SMT-based Symbolic Model-Checking for Operator Precedence Languages

Michele Chiari[1][0000−0001−7742−9233], Luca Geatti[2][0000−0002−7125−787X], Nicola Gigante[3][0000−0002−2254−4821], and Matteo Pradella[4][0000−0003−3039−1084]

[1] TU Wien, Treitlstraße 3, 1040 Vienna, Austria
michele.chiari@tuwien.ac.at
[2] University of Udine, Italy
luca.geatti@uniud.it
[3] Free University of Bozen-Bolzano, Italy
nicola.gigante@unibz.it
[4] Politecnico di Milano, Italy
matteo.pradella@polimi.it

Abstract. Operator Precedence Languages (OPL) have been recently identified as a suitable formalism for model checking recursive procedural programs, thanks to their ability of modeling the program stack. OPL requirements can be expressed in the *Precedence Oriented Temporal Logic* (POTL), which features modalities to reason on the natural matching between function calls and returns, exceptions, and other advanced programming constructs that previous approaches, such as Visibly Pushdown Languages, cannot model effectively. Existing approaches for model checking of POTL have been designed following the explicit-state, automata-based approach, a feature that severely limits their scalability. In this paper, we give the first symbolic, SMT-based approach for model checking POTL properties. While previous approaches construct the automaton for both the POTL formula and the model of the program, we encode them into a (sequence of) SMT formulas. The search of a trace of the model witnessing a violation of the formula is then carried out by an SMT-solver, in a Bounded Model Checking fashion. We carried out an experimental evaluation, which shows the effectiveness of the proposed solution.

Keywords: SMT-based Model Checking · Tree-shaped Tableau · Temporal Logic · Operator Precedence Languages.

1 Introduction

Operator Precedence Languages (OPL) [16] are very promising for software verification: as a subclass of context-free languages, they can naturally encode the typical stack-based behavior of programs, without the shortcomings of the better known Visibly Pushdown Languages (VPL), originally introduced as Input-driven languages [5,6,30]. In particular, the main characteristic of VPL is the one-to-one "matching" between a symbol representing a procedure call and the

symbol representing its corresponding return. Unfortunately, this feature makes them ill-suited to model several typical behaviors of programs that induce a many-to-one or one-to-many matching, such as exceptions, interrupts, dynamic memory management, transactions, and continuations.

OPL were introduced through grammars for deterministic parsing by Floyd in 1963, and were re-discovered and studied in more recent works, where containment of VPL and closure w.r.t. Boolean operations were proved [15], together with the following characterizations: automata-based, monadic second order logic [26], regular-like expressions [28], and syntactic congruence with finitely many equivalence classes [22]. OPL are also the biggest known class maintaining an important feature of Regular languages: first-order logic, star-free expressions, and aperiodicity define the same subclass [29]. A temporal logic called OPTL was defined in [11], and a subsequent extension called POTL (on which we focus in this work) was introduced in [12], and then proved to capture the first-order definable fragment of OPL in [13]. The linear temporal logics for VPL CaRet [4] and NWTL [2] were also proved to be less expressive than both OPTL [11] and POTL [13].

POTL contains explicit context-free modalities that interact not only with the linear order of events representing time, but also with the nested structure of function calls, returns, and exceptions. For instance, consider this formula:

$$\square(\mathbf{call} \wedge \mathrm{qs} \rightarrow \neg(\bigcirc^u \mathbf{exc} \vee \chi_F^u \mathbf{exc}))$$

Here $\square$ is the LTL globally operator, and **call** and **exc** hold respectively in positions that represent a function call and an exception. $\bigcirc^u \mathbf{exc}$ means that the *next* position is an exception (similarly to the LTL next), while $\chi_F^u \mathbf{exc}$ means that a subsequent position, which *terminates* the function call in the current position, is an exception. Thus, the formula means "function qs is never terminated by an exception" (or, equivalently, it never terminates or it always terminates with a normal return).

It is worth to note that VPL were originally proposed for automatic verification, thanks to their nice Regular-like closure properties, but effective Model Checking (MC) tools for them are still not publicly available, in particular supporting logics capable of expressing context-free specifications. This situation improved with the introduction of POMC [8,12,10], a model checker for structured context-free languages based on POTL, but that can be easily adapted to the simpler structure of VPL. POMC's core consists of an explicit-state tableau construction procedure, which yields nondeterministic automata of size at most singly exponential in the formula's length, and is shown to be quite effective in realistic cases in [32,10].

The main shortcoming of explicit-state MC tools is the state explosion problem, i.e. the exponential growth of the state space as the system size and complexity increase, which makes MC infeasible for large and realistic systems. Indeed, as reported in [10], managing longer arrays or variables encoded with a realistic number of bits was problematic. A classical way to address this issue is to use Symbolic Model Checking, which is a variant of MC that represents the

system and the specification using symbolic data structures, instead of explicit enumeration of states and transitions. One very successful symbolic technique is Bounded Model Checking (BMC) [7,14], where the model is unrolled for a fixed number of steps and encoded into SAT, i.e. Boolean Satisfiability, to leverage recent efficient SAT solvers, and later the more general Satisfiability Modulo Theories (SMT) solvers, such as Z3 [31].

In this paper we apply BMC to POTL by encoding its tableau into SMT, extending the approach used in the BLACK tool [19]. BLACK is a satisfiability checker and temporal reasoning framework based on an encoding into SAT of Reynolds' one-pass tableau system for classical linear temporal logic [18]. Currently, we consider the future fragment of the temporal logic POTL on finite-word semantics, but we plan to extend the encoding to cover full POTL and ω-words. SMT-based approaches were already introduced for verifying pushdown program models [23,25], but only against regular specifications. To the best of our knowledge, this is the first SMT encoding of a context-free temporal logic, proving that BMC can be beneficial to verification of this class of temporal logics, too.

We applied our tool to a number of realistic cases: an implementation of the Quicksort algorithm, a banking application, and C++ implementations of a generic stack data structure, where our approach is compared with the original POMC. The results are very promising, as our SMT-based approach was able to avoid POMC's exponential increase of the solving time in several cases.

The paper is structured as follows. OPL and the logic POTL are introduced in Section 2. Section 3 defines the tree-shaped tableau for POTL, while Section 4 presents its encoding into SMT. Section 5 illustrates the experimental evaluation. Last, Section 6 draws the conclusions.

2 Preliminaries

2.1 Operator Precedence Languages

We assume that the reader has some familiarity with formal language theory concepts such as context-free grammar, parsing, shift-reduce algorithm [20,21]. Operator Precedence Languages (OPL) were historically defined through their generating grammars [16]; in this paper, we characterize them through their automata [26], as they are more suitable for model checking. Readers not familiar with OPL may refer to [27] for more explanations on their basic concepts.

Let Σ be a finite alphabet, and ε the empty string. We use a special symbol $\# \notin \Sigma$ to mark the beginning and the end of any string. An *operator precedence matrix* (OPM) M over Σ is a partial function $(\Sigma \cup \{\#\})^2 \to \{\lessdot, \doteq, \gtrdot\}$, that, for each ordered pair (a, b), defines the *precedence relation* (PR) $M(a, b)$ holding between a and b. If the function is total we say that M is *complete*. We call the pair (Σ, M) an *operator precedence alphabet*. Relations $\lessdot, \doteq, \gtrdot$, are respectively named *yields precedence, equal in precedence*, and *takes precedence*. By convention, the initial # yields precedence, and other symbols take precedence on the ending #. If $M(a, b) = \pi$, where $\pi \in \{\lessdot, \doteq, \gtrdot\}$, we write $a\ \pi\ b$. For $u, v \in \Sigma^+$ we

write $u \pi v$ if $u = xa$ and $v = by$ with $a \pi b$. The role of PR is to give structure to words: they can be seen as special and more concise parentheses, where e.g. one "closing" $\gtrdot$ can match more than one "opening" $\lessdot$. It is important to remark that PR are not ordering relations, despite their graphical appearance.

Definition 1. *An* operator precedence automaton (OPA) *is a tuple* $\mathcal{A} = (\Sigma, M, Q, I, F, \delta)$ *where* (Σ, M) *is an operator precedence alphabet,* Q *is a finite set of states,* $I \subseteq Q$ *is the set of initial states,* $F \subseteq Q$ *is the set of final states,* δ *is a triple of transition relations* $\delta_{shift} \subseteq Q \times \Sigma \times Q$, $\delta_{push} \subseteq Q \times \Sigma \times Q$, *and* $\delta_{pop} \subseteq Q \times Q \times Q$. *An OPA is deterministic iff* I *is a singleton, and all three components of* δ *are functions.*

To define the semantics of OPA, we set some notation. Letters $p, q, p_i, q_i, \ldots$ denote states in Q. We use $q_0 \xrightarrow{a} q_1$ for $(q_0, a, q_1) \in \delta_{push}$, $q_0 \overset{a}{\dashrightarrow} q_1$ for $(q_0, a, q_1) \in \delta_{shift}$, $q_0 \overset{q_2}{\Longrightarrow} q_1$ for $(q_0, q_2, q_1) \in \delta_{pop}$, and $q_0 \overset{w}{\rightsquigarrow} q_1$, if the automaton can read $w \in \Sigma^*$ going from q_0 to q_1. Let $\Gamma = \Sigma \times Q$ and $\Gamma' = \Gamma \cup \{\bot\}$ be the *stack alphabet*; we denote symbols in Γ' as $[a,\ q]$ or $\bot$. We set $smb([a,\ q]) = a$, $smb(\bot) = \#$, and $st([a,\ q]) = q$. For a stack content $\gamma = \gamma_n \ldots \gamma_1 \bot$, with $\gamma_i \in \Gamma$, $n \geq 0$, we set $smb(\gamma) = smb(\gamma_n)$ if $n \geq 1$, $smb(\gamma) = \#$ if $n = 0$.

