Model-Checking Structured Context-Free Languages

Michele Chiari[1][0000−0001−7742−9233], Dino Mandrioli[1][0000−0002−0945−5947], and Matteo Pradella[1,2][0000−0003−3039−1084]

[1] DEIB, Politecnico di Milano, Italy, name.surname@polimi.it
[2] IEIIT, Consiglio Nazionale delle Ricerche, Italy

Abstract. The problem of model checking procedural programs has fostered much research towards the definition of temporal logics for reasoning on context-free structures. The most notable of such results are temporal logics on Nested Words, such as CaRet and NWTL. Recently, the logic OPTL was introduced, based on the class of Operator Precedence Languages (OPL), more powerful than Nested Words. We define the new OPL-based logic POTL, and provide a model checking procedure for it. POTL improves on NWTL by enabling the formulation of requirements involving pre/post-conditions, stack inspection, and others in the presence of exception-like constructs. It improves on OPTL by being FO-complete, and by expressing more easily stack inspection and function-local properties. We developed a model checking tool for POTL, which we experimentally evaluate on some interesting use-cases.

Keywords: Linear Temporal Logic · Operator Precedence Languages · Model Checking · Visibly Pushdown Languages · Input-Driven Languages

1 Introduction

Model checking is one of the most successful techniques for the verification of software programs. It consists in the exhaustive verification of the mathematical model of a program against a specification of its desired behavior. The kind of properties that can be proved in this way depends both on the formalism employed to model the program, and on the one used to express the specification. The initial and most classical frameworks consist in the use of operational formalisms, such as Transition Systems and Finite State Automata (generally Büchi automata) for the model, and temporal logics such as Linear-time Temporal Logic (LTL), Computation-Tree Logic (CTL) and CTL* for the specification [24]. The success of such logics is due to their ease in reasoning about linear or branching sequences of events over time, by expressing liveness and safety properties, their conciseness with respect to automata, and the complexity of their model checking.

In this paper we consider linear-time temporal domains. LTL limits its set of expressible properties to the First-Order Logic (FOL) definable fragment of regular languages. This is quite restrictive when compared with the most popular

abstract models of procedural programs, such as Pushdown Systems, Boolean Programs [10], and Recursive State Machines [3]. All such stack-based formalisms show behaviors that are expressible by means of Context-Free Languages (CFL), rather than regular ones. State and configuration reachability, fair computation problems, and model checking of *regular specifications* have been thoroughly studied for such formalisms [13,30,17,28,40,51,55,3,32,4]. To expand the expressive power of specification languages too, [12,14] augmented LTL with Presburger arithmetic constraints on the occurrences of states, obtaining a logic capable of even some context-sensitive specifications, but with only restricted decidable fragments. [41] introduced model checking of pushdown tree automata specifications on regular systems, and Dynamic Logic was extended to some limited classes of CFL [34]. Decision procedures for different kinds of regular constraints on stack contents have been given in [37,29,18].

A coherent approach came with the introduction of temporal logics based on Visibly Pushdown Languages (VPL) [7], a.k.a. Input-Driven Languages [47]. Such logics, namely CaRet [6] and its FO-complete successor NWTL [2], model the execution trace of a procedural program as a Nested Word [8], consisting in a linear ordering augmented with a one-to-one matching relation between function calls and returns. They are the first ones featuring temporal modalities that explicitly refer to the nesting structure of CFL [4]. This enables requirement specifications to include Hoare-style pre/post-conditions, stack-inspection properties, and more. A μ-calculus based on VPL extends model checking to branching-time semantics in [5], while [16] introduces a temporal logic capturing the whole class of VPL. Timed extensions of CaRet are given in [15].

VPL too have their limitations. They are more general than Parenthesis Languages [46], but their *matching relation* is essentially constrained to be one-to-one [43]. This hinders their suitability to model processes in which a single event must be put in relation with multiple ones. Unfortunately, computer programs often present such behaviors: exceptions and continuations are single events that cause the termination (or re-instantiation) of multiple functions on the stack.

To reason about such behaviors, temporal logics based on Operator Precedence Languages (OPL) have been proposed [22]. OPL were initially introduced with the purpose of efficient parsing [31], a field in which they continue to offer useful applications [11]. They are capable of capturing the syntax of arithmetic expressions, and other constructs whose context-free structure is not immediately visible. The generality of the structure of their syntax trees is much greater than that of VPL, which are strictly included in OPL [25]. Nevertheless, they retain the same closure properties that make regular languages and VPL suitable for automata-theoretic model checking: OPL are closed under Boolean operations, concatenation, Kleene *, and language emptiness and inclusion are decidable [42]. They have been characterized by means of push-down automata, Monadic Second-Order Logic and, recently, by an extension of Regular Expressions [42,44].

OPTL [22] is the first linear-time temporal logic for which a model checking procedure has been given on both finite and ω-words of OPL. It enables reasoning on procedural programs with exceptions, expressing properties about whether a

function can be terminated by an exception, or throw one, and also pre/post-conditions. NWTL can be translated into OPTL in linear time, thus the latter is capable of expressing all properties that can be formalized in CaRet and NWTL, and many more. [22] does not explore OPTL's expressiveness further, and does not investigate the practical applicability of their model checking construction.

In this article, we introduce Precedence Oriented Temporal Logic (POTL), which redefines the syntax and semantics of OPTL to be much closer to the context-free structure of words. With POTL, it is much easier to navigate a word's syntax tree, expressing requirements that are aware of its structure. From a more theoretical point of view, POTL is FO-complete whereas OPTL is not, so that CaRet, NWTL, OPTL and POTL constitute a strict hierarchy in terms of expressive power. Such a theoretical elaboration, however, is technically involved; thus, for length reasons, it is documented in a technical report [23].

In this paper, instead, we focus on the model-checking application of POTL. We provide a tableaux-construction procedure for model checking POTL, which yields nondeterministic automata of size at most singly exponential in the formula's length, and is thus not asymptotically greater than that of LTL, NWTL and OPTL. We implemented such a procedure in a tool called POMC, which we evaluate on several interesting case studies. POMC's performance is promising: almost all case studies are verified in seconds and with a reasonable memory consumption, with very few outliers. Such outliers are inevitable, due to the exponential complexity of the task.

The related work on tools is not as rich as the theoretical one. Tools and libraries such as VPAlib [48], VPAchecker [54], OpenNWA [27] and SymbolicAutomata [26] only implement operations such as union, intersection, universality/inclusion/emptiness check for Visibly Pushdown or Nested Word Automata, but have no model checking capabilities. PAL [19] uses nested-word based monitors to express program specifications, and a tool based on BLAST [36] implements its runtime monitoring and model checking. PAL follows the paradigm of program monitors, and is not—strictly speaking—a temporal logic. PTCaRet [52] is a past version of CaRet, and its runtime monitoring has been implemented in JavaMOP [20]. [49,50] describe a tool for model checking programs against CaRet specifications. Since its purpose is malware detection, it targets program binaries directly by modeling them as Pushdown Systems. Unfortunately, this tool does not seem to be available online. To the best of our knowledge, POMC is the only publicly-available[3] tool for model-checking temporal logics capable of expressing context-free properties.

The paper is organized as follows: we give some background on OPL in Section 2, we introduce POTL in Section 3 and its model checking in Section 4, and we evaluate our prototype model checker in Section 5. Due to space constraints, we leave all formal proofs to a technical report [21].

[3] https://github.com/michiari/POMC

2 Operator Precedence Languages

We assume some familiarity with classical formal language theory concepts such as context-free grammar, parsing, shift-reduce algorithm, syntax tree (ST) [33,35]. Operator Precedence Languages (OPL) are usually defined through their generating grammars [31]; in this paper, however, we characterize them through their accepting automata [42] which are the natural way for stating equivalence properties with logic characterization, and for model checking. Readers not familiar with OPL may refer to [43] for more explanations on the following basic concepts; an explanatory example is also given at the end of this section.

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" $\lessdot$. Despite their graphical appearance, PR are not ordering relations.

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 (disjoint from* Σ*),* $I \subseteq Q$ *is the set of initial states,* $F \subseteq Q$ *is the set of final states,* $\delta \subseteq Q \times (\Sigma \cup Q) \times Q$ *is the transition relation, which is the union of the three disjoint 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—possibly partial—functions.*

To define the semantics of OPA, we need some new notations. 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 $(q, p, r) \in \delta_{pop}$.

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

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

Fig. 1. OPM $M_{\mathbf{call}}$ (left) and a string with chains shown by brackets (right).

Shift and pop moves are not performed when the stack contains only $\perp$. 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 $L(\mathcal{A}) = \{x \in \Sigma^* \mid \langle x\#,\ q_I,\ \perp\rangle \vdash^* \langle \#,\ q_F,\ \perp\rangle, q_I \in I, q_F \in F\}$.

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 ${}^{c_0}[c_1c_2\dots c_\ell]^{c_{\ell+1}}$ *is a string* $c_0c_1c_2\dots c_\ell c_{\ell+1}$*, such that:* $c_0, c_{\ell+1} \in \Sigma \cup \{\#\}$*,* $c_i \in \Sigma$ *for every* $i = 1, 2, \dots \ell$ *(*$\ell \geq 1$*), and* $c_0 \lessdot c_1 \doteq c_2 \dots c_{\ell-1} \doteq c_\ell \gtrdot c_{\ell+1}$*. A* composed chain *is a string* $c_0s_0c_1s_1c_2\dots c_\ell s_\ell c_{\ell+1}$*, where* ${}^{c_0}[c_1c_2\dots c_\ell]^{c_{\ell+1}}$ *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* $i = 0, 1, \dots, \ell$ *(*$\ell \geq 1$*). Such a composed chain will be written as* ${}^{c_0}[s_0c_1s_1c_2\dots c_\ell s_\ell]^{c_{\ell+1}}$*.* c_0 *(resp.* $c_{\ell+1}$*) is called its* left *(resp.* right*)* context*; all symbols between them form its* body*.*

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 $w_{ex} =$ **call han call call exc call ret ret**, which is compatible with $M_{\mathbf{call}}$ (for a more complete treatment, cf. [43,33]). First, write all precedence relations between consecutive characters, according to $M_{\mathbf{call}}$. Then, recognize all innermost patterns of the form $a \lessdot c \doteq \dots \doteq c \gtrdot b$ 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} as follows:

