A Model Checker for Operator Precedence Languages*

MICHELE CHIARI, TU Wien, Austria
DINO MANDRIOLI, DEIB, Politecnico di Milano, Italy
FRANCESCO PONTIGGIA, TU Wien, Austria
MATTEO PRADELLA, DEIB, Politecnico di Milano and IEIIT, Consiglio Nazionale delle Ricerche, Italy

The problem of extending model checking from finite state machines to procedural programs has fostered much research toward 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, Precedence Oriented Temporal Logic (POTL) has been introduced to specify and prove properties of programs coded trough an Operator Precedence Language (OPL). POTL is complete w.r.t. the FO restriction of the MSO logic previously defined as a logic fully equivalent to OPL. POTL increases NWTL's expressive power in a perfectly parallel way as OPLs are more powerful that nested words.

In this paper we produce a model checker, named POMC, for OPL programs to prove properties expressed in POTL. To the best of our knowledge POMC is the first implemented and openly available model checker for proving tree-structured properties of recursive procedural programs. We also report on the experimental evaluation we performed on POMC on a nontrivial benchmark.

CCS Concepts: • **Software and its engineering** → *Formal software verification*; • **Theory of computation** → **Modal and temporal logics**; **Verification by model checking**.

Additional Key Words and Phrases: Linear Temporal Logic, Precedence Oriented Temporal Logic, Operator Precedence Languages, Model Checking, Visibly Pushdown Languages, Input-Driven Languages

1 INTRODUCTION

Model Checking is a well-established technique for the analysis of both hardware and software systems. In particular, the specification of *regular* properties has been extensively studied. To this regard, Linear Temporal Logic (LTL) has been introduced to express a large variety of safety and liveness linear time properties.

Operational models for the system under verification often paired with LTL specifications are Transition Systems (TSs) and Finite-State Automata (FSAs) (generally Büchi automata) [12]. Frameworks based on these formalisms, such as SPIN [50], affirmed themselves due to their ease in reasoning, the conciseness of their logics with respect to the automata representation, and the efficiency of the model checking algorithms when implemented in practice.

However, when focusing on procedural programs, the presence of the stack of activation records constitutes a non-negligible feature that FSAs cannot model. Therefore, more adequate abstract models of procedural programs

*This paper is a revised and extended version of [25] and [73].

Authors' addresses: Michele Chiari, michele.chiari@polimi.it, TU Wien, Treitlstraße, 3, Wien, Austria, 1040; Dino Mandrioli, dino.mandrioli@polimi.it, DEIB, Politecnico di Milano, Via Ponzio 34/5, Milano, Italy, 20133; Francesco Pontiggia, francesco.pontiggia@tuwien.ac.at, TU Wien, Treitlstraße, 3, Wien, Austria, 1040; Matteo Pradella, matteo.pradella@polimi.it, DEIB, Politecnico di Milano and IEIIT, Consiglio Nazionale delle Ricerche, Via Ponzio 34/5, Milano, Italy, 20133.

0164-0925/2023/8-ART $15.00
https://doi.org/10.1145/3608443

are represented by Boolean Programs [13], Pushdown Systems [18, 38] and Recursive State Machines [3]. For all these stack-based formalisms problems such as state and configuration reachability as well as the more complex model checking of *regular specifications* have been thoroughly studied [3, 4, 18, 20, 34, 38, 42, 55, 70, 80]. They are able to mock up many relevant behaviors of real-world programs, expressible by means of Context-Free Languages (CFLs), rather than regular languages. Unfortunately, the set of properties expressible with LTL corresponds only to the First-Order Logic (FOL) definable fragment of regular languages. Hence, this logic is not suitable to formulate constraints on the managing of the procedure stack.

Example interesting properties include Hoare-style pre/post conditions on procedure calls and returns, and stack-inspection properties at a particular execution point [51]. To fill the gap, some efforts have been made to define logics based on subclasses of CFLs. These subclasses, while being strictly more expressive than Regular Languages, retain the same properties that allow to use them in automata-theoretic model checking. They are informally defined as Structured CFLs [59], because the structure of the syntax tree of a sentence is built in the sentence itself, and in many cases immediately visible.

A coherent approach in this direction is based on Visibly Pushdown Languages (VPLs) [9], a.k.a Input-Driven Languages [64]. Two derived logics, namely CaRet [7] and its FO-complete successor Nested Words Temporal Logic (NWTL) [2], allow to reason about program traces structured as Nested Words [10]. Such execution traces consist of the usual LTL linear ordering of events augmented with a matching relation between procedure calls and returns. In this regard, they are the first logics equipped with temporal modalities explicitly referring to the nested structure of CFLs [4]. Through them, it is possible to express requirements regarding the mentioned context-free behaviors [4]. System models are represented by Visibly Pushdown Automata (VPAs), the automata class of VPLs. To complete the view, a μ-calculus based on VPLs extends model checking to branching-time semantics in [6], while [19] introduces a temporal logic capturing the whole class of VPLs.

On the practical side, the work on tools is not as rich as the theoretical one. Libraries such as VPAlib [66], VPAchecker [76], OpenNWA [33] and Symbolic Automata [31] only implement operations such as union, intersection, universality/inclusion/emptiness check for VPAs or Nested Words Automata (NWAs), but have no model checking capabilities. PAL [22] uses nested-word based monitors to express program specifications, and a tool based on BLAST [47] implements its runtime monitoring and model checking. PAL follows the paradigm of program monitors, and is not —strictly speaking— a temporal logic. [68, 69] 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.

VPLs have some theoretical limitations as well. While being more expressive than Parenthesis Languages [62], the matching relation is essentially constrained to be one-to-one. As a consequence, they fail to reason about those behaviors in which a single event must be put into relation with multiple ones. Such behaviors occur in programming languages that feature exceptions and exception handling, or in programs with control flow operators that allow the explicit management (or *reification*) of the current continuation (such as `call/cc`). In such contexts, a single event may cause the termination (or re-instantiation) of multiple procedures on the stack, by causing the pop (or the push) of the corresponding activation frames.

To model-check such behaviors the more powerful formalisms of *Operator Precedence Languages (OPLs)* and related logics have been proposed. OPLs —and their generating Operator Precedence Grammars (OPGs)— are a historical subfamily of CFLs invented by Robert Floyd [39] to support efficient parsing. They are general enough to formalize most syntactic constructs of mainstream programming languages [16, 32] including those mentioned above that cannot be expressed as VPLs. In fact, OPLs strictly include VPLs but, despite the increased expressive power, they are still closed under Boolean operations, concatenation, and Kleene *; thus, inclusion is decidable for them (since emptiness is decidable for any CFL) [29, 59]. Alongside, a class of automata accepting the OPL family has been given, namely Operator Precedence Automata (OPAs), together with their "ω-counterpart" i.e., Operator Precedence Büchi Automata (ωOPBAs) accepting infinite (or ω-) Operator Precedence words [58].

On the logic side, a Monadic Second-Order (MSO) logic equivalent to OPAs and OPGs has been defined [58] and on its basis the logic called Precedence Oriented Temporal Logic (POTL) has been presented which is complete w.r.t. the First-Order (FO) restriction of the MSO logic [27][1]; thus, POTL gains in expressive power w.r.t. NWTL in a perfectly parallel way as OPLs gain over VPLs.

Consider also that recently FO-definability of OPLs has been proved equivalent to the aperiodicity —or non-counting— property [60] as it happens for regular languages [63], a non-trivial and somewhat surprising result since the same does not hold for tree-languages [49, 78]. Whereas in the realm of finite state machines aperiodicity is not enjoyed in many practical cases —for instance, various hardware devices are counters modulo some integer $k > 1$— it is quite unusual to find counting features in normal programming languages: thus, POTL has a potential application breadth even larger than LTL has for regular languages.

This paper offers the final step needed to *model-check structured programs against structured properties*. Its main contributions are:

- A tableaux-construction procedure for model checking POTL, which yields nondeterministic automata. Although the technicalities of the construction are much more involved than the corresponding construction for LTL and even that for NWTL, its size is at most singly exponential in the formula's length, and is thus not asymptotically greater than that of LTL, CaRet and NWTL.
- An implementation of this procedure in a tool called Precedence Oriented Model Checker (POMC) [23]. POMC is able to build the corresponding OPA (or ωOPBA) of a POTL formula, and is equipped with both a Reachability (for OPAs) and a Fair Cycle Detection algorithm (for ωOPBAs) module; hence constitutes a full explicit-state model checker for POTL. To the best of our knowledge, POMC is the only publicly-available tool for temporal logics capable of expressing context-free properties. For user-friendliness, POMC is equipped with a domain-specific language called MiniProc. Programs written in MiniProc are then internally translated into the automaton representation through an operational semantics.
- An extensive evaluation of the complexity of the model checking algorithm in practice, to assess the trade-off between the greater expressive power of OPL-based model checking and its complexity.

An earlier version of POMC has been awarded the *Functional*, *Reusable* and *Available* badges by the CAV 2021 Artifact Evaluation Committee.

The paper is organized as follows: Section 2 recalls some background on OPLs; POTL is introduced in Section 3; the model checking procedure on finite words is given in Section 4, while the case of ω-words is studied in Section 5; Section 6 supplies some implementation details of POMC; Section 7 presents the benchmark adopted to evaluate the features and performances of POMC, the main results of experiments carried over, and a —qualitative more than quantitative— comparison with related tools. Finally, Section 8 summarizes our results in the context of previous literature and delineates some potential future works. To make the reading more fluid and to help focusing on the essentials a few technical details have been postponed to suitable appendices.

2 OPERATOR PRECEDENCE LANGUAGES

Operator Precedence Languages (OPLs) were originally defined through their generating grammars [39]: *Operator Precedence Grammars (OPGs)* are a special class of Context-Free Grammars (CFGs) in *operator normal form* —i.e., grammars in which right-hand sides (rhs's) of production rules contain no consecutive non-terminals[2]—. As a consequence, in the Syntax Trees (STs) generated by such grammars the children of any node never exhibit two consecutive internal nodes.

The distinguishing feature of OPGs is that they define three *Precedence Relations (PRs)* between pairs of input symbols which drive the deterministic parsing and therefore the construction of a unique ST, if any, associated

[1] On the contrary, an earlier temporal logic for OPLs [25] is not FO-complete.
[2] Every CFG can be effectively transformed into an equivalent one in operator form [46].

	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. The OPM $M_{\mathbf{call}}$.

with an input string. For this reason we consider OPLs a kind of *input-driven languages*, but larger then the original ones by K. Mehlhorn [64] (later known as VPLs [10]). The three PRs are denoted by the symbols $\lessdot, \doteq, \gtrdot$ and are respectively named *yields precedence, equal in precedence*, and *takes precedence*. They graphically resemble the traditional arithmetic relations but do not share their typical ordering and equivalence properties; we kept them for "historical reasons", but we recommend the reader not to be confused by the similarity.

Intuitively, given two input characters a, b belonging to a grammar's *terminal alphabet*, separated by at most one non-terminal, $a \lessdot b$ iff, in some grammar derivation, b is the first terminal character of a grammar's rhs following a whether or not the grammar rule contains a non-terminal character before b (for this reasons we also say that non-terminal characters are "transparent" in OPL parsing); $a \doteq b$ iff a and b occur consecutively in some rhs, possibly separated by one non-terminal; $a \gtrdot b$ iff a is the last terminal in a rhs —whether followed or not by a non-terminal—, and b follows that rhs in some derivation. The following example provides a first intuition of how a set of *unique* PRs drives the parsing of a string of terminal characters in a deterministic way; subsequently the above concepts are formalized.

Example 2.1. Consider the alphabet of terminal symbols $\Sigma_{\mathbf{call}} = \{\mathbf{call}, \mathbf{ret}, \mathbf{han}, \mathbf{exc}, \mathbf{stm}\}$: as the chosen identifiers suggest, **call** represents the fact that a procedure call occurs, **ret** represents the fact that a procedure terminates normally and returns to its caller, **exc** that an exception is raised, **han** that an exception handler is installed, and **stm** represents a statement that does not affect the stack, such as an assignment. We want to implement a policy such that an exception aborts all the pending calls up to the point where an appropriate handler is found in the stack, *if any*; after that, execution is resumed normally. Calls and returns, as well as possible pairing of handlers and exceptions are managed according to the usual LIFO policy. The alphabet symbols are written in boldface for reasons that will be explained later but are irrelevant for this example.

The above policy is implemented by the PRs described in Fig. 1 which displays the PRs through a square matrix, called *Operator Precedence Matrix (OPM)*, where the element of row i and column j is the PR between the symbol labeling row i and that of column j. We also add the special symbol # which is used as a string delimiter and state the convention that it yields precedence to all symbols in $\Sigma_{\mathbf{call}}$, and that all symbols in $\Sigma_{\mathbf{call}}$ take precedence over it.

Let us now see how the OPM of Fig. 1, named $M_{\mathbf{call}}$, drives the construction of a unique ST associated to a string on the alphabet $\Sigma_{\mathbf{call}}$ through a typical bottom-up parsing algorithm. We will see that the shape of the obtained ST depends only on the OPM and not on the particular grammar exhibiting the OPM. Consider the sample word w_{ex} = **call han call call call exc call ret call ret ret**. First, add the delimiter # at its boundaries and write all precedence relations between consecutive characters, according to $M_{\mathbf{call}}$. The result is row 0 of Fig. 2.

Then, select all innermost patterns of the form $a \lessdot c_1 \doteq \cdots \doteq c_\ell \gtrdot b$. In row 0 of Fig. 2 the only such pattern is the underscored **call** enclosed within the pair $(\lessdot, \gtrdot)$. This means that the ST we are going to build, if it exists, must contain an internal node with the terminal character **call** as its only child. We mark this fact by replacing

0 | # ⋖ **call** ⋖ **han** ⋖ **call** ⋖ **call** ⋖ **call** ⋗ **exc** ⋗ **call** ≐ **ret** ⋗ **call** ≐ **ret** ⋗ **ret** ⋗ #
1 | # ⋖ **call** ⋖ **han** ⋖ **call** ⋖ **call** N ⋗ **exc** ⋗ **call** ≐ **ret** ⋗ **call** ≐ **ret** ⋗ **ret** ⋗ #
2 | # ⋖ **call** ⋖ **han** ⋖ **call** N ⋗ **exc** ⋗ **call** ≐ **ret** ⋗ **call** ≐ **ret** ⋗ **ret** ⋗ #
3 | # ⋖ **call** ⋖ **han** ≐ N **exc** ⋗ **call** ≐ **ret** ⋗ **call** ≐ **ret** ⋗ **ret** ⋗ #
4 | # ⋖ **call** ⋖ N **call** ≐ **ret** ⋗ **call** ≐ **ret** ⋗ **ret** ⋗ #
5 | # ⋖ **call** ⋖ N **call** ≐ **ret** ⋗ **ret** ⋗ #
6 | # ⋖ **call** ≐ N **ret** ⋗ #
7 | # ≐ N#

Fig. 2. The sequence of bottom-up reductions during the parsing of w_{ex}.

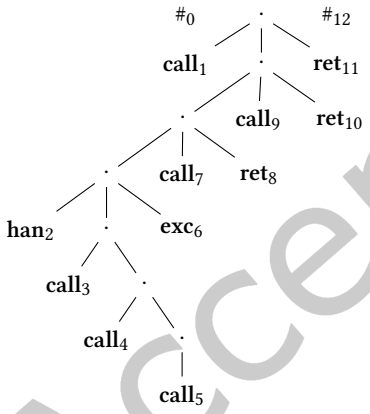


Fig. 3. The ST corresponding to word w_{ex}. Dots represent non-terminals.

the pattern ⋖**call**⋗ with a dummy non-terminal character, say N —i.e., we *reduce* **call** to N—. The result is row 1 of Fig. 2.

Next, we apply the same labeling to row 1 by simply ignoring the presence of the dummy symbol N and we find a new candidate for reduction, namely the pattern ⋖**call** N⋗. Notice that there is no doubt on building the candidate rhs as ⋖**call** N⋗: if we reduced just the **call** and replaced it by a new N, we would produce two adjacent internal nodes, which is impossible since the ST must be generated by a grammar in operator normal form.

By skipping the obvious reduction of row 2, we come to row 3. This time the terminal characters to be reduced, again, underscored, are two, with an ≐ and an N in between. This means that they embrace a subtree of the whole ST whose root is the node represented by the dummy symbol N. By executing the new reduction leading from row 3 to 4 we produce a new N immediately to the left of a **call** which is matched by an equal in precedence **ret**. Then, the procedure is repeated until the final row 7 is obtained, where, by convention we state the ≐ relation between the two delimiters.

Given that each reduction applied in Fig. 2 corresponds to a derivation step of a grammar and to the expansion of an internal node of the corresponding ST, it is immediate to realize that the ST of w_{ex} is the one depicted in Fig. 3, where the terminal symbols have been numbered according to their occurrence —including the conventional numbering of the delimiters— for future convenience, and labeling internal nodes has been omitted as useless.

The tree of Fig. 3 emphasizes the main difference between various types of parenthesis-like languages, such as VPLs, and OPLs: whereas in the former ones every open parenthesis is consumed by the only corresponding closed one[3], in our example a **call** can be matched by the appropriate **ret** but can also be "aborted" by an **exc** which in turn aborts all pending **call**s until its corresponding **han** —*if any*— is found.

Remark. The examples adopted in this paper are inspired by the important feature of exception-handling which is typical of most real-life programming languages but cannot be well-defined in terms of VPLs. Exception-handling, however, is not the only programming language feature that can be expressed in OPLs but not in less powerful formalisms [58]. Furthermore, although "hierarchies of exceptions" can be managed, e.g., in *colored VPLs* [8] and in *higher-order recursion schemes* [44], OPLs allow to express non-hierarchically typed exceptions *too*.

Thus, an OPM defines a *universe* of strings on the given alphabet that can be parsed according to it and assigns a unique ST —with unlabeled internal nodes— to each one of them. Such a universe is the whole Σ^* iff *the OPM is complete*, i.e. it has no empty cells, including those of the implicit row and column referring to the delimiters. In the early literature about OPLs, e.g., [30, 39] OPGs sharing a given OPM were used to define restricted languages w.r.t. the universe defined by the OPM and their algebraic properties have been investigated. Later on, the same operation has been defined by using different formalisms such as pushdown automata, monadic second order logic, and suitable extensions of regular expressions. In this paper we refer to the use automata and temporal logic, which are typical of model checking. As a side remark we mention that, in general, it may happen that in the same string there are several patterns ready to be reduced, without generating any ambiguity; this could enable the implementation of parallel parsing algorithms (see e.g., [16]) which however is not an issue of interest in this paper.

We now state the basics of OPLs needed for this paper in a formal way. Let Σ be a finite alphabet, and ε the empty string. We use the special symbol $\# \notin \Sigma$ to mark the beginning and the end of any string.

Definition 2.2. 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* $M(a, b)$ 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 (OP) alphabet*. By convention, the initial # yields precedence to other symbols, 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 $u, v \in (\Sigma \cup \{\#\})^+$ we write $u\ \pi\ v$ if $u = xa$ and $v = by$ with $a\ \pi\ b$.

The next concept of *chain* makes the connection between OP relations and ST structure explicit, through brackets.

Definition 2.3. A *simple chain* ${}^{c_0}[c_1 c_2 \dots c_\ell]^{c_{\ell+1}}$ is a string $c_0 c_1 c_2 \dots c_\ell c_{\ell+1}$, 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 $c_0 s_0 c_1 s_1 c_2 \dots c_\ell s_\ell c_{\ell+1}$, where ${}^{c_0}[c_1 c_2 \dots c_\ell]^{c_{\ell+1}}$ is a simple chain, and $s_i \in \Sigma^*$ is either 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 ${}^{c_0}[s_0 c_1 s_1 c_2 \dots c_\ell s_\ell]^{c_{\ell+1}}$. In a chain, simple or composed, c_0 is called its *left context* and $c_{\ell+1}$ its *right context*; the string of characters between them is called 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 $\#w\#$ which is a chain of the form ${}^{a}[y]^{b}$, $M(a, b)$ is defined. For a given OP alphabet (Σ, M) the set of all words compatible with M is called the *universe* of the OP alphabet.

The chain below is the chain defined by the OPM M_{call} of Fig. 1 for the word w_{ex}. It shows the natural isomorphism between STs with unlabeled internal nodes (see Fig. 3) and chains.

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

[3]To be precise, VPLs allow for unmatched closed parentheses but only at the beginning of a string and unmatched open ones at the end.

Note that, in composed chains, consecutive inner chains, if any, are always separated by at least one input symbol: this is due to the fact that OPL strings are generated by grammars in operator normal form.

Next, we introduce operator precedence automata as pushdown machines suitable to carve specific OPLs within the universe of an OP alphabet.

Definition 2.4. An *Operator Precedence Automaton (OPA)* is a tuple $\mathcal{A} = (\Sigma, M, Q, I, F, \delta)$ where: (Σ, M) is an OP 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, δ is a triple of transition relations $\delta_{shift} \subseteq Q \times \Sigma \times Q$, $\delta_{push} \subseteq Q \times \Sigma \times Q$, and $\delta_{pop} \subseteq Q \times Q \times Q$.

An OPA is deterministic iff I is a singleton, and all three components of δ are —possibly partial— functions.

To define the semantics of OPAs, 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 Γ be $\Sigma \times Q$ and $\Gamma' = \Gamma \cup \{\bot\}$ be the *stack alphabet*; we denote symbols in Γ as $[a,\ q]$. 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 $n \geq 1$, and $smb(\gamma) = \#$ 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 and the state in the removed stack symbol. They do not consume the input symbol, which is used only as a look-ahead to establish the $\gtrdot$ relation. The OPA accepts the language $L(\mathcal{A}) = \{x \in \Sigma^* \mid \langle x\#,\ q_I,\ \bot \rangle \vdash^* \langle \#,\ q_F,\ \bot \rangle, q_I \in I, q_F \in F\}$.