A *configuration* of an OPA is a triple $c = \langle w,\ q,\ \gamma \rangle$, where $w \in \Sigma^*\#$, $q \in Q$, and $\gamma \in \Gamma^*\bot$. A *computation* or *run* is a finite sequence $c_0 \vdash c_1 \vdash \ldots \vdash c_n$ of *moves* or *transitions* $c_i \vdash c_{i+1}$. There are three kinds of moves, depending on the PR between the symbol on top of the stack and the next input symbol:
push move: if $smb(\gamma) \lessdot a$ then $\langle ax,\ p,\ \gamma \rangle \vdash \langle x,\ q,\ [a,\ p]\gamma \rangle$, with $(p, a, q) \in \delta_{push}$;
shift move: if $a \doteq b$ then $\langle bx,\ q,\ [a,\ p]\gamma \rangle \vdash \langle x,\ r,\ [b,\ p]\gamma \rangle$, with $(q, b, r) \in \delta_{shift}$;
pop move: if $a \gtrdot b$ then $\langle bx,\ q,\ [a,\ p]\gamma \rangle \vdash \langle bx,\ r,\ \gamma \rangle$, with $(q, p, r) \in \delta_{pop}$.

Shift and pop moves are not performed when the stack contains only $\bot$. Push moves put a new element on top of the stack consisting of the input symbol together with the current state of the OPA. Shift moves update the top element of the stack by *changing its input symbol only.* Pop moves remove the element on top of the stack, and update the state of the OPA according to δ_{pop} on the basis of the current state of the OPA and the state of the removed stack symbol. They do not consume the input symbol, which is used only to establish the $\gtrdot$ relation, remaining available for the next move. The OPA accepts the language $L(\mathcal{A}) = \{x \in \Sigma^* \mid \langle x\#,\ q_I,\ \bot \rangle \vdash^* \langle \#,\ q_F,\ \bot \rangle, q_I \in I, q_F \in F\}$.

We now introduce the concept of *chain*, which makes the connection between OP relations and context-free structure explicit, through brackets.

Definition 2. *A* simple chain ${}^{c_0}[c_1 c_2 \ldots c_\ell]^{c_{\ell+1}}$ *is a string* $c_0 c_1 c_2 \ldots c_\ell c_{\ell+1}$, *such that:* $c_0, c_{\ell+1} \in \Sigma \cup \{\#\}$, $c_i \in \Sigma$ *for every* $i = 1, 2, \ldots \ell$ *(*$\ell \geq 1$*), and* $c_0 \lessdot c_1 \doteq c_2 \ldots c_{\ell-1} \doteq c_\ell \gtrdot c_{\ell+1}$. *A* composed chain *is a string* $c_0 s_0 c_1 s_1 c_2 \ldots c_\ell s_\ell c_{\ell+1}$, *where* ${}^{c_0}[c_1 c_2 \ldots c_\ell]^{c_{\ell+1}}$ *is a simple chain, and* $s_i \in \Sigma^*$ *is the empty string or is such that* ${}^{c_i}[s_i]^{c_{i+1}}$ *is a chain (simple or composed), for every* $i = 0, 1, \ldots, \ell$ *(*$\ell \geq 1$*). Such a composed chain will be written as* ${}^{c_0}[s_0 c_1 s_1 c_2 \ldots c_\ell s_\ell]^{c_{\ell+1}}$. c_0 *(resp.* $c_{\ell+1}$*) is called its* left *(resp.* right*)* context*; all symbols between them form its* body.

	call	**ret**	**han**	**exc**
call	$\lessdot$	$\doteq$	$\lessdot$	$\gtrdot$
ret	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
han	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$
exc	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$

1	$\# \lessdot \textbf{call} \lessdot \textbf{han} \lessdot \underline{\textbf{call}} \gtrdot \textbf{exc} \gtrdot \textbf{call} \doteq \textbf{ret} \gtrdot \textbf{ret} \gtrdot \#$
2	$\# \lessdot \textbf{call} \lessdot \underline{\textbf{han}} \doteq \underline{\textbf{exc}} \gtrdot \textbf{call} \doteq \textbf{ret} \gtrdot \textbf{ret} \gtrdot \#$
3	$\# \lessdot \textbf{call} \lessdot \underline{\textbf{call}} \doteq \underline{\textbf{ret}} \gtrdot \textbf{ret} \gtrdot \#$
4	$\# \lessdot \underline{\textbf{call}} \doteq \underline{\textbf{ret}} \gtrdot \#$
5	$\# \doteq \#$

$$\#[\textbf{call}[[\textbf{han}[\textbf{call}]\textbf{exc}]\textbf{call ret}]\textbf{ret}]\#$$

Fig. 1: OPM $M_{\textbf{call}}$ (left), a string with chains shown by brackets (bottom), and its parsing steps using the OP algorithm (right).

A finite word w over Σ is *compatible* with an OPM M iff for each pair of letters c, d, consecutive in w, $M(c, d)$ is defined and, for each substring x of $\#w\#$ that is a chain of the form ${}^{a}[y]^{b}$, $M(a, b)$ is defined.

Chains can be identified through the traditional operator precedence parsing algorithm. We apply it to the sample word $w_{ex} = \textbf{call han call exc call ret ret}$, which is compatible with $M_{\textbf{call}}$. First, write all precedence relations between consecutive characters, according to $M_{\textbf{call}}$. Then, recognize all innermost patterns of the form $a \lessdot c \doteq \cdots \doteq c \gtrdot b$ as simple chains, and remove their bodies. Then, write the precedence relations between the left and right contexts of the removed body, a and b, and iterate this process until only $\#\#$ remains. This procedure is applied to w_{ex} and illustrated in Fig. 1 (right). The chain body removed in each step is underlined. In step 1 we recognize the simple chain ${}^{\textbf{han}}[\underline{\textbf{call}}]^{\textbf{exc}}$, which can be removed. In the next steps we recognize as chains first ${}^{\textbf{call}}[\textbf{han exc}]^{\textbf{call}}$, then ${}^{\textbf{call}}[\textbf{call ret}]^{\textbf{ret}}$, and last ${}^{\#}[\textbf{call ret}]^{\#}$. Fig. 1 (bottom) reports the chain structure of w_{ex}.

Let $\mathcal{A}$ be an OPA. We call a *support* for the simple chain ${}^{c_0}[c_1 c_2 \dots c_\ell]^{c_{\ell+1}}$ any path in $\mathcal{A}$ of the form $q_0 \xrightarrow{c_1} q_1 \dashrightarrow \dots \dashrightarrow q_{\ell-1} \xrightarrow{c_\ell} q_\ell \overset{q_0}{\Longrightarrow} q_{\ell+1}$. The label of the last (and only) pop is exactly q_0, i.e. the first state of the path; this pop is executed because of relation $c_\ell \gtrdot c_{\ell+1}$. We call a *support for the composed chain* ${}^{c_0}[s_0 c_1 s_1 c_2 \dots c_\ell s_\ell]^{c_{\ell+1}}$ any path in $\mathcal{A}$ of the form $q_0 \overset{s_0}{\rightsquigarrow} q'_0 \xrightarrow{c_1} q_1 \overset{s_1}{\rightsquigarrow} q'_1 \xrightarrow{c_2} \dots \xrightarrow{c_\ell} q_\ell \overset{s_\ell}{\rightsquigarrow} q'_\ell \overset{q'_0}{\Longrightarrow} q_{\ell+1}$ where, for every $i = 0, 1, \dots, \ell$: if $s_i \neq \epsilon$, then $q_i \overset{s_i}{\rightsquigarrow} q'_i$ is a support for the chain ${}^{c_i}[s_i]^{c_{i+1}}$, else $q'_i = q_i$.

Chains fully determine the parsing structure of any OPA over (Σ, M). If the OPA performs the computation $\langle sb, q_i, [a, q_j]\gamma\rangle \vdash^* \langle b, q_k, \gamma\rangle$, then ${}^{a}[s]^{b}$ is necessarily a chain over (Σ, M), and there exists a support like the one above with $s = s_0 c_1 \dots c_\ell s_\ell$ and $q_{\ell+1} = q_k$. This corresponds to the parsing of the string $s_0 c_1 \dots c_\ell s_\ell$ within the contexts a,b, which contains all information needed to build the subtree whose frontier is that string.

In [15] it is proved that Visibly Pushdown Languages (VPL) [5] are strictly included in OPL. In VPL the input alphabet is partitioned into three disjoint sets, namely of *call* (Σ_c), *return* (Σ_r), and *internal* (Σ_i) symbols, where *calls* and *returns* respectively play the role of open and closed parentheses. Intuitively, the

string structure determined by these alphabets can be represented through an OPM as follows: $a \lessdot b$, for any $a \in \Sigma_c$, $b \in \Sigma_c \cup \Sigma_i$; $a \doteq b$, for any $a \in \Sigma_c$, $b \in \Sigma_r$; $a \gtrdot b$, for all the other cases. On the other hand, the OPM that we use in this paper cannot be expressed in VPL, because the typical behavior of exceptions cannot be modeled with the limited one-to-one structure of calls and returns.

To sum up, given an OP alphabet, the OPM M assigns a unique structure to any compatible string in Σ^*; unlike VPL, such a structure is not visible in the string, and must be built by means of a non-trivial parsing algorithm. An OPA defined on the OP alphabet selects an appropriate subset within the "universe" of strings compatible with M.

2.2 Precedence Oriented Temporal Logic

POTL is a propositional linear-time temporal logic featuring context-free modalities based on OPL. Here we are only interested in its future fragment, $\mathsf{POTL_f}$ (the letter "f" stands for "future"), with the addition of *weak* operators, which are needed for our tableau. In this paper, we focus on the finite words semantics for $\mathsf{POTL_f}$.

We fix a finite set of atomic propositions AP. $\mathsf{POTL_f}$ semantics are based on OP words, which are tuples $(U, <, M_{AP}, P)$, where $U = \{0, \dots, n\}$, $n \in \mathbb{N}$, is a finite set of word positions, $<$ a linear order on them, M_{AP} an OPM on $\mathcal{P}(AP)$, and $P : U \to \mathcal{P}(U)$ a labeling function, with $0, n \in P(\#)$. From M_{AP} follows the *chain relation* $\chi \subseteq U^2$, such that $\chi(i, j)$ holds iff i and j are resp. the left and right contexts of a chain. We only define the OPM on propositions in **bold**, called *structural*, and assume that only one of them holds in each position. If $\mathbf{l}_1 \sim \mathbf{l}_2$ for any PR $\sim$ and $i \in P(\mathbf{l}_1)$ and $j \in P(\mathbf{l}_2)$, we write $i \sim j$.

