Temporal Stream Logic modulo Theories*

Bernd Finkbeiner, Philippe Heim✉, and Noemi Passing

CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{finkbeiner, philippe.heim, noemi.passing}@cispa.de

Abstract. Temporal stream logic (TSL) extends LTL with updates and predicates over arbitrary function terms. This allows for specifying data-intensive systems for which LTL is not expressive enough. In the semantics of TSL, functions and predicates are left uninterpreted. In this paper, we extend TSL with first-order theories, enabling us to specify systems using interpreted functions and predicates such as incrementation or equality. We investigate the satisfiability problem of TSL modulo the standard underlying theory of uninterpreted functions as well as with respect to Presburger arithmetic and the theory of equality: For all three theories, TSL satisfiability is neither semi-decidable nor co-semi-decidable. Nevertheless, we identify three fragments of TSL for which the satisfiability problem is (semi-)decidable in the theory of uninterpreted functions. Despite the undecidability, we present an algorithm – which is not guaranteed to terminate – for checking the satisfiability of a TSL formula in the theory of uninterpreted functions and evaluate it: It scales well and is able to validate assumptions in a real-world system design.

1 Introduction

Linear-time temporal logic (LTL) [32] is one of the standard specification languages to describe properties of reactive systems. The success of LTL is largely due to its ability to abstract from the detailed data manipulations and to focus on the change of control over time. In data-intensive applications, such as smartphone apps, LTL is, however, often not expressive enough to capture the relevant properties. When specifying a music player app, for instance, we would like to state that if the user leaves the app, the track that is currently playing will be stored and will resume playing once the user returns to the app.

To specify data-intensive systems, extensions of LTL such as Constraint LTL (CLTL) [6] and, more recently, Temporal Stream Logic (TSL) [15] have been proposed. In CLTL, the atomic propositions of LTL are replaced with atomic constraints over a concrete domain D and an interpretation for relations. Relating variables with the equality relation, such as $x = y$, denoting that the value

*This work was partially supported by the German Research Foundation (DFG) as part of the Collaborative Research Center “Foundations of Perspicuous Software Systems” (TRR 248, 389792660), and by the European Research Council (ERC) Grant OSARES (No. 683300). Philippe Heim and Noemi Passing carried out this work as PhD candidates at Saarland University, Germany.

© The Author(s) 2022
P. Bouyer and L. Schröder (Eds.): FoSSaCS 2022, LNCS 13242, pp. 325–346, 2022.
https://doi.org/10.1007/978-3-030-99253-8_17

of x is equal to the value of y, allows for specifying assignment-like statements. In this paper, however, we focus on the logic TSL to specify data-intensive systems.

TSL extends LTL with updates and predicates over arbitrary function terms. An update $[\![x \leftarrowtail f(y)]\!]$ denotes that the result of applying function f to variable y is assigned to variable x. For the music player app, for instance, the update $[\![paused \leftarrowtail track(current)]\!]$ specifies that the track that is currently playing, obtained by applying function *track* to variable *current*, is stored in variable *paused*. Updates are the main characteristic of TSL that differentiates it from other first-order extensions of LTL: They allow for specifying the evolution of variables over time. Thus, programs can be represented in TSL and therefore, for instance, the model checking problem can be encoded.

In the semantics of TSL, functions and predicates are left uninterpreted, i.e., a system satisfies a TSL formula if the formula evaluates to *true* for all possible interpretations of the function and predicate symbols. This semantics has proven especially useful in the synthesis of reactive programs [15,17], where the synthesis algorithm builds a control structure, while the implementation of the functions and predicates is either done manually or provided by some library. One exemplary success story of TSL-based specification and synthesis of a reactive system is the arcade game Syntroids [17] realized on an FPGA.

In this paper, we define and investigate the satisfiability problem of TSL modulo the standard underlying theory of uninterpreted functions and with respect to other first-order theories such as the theory of equality and Presburger arithmetic. Intuitively, a TSL formula φ is satisfiable in a theory T if there is an execution satisfying φ that matches the function applications and predicate constraints of an interpretation in T. TSL validity in T is dual: A TSL formula φ is valid in a theory T if, and only if, $\neg\varphi$ is unsatisfiable in T.

For LTL, satisfiability is decidable [37] and efficient algorithms for checking the satisfiability of an LTL formula have been implemented in tools like Aalta [25]. Satisfiability checking has numerous applications in the specification and analysis of reactive systems, such as identifying inconsistent system requirements during the design process, comparing different formalizations of the same requirements, and various types of vacuity checking. TSL satisfiability checking extends these applications to data-intensive systems.

We present an algorithm for checking the satisfiability of a TSL formula in the theory of uninterpreted functions. It is based on *Büchi stream automata* (BSAs), a new kind of ω-automata that we introduce in this paper. BSAs can handle the predicates and updates occurring in TSL formulas. Similar to the relationship between LTL formulas and nondeterministic Büchi automata, BSAs are an automaton representation of TSL formulas, i.e., there exists an equivalent BSA for every TSL formula. Given a TSL formula φ, our algorithm constructs an equivalent BSA $\mathcal{B}_\varphi$ and then tries to prove satisfiability and unsatisfiability in parallel: For proving satisfiability, it searches for a lasso that ensures consistency of the function terms in an accepting run of $\mathcal{B}_\varphi$. If such a lasso is found, φ is satisfiable. For proving unsatisfiability, the algorithm discards inconsistent runs of $\mathcal{B}_\varphi$. If no accepting run is left, φ is unsatisfiable.

The algorithm does not always terminate. In fact, we show that TSL satisfiability is neither semi-decidable nor co-semi-decidable in the theory of uninterpreted functions. Thus, no complete algorithm exists. The undecidability result extends to the theory of equality and Presburger arithmetic. There exist, however, (semi-)decidable fragments of TSL in the theory of uninterpreted functions: For satisfiable formulas with a single variable as well as satisfiable reachability formulas, our algorithm is guaranteed to terminate. For slightly more restricted reachability formulas, satisfiability is decidable.

We have implemented the algorithm and evaluated it, clearly illustrating its applicability: It terminates within one second on many randomly generated formulas and scales particularly well for satisfiable formulas. Moreover, it is able to check realistic benchmarks for consistency and to (in-)validate their assumptions. Most notably, we successfully validate the assumptions of a Syntroids module.

A preliminary version of this paper has been published on arXiv [13]. This already lead to further research on TSL modulo theories: Maderbacher and Bloem show that the synthesis problem for TSL modulo theories is undecidable in general and present a synthesis procedure for TSL modulo theories based on a counter-example guided LTL synthesis loop [27].

Further details and proofs are available in the full version of this paper [14].

2 Preliminaries

We assume time to be discrete. A *value* can be of arbitrary type and we denote the set of all values by $\mathcal{V}$. The Boolean values are denoted by $\mathbb{B} \subseteq \mathcal{V}$. Given n values, an n-ary function $f : \mathcal{V}^n \to \mathcal{V}$ computes a new value. An n-ary predicate $p : \mathcal{V}^n \to \mathbb{B}$ determines whether a property over n values is satisfied. The sets of all functions and predicates are denoted by $\mathcal{F}$ and $\mathcal{P} \subseteq \mathcal{F}$, respectively. Constants are both functions of arity zero and values. Starting from 0, we denote the i-th position of an infinite word σ by σ_i and the i-th component of a tuple t by $\pi_i(t)$.

To argue about functions and predicates, we use a term based notation. *Function terms* τ_f are constructed from variables and functions, recursively applied to a set of function terms. *Predicate terms* τ_p are constructed by applying a predicate to function terms. The sets of all function and predicate terms are denoted by $\mathcal{T}_F$ and $\mathcal{T}_P \subseteq \mathcal{T}_F$, respectively. Given sets Σ_F, Σ_P of function and predicate symbols with $\Sigma_P \subseteq \Sigma_F$, a set V of variables, and a set $\mathcal{V}$ of values, let $\langle\cdot\rangle : V \cup \Sigma_F \to \mathcal{V} \cup \mathcal{F}$ be an *assignment function* assigning a concrete function (predicate) to each function (predicate) symbol and an initial value to each variable. We require $\langle v \rangle \in \mathcal{V}$, $\langle f \rangle \in \mathcal{F}$, and $\langle p \rangle \in \mathcal{P}$ for $v \in V$, $f \in \Sigma_F$, $p \in \Sigma_P$. The evaluation $\chi_{\langle\cdot\rangle} : \mathcal{T}_F \to \mathcal{V} \cup \mathbb{B}$ of function terms is defined by $\chi_{\langle\cdot\rangle}(v) := \langle v \rangle$ for $v \in V$, and by $\chi_{\langle\cdot\rangle}(f(\tau_0, \ldots, \tau_n)) := \langle f \rangle(\chi_{\langle\cdot\rangle}(\tau_0), \ldots, \chi_{\langle\cdot\rangle}(\tau_n))$ for $f \in \Sigma_F \cup \Sigma_P$.

Functions and predicates are not tied to a specific interpretation. To restrict the possible interpretations, we utilize *first-order theories*. A first-order theory T is a tuple $(\Sigma_F, \Sigma_P, \mathcal{A})$, where Σ_F and Σ_P are sets of function and predicate symbols, respectively, and $\mathcal{A}$ is a set of closed first-order logic formulas over Σ_F, Σ_P, and a set of variables V. For an introduction to first-order logic, we refer to the

full version [14]. The elements of $\mathcal{A}$ are called the *axioms* of T and $\Sigma_F \cup \Sigma_P$ is called the *signature* of T. A *model* $\mathcal{M}$ for a theory $T = (\Sigma_F, \Sigma_P, \mathcal{A})$ is a tuple $(\mathcal{V}, \langle\cdot\rangle)$, where $\mathcal{V}$ is a set of values and $\langle\cdot\rangle$ is an assignment function as introduced above. Furthermore, $(\mathcal{V}, \langle\cdot\rangle)$ is required to entail φ_A for each axiom $\varphi_A \in \mathcal{A}$. The set of all models of a theory T is denoted by $\mathit{Models}(T)$.

