Verification of Programs with Exceptions through Operator Precedence Automata

Francesco Pontiggia[1], Michele Chiari[1][0000−0001−7742−9233], 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. Operator Precedence Languages are one of the most expressive classes of context-free languages that enable Model Checking. Recently, the First-Order complete Precedence Oriented Temporal Logic (POTL) has been introduced for expressing properties on models defined through Operator Precedence Automata (OPA), a variant of Pushdown Automata for OPLs; moreover, an efficient tool called Precedence Oriented Model Checker (POMC) was devised for POTL. We propose here the core algorithms of POMC for on-the-fly depth-first exploration of the search space: for OPA, a *reachability* algorithm; for their ω-word variant, a *fair-cycle detection* algorithm. We have refined the tool with a user-friendly DSL called MiniProc for expressing procedural code with exceptions. We show how the expressiveness of POMC can be used to verify programs which make use of exceptions, thus overcoming the limits of LTL-based Model Checking. We demonstrate the effectiveness of POMC through a case study.

Keywords: Linear Temporal Logic · Operator Precedence Languages · Model Checking · Software Verification · Exceptions

1 Introduction

In Model Checking, some of the most critical aspects are how to specify the model and the properties to be verified. Different formalisms have been proposed in the literature, and some have been successfully exploited due to their ease of development and nice performances when implemented in practice. Well-established tools (such as SPIN [19]) usually support the verification of properties expressed in Linear Temporal Logic (LTL) on models provided as Transition Systems or Finite State Automata (generally Büchi automata). Unfortunately, LTL can express only the First-Order definable fragment of regular languages. Transition Systems, although they can have an infinite set of states, thus being non-regular, are more suitable for hardware verification, since they do not have the concept of stack. Conversely, many relevant program behaviors regard execution traces composed of matching and nested (possibly even recursive) function calls and returns, hence they involve the manipulation of the stack. They are context-free, and cannot be modeled by regular formalisms; likewise, many useful properties

cannot be specified on them with LTL. To mention one, the evolution of the call stack of active subroutines (to verify stack inspection properties at a certain point of the execution [13,20]).

To fill this gap, attempts have been made by introducing logics based on languages which are context-free, but enjoy many nice properties of regular languages, and are regarded as being in the middle between context-free and regular languages. They are informally defined as Structured Context Free Languages [23], because the structure of the syntax tree of a sentence is built in the sentence itself, and in many cases immediately visible. Remarkable results have been obtained with Visibly Pushdown Languages (VPL) [7], and the derived logics CaRet [6] and Nested Word Temporal Logic (NWTL) (which is First-Order complete) [2]. In VPLs, sentences embed matches between characters: these matches are used to model function calls and returns. Consequently, with NWTL it is possible to define specifications on generic procedural programs [4]. Unfortunately, the matching relation is necessarily one-to-one. This property makes VPLs and NWTL not suitable to deal with behaviors in which a single event must be put in relation with multiple ones: for example, exception handling (e.g., to verify exception safety properties [1]), and context-switching policies in real time operating systems.

Regarding the modelling formalisms, *Extended Recursive State Machines* (ERSMs) and *Pushdown Systems* are equivalent abstractions [3] which have been proposed to model generic imperative programming languages. On the practical side, the former is supported by the tool VERA [5], which adopts an *on-the-fly* approach to perform reachability and fair-cycle detection analysis. Conversely, the latter is supported by the MOPED model checker [21,12,14], a BDD-based LTL model checker. However, none of them accepts CaRet or NWTL specifications. Both tools are able to deal with the family of *Boolean Programs* [8], which have a closer syntax to that of a program with assignments. They present procedures with call-by-value parameter passing and recursion, and a restricted form of recursion. With respect to ERSMs and Pushdown Systems, they do not allow array or bounded-integer variables. All the three formalisms do not present exceptions. Boolean programs are used in the SLAM verification toolkit [9]. SLAM provides a regression test suite made of 64 C programs, that are automatically abstracted and translated into Boolean Programs, and then verified through the ad-hoc BEBOP [8] model checker.

Operator Precedence Languages (OPLs) are a class of Context Free Languages introduced for efficient parsing [15]. Recently, their investigation has been resumed and applied to verification. OPLs allow to specify a many-to-one or one-to-many relation between sentence characters, and thus strictly include the class of VPLs [23]. Therefore, this class is a good fit for the verification of the mentioned exception handling behaviors or context-switching policies. A new logic based on OPLs, named Precedence Oriented Temporal Logic (POTL) [11], has been introduced, overcoming the previous, less expressive Operator Precedence Temporal Logic (OPTL) [10].

Alongside, a formal definition of the class of automata corresponding to OPLs has been given, with Operator Precedence Automata (OPA), and Operator Precedence Büchi Automata (OPBA) [22] which are OPA accepting infinite (or ω-) Operator Precedence words. The languages accepted by OPBA are called Operator Precedence ω-Languages (ω-OPLs). A first step towards the practical application of OP languages to the verification of real world programs has been taken in [11], which deals with some simple case studies regarding only hand-made OPA—hence, finite-word—models. In this paper we go a step further, and present the latest version of POMC[3], the Precedence Oriented Model Checker. POMC has been completed with an implementation of the model checking algorithm for ω-languages, therefore the tool now fully supports OPBA models.

To this regard, we outline the implemented reachability (for OPA) and fair-cycle detection (for OPBA) algorithms. The models can be provided either as plain automata or through a domain-specific language (DSL) called *MiniProc*, internally converted into automata by POMC. Although not Turing complete, MiniProc resembles mainstream programming languages. Thanks to these advancements, we present a larger case study on the Quicksort algorithm. We study three different implementations of the recursive Quicksort algorithm by modeling them with MiniProc. In particular, the third one is equipped with exception handling constructs: we verify on it various relevant properties, ranging from exception safety to stack inspection.