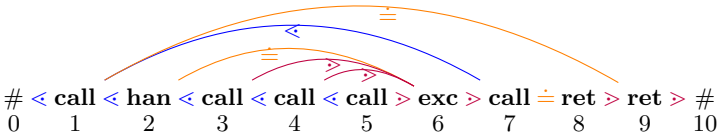


Fig. 2: An example OP word, with the χ relation depicted by arrows, and PRs. First, a procedure is called (pos. 1), which installs an exception handler in pos. 2. Then, another function throws an exception, which is caught by the handler. Another function is called and returns and, finally, the initial one also returns.

$\mathsf{POTL_f}$ offers next and until operators based on two different kinds of paths, which we define below, after fixing an OP word w.

Definition 3. *The* downward summary path (DSP) *between positions i and j, denoted $\pi^d_\chi(w, i, j)$, is a set of positions $i = i_1 < i_2 < \cdots < i_n = j$ such that, for*

each $1 \leq p < n$,

$$i_{p+1} = \begin{cases} k & \text{if } k = \max\{h \mid h \leq j \wedge \chi(i_p, h) \wedge (i_p \lessdot h \vee i_p \doteq h)\} \text{ exists;} \\ i_p + 1 & \text{otherwise, if } i_p \lessdot (i_p + 1) \text{ or } i_p \doteq (i_p + 1). \end{cases}$$

We write $\pi^d_\chi(w, i, j) = \varnothing$ *if no such path exists. The definition for* $\pi^u_\chi(w, i, j)$ *is obtained by substituting* $\gtrdot$ *for* $\lessdot$.

DSPs can either go downward in the nesting structure of the χ relation by following the linear order, or skip whole chain bodies by following the χ relation. What this means depends on the OPM: with $M_{\mathbf{call}}$, until operators on DSPs express properties local to a function invocation, including children calls. Their upward counterparts, instead, go from inner functions towards parent invocations. For instance, in Fig. 2 we have $\pi^d_\chi(w, 1, 6) = \{1, 5, 6\}$, and $\pi^u_\chi(w, 2, 7) = \{2, 4, 5, 6, 7\}$.

Definition 4. *The* downward hierarchical path *between positions* i *and* j*, denoted* $\pi^d_H(w, i, j)$*, is a sequence of positions* $i = i_1 < i_2 < \cdots < i_n = j$ *such that there exists* $h > j$ *such that for each* $1 \leq p \leq n$ *we have* $\chi(i_p, h)$ *and* $i_p \gtrdot h$*, and for each* $1 \leq q < n$ *there is no position* k *such that* $i_q < k < i_{q+1}$ *and* $\chi(k, h)$*.*

The upward hierarchical path $\pi^u_H(w, i, j)$ *is defined similarly, except* $h < j$ *and for all* $1 \leq p \leq n$ *we have* $\chi(h, i_p)$ *and* $h \lessdot i_p$*.*

We write $\pi^d_H(w, i, j) = \varnothing$ *or* $\pi^u_H(w, i, j) = \varnothing$ *if no such path exists.*

Hierarchical paths range between multiple positions in the χ relation with the same one. With $M_{\mathbf{call}}$, this means functions terminated by the same exception. For instance, in Fig. 2 we have $\pi^d_H(w, 3, 4) = \{3, 4\}$.

Let $\mathrm{a} \in AP$, and $t \in \{d, u\}$; the syntax of $\mathsf{POTL_f}$ is the following:

$$\begin{aligned} \varphi \coloneqq \mathrm{a} \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc^t \varphi \mid \tilde{\bigcirc}^t \varphi \mid \chi^t_F \varphi \mid \tilde{\chi}^t_F \varphi \mid \varphi \, \mathcal{U}^t_\chi \, \varphi \mid \varphi \, \mathcal{R}^t_\chi \, \varphi \\ \mid \bigcirc^t_H \varphi \mid \tilde{\bigcirc}^t_H \varphi \mid \varphi \, \mathcal{U}^t_H \, \varphi \mid \varphi \, \mathcal{R}^t_H \, \varphi \end{aligned}$$

The truth of $\mathsf{POTL_f}$ formulas is defined w.r.t. a single word position. Let w be a finite OP word, and $\mathrm{a} \in AP$; we set $\sim^d = \lessdot$ and $\sim^u = \gtrdot$. Then, for any position $i \in U$ of w and $t \in \{d, u\}$:

1. $(w, i) \models \mathrm{a}$ iff $i \in P(\mathrm{a})$;
2. $(w, i) \models \neg\varphi$ iff $(w, i) \not\models \varphi$;
3. $(w, i) \models \varphi_1 \vee \varphi_2$ iff $(w, i) \models \varphi_1$ or $(w, i) \models \varphi_2$;
4. $(w, i) \models \bigcirc^t \varphi$ iff $i < |w| - 1$, $(w, i + 1) \models \varphi$ and $i \sim^t (i + 1)$ or $i \doteq (i + 1)$;
5. $(w, i) \models \tilde{\bigcirc}^t \varphi$ iff $i = |w| - 1$ and $(i \sim^t (i+1)$ or $i \doteq (i+1))$ implies $(w, i+1) \models \varphi$;
6. $(w, i) \models \chi^t_F \varphi$ iff $\exists j > i$ such that $\chi(i, j)$, $i \sim^t j$ or $i \doteq j$, and $(w, j) \models \varphi$;
7. $(w, i) \models \tilde{\chi}^t_F \varphi$ iff $\forall j > i$ such that $\chi(i, j)$ and $(i \sim^t j$ or $i \doteq j)$, we have $(w, j) \models \varphi$;
8. $(w, i) \models \varphi_1 \, \mathcal{U}^t_\chi \, \varphi_2$ iff $\exists j \geq i$ such that $\pi^t_\chi(w, i, j) \neq \varnothing$, $(w, j) \models \varphi_2$ and $\forall j' < j$ in $\pi^t_\chi(w, i, j)$ we have $(w, j) \models \varphi_1$;
9. $(w, i) \models \varphi_1 \, \mathcal{R}^t_\chi \, \varphi_2$ iff $\forall j \geq i$ such that $\pi^t_\chi(w, i, j) \neq \varnothing$ we have either $(w, j') \models \varphi_2$ for all $j' \in \pi^t_\chi(w, i, j)$, or $\exists k \in \pi^t_\chi(w, i, j)$ such that $(w, k) \models \varphi_1$ and $\forall j' \leq k$ in $\pi^t_\chi(w, i, j)$ we have $(w, j) \models \varphi_2$;

10. $(w,i) \models \bigcirc^u_H \varphi$ iff there exist a position $h < i$ s.t. $\chi(h,i)$ and $h \lessdot i$ and a position $j = \min\{k \mid i < k \wedge \chi(h,k) \wedge h \lessdot k\}$ and $(w,j) \models \varphi$;
11. $(w,i) \models \tilde{\bigcirc}^u_H \varphi$ iff the existence of a position $h < i$ s.t. $\chi(h,i)$ and $h \lessdot i$ and a position $j = \min\{k \mid i < k \wedge \chi(h,k) \wedge h \lessdot k\}$ implies $(w,j) \models \varphi$;
12. $(w,i) \models \bigcirc^d_H \varphi$ iff there exist a position $h > i$ s.t. $\chi(i,h)$ and $i \gtrdot h$ and a position $j = \min\{k \mid i < k \wedge \chi(k,h) \wedge k \gtrdot h\}$ and $(w,j) \models \varphi$;
13. $(w,i) \models \tilde{\bigcirc}^d_H \varphi$ iff the existence of a position $h > i$ s.t. $\chi(i,h)$ and $i \gtrdot h$ and a position $j = \min\{k \mid i < k \wedge \chi(k,h) \wedge k \gtrdot h\}$ implies $(w,j) \models \varphi$;
14. $(w,i) \models \varphi_1 \, \mathcal{U}^t_H \, \varphi_2$ iff $\exists j \geq i$ such that $\pi^t_H(w,i,j) \neq \varnothing$, $(w,j) \models \varphi_2$ and $\forall j' < j$ in $\pi^t_H(w,i,j)$ we have $(w,j) \models \varphi_1$;
15. $(w,i) \models \varphi_1 \, \mathcal{R}^t_H \, \varphi_2$ iff $\forall j \geq i$ such that $\pi^t_H(w,i,j) \neq \varnothing$ we have either $(w,j') \models \varphi_2$ for all $j' \in \pi^t_\chi(w,i,j)$, or $\exists k \in \pi^t_H(w,i,j)$ such that $(w,k) \models \varphi_1$ and $\forall j' \leq k$ in $\pi^t_\chi(w,i,j)$ we have $(w,j) \models \varphi_2$.

We additionally employ $\wedge$ and $\rightarrow$ with the usual semantics.

For instance, formula $\top \, \mathcal{U}^d_\chi \, p$ evaluated in a function **call** means that p holds somewhere between the call and its matched return (or exception); formula $\chi^u_F \, p$, evaluated in a **call**, means that p will hold when it returns (this can be used to check post-conditions or, if $p =$ **exc**, to assert that the function is terminated by an exception). Formula $\top \, \mathcal{U}^d_H \, p$, when evaluated in a **call** terminated by an exception, means that p holds in one of the **call**s already terminated by the same exception. For a more in-depth presentation of POTL, we refer the reader to [13].

3 A tree-shaped tableau for $\mathsf{POTL_f}$

In this section, we describe our tableau system for $\mathsf{POTL_f}$, that will form the core of our bounded model checking procedure. Let Σ be a set of structural propositions, (Σ, M) an OP alphabet, AP a set of atomic propositions, and φ a formula over $\Sigma \cup AP$. Given $\Gamma \subseteq \mathrm{Cl}(\varphi)$, if $\Gamma \cap \Sigma = \{a\}$, then we define $\mathrm{struct}(\Gamma) = a$. Moreover, for $\Gamma, \Gamma' \subseteq \mathrm{Cl}(\varphi)$ and $\sim \in \{\lessdot, \doteq, \gtrdot\}$, we write $\Gamma \sim \Gamma'$ meaning $\mathrm{struct}(\Gamma) \sim \mathrm{struct}(\Gamma')$.