In the remainder of this paper, we focus on the following three theories: The *theory of uninterpreted functions* T_U is a theory without any axioms, i.e., every symbol is uninterpreted. It allows for arbitrarily many function and predicate symbols. The *theory of equality* T_E additionally includes equality, i.e., its axioms enforce the equality symbol $=$ to indeed represent equality. The *theory of Presburger arithmetic* $T_{\mathbb{N}}$ implements the idea of numbers. Its axioms define the constants 0 and 1 as well as equality and addition.

3 Temporal Stream Logic modulo Theories

In this section, we introduce *Temporal Stream Logic modulo theories*, an extension of the recently introduced logic Temporal Stream Logic (TSL) [15] with first-order theories. First, we recap the main idea of TSL as well as its syntax and semantics. Afterwards, we extend TSL with first-order theories and define the basic notions of satisfiability and validity for TSL formulas modulo theories.

3.1 Temporal Stream Logic

Temporal Stream Logic (TSL) [15] is a temporal logic that separates temporal control and pure data. Data is represented as infinite streams of arbitrary type. TSL allows for checks and manipulations of streams on an abstract level: It focuses on the control flow and abstracts away concrete implementation details. The temporal structure of the data is expressed by temporal operators as in LTL [32]. TSL is especially designed for reactive synthesis and thus distinguishes between uncontrollable input streams and controllable output streams, so-called *cells*. In this paper, this distinction is not necessary since we consider TSL independent of its usage in synthesis. Thus, we use the notions of streams and cells as synonyms. The finite set of all cells is denoted by $\mathbb{C}$.

In TSL, we use functions $f \in \mathcal{F}$ to modify cells and predicates $p \in \mathcal{P}$ to perform checks on cells. The cells $\mathsf{c} \in \mathbb{C}$ serve as variables for function terms. The sets of all function and predicate terms over Σ_F, Σ_P, and $\mathbb{C}$ are denoted by $\mathcal{T}_F$ and $\mathcal{T}_P$. TSL formulas are built according to the following grammar:

$$\varphi, \psi := \mathit{true} \mid \neg\varphi \mid \varphi \wedge \psi \mid \bigcirc\varphi \mid \varphi \,\mathcal{U}\, \psi \mid \tau_p \mid [\![\mathsf{c} \leftarrowtail \tau_f]\!]$$

where $\mathsf{c} \in \mathbb{C}$, $\tau_p \in \mathcal{T}_P$, and $\tau_f \in \mathcal{T}_F$. An *update* $[\![\mathsf{c} \leftarrowtail \tau_f]\!]$ denotes that the value of the function term τ_f is assigned to cell c. The value of τ_f may depend on the value of the cells occurring in τ_f. The temporal operators $\bigcirc\varphi$ and $\varphi \,\mathcal{U}\, \psi$ are similar to the ones in LTL. We define $\diamondsuit\varphi = \mathit{true}\,\mathcal{U}\,\varphi$ and $\square\varphi = \neg\diamondsuit\neg\varphi$.

Since functions and predicates are represented symbolically, they are not tied to a specific implementation. To assign an interpretation to them, we use an

assignment function $\langle\cdot\rangle : \mathbb{C} \cup \Sigma_F \to \mathcal{V} \cup \mathcal{F}$, where $\mathcal{V}$ is a set of values. We require $\langle \mathtt{c} \rangle \in \mathcal{V}$, $\langle f \rangle \in \mathcal{F}$ and $\langle p \rangle \in \mathcal{P}$ for $\mathtt{c} \in \mathbb{C}$, $f \in \Sigma_F$, and $p \in \Sigma_P$ Note that $\langle\cdot\rangle$ also assigns an initial value to each cell. Terms can be compared syntactically with the equivalence relation $\equiv$. The set of all assignments of cells $\mathtt{c} \in \mathbb{C}$ to function terms $\tau_f \in \mathcal{T}_F$ is denoted by $\mathcal{C}$. A *computation* $\varsigma \in \mathcal{C}^\omega$ is an infinite sequence of assignments of cells to function terms, capturing the behavior of cells over time. The satisfaction of a TSL formula φ with respect to ς, a set of values $\mathcal{V}$, an assignment function $\langle\cdot\rangle$, and a time step t is defined by:[1]

$$
\begin{array}{ll}
\varsigma, t \models_{\mathcal{V},\langle\cdot\rangle} \neg\varphi & :\Leftrightarrow \varsigma, t \not\models_{\mathcal{V},\langle\cdot\rangle} \varphi \\
\varsigma, t \models_{\mathcal{V},\langle\cdot\rangle} \varphi \wedge \psi & :\Leftrightarrow \varsigma, t \models_{\mathcal{V},\langle\cdot\rangle} \varphi \wedge \varsigma, t \models_{\mathcal{V},\langle\cdot\rangle} \psi \\
\varsigma, t \models_{\mathcal{V},\langle\cdot\rangle} \bigcirc\varphi & :\Leftrightarrow \varsigma, t+1 \models_{\mathcal{V},\langle\cdot\rangle} \varphi \\
\varsigma, t \models_{\mathcal{V},\langle\cdot\rangle} \varphi\,\mathcal{U}\,\psi & :\Leftrightarrow \exists t'' \geq t. \forall t \leq t' < t''.\ \varsigma, t' \models_{\mathcal{V},\langle\cdot\rangle} \varphi \ \wedge\ \varsigma, t'' \models_{\mathcal{V},\langle\cdot\rangle} \psi \\
\varsigma, t \models_{\mathcal{V},\langle\cdot\rangle} [\![\mathtt{c} \leftarrowtail \tau]\!] & :\Leftrightarrow \varsigma_t(\mathtt{c}) \equiv \tau \\
\varsigma, t \models_{\mathcal{V},\langle\cdot\rangle} p(\tau_0, \ldots, \tau_m) & :\Leftrightarrow \chi_{\langle\cdot\rangle}(\eta(\varsigma, t, p(\tau_0, \ldots, \tau_m))),
\end{array}
$$

where $\eta : \mathcal{C}^\omega \times \mathbb{N} \times \mathcal{T}_F \to \mathcal{T}_F$ is a symbolic evaluation function defined by

$$
\eta(\varsigma, t, \mathtt{c}) = \begin{cases} \mathtt{c} & \text{if } t = 0 \\ \eta(\varsigma, t-1, \varsigma_{t-1}(\mathtt{c})) & \text{if } t > 0 \end{cases}
$$

$$
\eta(\varsigma, t, f(\tau_0, \ldots, \tau_m)) = f(\eta(\varsigma, t, \tau_0), \ldots, \eta(\varsigma, t, \tau_m))
$$

We call $(\varsigma, \mathcal{V}, \langle\cdot\rangle)$ an *execution*. The satisfaction of a predicate depends on the current and the past steps in the computation. For updates, the satisfaction depends solely on the current step. While updates are only checked syntactically, the satisfaction of predicates depends on the given assignment $\langle\cdot\rangle$. An execution $(\varsigma, \mathcal{V}, \langle\cdot\rangle)$ *satisfies* a TSL formula φ, denoted $\varsigma \models_{\mathcal{V},\langle\cdot\rangle} \varphi$, if $\varsigma, 0 \models_{\mathcal{V},\langle\cdot\rangle} \varphi$ holds.

Example 1. Suppose that we have a single cell $\mathtt{x}$, i.e., $\mathbb{C} = \{\mathtt{x}\}$. Consider the computation $\varsigma = (\{\lambda\mathtt{c}.f(\mathtt{x})\})^\omega$, i.e., $f(\mathtt{x})$ is assigned to cell $\mathtt{x}$ in every time step. Let $\mathcal{V} = \mathbb{N}$ be the set of values and let $\langle\cdot\rangle$ be an assignment function such that the initial value of $\mathtt{x}$ is 1, function f corresponds to incrementation, and predicate p determines whether its argument is even (*true*) or odd (*false*). Consider the TSL formula $\varphi := [\![\mathtt{x} \leftarrowtail f(\mathtt{x})]\!] \wedge \neg p(\mathtt{x}) \wedge \bigcirc p(\mathtt{x})$. By the semantics of TSL, we have $\varsigma, 0 \models_{\mathcal{V},\langle\cdot\rangle} \varphi$ if, and only if, $(\varsigma_0(\mathtt{x}) = f(\mathtt{x})) \wedge (\neg\langle p\rangle(\langle\mathtt{x}\rangle)) \wedge (\langle p\rangle(\langle f\rangle(\langle\mathtt{x}\rangle)))$ holds. The first conjunct clearly holds by construction of ς. Since 1 is odd and $1+1=2$ is even, the other two conjuncts hold as well for the chosen assignment function. Hence, $(\varsigma, \mathcal{V}, \langle\cdot\rangle)$ satisfies φ for $\varsigma = (\{\lambda\mathtt{c}.f(\mathtt{x})\})^\omega$, $\mathcal{V} = \mathbb{N}$ and $\langle\cdot\rangle$.

A computation ς is called *finitary* with respect to φ, denoted $\mathrm{fin}_\varphi(\varsigma)$, if for all cells $\mathtt{c} \in \mathbb{C}$ and for all points in time t, either $\varsigma_t(\mathtt{c}) \equiv \mathtt{c}$ holds, or there is an update $[\![\mathtt{c} \leftarrowtail \tau]\!]$ in φ such that $\varsigma_t(\mathtt{c}) \equiv \tau$, i.e., a finitary computation only contains updates occurring in φ and self-updates. For ς and φ from Example 1, for instance, ς is finitary with respect to φ.