1 $\#$ $\lessdot$ **call** $\lessdot$ **han** $\lessdot$ **call** $\lessdot$ **<u>call</u>** $\gtrdot$ **exc** $\gtrdot$ **call** $\doteq$ **ret** $\gtrdot$ **ret** $\gtrdot$ $\#$
2 $\#$ $\lessdot$ **call** $\lessdot$ **han** $\lessdot$ **<u>call</u>** $\gtrdot$ **exc** $\gtrdot$ **call** $\doteq$ **ret** $\gtrdot$ **ret** $\gtrdot$ $\#$
3 $\#$ $\lessdot$ **call** $\lessdot$ **<u>han</u>** $\doteq$ **<u>exc</u>** $\gtrdot$ **call** $\doteq$ **ret** $\gtrdot$ **ret** $\gtrdot$ $\#$
4 $\#$ $\lessdot$ **call** $\lessdot$ **<u>call</u>** $\doteq$ **<u>ret</u>** $\gtrdot$ **ret** $\gtrdot$ $\#$
5 $\#$ $\lessdot$ **<u>call</u>** $\doteq$ **<u>ret</u>** $\gtrdot$ $\#$
6 $\#$ $\doteq$ $\#$

The chain body removed in each step is underlined. In step 1, $^{\mathbf{call}}[\underline{\mathbf{call}}]^{\mathbf{exc}}$ is a simple chain, so its body $\underline{\mathbf{call}}$ is removed. Then, in step 2 we recognize the simple chain $^{\mathbf{han}}[\underline{\mathbf{call}}]^{\mathbf{exc}}$, which means $^{\mathbf{han}}[\mathbf{call}[\mathbf{call}]]^{\mathbf{exc}}$, where [**call**] is the chain body removed in step 1, is a composed chain. This way, we recognize, e.g., $^{\mathbf{han}}[\mathbf{call}]^{\mathbf{exc}}$, $^{\mathbf{call}}[\mathbf{han}\,\mathbf{exc}]^{\mathbf{call}}$ as simple chains, and $^{\mathbf{han}}[\mathbf{call}[\mathbf{call}]]^{\mathbf{exc}}$ and $^{\mathbf{call}}[\mathbf{han}[\mathbf{call}[\mathbf{call}]]\mathbf{exc}]^{\mathbf{call}}$ as composed chains (with inner chain bodies enclosed in brackets). Fig. 1 shows the structure of a longer version of w_{ex}, which is an isomorphic representation of its ST as depicted in Fig. 4. Each chain corresponds to an internal node, and the fringe of the subtree rooted at it is the chain's body.

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 $q_0 \xrightarrow{c_1} q_1 \dashrightarrow \dots \dashrightarrow q_{\ell-1} \overset{c_\ell}{\dashrightarrow} q_\ell \overset{q_0}{\Longrightarrow} q_{\ell+1}$. 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 $q_0 \overset{s_0}{\rightsquigarrow} q'_0 \xrightarrow{c_1} q_1 \overset{s_1}{\rightsquigarrow} q'_1 \overset{c_2}{\dashrightarrow} \dots \overset{c_\ell}{\dashrightarrow} q_\ell \overset{s_\ell}{\rightsquigarrow} q'_\ell \overset{q'_0}{\Longrightarrow} q_{\ell+1}$ 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.

Consider the OPA $\mathcal{A}(\Sigma, M) = (\Sigma, M, \{q\}, \{q\}, \{q\}, \delta_{max})$ where $\delta_{max}(q, q) = q$, and $\delta_{max}(q, c) = q$, $\forall c \in \Sigma$. We call it the *OP Max-Automaton* over Σ, M. For a max-automaton, each chain has a support. Since there is a chain $^{\#}[s]^{\#}$ for any string s compatible with M, a string is accepted by $\mathcal{A}(\Sigma, M)$ iff it is compatible with M. If M is complete, each string is accepted by $\mathcal{A}(\Sigma, M)$, which defines the universal language Σ^* by assigning to any string the (unique) structure compatible with the OPM. With $M_{\mathbf{call}}$ of Fig. 1, if we take e.g. the string **ret call han**, it is accepted by the max-automaton with structure $\#[[\mathbf{ret}]\mathbf{call}[\mathbf{han}]]\#$.

In conclusion, 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. For a more complete description of the OPL family and of its relations with other CFL we refer the reader to [43].

2.1 Operator Precedence ω-Languages

All definitions regarding OPL are extended to infinite words in the usual way, but with a few distinctions. Given an OP alphabet (Σ, M), an ω-word $w \in \Sigma^\omega$ is compatible with M if every prefix of w is compatible with M. OP ω-words are not terminated by the delimiter #. An ω-word may contain never-ending chains of the form $c_0 \lessdot c_1 \doteq c_2 \doteq \cdots$, where the $\lessdot$ relation between c_0 and c_1 is never closed by a corresponding $\gtrdot$. Such chains are called *open chains* and

may be simple or composed. A composed open chain may contain both open and closed subchains. Of course, a closed chain cannot contain an open one. A terminal symbol $a \in \Sigma$ is *pending* if it is part of the body of an open chain and of no closed chains.

OPA classes accepting the whole class of ωOPL can be defined by augmenting Definition 1 with Büchi or Muller acceptance conditions [42]. In this paper, we only consider the former. The semantics of configurations, moves and infinite runs are defined as for finite OPA. For the acceptance condition, let ρ be a run on an ω-word w. Define

$$\mathrm{Inf}(\rho) = \{q \in Q \mid \text{there exist infinitely many positions } i \text{ s.t. } \langle \beta_i, q, x_i \rangle \in \rho\}$$

as the set of states that occur infinitely often in ρ. ρ is successful iff there exists a state $q_f \in F$ such that $q_f \in \mathrm{Inf}(\rho)$. An ωOPBA $\mathcal{A}$ accepts $w \in \Sigma^\omega$ iff there is a successful run of $\mathcal{A}$ on w. The ω-language recognized by $\mathcal{A}$ is $L(\mathcal{A}) = \{w \in \Sigma^\omega \mid \mathcal{A} \text{ accepts } w\}$. Unlike OPA, ωOPBA do not require the stack to be empty for word acceptance: when reading an open chain, the stack symbol pushed when the first character of the body of its underlying simple chain is read remains into the stack forever; it is at most updated by shift moves.

The most important closure properties of OPL are preserved by ωOPL, which form a Boolean algebra and are closed under concatenation of an OPL with an ωOPL [42]. The equivalence between deterministic and nondeterministic automata is lost in the infinite case, which is unsurprising, since it also happens for regular ω-languages and ωVPL.

2.2 Modeling programs with OPA

For readers not familiar with OPL, we show how OPA can naturally model programming languages such as Java and C++. Given a set AP of atomic propositions describing events and states of the program, we use $(\mathcal{P}(AP), M_{AP})$ as the OP alphabet. For convenience, we consider a partitioning of AP into a set of standard propositional labels (in round font), and *structural labels* (SL, in bold). SL define the OP structure of the word: M_{AP} is only defined for subsets of AP containing exactly one SL, so that given two SL $\mathbf{l}_1, \mathbf{l}_2$, for any $a, a', b, b' \in \mathcal{P}(AP)$ s.t. $\mathbf{l}_1 \in a, a'$ and $\mathbf{l}_2 \in b, b'$ we have $M_{AP}(a, b) = M_{AP}(a', b')$. Hence, we define an OPM on the entire $\mathcal{P}(AP)$ by only giving the relations between SL, as we did for $M_{\mathbf{call}}$. Fig. 2 shows how to model a procedural program with an OPA. The OPA simulates the program's behavior with respect to the stack, by expressing its execution traces with four event kinds: **call** (resp. **ret**) marks a procedure call (resp. return), **han** the installation of an exception handler by a `try` statement, and **exc** an exception being raised. OPM $M_{\mathbf{call}}$ defines the context-free structure of the word, which is strictly linked with the programming language semantics: the $\lessdot$ PR causes nesting (e.g., **call**s can be nested into other **call**s), and the $\doteq$ PR implies a one-to-one relation, e.g. between a **call** and the **ret** of the same function, and a **han** and the **exc** it catches. Each OPA state represents a line in the source code. First, procedure p_A is called by the program loader (M0),

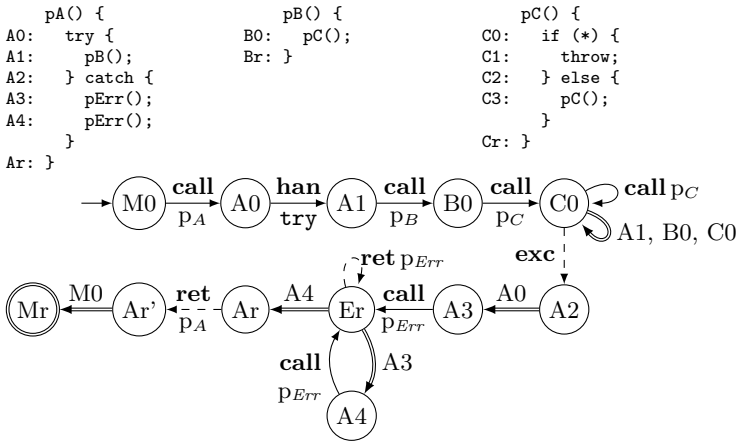

Fig. 2. Example procedural program (top) and the derived OPA (bottom). '$*$' implies a non-deterministic choice. Push, shift, pop moves are shown by, resp., solid, dashed and double arrows.

and $[\{\mathbf{call}, \mathrm{p}_A\}, \mathrm{M0}]$ is pushed onto the stack, to track the program state before the **call**. Then, the `try` statement at line `A0` of p_A installs a handler. All subsequent calls to p_B and p_C push new stack symbols on top of the one pushed with **han**. p_C may only call itself recursively, or throw an exception, but never return normally. This is reflected by **exc** being the only transition leading from state C0 to the accepting state Mr, and p_B and p_C having no way to a normal **ret**. The OPA has a look-ahead of one input symbol, so when it encounters **exc**, it must pop all symbols in the stack, corresponding to active function frames, until it finds the one with **han** in it, which cannot be popped because $\mathbf{han} \doteq \mathbf{exc}$. Notice that such behavior cannot be modeled by Visibly Pushdown Automata or Nested Word Automata, because they need to read an input symbol for each pop move. Thus, **han** protects the parent function from the exception. Since the state contained in **han**'s stack symbol is A0, the execution resumes in the `catch` clause of p_A. p_A then calls twice the error-handling function p_{Err}, which ends regularly both times, and returns. The string of Fig. 1 is accepted by this OPA.

In this example, we only model the stack behavior for simplicity, but other statements, such as assignments, and other behaviors, such as continuations, could be modeled by a different choice of the OPA and OPM, and other aspects of the program's state by appropriate abstractions [38].