Definition 2.5. Let $\mathcal{A}$ be an OPA. We call a *support* for the simple chain ${}^{c_0}[c_1 c_2 \ldots c_\ell]^{c_{\ell+1}}$ any path in $\mathcal{A}$ of the form $q_0 \xrightarrow{c_1} q_1 \dashrightarrow \ldots \dashrightarrow q_{\ell-1} \overset{c_\ell}{\dashrightarrow} q_\ell \overset{q_0}{\Longrightarrow} q_{\ell+1}$. 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* ${}^{c_0}[s_0 c_1 s_1 c_2 \ldots c_\ell s_\ell]^{c_{\ell+1}}$ any path in $\mathcal{A}$ of the form

$$q_0 \overset{s_0}{\rightsquigarrow} q_0' \xrightarrow{c_1} q_1 \overset{s_1}{\rightsquigarrow} q_1' \overset{c_2}{\dashrightarrow} \ldots \overset{c_\ell}{\dashrightarrow} q_\ell \overset{s_\ell}{\rightsquigarrow} q_\ell' \overset{q_0'}{\Longrightarrow} q_{\ell+1}$$

where, for every $i = 0, 1, \ldots, \ell$: if $s_i \neq \varepsilon$, then $q_i \overset{s_i}{\rightsquigarrow} q_i'$ is a support for the chain ${}^{c_i}[s_i]^{c_{i+1}}$, else $q_i' = q_i$.

Consider the OPA $\mathcal{A}(\Sigma, M) = (\Sigma, M, \{q\}, \{q\}, \{q\}, \delta^{max})$ where $\delta^{max}_{push}(q, c) = \delta^{max}_{shift}(q, c) = q$, $\forall c \in \Sigma$, and $\delta^{max}_{pop}(q, q) = q$. We call it the *OP Max-Automaton* over (Σ, M). For a max-automaton, each chain has a support; thus, a max-automaton accepts exactly the universe of the OP alphabet. If ${}^a[s]^b$ is a chain over (Σ, M), $\mathcal{A}(\Sigma, M)$ performs the computation $\langle sb, q, [a, q]\gamma \rangle \vdash^* \langle b, q, \gamma \rangle$, and there exists a support like the one above with $s = s_0 c_1 \ldots c_\ell s_\ell$. This corresponds to the parsing of the string $s_0 c_1 \ldots c_\ell s_\ell$ within the context a, b, which contains all information needed to build the subtree whose frontier is that string. If M is complete, the language accepted by $\mathcal{A}(\Sigma, M)$ is Σ^*. With reference to the OPM M_{call} of Fig. 1, the string **ret call han** is accepted by the max-automaton with structure defined by the chain #[[**ret**]**call**[**han**]]#. This string cannot be interpreted as a normal program trace: later, we will show how we can build OPAs that only accept strings that make sense as program traces.

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 the OP alphabet. OPAs form a Boolean algebra whose universal element is the max-automaton. The language classes recognized by deterministic and non-deterministic OPAs coincide. For a more complete description of the OPL family and of its relations with other CFLs we refer the reader to [59].

2.1 Operator Precedence ω-Languages

All definitions regarding OPLs are extended to infinite words in the usual way, but with a few distinctions [58].

Given a set of characters Δ, by Δ^ω we mean the set of all infinite words made of characters in Δ.

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 Operator Precedence ω-Languages (ωOPLs) can be defined by augmenting Definition 2.4 with Büchi or Muller acceptance conditions. In this paper, we only consider the former one.

Definition 2.6 (Operator Precedence Büchi Automaton (ωOPBA)). Let an ωOPBA $\mathcal{A}$, its configurations and moves be defined as for OPAs accepting finite strings.

An infinite run on an ω-word w is an infinite sequence $\rho = \langle x_0 = w, q_0, \gamma_0\rangle \vdash \langle x_1, q_1, \gamma_1\rangle \vdash \ldots$. Define the set of states that occur infinitely often in ρ as

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

A run ρ is successful iff there exists a state $q_f \in F$ such that $q_f \in \mathrm{Inf}(\rho)$. $\mathcal{A}$ 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 OPAs, ωOPBAs 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, which form a Boolean algebra and are closed under concatenation of an OPL with an ωOPL [58]. 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 ωVPLs.

In our model-checking procedures, we will need a slight variation on ωOPBAs:

Definition 2.7 (Generalized ωOPBA). A generalized ωOPBA is a tuple $\mathcal{A} = (\Sigma, M, Q, I, \mathbf{F}, \delta)$, where Σ, M, Q, I, δ are the same as in Definition 2.4, and $\mathbf{F} \subseteq \mathcal{P}(Q)$ is the set of sets of Büchi-final states.

The semantics of configurations, moves and runs are defined as for ωOPBAs. The acceptance condition is, again, different: a run ρ on an ω-word is successful iff for all $F_i \in \mathbf{F}$ there exists a state $q_i \in F_i$ such that $q_i \in \mathrm{Inf}(\rho)$.

Generalized ωOPBA can be translated to normal ωOPBA polynomially:

Theorem 2.8. *Let $\mathcal{A} = (\Sigma, M, Q, I, \mathbf{F}, \delta)$ be a generalized ωOPBA. It is possible to build an ωOPBA $\mathcal{A}'$ with $|Q| \cdot |\mathbf{F}|$ states such that $L(\mathcal{A}') = L(\mathcal{A})$.*

The proof uses a classic construction based on counters [54], and is thus omitted. Since the translation from simple to generalized ωOPBAs is trivial, the two classes are equivalent and enjoy the same closure properties.

A more complete treatment of OPLs' properties and parsing algorithms can be found in [43, 59]; ωOPLs are treated in-depth in [58].

```
PROGRAM := [DECL; ...] FUNCTION [FUNCTION ...]
DECL := TYPE IDENTIFIER [, IDENTIFIER ...]
TYPE := bool | uINT | sINT | uINT[INT] | sINT[INT]
FUNCTION := IDENTIFIER ([FARG, ...]) { [DECL; ...] STMT; [STMT; ...] }
FARG := TYPE IDENTIFIER | TYPE & IDENTIFIER
STMT := LVALUE = BEXPR
      | LVALUE = *
      | while (GUARD) { [STMT; ...] }
      | if (GUARD) { [STMT; ...] } else { [STMT; ...] }
      | try { [STMT; ...] } catch { [STMT; ...] }
      | IDENTIFIER([EXPR, ...])
      | throw
GUARD := * | EXPR
LVALUE := IDENTIFIER | IDENTIFIER[EXPR]
EXPR := EXPR || CONJ | CONJ
CONJ := CONJ && BTERM | BTERM
BTERM := IEXPR COMP IEXPR | IEXPR
COMP := == | != | < | <= | > | >=
IEXPR := IEXPR + PEXPR | IEXPR - PEXPR | PEXPR
PEXPR := PEXPR * ITERM | PEXPR / ITERM | ITERM
ITERM := !ITERM | (EXPR) | IDENTIFIER | IDENTIFIER[EXPR] | LITERAL
LITERAL := [+|-] INT[u|s]INT | true | false
```

(a) MiniProc syntax.

```
pA() {
  bool foo;
  foo = true;
  try { pB(foo); }
  catch { pErr(); pErr(); }
}
pB(bool bar) {
  if (bar) { pC(); }
  else {}
}
pC() {
  if (*) { throw; }
  else { pC(); }
}
pErr() {}
```

(b) A MiniProc program.

Fig. 4. MiniProc syntax and a MiniProc program. In the syntax, 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. An INT is an unsigned integer literal.

2.2 Modeling Procedural Programs by means of OPA

We now introduce the MiniProc language, which allows the programmer to express algorithms in a more customary style than the automata-theoretic OPA. MiniProc retains the most important features of traditional C-like programming languages; special attention is devoted to the exception handling mechanism and its consequences in the managing of the stack, which is a distinguishing feature of OPLs. With our tool POMC, the user has the choice of expressing their algorithms as OPAs or as MiniProc programs; in the latter case our tool automatically "compiles" MiniProc into the OPA formalism to be checked against the logic specification language POTL.

Fig. 4a shows the MiniProc syntax while Fig. 4b presents a first example of MiniProc program that will be exploited in Example 2.9.

A program starts with global variable declarations. MiniProc supports finite-width integer variables, both signed and unsigned, and arrays. Then, a sequence of functions is defined, the first one being the entry-point to the program. Functions may have positional parameters, passed by value (default) or by value-result (by adding &). Function bodies consist of semicolon-separated local variable declarations and statements. Assignments, `while` loops and `if`s have the usual semantics. The token `*` means nondeterminism: when used in a guard, it means that both branches can be taken nondeterministically; when used in an assignment, it means the assigned variable may take any value allowed by its type. 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 raised by the `throw` statement,

and they are not typed (i.e., there is no way to distinguish different kinds of exceptions). Functions can be called by prepending their name to actual parameters surrounded by parentheses. Integer expressions can be composed with the usual arithmetic operators, and Boolean operators, which automatically convert integers to Booleans (0 means false, $\neq 0$ true). All integer literals must be prepended to their type (e.g., `42u8` is the value 42 represented as an 8-bit unsigned integer).

OPAs —or ωOPBAs— semantically equivalent to MiniProc programs are automatically generated by POMC by following a path inspired by previous similar translations of the literature, e.g., [4].

The automaton's alphabet is that of OPM $M_{\mathbf{call}}$, whose symbols are paired with identifiers of the MiniProc language in such a way that there is a one-to-one correspondence between the automaton's alphabet and the statements of the program. Precisely, the assignment to a variable corresponds to symbol **stm** possibly paired with the identifier of the assigned variable, the call of a procedure corresponds to symbol **call** paired with the procedure identifier (notice that a procedure call may occur also as part of a `catch` block); the end of any procedure (syntactically its final }) corresponds to **ret** paired with the procedure identifier; the installation of a handler —the `try` part of `try-catch` block— corresponds to the **han** symbol paired with the identifier of the procedure in the scope of which the handler is declared.

Besides symbols in $\Sigma_{\mathbf{call}}$, transitions are also labeled with all variable identifiers that are in the scope of the statement they represent and that are true in that moment. For integer variables, we use the common convention that considers them "true" when they are non-zero. Note that, if a variable can take multiple values (e.g., because of a '`*`' assignment), a different execution path for each one of the possible values is created.

Finally, the `throw` statement corresponds to the **exc** symbol with no further labels since exceptions are not typed in MiniProc. The PRs of the new alphabet are the same as $M_{\mathbf{call}}$, simply "forgetting" the additional identifiers[4]. The construction of the OPAs is performed in two phases as in the similar cases of the literature.

First, an *extended* OPA is generated, in which every state corresponds to a position of the MiniProc program reached during its execution, and transitions are labeled by the statement whose execution they represent and, possibly, by Boolean expression guards that must be true for them to be performed. A shift transition labeled "dummy **exc**", with appropriate target state, is also added to represent the exit from a `try-catch` block in a symmetric way as the return from a procedure call, i.e., when the scope of the handler is closed without generating a corresponding exception. Pop transitions represent the "completion" of a statement but do not necessarily mean that the MiniProc interpreter deallocates anything from its stack: e.g., the pop transition that follows the reading of a **ret** does mean that an activation record is popped out from the MiniProc stack, but even an assignment statement is represented by a push transition immediately followed by a pop one in the OPA.

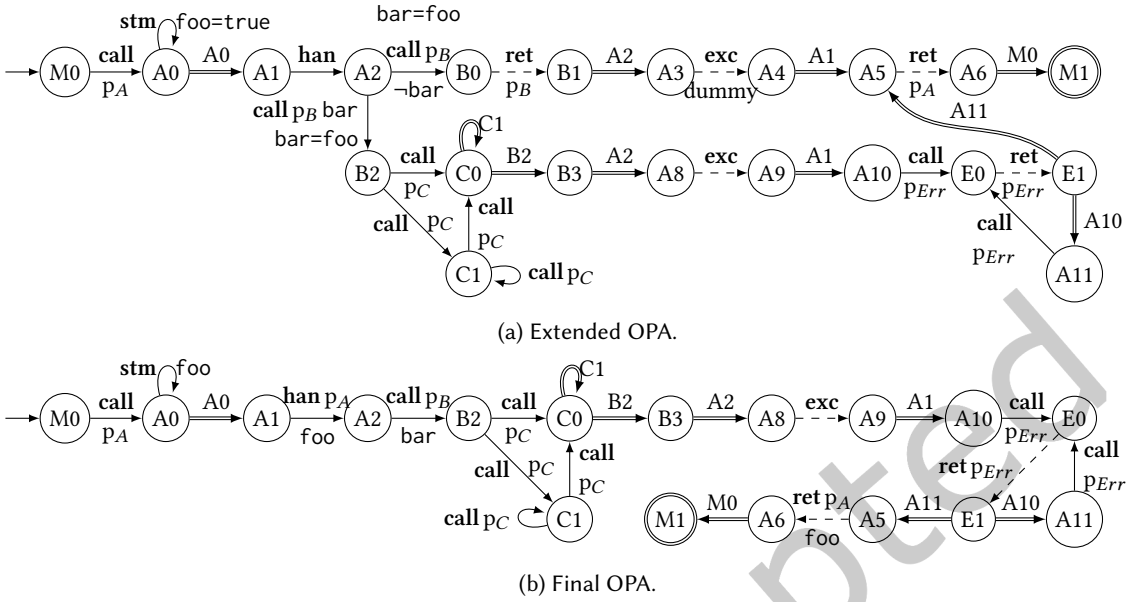
Then, the extended OPA is translated into a normal one. The key point here consists in enumerating all feasible variable assignments for each state, and by labeling transitions with Boolean variables that hold when they are triggered. This clearly exposes to the risk of a typical state-space explosion. On the other hand, however, some clean-up is performed, e.g., by eliminating (parts corresponding to) unfeasible branches of the code.

More details of the above construction will be shown in the following Example 2.9.

It should be now clear that the obtained OPA is equivalent to the original MiniProc program in the sense that the language it accepts is isomorphic to the successful runs of the MiniProc interpreter, and the sentences it rejects are isomorphic to the runs resulting into some error, *whenever its execution terminates*.

A problem arises, however, when MiniProc runs do not terminate. OPAs in fact, always terminate *by definition* since they need input explicitly terminated by a #, and the OPM is such that either they halt because some PR is not defined or they always reach the final #. Thus, when the MiniProc program does not terminate, the corresponding input string for the OPA becomes an ω-string. The obvious consequence is that OPAs can be used

[4]As already anticipated in Example 2.1, the purpose of using the boldface character for some labels will be fully clarified in Section 3.



(a) Extended OPA.

(b) Final OPA.

Fig. 5. The two steps of OPA generation from the code of Fig. 4b.

only to check properties of programs that we know a priori to terminate. In the opposite case we must resort to ωOPBAs, which, however, work only on ω-languages whose sentences are *all* infinite.

Many properties, therefore, typically just termination itself, could not be checked neither using OPA nor using ωOPBAs. In such cases we resort to a fairly typical solution: we transform a MiniProc program that exhibits both finite and infinite behaviors into an automaton that only accepts infinite traces, by creating an ωOPBA such that traces produced by a terminating MiniProc program are transformed into non-terminating ones by adding *stuttering states* after the return statement of the main procedure. These stuttering states are linked by transitions that actually read dummy input symbols: we chose to add an infinite sequence of calls and returns of a dummy function, but other choices are possible (e.g., a dummy **stm**, or a new symbol). In this way, to check termination, we can check reachability of the **ret** statement of the main function. With this construction, all properties checked on such an ωOPBA are checked on both finite and infinite traces of the original MiniProc program, and we can use appropriate formulas to restrict them to only finite or infinite ones.

Next, we give two examples with complementary purposes: Example 2.9 aims at illustrating the main features of extended OPAs, their translation into normal OPAs, and their typical managing of the stack that allows for popping several items without reading any input character —the non-real-time behavior that increases OPLs' expressive power w.r.t. other pushdown automata for structured languages.

Example 2.10 exploits the well-known algorithm Quicksort to argue that the OPL formalism is general enough to express even sophisticated algorithms in a natural way and to hint that the MiniProc language can easily evolve into a complete programming language. More arguments to support such a claim can be found in previous literature, e.g., [59]. Both examples are exploited in Section 7 as the core of the benchmark we adopted to evaluate the performances of our model-checking tool POMC.

Example 2.9. Fig. 5a shows the extended OPA derived from the code in Fig. 4b. The stack semantics of the two models coincide: a symbol is pushed for every function call (**call**), and popped after the corresponding return

```
program:
bool sorted;
u3[4] a;

main() {
    sorted = false;
    a[0s4] = *;
    a[1s4] = *;
    a[2s4] = *;
    a[3s4] = *;
    qs(0s4, 3s4);
}

swapElements(s4 swapLeft, s4 swapRight) {
    u3 tmp;

    tmp = a[swapLeft];
    a[swapLeft] = a[swapRight];
    a[swapRight] = tmp;
    sorted = a[0s4] <= a[1s4]
             && a[1s4] <= a[2s4]
             && a[2s4] <= a[3s4];
}
```

```
qs(s4 left, s4 right) {
    s4 lo, hi;
    u3 piv;

    if (left < right) {
        piv = a[right];
        lo = left;
        hi = right;
        while (lo <= hi) {
            if (a[hi] > piv){
                hi = hi - 1s4;
            } else {
                swapElements(lo, hi);
                lo = lo + 1s4;
           }
        }
        qs(left, hi);
        qs(lo, right);
    } else {}
}
```

Fig. 6. The "Buggy" Quicksort algorithm in MiniProc.

(**ret**) or exception (**exc**). Handlers (**han**) are paired with 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, whereas a simple **exc** is generated in correspondence of an explicit throw statement. Assignments are denoted by a **stm**, which causes a push immediately followed by a pop. M_{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.

The resulting OPA is in Fig. 5b. The assignment of `true` to `foo` is propagated forward from state A0, and the branch from A2 to B0 and B2 is removed, because B0 is unreachable. The last part of the OPA generation leads to a worst-case model size exponential in the number of non-deterministic assignments (not shown in the example for brevity). However, as we shall see in Section 7, it performs well in many practical cases, because only feasible states are generated.

When an OPA is generated, the set of final states only contains the last state of the "main" module (M1 in the example). When an ωOPBA is generated, all states are marked as final. If the MiniProc program contains an actual infinite loop, this will result in an accepting loop in the ωOPBA. An accepting stuttering state is also added at the end of the ωOPBA, so that finite behaviors can be modeled too. Thus, the ωOPBA accepts all possible traces of the program; the desirable ones will be discriminated by the requirement to be checked.

Example 2.10. Fig. 6 shows a recursive MiniProc implementation of the QuickSort algorithm. We show it to demonstrate the syntax of MiniProc through a classic example, but we do not report the resulting ωOPBA due to its size. In Section 7, we will use this and other programs as benchmarks for our model checking tool.

The goal of Quicksort is to sort in-place an input array in ascending order with a divide-and-conquer strategy: at every iteration, an element is chosen as the "pivot", and the array is split in two subarrays which contain, respectively, all the elements smaller than the pivot, and all the elements greater than the pivot. The two subarrays are recursively sorted with the same strategy. The program employs two global variables: the array '`a`' to be sorted, and Boolean variable '`sorted`', which indicates whether the array is sorted. The latter one is set and updated every time a swap of cell values is performed. In the figure, the array is composed of 4 elements, but in Section 7 we study it on larger and smaller arrays, too. '`u3`' indicates that array elements are 3-bit unsigned integers, hence their domain is $[0, 7]$. The `main()` procedure first assigns a nondeterministic value for each array cell, and then calls the QuickSort procedure `qs()` on the array. `qs()` uses 5 local variables: `left`, `right`, `lo`, `hi`, `piv`. The first four contain array indices: they are signed integers of 4 bits having domain $[-7, 7]$ (their type name is '`s4`'). `piv` contains an array value, so it has the same type as array cells. This procedure is taken from [37], where an equivalent C program is given. It is called "Buggy Quicksort" because it enters an infinite loop when the array contains two cells with the same value, thus termination is not guaranteed. In such a case, `qs()` will continue swapping and calling itself recursively on the same pair of cells. In Section 7.2.2, we prove that the generated ωOPBA has an accepting loop that pushes the same stack symbol at every recursive call, growing the stack indefinitely.

3 PRECEDENCE ORIENTED TEMPORAL LOGIC

POTL is a linear-time temporal logic, which extends the classical LTL. We recall that the semantics of LTL [71] is defined on a Dedekind-complete set of word positions U equipped with a total ordering, and monadic relations, called *Atomic Propositions (APs)*. In this paper, we consider a discrete timeline, hence $U = \{0, 1, \ldots, n\}$, with $n \in \mathbb{N}$, or $U = \mathbb{N}$. Each LTL formula φ is evaluated in a word position: we write $(w, i) \models \varphi$ to state that φ holds in position i of word w.

Besides operators from propositional logic, LTL features modalities that define relations between positions; e.g., the *Next* modality states that a formula holds in the subsequent position of the current one: $(w, i) \models \bigcirc \varphi$ iff $(w, i + 1) \models \varphi$; the *Until* modality states that there exists a *linear path*, made of consecutive positions and starting from the current one, such that a formula ψ holds in the last position of such path, and another formula φ holds in all previous positions. Formally, $(w, i) \models \varphi \,\mathcal{U}\, \psi$ iff there exists $j \geq i$ s.t. $(w, j) \models \psi$, and for all j', with $i \leq j' < j$, we have $(w, j') \models \varphi$.

The linear order, however, is not sufficient to express properties of more complex structures than the linear ones, typically the tree-shaped ones, which are the natural domain of context-free languages. The history of logic formalisms suitable to deal with CFLs somewhat parallels the path that lead from regular languages to tree-languages [77] or their equivalent counterpart in terms of strings, i.e. parenthesis languages [62].

A first logic mechanism aimed at "walking through the structure of a context-free sentence" was proposed in [57] and consists in a *matching condition* that relates the two extreme terminals of the rhs of a context-free grammar in so-called *double Greibach normal form*, i.e. a grammar whose production rhs exhibit a terminal character at both ends: in a sense such terminal characters play the role of explicit parentheses. [57] provides a logic language for general CFLs based on such a relation which however fails to extend the decidability properties of logics for regular languages due to lack of closure properties of CFLs. The matching condition was then resumed in [10] to define its MSO logic for VPLs and the temporal logics CaRet [7] and NWTL [2].

OPLs are structured but not "visibly structured" as they lack explicit parentheses (see Section 2). Nevertheless, a more sophisticated notion of matching relation has been introduced in [58] for OPLs by exploiting the fact that OPLs remain input-driven thanks to the OPM. We name the new matching condition *chain relation* and define it here below. We fix a finite set of atomic propositions AP, and an OPM M_{AP} on $\mathcal{P}(AP)$.