The paper is organized as follows: Sections 2 and 3 provide theoretical background and definitions; Section 4 describes the model-checking algorithms implemented in POMC; Section 5 describes the MiniProc DSL; Section 6 reports the QuickSort case study; Section 7 concludes with future work directions.

2 Background: 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) [17,18]. Operator Precedence Languages (OPLs) are usually defined through their generating grammars [15]; in this paper, however, we characterize them through their accepting automata [22] which are the natural way for stating equivalence properties with logic characterization, and for model checking. Readers not familiar with OPLs may refer to [23] for more explanations on the following 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

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

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 PRs 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, PRs 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 $(p, a, q) \in \delta_{push}$;

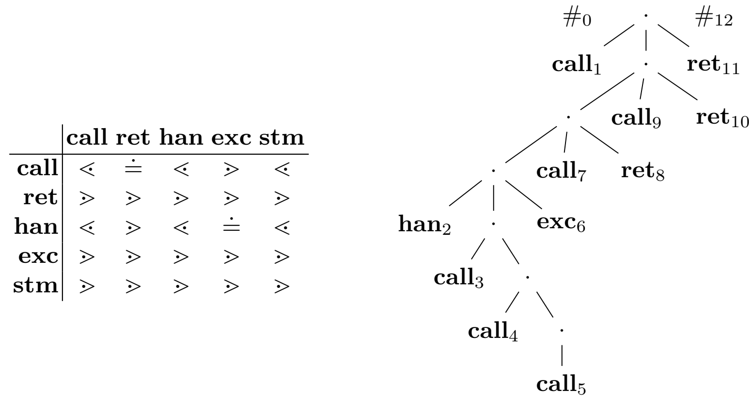
shift move: if $a \doteq b$ then $\langle bx,\ q,\ [a,\ p]\gamma \rangle \vdash \langle x,\ r,\ [b,\ p]\gamma \rangle$, with $(q, b, r) \in \delta_{shift}$;

pop move: if $a \gtrdot b$ then $\langle bx,\ q,\ [a,\ p]\gamma \rangle \vdash \langle bx,\ r,\ \gamma \rangle$, with $(q, p, r) \in \delta_{pop}$.

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

Fig. 1. OPM $M_{\mathbf{call}}$ (left) and the ST corresponding to word w'_{ex} (right). Dots are internal nodes.

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.

As an example, consider word $w_{ex} =$ **call han call call exc call ret ret** on alphabet $\Sigma_{\mathbf{call}} = \{\mathbf{call}, \mathbf{ret}, \mathbf{exc}, \mathbf{han}, \mathbf{stm}\}$, which is compatible with $M_{\mathbf{call}}$ of Fig. 1. w_{ex} models the execution trace of a program: first, a function is called; it installs an exception handler **han**, and then two more function calls occur. The last one throws an exception **exc**, which is caught by the handler. Before returning, the first function calls another one, which returns immediately. w_{ex} has a clear structure: **call**s should match **ret**s or **exc**s that terminate them, and **han**s to **exc**s they catch. Label **stm** represents a generic statement (e.g. an assignment), but we do not use it for now. Such structure is encoded by chains, which can be identified through the traditional operator precedence parsing algorithm. We apply it to w_{ex} (for a more complete treatment, cf. [23,17]).

First, write all precedence relations between consecutive characters, according to $M_{\mathbf{call}}$. Then, recognize all innermost patterns of the form $a \lessdot c \doteq \cdots \doteq c \gtrdot b$ 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} as follows:

$$
\begin{array}{r|l}
1 & \# \lessdot \mathbf{call} \lessdot \mathbf{han} \lessdot \mathbf{call} \lessdot \underline{\mathbf{call}} \gtrdot \mathbf{exc} \gtrdot \mathbf{call} \doteq \mathbf{ret} \gtrdot \mathbf{ret} \gtrdot \# \\
2 & \# \lessdot \mathbf{call} \lessdot \mathbf{han} \lessdot \underline{\mathbf{call}} \gtrdot \mathbf{exc} \gtrdot \mathbf{call} \doteq \mathbf{ret} \gtrdot \mathbf{ret} \gtrdot \# \\
3 & \# \lessdot \mathbf{call} \lessdot \underline{\mathbf{han}} \doteq \underline{\mathbf{exc}} \gtrdot \mathbf{call} \doteq \mathbf{ret} \gtrdot \mathbf{ret} \gtrdot \# \\
4 & \# \lessdot \mathbf{call} \lessdot \underline{\mathbf{call}} \doteq \underline{\mathbf{ret}} \gtrdot \mathbf{ret} \gtrdot \# \\
5 & \# \lessdot \underline{\mathbf{call}} \doteq \underline{\mathbf{ret}} \gtrdot \# \\
6 & \# \doteq \#
\end{array}
$$

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 $[\mathbf{call}]$ 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). Below we show the structure of w'_{ex} a longer version of w_{ex}, which is an isomorphic representation of its ST as depicted in Fig. 1.

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

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 $q_{\ell+1} = q_k$.

The *OP Max-Automaton* over Σ, M is $\mathcal{A}(\Sigma, M) = (\Sigma, M, \{q\}, \{q\}, \{q\}, \delta_{max})$ where $\delta_{max}(q, q) = q$, and $\delta_{max}(q, c) = q$, $\forall c \in \Sigma$. Each chain has a support in $\mathcal{A}(\Sigma, M)$. 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 M.

In conclusion, given an OP alphabet, the OPM M assigns a unique structure to any compatible string in Σ^*; unlike VPLs, 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 [23].

Operator Precedence ω-Languages. All definitions regarding OPLs are extended to infinite words in the usual way. Given an 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 #. 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 $\gtrdot$. Such chains are called *open chains* and may be simple or composed. A composed open chain may contain both open and closed chains.