3 POTL: Syntax and Semantics

Given a finite set of atomic propositions AP, the syntax of POTL follows:

$$\varphi ::= \mathrm{a} \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc^t \varphi \mid \ominus^t \varphi \mid \chi_F^t \varphi \mid \chi_P^t \varphi \mid \varphi\, \mathcal{U}_\chi^t\, \varphi \mid \varphi\, \mathcal{S}_\chi^t\, \varphi$$
$$\mid \bigcirc_H^t \varphi \mid \ominus_H^t \varphi \mid \varphi\, \mathcal{U}_H^t\, \varphi \mid \varphi\, \mathcal{S}_H^t\, \varphi$$

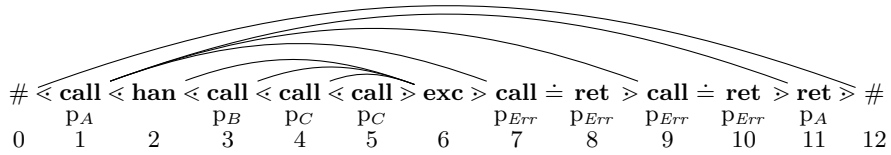

Fig. 3. The string of Fig. 1 as an OP word. Chains are shown by edges joining their contexts. Standard atomic propositions are shown below SL: p_l means a **call** or a **ret** is related to procedure p_l. First, procedure p_A is called (pos. 1), and it installs a handler in pos. 2. Then, three procedures are called, and one (p_C) throws an exception, which is caught by the handler. Two more functions are called and, finally, p_A returns.

where $\mathrm{a} \in AP$, and $t \in \{d, u\}$.

The semantics of POTL is based on the *word structure*—also called *OP word* for short—(U, M_{AP}, P), where $U = \{0, 1, \ldots, n, n{+}1\}$, with $n \in \mathbb{N}$ is a set of word positions; $P\colon U \to \mathcal{P}(AP)$ is a function associating each position in U with the set of atomic propositions holding in that position, with $P(0) = P(n{+}1) = \{\#\}$. Given two positions i, j and a PR π, we write $i\ \pi\ j$ to say $P(i)\ \pi\ P(j)$.

We define the chain relation $\chi \subseteq U \times U$ so that $\chi(i, j)$ holds between two positions i, j iff $i < j - 1$, and i and j are resp. the left and right contexts of the same chain. For composed chains, χ may not be one-to-one, but also one-to-many or many-to-one. Given $i, j \in U$, relation χ has the following properties:

1. It never crosses itself: if $\chi(i, j)$ and $\chi(h, k)$, for any $h, k \in U$, then we have $i < h < j \implies k \leq j$ and $i < k < j \implies i \leq h$.
2. If $\chi(i, j)$, then $i \lessdot i + 1$ and $j - 1 \gtrdot j$.
3. There exists at most one single position h, called *leftmost context* of j, s.t. $\chi(h, j)$ and $h \lessdot j$ or $h \doteq j$; for any k s.t. $\chi(k, j)$ and $k \gtrdot j$ we have $k > h$.
4. There exists at most one single position h, called *rightmost context* of i, s.t. $\chi(i, h)$ and $i \gtrdot h$ or $i \doteq h$; for any k s.t. $\chi(i, k)$ and $i \lessdot k$ we have $k < h$.

Property 4 says that when the chain relation is one-to-many, the contexts of the outermost chains are in the $\doteq$ or $\gtrdot$ relation, while the inner ones are in the $\lessdot$ relation. Property 3 says that contexts of outermost many-to-one chains are in the $\doteq$ or $\lessdot$ relation, the inner ones being in the $\gtrdot$ relation. In the ST, the right context j of a chain is at the *same level* as the left one i when $i \doteq j$ (e.g., in Fig. 4, pos. 1 and 11), at a *lower level* when $i \lessdot j$ (e.g., pos. 1 with 7, and 9), at a *higher level* if $i \gtrdot j$ (e.g., pos. 3 and 4 with 6).

The truth of POTL formulas is defined w.r.t. a single word position. Let w be an OP word, and $\mathrm{a} \in AP$. Then, for any position $i \in U$ of w, we have $(w, i) \models \mathrm{a}$ if $\mathrm{a} \in P(i)$. Operators such as $\wedge$ and $\neg$ have the usual semantics from propositional logic. Next, while giving the formal semantics of POTL operators, we illustrate it by showing how it can be used to express properties on program execution traces, such as the one of Fig. 3.

a) Next/back operators. The *downward* next and back operators $\bigcirc^d$ and $\ominus^d$ are like their LTL counterparts, except they are true only if the next (resp. current) position is at a lower or equal ST level than the current (resp. preceding) one. The *upward* next and back, $\bigcirc^u$ and $\ominus^u$, are symmetric. Formally, $(w,i) \models \bigcirc^d\varphi$ iff $(w,i{+}1) \models \varphi$ and $i \lessdot (i{+}1)$ or $i \doteq (i{+}1)$, and $(w,i) \models \ominus^d\varphi$ iff $(w,i{-}1) \models \varphi$, and $(i-1) \lessdot i$ or $(i-1) \doteq i$. Substitute $\lessdot$ with $\gtrdot$ to obtain the semantics for $\bigcirc^u$ and $\ominus^u$. E.g., we can write $\bigcirc^d$**call** to say that the next position is an inner call (it holds in pos. 2, 3, 4 of Fig. 3), $\ominus^d$**call** to say that the previous position is a **call**, and the current is the first of the body of a function (pos. 2, 4, 5), or the **ret** of an empty one (pos. 8, 10), and $\ominus^u$**call** to say that the current position terminates an empty function frame (holds in 6, 8, 10). In pos. 2 $\bigcirc^d \mathrm{p}_B$ holds, but $\bigcirc^u \mathrm{p}_B$ does not.

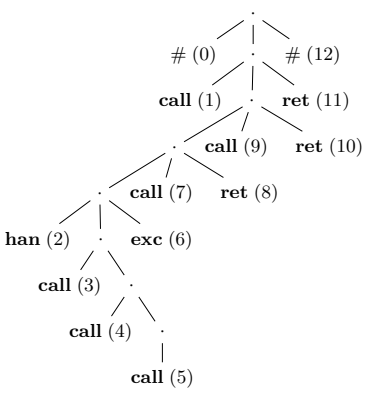

Fig. 4. The ST corresponding to the word of Fig. 3. Dots are internal nodes.

b) Chain next/back operators. The *chain* next and back operators χ_F^t and χ_P^t, $t \in \{d, u\}$, evaluate their argument respectively on future and past positions in the chain relation with the current one. The *downward* (resp. *upward*) variant only considers chains whose right context goes down (resp. up) in the ST. E.g., in pos. 1 of Fig. 3, $\chi_F^d \mathrm{p}_{Err}$ holds because $\chi(1,7)$ and $\chi(1,9)$, meaning that p_A calls p_{Err} at least once. Formally, $(w,i) \models \chi_F^d\varphi$ iff there exists a position $j > i$ such that $\chi(i,j)$, $i \lessdot j$ or $i \doteq j$, and $(w,j) \models \varphi$. $(w,i) \models \chi_P^d\varphi$ iff there exists a position $j < i$ such that $\chi(j,i)$, $j \lessdot i$ or $j \doteq i$, and $(w,j) \models \varphi$. Replace $\lessdot$ with $\gtrdot$ for the upward versions. In Fig. 3, χ_F^u**exc** is true in **call** positions whose procedure is terminated by an exception thrown by an inner procedure (e.g. pos. 3 and 4). χ_P^u**call** is true in **exc** statements that terminate at least one procedure other than the one raising it, such as the one in pos. 6. χ_F^d**ret** and χ_F^u**ret** hold in **call**s to non-empty procedures that terminate normally, and not due to an uncaught exception (e.g., pos. 1).

c) Until/Since operators. POTL has two kinds of until and since operators. They express properties on paths, which are sequences of positions obtained by iterating the different kinds of next or back operators. In general, a *path* of length $n \in \mathbb{N}$ between $i, j \in U$ is a sequence of positions $i = i_1 < i_2 < \cdots < i_n = j$. The *until* operator on a set of paths Γ is defined as follows: for any word w and position $i \in U$, and for any two POTL formulas φ and ψ, $(w,i) \models \varphi\, \mathcal{U}(\Gamma)\, \psi$ iff there exist a position $j \in U$, $j \geq i$, and a path $i_1 < i_2 < \cdots < i_n$ between i and j in Γ such that $(w,i_k) \models \varphi$ for any $1 \leq k < n$, and $(w,i_n) \models \psi$. *Since* operators are defined symmetrically. Note that, depending on Γ, a path from i to j may not exist. We define until/since operators by associating them with different sets of paths.

The *summary* until $\psi \mathcal{U}^t_\chi \theta$ (resp. since $\psi \mathcal{S}^t_\chi \theta$) operator is obtained by inductively applying the $\bigcirc^t$ and χ^t_F (resp. $\ominus^t$ and χ^t_P) operators. It holds in a position in which either θ holds, or ψ holds together with $\bigcirc^t(\psi \mathcal{U}^t_\chi \theta)$ (resp. $\ominus^t(\psi \mathcal{S}^t_\chi \theta)$) or $\chi^t_F(\psi \mathcal{U}^t_\chi \theta)$ (resp. $\chi^t_P(\psi \mathcal{S}^t_\chi \theta)$). It is an until operator on paths that can move not only between consecutive positions, but also between contexts of a chain, skipping its body. With the OPM of Fig. 1, this means skipping function bodies. The downward variants can move between positions at the same level in the ST (i.e., in the same simple chain body), or down in the nested chain structure. The upward ones remain at the same level, or move to higher levels of the ST.

Formula $\top \mathcal{U}^u_\chi$ **exc** is true in positions contained in the frame of a function that is terminated by an exception. It is true in pos. 3 of Fig. 3 because of path 3-6, and false in pos. 1, because no path can enter the chain whose contexts are pos. 1 and 11. Formula $\top \mathcal{U}^d_\chi$ **exc** is true in call positions whose function frame contains **exc**s, but that are not necessarily terminated by one of them, such as the one in pos. 1 (with path 1-2-6).