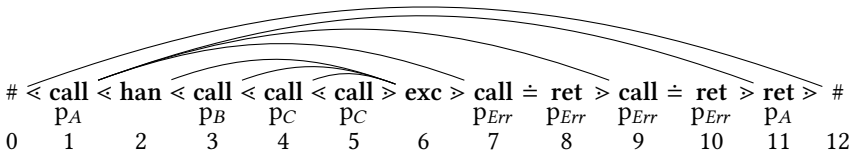


Fig. 7. The OP-word of which the string w_{ex} of Example 2.1 is the homomorphic image. Chains are highlighted by arrows joining their contexts; structural labels are in bold, and other atomic propositions are shown below them. p_l means that a **call** or a **ret** is related to procedure p_l. First, procedure p_A is called (pos. 1), and it installs an exception handler in pos. 2. Then, three nested procedures are called, and the innermost one (p_C) throws an exception, which is caught by the handler. Two more functions are called and, finally, p_A returns.

A *word structure* —also called *OP word* for short— is the tuple $(U, <, M_{AP}, P)$, where U, $<$, and M_{AP} are as above, and $P\colon AP \to \mathcal{P}(U)$ is a function associating each atomic proposition with the set of positions where it holds, with $0, (n+1) \in P(\#)$. For the time being, we consider just finite string languages; the necessary extensions needed to deal with ω-languages will be introduced in Section 3.2.

Definition 3.1 (Chain relation). The *chain relation* $\chi(i, j)$ holds between two positions $i, j \in U$ iff $i < j - 1$, and i and j are resp. the positions of the left and right contexts of the same chain (cf. Definition 2.3), according to M_{AP} and the labeling induced by P.

In the following, given two positions i, j and a PR π, we write $i \;\pi\; j$ to say $a \;\pi\; b$, where $a = \{\mathrm{p} \mid i \in P(\mathrm{p})\}$, and $b = \{\mathrm{p} \mid j \in P(\mathrm{p})\}$. For notational convenience, we partition AP into *structural labels*, written in bold face, which define a word's structure, and normal labels, in round face, defining predicates holding in a position. Thus, an OPM M can be defined on structural labels only, and M_{AP} is obtained by inverse homomorphism of M on subsets of AP containing exactly one of them.

To obtain an intuitive idea of how the chain relation augments the linear structure of a word with the tree-like structure of OPLs consider Fig. 7: it displays an OP-word on the alphabet of the OPA of Fig. 5b whose image under the homomorphism projecting the OPA's alphabet onto the boldface components is the word w_{ex} from Example 2.1. Since the OPM of the OPA is isomorphic to that of Fig. 1, the ST of the OP-word of Fig. 7 is isomorphic to that of Fig. 3. Thus, for simplicity, in the following we will refer to the ST of Fig. 3 as if it was the ST of the OP-word of Fig. 7. Notice also that, rigorously speaking, the OP-word of the figure is not accepted by the OPA although it is compatible with its OPM, since the OPA —and the MiniProc program from which it is derived— necessarily perform the assignment statement `foo = true` at the beginning of procedure p_A. We omitted that statement in the OP-word to help focusing on the stack management policy and how it is reflected in the χ relation.

Fig. 7 emphasizes the distinguishing feature of the relation, i.e. that, for composed chains, it may not be one-to-one, but also one-to-many or many-to-one. Notice also the correspondence between internal nodes in the ST of Fig. 3 and pairs of positions in the χ relation.

In a ST, we say that 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. 3, pos. 1 with 11 and 2 with 6), 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).

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) Consider all positions (if any) $i_1 < i_2 < \cdots < i_n$ s.t. $\chi(i_p, j)$ for all $1 \leq p \leq n$. We have $i_1 \lessdot j$ or $i_1 \doteq j$ and, if $n > 1$, $i_q \gtrdot j$ for all $2 \leq q \leq n$.
(4) Consider all positions (if any) $j_1 < j_2 < \cdots < j_n$ s.t. $\chi(i, j_p)$ for all $1 \leq p \leq n$. We have $i \gtrdot j_n$ or $i \doteq j_n$ and, if $n > 1$, $i \lessdot j_q$ for all $1 \leq q \leq n-1$.

Property 4 says that when the chain relation is one-to-many, the contexts of the outermost chain (i_1 and j) are in the $\doteq$ or $\gtrdot$ relation, while the inner ones are in the $\lessdot$ relation. We call i_1 the *leftmost context* of j. Property 3 says that contexts of outermost many-to-one chains (i and j_n) are in the $\doteq$ or $\lessdot$ relation, and the inner ones are in the $\gtrdot$ relation. We call j_n the *rightmost context* of i. Such properties are proved in [27] for readers unfamiliar with OPLs.

The χ relation is the core of the MSO logic characterization for OPLs given in [58]; as a natural consequence of the greater generality of OPLs over VPLs, the MSO logic for the former family has a greater expressive power than the one for the latter family. Indeed, such a greater power requires more technical analysis which, however, allows to prove the same important results in terms of closure properties, decidability and complexity of the constructions, as those holding for VPLs and the corresponding MSO logic.

Similarly, in [27] we show that the temporal logic POTL is FO-complete as well as NWTL, despite the greater complexity of the χ relation. To complete the path, in this paper we produce model checking algorithms for POTL and OPLs with the same order of complexity as those for NWTL and VPLs.

While LTL's linear paths only follow the ordering relation $<$, paths in POTL may follow the χ relation too. As a result, a POTL path through a string can simulate paths through the corresponding ST.

We envisage two basic types of path. The first one is that of *summary paths*. By following the chain relation, summary paths may skip chain bodies, which correspond to the fringe of a subtree in the ST. We distinguish between *downward* and *upward* summary paths (resp. DSP and USP). Both kinds can follow both the $<$ and the χ relations; DSPs can enter a chain body but cannot exit it so that they can move only downward in a ST or remain at the same level; conversely, USPs cannot enter one but can move upward by exiting the current one. In other words, if a position k is part of a DSP, and there are two positions i and j, with $i < k < j$ and $\chi(i, j)$ holds, the next position in the DSP cannot be $\geq j$. E.g., two of the DSPs starting from pos. 1 in Fig. 7 are 1-2-3, which enters chain $\chi(2, 6)$, and 1-2-6, which skips its body. USPs are symmetric, and some examples thereof are paths 3-6-7 and 4-6-7.

Since the χ relation can be many-to-one or one-to-many, it makes sense to write formulas that consider only left contexts of chains that share their right context, or vice versa. Thus, the paths of our second type, named *hierarchical paths*, are made of such positions, but excluding outermost chains. E.g., in Fig. 7, positions 2, 3 and 4 are all in the χ relation with 6, so 3-4 is a hierarchical path ($\chi(2, 6)$ is the outermost chain). Symmetrically, 7-9 is another hierarchical path. The reason for excluding the outermost chain is that, with most OPMs, such positions have a different semantic role than internal ones. E.g., positions 3 and 4 are both calls terminated by the same exception, while 2 is the handler. Positions 7 and 9 are both calls issued by the same function (the one called in position 1), while 11 is its return. This is a consequence of properties 3 and 4 above.

In the next subsection, we describe in a complete and formal way POTL for finite string OPLs while in the subsequent subsection we briefly describe the necessary changes to deal with ω-languages.

3.1 POTL Syntax and Semantics

Given a finite set of atomic propositions AP, let $\mathrm{a} \in AP$, and $t \in \{d, u\}$. The syntax of POTL is the following:

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

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 $(w, i) \models \mathrm{a}$ iff $i \in P(\mathrm{a})$. Propositional operators such as $\wedge$, $\vee$ and $\neg$ have their

usual semantics. Next, while giving the formal semantics of other 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. 7.

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 $\gtrdot$ for $\lessdot$ 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. 7), $\ominus^d$ **call** to say that the previous position is a **call**, and the current one 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.

Chain Next/Back. The *chain* next and back operators χ_F^t and χ_P^t 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) or remains at the same level in the ST. 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.

E.g., in pos. 1 of Fig. 7, $\chi_F^d \mathrm{p}_{Err}$ holds because $\chi(1, 7)$ and $\chi(1, 9)$, meaning that p_A calls p_{Err} at least once. Also, 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. Notice that, although the upper label of the χ_P^u operator is a u, the **call**s in pos. 3 and 4 are below the **exc** in pos. 6 in the ST: this is due to the fact that the u label refers to the left-to-right direction of the involved chain. 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).

(Summary) 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 $\psi \, \mathcal{U}_\chi^t \, \theta$ (resp. since $\psi \, \mathcal{S}_\chi^t \, \theta$) operator is obtained by inductively applying the $\bigcirc^t$ and χ_F^t (resp. $\ominus^t$ 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. $\ominus^t(\psi \, \mathcal{S}_\chi^t \, \theta)$) 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 reference to a MiniProc program modeled as an OPA, 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 $\top \, \mathcal{U}_\chi^u$ **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. 7 because of path 3-6, and false in pos. 1, because no upward path can enter the chain whose contexts are pos. 1 and 11. Formula $\top \, \mathcal{U}_\chi^d$ **exc** is true in call positions whose function frame contains **exc**s, but that are not directly terminated by one of them, such as the one in pos. 1 (with path 1-2-6).

We formally define *Downward Summary Paths (DSPs)* 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

$1 \le p < n$,

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

The Downward Summary (DS) until and since operators $\mathcal{U}^d_\chi$ and $\mathcal{S}^d_\chi$ use as Γ the set of DSPs starting in the position in which they are evaluated. The definition for the *Upward Summary Paths (USPs)*, on which Upward Summary (US) until and since are based, is obtained by substituting $\gtrdot$ for $\lessdot$. For instance, in Fig. 7, **call** $\mathcal{U}^d_\chi$ (**ret** $\wedge$ p_{Err}) holds in pos. 1 because of path 1-7-8 and 1-9-10, (**call** $\vee$ **exc**) $\mathcal{S}^u_\chi$ p_B in pos. 7 because of path 3-6-7, and (**call** $\vee$ **exc**) $\mathcal{U}^u_\chi$ **ret** in 3 because of path 3-6-7-8.

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 a right context of i, with $i \lessdot j, j'$. So, $\bigcirc^u_H \mathrm{p}_{Err}$ holds in pos. 7 of Fig. 7 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 $(w, j) \models \varphi$;
- $(w, i) \models \ominus^d_H \varphi$ iff there exist a position $h > i$ s.t. $\chi(i, h)$ and $i \gtrdot h$ and a position $j = \max\{k \mid k < i \wedge \chi(k, h) \wedge k \gtrdot h\}$ and $(w, j) \models \varphi$.

In the ST of Fig. 3, $\bigcirc^d_H$ and $\ominus^d_H$ go *down* and up among **call**s terminated by the same **exc**. For example, in pos. 3 $\bigcirc^d_H \mathrm{p}_C$ holds, because both pos. 3 and 4 are in the chain relation with 6. Similarly, in pos. 4 $\ominus^d_H \mathrm{p}_B$ holds. Note that these operators do not consider leftmost/rightmost contexts, so $\bigcirc^u_H$ **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 $1 \le p \le n$ we have $\chi(h, i_p)$ and $h \lessdot i_p$, and for each $1 \le q < n$ 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 UHPs starting in the position in which they are evaluated are denoted as $\mathcal{U}^u_H$ and $\mathcal{S}^u_H$. E.g., **call** $\mathcal{U}^u_H$ p_{Err} holds in pos. 7 because of the singleton path 7 and path 7-9, and **call** $\mathcal{S}^u_H$ 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 $1 \le p \le n$ we have $\chi(i_p, h)$ and $i_p \gtrdot h$, and for each $1 \le q < n$ 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 DHPs starting in the position in which they are evaluated are denoted as $\mathcal{U}^d_H$ and $\mathcal{S}^d_H$. In Fig. 7, **call** $\mathcal{U}^d_H$ p_C holds in pos. 3, and **call** $\mathcal{S}^d_H$ p_B in pos. 4, both because of path 3-4.

Equivalences. POTL until and since operators enjoy expansion laws similar to those of LTL. Here we give laws for until operators, those for their since counterparts being symmetric. They were originally formulated in [27]; those referring to hierarchical operators, however, suffered from a minor inaccuracy, which is fixed in the present

version.

$$\varphi\, \mathcal{U}^t_\chi\, \psi \equiv \psi \vee \Big(\varphi \wedge \big(\bigcirc^t(\varphi\, \mathcal{U}^t_\chi\, \psi) \vee \chi^t_F(\varphi\, \mathcal{U}^t_\chi\, \psi)\big)\Big) \quad \text{for } t \in \{d, u\}$$
$$\varphi\, \mathcal{U}^u_H\, \psi \equiv (\psi \wedge \chi^{\lessdot}_P \top) \vee \big(\varphi \wedge \bigcirc^u_H(\varphi\, \mathcal{U}^u_H\, \psi)\big)$$
$$\varphi\, \mathcal{U}^d_H\, \psi \equiv (\psi \wedge \chi^{\gtrdot}_F \top) \vee \big(\varphi \wedge \bigcirc^d_H(\varphi\, \mathcal{U}^d_H\, \psi)\big)$$

where $\chi^{\lessdot}_P \top$ is the restriction of $\chi^d_P \top$ to chains having their left context in the $\lessdot$ PR with the right one. Formally, for any POTL formula γ we define $\chi^{\lessdot}_P \gamma := \bigvee_{a,b \subseteq AP,\, a \lessdot b} (\sigma_a \wedge \chi^d_P(\sigma_b \wedge \gamma))$, where for any $c \subseteq AP$, $\sigma_c := \bigwedge_{\mathrm{p} \in c} \mathrm{p} \wedge \bigwedge_{\mathrm{q} \notin c} \neg \mathrm{q}$ holds in a position i iff c is the set of atomic propositions holding in i. $\chi^{\gtrdot}_F \top$ is defined symmetrically. We will make a more systematic use of these specialized chain operators in Section 4.

As in LTL, it is worth defining some useful derived operators. For $t \in \{d, u\}$, we define the downward/upward summary *eventually* as $\Diamond^t \varphi := \top\, \mathcal{U}^t_\chi\, \varphi$, and the downward/upward summary *globally* as $\Box^t \varphi := \neg \Diamond^t (\neg \varphi)$. $\Diamond^u \varphi$ and $\Box^u \varphi$ respectively say that φ holds in one or all positions in the path from the current position to the root of the ST. Their downward counterparts consider all positions in the current rhs and its subtrees, starting from the current position. $\Diamond^d \varphi$ says that φ holds in at least one of such positions, 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**. This way, the LTL *globally* operator $\Box \psi$ can be expressed in POTL as $\Box \psi := \neg \Diamond^u (\Diamond^d \neg \psi)$ [27].

3.2 POTL on ω-Words

Since applications in model checking usually require temporal logics on infinite words, we extend POTL to ω-words.

To define OP ω-words, it suffices to replace the finite set of positions U with the set of natural numbers $\mathbb{N}$ in the definition of OP words. OP ω-words contain open chains, and property 4 of the χ relation does not hold if a position i is the left context of an open chain. In fact, there may be positions $j_1 < j_2 < \cdots < j_n$ such that $\chi(i, j_p)$ and $i \lessdot j_p$ for all $1 \leq p \leq n$, but no position k such that $\chi(i, k)$ and $i \gtrdot k$ or $i \doteq k$.

The formal semantics of all POTL operators remains the same as in Section 3.1. The only difference in its intuitive meaning is caused by open chains. Due to the change in property 4, χ^u_F operators never hold on the left contexts of open chains, and χ^d_F may hold only in positions that are also left contexts of some closed chain, with contexts in the $\lessdot$ relation (provided the operand holds in the right context). Downward hierarchical operators also never hold when evaluated on left contexts of open chains.

3.3 Expressing Requirements in POTL

POTL can express many useful requirements of procedural programs: to illustrate its practical applications in automatic verification, we supply a few examples of typical program properties expressed as POTL formulas. In Section 7 we will show the outcomes of checking these—and many other—formulas against several benchmark programs.

POTL can express Hoare-style pre/postconditions. For instance, formula $\Box(\mathbf{call} \wedge \rho \implies \chi^d_F(\mathbf{ret} \wedge \theta))$ specifies that if the precondition ρ holds when a procedure is called, then the postcondition θ must hold when it returns. This formula is false if a call is terminated by an exception.

Unlike NWTL, POTL can easily express properties related to exception handling and interrupt management. The shortcut $\mathit{CallThr}(\psi) := \bigcirc^u(\mathbf{exc} \wedge \psi) \vee \chi^u_F(\mathbf{exc} \wedge \psi)$, evaluated in a **call**, states that the procedure currently invoked is terminated by an **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 (or basic) 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 [36, 51] is an important class of requirements that state something about the sequence of procedures active in the program's stack at a certain point of its execution. They can be expressed with shortcut $\mathit{Scall}(\varphi, \psi) := (\mathbf{call} \implies \varphi)\ \mathcal{S}^d_\chi\ (\mathbf{call} \wedge \psi)$, which means that ψ holds in a **call** representing one of the currently active function frames, and φ holds in all **call**s in the stack between it and the current position. This shortcut has the same purpose of the *call since* operator of CaRet, which is also a since operator on **call**s currently in the stack, but thanks to other POTL operators, it works with exceptions too. For instance, $\Box\left((\mathbf{call} \wedge \mathrm{p}_B \wedge \mathit{Scall}(\top, \mathrm{p}_A)) \implies \mathit{CallThr}(\top)\right)$ 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.

With reference to Example 2.10 formula $\chi^u_F(\mathbf{ret} \wedge \texttt{main})$ must hold in position 1 to guarantee that the program terminates on any input array, and formula $\chi^u_F(\texttt{sorted})$ —with an obvious definition of *sorted*— specifies that it is correct w.r.t. the sorting goal. We will see in Section 7.2.2 that they are not guaranteed, i.e. the program is "buggy".

For a thorough comparison of POTL with other temporal logics for structured languages see [27, Section 3.4].

4 FINITE-WORD MODEL CHECKING

The model checking procedure we give for POTL follows the classic automata-theoretic approach for LTL, adapting it to work with OPA for the finite-word case, and ωOPBA for the infinite-word case. Thus, we define a construction for automata that accept models of an arbitrary POTL formula, and prove its correctness. This construction is significantly more involved than the one for LTL and reflects the differences between regular languages and OPLs, although the final automaton size remains singly exponential in formula length.

We give the finite-word construction in this section, and then we adapt it to ω-words in Section 5. In Section 4.1 we describe the construction, and we prove its correctness in Section 4.2. Finally, Section 4.3 analyzes the computational complexity of POTL satisfiability and model checking.

4.1 Automaton Construction

Given an OP alphabet $(\mathcal{P}(AP), M_{AP})$ and a formula φ, we build an OPA $\mathcal{A}_\varphi = (\mathcal{P}(AP), M_{AP}, Q, I, F, \delta)$. We describe the construction of Q, I, F and δ based on fixpoint computations that build these sets starting from a set of constraints. Since POTL contains a large number of operators, some of which being quite complex, we define and explain constraints related to each operator separately. Their correctness proof follows the same strategy: we prove a correctness lemma for each operator, and finally combine them to prove correctness of the whole construction.

Following a fairly classical path, we begin by introducing the *closure* of φ, named $\mathrm{Cl}(\varphi)$, containing all subformulas of φ. The states of the automaton will be sets of formulas associated to a word position. To do so, however, we need to augment POTL's alphabet with a few auxiliary operators which are not needed to increase POTL's expressiveness but are useful to specify the behavior of its model checker. Such new operators are:

- ζ_L, which forces the current position to be the first one of a chain body;
- ζ_R, which lets the computation go on only if the previous transition was a pop, and the position associated with the current state is the right context of a chain;
- $\zeta_{\doteq}$, which appears in a state iff the next transition will be a shift.
- Furthermore we will use χ^π_F, χ^π_P, where π is a PR, as a kind of "specialization" of the original χ^t_F, χ^t_P: for instance, whereas $(w, i) \models \chi^d_F\, \psi$ iff there exists a position $j > i$ such that $\chi(i, j)$, $i \lessdot j$ or $i \doteq j$, and $(w, j) \models \psi$, we now need to treat separately the two cases $i \lessdot j$ and $i \doteq j$ so that the two PRs now replace the superscript 'd'. $\chi^{\lessdot}_P$ and $\chi^{\gtrdot}_F$ have already been formally defined and used in the expansion laws for

hierarchical until operators (cf. Section 3.1) as short-notations for more involved formulas that required explicitly the appropriate PR; their $\doteq$ counterparts are defined analogously.

The precise semantics of the above new operators will be formalized when they will be used.

Thus, we obtain $\mathrm{Cl}(\varphi)$ through a fixpoint computation. It is the smallest set satisfying the following constraints:

- $\varphi \in \mathrm{Cl}(\varphi)$,
- $AP \subseteq \mathrm{Cl}(\varphi)$,
- if $\psi \in \mathrm{Cl}(\varphi)$ and $\psi \neq \neg\theta$, then $\neg\psi \in \mathrm{Cl}(\varphi)$ (we identify $\neg\neg\psi$ with ψ);
- if $\neg\psi \in \mathrm{Cl}(\varphi)$, then $\psi \in \mathrm{Cl}(\varphi)$;
- if any of the unary temporal operators (e.g., $\bigcirc^d$, χ_F^d, χ_P^d, $\bigcirc_H^d$, …) is in $\mathrm{Cl}(\varphi)$, and ψ is its operand, then $\psi \in \mathrm{Cl}(\varphi)$;
- if any of the binary operators (e.g., $\wedge$, $\vee$, $\mathcal{U}_\chi^d$, $\mathcal{S}_\chi^d$, $\mathcal{U}_H^d$, …) is in $\mathrm{Cl}(\varphi)$, and ψ and θ are its operands, then $\psi, \theta \in \mathrm{Cl}(\varphi)$;
- if $\chi_F^d\,\psi \in \mathrm{Cl}(\varphi)$, then $\zeta_L, \chi_F^{\lessdot}\psi, \chi_F^{\doteq}\psi \in \mathrm{Cl}(\varphi)$;
- if $\chi_F^u\,\psi \in \mathrm{Cl}(\varphi)$, then $\zeta_L, \chi_F^{\gtrdot}\psi, \chi_F^{\doteq}\psi \in \mathrm{Cl}(\varphi)$;
- if $\chi_P^d\,\psi \in \mathrm{Cl}(\varphi)$, then $\zeta_R, \chi_P^{\lessdot}\psi, \chi_P^{\doteq}\psi \in \mathrm{Cl}(\varphi)$;
- if $\chi_P^u\,\psi \in \mathrm{Cl}(\varphi)$, then $\zeta_R, \zeta_{\doteq}, \chi_P^{\gtrdot}\psi, \chi_P^{\doteq}\psi \in \mathrm{Cl}(\varphi)$;
- if $\bigcirc_H^u\,\psi \in \mathrm{Cl}(\varphi)$, then $\zeta_R \in \mathrm{Cl}(\varphi)$;
- if $\ominus_H^u\,\psi \in \mathrm{Cl}(\varphi)$, then $\zeta_L, \zeta_R \in \mathrm{Cl}(\varphi)$;
- if $\theta \in \mathrm{Cl}(\varphi)$ such that $\theta = \bigcirc_H^d\,\psi$ or $\theta = \ominus_H^d\,\psi$, then $\zeta_L, \zeta_{\doteq}, (\ominus^d\psi \vee \chi_P^{\lessdot}\psi), (\ominus^d\theta \vee \chi_P^{\lessdot}\theta) \in \mathrm{Cl}(\varphi)$;
- if any until or since operator is in $\mathrm{Cl}(\varphi)$, then all operators required by its expansion law (cf. Section 3.1) are in $\mathrm{Cl}(\varphi)$. E.g., if $\psi\,\mathcal{U}^t\,\theta \in \mathrm{Cl}(\varphi)$ for $t \in \{d, u\}$, then $\bigcirc^t(\psi\,\mathcal{U}^t\,\theta), \chi_F^t(\psi\,\mathcal{U}^t\,\theta) \in \mathrm{Cl}(\varphi)$.