We define the class of automata accepting the whole class of ω-OPLs by augmenting Definition 1 with Büchi acceptance condition [22]. Hence, the name Operator Precedence Büchi Automata (OPBA). The semantics of configura-

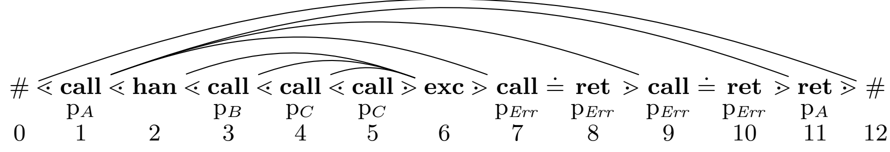


Fig. 2. w'_{ex} as an OP word, with edges showing the χ relation. Normal atomic propositions are below those in $M_{\mathbf{call}}$: p_l means a **call** or a **ret** is related to procedure p_l.

tions, 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 $\mathrm{Inf}(\rho) = \{q \in Q \mid$ there exist infinitely many positions i 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 OPBA $\mathcal{A}$ 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, OPBA 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 OPLs are preserved by ωOPLs.

3 Background: Precedence Oriented Temporal Logic

Here we only describe a fragment of POTL, as not all of its operators are needed for our case study. For the full syntax, see [11]. Given a finite set of atomic propositions AP, $\mathrm{a} \in AP$, and $t \in \{d, u\}$, the syntax of POTL follows:

$$\varphi ::= \mathrm{a} \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc^t \varphi \mid \circleddash^t \varphi \mid \chi_F^t \varphi \mid \chi_P^t \varphi \mid \varphi \,\mathcal{U}_\chi^t\, \varphi \mid \varphi \,\mathcal{S}_\chi^t\, \varphi.$$

The semantics of POTL is based on the *word structure*—also called *OP word* for short—(U, M_{AP}, P), where $U = \{0, 1, \dots, n, n+1\}$ is a set of word positions; $P \colon U \to \mathcal{P}(AP)$ is a function associating each position with the set of atomic propositions holding in it, with $P(0) = P(n+1) = \{\#\}$. M_{AP} is only defined on a subset of AP, and exactly one of such labels may hold in each position. Given $i, j \in U$ and a PR π, we write $i \;\pi\; j$ to say that $a \in P(i)$, $b \in P(j)$ and $a \;\pi\; b$.

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.

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 $\vee$ and $\neg$ have the usual semantics from propositional logic.

The *downward* next and back operators $\bigcirc^d$ and $\circleddash^d$ are true only if the next (resp. current) position is at a lower or equal ST level than the current (resp.

preceding) one; replace 'lower' with 'higher' for the *upward* versions $\bigcirc^u$ and $\circleddash^u$. Formally, $(w,i) \models \bigcirc^d \varphi$ iff $(w, i+1) \models \varphi$ and $i \lessdot (i+1)$ or $i \doteq (i+1)$, and $(w,i) \models \circleddash^d \varphi$ iff $(w, i-1) \models \varphi$, and $(i-1) \lessdot i$ or $(i-1) \doteq i$. Substitute $\gtrdot$ for $\lessdot$ to obtain the semantics for $\bigcirc^u$ and $\circleddash^u$. E.g., we can write $\bigcirc^d \mathbf{call}$ to say that the next position is an inner call (holds in pos. 2, 3, 4 of Fig. 2), $\circleddash^d \mathbf{call}$ 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).

The *chain* next and back operators χ_F^t and χ_P^t, $t \in \{d, u\}$, evaluate their argument resp. 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. Formally, $(w,i) \models \chi_F^d \varphi$ iff there exists $j > i$ 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 $j < i$ 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. 2, $\chi_F^u \mathbf{exc}$ is true in **call** positions whose procedure is terminated by an exception thrown by an inner procedure (e.g. pos. 3 and 4). $\chi_P^u \mathbf{call}$ is true in **exc** statements that terminate at least one procedure other than the one raising it, such as the one in pos. 6. Note that these examples are not meant to be exhaustive: e.g., $\chi_P^u \mathbf{call}$ holds also in position 11, and so on.

The *summary* until $\psi \, \mathcal{U}_\chi^t \, \theta$ (resp. since $\psi \, \mathcal{S}_\chi^t \, \theta$) operator is obtained by inductively applying the $\bigcirc^t$ and χ_F^t (resp. $\circleddash^t$ and χ_P^t) operators. It holds in a position in which either θ holds, or ψ holds together with $\bigcirc^t (\psi \, \mathcal{U}_\chi^t \, \theta)$ (resp. $\circleddash^t (\psi \, \mathcal{S}_\chi^t \, \theta)$) or $\chi_F^t (\psi \, \mathcal{U}_\chi^t \, \theta)$ (resp. $\chi_P^t (\psi \, \mathcal{S}_\chi^t \, \theta)$). 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 $M_{\mathbf{call}}$, 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.

E.g., $\top \, \mathcal{U}_\chi^u \, \mathbf{exc}$ is true in positions contained in the frame of a function terminated by an exception. It is true in pos. 3 of Fig. 2 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 $\top \mathcal{U}_\chi^d \mathbf{exc}$ is true in call positions whose function frame contains **exc**s, such as the one in pos. 1 (with path 1-2-6). $\mathbf{call} \, \mathcal{U}_\chi^d \, (\mathbf{ret} \wedge \mathrm{p}_{Err})$ holds in pos. 1 because of path 1-7-8 and 1-9-10, $(\mathbf{call} \vee \mathbf{exc}) \, \mathcal{S}_\chi^u \, \mathrm{p}_B$ in pos. 7 because of path 3-6-7, and $(\mathbf{call} \vee \mathbf{exc}) \, \mathcal{U}_\chi^u \, \mathbf{ret}$ in 3 because of path 3-6-7-8.

We additionally employ $\top$, $\wedge$, $\implies$ and $\iff$ with the usual semantics form propositional logic. We also use the operators $\Diamond$ and $\Box$ from LTL. They can be expressed in POTL as $\Box \psi := \neg(\top \, \mathcal{U}_\chi^u \, (\top \, \mathcal{U}_\chi^d \, \neg \psi))$ and $\Diamond \psi := \neg \Box \neg \psi$.