[1] Note that we use a slightly different, but equivalent, definition than [15]: Instead of evaluating the function and predicate symbols on the fly, we construct the whole term first and then evaluate it recursively using the evaluation function $\chi_{\langle\cdot\rangle}$.

3.2 Extending TSL with Theories

In this paper, we extend TSL with first-order theories. That is, we restrict the possible interpretations of predicate and function symbols to a theory. Hence, we define the notions of satisfiability and validity of a TSL formula *modulo a theory* T. Intuitively, a TSL formula φ is satisfiable in a theory T if there exists an execution satisfying φ whose domain and assignment function represent a model in T, i.e., that entail all axioms of T. Formally:

Definition 1 (TSL Satisfiability). *Let* $T = (\Sigma_F, \Sigma_P, \mathcal{A})$ *be a theory and let* φ *be a TSL formula over* Σ_F*,* Σ_P*, and* $\mathbb{C}$*. We call* φ satisfiable in T *if, and only if, there exists an execution* $(\varsigma, \mathcal{V}, \langle\cdot\rangle)$*, such that* $\varsigma \models_{\mathcal{V},\langle\cdot\rangle} \varphi$ *and* $(\mathcal{V}, \langle\cdot\rangle) \in \mathit{Models}(T)$ *hold. If additionally* $\mathrm{fin}_\varphi(\varsigma)$ *holds, then* φ *is called* finitary satisfiable in T.

Intuitively, a formula φ is valid in a theory T, if for all executions and all matching models of the theory the formula is satisfied. Formally:

Definition 2 (TSL Validity). *Let* $T = (\Sigma_F, \Sigma_P, \mathcal{A})$ *be a theory and let* φ *be a TSL formula over* Σ_F*,* Σ_P*, and* $\mathbb{C}$*. The formula* φ *is called* valid in T *if, and only if, for all executions* $(\varsigma, \mathcal{V}, \langle\cdot\rangle)$ *with* $(\mathcal{V}, \langle\cdot\rangle) \in \mathit{Models}(T)$*, we have* $\varsigma \models_{\mathcal{V},\langle\cdot\rangle} \varphi$*. If* $\varsigma \models_{\mathcal{V},\langle\cdot\rangle} \varphi$ *holds for all executions* $(\varsigma, \mathcal{V}, \langle\cdot\rangle)$ *with both* $(\mathcal{V}, \langle\cdot\rangle) \in \mathit{Models}(T)$ *and* $\mathrm{fin}_\varphi(\varsigma)$*, then* φ *is called* finitary valid in T.

It follows directly from their definitions that (finitary) TSL satisfiability and (finitary) TSL validity are dual. Thus, we focus on TSL satisfiability in the remainder of this paper as the results can easily be extended to TSL validity.

Theorem 1 (Duality of TSL Satisfiability and Validity). *Let* φ *be a TSL formula over* Σ_F*,* Σ_P*, and* $\mathbb{C}$ *and let* $T = (\Sigma_F, \Sigma_P, \mathcal{A})$ *be a theory. Then,* φ *is (finitary) satisfiable in* T *if, and only if,* $\neg\varphi$ *is not (finitary) valid in* T*.*

4 TSL modulo T_U Satisfiability Checking

In this section, we investigate the satisfiability of TSL modulo the theory of uninterpreted functions T_U. Since T_U has no axioms, there are no restrictions on how a model for T_U evaluates the function and predicate symbols. The only condition is that the evaluated symbols are indeed functions. Therefore, we have $(\varsigma, \mathcal{V}, \langle\cdot\rangle) \in \mathit{Models}(T_U)$ for all executions. Thus, finding some execution satisfying a TSL formula φ is sufficient for showing that φ is satisfied in T_U:

Lemma 1. *Let* φ *be a TSL formula over* Σ_F*,* Σ_P*, and* $\mathbb{C}$*. If there exists an execution* $(\varsigma, \mathcal{V}, \langle\cdot\rangle)$ *with* $\varsigma \models_{\mathcal{V},\langle\cdot\rangle} \varphi$*, then* φ *is satisfiable in* T_U*. If additionally* $\mathrm{fin}_\varphi(\varsigma)$ *holds, then* φ *is finitary satisfiable in* T_U*.*

In the following, we introduce an (incomplete) algorithm for checking the satisfiability of a TSL formula φ in the theory of uninterpreted functions. By

Lemma 1, it suffices to find an execution satisfying φ to prove its satisfiability in T_U. To search for such an execution, we introduce *Büchi stream automata* (BSAs), a new kind of ω-automata that reads executions and allows for dealing with predicates and updates. BSAs are, similar to the connection between LTL and Büchi automata, an automaton representation for TSL. Then, we present the algorithm for checking satisfiability in T_U based on BSAs.

4.1 Büchi Stream Automata

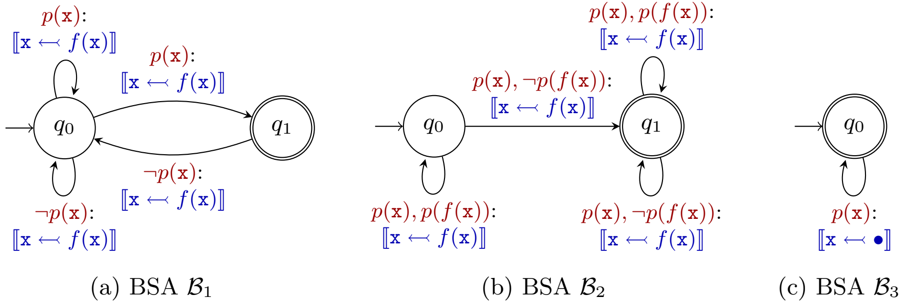
Intuitively, a *Büchi stream automaton* (BSA) is an ω-automaton with Büchi acceptance condition that reads infinite executions instead of infinite words. Furthermore, it is able to deal with predicates and updates occurring in TSL formulas. To do so, the transitions of a BSA are labeled with *guards* and *update terms*. Intuitively, the former define which predicates need to hold when taking the transition. The latter define how the corresponding cells are updated when taking the transition. Formally, a BSA is defined as follows:

Definition 3 (Büchi Stream Automaton). *Let Σ_F, Σ_P be sets of function and predicate symbols, respectively, and let $\mathbb{C}$ be a finite set of cells. A* Büchi Stream automaton *$\mathcal{B}$ over Σ_F, Σ_P, and $\mathbb{C}$ is a tuple $(Q, Q_0, F, \bullet, \mathcal{G}, \mathcal{U}, \delta)$, where Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of accepting states, $\bullet$ is a fresh term symbol such that $\bullet \notin \mathbb{C} \cup \Sigma_F \cup \Sigma_P$, $\mathcal{G} \subseteq \mathcal{T}_P$ is a finite set of predicate terms over Σ_F, Σ_P, and $\mathbb{C}$, called* guards*, $\mathcal{U} \subseteq \mathcal{T}_F \cup \{\bullet\}$ is a finite set of function terms over Σ_F, Σ_P, and $\mathbb{C}$, called* update terms*, and $\delta \subseteq Q \times (\mathcal{G} \to \mathbb{B}) \times (\mathbb{C} \to \mathcal{U}) \times Q$ is a finite transition relation.*

Note that by requiring the update terms $\mathcal{U}$ to be a *finite* set of function terms, not all executions can be read by a BSA: Non-finitary executions contain updates with function terms that do not occur in the given TSL formula. Thus, they may require infinitely many update terms. Therefore, we introduce the fresh term symbol $\bullet \notin \mathbb{C} \cup \Sigma_F \cup \Sigma_P$. If a transition in a BSA assigns $\bullet$ to a cell $\mathtt{c} \in \mathbb{C}$, then any function term can be assigned to $\mathtt{c}$. This allows for reading non-finitary executions while maintaining finite representability of BSAs.

Example 2. Consider the three BSAs depicted in Figure 1. If $\mathcal{B}_1$ is in state q_0 and $p(\mathtt{x})$ holds, then cell $\mathtt{x}$ is updated with $f(\mathtt{x})$ and $\mathcal{B}_1$ chooses nondeterministically to either stay in q_0 or to move to the accepting state q_1. In contrast, $\mathcal{B}_2$ is deterministic. Yet, it is incomplete: In both q_0 and q_1, no guard is satisfied if $\neg p(\mathtt{x})$ holds. Hence, $\mathcal{B}_2$ gets stuck, preventing satisfaction of the Büchi winning condition for any execution containing $\neg p(\mathtt{x})$. The BSA $\mathcal{B}_3$ makes use of the fresh term symbol $\bullet$: If $p(\mathtt{x})$ holds, any function term can be assigned to $\mathtt{x}$.

Given sets Σ_F, Σ_P, $\mathbb{C}$ and a BSA $\mathcal{B} = (Q, Q_0, F, \bullet, \mathcal{G}, \mathcal{U}, \delta)$ over Σ_F, Σ_P, $\mathbb{C}$, an infinite word $c \in (Q \times (\mathcal{G} \to \mathbb{B}) \times (\mathbb{C} \to \mathcal{U}) \times Q)^\omega$ is called *run of $\mathcal{B}$* if, and only if, the first state of c is an initial state, i.e., $\pi_1(c_0) \in Q_0$, and both $c_t \in \delta$ and $\pi_4(c_t) = \pi_1(c_{t+1})$ hold for all points in time $t \in \mathbb{N}_0$. Intuitively, a run c is an infinite sequence of tuples (q, g, u, q') encoding transitions in the BSA: q is the



(a) BSA $\mathcal{B}_1$ (b) BSA $\mathcal{B}_2$ (c) BSA $\mathcal{B}_3$

Fig. 1: Three exemplary Büchi stream automata. Accepting states are marked with double circles. Guards are highlighted in red, update terms in blue.

source state, q' is the target state, g determines which predicate terms hold, and u defines which updates are performed when taking the transition. A run c is called *accepting* if it contains infinitely many accepting states, i.e., for all points in time $t \in \mathbb{N}_0$, there exists a $t' > t$ such that $\pi_1(c_{t'}) \in F$ holds.