Next, we define the set $\mathrm{Atoms}(\varphi)$, which contains all *consistent* subsets of $\mathrm{Cl}(\varphi)$, i.e., all $\Phi \subseteq \mathrm{Cl}(\varphi)$ that satisfy a set of *Atomic consistency Constraints* $\mathscr{AC}$. What constraints are in $\mathscr{AC}$ depends on which operators appear in $\mathrm{Cl}(\varphi)$. In the following, we introduce constraints that appear in $\mathscr{AC}$ due to each operator's presence in $\mathrm{Cl}(\varphi)$ separately. We start by defining those related to propositional operators: for any $\Phi \in \mathrm{Atoms}(\varphi)$,

(a) $\psi \in \Phi$ iff $\neg\psi \notin \Phi$ for every $\psi \in \mathrm{Cl}(\varphi)$;
(b) $\psi \wedge \theta \in \Phi$, iff $\psi \in \Phi$ and $\theta \in \Phi$;
(c) $\psi \vee \theta \in \Phi$, iff $\psi \in \Phi$ or $\theta \in \Phi$, or both.

While (a) is always present in $\mathscr{AC}$, (b) and (c) only appear if any formula involving resp. $\wedge$ or $\vee$ is in $\mathrm{Cl}(\varphi)$.

$\mathcal{A}_\varphi$ works in a way similar to the classic LTL tableau [79]: each state contains an atom with formulas that hold in the next time instant, and transitions read the set of APs in that atom, guessing the next one. Whereas LTL and FSAs are strictly sequential, POTL is devised to express properties of tree-shaped structures and OPAs are pushdown machines. Furthermore OPAs, in general, are not real-time machines—unlike VPAs—, i.e., they also have pop transitions which do not read any character. Thus, $\mathcal{A}_\varphi$'s states are obtained by pairing atoms with another subset of $\mathrm{Cl}(\varphi)$ which aims at keeping track of what will happen (resp. happened) at the end (resp. beginning) of a chain. Formally, such a set of *pending formulas* is defined as

$$\mathrm{Cl}_{pend}(\varphi) = \left\{\theta \in \mathrm{Cl}(\varphi) \mid \theta \in \{\zeta_L, \zeta_R, \zeta_{\doteq}, \chi_F^\pi\,\psi, \chi_P^\pi\,\psi, \bigcirc_H^t\,\psi, \ominus_H^t\,\psi\} \text{ for some } \pi \in \{\lessdot, \doteq, \gtrdot\},\ t \in \{d, u\} \text{ and } \psi \in \mathrm{Cl}(\varphi)\right\}.$$

The states of $\mathcal{A}_\varphi$ are the set $Q = \mathrm{Atoms}(\varphi) \times \mathcal{P}(\mathrm{Cl}_{pend}(\varphi))$, and its elements, which we denote with Greek capital letters, are of the form $\Phi = (\Phi_c, \Phi_p)$, where Φ_c, called the *current* part of Φ, is the set of formulas that hold in the next position that $\mathcal{A}_\varphi$ is going to read, and Φ_p, or the *pending* part of Φ, is a set of *temporal obligations*. Φ_p keeps track of temporal operators such as $\chi_F^t\,\psi$ that, once guessed in a position i that is the left context of a chain, is satisfied by ψ holding in position j that is in the χ relation with i. States with pending formulas can be pushed to the stack, so that when they are popped the OPA "remembers" that some temporal operator must be satisfied.

The initial set I contains states of the form (Φ_c, Φ_p), with $\varphi \in \Phi_c$, and the final set F contains states of the form (Ψ_c, Ψ_p), s.t. $\Psi_c \cap AP = \{\#\}$ and Ψ_c contains no future operators. Φ_p and Ψ_p may contain only operators explicitly allowed in the following.

In the following, we use a notation that relates states to word positions. With this notation, $\Phi(i)$ is a "look-ahead" for a_i, which is the next symbol to be read. State $\Phi(i)$ is the one that guesses, in its Φ_c component, the formulas holding in position i, and is produced directly by the push or shift move reading position $i-1$ (in particular, the initial state is $\Phi(1)$). Since OPA's pop moves do not read any character, we introduce the notation $\Phi^g(i)$ to distinguish the state in which the automaton is ready to read the symbol in position i. Thus, when $\mathcal{A}_\varphi$ reads the symbol in position $i-1$, it goes from configuration $\langle a_{i-1}a_i x, \Phi^g(i-1), \gamma\rangle$ to $\langle a_i x, \Phi(i), \gamma'\rangle$, where a_i is the input symbol at position i; in its pending part, $\Phi(i)$ may contain further guesses on formulas holding beyond position i due to chain-next operators. Precisely, if $smb(\gamma') \lessdot a_i$ or $smb(\gamma') \doteq a_i$, then the next move is respectively a push or a shift, and $\Phi^g(i) = \Phi(i)$, bringing $\mathcal{A}_\varphi$ to $\langle x, \Phi(i+1), \gamma''\rangle$. If $smb(\gamma') \gtrdot a_i$, a pop transition occurs before reading the symbol in position i, and $\mathcal{A}_\varphi$ checks previous guesses and makes new ones about formulas holding in positions beyond i: the next state is called $\Phi'(i)$, and if more pops occur, we have similarly $\Phi''(i)$, $\Phi'''(i)$, etc.

The state resulting from the last pop before a_i is read by a shift or a push is $\Phi^g(i)$. For instance, if $\gamma' = [a_{j'}, \Phi^g(j)]$ $[a_{k'}, \Phi^g(k)]\gamma''$ and $a_{j'}, a_{k'} \gtrdot a_i$ for some $k \le k' < j \le j' < i$,[5] two pop moves occur and i is read by a push, causing the following sequence of transitions:

$$\rho = \langle a_i x, \Phi(i), [a_{j'}, \Phi^g(j)][a_{k'}, \Phi^g(k)]\gamma''\rangle \vdash \langle a_i x, \Phi'(i), [a_{k'}, \Phi^g(k)]\gamma''\rangle \vdash \langle a_i x, \Phi''(i), \gamma''\rangle \vdash \langle x, \Phi(i{+}1), [a_i, \Phi''(i)]\gamma''\rangle$$

At this point, $\Phi''(i)$ is also referred to as $\Phi^g(i)$, and the last push transition occurs.

Temporal obligations are enforced by the transition relation δ. As well as $\mathrm{Atoms}(\varphi)$ is the set of all subsets of $\mathrm{Cl}(\varphi)$ that satisfy the consistency constraints in $\mathscr{AC}$, the transition relation δ is the set of all transitions that satisfy a set of *δ-rules*, $\mathscr{DR}$. We will introduce $\mathscr{DR}$ in parallel with $\mathscr{AC}$ gradually for each operator: δ_{push} and δ_{shift} are the largest subsets of $Q \times \mathcal{P}(AP) \times Q$ satisfying all rules in $\mathscr{DR}$, and δ_{pop} is the largest subset of $Q \times Q \times Q$ satisfying all rules in $\mathscr{DR}$. Given a formula ψ, we denote as $\mathscr{DR}(\psi)$ the set of $\mathscr{DR}$ rules that are defined as a consequence of $\psi \in \mathrm{Cl}(\varphi)$. First, we introduce two $\mathscr{DR}$ rules that are always present and are not bound to a particular operator.

Each state of $\mathcal{A}_\varphi$ guesses the APs that will be read next. So, $\mathscr{DR}$ always contains the rule that

(1) for any $(\Phi, a, \Psi) \in \delta_{push/shift}$, with $\Phi, \Psi \in Q$ and $a \in \mathcal{P}(AP)$, we have $\Phi_c \cap AP = a$

(by $\delta_{push/shift}$ we mean $\delta_{push} \cup \delta_{shift}$, and by $\Phi_c \cap AP$ the set of atomic propositions in Φ_c). *Pop* moves, on the other hand, do not read input symbols, and $\mathcal{A}_\varphi$ remains at the same position when performing them: $\mathscr{DR}$ contains the rule

(2) for any $(\Phi, \Theta, \Psi) \in \delta_{pop}$ it must be $\Phi_c = \Psi_c$.

Referring to the above sequence ρ, thanks to $\mathscr{DR}$ rule (1), $\Phi_c(i) \cap AP = a_i$, and due to rule (2) we have $\Phi_c(i) = \Phi'_c(i) = \Phi''_c(i)$ —i.e., only pending parts can change during pop moves—.

Next, we examine all POTL's operators and derive $\mathscr{DR}$ rules therefrom with an informal explanation of their rationale. Then, in Section 4.2, we provide a formal correctness statement and proof thereof. Since the involved POTL operators are more complex than those of classic LTL —and of NWTL too— we structure such a correctness proof into a sequence of lemmas, one for each operator, followed by a global proof based on a natural induction on formula structure.

4.1.1 Next and Back Operators. If $\bigcirc^d \psi \in \mathrm{Cl}(\varphi)$ for some ψ, $\mathscr{DR}(\bigcirc^d \psi)$ contains a rule imposing that:

(3) for all $(\Phi, a, \Psi) \in \delta_{push/shift}$, it must be that $\bigcirc^d \psi \in \Phi_c$ iff ($\psi \in \Psi_c$ and either $a \lessdot b$ or $a \doteq b$, where $b = \Psi_c \cap AP$).

[5] Recall that the input symbol contained in stack elements can be changed by shift moves, leaving the state unchanged: this is why we need two more positions j' and k' besides j and k.

step	input	state	stack	PR	move
1	**call call exc** #	$\Phi^g(1) = (\{\textbf{call}, \bigcirc^d \bigcirc^u \textbf{exc}\}, \emptyset)$	$\bot$	# ⋖ **call**	push
2	**call exc** #	$\Phi^g(2) = (\{\textbf{call}, \bigcirc^u \textbf{exc}\}, \emptyset)$	$[\textbf{call}, \Phi^g(1)]\bot$	**call** ⋖ **call**	push
3	**exc** #	$\Phi(3) = (\{\textbf{exc}\}, \emptyset)$	$[\textbf{call}, \Phi^g(2)][\textbf{call}, \Phi^g(1)]\bot$	**call** ⋗ **exc**	pop
4	**exc** #	$\Phi'(3) = (\{\textbf{exc}\}, \emptyset)$	$[\textbf{call}, \Phi^g(1)]\bot$	**call** ⋗ **exc**	pop
5	**exc** #	$\Phi''(3) = \Phi^g(3) = (\{\textbf{exc}\}, \emptyset)$	$\bot$	# ⋖ **exc**	push
6	#	$\Phi(4) = (\{\#\}, \emptyset)$	$[\textbf{exc}, \Phi^g(3)]\bot$	**exc** ⋗ #	pop
7	#	$\Phi'(4) = \Phi^g(4) = (\{\#\}, \emptyset)$	$\bot$	–	–

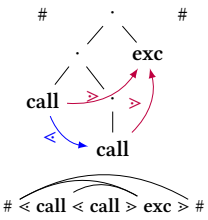


Fig. 8. Example run of the automaton for $\bigcirc^d \bigcirc^u$ **exc** (left), and ST (top right) and flat representation of the input word (bottom right).

For $\ominus^d \psi$, the rule is symmetric, i.e. the double implication is $\ominus^d \psi \in \Psi_c$ iff ($\psi \in \Phi_c$ and $a \lessdot b$ or $a \doteq b$). For the upward counterparts it suffices to replace ⋖ with ⋗ in the previous rules.

Not surprisingly, the above rules involve transitions that read an input symbol in a similar way as for LTL and FSAs. It also immediately appears, however, that there are conditions on PRs to consider, with an impact on the way the ST associated to any sentence is visited. If $t = d$, when a push or a shift transition reads a position where $\bigcirc^d \psi$ is guessed to hold, it only leads to a state containing APs such that the read label is in the ⋖ or $\doteq$ PR with them. This means the OPA guesses that the next transition will be, again, a push or a shift, and will read a symbol where ψ holds.

Things get slightly more complicated if $t = u$ because pop transitions may occur, so we illustrate this case with an example. Fig. 8 shows an accepting run of $\mathcal{A}_\varphi$, built on $(\Sigma_{\text{call}}, M_{\text{call}})$ with $\varphi = \bigcirc^d \bigcirc^u$ **exc**, on one of its models (but others exist). $\mathcal{A}_\varphi$ starts in a state $\Phi^g(1)$ with $\bigcirc^d \bigcirc^u$ **exc** in its current part, which is required for all initial states. A push move reads the first position, guessing that the next one will be a **call**, and state $\Phi^g(2)$ is reached. We have $\bigcirc^d \bigcirc^u$ **exc** $\in \Phi_c^g(1)$ and $\bigcirc^u$ **exc** $\in \Phi_c^g(2)$, and also **call** ⋖ **call**, so rule (3) is satisfied. The next position is again read by a push, but this time the next state is $\Phi(3)$, which guesses the next position being a **exc**. **call** ⋗ **exc** satisfies the u version of rule (3), but it means that the next move will be a pop. Indeed, the OPA pops all symbols in the stack, but because of $\mathscr{DR}$ rule (2), the current part of the state does not change, so the guess that **exc** will hold in the third position is preserved. Note that the pending parts of states are always empty, because they are not needed for this operator. Also, notice the ^{g} notation, which we use to distinguish states that are ready for a push or a shift move. States $\Phi(1)$ and $\Phi(2)$ can be also denoted as $\Phi^g(1)$ and $\Phi^g(2)$ respectively, while $\Phi^g(3) = \Phi''(3)$ and $\Phi^g(4) = \Phi'(4)$.

4.1.2 Chain Next Operators. To model check chain next operators we use auxiliary operators χ_F^π, with $\pi \in \{\lessdot, \doteq, \gtrdot\}$, that restrict their downward and upward counterparts to a single PR. Their semantics can be defined as follows: given an OP word w and a position i, we have $(w, i) \models \chi_F^\pi \psi$ iff there exists a position $j > i$ such that $\chi(i, j)$ and $i \; \pi \; j$, and $(w, j) \models \psi$. In particular, we have $\chi_F^d \psi \iff \chi_F^\lessdot \psi \vee \chi_F^\doteq \psi$ and $\chi_F^u \psi \iff \chi_F^\gtrdot \psi \vee \chi_F^\doteq \psi$, which justify the $\mathscr{AC}$ constraints below.
If $\chi_F^d \psi \in \text{Cl}(\varphi)$, $\mathscr{AC}$ contains the following constraint:

(d) for each $\Phi \in Q$ we have $\chi_F^d \psi \in \Phi_c$ iff ($\chi_F^\lessdot \psi \in \Phi_c$ or $\chi_F^\doteq \psi \in \Phi_c$).

For $\chi_F^u \psi \in \text{Cl}(\varphi)$, the constraint becomes

(e) for each $\Phi \in Q$ we have $\chi_F^u \psi \in \Phi_c$ iff ($\chi_F^\doteq \psi \in \Phi_c$ or $\chi_F^\gtrdot \psi \in \Phi_c$).

We also use the auxiliary symbol ζ_L to force the next position to be read to be the first one of a chain body. If we let the current state of $\mathcal{A}_\varphi$ be $\Phi \in Q$, then $\zeta_L \in \Phi_p$ iff the upcoming transition (i.e. the one reading the next position) is a push. This is accomplished by the following rules in $\mathscr{DR}(\zeta_L)$:

(4) if $(\Phi, a, \Psi) \in \delta_{shift}$ or $(\Phi, \Theta, \Psi) \in \delta_{pop}$, for any Φ, Θ, Ψ and a, then $\zeta_L \notin \Phi_p$;

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

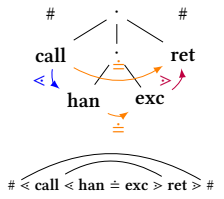


Fig. 9. Example run of the automaton for χ_F^d **ret** (left), and ST (top right) and flat representation of the input word (bottom right).

(5) if $(\Phi, a, \Psi) \in \delta_{push}$, then $\zeta_L \in \Phi_p$.

Moreover, for any initial state $(\Phi_c, \Phi_p) \in I$, we have $\zeta_L \in \Phi_p$ iff $\# \notin \Phi_c$.

If $\chi_F^{\doteq}\,\psi \in \mathrm{Cl}(\varphi)$, its satisfaction is ensured by the following rules in $\mathscr{DR}(\chi_F^{\doteq}\,\psi)$:

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

If $\chi_F^{\lessdot}\,\psi \in \mathrm{Cl}(\varphi)$, then $\chi_F^{\lessdot}\,\psi$ is allowed in the pending part of initial states, and $\mathscr{DR}(\chi_F^{\lessdot}\,\psi)$ contains the following rules:

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

The rules for $\chi_F^{\gtrdot}\,\psi$ only differ in ψ being enforced by a pop transition, triggered by the $\gtrdot$ relation between the left and right contexts of the chain on whose left context $\chi_F^{\gtrdot}\,\psi$ holds. Thus, if $\chi_F^{\gtrdot}\,\psi \in \mathrm{Cl}(\varphi)$, in $\mathscr{DR}(\chi_F^{\gtrdot}\,\psi)$ we have:

(11) Let $(\Phi, a, \Psi) \in \delta_{push/shift}$: then $\chi_F^{\gtrdot}\,\psi \in \Phi_c$ iff $\chi_F^{\gtrdot}\,\psi, \zeta_L \in \Psi_p$;
(12) let $(\Phi, \Theta, \Psi) \in \delta_{pop}$: $(\chi_F^{\gtrdot}\,\psi \in \Theta_p$ iff $\chi_F^{\gtrdot}\,\psi \in \Psi_p)$ and $(\chi_F^{\gtrdot}\,\psi \in \Phi_p$ iff $\psi \in \Phi_c)$;
(13) let $(\Phi, a, \Psi) \in \delta_{shift}$: then $\chi_F^{\gtrdot}\,\psi \notin \Phi_p$.

We illustrate how the construction works for $\chi_F^{\doteq}$ with the example of Fig. 9, which shows an accepting run of $\mathcal{A}_\varphi$ for $\varphi = \chi_F^d$ **ret**. The OPA starts in state $\Phi^g(1)$, with χ_F^d **ret** $\in \Phi_c^g(1)$, and guesses that χ_F^d will be fulfilled by $\chi_F^{\doteq}$, so $\chi_F^{\doteq}$ **ret** $\in \Phi_c^g(1)$, and that the next move will be a push, so $\zeta_L \in \Phi_p^g(1)$. **call** is read by a push move, resulting in state $\Phi^2(2)$. Again, the OPA guesses the next move will be a push, so $\zeta_L \in \Phi_p^g(2)$. By rule (6), we have $\chi_F^{\doteq}$ **ret** $\in \Phi_p^g(2)$. The last guess is immediately verified by the next push (step 2-3). Thus, the pending obligation for $\chi_F^{\doteq}$ **ret** is stored onto the stack in $\Phi^g(2)$. The OPA, then, reads **exc** with a shift, and pops the stack symbol containing $\Phi^g(2)$ (step 4-5). By rule (7), the temporal obligation is resumed in the next state $\Phi'(4)$, so $\chi_F^{\doteq}$ **ret** $\in \Phi'_p(4)$. Finally, **ret** is read by a shift which, by rule (8), may occur only if **ret** $\in \Phi'_c(4)$. Rule (8) verifies the guess that $\chi_F^{\doteq}$ **ret** holds in $\Phi^g(1)$, and fulfills the temporal obligation contained in $\Phi'_p(4)$, by preventing computations in which **ret** $\notin \Phi'_c(4)$ from continuing. Had the next transition been a pop (e.g., because there was no **ret** and **call** $\gtrdot$ #), the run would have been blocked by rule (7), preventing the OPA from reaching an accepting state.

4.1.3 Chain Back Operators. Despite the structure of chains in OPLs being symmetric, the way chain back operators work is quite different from chain next operators, because OPAs proceed left-to-right. Hence, while the OPA has to guess the presence of a $\chi_F^t\,\psi$ because ψ will hold in the future, with $\chi_P^t\,\psi$ the argument ψ is found first, so the OPA must make sure the chain back will hold in the future.