A tableau for φ is a tree built on top of a set of nodes N. Each node $u \in N$ has four labels: $\Gamma(u) \subseteq \mathrm{Cl}(\phi)$, $\mathrm{smb}(u) \in \Sigma$, $\mathrm{stack}(u) \in N \cup \{\bot\}$, $\mathrm{ctx}(u) \in N \cup \{\bot\}$. Each node u is a *push*, *shift*, or *pop* node if, respectively, $\mathrm{smb}(u) \lessdot \Gamma(u)$, $\mathrm{smb}(u) \doteq \Gamma(u)$, or $\mathrm{smb}(u) \gtrdot \Gamma(u)$.

The tableau is built from φ starting from the root u_0 which is labelled as $\Gamma(u_0) = \{\varphi\}$, $\mathrm{smb}(u_0) = \#$, $\mathrm{stack}(u_0) = \bot$, $\mathrm{ctx}(u_0) = \bot$. The tree is built by applying a set of *rules* to each leaf. Each rule may add new children nodes to the given leaf, while others may *accept* or *reject* the leaf. The construction continues until every leaf has been either accepted or rejected. The tableau rules can be divided into *expansion*, *termination*, *step*, and *guess* rules.

To each leaf of the tree, at first *expansion rules* are applied, which are summarised in Table 1. Each rule works as follows. If the formula ψ in the leftmost column belongs to $\Gamma(u)$, then for each $i \in \{1,2,3\}$ for which Γ_i is given in Table 1, a child u_i is added to u, whose labels are identical to u excepting that

Table 1: Expansion rules, where $t \in \{u, d\}$.

$\psi \in \Gamma(u)$	Γ_1	Γ_2	Γ_3
$\alpha \wedge \beta$	$\{\alpha, \beta\}$		
$\alpha \vee \beta$	$\{\alpha\}$	$\{\beta\}$	
$\alpha \,\mathcal{U}^u_H\, \beta$	$\{\alpha, \bigcirc^u_H(\alpha \,\mathcal{U}^u_H\, \beta)\}$	$\{\beta\}$ (only if condition 1 holds)	
$\alpha \,\mathcal{U}^d_H\, \beta$	$\{\alpha, \bigcirc^d_H(\alpha \,\mathcal{U}^d_H\, \beta)\}$	$\{\beta\}$ (only if condition 2 holds)	
$\alpha \,\mathcal{R}^t_\chi\, \beta$	$\{\alpha, \beta\}$	$\{\beta, \tilde{\bigcirc}^t(\alpha \,\mathcal{R}^t_\chi\, \beta), \tilde{\chi}^t_F(\alpha \,\mathcal{R}^t_\chi\, \beta)\}$	
$\alpha \,\mathcal{R}^u_H\, \beta$	$\{\alpha, \beta\}$	$\{\beta, \tilde{\bigcirc}^u_H(\alpha \,\mathcal{R}^u_H\, \beta)\}$	
$\alpha \,\mathcal{U}^t_\chi\, \beta$	$\{\beta\}$	$\{\alpha, \bigcirc^t(\alpha \,\mathcal{U}^t_\chi\, \beta)\}$	$\{\alpha, \chi^t_F(\alpha \,\mathcal{U}^t_\chi\, \beta)\}$
$\alpha \,\mathcal{R}^d_H\, \beta$	$\varnothing$	$\{\alpha, \beta\}$	$\{\beta, \tilde{\bigcirc}^u_H(\alpha \,\mathcal{R}^u_H\, \beta)\}$
		(only if condition 2 holds)	

condition 1: the closest step ancestor of u is a *pop* node u_p such that $\Gamma(\mathrm{ctx}(u_p)) \lessdot \Gamma(u_p)$

condition 2: the closest step ancestor of u is a *push* or *shift* node

$\Gamma(u_i) = (\Gamma(u) \setminus \{\psi\}) \cup \Gamma_i$. If multiple rules can be applied, the order in which they are applied does not matter.

When no expansion rules are applicable to a leaf u, and $\Gamma(u) \cap (\Sigma \cup \{\#\}) = \varnothing$, then one child u_a, for each $a \in \Sigma \cup \{\#\}$, is added to u whose labels are the same as u except that $\Gamma(u_a) = \Gamma(u) \cup \{a\}$.

When no expansion rules are applicable to a leaf u and $\Gamma(u) \cup \Sigma \neq \varnothing$, u is called a *step* node. In this case, *termination* rules are checked to decide whether the leaf can be either rejected or accepted. Rejecting rules are described in Table 2. Most rules depend on the type of the leaf node u where they are applied (*i.e.*, it being a push, pop, or shift node), and the type of the closest step ancestor u_s of u. The rule in a given row of the table fires when u and u_s are of the stated type (if any) and where the condition in the last column is met. In this case, u is rejected. We need to set up the following terminology in order to understand some of those rules.

Definition 5 (Fulfillment of a chain next operator). *A $\chi^d_F\, \alpha$ operator is said to be* fulfilled *in a node u iff $\chi^d_F\, \alpha \in \Gamma(u)$, and there exists a pop node descendant u_p such that* $\mathrm{ctx}(u_p) = u$ *and:*

1. *$\Gamma(u) \lessdot \Gamma(u_p)$ or $\Gamma(u) \doteq \Gamma(u_p)$, and*
2. *$\alpha \in \Gamma(u_s)$, where u_s is the closest push or shift node descending from u_p.*

Replace χ^d_F with χ^u_F and $\lessdot$ with $\gtrdot$ for the upward case.

Definition 6 (Pending node). *A node u is* pending *iff either:*

1. *u is a push node and no pop node u_p exists such that* $\mathrm{stack}(u_p) = u$*, or*

2. u is a shift node and no pop node u_p exists such that $\mathrm{stack}(u_p) = \mathrm{stack}(u)$.

Definition 7 (Equivalent nodes). *Two nodes u and u' belonging to the same branch are said to be* equivalent *if the following hold:*

1. $\Gamma(u) = \Gamma(u')$;
2. $\mathrm{smb}(u) = \mathrm{smb}(u')$;
3. $\Gamma(\mathrm{stack}(u)) = \Gamma(\mathrm{stack}(u'))$; *and*
4. $\Gamma(\mathrm{ctx}(u)) = \Gamma(\mathrm{ctx}(u'))$.

In contrast to rejecting rules, there is only one simple *accepting rule*: u is accepted when $\Gamma(u) = \{\#\}$ and $\mathrm{stack}(u) = \bot$.

If no termination rules fire on a step node u, the construction can proceed by a *temporal step*. To understand how it works, we need the following notation: given a node u and a unary temporal operator $\odot$, we denote the set of all the formulas that appear as arguments of $\odot$ inside $\Gamma(u)$ as $\mathcal{G}_{\odot}(u) = \{\alpha \mid \odot\,\alpha \in \Gamma(u)\}$, and for a set of operators $\{\odot_1, \ldots, \odot_n\}$ we define $\mathcal{G}_{\odot_1,\ldots,\odot_n}(u) = \mathcal{G}_{\odot_1}(u) \cup \ldots \cup \mathcal{G}_{\odot_n}(u)$. The temporal step consists in two parts: the application of one *step* rule, and of one *guess* rule. The step rules, summarised in Table 3, are chosen depending on the type of the leaf at hand, and of its closest step ancestor. Each rule adds exactly one child u' to the leaf u, whose label is described in the table. The child u' is then fed to one of the *guess* rules described in Table 4. The applicability of the guess rules depend on the type of u and some other conditions, in a way such that in each case at most one guess rule is applicable to u'. If any is applicable, the selected rule defines a set of formulas $\mathcal{G}$ as described in the table, and for each $G \subseteq \mathcal{G}$ adds a child u''_G such that $\Gamma(u''_G) = \Gamma(u') \cup G$, $\mathrm{smb}(u''_G) = \mathrm{smb}(u')$, $\mathrm{stack}(u''_G) = \mathrm{stack}(u')$, and $\mathrm{ctx}(u''_G) = \mathrm{ctx}(u')$. After the temporal step is completed, the construction continues with the expansion rules again, and everything repeats.

We can now sketch a soundness and termination argument for the tableau.

Theorem 1 (Soundness). *If the tableau for ϕ has an accepted branch, then ϕ is satisfiable.*

Proof (Sketch). $\mathrm{Cl}(\phi)$ is finite, and so is the number of possible node labels. Thus, unless they are rejected by a rule other than 13, all branches of the tableau must eventually reach a node that is *equivalent* (cf. Definition 7) to a previous one. Then, they are rejected by Rule 13. Thus, once fully expanded, the tableau for a formula ϕ is also finite. Then, soundness of the tableau can be proved by building a word out of any accepted tableau branch, with a mapping from push and shift step nodes of the branch to letters in the word. Chain supports in the word correspond to sequences of step nodes. See [9] for the full proof.