We define *Downward Summary Paths* (DSP) as follows. Given an OP word w, and two positions $i \leq j$ in w, the DSP between i and j, if it exists, is a sequence of positions $i = i_1 < i_2 < \cdots < i_n = j$ such that, for each $1 \leq p < n$,

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

The Downward Summary (DS) until and since operators $\mathcal{U}^d_\chi$ and $\mathcal{S}^d_\chi$ use as Γ the set of DSP starting in the position in which they are evaluated. The definition for the upward counterparts is, again, obtained by substituting $\lessdot$ with $\gtrdot$. In Fig. 3, $\mathbf{call}\, \mathcal{U}^d_\chi\, (\mathbf{ret} \wedge \mathrm{p}_{Err})$ holds in pos. 1 because of path 1-7-8 and 1-9-10, $(\mathbf{call} \vee \mathbf{exc})\, \mathcal{S}^u_\chi\, \mathrm{p}_B$ in pos. 7 because of path 3-6-7, and $(\mathbf{call} \vee \mathbf{exc})\, \mathcal{U}^u_\chi\, \mathbf{ret}$ in 3 because of path 3-6-7-8.

d) Hierarchical operators. A single position may be the left or right context of multiple chains. The operators seen so far cannot keep this fact into account, since they "forget" about a left context when they jump to the right one. Thus, we introduce the *hierarchical* next and back operators. The *upward* hierarchical next (resp. back), $\bigcirc^u_H \psi$ (resp. $\ominus^u_H \psi$), is true iff the current position j is the right context of a chain whose left context is i, and ψ holds in the next (resp. previous) pos. j' that is the right context of i, with $i \lessdot j, j'$. So, $\bigcirc^u_H \mathrm{p}_{Err}$ holds in pos. 7 of Fig. 3 because p_{Err} holds in 9, and $\ominus^u_H \mathrm{p}_{Err}$ in 9 because p_{Err} holds in 7. In the ST, $\bigcirc^u_H$ goes *up* between **call**s to p_{Err}, while $\ominus^u_H$ goes down. Their *downward* counterparts behave symmetrically, and consider multiple inner chains sharing their right context. They are formally defined as:

- $(w, i) \models \bigcirc^u_H \varphi$ 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 $(w, j) \models \varphi$;
- $(w, i) \models \ominus^u_H \varphi$ iff there exist a position $h < i$ s.t. $\chi(h, i)$ and $h \lessdot i$ and a position $j = \max\{k \mid k < i \wedge \chi(h, k) \wedge h \lessdot k\}$ and $(w, j) \models \varphi$;
- $(w, i) \models \bigcirc^d_H \varphi$ 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 $(w, j) \models \varphi$;

– $(w, i) \models \ominus_H^d \varphi$ iff there exist a position $h > i$ s.t. $\chi(i, h)$ and $i \succcurlyeq h$ and a position $j = \max\{k \mid k < i \wedge \chi(k, h) \wedge k \succcurlyeq h\}$ and $(w, j) \models \varphi$.

In the ST of Fig. 4, $\bigcirc_H^d$ and $\ominus_H^d$ go *down* and up among **call**s terminated by the same **exc**. For example, in pos. 3 $\bigcirc_H^d \mathrm{p}_C$ holds, because both pos. 3 and 4 are in the chain relation with 6. Similarly, in pos. 4 $\ominus_H^d \mathrm{p}_B$ holds. Note that these operators do not consider leftmost/rightmost contexts, so $\bigcirc_H^u$**ret** is false in pos. 9, as **call** $\doteq$ **ret**, and pos. 11 is the rightmost context of pos. 1.

The hierarchical until and since operators are defined by iterating these next and back operators. The upward hierarchical path (UHP) between i and j is a sequence of positions $i = i_1 < i_2 < \cdots < i_n = j$ such that there exists a position $h < i$ such that for each $1 \leq p \leq n$ we have $\chi(h, i_p)$ and $h \preccurlyeq i_p$, and for each $1 \leq q < n$ there exists no position k such that $i_q < k < i_{q+1}$ and $\chi(h, k)$. The until and since operators based on the set of UHP starting in the position in which they are evaluated are denoted as $\mathcal{U}_H^u$ and $\mathcal{S}_H^u$. E.g., **call** $\mathcal{U}_H^u\, \mathrm{p}_{Err}$ holds in pos. 7 because of the singleton path 7 and path 7-9, and **call** $\mathcal{S}_H^u\, \mathrm{p}_{Err}$ in pos. 9 because of paths 9 and 7-9.

The downward hierarchical path (DHP) between i and j is a sequence of positions $i = i_1 < i_2 < \cdots < i_n = j$ such that there exists a position $h > j$ such that for each $1 \leq p \leq n$ we have $\chi(i_p, h)$ and $i_p \succcurlyeq h$, and for each $1 \leq q < n$ there exists no position k such that $i_q < k < i_{q+1}$ and $\chi(k, h)$. The until and since operators based on the set of DHP starting in the position in which they are evaluated are denoted as $\mathcal{U}_H^d$ and $\mathcal{S}_H^d$. In Fig. 3, **call** $\mathcal{U}_H^d\, \mathrm{p}_C$ holds in pos. 3, and **call** $\mathcal{S}_H^d\, \mathrm{p}_B$ in pos. 4, both because of path 3-4.

The POTL until and since operators enjoy expansion laws similar to those of LTL. Here we give those for two until operators, those for their since and downward counterparts being symmetric.

$$\varphi\, \mathcal{U}_\chi^t\, \psi \equiv \psi \vee \Big(\varphi \wedge \big(\bigcirc^t (\varphi\, \mathcal{U}_\chi^t\, \psi) \vee \chi_F^t(\varphi\, \mathcal{U}_\chi^t\, \psi)\big)\Big)$$
$$\varphi\, \mathcal{U}_H^u\, \psi \equiv (\psi \wedge \chi_P^d \top \wedge \neg \chi_P^u \top) \vee \big(\varphi \wedge \bigcirc_H^u (\varphi\, \mathcal{U}_H^u\, \psi)\big)$$

3.1 Expressiveness of POTL

We first define some derived operators. For $t \in \{d, u\}$, we define the downward/upward summary *eventually* as $\Diamond^t \varphi := \top\, \mathcal{U}_\chi^t\, \varphi$, and the downward/upward summary *globally* as $\Box^t \varphi := \neg \Diamond^t (\neg \varphi)$. $\Diamond^u \varphi$ and $\Box^u \varphi$ resp. say that φ holds in one or all positions in the path from the current position to the root of the ST. $\Diamond^d \varphi$ says that φ holds in at least one position in the current subtree, and $\Box^d \varphi$ in all of them. E.g., if $\Box^d(\neg \mathrm{p}_A)$ holds in a **call**, it means that p_A never holds in its whole function body, which is the subtree rooted next to the **call**.

In the technical report, we prove

Theorem 1 ([23]). *POTL = FOL with one free variable on OP words.*

Equivalence to FOL on the relevant algebraic structure is a desirable feature of linear-time temporal logics, and it was proved for LTL [39] and NWTL [2]. It

is in some sense a theoretical assurance of the sufficient expressive power of the logic. Moreover, NWTL $\subset$ OPTL was proved in [22], and OPTL $\subseteq$ POTL comes from Theorem 1 and the semantics of OPTL being expressible in FOL. In [23], we also prove that there exist POTL formulas not expressible in OPTL. Thus, we can claim CaRet [6] $\subseteq$ NWTL $\subset$ OPTL $\subset$ POTL. One of such formulas is $\Diamond^d \mathrm{p}_A$ which, evaluated e.g. on a **han** position with a matched **exc**, states that p_A holds in one of the positions in the same subtree.

More importantly, POTL can express many useful requirements of procedural programs. To emphasize the potential practical applications in automatic verification, we supply a few examples of typical program properties expressed as POTL formulas, not all of them being expressible in the other above languages.

The LTL *globally* can be written as $\Box\psi := \neg\Diamond^u(\Diamond^d\neg\psi)$. The two nested eventually operators enumerate all future positions by going up and then down in any direction in the syntax tree: when negated, this means $\neg\psi$ may never hold. POTL can express Hoare-style pre/postconditions with formulae such as $\Box(\mathbf{call} \wedge \rho \implies \chi_F^d(\mathbf{ret} \wedge \theta))$, where ρ is the precondition, and θ is the postcondition.

Unlike NWTL, POTL can easily express properties related to exception handling and interrupt management [43]. E.g., the shortcut $\mathit{CallThr}(\psi) := \bigcirc^u(\mathbf{exc} \wedge \psi) \vee \chi_F^u(\mathbf{exc} \wedge \psi)$, evaluated in a **call**, states that the procedure currently started is terminated by a **exc** in which ψ holds. So, $\Box(\mathbf{call} \wedge \rho \wedge \mathit{CallThr}(\top) \implies \mathit{CallThr}(\theta))$ means that if precondition ρ holds when a procedure is called, then postcondition θ must hold if that procedure is terminated by an exception. In object oriented programming languages, if $\rho \equiv \theta$ is a class invariant asserting that a class instance's state is valid, this formula expresses *weak exception safety* [1], and *strong exception safety* if ρ and θ express particular states of the class instance. The *no-throw guarantee* can be stated with $\Box(\mathbf{call} \wedge \mathrm{p}_A \implies \neg\mathit{CallThr}(\top))$, meaning procedure p_A is never interrupted by an exception.

Stack inspection [29,37], i.e. properties regarding the sequence of procedures active in the program's stack at a certain point of its execution, is an important class of requirements that can be expressed with shortcut $\mathit{Scall}(\varphi, \psi) := (\mathbf{call} \implies \varphi)\, \mathcal{S}_\chi^d\, (\mathbf{call} \wedge \psi)$, which subsumes the *call since* of CaRet, as it also works with exceptions. E.g., $\Box\big((\mathbf{call} \wedge \mathrm{p}_B \wedge \mathit{Scall}(\top, \mathrm{p}_A)) \implies \mathit{CallThr}(\top)\big)$ means that whenever p_B is executed and at least one instance of p_A is on the stack, p_B is terminated by an exception. The OPA of Fig. 2 satisfies this formula, because p_B is called by p_A, and p_C throws.

4 Model Checking

Given an OP alphabet $(\mathcal{P}(AP), M_{AP})$, where AP is a finite set of atomic propositions, and a POTL formula φ, we build an OPA $\mathcal{A}_\varphi = (\mathcal{P}(AP), M_{AP}, Q, I, F, \delta)$ that accepts models of φ. The construction of $\mathcal{A}_\varphi$ resembles the classical one for LTL and the ones for NWTL and OPTL, diverging from them significantly when dealing with temporal obligations that involve positions in the chain relation.

We first introduce $\mathit{Cl}(\varphi)$, the *closure* of φ, containing all subformulas of φ, and some auxiliary operators. The latter are needed to model-check chain

next and back operators. For any PR $\pi \in \{\lessdot, \doteq, \gtrdot\}$, we define them as follows: $(w,i) \models \chi_F^\pi \varphi$ iff there exists $j > i$ such that $\chi(i,j)$, $i \ \pi \ j$, and $(w,j) \models \varphi$; $(w,i) \models \chi_P^\pi \varphi$ iff there exists $j < i$ such that $\chi(j,i)$, $j \ \pi \ i$, and $(w,j) \models \varphi$.