To model check the $\chi_P^d\,\psi$ and $\chi_P^u\,\psi$ operators, we employ the auxiliary operator $\chi_P^\pi\,\psi$, with $\pi \in \{\lessdot, \doteq, \gtrdot\}$. Given an OP word w and a position i in it, we have $(w, i) \models \chi_P^\pi\,\psi$ iff there exists a position $j < i$ such that $\chi(j, i)$

step	input	state	stack	PR	move
1	**call han exc ret #**	$\Phi^g(1) = (\{\textbf{call}, \bigcirc^d \bigcirc^d \bigcirc^u \chi_P^d\,\textbf{call}\}, \emptyset)$	$\bot$	# ⋖ **call**	push
2	**han exc ret #**	$\Phi^g(2) = (\{\textbf{han}, \bigcirc^d \bigcirc^u \chi_P^d\,\textbf{call}\}, \{\chi_P^{\doteq}\,\textbf{call}\})$	$[\textbf{call}, \Phi^g(1)]\bot$	**call** ⋖ **han**	push
3	**exc ret #**	$\Phi^g(3) = (\{\textbf{exc}, \bigcirc^u \chi_P^d\,\textbf{call}\}, \emptyset)$	$[\textbf{han}, \Phi^g(2)][\textbf{call}, \Phi^g(1)]\bot$	**han** ≐ **exc**	shift
4	**ret #**	$\Phi(4) = (\{\textbf{ret}, \chi_P^d\,\textbf{call}, \chi_P^{\doteq}\,\textbf{call}\}, \emptyset)$	$[\textbf{exc}, \Phi^g(2)][\textbf{call}, \Phi^g(1)]\bot$	**exc** ⋗ **ret**	pop
5	**ret #**	$\Phi'(4) = \Phi^g(4) = (\{\textbf{ret}, \chi_P^d\,\textbf{call}, \chi_P^{\doteq}\,\textbf{call}\}, \{\chi_P^{\doteq}\,\textbf{call}, \zeta_R\})$	$[\textbf{call}, \Phi^g(1)]\bot$	**call** ≐ **ret**	shift
6	#	$\Phi(5) = (\{\#\}, \emptyset)$	$[\textbf{ret}, \Phi^g(1)]\bot$	**ret** ⋗ #	pop
7	#	$\Phi'(5) = \Phi(5) = (\{\#\}, \emptyset)$	$\bot$	–	–

Fig. 10. Example run of the automaton for $\bigcirc^d \bigcirc^d \bigcirc^u \chi_P^d\,\textbf{call}$ on the same word as in Fig. 9.

and $j\ \pi\ i$, and $(w, j) \models \psi$. The $\mathscr{AC}$ constraints below rely on the equivalences $\chi_P^d\,\psi \iff \chi_P^{\lessdot}\,\psi \vee \chi_P^{\doteq}\,\psi$ and $\chi_P^u\,\psi \iff \chi_P^{\gtrdot}\,\psi \vee \chi_P^{\doteq}\,\psi$.

If $\chi_P^d\,\psi \in \mathrm{Cl}(\varphi)$, $\mathscr{AC}$ contains the following constraint:

(f) for any $\Phi \in Q$ we have $\chi_P^d\,\psi \in \Phi_c$ iff ($\chi_P^{\doteq}\,\psi \in \Phi_c$ or $\chi_P^{\lessdot}\,\psi \in \Phi_c$).

For $\chi_P^u\,\psi \in \mathrm{Cl}(\varphi)$, $\mathscr{AC}$ contains

(g) for any $\Phi \in Q$ we have $\chi_P^u\,\psi \in \Phi_c$ iff ($\chi_P^{\doteq}\,\psi \in \Phi_c$ or $\chi_P^{\gtrdot}\,\psi \in \Phi_c$).

We use symbol ζ_R, which is symmetric to ζ_L: it lets the computation go on only if the previous transition was a pop, and the next position to be read is the right context of a chain. So, we define the following $\mathscr{DR}(\zeta_R)$ rules:

(14) for any $(\Phi, a, \Psi) \in \delta_{push/shift}$, we have $\zeta_R \notin \Psi_p$;
(15) for any $(\Phi, \Theta, \Psi) \in \delta_{pop}$, we have $\zeta_R \in \Psi_p$.

ζ_R is allowed in the pending part of final states.

If $\chi_P^{\doteq}\,\psi \in \mathrm{Cl}(\varphi)$, $\mathscr{DR}(\chi_P^{\doteq}\,\psi)$ contains the following rules:

(16) Let $(\Phi, a, \Psi) \in \delta_{shift}$: then $\chi_P^{\doteq}\,\psi \in \Phi_c$ iff $\chi_P^{\doteq}\,\psi, \zeta_R \in \Phi_p$;
(17) let $(\Phi, a, \Psi) \in \delta_{push}$: then $\chi_P^{\doteq}\,\psi \notin \Phi_c$;
(18) let $(\Phi, \Theta, \Psi) \in \delta_{pop}$: then $\chi_P^{\doteq}\,\psi \in \Psi_p$ iff $\chi_P^{\doteq}\,\psi \in \Theta_p$;
(19) let $(\Phi, a, \Psi) \in \delta_{push/shift}$: then $\chi_P^{\doteq}\,\psi \in \Psi_p$ iff $\psi \in \Phi_c$.

The rules in $\mathscr{DR}(\chi_P^{\lessdot}\,\psi)$ if $\chi_P^{\lessdot}\,\psi \in \mathrm{Cl}(\varphi)$ follow:

(20) Let $(\Phi, a, \Psi) \in \delta_{push}$: then $\chi_P^{\lessdot}\,\psi \in \Phi_c$ iff $\chi_P^{\lessdot}\,\psi, \zeta_R \in \Phi_p$;
(21) let $(\Phi, a, \Psi) \in \delta_{shift}$: then $\chi_P^{\lessdot}\,\psi \notin \Phi_c$;
(22) let $(\Phi, \Theta, \Psi) \in \delta_{pop}$: then $\chi_P^{\lessdot}\,\psi \in \Psi_p$ iff $\chi_P^{\lessdot}\,\psi \in \Theta_p$;
(23) let $(\Phi, a, \Psi) \in \delta_{push/shift}$: then $\chi_P^{\lessdot}\,\psi \in \Psi_p$ iff $\psi \in \Phi_c$.

Finally, for $\chi_P^{\gtrdot}$, we use symbol $\zeta_{\doteq}$, which appears in a state iff the next transition will be a shift. $\mathscr{DR}(\zeta_{\doteq})$ contains:

(24) for any $(\Phi, a, \Psi) \in \delta_{push}$ and $(\Phi, \Theta, \Psi) \in \delta_{pop}$, $\zeta_{\doteq} \notin \Phi_p$;
(25) for any $(\Phi, a, \Psi) \in \delta_{shift}$, $\zeta_{\doteq} \in \Phi_p$.

$\chi_P^{\gtrdot}\,\psi$ and $\zeta_{\doteq}$ are allowed in the pending part of final states.

If $\chi_P^{\gtrdot}\,\psi \in \mathrm{Cl}(\varphi)$, $\mathscr{DR}(\chi_P^{\gtrdot}\,\psi)$ contains the rules below:

For any $(\Phi, a, \Psi) \in \delta_{push/shift}$,

(26) $\chi_P^{\gtrdot}\,\psi \notin \Psi_p$;
(27) $\chi_P^{\gtrdot}\,\psi \in \Phi_c$ iff $\chi_P^{\gtrdot}\,\psi, \zeta_R \in \Phi_p$;

for any $(\Phi, \Theta, \Psi) \in \delta_{pop}$,

(28) if ($\zeta_L \in \Psi_p$ or $\zeta_{\doteq} \in \Psi_p$), then $\chi_P^{\gtrdot}\,\psi \in \Psi_p$ iff $\chi_P^{\gtrdot}\,\psi \in \Phi_p$;
(29) if $\zeta_L, \zeta_{\doteq} \notin \Psi_p$, then $\chi_P^{\gtrdot}\,\psi \in \Psi_p$ iff (either $\chi_P^{\lessdot}\,\psi \vee \ominus^d\,\psi \in \Theta_c$ or $\chi_P^{\gtrdot}\,\psi \in \Phi_p$).

In Fig. 10, we show how the construction works through an example run of $\mathcal{A}_\varphi$ built for $\varphi = \bigcirc^d \bigcirc^d \bigcirc^u \chi_P^d$ **call** on the same word as in Fig. 9. Position 1 is read by a push move, and since **call** $\in \Phi^g(1)$, according to $\mathscr{DR}$ rule (19), $\chi_P^{\doteq}$ **call** is stored in the pending part of the next state $\Phi^g(2)$. Here the OPA has just made two guesses: that position 1 is the left context of a chain, and that in its right context χ_P^d **call** will be fulfilled by $\chi_P^{\doteq}$ **call**. The OPA then proceeds by reading position 2 with a push that stores $\Phi^g(2)$ on the stack: the guess about $\chi_P^{\doteq}$ **call** will be checked when it is popped. Next, **exc** in position 3 is read by a shift move that updates the character in the topmost stack symbol and, more importantly, guesses that $\chi_P^{\doteq}$ **call** will hold in position 4. Due to $\mathscr{AC}$ constraint (f), χ_P^d **call** is also in $\Phi_c(4)$, fulfilling the $\mathscr{DR}$ rule for the $\bigcirc^u$ operator. State $\Phi^g(2)$ is then popped, and since $\chi_P^{\doteq}$ **call** $\in \Phi_p^g(2)$, by rule (18) we have $\chi_P^{\doteq}$ **call** $\in \Phi'_p(4)$. By rule (15), $\zeta_R \in \Phi'_p(4)$. Thus, $\Phi'(4) = \Phi^g(4)$ contains all formulas needed for rule (16) to confirm the guess that $\chi_P^{\doteq}$ **call** holds in position 4. Note that rule (16) only holds for shift transitions, while rule (17) forbids push moves with $\chi_P^{\doteq}$ **call** in the current part of the starting state: this makes sure the two chain contexts are in the $\doteq$ relation.

4.1.4 Summary Until and Since. The construction for these operators relies on $\mathscr{AC}$ constraints based on their expansion laws (cf. Section 3.1). The constraints for until follow, and those for since are symmetric.

(h) For any $\Phi \in Q$, we have $\psi \,\mathcal{U}^t\, \theta \in \Phi_c$, with $t \in \{d, u\}$ being a direction, iff either:
- $\theta \in \Phi_c$, or
- $\bigcirc^t(\psi \,\mathcal{U}^t\, \theta), \psi \in \Phi_c$, or
- $\chi_F^t(\psi \,\mathcal{U}^t\, \theta), \psi \in \Phi_c$.

4.1.5 Hierarchical Next and Back Operators. If $\bigcirc_H^u \psi \in \mathrm{Cl}(\varphi)$ for some ψ, $\mathscr{DR}(\bigcirc_H^u \psi)$ contains the following rules:
for any $(\Phi, a, \Psi) \in \delta_{push}$,

(30) if $\bigcirc_H^u \psi \in \Phi_c$, then $\zeta_R \in \Phi_p$;
(31) $\bigcirc_H^u \psi \in \Phi_p$ iff ($\psi \in \Phi_c$ and $\zeta_R \in \Phi_p$);

for any $(\Phi, \Theta, \Psi) \in \delta_{pop}$,

(32) if $\zeta_R \in \Theta_p$, then $\bigcirc_H^u \psi \in \Theta_c$ iff $\bigcirc_H^u \psi \in \Psi_p$;
(33) $\bigcirc_H^u \psi \notin \Phi_p$;

for any $(\Phi, a, \Psi) \in \delta_{shift}$:

(34) $\bigcirc_H^u \psi \notin \Phi_p$ and $\bigcirc_H^u \psi \notin \Phi_c$.

If $\ominus_H^u \psi \in \mathrm{Cl}(\varphi)$ for some ψ, $\mathscr{DR}(\ominus_H^u \psi)$ contains the following rules:

(35) for any $(\Phi, a, \Psi) \in \delta_{push}$, if $\ominus_H^u \psi \in \Phi_c$, then $\zeta_R, \zeta_L \in \Phi_p$;
(36) for any $(\Phi, \Theta, \Psi) \in \delta_{pop}$, if $\zeta_L \in \Psi_p$, then $\ominus_H^u \psi \in \Psi_c$ iff $\psi \in \Theta_c$ and $\zeta_R \in \Theta_p$;
(37) for any $(\Phi, a, \Psi) \in \delta_{shift}$, $\ominus_H^u \psi \notin \Phi_c$.

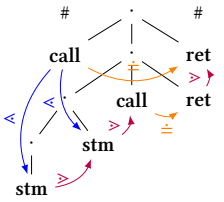
If $\bigcirc_H^d \psi \in \mathrm{Cl}(\varphi)$ for some ψ, $\mathscr{DR}(\bigcirc_H^d \psi)$ contains the following rules:
for any $(\Phi, \Theta, \Psi) \in \delta_{pop}$,

(38) if $\zeta_L, \zeta_{\doteq} \notin \Psi_p$, then $(\ominus^d \psi \vee \chi_P^{\lessdot} \psi) \in \Theta_c$ iff $\bigcirc_H^d \psi \in \Psi_p$;
(39) if $\zeta_L, \zeta_{\doteq} \notin \Psi_p$, then $\bigcirc_H^d \psi \in \Phi_p$ iff $(\ominus^d(\bigcirc_H^d \psi) \vee \chi_P^{\lessdot}(\bigcirc_H^d \psi)) \in \Theta_c$;
(40) if $(\ominus^d(\bigcirc_H^d \psi) \vee \chi_P^{\lessdot}(\bigcirc_H^d \psi)) \in \Theta_c$, then $\zeta_{\doteq} \notin \Psi_p$;

for any $(\Phi, a, \Psi) \in \delta_{push/shift}$,

(41) if $\bigcirc_H^d \psi \in \Phi_c$, then $\zeta_L \in \Psi_p$;
(42) $\bigcirc_H^d \psi \notin \Psi_p$.

step	input	state	stack	PR	move
1	**call stm stm call ret ret #**	$\Phi^g(1) = (\{\mathbf{call}, \bigcirc^d \bigcirc^u \bigcirc^u_H \mathbf{call}\}, \emptyset)$	$\bot$	$\# \lessdot \mathbf{call}$	push
2	**stm stm call ret ret #**	$\Phi^g(2) = (\{\mathbf{stm}, \bigcirc^u \bigcirc^u_H \mathbf{call}\}, \emptyset)$	$[\mathbf{call}, \Phi^g(1)]\bot$	$\mathbf{call} \lessdot \mathbf{stm}$	push
3	**stm call ret ret #**	$\Phi(3) = (\{\mathbf{stm}, \bigcirc^u_H \mathbf{call}\}, \emptyset)$	$[\mathbf{stm}, \Phi^g(2)][\mathbf{call}, \Phi^g(1)]\bot$	$\mathbf{stm} \gtrdot \mathbf{stm}$	pop
4	**stm call ret ret #**	$\Phi'(3) = \Phi^g(3) = (\{\mathbf{stm}, \bigcirc^u_H \mathbf{call}\}, \{\zeta_R\})$	$[\mathbf{call}, \Phi^g(1)]\bot$	$\mathbf{call} \lessdot \mathbf{stm}$	push
5	**call ret ret #**	$\Phi(4) = (\mathbf{call}, \emptyset)$	$[\mathbf{stm}, \Phi^g(3)][\mathbf{call}, \Phi^g(1)]\bot$	$\mathbf{stm} \gtrdot \mathbf{call}$	pop
6	**call ret ret #**	$\Phi'(4) = \Phi^g(4) = (\mathbf{call}, \{\zeta_R, \bigcirc^u_H \mathbf{call}\})$	$[\mathbf{call}, \Phi^g(1)]\bot$	$\mathbf{call} \lessdot \mathbf{call}$	push
7	**ret ret #**	$\Phi^g(5) = (\{\mathbf{ret}\}, \emptyset)$	$[\mathbf{call}, \Phi^g(4)][\mathbf{call}, \Phi^g(1)]\bot$	$\mathbf{call} \doteq \mathbf{ret}$	shift
8	**ret #**	$\Phi(6)$	$[\mathbf{ret}, \Phi^g(4)][\mathbf{call}, \Phi^g(1)]\bot$	$\mathbf{ret} \gtrdot \mathbf{ret}$	pop
9	**ret #**	$\Phi'(6) = \Phi^g(6) = (\{\mathbf{ret}\}, \{\zeta_R\})$	$[\mathbf{call}, \Phi^g(1)]\bot$	$\mathbf{call} \doteq \mathbf{ret}$	shift
10	#	$\Phi(7) = (\{\#\}, \emptyset)$	$[\mathbf{ret}, \Phi^g(1)]\bot$	$\mathbf{ret} \gtrdot \#$	pop
11	#	$\Phi'(7) = (\{\#\}, \zeta_R)$	$\bot$	–	–



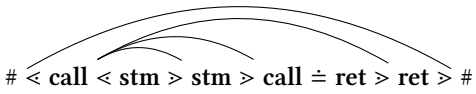


Fig. 11. Example run of the automaton for $\bigcirc^d \bigcirc^u \bigcirc^u_H \mathbf{call}$ (top left), ST (top right) and flat representation (bottom) of the input word.

If $\ominus^d_H \psi \in \mathrm{Cl}(\varphi)$ for some ψ, $\mathscr{DR}(\ominus^d_H \psi)$ contains the following rules:
for any $(\Phi, \Theta, \Psi) \in \delta_{pop}$,

(43) if $\zeta_L, \zeta_{\doteq} \notin \Psi_p$, then $(\ominus^d \psi \vee \chi^{\lessdot}_P \psi) \in \Theta_c$ iff $\ominus^d_H \psi \in \Phi_p$;
(44) if $\zeta_L \notin \Psi_p$, then $\ominus^d_H \psi \in \Psi_p$ iff $(\ominus^d(\ominus^d_H \psi) \vee \chi^{\lessdot}_P(\ominus^d_H \psi)) \in \Theta_c$;
(45) if $\ominus^d_H \psi \in \Phi_p$, then $\zeta_L, \zeta_{\doteq} \notin \Psi_p$;

for any $(\Phi, a, \Psi) \in \delta_{push/shift}$,

(46) if $\ominus^d_H \psi \in \Phi_c$, then $\zeta_L \in \Psi_p$;
(47) $\ominus^d_H \psi \notin \Phi_p$.

We illustrate how the construction of the $\bigcirc^u_H$ works through the example of Fig. 11, which shows an accepting run of the automaton built for formula $\bigcirc^d \bigcirc^u \bigcirc^u_H \mathbf{call}$. The computation goes on normally until, in step 3, the second **stm** is reached. Thanks to the two nested next operators, $\bigcirc^u_H \mathbf{call}$ is forced to hold here, so $\bigcirc^u_H \mathbf{call} \in \Phi_c(3)$. A pop transition then brings the automaton to $\Phi'(3)$: none of rules (32) and (33) apply but, since $\zeta_R \in \Phi'_p(3)$, rule (30) is satisfied and a push move reads **stm**. The next input symbol is a **call**, so a pop move is triggered. Since the popped symbol contains $\Phi'(3)$ and $\zeta_R \in \Phi'_p(3)$, rule (32) applies. The next state is thus $\Phi'(4)$ with $\bigcirc^u_H \mathbf{call} \in \Phi'_p(4)$, because $\bigcirc^u_H \mathbf{call} \in \Phi'_c(3)$. State $\Phi'(4)$ satisfies rule (31) because $\mathbf{call} \in \Phi'_c(4)$ and, due to the previous pop move, $\zeta_R \in \Phi'_p(4)$. A push move can therefore read **call**, and the computation goes on normally until acceptance.

4.1.6 Hierarchical Until and Since Operators. The construction for this kind of until and since operators also relies on $\mathscr{AC}$ constraints based on expansion laws (cf. Section 3.1). Here we only report the constraints for $\mathcal{U}^u_H$, as the others are symmetric.

(i) For any $\Phi \in Q$, we have $\psi \,\mathcal{U}^u_H\, \theta \in \Phi_c$ iff either:
- $\theta, \chi^{\lessdot}_P \top \in \Phi_c$ or
- $\psi, \bigcirc^u_H(\psi \,\mathcal{U}^u_H\, \theta) \in \Phi_c$.

4.2 Correctness Proof

We prove that $\mathcal{A}_\varphi$ accepts all and only words in which φ holds in position 1. The strategy we follow is to prove by induction on the syntactic structure of φ the claim that the computation of $\mathcal{A}_\varphi$ in each accepted word w is

such that, for all positions i, we have $(w, i) \models \psi$ iff $\psi \in \Phi_c^g(i)$ for all $\psi \in \mathrm{Cl}(\varphi)$. Since φ is included in all of its initial states, $\mathcal{A}_\varphi$ only accepts words read by a run where $\varphi \in \Phi_c^g(1)$, and consequently such that $(w, 1) \models \varphi$.

The overall induction argument is given in Theorem 4.4; before, the inductive step is proved separately for each operator. In each of the following lemmas, given a sub-formula θ of φ, we start from the *inductive assumption* that for each sub-formula ψ of θ except θ itself we have $(w, i) \models \psi$ iff $\psi \in \Phi_c^g(i)$ for all positions i in $\mathcal{A}_\varphi$'s accepting computations. Then, we prove the *inductive claim* stating that for a computation of $\mathcal{A}_\varphi$ to be accepting, the same must hold for θ, i.e., $(w, i) \models \theta$ iff $\theta \in \Phi_c^g(i)$ for all i. We also prove that each operator does not interfere with rules regarding other formulas, so that words that satisfy other formulas are accepted if they also satisfy the one at hand. To do this, we consider $\mathcal{A}_{\varphi-\theta}$, an OPA built as $\mathcal{A}_\varphi$ but using $\mathscr{DR} \setminus \mathscr{DR}(\theta)$ for δ, i.e., without rules related to θ. We show that if a computation is accepting for $\mathcal{A}_{\varphi-\theta}$ and satisfies the rules for θ, then it is accepting for $\mathcal{A}_\varphi$ too.

Before going on, we clarify that the set of sub-formulas $\mathrm{subf}(\varphi)$ of a formula φ is the smallest set such that:

- $\varphi \in \mathrm{subf}(\varphi)$;
- if any of the unary operators (e.g., $\neg$, $\bigcirc^d$, χ_F^d, χ_P^d, …) is in $\mathrm{subf}(\varphi)$, and ψ is its operand, then $\psi \in \mathrm{subf}(\varphi)$;
- if any of the binary operators (e.g., $\wedge$, $\vee$, $\mathcal{U}_\chi^d$, $\mathcal{S}_\chi^d$, …) is in $\mathrm{subf}(\varphi)$, and ψ and θ are its operands, then $\psi, \theta \in \mathrm{subf}(\varphi)$.

The set of *strict* sub-formulas of φ is $\mathrm{ssubf}(\varphi) = \mathrm{subf}(\varphi) \setminus \{\varphi\}$.

4.2.1 Lemmas about Next Operators. Lemma 4.1 proves the correctness of the rules given in Section 4.1.1 for the next operator, the proof for back being symmetric. We examine in detail the behavior of $\mathcal{A}_\varphi$ when reading input symbols; since between two push or shift moves an unbounded number of pop moves may occur, we consider the scanning of two consecutive characters.