4 SMT Encoding of the Tableau

Our technique for symbolic model checking of $\mathsf{POTL_f}$ properties does not directly construct the tableau described in Section 3, but rather, it *encodes* it into SMT

Table 2: Rejecting termination rules.

n°	type of u	type of u_s[1]	condition
1.			$\{p, \neg p\} \subseteq \Gamma(u)$
2.			$\lvert\Gamma(u) \cap \Sigma\rvert > 1$
3.			$\{\psi, \#\} \subseteq \Gamma(u)$ and ψ is strong[2]
4.		push/shift	$\Gamma(u_s) \gtrdot \Gamma(u)$ and some $\bigcirc^d \alpha \in \Gamma(u_s)$[1]
		push/shift	$\Gamma(u_s) \lessdot \Gamma(u)$ and some $\bigcirc^u \alpha \in \Gamma(u_s)$[1]
5.		push/shift	$\Gamma(u_s) \lessdot \Gamma(u)$ or $\Gamma(u_s) \doteq \Gamma(u)$, and some $\tilde{\bigcirc}^d \alpha \in \Gamma(u_s)$, but $\alpha \notin \Gamma(u)$
		push/shift	$\Gamma(u_s) \gtrdot \Gamma(u)$ or $\Gamma(u_s) \doteq \Gamma(u)$, and some $\tilde{\bigcirc}^u \alpha \in \Gamma(u_s)$, but $\alpha \notin \Gamma(u)$
6.	pop		$\chi_F^t \alpha$ is *not* fulfilled in u', for some $u' \in G$ such that $\chi_F^t \alpha \in \Gamma(u')$[3], for $t \in \{d, u\}$
7.	push	pop	$\tilde{\chi}_F^d \alpha \in \mathrm{ctx}(u_s)$ and $\alpha \notin \Gamma(u)$
	shift	pop	$\tilde{\chi}_F^t \alpha \in \mathrm{ctx}(u_s)$ and $\alpha \notin \Gamma(u)$, for $t \in \{d, u\}$
	pop	pop	$\tilde{\chi}_F^u \alpha \in \mathrm{ctx}(u_s)$ and $\alpha \notin \Gamma(u)$
8.	pop		$\bigcirc_H^u \alpha \in \Gamma(\mathrm{stack}(u))$ and $\Gamma(\mathrm{ctx}(u)) \not\lessdot \Gamma(u)$
	push	pop	$\bigcirc_H^u \alpha \in \Gamma(\mathrm{stack}(u_s))$ and $\alpha \notin \Gamma(u)$
	push	push/shift	$\bigcirc_H^u \alpha \in \Gamma(u)$
	shift		$\bigcirc_H^u \alpha \in \Gamma(u)$
9.	push	pop	$\tilde{\bigcirc}_H^u \alpha \in \Gamma(\mathrm{stack}(u_s))$, $\mathrm{stack}(u_s)$ is a push node, the closest step ancestor of $\mathrm{stack}(u_s)$ is a pop node, and $\alpha \notin \Gamma(u)$
10.	pop		$\bigcirc_H^d \alpha \in \Gamma(\mathrm{ctx}(u))$ and $\mathrm{smb}(\mathrm{stack}(u)) \doteq \Gamma(u)$
	pop	push/shift	$\bigcirc_H^d \alpha \in \Gamma(\mathrm{ctx}(u))$
	pop	pop	$\bigcirc_H^d \alpha \in \Gamma(\mathrm{ctx}(u))$ and $\alpha \notin \Gamma(\mathrm{ctx}(u_s))$[1]
	pop/shift	pop/shift	$\bigcirc_H^d \alpha \in \Gamma(u_s)$[1]
11.	pop	pop	$\tilde{\bigcirc}_H^d \alpha \in \Gamma(\mathrm{ctx}(u))$, $\mathrm{smb}(\mathrm{stack}(u)) \gtrdot \Gamma(u)$, and $\alpha \notin \Gamma(\mathrm{ctx}(u_s))$
12.	pop/shift	push/shift	$\alpha \,\mathcal{U}_H^d\, \beta \in \Gamma(u_s)$
	push/shift	pop	$\alpha \,\mathcal{R}_H^d\, \beta$ appears in one of the nodes between $\mathrm{ctx}(u_s)$ and the closest step ancestor of u_s (exclusive)
	pop	pop	$\alpha, \tilde{\bigcirc}_H^d(\alpha \,\mathcal{R}_H^d\, \beta) \notin \Gamma(\mathrm{ctx}(u_s))$, $\alpha, \beta \notin \Gamma(\mathrm{ctx}(u_s))$, and $\alpha \,\mathcal{R}_H^d\, \beta$ appears in one of the nodes between $\mathrm{ctx}(u_s)$ and the closest step ancestor of u_s (exclusive)
13.	push/shift		there is a *pending* ancestor u_i of u *equivalent* to u[4]

[1] u_s is the closest step ancestor of u
[2] ψ is strong if it is a positive literal or a strong tomorrow
[3] $G = \{\mathrm{stack}(u)\} \cup \{u' \mid \mathrm{stack}(u') = \mathrm{stack}(u) \text{ and } u' \text{ is a shift node}\}$
[4] See Definitions 6 and 7.

Table 3: Step rules

u	u_s[1]	$\Gamma(u')$	smb(u')	stack(u')	ctx(u')
push	push/shift	$\mathcal{G}_{\bigcirc^d,\bigcirc^u}(u)$	struct($\Gamma(u)$)	u	u_s or $\bot$[2]
push	pop	$\mathcal{G}_{\bigcirc^d,\bigcirc^u}(u)$	struct($\Gamma(u)$)	u	ctx(u_s) or $\bot$[2]
shift		$\mathcal{G}_{\bigcirc^d,\bigcirc^u}(u)$	struct($\Gamma(u)$)	stack(u)	ctx(u)
pop		$\Gamma(u)$	smb(stack(u))	stack(stack(u))	ctx(stack(u))

[1] u_s is the closest step ancestor of u
[2] ctx(u') $= \bot$ if stack(u) $= \bot$

Table 4: Guess rules

u	only if	$\mathcal{G}$
push/shift		$\mathcal{G}_{\tilde{\bigcirc}^d}(u_s) \cup \mathcal{G}_{\tilde{\bigcirc}^u}(u_s) \cup \mathcal{G}_{\bigcirc^d_H}(u_s) \cup \mathcal{G}_{\tilde{\bigcirc}^d_H}(u_s)$
pop	$u_c \neq \bot$[1]	$\bigcup \begin{cases} \mathcal{G}_{\chi^d_F}(u_c) \cup \mathcal{G}_{\chi^u_F}(u_c) \\ \mathcal{G}_{\tilde{\chi}^d_F}(u_c) \cup \mathcal{G}_{\tilde{\chi}^u_F}(u_c) \\ \mathcal{G}_{\bigcirc^d_H}(u_c) \cup \mathcal{G}_{\tilde{\bigcirc}^d_H}(u_c) \\ \mathcal{G}_{\tilde{\bigcirc}^u_H}(\text{stack}(u_s)) \cup \mathcal{G}_{\bigcirc^u_H}(\text{stack}(u_s)) \end{cases}$

[1] u_s is the closest step ancestor of u and $u_c = \text{ctx}(u_s)$

formulas that can be efficiently handled by off-the-shelf solvers. Iterating over a growing index $k > 1$, at each step our procedure produces an SMT formula that encodes the branches of the tableau of length up to k step nodes, such that the formula is satisfiable if and only if an accepted branch of the tableau exists. If not, we increment k and proceed. In this respect, the procedure reminds of classic *bounded model checking* [7,14]. Here we summarize the working principles of the tableau encoding. The full details are available in [9].

The encoding produces formulas whose *models*, when they exist, represent single branches of the tableau. At a given step k, the formulas are interpreted over a restricted form of quantified[5] EUF, over two *finite, enumerated, ordered*[6] sorts: a sort $\mathcal{N}_k$, of exactly $k+1$ elements used to identify the nodes in the branch, and a sort called $\mathcal{S}$ that contains a finite set of symbols used in the encoding to represent the letters of the formula's alphabet. We suppose to have a finite number of constants for the values in $\mathcal{S}$. Among those, we have $p \in \mathcal{S}$ for each $p \in \Sigma \cup AP$. Others will be introduced when needed. We also exploit a fixed arbitrary ordering between elements of $\mathcal{N}_k$, and we abuse notation by denoting the constants for sort $\mathcal{N}_k$ as $0, 1, \ldots, k$, and writing $x+1$ and $x-1$ for an element $x \in \mathcal{N}_k$ to denote its predecessor and successor in this order.

For each proposition $p \in \Sigma \cup AP$, the encoding uses a binary predicate $\Gamma(p, x)$ whose first argument ranges among $\mathcal{S}$ and the second among $\mathcal{N}_k$. The intuitive

[5] Thanks to finite sorts, quantifiers are in fact expanded to disjunctions/conjunctions.
[6] The sort returned by the `Z3_mk_finite_domain_sort()` function of the Z3 C API.

meaning of $\Gamma(p, x)$ is that $p \in \Gamma(u)$ if u is the x-th step node of the current branch of the tableau. The encoding also uses some function symbols. A unary predicate $\bar{\Sigma}$ ranging over $\mathcal{S}$ tells which symbols from $\mathcal{S}$ are structural symbols. A function $\mathrm{smb}(x) : \mathcal{N}_k \to \mathcal{S}$ is used to represent the $\mathrm{smb}(u_x)$ symbol. A function symbol $\mathrm{struct}(x) : \mathcal{N}_k \to \mathcal{S}$ represents $\Gamma(u_x) \cap \Sigma$. Two functions $\mathrm{stack}(x) : \mathcal{N}_k \to \mathcal{N}_k$ and $\mathrm{ctx}(x) : \mathcal{N}_k \to \mathcal{N}_k$ represent the corresponding functions in the tableau. When $\mathrm{stack}(u) = \bot$, we denote it as $\mathrm{stack}(x) = 0$, and similarly for $\mathrm{ctx}(x)$.

For any strong or weak *next* or *chain next* temporal formula in the closure of ϕ we also introduce a corresponding *propositional* symbol in $\mathcal{S}$. Specifically, for each formula $\bigcirc^t \psi$, $\chi_F^t \psi$, $\tilde{\bigcirc}^t \psi$ and $\tilde{\chi}_F^t \psi$ in the closure, $\mathcal{S}$ contains the following propositional symbols, which we call *grounded*: $(\bigcirc^t \psi)_G$, $(\chi_F^t \psi)_G$, $(\tilde{\bigcirc}^t \psi)_G$, $(\tilde{\chi}_F^t \psi)_G$, and $(\bigcirc^t \psi)_G$, $(\chi_F^t \psi)_G$, $(\tilde{\bigcirc}^t \psi)_G$, $(\tilde{\chi}_F^t \psi)_G$.

The core building block of the encoding is the following *normal form* for $\mathsf{POTL_f}$ formulas.