$Cl(\varphi)$ is the smallest set such that, for $t \in \{d,u\}$:

1. $\varphi \in Cl(\varphi)$,
2. $AP \subseteq Cl(\varphi)$,
3. if $\psi \in Cl(\varphi)$ and $\psi \neq \neg\theta$, then $\neg\psi \in Cl(\varphi)$ (we identify $\neg\neg\psi$ with ψ);
4. if $\neg\psi \in Cl(\varphi)$, then $\psi \in Cl(\varphi)$;
5. if any of $\psi \wedge \theta$ or $\psi \vee \theta$ is in $Cl(\varphi)$, then $\psi, \theta \in Cl(\varphi)$;
6. if any of $\bigcirc^t\psi$, $\ominus^t\psi$, $\chi_F^t\psi$, or $\chi_P^t\psi$ is in $Cl(\varphi)$, then $\psi \in Cl(\varphi)$;
7. if $\chi_F^d\psi$ (resp. $\chi_F^u\psi$) is in $Cl(\varphi)$, then $\chi_F^{\lessdot}\psi$ (resp. $\chi_F^{\gtrdot}\psi$), $\chi_F^{\doteq}\psi$, χ_L are in it;
8. if $\chi_P^d\psi$ (resp. $\chi_P^u\psi$) is in $Cl(\varphi)$, then $\chi_P^{\lessdot}\psi$ (resp. $\chi_P^{\gtrdot}\psi$), $\chi_P^{\doteq}\psi$ are in it;
9. if any of $\psi \,\mathcal{U}_\chi^t\, \theta$, $\psi \,\mathcal{S}_\chi^t\, \theta$, $\psi \,\mathcal{U}_H^t\, \theta$, or $\psi \,\mathcal{S}_H^t\, \theta$ is in $Cl(\varphi)$, then $\psi, \theta \in Cl(\varphi)$;
10. if $\psi \,\mathcal{U}_\chi^t\, \theta \in Cl(\varphi)$, then $\bigcirc^t(\psi \,\mathcal{U}_\chi^t\, \theta), \chi_F^t(\psi \,\mathcal{U}_\chi^t\, \theta) \in Cl(\varphi)$ (since is symmetric).

The set $Atoms(\varphi)$ contains all consistent subsets of $Cl(\varphi)$, i.e. all $\Phi \subseteq Cl(\varphi)$ s.t.

- for every $\psi \in Cl(\varphi)$, $\psi \in \Phi$ iff $\neg\psi \notin \Phi$;
- $\psi \wedge \theta \in \Phi$, iff $\psi \in \Phi$ and $\theta \in \Phi$;
- $\psi \vee \theta \in \Phi$, iff $\psi \in \Phi$ or $\theta \in \Phi$, or both.

The consistency constraints on $Atoms(\varphi)$ will be augmented incrementally in the following, for each operator.

The set of states of $\mathcal{A}_\varphi$ is $Q = Atoms(\varphi)^2$, and its elements, which we denote with Greek capital letters, are of the form $\Phi = (\Phi_c, \Phi_p)$, where Φ_c is the set of formulas that hold in the current position, and Φ_p is the set of temporal obligations. The latter keep track of arguments of temporal operators that must be satisfied after a chain body, skipping it. The way they do so depends on the transition relation δ, which we also define incrementally. Each automaton state is associated to word positions. So, for $(\Phi, a, \Psi) \in \delta_{push/shift}$, with $\Phi \in Atoms(\varphi)^2$ and $a \in \mathcal{P}(AP)$, we have $\Phi_c \cap AP = a$ (by $\Phi_c \cap AP$ we mean the set of atomic propositions in Φ_c). *Pop* moves do not read input symbols, and the automaton remains at the same position when performing them: for any $(\Phi, \Theta, \Psi) \in \delta_{pop}$ we impose $\Phi_c = \Psi_c$. The initial set I contains states of the form (Φ_c, Φ_p), with $\varphi \in \Phi_c$, and the final set F states of the form (Ψ_c, Ψ_p), s.t. $\Psi_c \cap AP = \{\#\}$ and Ψ_c contains no future operators. We extend the construction to the most important operators, leaving the others and correctness proofs to [21].

Next/Back Operators. Let $(\Phi, a, \Psi) \in \delta_{shift} \cup \delta_{push}$, with $\Phi, \Psi \in Atoms(\varphi)^2$, $a \in \mathcal{P}(AP)$, and let $b = \Psi_c \cap AP$: we have $\bigcirc^d\psi \in \Phi_c$ iff $\psi \in \Psi_c$ and either $a \lessdot b$ or $a \doteq b$. The constraints introduced for the $\ominus^d$ operator are symmetric, and for their upward counterparts it suffices to replace $\lessdot$ with $\gtrdot$.

If $\chi_F^d\psi \in Cl(\varphi)$, for each $\Phi \in Atoms(\varphi)^2$ we impose that $\chi_F^d\psi \in \Phi_c$ iff $\chi_F^{\lessdot}\psi \in \Phi_c$ or $\chi_F^{\doteq}\psi \in \Phi_c$. Analogous rules are defined for the upward and past chain operators. The auxiliary symbol χ_L forces the current position to be the first one of a chain body. Let the current state of the OPA be $\Phi \in Atoms(\varphi)^2$: $\chi_L \in \Phi_p$ iff the next transition (i.e. the one reading the current position) is a

	input	state	stack	PR	move
1	**call han exc ret** #	$\Phi_c^0 = \{\mathbf{call}, \chi_F^d\mathbf{ret}, \chi_F^{\doteq}\mathbf{ret}\}$, $\Phi_p^0 = \{\chi_L\}$	$\bot$	$\# \lessdot \mathbf{call}$	push
2	**han exc ret** #	$\Phi^1 = (\{\mathbf{han}\}, \{\chi_F^{\doteq}\mathbf{ret}, \chi_L\})$	$[\mathbf{call}, \Phi^0]\bot$	$\mathbf{call} \lessdot \mathbf{han}$	push
3	**exc ret** #	$\Phi^2 = (\{\mathbf{exc}\}, \emptyset)$	$[\mathbf{han}, \Phi^1][\mathbf{call}, \Phi^0]\bot$	$\mathbf{han} \doteq \mathbf{exc}$	shift
4	**ret** #	$\Phi^3 = (\{\mathbf{ret}\}, \emptyset)$	$[\mathbf{exc}, \Phi^1][\mathbf{call}, \Phi^0]\bot$	$\mathbf{exc} \gtrdot \mathbf{ret}$	pop
5	**ret** #	$\Phi^4 = (\{\mathbf{ret}\}, \{\chi_F^{\doteq}\mathbf{ret}\})$	$[\mathbf{call}, \Phi^0]\bot$	$\mathbf{call} \doteq \mathbf{ret}$	shift
6	#	$\Phi^5 = (\{\#\}, \emptyset)$	$[\mathbf{ret}, \Phi^0]\bot$	$\mathbf{ret} \gtrdot \#$	pop
7	#	$\Phi^5 = (\{\#\}, \emptyset)$	$\bot$	–	–

Fig. 5. Example accepting run of the automaton for $\chi_F^d\mathbf{ret}$.

push. Formally, if $(\Phi, a, \Psi) \in \delta_{shift}$ or $(\Phi, \Theta, \Psi) \in \delta_{pop}$, for any Φ, Θ, Ψ and a, then $\chi_L \notin \Phi_p$. If $(\Phi, a, \Psi) \in \delta_{push}$, then $\chi_L \in \Phi_p$. For any initial state $(\Phi_c, \Phi_p) \in I$, we have $\chi_L \in \Phi_p$ iff $\# \notin \Phi_c$.

If $\chi_F^{\doteq}\psi \in Cl(\varphi)$, its satisfaction is ensured by the following constraints on δ:

1. Let $(\Phi, a, \Psi) \in \delta_{push/shift}$: then $\chi_F^{\doteq}\psi \in \Phi_c$ iff $\chi_F^{\doteq}\psi, \chi_L \in \Psi_p$;
2. let $(\Phi, \Theta, \Psi) \in \delta_{pop}$: then $\chi_F^{\doteq}\psi \notin \Phi_p$, and $\chi_F^{\doteq}\psi \in \Theta_p$ iff $\chi_F^{\doteq}\psi \in \Psi_p$;
3. let $(\Phi, a, \Psi) \in \delta_{shift}$: then $\chi_F^{\doteq}\psi \in \Phi_p$ iff $\psi \in \Phi_c$.

If $\chi_F^{\lessdot}\psi \in Cl(\varphi)$, $\chi_F^{\lessdot}\psi$ is allowed in the pending part of initial states, and we add the following constraints:

4. Let $(\Phi, a, \Psi) \in \delta_{push/shift}$: then $\chi_F^{\lessdot}\psi \in \Phi_c$ iff $\chi_F^{\lessdot}\psi, \chi_L \in \Psi_p$;
5. let $(\Phi, \Theta, \Psi) \in \delta_{pop}$: then $\chi_F^{\lessdot}\psi \in \Theta_p$ iff $\chi_L \in \Psi_p$, and either $\chi_F^{\lessdot}\psi \in \Psi_p$ or $\psi \in \Phi_c$.

We illustrate how the construction works for $\chi_F^{\doteq}$ with the example of Fig. 5. The OPA starts in state Φ^0, with $\chi_F^d\mathbf{ret} \in \Phi_c^0$, and guesses that χ_F^d will be fulfilled by $\chi_F^{\doteq}$, so $\chi_F^{\doteq}\mathbf{ret} \in \Phi_c^0$. **call** is read by a push move, resulting in state Φ^1. The OPA guesses the next move will be a push, so $\chi_L \in \Phi_p^1$. By rule 1, we have $\chi_F^{\doteq}\mathbf{ret} \in \Phi_p^1$. The last guess is immediately verified by the next push (step 2-3). Thus, the pending obligation for $\chi_F^{\doteq}\mathbf{ret}$ is stored onto the stack in Φ^1. The OPA, then, reads **exc** with a shift, and pops the stack symbol containing Φ^1 (step 4-5). By rule 2, the temporal obligation is resumed in the next state Φ^4, so $\chi_F^{\doteq}\mathbf{ret} \in \Phi_p^4$. Finally, **ret** is read by a shift which, by rule 3, may occur only if $\mathbf{ret} \in \Phi_c^4$. Rule 3 verifies the guess that $\chi_F^{\doteq}\mathbf{ret}$ holds in Φ_0, and fulfills the temporal obligation contained in Φ_p^4, by preventing computations in which $\mathbf{ret} \notin \Phi_c^4$ from continuing. Had the next transition been a pop (e.g. because there was no **ret** and $\mathbf{call} \gtrdot \#$), the run would have been blocked by rule 2, preventing the OPA from reaching an accepting state, and from emptying the stack.