4 Model Checking OPA

We model-check POTL through the automata-theoretic procedure introduced in [11]. For any formula φ, we build an OPA (or OPBA) $\mathcal{A}_\varphi$ that only accepts models of φ. Then, given an OPA $\mathcal{A}$ to be checked, we check the product automaton $\mathcal{A}_\varphi \otimes \mathcal{A}$ for emptiness. The product automaton can be computed on-the-fly in a way similar to finite-state automata [22].

To cope with the state-space explosion problem, we propose an *on-the-fly* depth-first explicit-state exploration of the search space. We generate OPA states just before they are visited, and avoid wasting memory and time by generating unreachable states. Other tools [5,19] showed the benefits of this approach, especially when combined with the *early-termination* property, i.e. returning immediately when a counterexample is found. This has no benefits if no accepting state is ever reached, and the entire search space needs to be visited. On the other hand, it speeds up considerably cases when there is a counterexample.

Reachability. OPA are equipped with a stack, which must be considered when exploring the search space. Given an OP alphabet (Σ, M), let $\mathcal{A} = (\Sigma, M, Q, I, F, \delta)$ be an OPA, and $\Gamma = \Sigma \times Q \times \{\bot\}$ be the set of stack symbols.

Definition 3. *A semi-configuration of* $\mathcal{A}$ *is an element of* $\mathcal{C} = Q \times \Gamma$.

Definition 4. *The reachability relation is defined as* $\mathcal{R}_{reach} \subseteq \mathcal{C} \times \mathcal{C} \times \Sigma$ *so that, for any* $p, q \in Q$*, look-ahead* $a \in \Sigma$*, and* $g_0, g_1 \in \Gamma$*, we have*

$$\mathcal{R}_{reach}(p, g_0, q, g_1, a) \text{ iff } \langle xay,\ p,\ g_0\rangle \vdash^* \langle ay,\ q,\ g_1\gamma\rangle,$$

for some $x, y \in \Sigma^*$, $\gamma \in \Gamma^*$.

To determine the (non) emptiness of $L(\mathcal{A})$, we must establish whether there exist some $q_i \in I$, $q_f \in F$ such that $(q_i, \bot, q_f, \bot, \#) \in \mathcal{R}_{reach}$.

Algorithm 1 solves the reachability problem for OPA by adapting a DFS to the use of summaries, similarly to [5]. It consists of an on-the-fly, early-terminating exploration to check if a given set $Q_R \times \Gamma_R$ of target semiconfigurations is reachable in an OPA $\mathcal{A}$. Function Reach receives as its arguments a state $q \in Q$, a stack symbol $g \in \Gamma$, a character $c \in \Sigma$, and a look-ahead $\ell \in \Sigma \cup \{*\}$. If $\ell = *$, then the look-ahead may be any character in Σ. The algorithm searches the transition graph and stops when it reaches a semiconfiguration $(q, g) \in Q_R \times \Gamma_R$. To solve the emptiness problem for OPA, we pose $Q_R = F$ and $\Gamma_R = \{\bot\}$, and call Reach$(q, \bot, \#, *)$ for each $q \in I$. Each call to Reach has worst-case time complexity $O(|\delta||\delta_{push}|^2|\Sigma|)$ and space complexity $O(|\delta||\delta_{push}||\Sigma|)$. Note that the above bounds are reached only if the whole OPA is visited, i.e. when $L(\mathcal{A})$ is empty. Also, if Σ contains sets of atomic propositions, we consider only those on which the OPM is defined. E.g., with $M_{\mathbf{call}}$ we use only elements of $\Sigma_{\mathbf{call}}$ as look-aheads, and $|\Sigma_{\mathbf{call}}|$ is a small constant.

Summary Transitions. Suppose we are in a semiconfiguration (q_l, g) which can be followed by a push transition (q_l, b, r). This transition is the beginning of a chain support (let it be σ) that starts with symbol b. If we apply the reachability algorithm recursively, we may meet the push transition (q_l, b, r) again, which would lead us to the beginning of σ. To avoid a never-ending computation, we cannot follow it. At the same time, a semiconfiguration $s_p = (q_p, g_p)$ may be reachable such that q_p has a pop transition (q_p, q_l, q_r) which completes σ. s_p may not have been visited yet, due to the depth-first nature of the search, although

it would allow us to continue the visit without getting stuck. We need to find a way to “suspend” the search and resume it later.

As a solution, we use a global variable (called *SupportStarts*) where we store semiconfigurations corresponding to the beginning of a chain support σ. While visiting σ, the algorithm matches all saved semiconfigurations for σ trying to perform a pop transition, thus completing σ and resuming the suspended explorations.

To establish if a saved semiconfiguration (q_l^{cand}, g^{cand}) is valid for σ, it must make sure that $smb(g^{cand})$ yields precedence to the first symbol read by σ (b in our example). This symbol is carried by parameter c in the Reach algorithm. Likewise, parameter l is used to restrict the set of possible characters to read after a pop transition only to those with which the input part of the stack symbol before popping is in a $\gtrdot$ relation.

This solution allows us to suspend the search safely when we encounter the beginning of a support in an already-visited semiconfiguration: if a way to go beyond exists, it will be explored. Therefore, we introduce *summary* transitions, which connect the first state of a chain support (q_l in our example) to the corresponding last state (q_r).

The algorithm also stores in *SupportEnds* a semiconfiguration whenever it exits a chain support. Thus, when it finds in a semiconfiguration the beginning of a support that has already been visited, it uses this pre-computed information to jump to the corresponding pop move directly.