Definition 8 (Next Normal Form). *Let ϕ be a $\mathsf{POTL_f}$ formula. The* next normal form *of ϕ, denoted $\mathrm{xnf}(\phi)$ is defined as follows:*

$$\mathrm{xnf}(p) = p \quad \textit{for } p \in \Sigma \qquad \mathrm{xnf}(\neg p) = \neg p \quad \textit{for } p \in \Sigma$$
$$\mathrm{xnf}(\tilde{\bigcirc}^t \psi) = \tilde{\bigcirc}^t \psi \qquad \mathrm{xnf}(\tilde{\chi}_F^t \psi) = \tilde{\chi}_F^t \psi$$
$$\mathrm{xnf}(\alpha \circ \beta) = \mathrm{xnf}(\alpha) \circ \mathrm{xnf}(\beta) \quad \textit{for } \circ \in \{\vee, \wedge\}$$
$$\mathrm{xnf}(\alpha \, \mathcal{U}_\chi^t \, \beta) = \mathrm{xnf}(\beta) \vee \big(\mathrm{xnf}(\alpha) \wedge (\bigcirc^t(\alpha \, \mathcal{U}_\chi^t \, \beta) \vee \chi_F^t(\alpha \, \mathcal{U}_\chi^t \, \beta))\big)$$
$$\mathrm{xnf}(\alpha \, \mathcal{R}_\chi^t \, \beta) = \mathrm{xnf}(\beta) \wedge \big(\mathrm{xnf}(\alpha) \vee (\tilde{\bigcirc}^t(\alpha \, \mathcal{R}_\chi^t \, \beta) \wedge \tilde{\chi}_F^t(\alpha \, \mathcal{R}_\chi^t \, \beta))\big)$$

Intuitively, $\mathrm{xnf}(\phi)$ encodes the *expansion rules* of the tableau (Table 1). Given ϕ and a fresh variable x of sort $\mathcal{N}_k$, we denote as $\mathrm{xnf}(\phi)_G$ the formula obtained from $\mathrm{xnf}(\phi)$ by replacing any proposition p with $\Gamma(p, x)$. Note that $\mathrm{xnf}(\phi)_G$ does not contain temporal operators: it is a first-order formula with a single free variable x.

We can now show the encoding itself. We start by constraining the meaning of the $\bar{\Sigma}$ predicate and the struct and smb functions. We define a formula ϕ_{axioms} that states that the $\bar{\Sigma}$ predicate identifies structural symbols and the $\mathrm{struct}(x)$ and $\mathrm{smb}(x)$ functions only return structural symbols, and we write a formula ϕ_{OPM} that explicitly models the $\lessdot$, $\doteq$ and $\gtrdot$ relations between symbols in $\mathcal{S}$ as binary predicates in the SMT encoding. The predicates range over the whole $\mathcal{S}$ but only the relationship between symbols in Σ will matter. With these in place, we can identify the *type* of each step node depending on the PR between $\mathrm{smb}(x)$ and $\mathrm{struct}(x)$. We encode this by the following three predicates:

$$\mathrm{push}(x) \equiv \mathrm{smb}(x) \lessdot \mathrm{struct}(x) \quad \mathrm{shift}(x) \equiv \mathrm{smb}(x) \doteq \mathrm{struct}(x)$$
$$\mathrm{pop}(x) \equiv \mathrm{smb}(x) \gtrdot \mathrm{struct}(x)$$

A formula ϕ_{init} encodes how the root node of the tableau looks like. In particular, it includes the conjunct $\mathrm{xnf}(\phi)_G(1)$, to say that its label contains ϕ.

We can now encode the step rules of Table 3. For space constraints we only show here the encoding of the step rules concerning *push* nodes (first two lines of Table 3). The encoding of such rules is the following:

$$\begin{aligned}\mathrm{step}_{\mathrm{push}}(x) \equiv \bigwedge_{\bigcirc^t \alpha \in \mathrm{Cl}(\phi)} \big(\Gamma((\bigcirc^t \alpha)_G, x) \rightarrow \mathrm{xnf}(\alpha)_G(x+1)\big) \\ \wedge\ \mathrm{smb}(x+1) = \mathrm{struct}(x) \wedge \mathrm{stack}(x+1) = x \\ \wedge\ (\mathrm{stack}(x) = 0 \rightarrow \mathrm{ctx}(x+1) = 0) \\ \wedge\ ((\mathrm{stack}(x) \neq 0 \wedge (\mathrm{push}(x-1) \vee \mathrm{shift}(x-1))) \rightarrow \mathrm{ctx}(x+1) = x-1) \\ \wedge\ ((\mathrm{stack}(x) \neq 0 \wedge \mathrm{pop}(x-1)) \rightarrow \mathrm{ctx}(x+1) = \mathrm{ctx}(x-1))\end{aligned}$$

We can similarly obtain two formulas $\mathrm{step}_{\mathrm{shift}}(x)$ and $\mathrm{step}_{\mathrm{pop}}(x)$. It is worth to note the first line of the above definition, where $\mathrm{xnf}(\alpha)$ is imposed to hold on $x+1$ if a next operator on α is present on x.

Next, we can encode the rejecting rules of Table 2. Since there are so many of them, we only show some examples (see [9] for the full list). What we actually encode is the *negation* of the rejecting rules, that describes what a node has to satisfy to *not* be rejected. We start to note that Rule 1 does not need to be encoded, since it just states that a proposition cannot hold together with its negation, which is trivially implied by the logic. Then, the simplest ones are Rules 2 and 3 of Table 2, and can be encoded as follows:

$$\begin{aligned}\mathrm{r}_2(x) &\equiv \forall p \forall q(\Sigma(p) \wedge \Sigma(q) \wedge \Gamma(p,x) \wedge \Gamma(q,x) \rightarrow p = q) \\ \mathrm{r}_3(x) &\equiv \Gamma(\#, x) \rightarrow \big(\bigwedge_{\bigcirc^t \alpha \in \mathrm{Cl}(\phi)} (\neg\Gamma((\bigcirc^t \alpha)_G, x)) \wedge \bigwedge_{p \in AP} (\neg\Gamma(p,x))\big)\end{aligned}$$

We similarly have a formula $r_i(x)$ encoding the negation of each block of lines from Rule 4 to 13. With these in place, we define a formula $[\![\phi]\!]_k$ called the *k-unraveling* of ϕ, that encodes all the non-rejected branches of the tableau of up to k step nodes.

$$\begin{aligned}&\phi_{\mathrm{axioms}} \wedge \phi_{OPM} \wedge \phi_{\mathrm{init}} \wedge \forall x \left(x > 1 \rightarrow \bigwedge\nolimits_{i=2}^{13} r_i(x)\right) \wedge \\ &\forall x \left[1 \leq x < k \rightarrow \begin{pmatrix} (\mathrm{push}(x) \rightarrow \mathrm{step}_{\mathrm{push}}(x)) \\ \wedge\ (\mathrm{shift}(x) \rightarrow \mathrm{step}_{\mathrm{shift}}(x)) \\ \wedge\ (\mathrm{pop}(x) \rightarrow \mathrm{step}_{\mathrm{pop}}(x)) \end{pmatrix}\right]\end{aligned}$$

The only *acceptance* rule of the tableau is encoded by a formula $e(x)$ defined as $e(x) \equiv \Gamma(\#, x) \wedge \mathrm{stack}(x) = 0$.

Finally, we have the following.

Theorem 2. *If $[\![\phi]\!]_k \wedge e(k)$ is satisfiable for some $k > 0$, then ϕ is satisfiable.*

We exploit this encoding of $\mathsf{POTL_f}$ satisfiability for model checking a formula ϕ through an algorithm that iterates on k starting from $k = 1$. First, we check

satisfiability of $[\![\neg\phi]\!]_k \wedge [\![\mathcal{M}]\!]_k$, where $[\![\mathcal{M}]\!]_k$ encodes a length-k prefix of a trace of the program $\mathcal{M}$ to be checked. We automatically translate programs to OPA whose transitions are labeled with program statements in the same way as [3,10], so that the automaton's stack simulates the program stack. Such *extended* OPA are then directly encoded into SMT in a straightforward manner, using the theories of fixed-size bit vectors and arrays to represent variables (cf. [9]). If this satisfiability check fails, it means no trace of $\mathcal{M}$ of length $\geq k$ violates ϕ, proving that $\mathcal{M}$ satisfies ϕ. Otherwise, we check whether $e(k)$ is satisfied when conjoined with the previous assertions. If it is, then we have found a counterexample trace that violates ϕ. Otherwise, we increase k by 1 and repeat. Since the tableau is finite, we eventually either find a counterexample, or hit a value of k such that Rule 13 rejects all branches, and the initial satisfiability check fails.

5 Experimental Evaluation

We implemented the encoding described in Section 4 in a SMT-based model checker that leverages the Z3 SMT solver [31]. We developed it within POMC [8], an explicit-state model checker for POTL developed by the authors of [10].

We compare our SMT-based approach with the explicit-state algorithm powering POMC, which performs the following steps on-the-fly: (i) it builds an OPA $\mathcal{A}_\varphi$ encoding the negation of the formula φ to be checked; (ii) it constructs the synchronized product between $\mathcal{A}_\varphi$ and the model of the system; (iii) it checks the nonemptiness of the product automaton, witnessing a counterexample to the property in the model, in a depth-first fashion.

We ran our experiments on server with a 2.0 GHz AMD CPU and RAM capped at 30 GiB.

5.1 Description of the benchmarks

We evaluate the two tools on a set of benchmarks adapted from [10], divided in three categories (Quicksort, Jensen, Stack). We modeled all benchmarks in MiniProc, the modeling language of the POMC tool. The checked formulas are reported in Table 5. Below, we give a brief description of each category.

Quicksort. We modeled a Java implementation of the Quicksort sorting algorithm. The algorithm is implemented as a recursive function qs, called by the main function in a `try-catch` block, and is applied to an array of integers that may contain null values, which cause a `NullPointerException`. We vary the length of the arrays from 1 to 5 elements and the width of the elements from 2 to 16 bits. Formulas 1 and 2 both check that the main function returns without exceptions, while 3 checks the same for the qs (QuickSort) function. Formulas 4 (resp., Formula 5) states that the array is sorted when the main function (resp., the qs function) returns without exceptions. Finally, Formula 6 states that either qs throws an exception or the array is sorted (and qs returns normally).

Table 5: Benchmark formulas. The last column states whether they are true (T) or false (F) in each model. $\Box$ is the LTL always, which we implemented as in [17].