Example 3. Let $g_1(p(\mathtt{x})) = \mathit{true}$, $g_2(p(\mathtt{x})) = \mathit{false}$, and $u(\mathtt{x}) = f(\mathtt{x})$. The infinite word $c = (q_0, g_1, u, q_1)(q_1, g_2, u, q_0)(q_0, g_1, u, q_1)(q_1, g_2, u, q_0)\dots$ is a run of BSA $\mathcal{B}_1$ from Figure 1a. It is accepting as it visits q_1 infinitely often.

The characteristics of a BSA are its predicates and updates. Thus, it is not sufficient to solely consider accepting runs since the constraints produced by the predicates might be inconsistent. Therefore, we define the *execution of a BSA* that only permits consistent accepting runs. Intuitively, given a run c of a BSA $\mathcal{B}$, an *execution of* c consists of a computation $\varsigma \in \mathcal{C}^\omega$, a domain $\mathcal{V}$, and an assignment $\langle\cdot\rangle$ such that the updates in ς match the updates in c and such that the recursive evaluation of a predicate term using $\langle\cdot\rangle$ matches its truth value in ς. To capture the constraints accumulated in ς as well as their truth values, we define the *constraint trace* $\varrho : (\tau_p \times \mathbb{B})^\omega$ *of* ς *and* c: Formally, ϱ for ς and c is defined by $\varrho_t := \emptyset$ if $t = 0$, and $\varrho_t := \varrho_{t-1} \cup \{(\eta(\varsigma, t-1, \tau_p), \pi_2(c_{t-1})(\tau_p)) \mid \tau_p \in \mathcal{G}\}$ if $t > 0$. As an example, reconsider the computation ς from Example 1 and the run c of BSA $\mathcal{B}_1$ from Example 3. The constraint trace of ς and c is given by $\varrho = \emptyset\{(p(\mathtt{x}), \mathit{true})\}\{(p(\mathtt{x}), \mathit{true}), (p(f(\mathtt{x})), \mathit{false})\}\dots$. A constraint trace ϱ is called *consistent* if there is no predicate term $\tau_p \in \mathcal{T}_P$ such that both (τ_p, true) and (τ_p, false) occur in $\bigcup_{t\in\mathbb{N}_0} \varrho_t$. ϱ from the example above is consistent. Using constraint traces, we now formally define the execution of a BSA:

Definition 4 (Execution of a BSA). *Let Σ_F and Σ_P be sets of function and predicate symbols, respectively, and let $\mathbb{C}$ be a finite set of cells. Let $\mathcal{B}$ be a BSA over Σ_F, Σ_P, and $\mathbb{C}$ and let c be a run of $\mathcal{B}$. Let $\varsigma \in \mathcal{C}^\omega$ be an infinite computation and let $\langle\cdot\rangle : \mathbb{C} \cup \Sigma_F \to \mathcal{V} \cup \mathcal{F}$ be an assignment function. Let ϱ be the constraint trace of ς and c. We call $(\varsigma, \mathcal{V}, \langle\cdot\rangle)$* execution for *$c$ if (1) for all points in time $t \in \mathbb{N}_0$ and all cells $\mathsf{c} \in \mathbb{C}$, we have either $\pi_3(c_t)(\mathsf{c}) = \varsigma_t(\mathsf{c})$ or $\pi_3(c_t)(\mathsf{c}) = \bullet$, and (2) for all $(\tau_p, b) \in \bigcup_{t\in\mathbb{N}_0} \varrho_t$, we have $\chi_{\langle\cdot\rangle}(\tau_p) = b$.*

Note that the second requirement can only be fulfilled if the constraint trace is consistent. Consider the computation ς and the assignment function $\langle\cdot\rangle$ from Example 1, the run c of $\mathcal{B}_1$ from Example 3, and the constraint trace ϱ of ς and c given above. Then, $(\varsigma, \mathbb{N}, \langle\cdot\rangle)$ is an execution for c: Since in both ς and c, cell $\mathtt{x}$ is always updated with $f(\mathtt{x})$, the updates in ς and c coincide at every point in time. Furthermore, by construction of $\langle\cdot\rangle$, the constraints of ϱ match the truth values obtained by recursively evaluating $\langle\cdot\rangle$ for all predicate terms.

We define two languages of a BSA $\mathcal{B}$: The *symbolic language* $\mathcal{L}(\mathcal{B})$ is the set of all executions that have a respective accepting run, i.e., $(\varsigma, \mathcal{V}, \langle\cdot\rangle) \in \mathcal{L}(\mathcal{B})$ if, and only if, there exists an accepting run c such that $(\varsigma, \mathcal{V}, \langle\cdot\rangle)$ is an execution for c. The *language* $\mathcal{L}_T(\mathcal{B})$ *in a theory* T is the set of all executions whose domain and assignment function additionally form a model in T, i.e., $(\varsigma, \mathcal{V}, \langle\cdot\rangle) \in \mathcal{L}_T(\mathcal{B})$ if, and only if, $(\varsigma, \mathcal{V}, \langle\cdot\rangle) \in \mathcal{L}(\mathcal{B})$ and $(\mathcal{V}, \langle\cdot\rangle) \in \mathit{Models}(T)$.

We call a BSA $\mathcal{B} = (Q, Q_0, F, \bullet, \mathcal{G}, \mathcal{U}, \delta)$ *finitary* if $\bullet \notin \mathcal{U}$ holds. Hence, every run c of a finitary BSA, has a unique computation ς and thus a unique constraint trace ϱ. Therefore, for a finite prefix c_p of c, we can compute its *execution effect* $\text{effect}(c_p) := (\lambda \mathsf{c}.\, \eta(\varsigma, |c_p|, \mathsf{c}), \varrho_{|c_p|})$ from c_p itself, i.e., without considering ς and ϱ explicitly. Intuitively, c_p's execution effect consists of the function terms assigned to the cells during the execution of c_p as well as the constraints and their truth values on the transitions taken with c_p in the BSA. The BSAs $\mathcal{B}_1$ and $\mathcal{B}_2$, depicted in Figure 1, are finitary while $\mathcal{B}_3$ is not. Since $\mathcal{B}_1$ is finitary, consider the prefix $c_p = (q_0, g_1, u, q_1)(q_1, g_2, u, q_0)$ of the run c of $\mathcal{B}_1$ presented in Example 3. Its execution effect is given by $\text{effect}(c_p) = (\lambda \mathsf{c}. f(f(\mathtt{x})), \{(p(\mathtt{x}), \mathit{true}), (p(f(\mathtt{x})), \mathit{false})\})$.

An LTL formula φ can be translated into a nondeterministic Büchi automaton (NBA) $\mathcal{A}_\varphi$ with $\mathcal{L}(\varphi) = \mathcal{L}(\mathcal{A}_\varphi)$ [38]. An analogous relation exists between TSL formulas and BSAs: A TSL formula φ can be translated into an equivalent BSA $\mathcal{B}_\varphi$: First, we approximate φ by an LTL formula φ_{LTL}, similarly to the approximation described in [15]. The main idea of the approximation is to represent every function and predicate term as well as every update occurring in φ_{LTL} by an atomic proposition and to add conjuncts that ensure that exactly one update is performed for every cell in every time step. Second, we build an equivalent NBA $\mathcal{A}_{\varphi_{LTL}}$ from φ_{LTL}. Third, we construct a BSA $\mathcal{B}_\varphi$ from $\mathcal{A}_{\varphi_{LTL}}$ by, intuitively, translating the atomic propositions back into predicate terms and updates and by dividing them into guards and update terms, while maintaining the structure of the NBA $\mathcal{A}_{\varphi_{LTL}}$. The full construction of an equivalent BSA $\mathcal{B}_\varphi$ from a TSL formula φ is given in the full version [14].

Theorem 2 (TSL to BSA Translation). *Given a TSL formula φ, there exists an equivalent (finitary) Büchi stream automaton $\mathcal{B}$ such that for all theories T, $\mathcal{L}_T(\mathcal{B}) \neq \emptyset$ holds if, and only if, φ is (finitary) satisfiable in T.*

For instance, the TSL formula $\varphi_1 := \square [\![\mathtt{x} \leftarrowtail f(\mathtt{x})]\!] \wedge \square\diamondsuit(p(\mathtt{x}) \wedge \bigcirc\neg p(\mathtt{x}))$ is finitary satisfiable in a theory T if, and only if, $\mathcal{L}_T(\mathcal{B}_1) \neq \emptyset$ holds for the BSA $\mathcal{B}_1$ from Figure 1a. Analogously, $\varphi_2 := \square([\![\mathtt{x} \leftarrowtail f(\mathtt{x})]\!] \wedge p(\mathtt{x})) \wedge \diamondsuit\neg p(f(\mathtt{x}))$, and $\varphi_3 := \square p(\mathtt{x})$ correspond to the BSAs $\mathcal{B}_2$ and $\mathcal{B}_3$ from Figure 1b and Figure 1c.