Algorithm 1 OPA semi-configuration reachability

1: $(\Sigma, M_\Sigma, Q, I, F, (\delta_{push}, \delta_{shift}, \delta_{pop})) := \mathcal{A}$
2: $V := \mathit{SupportStarts} := \mathit{SupportEnds} := \emptyset$
3: **function** Reach(q, g, c, ℓ)
4: **if** $(q, g, \ell) \in V \lor (q, g, *) \in V$ **then return** false
5: $V := V \cup (q, g, \ell)$
6: **if** $q \in Q_R \land g \in \Gamma_R$ **then return** true
7: $a := smb(g)$
8: **for all** $(q, b, p) \in \delta_{push}$ s.t. $a \lessdot b \land (b = \ell \lor \ell = *)$ **do**
9: $\mathit{SupportStarts} := \mathit{SupportStarts} \cup \{(q, g, c)\}$
10: **if** Reach$(p, [b, q], b, *)$ **then return** true
11: **for all** $(s, q, c', \ell') \in \mathit{SupportEnds}$ s.t. $a \lessdot c'$ **do**
12: **if** Reach(s, g, c, ℓ') **then return** true
13: **if** $g \neq \bot$ **then**
14: $[a, r] := g$
15: **for all** $(q, b, p) \in \delta_{shift}$ s.t. $a \doteq b \land (b = \ell \lor \ell = *)$ **do**
16: **if** Reach$(p, [b, r], c, *)$ **then return** true
17: **for all** $(q, r, p) \in \delta_{pop}, b \in \Sigma \cup \{\#\}$ s.t. $a \gtrdot b \land (b = \ell \lor \ell = *)$ **do**
18: $\mathit{SupportEnds} := \mathit{SupportEnds} \cup \{(p, r, c, b)\}$
19: **for all** $(r, g', c') \in \mathit{SupportStarts}$ s.t. $smb(g') \lessdot c$ **do**
20: **if** Reach(p, g', c', b) **then return** true
21: **return** false

Fair-Cycle Detection. We propose an adaptation of the reachability algorithm to the *fair-cycle detection* problem. Given an OPBA $\mathcal{A}^\omega_\varphi = \langle \mathcal{P}(AP), M_{AP}, Q_\omega, I, F, \delta\rangle$, algorithm Fair-cycle-detect models the search space as a graph where vertices are semiconfigurations and edges are OPBA transitions, and looks for *fair* cycles, i.e. loops containing a state $q_\omega \in F$. To preserve the early-termination and on-the-fly properties, we follow an online approach: we represent and update Strongly Connected Components (SCCs) using an incremental algorithm while the Reach procedure discovers new portions of the graph. The algorithm we chose is a path-based depth-first search due to H. Gabow [16]. This algorithm finds SCCs and updates them dynamically in time linear on the number of graph nodes. It allows us to stop the search as soon as a non-trivial fair SCC is found. Otherwise, at the end it outputs the SCCs graph. It represents SCCs with simple auxiliary data structures such as stacks and arrays (hence the name *list-based* used in [16]) without contracting nodes in the actual graph. For performance reasons, we slightly modify our implementation to perform contractions.

Combining the Reach and Gabow routines into Fair-cycle-detect is a crucial issue. A summary edge (corresponding to a summary transition) is added to the graph when we encounter a pop transition at the end of a chain support. Thus, the edge may be in a completely different part of the graph with respect to the current node. If followed, it breaks the depth-first property, which is required by the Gabow algorithm. Instead of restarting it for every pop transition, our solution is to save all the summary edges and process them later. Fair-cycle-detect is then divided into two phases:

- a **search phase** when we discover new edges, following OPBA transitions, starting from the current initial semiconfigurations. If we find a summary edge, we do not feed it to the Gabow routine, but store it in the set *Summ*.
- a **collapse phase** when we add to the graph the summary edges in *Summ*, and run only the dynamic Gabow routine on it.

At the end of the collapse phase, we resume the exploration from semiconfigurations corresponding to the discovered summary transitions in *Summ*. If the set *Summ* is empty at the end of a search phase, there is no reachable fair cycle, and the algorithm terminates.

5 Modeling Procedural Programs

We use a simple procedural programming language with exceptions called MiniProc, which only admits Boolean variables. Its syntax is shown in Fig. 3.

A program starts with a variable declaration, which must include all variables used in the program. Then, a sequence of functions are defined, the first one being the entry-point to the program. Function bodies consist of semicolon-separated statements. Assignments, while loops and ifs have the usual semantics. The try-catch statement executes the catch block whenever an exception is thrown by any statement in the try block (or any function it calls). Exceptions are thrown by the `throw` statement, and they are not typed (i.e., there is no way to distinguish

```
PROGRAM = [DECLS] FUNCTION [FUNCTION ...]
DECLS = var IDENTIFIER [, IDENTIFIER ...] ;
FUNCTION = IDENTIFIER () { STMT; [STMT; ...] }
STMT = IDENTIFIER := BEXPR
      | while (BEXPR) { [STMT; ...] }
      | if (BEXPR) { [STMT; ...] } else { [STMT; ...] }
      | try { [STMT; ...] } catch { [STMT; ...] }
      | IDENTIFIER()
      | throw
BEXPR = BEXPR && BDISJ | BDISJ
BDISJ = BDISJ || BTERM | BTERM
BTERM = !BTERM | (BEXPR) | IDENTIFIER | true | false
```

```
program:
var foo;
pa() {
  foo = true;
  try { pb(); }
  catch { pc(); }
}
pb() {
  if (foo) { throw; }
  else {}
}
pc() { }
```

Fig. 3. MiniProc syntax (left) and a MiniProc program (right). Non-terminals are uppercase, and keywords lowercase. Parts in square brackets are optional, and ellipses mean that the enclosing group can be repeated zero or more times. An `IDENTIFIER` is any sequence of letters, numbers, or characters '`.`', '`:`' and '_', starting with a letter or an underscore.