QuickSort	1	$\chi^u_F(\mathbf{ret} \wedge \text{main})$	T
	2	$\mathbf{call} \wedge \text{main} \rightarrow \neg(\bigcirc^u \mathbf{exc} \vee \chi^u_F \mathbf{exc})$	T
	3	$\Box(\mathbf{call} \wedge \text{qs} \rightarrow \neg(\bigcirc^u \mathbf{exc} \vee \chi^u_F \mathbf{exc}))$	F
	4	$\chi^u_F \text{ sorted}$	F
	5	$\Box(\mathbf{call} \wedge \text{qs} \rightarrow \chi^u_F \text{ sorted})$	F
	6	$\chi^d_F(\mathbf{han} \wedge \bigcirc^d(\mathbf{call} \wedge \text{qs} \wedge \chi^u_F(\mathbf{exc} \vee \text{sorted})))$	T
Jensen	8	$\Box(\mathbf{call} \wedge \neg P_{\text{cp}} \rightarrow \neg(\top \mathcal{U}^d_\chi (\mathbf{call} \wedge \text{read})))$	T
	9	$\Box(\mathbf{call} \wedge \neg P_{\text{db}} \rightarrow \neg(\top \mathcal{U}^d_\chi (\mathbf{call} \wedge \text{write})))$	T
	10	$\Box(\mathbf{call} \wedge ((\text{canpay} \wedge \neg P_{\text{cp}}) \vee (\text{debit} \wedge \neg P_{\text{db}})) \rightarrow \bigcirc^u \mathbf{exc} \vee \chi^u_F \mathbf{exc})$	T
	11	$\neg(\top \mathcal{U}^d_\chi (\text{balance} < 0))$	T
Stack	12	$\Box(\text{modified} \rightarrow \neg(\bigcirc^u \mathbf{exc} \vee \chi^u_F \mathbf{exc}))$	T/F
	13	$\Box(\mathbf{call} \wedge (\text{push} \vee \text{pop}) \rightarrow \neg(\top \mathcal{U}^d_H \text{ modified}))$	T/F
	14	$\Box(\mathbf{call} \wedge (\text{push} \vee \text{pop}) \wedge \chi^d_F \mathbf{ret} \rightarrow \neg(\top \mathcal{U}^d_\chi (\mathbf{han} \wedge \text{Stack} \wedge (\neg \mathbf{han} \mathcal{U}^d_\chi (\text{T} \wedge \bigcirc^u \mathbf{exc})))))$	T/T

Bank Account. This category consists of a simple banking application taken from [24] which allows users to withdraw money or check their balance. The variable representing the balance is protected by a Java AccessController, which prevents unauthorized users from accessing it by raising exceptions. We modeled the balance with an integer variable. Formula 8 (resp., Formula 9) checks that, whenever a function is called without having permission to check the balance (resp., to make a payment), then there is no read-access (resp., write access) to the variable holding the balance. The permission of checking the balance and to make a payment are modeled by the variables P_{cp} and P_{db}, respectively. Formula 10 checks that if the functions that check the balance (canpay) and make a payment (debit) are called without permission, an exception is thrown. Formula 11 checks that the balance never becomes negative, because payments are only made if the account has enough money.

Stack. We model two C++ implementations of a generic stack data structure taken from [33], where constructors of contained elements may throw exceptions. Only one of the two implementations is exception safe. The `pop` method of the safe implementation does not return the popped element, which must be accessed through the `top` method, and it performs other operations on a new copy of the internal data structure, to prevent exceptions from leaving it in an inconsistent state. In contrast with [10] which uses a manually-crafted abstraction for the elements in the stack, our model implements the stack with actual arrays of fixed-width integers. Formulas 12 and 13 check *strong exception safety* [1], *i.e.*, that each operation on the data structure is rolled back if any functions related

to the element type T throw an exception, leaving the stack in a consistent state. Formula 14 checks *exception neutrality* [1], which means that exceptions thrown by element functions are always propagated by the stack's methods.

5.2 Description of the plots

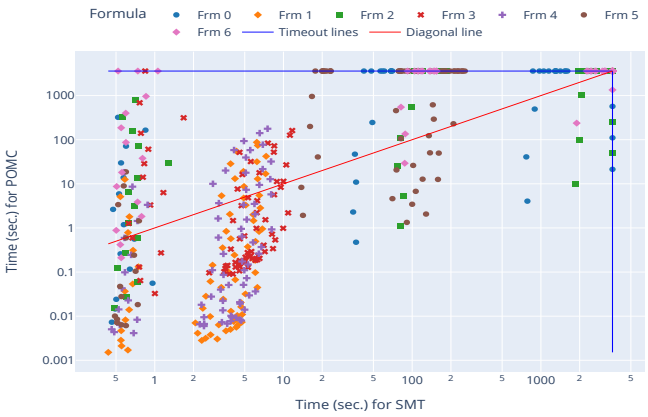
We compare the time (measured in seconds) taken by the SMT-based approach (in the plots referred to as SMT) with the time taken by POMC, dividing the plots by the three categories of benchmarks (Quicksort, Jensen, and Stack). For each category, we show a scatter plot (Fig. 3) and a survival plot (Fig. 4).

We first look at the scatter plots in Fig. 3. The x-axis refers to the solving time for the SMT-based approach while the y-axis to the solving time for POMC, both measured in seconds. The blue border lines indicate the timeout (set to 3600 seconds) for the tools, while the red line denotes the diagonal of the plot.

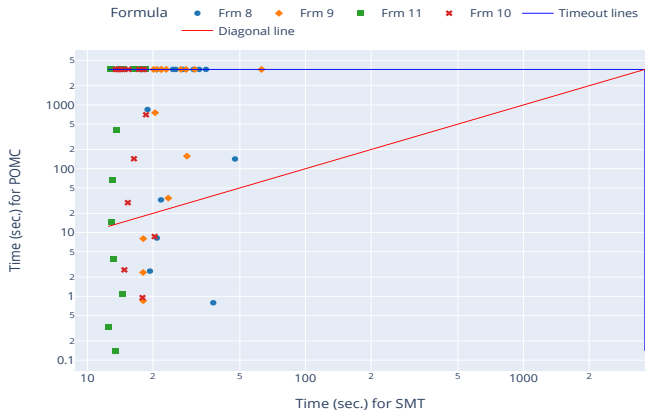
For all three categories of benchmarks, the scatter plots reveal an exponential blow up for the solving time of the POMC tool; on the contrary, the SMT-based approach does not incur in such a blow up. As an example, we take the scatter plot for the Quicksort category in Fig. 3 (a) and we consider the brown circles in the middle of the plot, corresponding to the Formula 5 of Table 5 checked on an array of size 2 containing numbers of increasing bitvector-size. For the case of numbers of bitvector-size of 3, 4, 5, and 6 bits, the solving time of POMC is of 8, 40, 199, and 956 seconds, respectively, while the time required by the SMT-based approach is of 13, 18, 16 and 16 seconds, respectively. Moreover, while for bitvector-size greater than 6 bits POMC reaches always the timeout for Formula 5, the SMT-based approach solves the benchmarks of all bitvector-size (*i.e.*, up to 16 bits) in time always less than 23 seconds.

A similar consideration can be done for the Jensen and the Stack categories. Take, for example, the blue circles in Fig. 3 (c) corresponding to Formula 14 in Table 5. For this case, the solving times of the SMT-based approach are consistently better than the ones of POMC. The reason may be that this formula contains hierarchical operators, which tend to yield to automata that make more non-deterministic guesses. This, in turn, causes the explicit-state model checker to perform, in general, many steps of backtracking during its depth-first model checking algorithm. Conversely, in the SMT-based approach, this part is managed (efficiently) by the DPLL algorithm inside the SMT-solver.

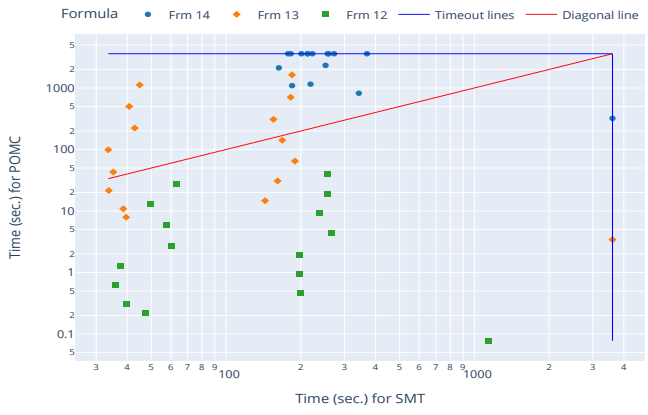
The exponential trend of POMC is reflected also in the survival plot (Fig. 4). Here, the x-axis represents the time (in seconds) while the y-axis represents the percentage of solved benchmarks. From the blue and yellow lines in Fig. 4, which correspond to the categories Stack and Jensen, respectively, it is clear that the POMC tool gets stuck solving (approximately) the 80% and the 60% of the benchmarks in the corresponding category. Conversely, the SMT-based approach solves all benchmarks in these two categories. If we take a look to the survival plot only for the Quicksort category in Fig. 5 (which reports the absolute number of solved benchmarks), we observe that the POMC tool gets stuck solving (approximately) 330 benchmarks, while the SMT-based approach solves circa 430 benchmarks.



(a) Scatter plot for category Quicksort.



(b) Scatter plot for category Jensen.



(c) Scatter plot for category Stack.