Algorithm 1: Algorithm for Checking TSL modulo T_U Satisfiability

```
    Input: φ: TSL Formula
    Output: SAT, UNSAT
 1  B := Finitary BSA for φ as defined in Theorem 2;
 2  R := Set of runs of B;
 3  Function SatSearch
 4      for pref.rec^ω ∈ {c | c ∈ R ∧ accepting(c)} do
 5          (v_p, _):=effect(pref);
 6          (v_r, P):=effect(pref.rec);
 7          if SMT(( ⋀_{(t_p,v)∈P} { t_p if v = true ; ¬t_p if v = false } ) ∧ ⋀_{c∈ℂ} v_p(c) = v_r(c)) = SAT then
 8              return SAT
 9  Function UnsatSearch
10      for n ∈ ℕ_0 do
11          for c ∈ {c | c ∈ finiteSubwords(R) ∧ |c| = n} do
12              (_, P):=effect(c);
13              if ∃t_p. (t_p, true), (t_p, false) ∈ P then
14                  R := R \ {c' | ∃m ∈ ℕ_0. ∀0 ≤ i < n. c'_{i+m} = c_i}
15              if {c | c ∈ R ∧ accepting(c)} = ∅ then
16                  return UNSAT
17  return parallel(SatSearch, UnsatSearch)
```

Line 7: $\text{if SMT}\left(\left(\bigwedge_{(t_p,v)\in P} \begin{cases} t_p & \text{if } v = \mathit{true} \\ \neg t_p & \text{if } v = \mathit{false} \end{cases}\right) \wedge \bigwedge_{\mathsf{c}\in\mathbb{C}} v_p(\mathsf{c}) = v_r(\mathsf{c})\right) = \text{SAT then}$

4.2 An Algorithm for TSL modulo T_U Satisfiability Checking

Utilizing BSAs, we present an algorithm for checking the satisfiability of a TSL formula in the theory of uninterpreted functions T_U in the following. First, recall that finitary computations only perform self-updates or updates that occur in the given TSL formula. Since there are only finitely many cells, the behavior of finitary computations is thus restricted to a finite set of possibilities in each step. Hence, reasoning with finitary computations is easier than reasoning with non-finitary ones. In the algorithm, we make use of the fact that satisfiability can be reduced to finitary satisfiability in the theory of uninterpreted functions, enabling us to focus on finitary computations. The main idea of the reduction is to introduce a new cell for each cell of a given TSL formula. The new cells then capture the values that are constructed by the non-finitary parts of a computation. The proof is given in the full version [14].

Lemma 2. *Let φ be a TSL formula. Then, there is a TSL formula φ_{fin} such that φ is satisfiable in T_U if, and only if, $\varphi \wedge \varphi_{fin}$ is finitary satisfiable in T_U.*

Algorithm 1 shows the algorithm for checking TSL modulo T_U satisfiability. It directly works on Büchi stream automata. First, an equivalent BSA $\mathcal{B}$ is generated for the input formula φ. Then, in parallel, *SatSearch* tries to prove that φ is satisfiable in T_U while *UnsatSearch* tries to prove unsatisfiability of φ.

SatSearch enumerates all lasso-shaped accepting runs $\mathit{pref}.\mathit{rec}^\omega$ of $\mathcal{B}$, i.e., accepting runs consisting of a finite prefix *pref* and a finite recurring part *rec* that is repeated infinitely often. Both *pref* and *rec* need to end in the same state of $\mathcal{B}$. Then, the execution effects of *pref* and *pref.rec* are computed. *SatSearch* checks if it is possible to satisfy all predicate constraints induced by *pref.rec* under the condition that, for each cell, *pref* and *pref.rec* construct equal function terms. For this, it utilizes an SMT solver to check the satisfiability of a quantifier-free first-order logic formula, encoding the consistency requirement, in the theory of equality. If the check succeeds, adding *rec* to *pref* des not create an inconsistency and hence repeating *rec* infinitely often is consistent. Therefore, there exists an execution for $\mathit{pref}.\mathit{rec}^\omega$ and thus φ is finitary satisfiable in T_U by Lemma 1.

UnsatSearch computes the execution effect of finite subwords of runs of $\mathcal{B}$ and checks whether they are consistent. If a subword is inconsistent, then every run that contains this subword is inconsistent. Hence, there do not exist executions for these runs and therefore they are removed from the set of candidate runs. If there is no accepting candidate run left, then $\mathcal{B}$ has an empty symbolic language and thus, by Theorem 2, φ is unsatisfiable in T_U.

Example 4. Consider the finitary BSAs $\mathcal{B}_1$ and $\mathcal{B}_2$ from Figures 1a and 1b as well as their respective TSL formulas $\varphi_1 := \square [\![\mathtt{x} \leftarrowtail f(\mathtt{x})]\!] \wedge \square\diamondsuit(p(\mathtt{x}) \wedge \bigcirc \neg p(\mathtt{x}))$ and $\varphi_2 := (\square([\![\mathtt{x} \leftarrowtail f(\mathtt{x})]\!] \wedge p(\mathtt{x})) \wedge \diamondsuit \neg p(f(\mathtt{x}))$. If we execute Algorithm 1 on φ_1, *SatSearch* considers the accepting lasso $q_0 \to q_1 \to q_0$ in $\mathcal{B}_1$ at some point. Then, $\mathit{pref} = \varepsilon$ and $\mathit{rec} = (q_0, g_1, u, q_1)(q_1, g_2, u, q_0)$. Note that *pref.rec* is the finite prefix c_p of a run of $\mathcal{B}_1$ from Example 3. Thus, effect(*pref.rec*) is given by $(\lambda \mathtt{c}.f(f(\mathtt{x})), \{(p(\mathtt{x}), \mathit{true}), (p(f(\mathtt{x})), \mathit{false})\})$. Since $\text{effect}(\mathit{pref}) = (\lambda \mathtt{c}.c, \emptyset)$ holds, *SatSearch* generates the query $p(\mathtt{x}) \wedge \neg p(f(\mathtt{x})) \wedge \mathtt{x} = f(f(\mathtt{x}))$ which is satisfiable in T_E. Hence, we can repeat the lasso $q_0 \to q_1 \to q_0$ infinitely often without getting any inconsistent constraints and thus φ_1 is satisfiable.

If we execute Algorithm 1 on φ_2, *UnsatSearch* checks at some point wether in $\mathcal{B}_2$ the transition sequence $q_0 \to q_1$ followed by the upper self-loop is consistent. This is not the case as it requires $p(f(\mathtt{x}))$ to be true (first transition) and false (second transition): We have $\varrho_1 = \{(p(\mathtt{x}), \mathit{true}), (p(f(\mathtt{x})), \mathit{false})\}$ and $\varrho_2 = \varrho_1 \cup \{(p(f(\mathtt{x})), \mathit{true}), (p(f(f(\mathtt{x}))), \mathit{true})\}$ by definition of the constraint trace. *UnsatSearch* also checks the transition sequence $q_0 \to q_1$ followed by the lower self-loop which is also inconsistent. Hence, there is no consistent transition after $q_0 \to q_1$ and thus there is no valid accepting run. Hence, φ_2 is unsatisfiable.

Note that the presentation of Algorithm 1 omits implementation details such as the enumeration of accepting loops and the implementation of the infinite set $\mathcal{R}$. A more detailed description addressing these issues is given in [14].

Algorithm 1 is correct. Intuitively, it terminates with SAT if the constraint trace ϱ of the unique computation ς of $\mathit{pref}.\mathit{rec}^\omega$ is consistent. Hence, ϱ defines an assignment $\langle\cdot\rangle$ such that $(\varsigma, \mathcal{V}, \langle\cdot\rangle)$ is an execution of $\mathit{pref}.\mathit{rec}^\omega$, implying satisfiability of φ in T_U. If the algorithm terminates with UNSAT, then all accepting runs of the BSA are inconsistent and thus no finitary execution satisfying φ exists. For the proof, we refer the reader to the full version [14].

Theorem 3 (Correctness of Algorithm 1). *Let φ be a TSL formula. If Algorithm 1 terminates on φ with* SAT*, then there exists an execution $(\varsigma, \mathcal{V}, \langle\cdot\rangle)$ such that both $\varsigma \models_{\mathcal{V},\langle\cdot\rangle} \varphi$ and $\mathrm{fin}_\varphi(\varsigma)$ hold. If Algorithm 1 terminates with* UNSAT*, then for all executions $(\varsigma, \mathcal{V}, \langle\cdot\rangle)$ with $\mathrm{fin}_\varphi(ctr)$, $\varsigma \not\models_{\mathcal{V},\langle\cdot\rangle} \varphi$ holds.*

5 Undecidability of TSL modulo T_U Satisfiability

The algorithm for TSL satisfiability checking in T_U presented in the previous section does not necessarily terminate. In this section, we show that no complete algorithm exists: The satisfiability of a TSL formula in the theory of uninterpreted functions T_U (TSL-T_U-SAT) is neither semi-decidable nor co-semi-decidable:

Theorem 4 (Undecidability of TSL-T_U-SAT). *The satisfiability (validity) problem of TSL in T_U is neither semi-decidable nor co-semi-decidable.*

The main intuition behind the undecidability result is that we can encode numbers with TSL in the theory of uninterpreted functions. That is, we are able to encode incrementation, resetting some variable to zero, and equality. We only give the encoding here, for the proof of its correctness we refer to [14].

Lemma 3. *Let f be a unary function, let $\widehat{=}$ be a binary predicate, and let z be a constant. Let $f^x(z)$ correspond to applying f x-times to z. There exists a TSL formula φ_{num} such that every execution entailing φ_{num} requires from its models that for all $a, b \in \mathbb{N}_0$, $a = b$ holds if, and only if, $f^a(z) \widehat{=} f^b(z)$ holds.*

Proof (Sketch). We construct $\varphi_{num} = \varphi_1 \wedge \varphi_2$ as follows: The first conjunct is defined by $\varphi_1 := [\![e \leftarrowtail z]\!] \wedge \bigcirc\square([\![e \leftarrowtail f(e)]\!] \wedge e \widehat{=} e)$. Let

$$\varphi_{eq} := (x \widehat{=} b) \rightarrow ([\![x \leftarrowtail z]\!] \wedge [\![b \leftarrowtail f(b)]\!] \wedge \neg\,(b \widehat{=} f(b)) \wedge \neg\,(f(b) \widehat{=} b))$$
$$\varphi_{neq} := \neg\,(x \widehat{=} b) \rightarrow ([\![x \leftarrowtail f(x)]\!] \wedge [\![b \leftarrowtail b]\!] \wedge \neg\,(x \widehat{=} f(b)) \wedge \neg\,(f(b) \widehat{=} x))\,.$$

Then, φ_2 is defined by $\varphi_2 := [\![x \leftarrowtail z]\!] \wedge [\![b \leftarrowtail z]\!] \wedge \bigcirc\square(\varphi_{eq} \wedge \varphi_{neq})$.