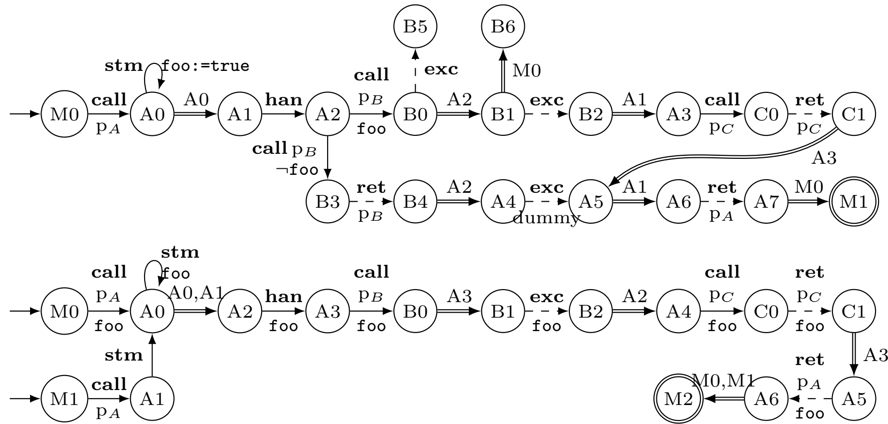


Fig. 4. Extended OPA (top) and OPA (bottom) generated from the code of Fig. 3.

different kinds of exceptions). Functions can be called by prepending their name to the `()` token (they do not admit arguments, as all variables are global). Since all variables are Boolean, expressions can be composed with the logical and (`&&`), or (`||`) and negation (`!`) operators.

OPA and OPBA semantically equivalent to a MiniProc program can be generated automatically, both based on OPM $M_{\mathbf{call}}$. We illustrate their construction through examples. First, an *extended* OPA is generated, in which every state corresponds to some program state, and transitions can be labeled with Boolean expression guards that must be true for them to be performed, or variable assignments. Fig. 4 shows the extended OPA from the code in Fig. 3. The stack semantics of the two models coincide: a symbol is pushed for every function call, and popped after the corresponding return (or exception). Handlers are paired to the exception they catch by a shift move updating the same symbol; a dummy exception is placed after the try body to uninstall the handler. The

model-checking procedures of Section 4 do not take guards into account, so the extended OPA must be transformed into a normal one. This is done by enumerating all possible Boolean variable assignments for each state, leaving only those that are actually reachable. The resulting OPA for our example is in Fig. 4.

The last part of the OPA generation leads to a worst-case model size exponential in the number of variables. However, it performs well in most practical cases, since only feasible states are generated.

6 Experiments

We show how to use POMC to verify programs with exceptions with some experiments on real code for a well-known recursive algorithm, Quicksort. The properties we want to verify on an implementation of this algorithm are:

termination: the program always terminates for any input array.
correctness: any input array is correctly sorted at the end of the program.

We begin with two C programs packaged with the first version of the MOPED model checker [21,12,14]. Then, we move to a refinement which is targeted specifically to POMC. We call it *semi-safe* Java Quicksort, because of how it handles possible `NullPointerException`s. This experiment cannot be conducted on MOPED or tools such as VERA [5] or BEBOP [8], since their models require a matching single return statement for every function call. On the other hand, exceptions pop an indefinite amount of function frames on the stack until they meet a handler. With such formalisms, the automaton would be forced to read as many input symbols as the amount of popped function frames, thus consuming a portion of code from the `catch` block. In the following, the comparison of the three examples will allow the reader to grasp the greater expressive power of POMC. The experimental results and their execution times are reported in Table 1.

Buggy C QuickSort. The first one, called *Buggy Quicksort*, and contained in file `quicksort_error.pds`, has been proven to run into a infinite loop for certain values of the input array, violating the termination property. When modeling it with MiniProc, we consider an array of 2 values for performance reasons: `a[left]` and `a[right]`. We abstract away from the actual content of the array and replace all comparisons with non-deterministic choices, to get a smaller and better performing model. We use Boolean variables to represent the inequalities $(<, =, >)$ between local integer variables in the program. Then, we use the fair-cycle detection module of POMC to detect the mentioned never-ending loop. The problem can be expressed in terms of checking whether the `main` procedure always reaches the **ret** statement ($\chi_F^u(\mathbf{ret} \wedge \mathrm{main})$, experiment **B.2**). With this formula, we force the first position (the one reading **call** main) to be in the χ relation with the one reading the corresponding **ret** main. Anyway, **call** is in the $\doteq$ or $\gtrdot$ (since we are imposing the upward variant) precedence relation only with

```
main() {
  // list may contain null elements
  try {
    qs();
  } catch (){
    parseList();
    // null elements removed
    qs();
  }
}
```

```
qs() {
  . . .
  accessValues();
  . . .
}
parseList() {
  hasParsed = true;
}
```

```
accessValues() {
  if (*) {
    throw;
  } else {}
}
```

Fig. 5. A portion of the semi-safe Java Quicksort model in MiniProc.

ret and **exc** (see Figure 1), and no exception is thrown in this model, so **exc** cannot be encountered. Therefore, we could simply use the formula $\chi_F^u \top$. However, the latter would be less clear. Moreover, the same reasoning does not hold for the third experiment: it contains exceptions. Therefore, we opt for the former. As expected, POMC returns False. Note that we can verify the termination property also with a simple LTL formula (**B.1**). Since termination is not guaranteed, it's meaningless to investigate the correctness for this implementation.

Correct C Quicksort. The implementation of file `quicksort_correct.pds` is known to satisfy both the termination and correctness properties. When modeling it with MiniProc, again by considering an array of 2 values, we introduce two boolean variables to indicate the relation between `a[left]` and `a[right]` explicitly. They are `aleftGTaright` and `aleftEQaright`, which respectively mean `a[left] > a[right]` and `a[left] = a[right]`. `a[left] < a[right]` is indicated by the expression `!aleftGTaright && !aleftEQright`.