Lemma 4.1. *Given a finite set of atomic propositions AP, an OP alphabet $(\mathcal{P}(AP), M_{AP})$ and a formula $\bigcirc^t \psi$ with $t \in \{d, u\}$, let $\mathcal{A}_\varphi$ be the OPA built for a formula φ such that $\bigcirc^t \psi \in \mathrm{Cl}(\varphi)$; and let $\mathcal{A}_{\varphi-\bigcirc^t \psi}$ be the OPA built as $\mathcal{A}_\varphi$ but using $\mathscr{DR} \setminus \mathscr{DR}(\bigcirc^t \psi)$ for δ.*
Inductive assumption: *in all accepting computations of $\mathcal{A}_\varphi$ for each position i in the input word w and for each sub-formula $\psi' \in \mathrm{ssubf}(\bigcirc^t \psi)$ we have $(w, i) \models \psi'$ iff $\psi' \in \Phi_c^g(i)$.*
***Inductive claim** I:* *$[I_1]$ A computation ρ of $\mathcal{A}_\varphi$ is accepting if and only if $[I_2]$ ρ is accepting for $\mathcal{A}_{\varphi-\bigcirc^t \psi}$ and for each position i in the input word w we have $(w, i) \models \bigcirc^t \psi$ iff $\bigcirc^t \psi \in \Phi_c^g(i)$.*

Proof. To prove the inductive claim, we first prove an **auxiliary claim** based on the following assertions:

- let $[A_1]$ be: $(w, i) \models \bigcirc^t \psi$;
- let $[A_2]$ be: all accepting computations of $\mathcal{A}_\varphi$ bring it from configuration $\langle a_i y, \Phi^g(i), \gamma\rangle$ with $\bigcirc^t \psi \in \Phi_c^g(i)$ to a new configuration $\langle y, \Phi^g(i+1), \gamma'\rangle$.

We prove that for any word $w = \#xa_i y\#$ with $a_i \in \mathcal{P}(AP)$ and position $i = |x| + 1$ in w, $A_1 \iff A_2$.

$[A_1 \Rightarrow A_2]$ $\mathcal{A}_\varphi$ makes the initial guess that $\bigcirc^t \psi$ holds in i, so when it reaches configuration $\langle a_i y, \Phi^g(i), \gamma\rangle$, we have $\bigcirc^t \psi \in \Phi_c^g(i)$. Later we show that a computation cannot be accepting without this guess.

Then, a transition reads symbol a in position i and $\mathcal{A}_\varphi$ reaches a configuration $\langle y, \Phi(i+1), [a_i, \Phi^g(j)] \dots \perp\rangle$, with $j = i$ if the transition was a push, and $j < i$ if it was a shift. In doing so, by rule (1) it guesses the first character of y, which we call a_{i+1}, so $\Phi_c(i+1) \cap AP = a_{i+1}$. This guess is possible because, if $\bigcirc^t \psi$ holds in i, the PR between a_i and a_{i+1} is the right one according to rule (3). Since $\bigcirc^t \psi$ holds in i, ψ, which is trivially a sub-formula of $\bigcirc^t \psi$, holds in $i+1$ and therefore is in $\Phi_c(i+1)$ by the inductive assumption. This satisfies rule (3). Also, note that according to the same rule there is no transition that goes from i with $\bigcirc^t \psi \notin \Phi_c^g(i)$ to $i+1$ with $\psi \in \Phi_c(i+1)$, so runs that do not make the initial guess cannot be accepting.

Now, let $t = d$: then we have either $a_i \lessdot a_{i+1}$ or $a_i \doteq a_{i+1}$. The automaton is now ready to read a_{i+1} with, respectively, a push or a shift move, because a_i is on top of the stack.

Suppose, instead, $t = u$: then either $a_i \doteq a_{i+1}$ or $a_i \gtrdot a_{i+1}$. In the former case, a_{i+1} can be read directly by a shift move. In the latter, the topmost stack symbol is popped, and a sequence of pop transitions brings the automaton to configuration $\langle y, \Phi^g(i+1), \gamma' \rangle$ where the topmost symbol in γ' is $[a_k, \Phi^g(j)]$ such that $a_k \lessdot a_{i+1}$ or $a_k \doteq a_{i+1}$, for some positions $j \leq k < i$. However, $\mathcal{DR}$ rule (2) imposes that the current part of the automaton's state does not change during pop moves. Thus, $\Phi^g_c(i+1) = \Phi_c(i+1)$, including $\psi \in \Phi^g_c(i+1)$. Then, the automaton is ready to proceed with a push or shift transition reading a_{i+1}.

$[A_2 \Rightarrow A_1]$ Suppose that an accepting computation contains configuration $\langle a_i y, \Phi^g(i), \gamma \rangle$ with $\bigcirc^t \psi \in \Phi^g_c(i)$. After reading i, it reaches a configuration $\langle y, \Phi(i+1), [a_i, \Phi(j)] \dots \bot \rangle$, with $j = i$ or $j < i$ depending on the move that read i. By rule (3), we must have $\psi \in \Phi_c(i+1)$, and the PR between a and $\Phi_c(i+1) \cap AP$ must be the right one according to t. Atom $\Phi_c(i+1)$ contains a guess of a_{i+1}, and rule (2) enforces it even if pop moves occur before a_{i+1} is read. Thus, since $\psi \in \mathrm{Cl}(\psi)$, by hypothesis $\psi \in \Phi^g_c(i+1)$ implies $(w, i+1) \models \psi$ and, thus, $\bigcirc^t \psi$ holds in i.

We can now prove the inductive claim. The $[I_1 \Rightarrow I_2]$ part follows from the auxiliary claim, together with the fact that $\bigcirc^t \psi$ is a future operator, so $\Phi(i)$ cannot be final if $\bigcirc^t \psi \in \Phi_c(i)$, and i is followed by another position in all accepting computations (i.e., $|y| \geq 1$). Moreover, $\mathcal{A}_\varphi$'s transition relation is a subset of that of $\mathcal{A}_{\varphi - \bigcirc^t \psi}$ because the latter has fewer rules than the former, so if a word is accepted by $\mathcal{A}_\varphi$, then $\mathcal{A}_{\varphi - \bigcirc^t \psi}$ must be able to perform the same accepting run.

For the $[I_2 \Rightarrow I_1]$ side, note that according to the proof above $\Phi(i+1)$ does not necessarily contain $\bigcirc^t \psi$ (unless $\bigcirc^t \psi$ holds in $i+1$) so $\mathcal{DR}$ rule (3) cannot prevent it from reaching an accepting configuration. Thus, if a computation is accepting for $\mathcal{A}_{\varphi - \bigcirc^t \psi}$ and satisfies $(w, i) \models \bigcirc^t \psi$ iff $\bigcirc^t \psi \in \Phi_c(i)$, it can go on past $i+1$ and be accepting. □

We now prove the correctness of rules given in Section 4.1.2 for the $\chi^{\doteq}_F$ operator in Lemma 4.2. The lemmas for the $\chi^{\lessdot}_F$ and chain back operators follow a very similar structure, so we postpone them to Appendix A.

In the following, we denote as $\mathrm{first}(w)$ the first position of a word w; we use initial letters of the alphabet $a, b, c, \dots$ to denote single input symbols, and u, v to denote sub-words. We use Fig. 12, which represents the generic structure of any one-to-many composed chain. In the left tree, the contexts of the outermost chain (a and d) are in the $\doteq$ PR, and in the right one they are in the $\gtrdot$ PR (cf. Property 4 of the χ relation). We use the left tree when proving case $\chi^{\doteq}_F$; the $\chi^{\gtrdot}_F$ case can be proved identically to $\chi^{\doteq}_F$ by referring to the right tree and is therefore omitted. Both sides, instead, are used for $\chi^{\lessdot}_F$, but note that the $\lessdot$ PR does not hold between a and d, but rather between a and all b_i's. When referring to Fig. 12, we denote by i_c the word position of any input symbol c, and by i_v the position of $\mathrm{first}(v)$ for any sub-word v.

Lemma 4.2 ($\chi^{\doteq}_F$ operator). *Given a finite set of atomic propositions AP, an OP alphabet $(\mathcal{P}(AP), M_{AP})$ and a formula $\chi^{\doteq}_F \psi$, let $\mathcal{A}_\varphi$ be the OPA built for a formula φ such that $\chi^{\doteq}_F \psi \in \mathrm{Cl}(\varphi)$; and let $\mathcal{A}_{\varphi - \chi^{\doteq}_F \psi}$ be the OPA built as $\mathcal{A}_\varphi$ but using $\mathcal{DR} \setminus \mathcal{DR}(\chi^{\doteq}_F \psi)$ for δ.*
Inductive assumption: *in all accepting computations of $\mathcal{A}_\varphi$ for each position i in the input word w and for each sub-formula $\psi' \in \mathrm{ssubf}(\chi^{\doteq}_F \psi)$ we have $(w, i) \models \psi'$ iff $\psi' \in \Phi^g_c(i)$.*
***Inductive claim** I:* $[I_1]$ *A computation ρ of $\mathcal{A}_\varphi$ is accepting if and only if $[I_2]$ ρ is accepting for $\mathcal{A}_{\varphi - \chi^{\doteq}_F \psi}$ and for each position i in the input word w we have $(w, i) \models \chi^{\doteq}_F \psi$ iff $\chi^{\doteq}_F \psi \in \Phi^g_c(i)$.*

Proof. We first prove two **auxiliary claims**, built on the following assertions:

- Let $[A_1]$ be: $(w, i) \models \chi^{\doteq}_F \psi$;
- let $[A_2]$ be: all accepting computations of $\mathcal{A}_\varphi$ bring it from a configuration $\langle yz, \Phi^g(i), \alpha\gamma \rangle$ with $\chi^{\doteq}_F \psi \in \Phi^g_c(i)$ to a configuration $\langle z, \Phi^g(i_z), \alpha'\gamma \rangle$ such that $\chi^{\doteq}_F \psi \notin \Phi^g_p(i_z)$, $|\alpha| = 1$ and $|\alpha'| = 1$ if $\mathrm{first}(y)$ is read by a shift move, $|\alpha'| = 2$ if it is read by a push move.

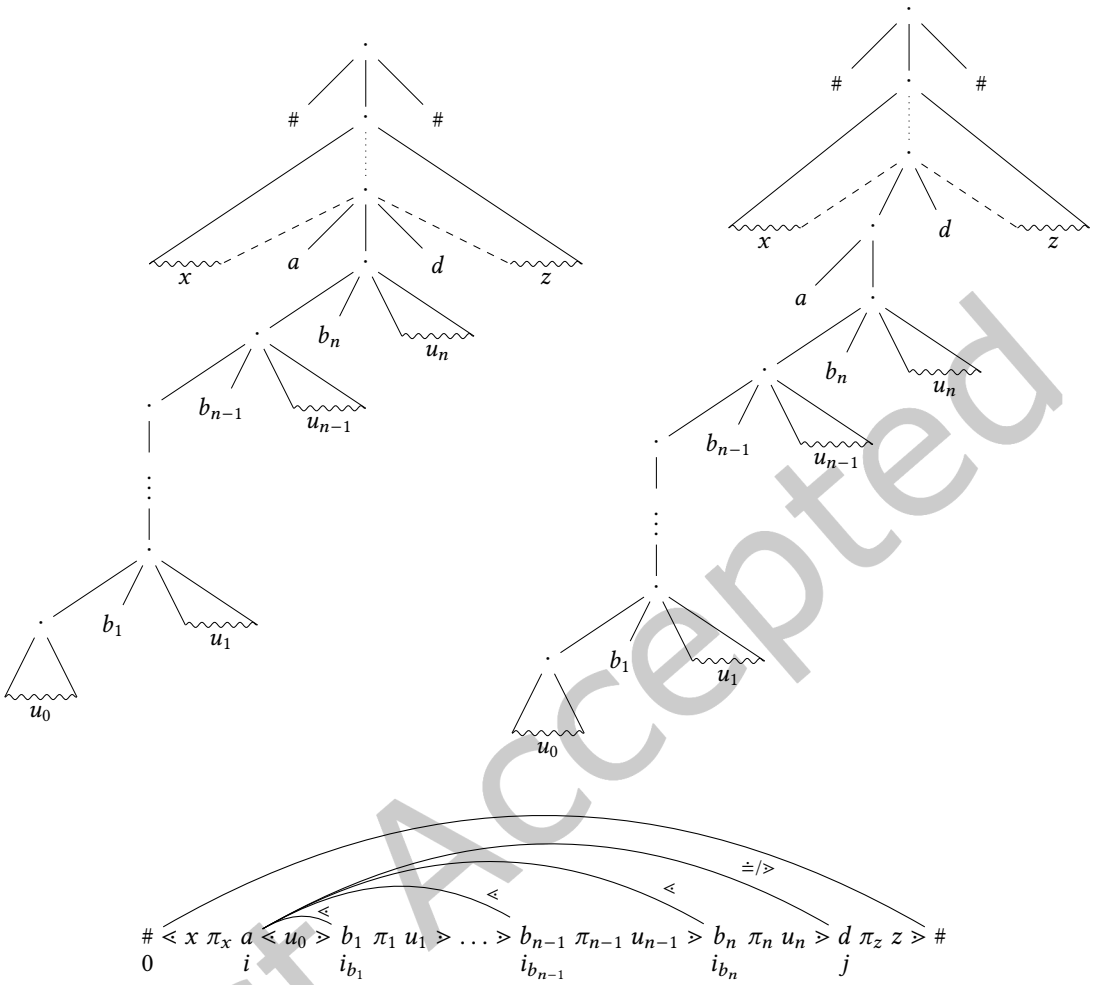


Fig. 12. The two possible STs of a generic OP word $w = xyz$ (top), and its flat representation with chains (bottom). Wavy lines are placeholders for frontiers of subtrees or parts thereof. We have either $a \doteq d$ (top left), or $a \gtrdot d$ (top right). In both trees, $a \lessdot b_k$ for $1 \leq k \leq n$, and the corresponding word positions are in the chain relation. For $1 \leq k \leq n$, u_k is the word generated by the right part of the rhs whose first terminal is b_k. So, either ${}^{b_k}[u_k]^{b_{k+1}}$, or u_k is of the form $v_0^k c_0^k v_1^k c_1^k \ldots c_{m_k}^k v_{m_k+1}^k$, where $c_p^k \doteq c_{p+1}^k$ for $0 \leq p < m_k$, $b_k \doteq c_0^k$, and resp. $c_{m_k}^k \gtrdot b_{k+1}$ and $c_{m_n}^n \gtrdot d$ (cf. Fig. 13). Moreover, for each $0 \leq p < m_k$, either $v_{p+1}^k = \varepsilon$ or ${}^{c_p^k}[v_{p+1}^k]^{c_{p+1}^k}$; either $v_0^k = \varepsilon$ or ${}^{b_k}[v_0^k]^{c_0^k}$, and either $v_{m_k+1}^k = \varepsilon$ or ${}^{c_{m_k}^k}[v_{m_k+1}^k]^{b_{k+1}}$ (resp. ${}^{c_{m_n}^n}[v_{m_n+1}^n]^d$). u_0 has this latter form, except $v_0^0 = \varepsilon$ and $a \lessdot c_0^0$. In the bottom representation, the π_ks are placeholders for precedence relations, that depend on the surrounding characters. Also, chains that may or may not exist depending on the form of each u_k are not shown by edges (e.g., between b_n and d).

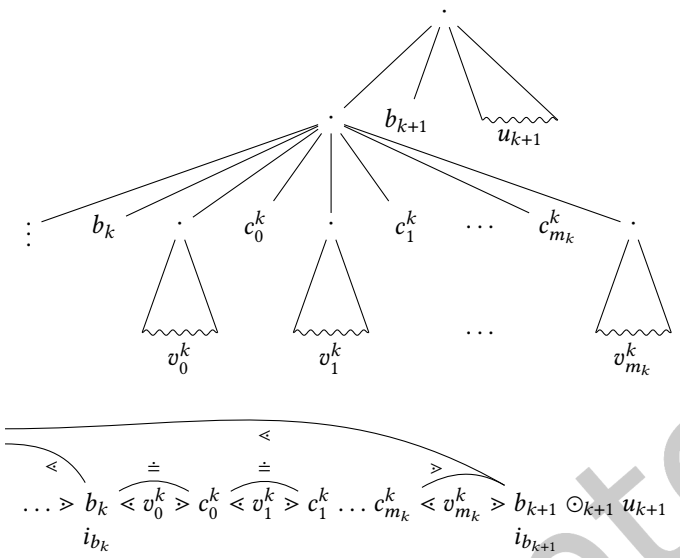


Fig. 13. The structure of u_k in the word of Fig. 12.

We prove that for any word $w = \#xyz\#$ and positions $i = |x| + 1$, $i_z = |xy| + 1$ in w, $A_1 \iff A_2$.

$[A_1 \implies A_2]$ Suppose $\chi_F^{\doteq}\,\psi$ holds in position i, labeled with terminal symbol a. Then, i is the left context of a chain with right context d and $a \doteq d$, so w has the form of Fig. 12 (left), possibly with $n = 0$ (cf. the caption for notation). In all accepting computations, the OPA reaches configuration $\langle a \ldots z, \Phi^g(i), [f, \Phi^g(k)]\gamma\rangle$, where $k < i$ and $\alpha = [f, \Phi^g(k)]$, and guesses that $\chi_F^{\doteq}\,\psi$ holds in i, so $\chi_F^{\doteq}\,\psi \in \Phi_c^g(i)$. We show later in the proof that all accepting computations must make this guess. a is read by either a push or a shift transition, leading the OPA to configuration $\langle c_0^0 \ldots z, \Phi(i_{c_0^0}), \beta\rangle$, with either $\beta = [a, \Phi^g(i)][f, \Phi^g(k)]\gamma$ or $\beta = [a, \Phi^g(k)]\gamma$, respectively. Moreover, $\chi_F^{\doteq}\,\psi \in \Phi_p(i_{c_0^0})$ and $\zeta_L \in \Phi_p(i_{c_0^0})$ due to rule (6). Since $\chi_F^{\doteq}\,\psi$ holds in i, a is the left context of a chain, so the next transition is a push, satisfying rule (5), and $\Phi(i_{c_0^0}) = \Phi^g(i_{c_0^0})$. Any accepting computation must go through the support for this chain. The next configuration is $\langle v_1^0 \ldots z, \Phi(i_{v_1^0}), [c_0^0, \Phi^g(i_{c_0^0})]\beta\rangle$, with $\chi_F^{\doteq}\,\psi \in \Phi_p^g(i_{c_0^0})$. Then, the computation goes on normally. Note that, when reading an inner chain body such as v_1^0, the automaton does not touch the stack symbol containing $\Phi^g(i_{c_0^0})$, and other symbols in the body of the same simple chain, i.e. $c_1^0, c_2^0 \ldots$, are read with shift moves that update the topmost stack symbol with the new terminal, leaving state $\Phi^g(i_{c_0^0})$ untouched.

If a is the left context of more than one chain (i.e. $n > 0$ in the figure), the OPA then reaches configuration $\langle b_1 \ldots z, \Phi(i_{b_1}), [c_{m_0}^0, \Phi^g(i_{c_0^0})]\beta\rangle$. Since $c_{m_0}^0 \gtrdot b_1$, the next transition is a pop. We have $\chi_F^{\doteq}\,\psi \in \Phi_p^g(i_{c_0^0})$, so by rule (7), the automaton reaches configuration $\langle b_1 \ldots z, \Phi'(i_{b_1}), \beta\rangle$ with $\chi_F^{\doteq}\,\psi \in \Phi'_p(i_{b_1})$. Then, since a is contained in the topmost stack symbol and $a \lessdot b_1$, the next move is a push, leading to $\langle v_0^1 \ldots z, \Phi(i_{v_0^1}), [b_1, \Phi'_p(i_{b_1})]\beta\rangle$. Notice how $\chi_F^{\doteq}\,\psi$ is again stored as a pending obligation in the topmost stack symbol. The OPA run goes on in the same way for each terminal b_p, $1 \le p \le n$, until the automaton reaches configuration $\langle d \ldots z, \Phi(j), [c_{m_n}^n, \Phi^g(i_{b_n})]\beta\rangle$ with $\chi_F^{\doteq}\,\psi \in \Phi_p^g(i_{b_n})$. If a is the left context of only one chain, this is the configuration reached after reading the body of such a chain, with $n = 0$. Since $c_{m_n}^n \gtrdot d$, a pop transition leads to $\langle d \ldots z, \Phi'(j), \beta\rangle$, with $\chi_F^{\doteq}\,\psi \in \Phi'_p(j)$, by rule (7) (recall j is the position of d). Note that there exists a computation in which $\chi_F^{\doteq}\,\psi \notin \Phi_p(j)$, because no

other rule prevents it, so rule (7) applies. Then, if $\chi_F^{\doteq}\,\psi$ holds in i, since a is the terminal in the topmost stack symbol, we must have $a \doteq d$. So d is read by a shift move, leading to $\langle z, \Phi(i_z), \alpha'\gamma\rangle$ with $\alpha' = [d, \Phi^g(i)][f, \Phi^g(k)]$ or $\alpha' = [d, \Phi^g(k)]$, depending on which kind of move previously read a. Since $\chi_F^{\doteq}\,\psi$ holds in i, ψ holds in j, and $\psi \in \Phi_c'^d$, because we assume the correctness of the construction for all other operators. This satisfies rule (8), and verifies the initial guess that $\chi_F^{\doteq}\,\psi$ holds in i. By rule (8), any computation in which ψ holds in j must have $\chi_F^{\doteq}\,\psi \in \Phi_p'(j)$, which is only the case if the OPA makes such initial guess. Finally, there exists a computation in which $\chi_F^{\doteq}\,\psi \notin \Phi_p(i_z)$, satisfying A_2. Note that all computations of this form may then proceed normally until acceptance, if they are not blocked by rules other than 6-8.