Intuitively, f corresponds to incrementation, z to resetting a variable to zero, and $\widehat{=}$ to equality: φ_1 ensures that $f^n(z) \widehat{=} f^n(z)$ holds for all $n \in \mathbb{N}_0$. In contrast, φ_2 ensures that if $a \neq b$ holds, then $\neg(f^a(z) \widehat{=} f^b(z))$: Starting with $x = b = z$, φ_1 ensures that $x \widehat{=} b$ holds initially. Then, φ_{eq} resets x to z and "increments" b, while ensuring that $\neg(f^k(z) \widehat{=} f^{k+1}(z))$ holds, where $b = f^k(z)$. Then, $\neg(x \widehat{=} b)$ holds and thus φ_{neq} "increments" x until it reaches $b = f^{k+1}(z)$, while ensuring that $\neg(f^{k+1}(z) \widehat{=} f^\ell(z))$ holds for all $\ell < k + 1$.

Using this encoding in TSL modulo T_U, we can construct a TSL formula $\varphi_{\mathcal{G}}$ for every GOTO-program $\mathcal{G}$ such that $\varphi_{\mathcal{G}} \wedge \varphi_{num}$ is satisfiable in T_U if, and only if, $\mathcal{G}$ terminates on every input. Intuitively, $\varphi_{\mathcal{G}}$ "simulates" $\mathcal{G}$ on different inputs by starting with input zero and incrementing the input if the halting location was reached. The temporal operators of TSL allow for requiring that $\mathcal{G}$ terminates infinitely often. The construction of $\varphi_{\mathcal{G}}$ is given in the full version [14]. Since the universal halting problem for GOTO programs is neither semi-decidable nor

co-semi-decidable, the same undecidability result follows for the satisfiability of a TSL formula modulo T_U, proving Theorem 4.

Since the theory of Presburger arithmetic $T_{\mathbb{N}}$ allows for incrementation, resetting a variable to zero, and equality, we can reuse the TSL formula $\varphi_{\mathcal{G}}$ from above to reduce the universal halting problem for GOTO programs to TSL satisfiability modulo $T_{\mathbb{N}}$ (TSL-$T_{\mathbb{N}}$-SAT), proving undecidability of TSL-$T_{\mathbb{N}}$-SAT. Note that this result holds for other theories that can express incrementation, reset, and equality, for instance Peano Arithmetic, as well.

Theorem 5 (Undecidability of TSL-$T_{\mathbb{N}}$-SAT). *The satisfiability (validity) problem of TSL in $T_{\mathbb{N}}$ is neither semi-decidable nor co-semi-decidable.*

Furthermore, equality allows for encoding incrementation and resetting a variable to zero. Hence, similarly to T_U, there exists a TSL formula φ_{enc} that, if entailed, enforces a binary function and a constant to behave as incrementation and a reset, respectively. The construction of φ_{enc} is given in the full version [14]. Thus, the TSL formula $\varphi_{\mathcal{G}}$ constructed as above for a GOTO program $\mathcal{G}$ ensures that $\varphi_{\mathcal{G}} \wedge \varphi_{enc}$ is satisfiable in the theory of equality T_E if, and only if, $\mathcal{G}$ terminates on every input. Hence, undecidability of TSL-T_E-SAT follows:

Theorem 6 (Undecidability of TSL-T_E-SAT). *The satisfiability (validity) problem of TSL in T_E is neither semi-decidable nor co-semi-decidable.*

6 (Semi-)Decidable Fragments

In Section 5, we showed that TSL satisfiability is undecidable in T_U. In this section, however, we identify fragments of TSL on which Algorithm 1 terminates for certain inputs. In fact, we present one fragment for which TSL-T_U-SAT is decidable and two fragments for which TSL-T_U-SAT is semi-decidable.

First, we consider the TSL reachability fragment, i.e., the fragment of TSL that only permits the next operator and the eventually operator as temporal operators. In our applications, this fragment corresponds to finding counterexamples to safety properties. For satisfiable reachability formulas, Algorithm 1 terminates. The main idea behind the termination is that the BSA of a reachability formula has an accepting lasso-shaped run and since φ is satisfiable, this run is consistent. For the proof, we refer to the full version [14].

Lemma 4. *Let φ be a TSL formula in the reachability fragment. If φ is finitary satisfiable in T_U, then Algorithm 1 terminates on φ.*

Restricting the reachability fragment further, we consider TSL formulas with updates, predicates, logical operators, next operators, and at most one top-level eventually operator. Such formulas are either completely time-bounded or they are of the form $\varphi = \Diamond \varphi'$, where φ' is time-bounded. In the dual validity problem, such formulas correspond to invariants on a fixed time window, a useful property for many applications. Algorithm 1 is guaranteed to terminate for satisfiable and unsatisfiable formulas of the above form if a *suitable* BSA is constructed. Such a

suitable BSA has a single accepting state q indicating that the time-bounded part has been satisfied. Intuitively, a suitable BSA ensures that all runs reaching q are accepting and that only finitely many transition sequences lead to q. Then, if φ is unsatisfiable, Algorithm 1 is able to exclude all transition sequences leading to q and thus to terminate. A BSA with infinitely many transition sequences leading to q, in contrast, may cause the algorithm to diverge as it may consider infinitely many consistent subsequences before finding the inconsistent one yielding the exclusion of the sequences leading to q. A suitable BSA exists for every TSL formula in the considered fragment. For the proof, including a more detailed description of suitable BSAs, we refer to the full version [14].

Lemma 5. *Let φ be a TSL formula with only logical operators, predicates, updates, next operators, and at most one top-level eventually operator. Algorithm 1 terminates on φ if it picks a suitable respective BSA.*

Note that Algorithm 1 is only a formal decider for this fragment if we ensure that a suitable BSA is always generated. In practice, we experienced that this is usually the case even without posing restrictions on the BSA construction. Lastly, we consider a fragment of TSL that does not restrict the temporal structure of the formula but the number of used cells. For TSL formulas with a single cell, Algorithm 1 always terminates on satisfiable inputs:

Lemma 6. *Let φ be a TSL formula such that $|\mathbb{C}| = 1$. If φ is finitary satisfiable in the theory of uninterpreted functions, then Algorithm 1 terminates on φ.*

Intuitively, restricting the TSL formula to use only a single cell prevents us from simulating arbitrary computations and thus from reducing from the universal halting problem of Goto programs as in the general undecidability proof. The formal proof, given in the full version [14], however, is unrelated to the above intuition. Combining the three observations, we obtain the following (semi-)decidability results for the satisfiability of fragments of TSL modulo T_U:

Theorem 7. *The satisfiability problem of TSL formulas in T_U is (1) semi-decidable for the reachability fragment of TSL, (2) decidable for formulas consisting of only logical operators, predicates, updates, next operators, and at most one top-level eventually operator, and (3) semi-decidable for formulas with one cell.*

7 Evaluation

We implemented the algorithm for checking TSL modulo T_U satisfiability[2]. We used *TSL tools*[3] to handle TSL, *spot* [11] to transform the approximated LTL formulas into NBAs, *SyFCo* [20] for LTL transformations, and *z3* [31] to solve SMT queries. The implementation follows the extended algorithm described in [14]. Since in some cases the default optimizations of *spot* produce a large overhead in

[2] https://github.com/reactive-systems/tsl-satisfiability-modulo-theories

[3] https://github.com/reactive-systems/tsltools

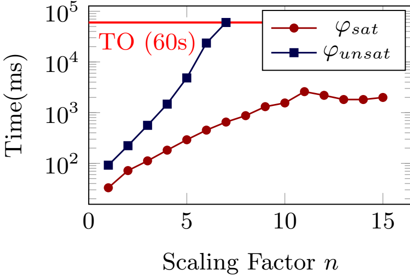


Fig. 2: Execution times in milliseconds of the scalability benchmark series.

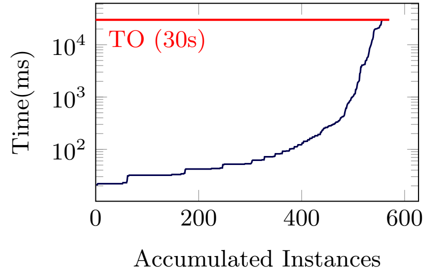


Fig. 3: Execution times in milliseconds of the random benchmark series.

computation time, we first execute it with these and if this does not terminate within 20s, we execute it without optimizations. We evaluated the implementation on three benchmark classes and a machine with an AMD Ryzen 7 processor, using a virtual machine with two logical cores and 6 GB of RAM.

Scalability Benchmark Series. We test the scalability of the algorithm with parameterized decidable benchmarks. The timeout is one minute. Note that *spot* can always perform its optimizations. The satisfiable benchmarks are defined by $\varphi_{sat}(n) := (\Box⟦\mathtt{x} \leftarrowtail f(\mathtt{x})⟧) \wedge (\Diamond \neg p(\mathtt{x})) \wedge \left(\bigwedge_{i=0}^{n} p(f^i(\mathtt{x}))\right)$. The parameter n corresponds to the number of updates that have to be performed to find a satisfiable lasso. By Lemma 6, the algorithm always terminates. The TSL formula $\varphi_{unsat}(n) := (\Box(q(\mathtt{x}) \leftrightarrow \neg q(f^n(\mathtt{x})))) \wedge (\Box⟦\mathtt{x} \leftarrowtail f(\mathtt{x})⟧) \wedge \Diamond(q(\mathtt{x}) \wedge \bigcirc^n q(\mathtt{x}))$ defines the unsatisfiable benchmarks. The parameter n corresponds to the "distance" in time and number of updates of the conflict causing unsatisfiability. The algorithm always terminates. The results are shown in Figure 2. The algorithm scales particularly well for the satisfiable formulas. However, the experiments indicate an exponential complexity of the algorithm for the unsatisfiable formulas.