First, we check for termination with the same formulas as in Buggy Quicksort (experiments **C.1** and **C.2**). Coherently with the early-termination property, the execution time is much greater in this case because the formulas hold. Then, we prove the correctness of the implementation. The algorithm is supposed to sort the array in ascending order, so we verify that `!aleftGTaright` holds at the end of the execution. There are two ways to state this. With LTL, we impose that sooner or later `!aleftGTaright` will hold forever (**C.3**). With POTL, we impose that the first position (the one which reads **call** main) is in the χ relation with a position where `!aleftGTaright` holds, using the upward variant of the operator. As introduced in the previous example, this position is necessarily the one reading the `return` statement. Therefore, experiments **C.4** and **C.6** equally verify the correctness of this implementation and both imply also the termination property. Inserting **ret** main is superfluous. Finally, **C.5** is meant to verify both correctness and termination together, but with a simple LTL formula.

Semi-safe Java Quicksort. We consider the case of a Java implementation where the elements of the Array are of a non-primitive type. Since Java does not enforce *void-* (or *null-*) *safety* [24], accessing array elements may lead to a `NullPointerException` at runtime. A *semi-safe* solution is to:

– First, call the Quicksort procedure inside a try-catch construct, to handle potential exceptions.

- When the first null element is encountered and an exception is raised, parse the array in the catch body to remove all null elements.
- Last, call again the Quicksort procedure inside the catch body. This is potentially unsafe because the call is not contained in a try-catch construct. Thus, if accessing an element raised an exception, there would be no matching catch to handle it, and the `main` function would terminate abnormally. However, we know that it should not happen because of the previous processing of the array, hence the name *semi-safe*. Overall termination and correctness are not guaranteed *a priori*. They depend on the correctness of the parsing function which ensures void safety, thus preventing the throwing of non handled exceptions.

We model the entire procedure as a MiniProc program, by still considering a two-elements array. A sketch is reported in Fig. 5. We hide the Quicksort implementation with dots to highlight the exception-handling constructs. Note how the program strictly resembles real Java code. We treat the parsing function as a black box: we only use the Boolean variable `hasParsed` to indicate that the array has been processed to remove null elements. We introduce function `accessValues` to represent data access.

Firstly, we note that checking general termination (**S.1** and **S.2**) and correctness (**S.3** and **S.4**) returns False in all cases. As a remark, since exceptions are involved, here **ret** main is required in the formula of experiment **S.2**. Likewise, $\chi_F^u(\neg \mathrm{aleftGTaright})$ (S.4) does not imply the termination of the `main` procedure anymore. With the use of the upward variant it imposes that at the end of program, no matter how it terminates, the array is correctly sorted.

Secondly, we examine the program-handling of the potential null-pointer exceptions. POTL can easily express properties related to exception handling [23]. 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 an **exc** in which ψ holds. So, $\square(\mathbf{call} \wedge \rho \wedge \mathit{CallThr}(\top) \implies \mathit{CallThr}(\theta))$ means that if precondition ρ holds when a procedure is called, 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 (or basic) exception safety* [1], and *strong exception safety* if ρ and θ express particular states of the class instance. Alternatively, postconditions may regard the type of exception which has occurred. The *no-throw guarantee* can be stated with $\square(\mathbf{call} \wedge \mathrm{p}_A \implies \neg \mathit{CallThr}(\top))$, meaning procedure p_A is never interrupted by an exception. To begin with, we verify whether procedures `main` and `qs` satisfy the *no-throw guarantee* with experiments **S.5** and **S.6**: the result is False. Therefore, we inquire the conditions that lead to the potentially raised exceptions. The formula $\square(\mathbf{call} \wedge \mathrm{main} \wedge \mathit{CallThr}(\top) \implies \mathit{CallThr}(\mathrm{hasParsed}))$ specifies an exception-safety property meaning that, whenever a call to the function `main` is terminated by an exception, the array list has been parsed to ensure void safety. The property can be slightly modified into

experiment **S.7**, which is verified. A second exception-safety property (**S.8**) verifies whether, in the case the `main` procedure is interrupted by an exception, correctness holds after the interruption. Unfortunately, the result is False.

However, the *stack inspection* property of experiment **S.9** holds. It means that every time the program accesses array values, either: i) there is a handler on the stack to handle a potential exception, or ii) we have already processed the array to remove null elements, thus guaranteeing void safety.

Lastly, we prove the *conditional* termination of this implementation (**S.10** and **S.11**), meaning that either the program terminates or an exception is raised after the parsing function has been called, indicating a bug in the parsing function itself. Likewise, we prove the *conditional* correctness (**S.12** and **S.13**), i.e. that either the array is correctly sorted at the end of the execution or an exception is raised after the parsing function has been called. Formula **S.14** verifies both conditional termination and correctness together.

6.1 Discussion

A limitation of the Case Study at hand is represented by the fact that all experiments deal with Quicksort implementations on arrays of only 2 cells. Indeed, it would be interesting to analyze how the tool's performances scale to bigger models. This is hindered by our tool's current lack of automatic abstraction and modeling techniques for real-world programs. For the time being, the only feasible approach is to model by hand all the possible execution traces, which becomes intractable for large arrays. We leave this task to future work. However, the preliminary works on arrays of 3 cells confirms our theoretical results that the latency is dominated by the formula (and especially formula size), and not by the model under verification.

7 Conclusions

We presented efficient algorithms for reachability and fair-cycle detection for OPA and OPBA. We implemented them in the POMC tool, together with a user-friendly DSL (MiniProc) which allows to model procedural code with exceptions. We reported on a case study on the Quicksort algorithm to show the suitability of POMC for the verification of programs with exceptions. As future work, we plan to investigate the possibility of using POMC to verify properties on real-world programming languages through suitable automated abstractions, such as iterative abstraction refinement techniques. As discussed in Section 6.1, these techniques could address the quest for investigating the tool's performance scaling.

Table 1. Results of verification of the Buggy Quicksort model (2259 OPBA states) (formulas **B.1** and **B.2**), the Correct Quicksort model (83980 OPBA states) (experiments from **C.1** to **C.6**), and the Semi Safe Correct Quicksort model (188456 OPBA states) (formulas from **S.1** to **S.14**). The experiments have been run on a server with a 2.0 GHz AMD CPU and 500 GB of RAM.

