notes in Computer Science, R. Cleaveland, Ed., vol. 1579. Springer, 1999, pp. 193–207.

[8] T. J. Schaefer, "The complexity of satisfiability problems," in *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, R. J. Lipton, W. A. Burkhard, W. J. Savitch, E. P. Friedman, and A. V. Aho, Eds. ACM, 1978, pp. 216–226.

[9] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability - 2nd Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 1267–1329.

[10] C. Baier and J. Katoen, *Principles of model checking*. MIT Press, 2008.

[11] G. D. Giacomo and M. Y. Vardi, "Linear temporal logic and linear dynamic logic on finite traces," in *Proceedings of the 23rd International Joint Conference on Artificial Intelligence - IJCAI 2013*, F. Rossi, Ed. IJCAI/AAAI, 2013, pp. 854–860.

[12] A. Biere, "Bounded model checking," in *Handbook of Satisfiability - 2nd Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 739–764.

[13] M. Presburger, "Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt," in *Comptes Rendus du I congrès de Mathématiciens des Pays Slaves*, 1929, pp. 92–101.

[14] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability - 2nd Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 133–182.

[15] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T)," *J. ACM*, vol. 53, no. 6, pp. 937–977, 2006.

[16] K. L. McMillan, *Symbolic model checking*. Springer, 1993, pp. 25–60.

[17] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Proceedings of the 12th International Conference on Computer Aided Verification - CAV 2000*, ser. Lecture Notes in Computer Science, E. A. Emerson and A. P. Sistla, Eds., vol. 1855. Springer, 2000, pp. 154–169.

[18] M. D'Agostino, D. M. Gabbay, R. Hähnle, and J. Posegga, *Handbook of Tableau Methods*. Springer Science & Business Media, 2013.

[19] M. Y. Vardi, "An automata-theoretic approach to linear temporal logic," in *Proceedings of the 8th Banff Higher Order Workshop - Logics for Concurrency - Structure versus Automata*, ser. Lecture Notes in Computer Science, F. Moller and G. M. Birtwistle, Eds., vol. 1043. Springer, 1995, pp. 238–266.

[20] L. Geatti, N. Gigante, and A. Montanari, "BLACK: A fast, flexible and reliable LTL satisfiability checker," in *Proceedings of the 3rd Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis hosted by the Twelfth International Symposium on Games, Automata, Logics, and Formal Verification - GandALF 2021*, ser. CEUR Workshop Proceedings, D. D. Monica, G. L. Pozzato, and E. Scala, Eds., vol. 2987. CEUR-WS.org, 2021, pp. 7–12.

[21] L. Geatti, N. Gigante, A. Montanari, and G. Venturato, "Past matters: Supporting LTL+Past in the BLACK satisfiability checker," in *Proceedings of the 28th International Symposium on Temporal Representation and Reasoning - TIME 2021*, ser. LIPIcs, C. Combi, J. Eder, and M. Reynolds, Eds., vol. 206. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 8:1–8:17.

[22] L. Geatti, A. Gianola, and N. Gigante, "Linear temporal logic modulo theories over finite traces," in *Proceedings of the 31st International Joint Conference on Artificial Intelligence - IJCAI 2022*, L. D. Raedt, Ed. ijcai.org, 2022, pp. 2641–2647.

[23] M. Reynolds, "A new rule for LTL tableaux," in *Proceedings of the 7th International Symposium on Games, Automata, Logics and Formal Verification - GandALF 2016*, ser. EPTCS, D. Cantone and G. Delzanno, Eds., vol. 226, 2016, pp. 287–301.

[24] A. Cimatti, A. Griggio, and R. Sebastiani, "The MathSAT5 SMT solver," in *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems - TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software - ETAPS 2013*, ser. Lecture Notes in Computer Science, N. Piterman and S. A. Smolka, Eds., vol. 7795. Springer, 2013, pp. 93–107.