Random Benchmark Series. We implemented a random TSL formula generator that uses *spot*'s `ltlrand` to generate random LTL formulas and then substitutes the atomic propositions with random updates and predicates. The generated TSL formulas have one to three cells, one to three different updates and one to three different predicates. For the LTL formulas generated by `ltlrand` we use tree sizes from 5 to 95 in steps of five. For each of the tree sizes, we generate 30 formulas; 570 in total. The execution times are shown in Figure 3. On many formulas, the algorithm terminates within one second. The implementation returns SAT for 513 of the 570 formulas. It times out after 30s on 29 formulas. However, the timeouts already occur in the automaton construction, both with and without *spot*'s optimizations. Only 28 formulas are unsatisfiable. For 25 of these unsatisfiable formulas, the intermediate LTL approximation formula is already unsatisfiable, i.e., only for three formulas there is some conflict due to updates and predicate evaluation.

Table 1: Execution times in seconds of the application benchmark series.

Benchmark	Result	Time	Benchmark	Result	Time
Chain	SAT	7.06	Inductive Ass.	UNSAT	0.25
Filter	UNSAT	0.33	One Of Two	UNSAT	1.20
Gamemodechooser Ass.	UNSAT	35.55	One Of Three	UNSAT	4.25
Holding Arbiter	SAT	11.75	Injector	UNSAT	1.52
Small Holding Arbiter	SAT	36.69	Invariant Holding	UNSAT	2.33
P. T. Arbiter	UNSAT	56.03	Scheduler	UNSAT	3.87
Approx. P. T. Arbiter	UNSAT	940.03			

Applications Benchmark Series. These benchmarks correspond to checking consistency of a specification and validating assumptions of a system. Hence, they illustrate how satisfiability results can aid the system design. The results are presented in Table 1. We introduce two of the benchmarks in more detail here. The other, slightly larger, ones, including different kinds of arbiters, a scheduler, and modules of the Syntroids [17] arcade game, are described in [14].

The *Chain* benchmark considers a compound system of two chained modules m_1 and m_2 that receive an input value, store it, and forward it to the next system: $\varphi_i := \Box\Diamond(⟦\mathtt{mem}_i \leftarrowtail \mathtt{in}_i⟧ \wedge \bigcirc⟦\mathtt{in}_{i+1} \leftarrowtail \mathtt{mem}_i⟧)$ for $i \in \{1, 2\}$. To simulate the input of the first module, we use an update with an uninterpreted function: $\varphi_{inp} := \Box⟦\mathtt{in}_1 \leftarrowtail f(\mathtt{in}_1)⟧$. We require that if some property p holds on m_1's input, p also needs to hold hold on m_2's output: $\varphi_{spec} := \Box(p(\mathtt{in}_1) \rightarrow \Diamond p(\mathtt{in}_3))$. Our algorithm determines within 8s that $(\varphi_{inp} \wedge \varphi_1 \wedge \varphi_2) \wedge \neg\varphi_{spec}$ is satisfiable, detecting an inconsistency: If m_1 stores some value on which p holds, it may overwrite it before m_2 copies it, preventing the value to reach m_2's output.

The *Filter* benchmark studies a system that "passes through" an input value to a cell if it fulfills a certain property p and holds the previous value otherwise: $\varphi_{filter} := ⟦\mathtt{out} \leftarrowtail d()⟧ \wedge \bigcirc\Box((p(\mathtt{in}) \rightarrow ⟦\mathtt{out} \leftarrowtail \mathtt{in}⟧) \wedge (\neg p(\mathtt{in}) \rightarrow ⟦\mathtt{out} \leftarrowtail \mathtt{out}⟧))$, where d is a constant representing an initial default value. The default value fulfills p, i.e., $\varphi_{fact} := \Box p(d())$. As for the chain, $\varphi_{inp} := \Box⟦\mathtt{in} \leftarrowtail f(\mathtt{in})⟧$ simulates the input. The filter is valid if p holds on all outputs after the initialization: $\varphi_{spec} := \bigcirc\Box p(\mathtt{out})$. Within 400ms, the algorithm confirms that $(\varphi_{inp} \wedge \varphi_{fact} \wedge \varphi_{filter}) \wedge \neg\varphi_{spec}$ is unsatisfiable, validating the filter.

8 Related Work

Linear-time temporal logic (LTL) [32] is one of the most popular specification languages for reactive systems. It is based on an underlying assertion logic, such as propositional logic, which is extended with temporal modalities. Satisfiability of propositional LTL has long known to be decidable [37] and there are efficient tools for LTL satisfiability checking [36,25].

While propositional LTL is very common, especially in hardware verification, LTL with richer assertion logics, such as first-order logic and various theories, have long been used in verification (cf. [28]). Temporal Stream Logic (TSL) [15]

was introduced as a new temporal logic for reactive synthesis. In the original TSL semantics, all functions and predicates are uninterpreted. TSL synthesis is undecidable in general, even without inputs or equality, but can be under-approximated by the decidable LTL synthesis problem [15]. TSL has been used to specify and synthesize an arcade game realized on an FPGA [17].

Constraint LTL (CLTL) [6] extends LTL with the possibility of expressing constraints between variables at bounded distance. A constraint system $\mathcal{D}$ consists of a concrete domain and an interpretation of relations on the domain. In Constraint LTL over $\mathcal{D}$ (CLTL($\mathcal{D}$)), one can relate variables with relations defined in $\mathcal{D}$. Similar to updates in TSL, CLTL can specify assignment-like statements by utilizing the equality relation. Like for all constraints allowing for a counting mechanism, LTL with Presburger constraints, i.e., CLTL($\mathbb{Z}, =, +$), is undecidable [6]. However, there exist decidable fragments such as LTL with finite constraint systems [4] and LTL with integer periodicity constraints [5]. Permitting constraints between variables at an unbounded distance leads to undecidability even for constraint systems that only allow equality checks on natural numbers. Restricting such systems to a finite number of constraints yields decidability again [9]. In TSL modulo theories, a theory is given from which a model can be chosen. In CLTL, in contrast, the concrete model is fixed. Therefore, TSL modulo theories cannot be encoded into CLTL in general.

LTL has been extended with the *freeze operator* [8,7], allowing for storing an input in a register. Then, the stored value can be compared with a current value for equality. Freeze LTL with two registers is undecidable [26,10] . For flat formulas, i.e., formulas where the possible occurrences of the freeze operator are restricted, decidability is regained [10]. Similar to the freeze operator, updates in TSL allow for storing values in cells and in TSL modulo the theory of equality the equality check can be performed. In TSL, we can perform computations on the stored values which is not possible in freeze LTL. Hence, freeze LTL can be seen as a special case of TSL. Constraint LTL has been augmented with the freeze operator as well [10]. For an infinite domain equipped with the equality relation, it is undecidable. For flat formulas, decidability is regained [10].

The temporal logic of actions (TLA) [24] is designed to model computer systems. States are assignments of values to variables and actions relate states. Actions can, similar to updates in TSL, describe assignments of variables. A TLA formula may contain state functions and predicates. Actions and state functions are combined with the temporal operators $\square$ and $\diamondsuit$. In contrast to TSL, $\bigcirc$ and $\mathcal{U}$ are not permitted. The validity problem for the propositional fragment of TLA, i.e., with uninterpreted functions and predicates, is PSPACE complete [35].

Similar to temporal logics, dynamic logic [33,19] is an extension of modal logic to reason about computer programs. Dynamic logic allows for stating that after action a, it is necessarily the case that p holds, or it is possible that p holds. Compound actions can be build up from smaller actions. In propositional dynamic logic (PDL) [16], data is omitted, i.e., its terms are actions and propositions. PDL satisfiability is decidable in EXPTIME [34]. First-order dynamic logic (FODL) [18] allows for including data: First-order quantification over a

first-order structure, the so-called domain of computation, is allowed. Dynamic logic does not contain temporal operators such as $\square$ or $\diamond$. Since we consider reactive systems, i.e., systems that continually interact with their environment, temporal logics are better suited than dynamic logics for our setting.

Symbolic automata (see e.g. [2,3]) and *register automata* [21] are extensions of finite automata that are capable of handling large or infinite alphabets. Register automata have additionally been considered over infinite words in some works (see e.g. [8,22,12]). Similar to BSAs, transitions of symbolic automata are labeled with predicates over a domain of alphabet symbols. Register automata are equipped with a finite amount of registers that, similar to cells in BSAs, can store input values. *Symbolic register automata* (SRAs) [1] combine the features of both automata models. BSAs have the additional ability to modify the stored values and thus to perform actual computations on them. Moreover, they read infinite instead of finite words. Thus, SRAs can be seen as a special case of BSAs.

More recently, the verification of uninterpreted programs has been investigated [29]. Uninterpreted programs are similar to WHILE-programs with equality and uninterpreted functions and predicates. They are annotated with assumptions. The verification of uninterpreted programs is undecidable in general; for the subclass of coherent uninterpreted programs, however, it is decidable [29]. The verification problem has been extended with theories, i.e., with axioms over the functions and predicates [30]. Adding axioms to coherent uninterpreted programs preserves decidability for some axioms, e.g., idempotence, while it yields undecidability for others, e.g., associativity. The synthesis problem for uninterpreted programs is undecidable in general, but decidable for coherent ones [23].