#	Formula	Time (s)	Result
B.1	$\Diamond(\mathbf{ret} \land \mathrm{main})$	0.067	False
B.2	$\chi_F^u(\mathbf{ret} \land \mathrm{main})$	1.011	False
C.1	$\Diamond(\mathbf{ret} \land \mathrm{main})$	36.3	True
C.2	$\chi_F^u(\mathbf{ret} \land \mathrm{main})$	101.8	True
C.3	$\Diamond(\Box\neg\mathrm{aleftGTaright})$	66.3	True
C.4	$\chi_F^u(\neg\mathrm{aleftGTaright})$	123.2	True
C.5	$\Diamond(\mathbf{ret} \land \mathrm{main} \land \neg\mathrm{aleftGTaright})$	49.0	True
C.6	$\chi_F^u(\mathbf{ret} \land \mathrm{main} \land \neg\mathrm{aleftGTaright})$	222.8	True
S.1	$\Diamond(\mathbf{ret} \land \mathrm{main})$	284.4	False
S.2	$\chi_F^u(\mathbf{ret} \land \mathrm{main})$	289.0	False
S.3	$\Diamond(\Box\neg\mathrm{aleftGTaright})$	279.5	False
S.4	$\chi_F^u(\neg\mathrm{aleftGTaright})$	276.5	False
S.5	$\Box((\mathbf{call} \land \mathrm{main}) \implies \neg(\bigcirc^u\mathbf{exc} \lor \chi_F^u\mathbf{exc}))$	245.9	False
S.6	$\Box((\mathbf{call} \land \mathrm{qs}) \implies \neg(\bigcirc^u\mathbf{exc} \lor \chi_F^u\mathbf{exc}))$	246.9	False
S.7	$(\bigcirc^u\mathbf{exc} \lor \chi_F^u\mathbf{exc}) \implies (\bigcirc^u\mathbf{exc} \land \mathrm{hasParsed}) \lor (\chi_F^u\mathbf{exc} \land \mathrm{hasParsed})$	19617.0	True
S.8	$(\bigcirc^u\mathbf{exc} \lor \chi_F^u\mathbf{exc}) \implies (\bigcirc^u\mathbf{exc} \land \neg\mathrm{aleftGTaright}) \lor (\chi_F^u\mathbf{exc} \land \neg\mathrm{aleftGTaright})$	387.2	False
S.9	$\Box((\mathbf{call} \land \mathrm{accessValues}) \implies \mathrm{hasParsed} \lor (\top\ \mathcal{S}_\chi^d\ \mathbf{han}))$	446.2	True
S.10	$(\Diamond(\mathbf{ret} \land \mathrm{main})) \lor (\chi_F^u(\mathbf{exc} \land \mathrm{hasParsed}))$	1124.0	True
S.11	$(\chi_F^u(\mathbf{ret} \land \mathrm{main})) \lor (\chi_F^u(\mathbf{exc} \land \mathrm{hasParsed}))$	12809.0	True
S.12	$(\Diamond(\Box\neg\mathrm{aleftGTaright})) \lor (\chi_F^u(\mathbf{exc} \land \mathrm{hasParsed}))$	1615.0	True
S.13	$(\chi_F^u(\neg\mathrm{aleftGTaright})) \lor (\chi_F^u(\mathbf{exc} \land \mathrm{hasParsed}))$	12736.0	True
S.14	$(\Diamond(\mathbf{ret} \land \mathrm{main} \land \neg\mathrm{aleftGTaright})) \lor \chi_F^u(\mathbf{exc} \land \mathrm{hasParsed})$	2247.0	True

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). https://doi.org/10.2168/LMCS-4(4:11)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). https://doi.org/10.1007/978-3-319-10575-8_17
5. 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
6. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: TACAS 2004. pp. 467–481. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_35
7. Alur, R., Madhusudan, P.: Visibly Pushdown Languages. In: ACM STOC (2004)
8. 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). https://doi.org/10.1007/10722468_7
9. Ball, T., Rajamani, S.K.: The SLAM toolkit. In: CAV '01. LNCS, vol. 2102, pp. 260–264. Springer (2001). https://doi.org/10.1007/3-540-44585-4_25
10. 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
11. Chiari, M., Mandrioli, D., Pradella, M.: Model-checking structured context-free languages. In: CAV '21. LNCS, vol. 12760, pp. 387—410. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_18
12. 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)
13. Esparza, J., Kučera, A., Schwoon, S.: Model checking LTL with regular valuations for pushdown systems. Information and Computation **186**(2), 355–376 (2003). https://doi.org/10.1016/S0890-5401(03)00139-1
14. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: Proceedings of the 13th International Conference on Computer Aided Verification. p. 324–336. CAV '01, Springer-Verlag (2001)
15. Floyd, R.W.: Syntactic Analysis and Operator Precedence. JACM **10**(3), 316–333 (1963). https://doi.org/10.1145/321172.321179
16. Gabow, H.N.: Path-based depth-first search for strong and biconnected components. Information Processing Letters **74**(3), 107–114 (2000). https://doi.org/10.1016/S0020-0190(00)00051-X
17. Grune, D., Jacobs, C.J.: Parsing techniques: a practical guide. Springer, New York (2008). https://doi.org/10.1007/978-0-387-68954-8
18. Harrison, M.A.: Introduction to Formal Language Theory. Addison Wesley (1978)
19. Holzmann, G.: The model checker SPIN. IEEE Transactions on Software Engineering **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521

20. 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
21. Kiefer, S., Schwoon, S., Suwimonteerabuth, D.: Moped. http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/moped/
22. 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
23. 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
24. Meyer, B.: Attached types and their application to three open problems of object-oriented programming. In: ECOOP 2005 - Object-Oriented Programming. pp. 1–32. Springer (2005)