[25] L. M. de Moura and N. S. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems - TACAS 2008, Held as Part of the European Joint Conferences on Theory and Practice of Software - ETAPS 2008*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[26] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "Cvc5: A versatile and industrial-strength SMT solver," in *Part I of the Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems - TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software - ETAPS 2022*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 415–442.

[27] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Selected Revised Papers of the 6th International Conference on Theory and Applications of Satisfiability Testing - SAT 2003*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518.

[28] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing - SAT 2009*, ser. Lecture Notes in Computer Science, O. Kullmann, Ed., vol. 5584. Springer, 2009, pp. 244–257.

[29] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuxmv symbolic model checker," in *Proceedings of the 26th International Conference on Computer Aided Verification - CAV 2014, Held as Part of the Vienna Summer of Logic - VSL 2014*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 334–342.

[30] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *Proceedings of the 14th International Conference on Computer Aided Verification - CAV 2002*, ser. Lecture Notes in Computer Science, E. Brinksma and K. G. Larsen, Eds., vol. 2404. Springer, 2002, pp. 359–364.

[31] D. Thomas, S. Chakraborty, and P. K. Pandya, "Efficient guided symbolic reachability using reachability expressions," in *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems - TACAS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software - ETAPS 2006*, ser. Lecture Notes in Computer Science, H. Hermanns and J. Palsberg, Eds., vol. 3920. Springer, 2006, pp. 120–134.

[32] K. L. McMillan, "Interpolation and sat-based model checking," in *Proceedings of the 15th International Conference on Computer Aided Verification - CAV 2003*, ser. Lecture Notes in Computer Science, W. A. H. Jr. and F. Somenzi, Eds., vol. 2725. Springer, 2003, pp. 1–13.

[33] A. R. Bradley, "Understanding IC3," in *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing - SAT 2012*, ser. Lecture Notes in Computer Science, A. Cimatti and R. Sebastiani, Eds., vol. 7317. Springer, 2012, pp. 1–14.

[34] K. Claessen and N. Sörensson, "A liveness checking algorithm that counts," in *Formal Methods in Computer-Aided Design - FMCAD 2012*, G. Cabodi and S. Singh, Eds. IEEE, 2012, pp. 52–59.

[35] S. Tonetta, "Abstract model checking without computing the abstraction," in *Proceedings of the 2nd World Congress on Formal Methods - FM 2009*, ser. Lecture Notes in Computer Science, A. Cavalcanti and D. Dams, Eds., vol. 5850. Springer, 2009, pp. 89–105.

[36] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "Verifying LTL properties of hybrid systems with k-liveness," in *Proceedings of the 26th International Conference on Computer Aided Verification - CAV 2014, Held as Part of the Vienna Summer of Logic - VSL 2014*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 424–440.

[37] M. Bozzano, R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, *nuXmv 2.1.0 User Manual*. Fondazione Bruno Kessler, Tech. Rept., 2024, ch. LTL Specifications, pp. 41–44.

[38] D. Jackson, "Alloy: a language and tool for exploring software designs," *Commun. ACM*, vol. 62, no. 9, pp. 66–76, 2019.

[39] N. Macedo, J. Brunel, D. Chemouil, A. Cunha, and D. Kuperberg, "Lightweight specification and analysis of dynamic systems with rich configurations," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, T. Zimmermann, J. Cleland-Huang, and Z. Su, Eds. ACM, 2016, pp. 373–383.

[40] J. Brunel, D. Chemouil, A. Cunha, and N. Macedo, *Formal Software Design with Alloy 6*. HASLab, 2021.

[41] A. Gianola, M. Montali, and S. Winkler, "Linear-time verification of data-aware processes modulo theories via covers and automata," in *Proceedings of the 38th AAAI Conference on Artificial Intelligence - AAAI 2024, 36th Conference on Innovative Applications of Artificial Intelligence - IAAI 2024, 14th Symposium on Educational Advances in Artificial Intelligence - EAAI 2014*, M. J. Wooldridge, J. G. Dy, and S. Natarajan, Eds. AAAI Press, 2024, pp. 10 525–10 534.