9 Conclusion

We have extended Temporal Stream Logic (TSL) with first-order theories and formalized the satisfiability and validity of a TSL formula in a theory. While we show that TSL satisfiability is neither semi-decidable nor co-semi-decidable in the theory of uninterpreted functions T_U, the theory of equality T_E, and Presburger arithmetic $T_{\mathbb{N}}$, we identify three fragments for which satisfiability in T_U is (semi-)decidable: For reachability formulas as well as for formulas with a single cell, TSL satisfiability in T_U is semi-decidable. For slightly more restricted reachability formulas, it is decidable. Moreover, we have presented an algorithm for checking the satisfiability of a TSL formula in the theory of uninterpreted functions that is based on Büchi stream automata, an automaton representation of TSL formulas introduced in this paper. Satisfiability checking has various applications in the specification and analysis of reactive systems such as identifying inconsistent requirements during the design process. We have implemented the algorithm and evaluated it on three different benchmark series, including consistency checks and assumption validations: The algorithm terminates on many randomly generated formulas within one second and scales particularly well for satisfiable formulas. Moreover, it is able to prove or disprove consistency of realistic benchmarks and to validate or invalidate their assumptions.

References

1. D'Antoni, L., Ferreira, T., Sammartino, M., Silva, A.: Symbolic Register Automata. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11561, pp. 3–21. Springer (2019), https://doi.org/10.1007/978-3-030-25540-4_1
2. D'Antoni, L., Veanes, M.: The Power of Symbolic Automata and Transducers. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10426, pp. 47–67. Springer (2017), https://doi.org/10.1007/978-3-319-63387-9_3
3. D'Antoni, L., Veanes, M.: Automata modulo Theories. Commun. ACM **64**(5), 86–95 (2021), https://doi.org/10.1145/3419404
4. Demri, S.: Linear-time Temporal Logics with Presburger Constraints: An Overview. J. Appl. Non Class. Logics **16**(3-4), 311–348 (2006), https://doi.org/10.3166/jancl.16.311-347
5. Demri, S.: LTL Over Integer Periodicity Constraints. Theor. Comput. Sci. **360**(1-3), 96–123 (2006), https://doi.org/10.1016/j.tcs.2006.02.019
6. Demri, S., D'Souza, D.: An Automata-Theoretic Approach to Constraint LTL. Inf. Comput. **205**(3), 380–415 (2007), https://doi.org/10.1016/j.ic.2006.09.006
7. Demri, S., D'Souza, D., Gascon, R.: A Decidable Temporal Logic of Repeating Values. In: Artëmov, S.N., Nerode, A. (eds.) Logical Foundations of Computer Science, International Symposium, LFCS 2007, New York, NY, USA, June 4-7, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4514, pp. 180–194. Springer (2007), https://doi.org/10.1007/978-3-540-72734-7_13
8. Demri, S., Lazic, R.: LTL with the Freeze Quantifier and Register Automata. In: 21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings. pp. 17–26. IEEE Computer Society (2006), https://doi.org/10.1109/LICS.2006.31
9. Demri, S., Lazic, R., Nowak, D.: On the Freeze Quantifier in Constraint LTL: Decidability and Complexity. In: 12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA. pp. 113–121. IEEE Computer Society (2005), https://doi.org/10.1109/TIME.2005.28
10. Demri, S., Lazic, R., Nowak, D.: On the Freeze Quantifier in Constraint LTL: Decidability and Complexity. In: 12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA. pp. 113–121. IEEE Computer Society (2005), https://doi.org/10.1109/TIME.2005.28
11. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0 - A Framework for LTL and ω-Automata Manipulation. In: Artho, C., Legay, A., Peled, D. (eds.) Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9938, pp. 122–129 (2016), https://doi.org/10.1007/978-3-319-46520-3_8
12. Exibard, L., Filiot, E., Reynier, P.: Synthesis of Data Word Transducers. Log. Methods Comput. Sci. **17**(1) (2021), https://lmcs.episciences.org/7279
13. Finkbeiner, B., Heim, P., Passing, N.: Temporal Stream Logic modulo Theories. CoRR **abs/2104.14988v1** (2021), https://arxiv.org/abs/2104.14988v1

14. Finkbeiner, B., Heim, P., Passing, N.: Temporal Stream Logic modulo Theories (Full Version). CoRR **abs/2104.14988v2** (2021), https://arxiv.org/abs/2104.14988v2
15. Finkbeiner, B., Klein, F., Piskac, R., Santolucito, M.: Temporal Stream Logic: Synthesis Beyond the Bools. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11561, pp. 609–629. Springer (2019), https://doi.org/10.1007/978-3-030-25540-4_35
16. Fischer, M.J., Ladner, R.E.: Propositional Dynamic Logic of Regular Programs. J. Comput. Syst. Sci. **18**(2), 194–211 (1979), https://doi.org/10.1016/0022-0000(79)90046-1
17. Geier, G., Heim, P., Klein, F., Finkbeiner, B.: Syntroids: Synthesizing a Game for FPGAs using Temporal Logic Specifications. In: 2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019. pp. 138–146. IEEE (2019), https://doi.org/10.23919/FMCAD.2019.8894261
18. Harel, D.: First-Order Dynamic Logic, Lecture Notes in Computer Science, vol. 68. Springer (1979), https://doi.org/10.1007/3-540-09237-4
19. Harel, D., Meyer, A.R., Pratt, V.R.: Computability and Completeness in Logics of Programs (Preliminary Report). In: Hopcroft, J.E., Friedman, E.P., Harrison, M.A. (eds.) Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA. pp. 261–268. ACM (1977), https://doi.org/10.1145/800105.803416
20. Jacobs, S., Klein, F., Schirmer, S.: A High-level LTL Synthesis Format: TLSF v1.1. In: Piskac, R., Dimitrova, R. (eds.) Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016. EPTCS, vol. 229, pp. 112–132 (2016), https://doi.org/10.4204/EPTCS.229.10
21. Kaminski, M., Francez, N.: Finite-Memory Automata. Theor. Comput. Sci. **134**(2), 329–363 (1994), https://doi.org/10.1016/0304-3975(94)90242-9
22. Khalimov, A., Maderbacher, B., Bloem, R.: Bounded Synthesis of Register Transducers. In: Lahiri, S.K., Wang, C. (eds.) Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11138, pp. 494–510. Springer (2018), https://doi.org/10.1007/978-3-030-01090-4_29
23. Krogmeier, P., Mathur, U., Murali, A., Madhusudan, P., Viswanathan, M.: Decidable Synthesis of Programs with Uninterpreted Functions. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12225, pp. 634–657. Springer (2020), https://doi.org/10.1007/978-3-030-53291-8_32
24. Lamport, L.: The Temporal Logic of Actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (1994), https://doi.org/10.1145/177492.177726
25. Li, J., Zhang, L., Pu, G., Vardi, M.Y., He, J.: LTL Satisfiability Checking Revisited. In: Sánchez, C., Venable, K.B., Zimányi, E. (eds.) 2013 20th International Symposium on Temporal Representation and Reasoning, Pensacola, FL, USA, September 26-28, 2013. pp. 91–98. IEEE Computer Society (2013), https://doi.org/10.1109/TIME.2013.19
26. Lisitsa, A., Potapov, I.: Temporal Logic with Predicate λ-Abstraction. In: 12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA. pp. 147–155. IEEE Computer Society (2005), https://doi.org/10.1109/TIME.2005.34

27. Maderbacher, B., Bloem, R.: Reactive Synthesis Modulo Theories Using Abstraction Refinement. CoRR **abs/2108.00090** (2021), https://arxiv.org/abs/2108.00090
28. Manna, Z., Pnueli, A.: Verification of Concurrent Programs: The Temporal Framework. In: Boyer, R.S., Moore, J.S. (eds.) The Correctness Problem in Computer Science. Academic Press, London (1981)
29. Mathur, U., Madhusudan, P., Viswanathan, M.: Decidable Verification of Uninterpreted Programs. Proc. ACM Program. Lang. **3**(POPL), 46:1–46:29 (2019), https://doi.org/10.1145/3290359
30. Mathur, U., Madhusudan, P., Viswanathan, M.: What's Decidable About Program Verification Modulo Axioms? In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12079, pp. 158–177. Springer (2020), https://doi.org/10.1007/978-3-030-45237-7_10
31. de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008), https://doi.org/10.1007/978-3-540-78800-3_24
32. Pnueli, A.: The Temporal Logic of Programs. In: Annual Symposium on Foundations of Computer Science, 1977. pp. 46–57. IEEE Computer Society (1977)
33. Pratt, V.R.: Semantical Considerations on Floyd-Hoare Logic. In: 17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25-27 October 1976. pp. 109–121. IEEE Computer Society (1976), https://doi.org/10.1109/SFCS.1976.27
34. Pratt, V.R.: A Practical Decision Method for Propositional Dynamic Logic: Preliminary Report. In: Lipton, R.J., Burkhard, W.A., Savitch, W.J., Friedman, E.P., Aho, A.V. (eds.) Proceedings of the 10th Annual ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, USA. pp. 326–337. ACM (1978), https://doi.org/10.1145/800133.804362
35. Ramakrishna, Y.S.: On the Satisfiability Problem for Lamport's Propositional Temporal Logic of Actions and Some of Its Extensions. Fundam. Informaticae **24**(4), 387–405 (1995), https://doi.org/10.3233/FI-1995-2444
36. Rozier, K.Y., Vardi, M.Y.: LTL Satisfiability Checking. In: Bosnacki, D., Edelkamp, S. (eds.) Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4595, pp. 149–167. Springer (2007), https://doi.org/10.1007/978-3-540-73370-6_11
37. Sistla, A.P., Clarke, E.M.: The Complexity of Propositional Linear Temporal Logics. J. ACM **32**(3), 733–749 (1985), https://doi.org/10.1145/3828.3837
38. Vardi, M.Y., Wolper, P.: Reasoning About Infinite Computations. Inf. Comput. **115**(1), 1–37 (1994), https://doi.org/10.1006/inco.1994.1092

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.