Summary Until and Since. The construction for these operators is based on their expansion laws. For any $\Phi \in Atoms(\varphi)^2$, we have $\psi\, \mathcal{U}_\chi^t\, \theta \in \Phi_c$, with

$t \in \{d, u\}$ being a direction, iff either: 1. $\theta \in \Phi_c$, 2. $\bigcirc^t(\psi\, \mathcal{U}^t_\chi\, \theta), \psi \in \Phi_c$, or 3. $\chi^t_F(\psi\, \mathcal{U}^t_\chi\, \theta), \psi \in \Phi_c$. The rules for since are symmetric.

Hierarchical Operators. For the hierarchical operators, we do not give an explicit OPA construction, but we rely on a translation into other POTL operands. For each hierarchical operator η in φ, we add a propositional symbol $\mathrm{q}_{(\eta)}$. The upward hierarchical operators consider the right contexts of chains sharing the same left context. To distinguish such positions, we define formula $\gamma_{L,\eta} := \chi^{\lessdot}_P\big(\mathrm{q}_{(\eta)} \wedge \bigcirc(\Box\neg \mathrm{q}_{(\eta)}) \wedge \ominus(\boxminus\neg \mathrm{q}_{(\eta)})\big)$, where $\Box$ and $\boxminus$ are as in Section 3.1. $\bigcirc$ and $\ominus$ are the LTL next and back operators, for which model checking can be done as for $\bigcirc^d$ and $\ominus^d$, but removing the restrictions on PR. $\gamma_{L,\eta}$, evaluated on a position i, asserts that $\mathrm{q}_{(\eta)}$ holds in the unique position h such that $\chi(h, i)$ and $h \lessdot i$. Thus, $\mathrm{q}_{(\eta)}$ can be used to distinguish other positions j such that $\chi(h, j)$ and $h \lessdot j$, as $\chi^{\lessdot}_P \mathrm{q}_{(\eta)}$ holds in them. The translations for future upward hierarchical operators follow, the others being analogous.

$$\bigcirc^u_H \psi := \gamma_{L,\bigcirc^u_H \psi} \wedge \bigcirc\big((\neg\chi^{\lessdot}_P \mathrm{q}_{(\bigcirc^u_H \psi)})\, \mathcal{U}^u_\chi\, (\chi^{\lessdot}_P \mathrm{q}_{(\bigcirc^u_H \psi)} \wedge \psi)\big)$$
$$\psi\, \mathcal{U}^u_H\, \theta := \gamma_{L,\psi \mathcal{U}^u_H \theta} \wedge (\chi^{\lessdot}_P \mathrm{q}_{(\psi \mathcal{U}^u_H \theta)} \implies \psi)\, \mathcal{U}^u_\chi\, (\chi^{\lessdot}_P \mathrm{q}_{(\psi \mathcal{U}^u_H \theta)} \wedge \theta)$$

4.1 Model Checking for ω-Words

To perform model checking of a POTL formula φ on OP ω-words, we build a generalized ωOPBA $\mathcal{A}^\omega_\varphi = (\mathcal{P}(AP), M_{AP}, Q_\omega, I, \mathbf{F}, \delta)$, where $Q_\omega = Atoms(\varphi)^2 \times \mathcal{P}(Cl_{stack}(\varphi))$, which differs from the finite-word OPA only for the state set and the acceptance condition. As in [2], the generalized Büchi acceptance condition is a slight variation on the one shown in Section 2.1: $\mathbf{F}$ is the set of sets of Büchi final states, and an ω-word is accepted iff at least one state from each one of the sets contained in $\mathbf{F}$ is visited infinitely often during the computation.

In finite words, the stack is empty at the end of every accepting computation, which implies the satisfaction of all temporal constraints tracked by the pending part of stack symbols. In ωOPBAs, the stack may never be empty, and symbols with a non-empty pending part may remain in it indefinitely, never enforcing the satisfaction of the respective formulas. To overcome this issue, we use $Atoms(\varphi)^2 \times \mathcal{P}(Cl_{stack}(\varphi))$, with $Cl_{stack}(\varphi) \subseteq Cl(\varphi)$, as the state set of the ωOPBA. Such states have the form $\Phi = (\Phi_c, \Phi_p, \Phi_s)$, where Φ_c and Φ_p have the same role as in the finite-word case, and Φ_s is the *in-stack* part of Φ. All rules previously defined for Φ_c and Φ_p remain the same. Φ_s contains elements of $Cl_{stack}(\varphi)$ contained in any symbol currently on the stack. $Cl_{stack}(\varphi)$ contains formulas in $Cl(\varphi)$ that use the stack to ensure the satisfaction of future temporal requirements, namely all $\chi^\pi_F \psi \in Cl(\varphi)$, with $\pi \in \{\lessdot, \doteq, \gtrdot\}$. Thus, pending temporal obligations are moved from the stack to the ωOPBA state, and they can be considered by the Büchi acceptance condition.

Suppose we want to model check $\chi^{\doteq}_F \psi$. Formula $\chi^{\doteq}_F \psi$ must be inserted in the in-stack part of the current state whenever a stack symbol containing it in its pending part is pushed. It must be kept in the in-stack part of the current state until the last stack symbol containing it in its pending part is popped, marking

	input	state	stack	PR
1	$\mathbf{call}\,\mathbf{call}\,\mathbf{han}\,\mathbf{exc}\,\mathbf{ret}\,\mathbf{ret}\,(\mathbf{call})^\omega$	$\Phi^0 = (\{\mathbf{call}, \chi_F^d\mathbf{ret}, \chi_F^{\doteq}\mathbf{ret}\}, \{\chi_L\}, \emptyset)$	$\bot$	$\lessdot$
2	$\mathbf{call}\,\mathbf{han}\,\mathbf{exc}\,\mathbf{ret}\,\mathbf{ret}\,(\mathbf{call})^\omega$	$\Phi^1 = (\{\mathbf{call}, \chi_F^d\mathbf{ret}, \chi_F^{\doteq}\mathbf{ret}\}, \{\chi_L, \chi_F^{\doteq}\mathbf{ret}\}, \emptyset)$	$[\mathbf{call}, \Phi^0]\bot$	$\lessdot$
3	$\mathbf{han}\,\mathbf{exc}\,\mathbf{ret}\,\mathbf{ret}\,(\mathbf{call})^\omega$	$\Phi^2 = (\{\mathbf{han}\}, \{\chi_L, \chi_F^{\doteq}\mathbf{ret}\}, \{\chi_F^{\doteq}\mathbf{ret}\})$	$[\mathbf{call}, \Phi^1][\mathbf{call}, \Phi^0]\bot$	$\lessdot$
4	$\mathbf{exc}\,\mathbf{ret}\,\mathbf{ret}\,(\mathbf{call})^\omega$	$\Phi^3 = (\{\mathbf{exc}\}, \emptyset, \{\chi_F^{\doteq}\mathbf{ret}\})$	$[\mathbf{han}, \Phi^2][\mathbf{call}, \Phi^1][\mathbf{call}, \Phi^0]\bot$	$\doteq$
5	$\mathbf{ret}\,\mathbf{ret}\,(\mathbf{call})^\omega$	$\Phi^4 = (\{\mathbf{ret}\}, \emptyset, \{\chi_F^{\doteq}\mathbf{ret}\})$	$[\mathbf{exc}, \Phi^2][\mathbf{call}, \Phi^1][\mathbf{call}, \Phi^0]\bot$	$\gtrdot$
6	$\mathbf{ret}\,\mathbf{ret}\,(\mathbf{call})^\omega$	$\Phi^5 = (\{\mathbf{ret}\}, \{\chi_F^{\doteq}\mathbf{ret}\}, \{\chi_F^{\doteq}\mathbf{ret}\})$	$[\mathbf{call}, \Phi^1][\mathbf{call}, \Phi^0]\bot$	$\doteq$
7	$\mathbf{ret}\,(\mathbf{call})^\omega$	Φ^4	$[\mathbf{ret}, \Phi^1][\mathbf{call}, \Phi^0]\bot$	$\doteq$
8	$\mathbf{ret}\,(\mathbf{call})^\omega$	$\Phi^6 = (\{\mathbf{ret}\}, \{\chi_F^{\doteq}\mathbf{ret}\}, \emptyset)$	$[\mathbf{call}, \Phi^0]\bot$	$\doteq$
9	$(\mathbf{call})^\omega$	$\Phi^7 = (\{\mathbf{call}\}, \emptyset, \emptyset)$	$[\mathbf{ret}, \Phi^0]\bot$	$\gtrdot$

Fig. 6. Prefix of an accepting run of the automaton for $\chi_F^d\mathbf{ret}$.

the satisfaction of its temporal requirement. Then, it is possible to define an acceptance set $F_{\chi_F^{\doteq}\psi} \in \mathbf{F}$, as the set of states not containing $\chi_F^{\doteq}\psi$ in any part. Fig. 6 shows an ωOPBA run of this kind. Notice that after step 7 $\chi_F^{\doteq}\psi$ does not appear in any state's in-stack part, so the run is accepting.

This construction is formalized as follows. Let $\psi \in Cl_{stack}(\varphi)$. We add a few constraints on the transition relations. For any $\Phi, \Theta, \Psi \in Q_\omega$ and $a \in \mathcal{P}(AP)$:

6. let $(\Phi, a, \Theta) \in \delta_{push}$: if $\psi \in \Phi_p$, then $\psi \in \Theta_s$;
7. let $(\Phi, a, \Theta) \in \delta_{push/shift}$: if $\psi \in \Phi_s$, then $\psi \in \Theta_s$;
8. let $(\Phi, \Theta, \Psi) \in \delta_{pop}$: if $\psi \in \Phi_s$ and $\psi \in \Theta_s$, then $\psi \in \Psi_s$.

An acceptance condition for summary until operators is also needed. For $\psi\,\mathcal{U}_\chi^d\,\theta \in Cl(\varphi)$, we add an acceptance set $\mathbf{F}_{\psi\mathcal{U}_\chi^d\theta}$ such that for any Φ in it we have $\chi_F^{\lessdot}(\psi\,\mathcal{U}_\chi^d\,\theta), \chi_F^{\doteq}(\psi\,\mathcal{U}_\chi^d\,\theta) \notin \Phi_s$, and either $\psi\,\mathcal{U}_\chi^d\,\theta \notin \Phi_c$ or $\theta \in \Phi_c$. The condition for $\psi\,\mathcal{U}_\chi^u\,\theta$ is symmetric.

4.2 Complexity

The set $Cl(\varphi)$ is linear in $|\varphi|$, the length of φ. $Atoms(\varphi)$ has size at most $2^{|\,Cl(\varphi)|} = 2^{O(|\varphi|)}$, and the size of the set of states is the square of that in the finite case, and is bounded by its cube in the ω-case. Moreover, the use of the equivalences for the hierarchical operators causes only a linear increase in the length of φ. Therefore,

Theorem 2. *Given a POTL formula φ, it is possible to build an OPA or an ωOPBA $\mathcal{A}_\varphi$ accepting the language denoted by φ with at most $2^{O(|\varphi|)}$ states.*

$\mathcal{A}_\varphi$ can then be intersected [42] with an OPA/ωOPBA modeling a program (e.g. Fig. 2), and emptiness can be decided with *summarization* techniques [4].