Fig. 3: Scatter plots

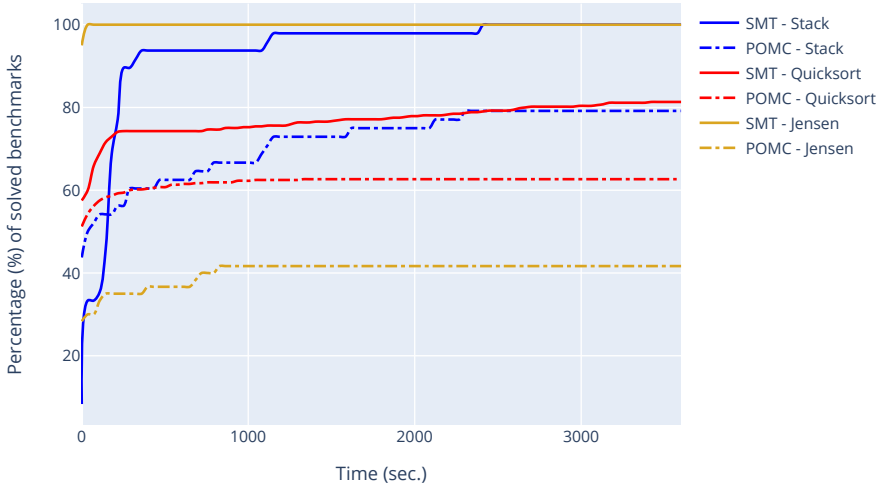


Fig. 4: Survival plot

In our benchmarks, we found only one case in which the solving time of POMC is always better than the one of the SMT-based approach. It corresponds to the green squares on the scatter plots in Fig. 3 (c) for the Stack category, corresponding to Formula 12. The reason is that this formula requires very few nondeterministic transitions in the explicit-state automaton. This, in turn, makes the search of the state-space a (almost) deterministic step, and thus very efficient for the depth-first algorithm of POMC. On the contrary, the breadth-first algorithm of the SMT-based approach seems to perform worse.

6 Conclusions

We have introduced a tree-shaped tableau for the future fragment of the temporal logic POTL on finite-word semantics, and encoded it in SMT to perform symbolic model checking of procedural programs. This is the first time both of these techniques have been used for checking a temporal logic with context-free modalities. The experimental evaluation shows that our symbolic approach scales better than the state-of-the-art explicit-state one.

Extending the tableau to past POTL operators and to infinite words seems a promising future direction, which should be achievable through an approach similar to related work on the tree-shaped tableau for LTL [18].

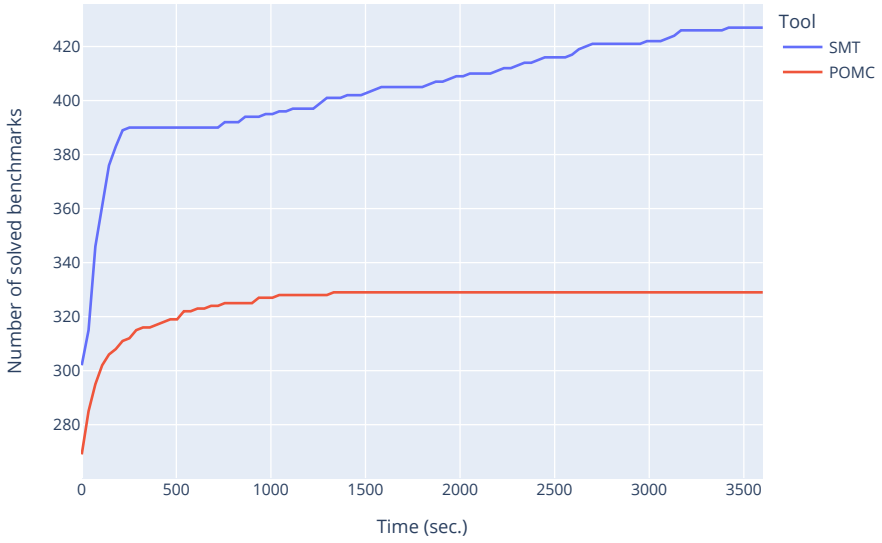


Fig. 5: Survival plot for the Quicksort category.

Funded by the European Union

Acknowledgments. This work was partially funded by the Vienna Science and Technology Fund (WWTF) grant [10.47379/ICT19018] (ProbInG), and by the EU Commission in the Horizon Europe research and innovation programme under grant agreement No. 101107303 (MSCA Postdoctoral Fellowship CORPORA).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abrahams, D.: Exception-safety in generic components. In: Generic Programming. LNCS, vol. 1766, pp. 69–79. Springer, Berlin, Heidelberg (1998). https://doi.org/10.1007/3-540-39953-4_6
2. Alur, R., Arenas, M., Barceló, P., Etessami, K., Immerman, N., Libkin, L.: First-order and temporal logics for nested words. LMCS **4**(4) (2008). https://doi.org/10.2168/LMCS-4(4:11)2008
3. Alur, R., Bouajjani, A., Esparza, J.: Model checking procedural programs. In: Handbook of Model Checking, pp. 541–572. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_17
4. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: TACAS '04. LNCS, vol. 2988, pp. 467–481. Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_35

5. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: STOC '04. pp. 202–211. ACM (2004). https://doi.org/10.1145/1007352.1007390
6. Alur, R., Madhusudan, P.: Adding nesting structure to words. J. ACM **56**(3), 16:1–16:43 (2009). https://doi.org/10.1145/1516512.1516518
7. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003). https://doi.org/10.1016/S0065-2458(03)58003-2, https://doi.org/10.1016/S0065-2458(03)58003-2
8. Chiari, M., Bergamaschi, D., Pontiggia, F.: POMC (2024), https://github.com/michiari/POMC
9. Chiari, M., Geatti, L., Gigante, N., Pradella, M.: SMT-based symbolic model-checking for operator precedence languages. CoRR **abs/2405.11327** (2024), https://arxiv.org/abs/2405.11327
10. Chiari, M., Mandrioli, D., Pontiggia, F., Pradella, M.: A model checker for operator precedence languages. ACM Trans. Program. Lang. Syst. **45**(3), 19:1–19:66 (2023). https://doi.org/10.1145/3608443
11. Chiari, M., Mandrioli, D., Pradella, M.: Operator precedence temporal logic and model checking. Theor. Comput. Sci. **848**, 47–81 (2020). https://doi.org/10.1016/j.tcs.2020.08.034
12. Chiari, M., Mandrioli, D., Pradella, M.: Model-checking structured context-free languages. In: CAV'21. LNCS, vol. 12760, p. 387–410. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_18
13. Chiari, M., Mandrioli, D., Pradella, M.: A first-order complete temporal logic for structured context-free languages. Log. Methods Comput. Sci. **18:3** (2022). https://doi.org/10.46298/LMCS-18(3:11)2022
14. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods Syst. Des. **19**(1), 7–34 (2001). https://doi.org/10.1023/A:1011276507260
15. Crespi Reghizzi, S., Mandrioli, D.: Operator precedence and the visibly pushdown property. J. Comput. Syst. Sci. **78**(6), 1837–1867 (2012). https://doi.org/10.1016/j.jcss.2011.12.006
16. Floyd, R.W.: Syntactic analysis and operator precedence. J. ACM **10**(3), 316–333 (1963). https://doi.org/10.1145/321172.321179
17. Geatti, L., Gigante, N., Montanari, A.: A SAT-based encoding of the one-pass and tree-shaped tableau system for LTL. In: TABLEAUX'19. LNCS, vol. 11714, pp. 3–20. Springer (2019). https://doi.org/10.1007/978-3-030-29026-9_1
18. Geatti, L., Gigante, N., Montanari, A., Reynolds, M.: One-pass and tree-shaped tableau systems for TPTL and $\mathrm{TPTL_b}$+Past. Inf. Comput. **278**, 104599 (2021). https://doi.org/10.1016/j.ic.2020.104599
19. Geatti, L., Gigante, N., Montanari, A., Venturato, G.: SAT meets tableaux for linear temporal logic satisfiability. J. Autom. Reason. **68**(2), 6 (2024). https://doi.org/10.1007/S10817-023-09691-1
20. Grune, D., Jacobs, C.J.: Parsing techniques: a practical guide. Springer, New York (2008). https://doi.org/10.1007/978-0-387-68954-8
21. Harrison, M.A.: Introduction to Formal Language Theory. Addison Wesley, Boston, MA, USA (1978)
22. Henzinger, T.A., Kebis, P., Mazzocchi, N., Saraç, N.E.: Regular methods for operator precedence languages. In: ICALP'23. LIPIcs, vol. 261, pp. 129:1–129:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.ICALP.2023.129

23. Huang, G., Wang, B.: Complete SAT-based model checking for context-free processes. Int. J. Found. Comput. Sci. **21**(2), 115–134 (2010). https://doi.org/10.1142/S0129054110007179
24. Jensen, T.P., Le Métayer, D., Thorn, T.: Verification of control flow based security properties. In: Proc. '99 IEEE Symp. Secur. Privacy. pp. 89–103. IEEE Computer Society, Oakland, California, USA (1999). https://doi.org/10.1109/SECPRI.1999.766902
25. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods Syst. Des. **48**(3), 175–205 (2016). https://doi.org/10.1007/S10703-016-0249-4
26. Lonati, V., Mandrioli, D., Panella, F., Pradella, M.: Operator precedence languages: Their automata-theoretic and logic characterization. SIAM J. Comput. **44**(4), 1026–1088 (2015). https://doi.org/10.1137/140978818
27. Mandrioli, D., Pradella, M.: Generalizing input-driven languages: Theoretical and practical benefits. Computer Science Review **27**, 61–87 (2018). https://doi.org/10.1016/j.cosrev.2017.12.001
28. Mandrioli, D., Pradella, M., Crespi Reghizzi, S.: Star-freeness, first-order definability and aperiodicity of structured context-free languages. In: ICTAC '20. vol. 12545, pp. 161–180. Springer (2020). https://doi.org/10.1007/978-3-030-64276-1_9
29. Mandrioli, D., Pradella, M., Crespi Reghizzi, S.: Aperiodicity, star-freeness, and first-order logic definability of operator precedence languages. Log. Methods Comput. Sci. **19:4** (2023). https://doi.org/10.46298/lmcs-19(4:12)2023
30. Mehlhorn, K.: Pebbling mountain ranges and its application of DCFL-recognition. In: ICALP '80. LNCS, vol. 85, pp. 422–435 (1980). https://doi.org/10.1007/3-540-10003-2_89
31. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: TACAS'08. LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
32. Pontiggia, F., Chiari, M., Pradella, M.: Verification of programs with exceptions through operator precedence automata. In: SEFM'21. LNCS, vol. 13085, pp. 293–311. Springer, Berlin, Heidelberg (2021). https://doi.org/10.1007/978-3-030-92124-8_17
33. Sutter, H.: Exception-safe generic containers. C++ Report **9** (1997), https://ptgmedia.pearsoncmg.com/imprint_downloads/informit/aw/meyerscddemo/DEMO/MAGAZINE/SU_FRAME.HTM