$[A_2 \Rightarrow A_1]$ Suppose that an accepting computation reaches configuration $\langle a \dots z, \Phi^g(i), [f, \Phi^g(k)]\gamma\rangle$, with $k < i$, $\chi_F^{\doteq}\,\psi \in \Phi_c^g(i)$, $\alpha = [f, \Phi^g(k)]$, and $f \lessdot a$ (the case $f \doteq a$ is analogous). a is read by a push move in this case, which leads the OPA to configuration $\langle c_0^0 \dots z, \Phi(i_{c_0^0}), [a, \Phi^g(i)][f, \Phi^g(k)]\gamma\rangle$, with $\chi_F^{\doteq}\,\psi, \zeta_L \in \Phi_p(i_{c_0^0})$. Since $\zeta_L \in \Phi_p(i_{c_0^0})$, the next transition must be a push, so $a \lessdot c_0^0$, a is the left context of a chain and w has one of the structures of Fig. 12. The push move brings the OPA to configuration $\langle v_0^0 \dots z, \Phi(i_{v_0^0}), [c_0^0, \Phi(i_{c_0^0})][a, \Phi^g(i)][f, \Phi^g(k)]\gamma\rangle$. Notice that the stack size is now $|\gamma| + 3$. To fulfill A_2, the automaton must eventually reach a configuration in which the stack size is $|\gamma| + 2$. This can be achieved if $[c_0^0, \Phi(i_{c_0^0})]$ is popped, so $\alpha' = [a, \Phi^g(i)][f, \Phi^g(k)]$. In a generic word such as the one of Fig. 12, this happens only before reading b_p, $1 \le i \le n$, or d.

In both cases, let $[c_{m_k}^k, \Phi^g(i_{b_k})]$ be the popped stack symbol. We have $\chi_F^{\doteq}\,\psi \in \Phi_p^g(i_{b_k})$. Let Φ' be the destination state of the pop move: by rule (7), $\chi_F^{\doteq}\,\psi \in \Phi_p'$, so Φ' is not $\Phi^g(i_z)$ from claim A_2. If the next move is a push (such as when reading any b_p, $1 \le p \le n$), the stack length increases again, which also does not satisfy the thesis. If the next move is a pop, rule (7) blocks the computation. So, the next move must be a shift, updating symbol $[a, \Phi^g(i)]$ to $[d, \Phi^g(i)]$, where d is the just-read terminal symbol. This means the OPA reached the right context of the chain whose left context is i (i.e. a), and the two positions are in the $\doteq$ relation. By rule (8), ψ is part of the starting state of this move, so ψ holds in this position, satisfying $\chi_F^{\doteq}\,\psi$ in i.

$[I_1 \Rightarrow I_2]$ follows directly from $A_1 \Rightarrow A_2$ and $\mathcal{A}_{\varphi - \chi_F^{\doteq}\psi}$'s $\mathscr{DR}$ rules being a strict subset of $\mathcal{A}_\varphi$'s. $[I_2 \Rightarrow I_1]$ again follows from $A_2 \Rightarrow A_1$, and the fact that $\Phi^g(i_z)$ may not contain $\chi_F^{\doteq}\,\psi$, nor states in α', so rules (6)-(8) may not prevent the computation from reaching a final state. □

We now prove the correctness of the construction for the $\bigcirc_H^u$ operator; we omit the proofs for the other hierarchical next and back operators as they are very similar.

Lemma 4.3 ($\bigcirc_H^u$ operator). *Given a finite set of atomic propositions AP, an OP alphabet $(\mathcal{P}(AP), M_{AP})$ and a formula $\bigcirc_H^u\,\psi$, let $\mathcal{A}_\varphi$ be the OPA built for a formula φ such that $\bigcirc_H^u\,\psi \in \mathrm{Cl}(\varphi)$; and let $\mathcal{A}_{\varphi - \bigcirc_H^u\psi}$ be the OPA built as $\mathcal{A}_\varphi$ but using $\mathscr{DR} \setminus \mathscr{DR}(\bigcirc_H^u\,\psi)$ for δ.*
***Inductive assumption:** in all accepting computations of $\mathcal{A}_\varphi$ for each position i in the input word w and for each sub-formula $\psi' \in \mathrm{ssubf}(\bigcirc_H^u\,\psi)$ we have $(w, i) \models \psi'$ iff $\psi' \in \Phi_c^g(i)$.*
***Inductive claim** I: $[I_1]$ A computation ρ of $\mathcal{A}_\varphi$ is accepting if and only if $[I_2]$ ρ is accepting for $\mathcal{A}_{\varphi - \bigcirc_H^u\psi}$ and for each position i in the input word w we have $(w, i) \models \bigcirc_H^u\,\psi$ iff $\bigcirc_H^u\,\psi \in \Phi_c^g(i)$.*

Proof. We first prove two **auxiliary claims**, built on the following assertions:

- Let $[A_1]$ be: $(w, i') \models \bigcirc_H^u\,\psi$;
- let $[A_2]$ be: all accepting computations of $\mathcal{A}_\varphi$ bring it from a configuration $\langle yz, \Phi^g(i'), \gamma\rangle$ with $\bigcirc_H^u\,\psi \in \Phi_c^g(i')$ to a configuration $\langle z, \Phi^g(i_z), \alpha\gamma\rangle$ such that $\bigcirc_H^u\,\psi \notin \Phi_p^g(i_z)$ and $|\alpha| = 1$.

We prove that for any word $w = \#xyz\#$ and positions $i = |x| + 1$, $i_z = |xy| + 1$ in w, $A_1 \iff A_2$.

$[A_1 \implies A_2]$ Suppose $\bigcirc_H^u\,\psi$ holds in position i'. Then, by the semantics of $\bigcirc_H^u$ there exists a position i such that $\chi(i, i')$ and $i \lessdot i'$. By property 4 of the χ relation, w has the form of Fig. 12 with $n \ge 2$, and $i' = i_{b_k}$ for some $1 \le k \le n - 1$. Also, ψ holds in $i_{b_{k+1}}$. In all accepting computations, the OPA reaches configuration $\langle b_k \dots z,$

$\Phi^g(i_{b_k}), [a, \Phi^g(k)]\gamma'\rangle$ where $k \leq i$ and $\gamma = [a, \Phi^g(k)]\gamma'$, and guesses that $\bigcirc^u_H \psi$ holds in i_{b_k}, so $\bigcirc^u_H \psi \in \Phi^g_c(i_{b_k})$. Previously, if $k \geq 2$ the OPA must have read b_{k-1} and u_{k-1}, or u_0 if $k = 1$. Both $b_{k-1}u_{k-1}$ and u_0 end with a position in the $\gtrdot$ PR with b_k, which triggers a pop move, and therefore $\zeta_R \in \Phi^g_p(i_{b_k})$, so rule (30) is satisfied. The OPA then reads b_k with a push move, leading to configuration $\langle v_0^k \dots z, \Phi(i_{v_0^k}), [b_k, \Phi^g(i_{b_k})]\gamma\rangle$.

The computation then goes on normally by reading the rest of u_k, until reaching configuration $\langle b_{k+1} \dots z, \Phi(i_{b_{k+1}}), [c^k_{m_k}, \Phi^g(i_{b_k})]\gamma\rangle$, where $\bigcirc^u_H \psi \notin \Phi_p(i_{b_{k+1}})$ (otherwise the computation would not be accepting by rule (33)). Since $c^k_{m_k} \gtrdot b_{k+1}$, a pop move is triggered. We have $\zeta_R \in \Phi^g_p(i_{b_k})$, so by rule (32) we reach configuration $\langle b_{k+1} \dots z, \Phi'(i_{b_{k+1}}), \gamma\rangle$, where $\bigcirc^u_H \psi \in \Phi'_p(i_{b_{k+1}})$. Since this move is a pop, we have $\zeta_R \in \Phi'(i_{b_{k+1}})$. Because $b_{k+1} \lessdot a$, the next move is a push that reads b_{k+1}, leading to $\langle v_0^{k+1} \dots z, \Phi(i_{v_0^{k+1}}), [b_{k+1}, \Phi^g(i_{b_{k+1}})]\gamma\rangle$, where $\Phi^g(i_{b_{k+1}}) = \Phi'(i_{b_{k+1}})$. By rule (31), since $\bigcirc^u_H \psi \in \Phi'_p(i_{b_{k+1}})$, we have $\psi \in \Phi_c(i_{v_0^{k+1}})$. If we set $z = \text{first}(v_0^{k+1})$ (or $z = c_0^{k+1}$, or $z = b_{k+2}$, if resp. $v_0^{k+1} = \varepsilon$ or $u_{k+1} = \varepsilon$) and $\alpha = [b_{k+1}, \Phi^g(i_{b_{k+1}})]$, then $A_1 \implies A_2$ is proven.

$[A_2 \implies A_1]$ Suppose that an accepting computation reaches a configuration $\langle yz, \Phi^g(i'), \gamma\rangle$ with $\bigcirc^u_H \psi \in \Phi^g_c(i')$. By rule (34), the next position cannot be read by a shift move, but only by a push. Thus, there exists a position i labeled with a such that $a \lessdot \text{first}(y)$. By rule (30), $\zeta_R \in \Phi^g_p(i')$, so we also have $\chi(i, i')$. Thus, by property 4 of the χ relation, w has the form of Fig. 12 with $n \geq 1$, and $i' = i_{b_k}$ for some $1 \leq k \leq n$. Such a push move reads b_k and stores a stack symbol containing $\Phi^g(i_{b_k})$. Since the computation is accepting, at some point this stack symbol must be popped. Let $\langle b \dots z, \Phi(i_b), [c^k_{m_k}, \Phi^g(i_{b_k})]\gamma\rangle$ be the configuration right before this pop move occurs (we name the look-ahead as b). Recall that $\zeta_R \in \Phi^g_p(i_{b_k})$ and $\bigcirc^u_H \psi \in \Phi^g_c(i_{b_k})$, so by rule (32) the next configuration is $\langle b \dots z, \Phi'(i_b), \gamma\rangle$ with $\bigcirc^u_H \psi \in \Phi'_p(i_b)$ (and also $\zeta_R \in \Phi'_p(i_b)$ because of the pop move). By rules (33) and (34), the next move has to be a push, so $\Phi^g(i_b) = \Phi'(i_b)$. Therefore, we have $a \lessdot b$ and $b = b_{k+1}$ (cf. Fig. 12, it cannot be $b = d$). Also, rule (31) applies, and we have $\psi \in \Phi^g(i_b)$. By the semantics of $\bigcirc^u_H \psi$, we can claim $(w, i') \models \bigcirc^u_H \psi$ (recall $i' = i_{b_k}$), which proves $A_2 \implies A_1$.

$[I_1 \Rightarrow I_2]$ follows directly from $A_1 \Rightarrow A_2$ and $\mathcal{A}_{\varphi - \bigcirc^u_H \psi}$'s $\mathscr{DR}$ rules being a strict subset of $\mathcal{A}_\varphi$'s. $[I_2 \Rightarrow I_1]$ again follows from $A_2 \Rightarrow A_1$, and the fact that $\Phi^g(i_z)$ may not contain $\bigcirc^u_H \psi$, nor states in α, so rules (30)-(34) may not prevent the computation from reaching a final state. □

4.2.2 *Wrap-Up.*

Theorem 4.4 (Correctness of Finite-Word Model Checking). *Given a finite set of atomic propositions AP, an OP alphabet $(\mathcal{P}(AP), M_{AP})$, a word w on it, and a POTL formula φ, automaton $\mathcal{A}_\varphi$ is such that it performs at least one accepting computation on w if and only if $(w, 1) \models \varphi$.*

Proof. We prove by structural induction on formula syntax the following statement: for each $\theta \in \text{subf}(\varphi)$, a computation of $\mathcal{A}_\varphi$ is accepting if and only if it is accepting for $\mathcal{A}_{\varphi-\theta}$ and for each position i in the input word w we have $(w, i) \models \theta$ iff $\theta \in \Phi^g_c(i)$.

From this, it directly follows that in all $\mathcal{A}_\varphi$'s accepting computations we have $(w, 1) \models \varphi$ iff $\varphi \in \Phi^g_c(1)$. Since $\Phi^g(1)$ is an initial state, we always have $\varphi \in \Phi^g_c(1)$, hence $\mathcal{A}_\varphi$ accepts only words such that $(w, 1) \models \varphi$.

The base case of the induction are members of AP. The only $\mathscr{DR}$ rule that applies to them directly is (1). Clearly, due to (1), all computations of $\mathcal{A}_\varphi$ that reach a final configuration must be such that for all $a \in AP$ we have $(w, i) \models a$ iff $a \in \Phi^g_c(i)$, or no input symbols could be read by push or shift moves. The other side of the implication is also trivial. As for negated atomic propositions note that, by $\mathscr{AC}$ constraint (a), $a \notin \Phi_c$ implies $\neg a \in \Phi_c$, so we also have $(w, i) \models \neg a$ iff $\neg a \in \Phi^g_c(i)$.

For the inductive hypothesis, we assume that in all accepting computations of $\mathcal{A}_\varphi$ for each position i in the input word w and formula $\psi \in \text{ssubf}(\theta) \setminus \{\theta\}$ we have $(w, i) \models \psi$ iff $\psi \in \Phi^g_c(i)$.

We proved the inductive step for all temporal operators in Lemmas 4.1, 4.2, 4.3, A.1, A.2 and A.3, while other proofs have been omitted due to their similarity with the previous ones.

For the $\wedge$ propositional operator, there are no $\mathscr{DR}$ rules involved, so $\mathcal{A}_{\varphi-(\psi\wedge\psi')} = \mathcal{A}_\varphi$. The fact that $(w,i) \models \psi \wedge \psi'$ iff $\psi \wedge \psi' \in \Phi_c^g(i)$ follows from $\mathscr{AC}$ constraint (b) and the inductive hypothesis. The proof for $\vee$ is analogous. As for $\neg$, note that, by $\mathscr{AC}$ constraint (a), $\theta \notin \Phi_c$ implies $\neg\theta \in \Phi_c$, so from $(w,i) \models \theta$ iff $\theta \in \Phi_c^g(i)$ in the hypothesis we derive $(w,i) \models \neg\theta$ iff $\neg\theta \in \Phi_c^g(i)$.

Until and since operators rely on $\mathscr{AC}$ constraints whose correctness derives from the expansion laws in Section 3.1. The inductive step for them follows from those of the next and back operators appearing in the right-hand-sides of the expansion laws.

This concludes our induction argument.

Now, we need to prove that if $(w,1) \models \varphi$, then $\mathcal{A}_\varphi$ has at least one accepting computation. This computation is such that for each $\theta \in \mathrm{Cl}(\varphi)$ and position i in the word w it reads, we have $(w,i) \models \theta$ iff $\theta \in \Phi_c^g(i)$. First, consider a version of $\mathcal{A}_\varphi$, called $\mathcal{A}'_\varphi$, built with an empty $\mathscr{DR}$, so that its transition relation is a complete graph. Clearly, $\mathcal{A}'_\varphi$ performs at least a computation with the above feature. Since we proved that for each $\theta \in \mathrm{Cl}(\varphi)$ the rules in $\mathscr{DR}(\theta)$ do not block it if it is such that $(w,i) \models \theta$ iff $\theta \in \Phi_c^g(i)$, we can conclude that this computation is accepting in $\mathcal{A}_\varphi$ too. □

4.3 Complexity

The set $\mathrm{Cl}(\varphi)$ is linear in $|\varphi|$, the length of φ. $\mathrm{Atoms}(\varphi)$ has size at most $2^{|\mathrm{Cl}(\varphi)|} = 2^{O(|\varphi|)}$, and the size of the set of states is the square of that. Therefore,

Theorem 4.5. *Given a POTL formula φ, it is possible to build an OPA $\mathcal{A}_\varphi$ accepting the language denoted by φ with at most $2^{O(|\varphi|)}$ states.*

$\mathcal{A}_\varphi$ can then be intersected with an OPA modeling a program, and emptiness can be decided with polynomial-time reachability algorithms that we will present in Section 6.

Since it is possible to linearly translate NWTL into POTL in a way similar to what we did with Operator Precedence Temporal Logic (OPTL) in [27], we can exploit the same lower bounds for decision problems:

Theorem 4.6. *POTL model checking and satisfiability on finite OP words are EXPTIME-complete.*

Therefore, POTL does not have a worse computational complexity than NWTL and OPTL, despite its greater expressive power.

5 ω-WORD MODEL CHECKING

To perform model checking of a POTL formula φ on OP ω-words, we adapt the approach used in [24] for OPTL. We build a generalized ωOPBA (cf. Definition 2.7) $\mathcal{A}^\omega_\varphi = (\mathcal{P}(AP), M_{AP}, Q_\omega, I_\omega, \mathbf{F}, \delta)$, where $Q_\omega = \mathrm{Atoms}(\varphi) \times \mathcal{P}(\mathrm{Cl}_{pend}(\varphi)) \times \mathcal{P}(\mathrm{Cl}_{st}(\varphi))$, and $\mathrm{Cl}_{st}(\varphi) = \{\chi_F^\pi\,\psi \in \mathrm{Cl}(\varphi) \mid \pi \in \{\lessdot, \doteq, \gtrdot\}\} \cup \{\bigcirc_H^t \in \mathrm{Cl}(\varphi) \mid t \in \{d,u\}\}$.

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 states in 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 add a subset of $\mathrm{Cl}_{st}(\varphi)$ to states obtained according to the OPA construction of Section 4.1. $\mathcal{A}^\omega_\varphi$'s 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 defined in Section 4.1 for Φ_c and Φ_p remain the same. At any point in a computation, Φ_s contains any element of $\mathrm{Cl}_{st}(\varphi)$ that is present in the pending part of any symbol currently in the stack. Thus, pending temporal obligations are copied from the stack to the ωOPBA state, so that they can be taken into account by the Büchi acceptance condition. Initial states are the same as in the finite case except their in-stack part is empty: $I_\omega = \{(\Phi_c, \Phi_p, \emptyset) \mid \Phi \in I\}$, where I is the initial set of $\mathcal{A}_\varphi$.

step	input	state	stack	PR	move
1	**call call han exc ret ret call** …	$\Phi^g(1) = (\{\textbf{call}, \chi^d_F\,\textbf{ret}, \chi^{\doteq}_F\,\textbf{ret}\}, \{\zeta_L\}, \emptyset)$	$\bot$	# ⋖ **call**	push
2	**call han exc ret ret call** …	$\Phi^g(2) = (\{\textbf{call}, \chi^d_F\,\textbf{ret}, \chi^{\doteq}_F\,\textbf{ret}\}, \{\chi^{\doteq}_F\,\textbf{ret}, \zeta_L\}, \emptyset)$	$[\textbf{call}, \Phi^g(1)]\bot$	**call** ⋖ **call**	push
3	**han exc ret ret call** …	$\Phi^g(3) = (\{\textbf{han}\}, \{\chi^{\doteq}_F\,\textbf{ret}, \zeta_L\}, \{\chi^{\doteq}_F\,\textbf{ret}\})$	$[\textbf{call}, \Phi^g(2)][\textbf{call}, \Phi^g(1)]\bot$	**call** ⋖ **han**	push
4	**exc ret ret call** …	$\Phi^g(4) = (\{\textbf{exc}\}, \emptyset, \{\chi^{\doteq}_F\,\textbf{ret}\})$	$[\textbf{han}, \Phi^g(3)][\textbf{call}, \Phi^g(2)][\textbf{call}, \Phi^g(1)]\bot$	**han** ≐ **exc**	shift
5	**ret ret call** …	$\Phi(5) = (\{\textbf{ret}\}, \emptyset, \{\chi^{\doteq}_F\,\textbf{ret}\})$	$[\textbf{exc}, \Phi^g(3)][\textbf{call}, \Phi^g(2)][\textbf{call}, \Phi^g(1)]\bot$	**exc** ⋗ **ret**	pop
6	**ret ret call** …	$\Phi'(5) = \Phi^g(5) = (\{\textbf{ret}\}, \{\chi^{\doteq}_F\,\textbf{ret}\}, \{\chi^{\doteq}_F\,\textbf{ret}\})$	$[\textbf{call}, \Phi^g(2)][\textbf{call}, \Phi^g(1)]\bot$	**call** ≐ **ret**	shift
7	**ret call** …	$\Phi(6) = (\{\textbf{ret}\}, \emptyset, \{\chi^{\doteq}_F\,\textbf{ret}\})$	$[\textbf{ret}, \Phi^g(2)][\textbf{call}, \Phi^g(1)]\bot$	**ret** ⋗ **ret**	pop
8	**ret call** …	$\Phi'(6) = \Phi^g(6) = (\{\textbf{ret}\}, \{\chi^{\doteq}_F\,\textbf{ret}\}, \emptyset)$	$[\textbf{call}, \Phi^g(1)]\bot$	**call** ≐ **ret**	shift
9	**call** …	$\Phi(7) = (\{\textbf{call}, \chi^d_F\,\textbf{ret}, \chi^{\doteq}_F\,\textbf{ret}\}, \emptyset, \emptyset)$	$[\textbf{ret}, \Phi^g(1)]\bot$	**ret** ⋗ **call**	pop
10	**call** …	$\Phi'(7) = \Phi^g(7) = \Phi^g(1)$	$\bot$	# ⋖ **call**	push

Fig. 14. Example run of the automaton for $\chi^d_F\,\textbf{ret}$ on word $(\textbf{call call han exc ret ret})^\omega$.

Suppose we want to verify $\chi^{\doteq}_F\,\psi$. 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 the satisfaction of its temporal requirement. Then, we can define an acceptance set $\bar{F}_{\chi^{\doteq}_F\,\psi} \in \mathbf{F}$ as the set of states *not* containing $\chi^{\doteq}_F\,\psi$ in their pending or in-stack parts. The same holds for $\chi^{\gtrdot}_F$. Formally, $\bar{F}_{\chi^{\pi}_F\,\psi} = \{\Phi \in Q_\omega \mid \chi^{\pi}_F\,\psi \notin \Phi_p \cup \Phi_s\}$, for $\pi \in \{\doteq, \gtrdot\}$. Things are slightly more complicated with $\chi^{\lessdot}_F$, as we have $\bar{F}_{\chi^{\lessdot}_F\,\psi} = \{\Phi \in Q_\omega \mid \chi^{\lessdot}_F\,\psi \notin \Phi_s \wedge (\chi^{\lessdot}_F\,\psi \notin \Phi_p \vee \psi \in \Phi_c)\}$. Why this is needed will be clarified by Example 5.2.

A similar issue occurs for the hierarchical next operators, and it can be overcome likewise by setting $\bar{F}_{\bigcirc^t_H\,\psi} = \{\Phi \in Q_\omega \mid \bigcirc^t_H\,\psi \notin \Phi_s\}$ for $t \in \{d, u\}$.