Table 1. Results of the evaluation. '# states' refers to the OPA to be verified.

	Benchmark name	# states	Time (ms)	Memory (KiB)		Result
				Total	MC only	
1	generic (Fig. 2)	12	867	70,040	10,166	True
2	generic medium	24	673	70,064	4,043	False
3	generic larger	30	1,014	70,063	14,160	True
4	Jensen	42	305	70,050	3,154	True
5	unsafe stack	63	1,493	109,610	43,177	False
6	safe stack	77	637	70,089	7,234	True
7	unsafe stack neutrality	63	5,286	383,312	167,654	True
8	safe stack neutrality	77	840	70,077	16,773	True

5 Experimental Evaluation

We implemented the OPA construction of Section 4 in an explicit-state model checking tool called POMC. The tool is written in Haskell [45], a purely functional, statically typed programming language with lazy evaluation. POMC checks OPA for emptiness by checking the reachability of an accepting configuration, by means of a modified DFS of the transition relation. This algorithm, similar to the one in [9], exploits the fact that all transitions only consider the topmost stack symbol, so reachability is actually computed only for *semi-configurations* made of one stack symbol and one state. Each time a chain support is explored, its ending semi-configuration is saved and associated with the starting one, so the next time the latter is reached, the support does not have to be re-explored. This allows the algorithm to exploit the cyclicities of OPA to terminate after having explored the whole transition relation. Given a POTL specification φ and an OPA $\mathcal{A}$ to be checked, POMC executes the reachability algorithm, generating the product between $\mathcal{A}$ and the OPA for $\neg\varphi$ on-the-fly. The present prototype of POMC only supports finite-word model checking; its extension to deal with ω-languages is under development.

We checked with POMC several requirements on three case studies and we report the results in Table 1. Some additional formulas we checked are in Table 2. Such results can be reproduced through a publicly available artifact.[4] The experiments were executed on a laptop with a 2.2 GHz Intel processor and 15 GiB of RAM, running Ubuntu GNU/Linux 20.04. In the tables, by "Total" memory we mean the maximum resident memory including the Haskell runtime (which allocates 70 MiB by default), and by "MC only" the maximum memory used by model checking as reported by the runtime. Since model checking is polynomial in OPA size and exponential in formula length, we focus on checking a variety of requirements, rather than large OPA.

Generic procedural program. We checked formula

$$\square\big((\mathbf{call} \wedge \mathrm{p}_B \wedge \mathit{Scall}(\top, \mathrm{p}_A)) \implies \mathit{CallThr}(\top)\big)$$

[4] https://doi.org/10.5281/zenodo.4723741

from Section 3.1 on the OPA of Fig. 2 (bench. 1), and also against two larger OPA (2, where the property does not hold, and 3, where it holds).

We also checked the largest of such OPA against a set of formulas devised with the purpose of testing all POTL operators. The results are reported in Table 2. All formulas are checked very quickly, with only one outlier that runs out of memory. We ran the same experiment on a machine with a 2.0 GHz AMD CPU and 512 GiB of RAM running Debian GNU/Linux 10, obtaining a time of 367 s with a memory occupancy of 16.3 GiB.

Stack Inspection. The security framework of the Java Development Kit (JDK) is based on stack inspection, i.e. the analysis of the contents of the program's stack during the execution. The JDK provides method `checkPermission(perm)` from class `AccessController`, which searches the stack for frames of functions that have not been granted permission `perm`. If any are found, an exception is thrown. Such permission checks prevent the execution of privileged code by unauthorized parts of the program, but they must be placed in sensitive points manually. Failure to place them appropriately may cause the unauthorized execution of privileged code. An automated tool to check that no code can escape such checks is thus desirable. Any such tool would need the ability to model exceptions, as they are used to avoid code execution in case of security violations.

[37] explains such needs by providing an example Java program for managing a bank account. It allows the user to check the account balance, and to withdraw money. To perform such tasks, the invoking program must have been granted permissions `CanPay` and `Debit`, respectively. We modeled such program as an OPA (4), and proved that the program enforces such security measures effectively by checking it against the formula

$$\Box(\mathbf{call} \wedge \texttt{read} \implies \neg(\top\, \mathcal{S}^d_\chi\, (\mathbf{call} \wedge \neg\texttt{CanPay} \wedge \neg\texttt{read})))$$

meaning that the account balance cannot be read if some function in the stack lacks the `CanPay` permission (a similar formula checks the `Debit` permission).

Exception Safety. [53] is a tutorial on how to make exception-safe generic containers in C++. It presents two implementations of a generic stack data structure, parametric on the element type `T`. The first one is not exception-safe: if the constructor of `T` throws an exception during a pop action, the topmost element is removed, but it is not returned, and it is lost. This violates the strong exception safety requirement that each operation is rolled back if an exception is thrown. The second version of the data structure instead satisfies such requirement.

While exception safety is, in general, undecidable, it is possible to prove the stronger requirement that each modification to the data structure is only committed once no more exceptions can be thrown. We modeled both versions as OPA, and checked such requirement with the following formula:

$$\Box(\mathbf{exc} \implies \neg((\ominus^u \texttt{modified} \vee \chi^u_P \texttt{modified}) \wedge \chi^u_P(\texttt{Stack} :: \texttt{push} \vee \texttt{Stack} :: \texttt{pop})))$$

POMC successfully found a counterexample for the first implementation (5), and proved the safety of the second one (6).

Additionally, we proved that both implementations are *exception neutral* (7, 8), i.e. `Stack` functions do not block exceptions thrown by the underlying type `T`. This was accomplished by checking the following formula:

$$\square(\mathbf{exc} \wedge \ominus^u \mathtt{T} \wedge \chi_P^d(\mathbf{han} \wedge \chi_P^d \mathtt{Stack}) \implies \chi_P^d \chi_P^d \chi_F^u \mathbf{exc}).$$

Table 2. Results of the additional experiments on OPA "generic larger".

Formula	Time (ms)	Memory (KiB) Tot.	MC	Result
$\chi_F^d \mathrm{p}_{Err}$	1.1	70,095	175	False
$\bigcirc^d(\bigcirc^d(\mathbf{call} \wedge \chi_F^u \mathbf{exc}))$	21.0	70,095	1,290	False
$\bigcirc^d(\mathbf{han} \wedge (\chi_F^d(\mathbf{exc} \wedge \chi_P^u \mathbf{call})))$	42.2	70,088	2,297	False
$\square(\mathbf{exc} \implies \chi_P^u \mathbf{call})$	10.7	70,099	839	True
$\top\, \mathcal{U}_\chi^d\, \mathbf{exc}$	2.2	70,093	121	False
$\bigcirc^d(\bigcirc^d(\top\, \mathcal{U}_\chi^d\, \mathbf{exc}))$	4.3	70,094	113	False
$\square((\mathbf{call} \wedge \mathrm{p}_A \wedge (\neg \mathbf{ret}\, \mathcal{U}_\chi^d\, \mathrm{WRx})) \implies \chi_F^u \mathbf{exc})$	3,257.7	238,833	102,582	True
$\bigcirc^d(\bigcirc^u \mathbf{call})$	0.7	70,094	139	False
$\bigcirc^d(\bigcirc^d(\bigcirc^d(\ominus^u \mathbf{call})))$	3.4	70,108	126	False
$\chi_F^d(\bigcirc^d(\ominus^u \mathbf{call}))$	1.3	70,096	137	False
$\square((\mathbf{call} \wedge \mathrm{p}_A \wedge \mathit{CallThr}(\top)) \implies \mathit{CallThr}(\mathrm{e}_B))$	7,793.7	402,420	173,639	False
$\diamond(\bigcirc_H^d \mathrm{p}_B)$	2.1	70,097	114	False
$\diamond(\ominus_H^d \mathrm{p}_B)$	2.8	70,097	114	False
$\diamond(\mathrm{p}_A \wedge (\mathbf{call}\, \mathcal{U}_H^d\, \mathrm{p}_C))$	594.9	77,806	29,786	True
$\diamond(\mathrm{p}_C \wedge (\mathbf{call}\, \mathcal{S}_H^d\, \mathrm{p}_A))$	676.6	96,296	37,949	True
$\square((\mathrm{p}_C \wedge \chi_F^u \mathbf{exc}) \implies (\neg \mathrm{p}_A\, \mathcal{S}_H^d\, \mathrm{p}_B))$	—	—	—	OOM
$\square(\mathbf{call} \wedge \mathrm{p}_B \implies \neg \mathrm{p}_C\, \mathcal{U}_H^u\, \mathrm{p}_{Err})$	198.2	70,088	10,606	True
$\diamond(\bigcirc_H^u \mathrm{p}_{Err})$	1.1	70,093	114	False
$\diamond(\ominus_H^u \mathrm{p}_{Err})$	1.2	70,089	114	False
$\diamond(\mathrm{p}_A \wedge (\mathbf{call}\, \mathcal{U}_H^u\, \mathrm{p}_B))$	10.3	70,105	115	False
$\diamond(\mathrm{p}_B \wedge (\mathbf{call}\, \mathcal{S}_H^u\, \mathrm{p}_A))$	10.8	70,095	115	False
$\square(\mathbf{call} \implies \chi_F^d \mathbf{ret})$	3.0	70,095	112	False
$\square(\mathbf{call} \implies \neg \bigcirc^u \mathbf{exc})$	1.9	70,106	113	False
$\square(\mathbf{call} \wedge \mathrm{p}_A \implies \neg \mathit{CallThr}(\top))$	110.7	70,094	4,937	False
$\square(\mathbf{exc} \implies \neg(\ominus^u(\mathbf{call} \wedge \mathrm{p}_A) \vee \chi_P^u(\mathbf{call} \wedge \mathrm{p}_A)))$	28.9	70,095	112	False
$\square((\mathbf{call} \wedge \mathrm{p}_B \wedge (\mathbf{call}\, \mathcal{S}_\chi^d\, (\mathbf{call} \wedge \mathrm{p}_A))) \implies \mathit{CallThr}(\top)$	926.1	70,104	13,310	True
$\square(\mathbf{han} \implies \chi_F^u \mathbf{ret})$	17.0	70,079	1,252	True
$\top\, \mathcal{U}_\chi^u\, \mathbf{exc}$	7.7	70,101	121	True
$\bigcirc^d(\bigcirc^d(\top\, \mathcal{U}_\chi^u\, \mathbf{exc}))$	44.6	70,104	2,376	True
$\bigcirc^d(\bigcirc^d(\bigcirc^d(\top\, \mathcal{U}_\chi^u\, \mathbf{exc})))$	123.7	70,090	5,261	False
$\square(\mathbf{call} \wedge \mathrm{p}_C \implies (\top\, \mathcal{U}_\chi^u\, \mathbf{exc} \wedge \chi_P^d \mathbf{han}))$	92.9	70,096	1,346	False
$\mathbf{call}\, \mathcal{U}_\chi^d\, (\mathbf{ret} \wedge \mathrm{p}_{Err})$	1.8	70,107	114	False
$\chi_F^d(\mathbf{call} \wedge ((\mathbf{call} \vee \mathbf{exc})\, \mathcal{S}_\chi^u\, \mathrm{p}_B))$	10.8	70,086	117	False
$\bigcirc^d(\bigcirc^d((\mathbf{call} \vee \mathbf{exc})\, \mathcal{U}_\chi^u\, \mathbf{ret}))$	5.3	70,094	114	False

6 Conclusions

We introduced the temporal logic POTL, gave an automata-theoretic model checking procedure, and implemented it in a prototype tool. The results obtained

in its experimental evaluation are promising. Additionally, POTL is proved to be FO-complete in a technical report [23]. We argue that the strong gain in expressive power w.r.t. previous approaches to model checking CFL, which comes without an increase in computational complexity, is worth the technicalities needed to achieve the present—and future—results.

In the evaluation, we used models directly coded into OPAs. To ease user interaction with our tool, we additionally implemented a new input format based on a simple procedural language with exceptions and Boolean variables, which is automatically translated into OPA. Moreover, we are currently working on the implementation of the model checking for ω-words, described in Section 4.1.

As a future research step, we plan to develop user-friendly domain-specific languages for specification too, to prove that OP languages and logics are suitable in practice to program verification.

Acknowledgments

We are thankful to Davide Bergamaschi for developing an early POMC prototype, and to Francesco Pontiggia for implementing performance optimizations.

References

1. Abrahams, D.: Exception-Safety in Generic Components. In: Generic Programming. pp. 69–79. Springer (2000). 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)
3. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T., Yannakakis, M.: Analysis of recursive state machines. ACM Trans. Program. Lang. Syst. **27**(4), 786–818 (2005). https://doi.org/10.1145/1075382.1075387
4. Alur, R., Bouajjani, A., Esparza, J.: Model checking procedural programs. In: Handbook of Model Checking, pp. 541–572. Springer (2018)
5. Alur, R., Chaudhuri, S., Madhusudan, P.: Software model checking using languages of nested trees. ACM Trans. Program. Lang. Syst. **33**(5), 15:1–15:45 (2011)
6. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: TACAS 2004. pp. 467–481. Springer (2004)
7. Alur, R., Madhusudan, P.: Visibly Pushdown Languages. In: ACM STOC (2004)
8. Alur, R., Madhusudan, P.: Adding nesting structure to words. JACM **56**(3) (2009)
9. Alur, R., Chaudhuri, S., Etessami, K., Madhusudan, P.: On-the-fly reachability and cycle detection for recursive state machines. In: TACAS 2005. LNCS, vol. 3440, pp. 61–76. Springer (2005). https://doi.org/10.1007/978-3-540-31980-1_5
10. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for Boolean programs. In: SPIN Model Checking and Software Verification. pp. 113–130. Springer (2000)
11. Barenghi, A., Crespi Reghizzi, S., Mandrioli, D., Panella, F., Pradella, M.: Parallel parsing made practical. Sci. Comput. Program. **112**, 195–226 (2015). https://doi.org/10.1016/j.scico.2015.09.002
12. Bouajjani, A., Echahed, R., Habermehl, P.: On the verification problem of nonregular properties for nonregular processes. In: LICS 95. pp. 123–133 (1995)

13. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: CONCUR '97. pp. 135–150. Springer (1997)
14. Bouajjani, A., Habermehl, P.: Constrained properties, semilinear systems, and petri nets. In: CONCUR '96. LNCS, vol. 1119, pp. 481–497. Springer (1996)
15. Bozzelli, L., Murano, A., Peron, A.: Timed context-free temporal logics. In: GandALF 2018. EPTCS, vol. 277, pp. 235–249. Open Publishing Association (2018). https://doi.org/10.4204/EPTCS.277.17
16. Bozzelli, L., Sánchez, C.: Visibly linear temporal logic. In: Automated Reasoning. pp. 418–433. Springer (2014). https://doi.org/10.1007/978-3-319-08587-6_33
17. Burkart, O., Steffen, B.: Model checking the full modal mu-calculus for infinite sequential processes. Theor. Comput. Sci. **221**(1-2), 251–270 (1999). https://doi.org/10.1016/S0304-3975(99)00034-1
18. Chatterjee, K., Ma, D., Majumdar, R., Zhao, T., Henzinger, T.A., Palsberg, J.: Stack size analysis for interrupt-driven programs. Inf. Comput. **194**(2), 144–174 (2004). https://doi.org/10.1016/j.ic.2004.06.001
19. Chaudhuri, S., Alur, R.: Instrumenting C programs with nested word monitors. In: SPIN '07. LNCS, vol. 4595, pp. 279–283. Springer (2007). https://doi.org/10.1007/978-3-540-73370-6_20
20. Chen, F., Rosu, G.: Java-MOP: A monitoring oriented programming environment for Java. In: TACAS 2005. LNCS, vol. 3440, pp. 546–550. Springer (2005). https://doi.org/10.1007/978-3-540-31980-1_36
21. Chiari, M., Mandrioli, D., Pradella, M.: POTL: A first-order complete temporal logic for operator precedence languages. CoRR **abs/1910.09327** (2019), http://arxiv.org/abs/1910.09327
22. 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
23. Chiari, M., Mandrioli, D., Pradella, M.: A first-order complete temporal logic for structured context-free languages. CoRR **abs/2105.10740** (2021), https://arxiv.org/abs/2105.10740
24. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8
25. Crespi Reghizzi, S., Mandrioli, D.: Operator Precedence and the Visibly Pushdown Property. JCSS **78**(6), 1837–1867 (2012). https://doi.org/10.1016/j.jcss.2011.12.006
26. D'Antoni, L.: A symbolic automata library, https://github.com/lorisdanto/symbolicautomata
27. Driscoll, E., Thakur, A.V., Reps, T.W.: OpenNWA: A Nested-Word Automaton Library. In: CAV 2012. LNCS, vol. 7358, pp. 665–671. Springer (2012)
28. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer (2000)
29. Esparza, J., Kučera, A., Schwoon, S.: Model checking LTL with regular valuations for pushdown systems. Information and Computation **186**(2), 355–376 (2003)
30. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. In: Infinity 1997. ENTCS, vol. 9, pp. 27–37. Elsevier (1997). https://doi.org/10.1016/S1571-0661(05)80426-8
31. Floyd, R.W.: Syntactic Analysis and Operator Precedence. JACM **10**(3), 316–333 (1963). https://doi.org/10.1145/321172.321179
32. Godefroid, P., Yannakakis, M.: Analysis of boolean programs. In: TACAS 2013. pp. 214–229. Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_16

33. Grune, D., Jacobs, C.J.: Parsing techniques: a practical guide. Springer, New York (2008). https://doi.org/10.1007/978-0-387-68954-8
34. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic, pp. 99–217. Springer (2002)
35. Harrison, M.A.: Introduction to Formal Language Theory. Addison Wesley (1978)
36. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In: SPIN 2003. LNCS, vol. 2648, pp. 235–239. Springer (2003)
37. Jensen, T., Le Metayer, D., Thorn, T.: Verification of control flow based security properties. In: Proc. '99 IEEE Symp. on Security and Privacy. pp. 89–103 (1999). https://doi.org/10.1109/SECPRI.1999.766902
38. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Handbook of Model Checking, pp. 447–491. Springer (2018)
39. Kamp, H.: Tense logic and the theory of linear order. Ph.D. thesis, University of California, Los Angeles (1968)
40. Kupferman, O., Piterman, N., Vardi, M.Y.: Model Checking Linear Properties of Prefix-Recognizable Systems. In: CAV 2002. LNCS, vol. 2404, pp. 371–385. Springer (2002). https://doi.org/10.1007/3-540-45657-0_31
41. Kupferman, O., Piterman, N., Vardi, M.Y.: Pushdown Specifications. In: LPAR 2002. LNCS, vol. 2514, pp. 262–277. Springer (2002)
42. 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
43. 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
44. Mandrioli, D., Pradella, M., Crespi Reghizzi, S.: Star-freeness, first-order definability and aperiodicity of structured context-free languages. In: ICTAC 2020. LNCS, vol. 12545, pp. 161–180. Springer (2020). https://doi.org/10.1007/978-3-030-64276-1_9
45. Marlow, S.: Haskell 2010 language report. https://www.haskell.org/onlinereport/haskell2010/ (2010)
46. McNaughton, R.: Parenthesis Grammars. JACM **14**(3), 490–500 (1967)
47. Mehlhorn, K.: Pebbling mountain ranges and its application of DCFL-recognition. In: ICALP '80. LNCS, vol. 85, pp. 422–435 (1980)
48. Nguyen, H.: Visibly pushdown automata library (2006), https://web.imt-atlantique.fr/x-info/hnguyen/vpa
49. Nguyen, H., Touili, T.: CARET model checking for malware detection. In: SPIN 2017. pp. 152–161. ACM (2017). https://doi.org/10.1145/3092282.3092301
50. Nguyen, H., Touili, T.: CARET model checking for pushdown systems. In: SAC 2017. pp. 1393–1400. ACM (2017). https://doi.org/10.1145/3019612.3019829
51. Piterman, N., Vardi, M.Y.: Global model-checking of infinite-state systems. In: CAV 2004. LNCS, vol. 3114, pp. 387–400. Springer (2004)
52. Rosu, G., Chen, F., Ball, T.: Synthesizing monitors for safety properties: This time with calls and returns. In: RV 2008. LNCS, vol. 5289, pp. 51–68. Springer (2008)
53. Sutter, H.: Exception-safe generic containers. C++ Report (1997), https://ptgmedia.pearsoncmg.com/imprint_downloads/informit/aw/meyerscddemo/DEMO/MAGAZINE/SU_FRAME.HTM
54. Tang, N.V., Ohsaki, H.: Checking on-the-fly universality and inclusion problems of visibly pushdown automata. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **94-A**(12), 2794–2801 (2011). https://doi.org/10.1587/transfun.E94.A.2794
55. Walukiewicz, I.: Pushdown processes: Games and model-checking. Information and Computation **164**(2), 234–263 (2001). https://doi.org/10.1006/inco.2000.2894