We first show the $\mathscr{DR}$ rules governing the in-stack part of states. If $\psi = \chi^{\pi}_F\,\theta \in \mathrm{Cl}(\varphi)$, $\mathscr{DR}(\psi)$ contains the following rules, besides those defined in Section 4.1:

(48) for any $(\Phi, a, \Theta) \in \delta_{push}$, ($\psi \in \Phi_p$ or $\psi \in \Phi_s$) iff $\psi \in \Theta_s$;
(49) for any $(\Phi, a, \Theta) \in \delta_{shift}$, $\psi \in \Phi_s$ iff $\psi \in \Theta_s$;
(50) for any $(\Phi, \Theta, \Psi) \in \delta_{pop}$, ($\psi \in \Phi_s$ and $\psi \in \Theta_s$) iff $\psi \in \Psi_s$.

If $\psi = \bigcirc^t_H\,\theta \in \mathrm{Cl}(\varphi)$, $\mathscr{DR}(\psi)$ contains, besides those defined in Section 4.1, rules (49) and (50), and the following:

(51) for any $(\Phi, a, \Theta) \in \delta_{push}$, ($\psi \in \Phi_c$ or $\psi \in \Phi_s$) iff $\psi \in \Theta_s$.

We show how the rules for χ^{π}_F operators work through a few examples; those for $\bigcirc^t_H$ work in the same way.

Example 5.1. Fig. 14 shows a prefix of an accepting run of $\mathcal{A}^\omega_\varphi$ for $\varphi = \chi^d_F\,\textbf{ret}$, which holds in positions 1 and 2 of the infinite word $(\textbf{call call han exc ret ret})^\omega$. In the initial state, the automaton guesses that $\chi^d_F\,\textbf{ret}$ will be satisfied by $\chi^{\doteq}_F\,\textbf{ret}$, and so does in $\Phi^g(2)$. The first push move puts $\chi^{\doteq}_F\,\textbf{ret}$ in $\Phi^g(2)$ as a pending obligation, and the next one stores $\Phi^g(2)$ in the stack. Due to rule (48), $\chi^{\doteq}_F\,\textbf{ret}$ is stored in the in-stack part of the next state $\Phi^g(3)$, to signal that $\chi^{\doteq}_F\,\textbf{ret}$ is pending in the stack. The run then goes on according to the $\mathscr{DR}$ rules for the $\chi^{\doteq}_F$ operator given in Section 4.1. All the next push and shift moves propagate it in the in-stack parts thanks to resp. rules (48) and (49). When reaching the first **ret**, state $\Phi^g(3)$, which contains a pending $\chi^{\doteq}_F\,\textbf{ret}$, is popped. However, another instance of $\chi^{\doteq}_F\,\textbf{ret}$ is still pending in $\Phi^g(2)$, so $\chi^{\doteq}_F\,\textbf{ret}$ must be kept into the in-stack part. This is accomplished by rule (50): the fact that $\chi^{\doteq}_F\,\textbf{ret}$ is in the in-stack part of the popped state means that another instance of it was pending when it was pushed, so it is propagated into the in-stack part of the next state. Its propagation stops when $\Phi^g(3)$ is popped (step 7–8): since $\chi^{\doteq}_F\,\textbf{ret} \notin \Phi^g_s(3)$, rule (50) does not allow it into $\Phi'_s(6)$, so $\chi^{\doteq}_F\,\textbf{ret} \notin \Phi'_s(6)$. The $\mathscr{DR}$ rules for $\chi^{\doteq}_F\,\textbf{ret}$ put it in $\Phi'_p(6)$ as it has not yet been satisfied. This happens with the shift move that reads the second **ret**, and state $\Phi(7)$ does not contain $\chi^{\doteq}_F\,\textbf{ret}$ in its pending or in-stack part, so $\Phi(7) \in \bar{F}_{\chi^{\doteq}_F\,\textbf{ret}}$. $\mathcal{A}^\omega_\varphi$

step	input	state	stack	PR	move
1	**call call ret call** ...	$\Phi^g(1) = (\{\textbf{call}, \chi_F^d\ \textbf{ret}, \chi_F^{\doteq}\ \textbf{ret}\}, \{\zeta_L\}, \emptyset)$	$\bot$	# $\lessdot$ **call**	push
2	**call ret call** ...	$\Phi^g(2) = (\{\textbf{call}\}, \{\chi_F^{\doteq}\ \textbf{ret}, \zeta_L\}, \emptyset)$	$[\textbf{call}, \Phi^g(1)]\bot$	**call** $\lessdot$ **call**	push
3	**ret call** ...	$\Phi^g(3) = (\{\textbf{ret}\}, \emptyset, \{\chi_F^{\doteq}\ \textbf{ret}\})$	$[\textbf{call}, \Phi^g(2)][\textbf{call}, \Phi^g(1)]\bot$	**call** $\doteq$ **ret**	shift
4	**call** ...	$\Phi(4) = (\{\textbf{call}\}, \emptyset, \{\chi_F^{\doteq}\ \textbf{ret}\})$	$[\textbf{ret}, \Phi^g(2)][\textbf{call}, \Phi^g(1)]\bot$	**ret** $\gtrdot$ **call**	pop
5	**call** ...	$\Phi'(4) = \Phi^g(4) = \Phi^g(2)$	$[\textbf{call}, \Phi^g(1)]\bot$	**call** $\lessdot$ **call**	push

Fig. 15. Example run of the automaton for χ_F^d **ret** on word **call**(**call ret**)$^\omega$.

step	input	state	stack	PR	move
1	**call call ret call** ...	$\Phi^g(1) = (\{\textbf{call}, \chi_F^d\ \textbf{call}, \chi_F^{\lessdot}\ \textbf{call}\}, \{\zeta_L\}, \emptyset)$	$\bot$	# $\lessdot$ **call**	push
2	**call ret call** ...	$\Phi^g(2) = (\{\textbf{call}\}, \{\chi_F^{\lessdot}\ \textbf{call}, \zeta_L\}, \emptyset)$	$[\textbf{call}, \Phi^g(1)]\bot$	**call** $\lessdot$ **call**	push
3	**ret call** ...	$\Phi^g(3) = (\{\textbf{ret}\}, \emptyset, \{\chi_F^{\lessdot}\ \textbf{call}\})$	$[\textbf{call}, \Phi^g(2)][\textbf{call}, \Phi^g(1)]\bot$	**call** $\doteq$ **ret**	shift
4	**call** ...	$\Phi(4) = (\{\textbf{call}\}, \emptyset, \{\chi_F^{\lessdot}\ \textbf{call}\})$	$[\textbf{ret}, \Phi^g(2)][\textbf{call}, \Phi^g(1)]\bot$	**ret** $\gtrdot$ **call**	pop
5	**call** ...	$\Phi'(4) = \Phi^g(4) = \Phi(2)$	$[\textbf{call}, \Phi^g(1)]\bot$	**call** $\lessdot$ **call**	push

Fig. 16. Example run of the automaton for χ_F^d **call** on word **call**(**call ret**)$^\omega$.

then goes back to $\Phi^g(1)$, and its subsequent behavior is cyclic. Thus, $\Phi(7)$ is visited infinitely often, and the run is accepting.

Fig. 15 shows a prefix of a rejecting run of the same automaton on word **call**(**call ret**)$^\omega$. The run starts with $\Phi^g(1)$, where $\mathcal{A}_\varphi^\omega$ guesses —wrongly— that χ_F^d **ret** will be satisfied by $\chi_F^{\doteq}$ **ret**. Rules from Section 4.1 put $\chi_F^{\doteq}$ **ret** in $\Phi_p^g(2)$, which is then pushed to the stack when reading the second **call**. Thus, according to rule (48), we have $\chi_F^{\doteq}$ **ret** $\in \Phi_s^g(3)$. $\chi_F^{\doteq}$ **ret** is propagated to $\Phi_s(4)$ by the subsequent shift move, but it is removed from the in-stack part by the next pop move, because $\chi_F^{\doteq}$ **ret** $\notin \Phi_s^g(2)$. However, rule (7) imposes that $\chi_F^{\doteq}$ **ret** $\in \Phi_p^g(2)$, marking that the satisfaction of $\chi_F^{\doteq}$ **ret** is still pending. From now on, the automaton cycles between states $\Phi^g(2)$, $\Phi^g(3)$ and $\Phi(4)$. These states contain $\chi_F^{\doteq}$ **ret** either in their pending or in-stack part, so none of them is accepting, and the run is rejected. Indeed, $\chi_F^{\doteq}$ **ret** does not hold in position 1, nor does χ_F^d **ret**: the run originating from $\mathcal{A}_\varphi^\omega$ initially guessing $\chi_F^{\lessdot}$ **ret** is symmetric and also rejecting.

Example 5.2. Fig. 16 shows a prefix of an accepting run of $\mathcal{A}_\varphi^\omega$ for $\varphi = \chi_F^d$ **call**, which holds in position 1 of the infinite word **call**(**call ret**)$^\omega$. First, the automaton guesses that χ_F^d **call** will be satisfied by $\chi_F^{\lessdot}$ **call**, which becomes pending after the first push move by rule (9). Then, $\chi_F^{\lessdot}$ **call** is put into the in-stack part of the current state by rule (48) and propagated by (49). The next pop move stops its propagation, but $\chi_F^{\lessdot}$ **call** is in the pending part of $\Phi'(4)$ by rule (10). Note that here we can conclude that $\chi_F^{\lessdot}$ **call** is satisfied in position 1, because $\chi(1,4)$, **call** $\lessdot$ **call** and **call** holds in position 4. Indeed, $\Phi'(4) = \Phi^g(2) \in \bar{F}_{\chi_F^{\lessdot}\,\textbf{call}}$ even if $\chi_F^{\lessdot}$ **call** $\in \Phi_p^g(2)$, because $\chi_F^{\lessdot}$ **call** $\notin \Phi_s^g(2)$ and **call** $\in \Phi_c^g(2)$. Since the rest of the run cycles between $\Phi^g(2)$, $\Phi^g(3)$ and $\Phi(4)$, it gets accepted.

Thus, we state the following:

Lemma 5.3. *Given a finite set of atomic propositions AP, an OP alphabet $(\mathcal{P}(AP), M_{AP})$, and a formula $\psi \in \mathrm{Cl}_{st}(\varphi)$, let $\mathcal{A}_\varphi^\omega$ be the ωOPBA built for a formula φ such that $\psi \in \mathrm{Cl}(\varphi)$.*

For any ω-word $w = \#xy$ on $(\mathcal{P}(AP), M_{AP})$, let $\langle y, \Phi, \gamma\rangle$ be $\mathcal{A}_\varphi^\omega$'s configuration after reading x.

If $\psi = \chi_F^\pi\ \theta$ for $\pi \in \{\lessdot, \doteq, \gtrdot\}$ (resp. $\psi = \bigcirc_H^t\ \theta$ for $t \in \{d, u\}$), then there exists a stack symbol $[a, \Theta] \in \gamma$ such that $\psi \in \Theta_p$ (resp. $\psi \in \Theta_c$) iff $\psi \in \Phi_s$.

We omit the proof, as it is substantially similar to the one of Lemma 6.1 in [24].

Only chain next operators need to be in $\mathrm{Cl}_{st}(\varphi)$, because satisfaction of until operators depends on them. Correctness proofs for past and next operators still hold in their current form; we need, instead, to re-prove Lemmas 4.2, A.1, and 4.3, to show that their *inductive claim* also holds with the generalized Büchi acceptance condition. We only re-prove those for chain next operators, because the modifications required for Lemma 4.3 are similar.

Lemma 5.4. *Given a finite set of atomic propositions AP, an OP alphabet $(\mathcal{P}(AP), M_{AP})$, and a formula $\chi_F^\pi \psi$ with $\pi \in \{\lessdot, \doteq, \gtrdot\}$, let $\mathcal{A}_\varphi^\omega$ be the ωOPBA built for a formula φ such that $\chi_F^\pi \psi \in \mathrm{Cl}(\varphi)$; and let $\mathcal{A}^\omega_{\varphi - \chi_F^\pi \psi}$ be the ωOPBA built as $\mathcal{A}_\varphi^\omega$ but using $\mathscr{DR} \setminus \mathscr{DR}(\chi_F^\pi \psi)$ for δ, and $\mathbf{F} \setminus \{\bar{F}_{\chi_F^\pi \psi}\}$ as the set of acceptance sets.*
Inductive assumption: *in all accepting computations of $\mathcal{A}_\varphi^\omega$ for each position i in the input word w and for each sub-formula $\psi' \in \mathrm{ssubf}(\chi_F^\pi \psi)$ we have $(w, i) \models \psi'$ iff $\psi' \in \Phi_c^g(i)$.*
***Inductive claim** I: $[I_1]$ A computation ρ of $\mathcal{A}_\varphi^\omega$ is accepting if and only if $[I_2]$ ρ is accepting for $\mathcal{A}^\omega_{\varphi - \chi_F^\pi \psi}$ and for each position i in the input word w we have $(w, i) \models \chi_F^\pi \psi$ iff $\chi_F^\pi \psi \in \Phi_c^g(i)$.*

Proof. We prove an **auxiliary claim** built on the following assertions:

- Let $[A_1^\omega]$ be: $(w, i) \models \chi_F^\pi \psi$;
- let $[A_2^\omega]$ be: all accepting computations of $\mathcal{A}_\varphi^\omega$ bring it from a configuration $\langle yz, \Phi^g(i), \alpha\gamma\rangle$ with $\chi_F^\pi \psi \in \Phi_c^g(i)$ to a configuration $\langle z, \Phi^g(i_z), \alpha'\gamma\rangle$ such that $\chi_F^\pi \psi \notin \Phi_p^g(i_z)$, $|\alpha| = 1$ and $|\alpha'| = 1$ if $\mathrm{first}(y)$ is read by a shift move, $|\alpha'| = 2$ if it is read by a push move;
- let $[A_3^\omega]$ be: $\pi = \lessdot$ and all accepting computations of $\mathcal{A}_\varphi^\omega$ bring it from a configuration $\langle yz, \Phi^g(i), \alpha\gamma\rangle$ with $\chi_F^{\lessdot} \psi \in \Phi_c^g(i)$ to an infinite sequence of configurations $\langle b_k \ldots, \Phi^g(i_{b_k}), \alpha'\gamma\rangle$, such that $k \in \mathbb{N}$, $\chi_F^{\lessdot} \psi \in \Phi_p^g(i_{b_k})$, $\psi \in \Phi_c^g(i_{b_k})$, $|\alpha| = 1$ and $|\alpha'| = 1$ if $\mathrm{first}(y)$ is read by a shift move, $|\alpha'| = 2$ if it is read by a push move.

We prove that for any ω-word $w = \#xyz$ and positions $i = |x| + 1$, $i_z = |xy| + 1$ in w, $A_1^\omega \iff (A_2^\omega \vee A_3^\omega)$.

In the proofs of Lemmas 4.2 and A.1 we proved that the same auxiliary claim holds in the finite-word case for $\chi_F^\pi \psi$ with any π, so we would like to show that $A_1 \iff A_2$ implies $A_1^\omega \iff A_2^\omega$. However, if w is an ω-word and $\pi = \lessdot$, $\chi_F^{\lessdot} \psi$ may be satisfied in i because there exist infinitely many positions i_{b_k} such that $\chi(i, i_{b_k})$, $i \lessdot i_{b_k}$, and $(w, i_{b_k}) \models \psi$ (cf. Example 5.2). In this case, $A_1^\omega \iff A_2^\omega$ does not hold, but $A_1^\omega \iff A_3^\omega$ does.

First, we note that $\mathscr{DR}$ rules of $\mathcal{A}_\varphi$ are a subset of $\mathcal{A}_\varphi^\omega$'s, and $\mathcal{A}_\varphi^\omega$'s additional rules (48)–(50) do not interfere with others. In fact, by Lemma 5.3 any computation of $\mathcal{A}_\varphi$ can be transformed into one of $\mathcal{A}_\varphi^\omega$ by adding in-stack parts to states Φ so that, for all $\theta \in \mathrm{Cl}_{st}(\varphi)$, if the stack contains a symbol $[a, \Theta]$ such that $\theta \in \Theta_p$, then $\theta \in \Phi_s$.

Moreover, $\mathcal{A}_\varphi^\omega$'s computations are infinite, so they all reach i_z, if it exists. Thus, $A_1 \Rightarrow A_2$ clearly implies $A_1^\omega \Rightarrow A_2^\omega$. Concerning $A_2^\omega \Rightarrow A_1^\omega$, we note that in the proofs of $A_2 \Rightarrow A_1$ the acceptance condition of $\mathcal{A}_\varphi$ does not matter, and A_1 follows from the computation reaching i_z. This proves that $A_1 \iff A_2$ implies $A_1^\omega \iff A_2^\omega$.

As stated earlier, $A_1 \iff A_2$ may not always hold in the ω case. The only case when this happens is when $\pi = \lessdot$ and $\chi_F^{\lessdot} \psi$ is satisfied by infinite positions i_{b_k}, for $k \in \mathbb{N}$: in this case there exists no i_z such that A_2^ω holds, and we must prove $A_1^\omega \iff A_3^\omega$. The proof of this claim closely resembles the one of Lemma A.1, so we do not repeat it fully. $A_3^\omega \Rightarrow A_1^\omega$ easily follows by the existence of infinitely many positions where ψ holds, and $A_1^\omega \Rightarrow A_3^\omega$ can be shown by detailing a generic computation of $\mathcal{A}_\varphi^\omega$ on the word structure of Fig. 12.

Now, we prove the inductive claim.

$I_1 \Rightarrow I_2$ follows from $A_1^\omega \iff (A_2^\omega \vee A_3^\omega)$ and $\mathcal{A}_\varphi^\omega$'s transition relation being a subset of $\mathcal{A}^\omega_{\varphi - \chi_F^\pi \psi}$'s.

$I_2 \Rightarrow I_1$ also follows from the auxiliary claim, but we must also show that computations that satisfy I_2 are accepting for $\mathcal{A}_\varphi^\omega$. If a computation is accepting for $\mathcal{A}^\omega_{\varphi - \chi_F^\pi \psi}$, then it satisfies all acceptance sets except possibly $\bar{F}_{\chi_F^\pi \psi}$. We must show that a state in $\bar{F}_{\chi_F^\pi \psi}$ occurs infinitely often if I_2 holds.

If $\chi_F^\pi \psi$ is not in $\Phi_p^g(i)$ nor in the pending part of any state in $\alpha\gamma$, then it is also not in the pending part of any state in $\alpha'\gamma$. This can be easily shown by noting that γ remains the same, and according to the proofs of

Lemmas 4.2 and A.1, α' is made at most by one symbol from α updated by a shift move (which does not change its state), and a newly-pushed symbol which, however, is not constrained to contain $\chi_F^\pi\,\psi$ in its pending part by any $\mathscr{D}\mathscr{R}$ rule. By Lemma 5.3, this means $\chi_F^\pi\,\psi \notin \Phi_s^g(i_z)$, so $\Phi^g(i_z) \in \bar{F}_{\chi_F^\pi\,\psi}$ if A_2^ω holds. If $\chi_F^\pi\,\psi$ never holds again in the computation, it is trivially accepting; if it holds infinitely many times, a state in $\bar{F}_{\chi_F^\pi\,\psi}$ is also visited infinitely many times, so the computation is accepting. If, instead, A_3^ω holds, then $\mathcal{A}_\varphi^\omega$ visits a sequence of states that contain $\chi_F^\lessdot\,\psi$ in their pending part and ψ in their current part, which makes them accepting for $\bar{F}_{\chi_F^\lessdot\,\psi}$.

If $\chi_F^\pi\,\psi$ is in $\Phi_p^g(i)$ or in the pending part of some state(s) in $\alpha\gamma$, it means one or more previous instances of $\chi_F^\pi\,\psi$ appeared in previous states of the computation, and are pending. Let i' be the leftmost word position where one of such instances holds: the same reasoning we made for i can be applied to the instance holding in i', proving that the computation reaches a configuration where $\chi_F^\pi\,\psi$ does not appear any more in the current state nor in the stack. □

An acceptance condition for summary until operators is also needed, so that computations in which the satisfaction of an until operator is postponed forever are rejected. For $\psi\,\mathcal{U}_\chi^d\,\theta \in \mathrm{Cl}(\varphi)$, we add an acceptance set

$$\bar{F}_{\psi\mathcal{U}_\chi^d\theta} = \bar{F}_{\chi_F^{\doteq}(\psi\mathcal{U}_\chi^d\theta)} \cap \bar{F}_{\chi_F^{\lessdot}(\psi\mathcal{U}_\chi^d\theta)} \cap \{\Phi \in Q_\omega \mid \psi\,\mathcal{U}_\chi^d\,\theta \notin \Phi_c \vee \theta \in \Phi_c\}.$$

The condition that $\psi\,\mathcal{U}_\chi^d\,\theta \notin \Phi_c$ or $\theta \in \Phi_c$ allows for accepting computations where each instance of $\psi\,\mathcal{U}_\chi^d\,\theta$ is satisfied, possibly occurring infinitely often. These states are intersected with those accepting for $\chi_F^{\doteq}(\psi\,\mathcal{U}_\chi^d\,\theta)$ and $\chi_F^{\lessdot}(\psi\,\mathcal{U}_\chi^d\,\theta)$, to make sure that no until operator is pending or "hidden" in the stack.

The condition for $\psi\,\mathcal{U}_\chi^u\,\theta$ is obtained by substituting $\gtrdot$ for $\lessdot$. The conditions for hierarchical operators are similar:

$$\bar{F}_{\psi\mathcal{U}_H^u\theta} = \bar{F}_{\bigcirc_H^u(\psi\mathcal{U}_H^u\theta)} \cap \{\Phi \in Q_\omega \mid \psi\,\mathcal{U}_H^u\,\theta \notin \Phi_c \vee \theta \in \Phi_c\},$$
$$\bar{F}_{\psi\mathcal{U}_H^d\theta} = \bar{F}_{\bigcirc_H^d(\psi\mathcal{U}_H^d\theta)} \cap \{\Phi \in Q_\omega \mid \psi\,\mathcal{U}_H^u\,\theta \notin \Phi_c \vee \theta \in \Phi_c\} \cap \bar{F}_{\chi_F^{\gtrdot}\top}.$$

We can now conclude the proof:

Theorem 5.5 (Correctness of ω-Word Model Checking). *Given a finite set of atomic propositions AP, an OP alphabet $(\mathcal{P}(AP), M_{AP})$, an ω-word w on it, and a POTL formula φ with no hierarchical operators, automaton $\mathcal{A}_\varphi^\omega$ is such that it performs at least one accepting computation on w if and only if $(w, 1) \models \varphi$.*

Proof. The proof follows the one of Theorem 4.4 verbatim, except Lemmas 4.2, A.1 and 4.3 are replaced by 5.4. □

5.1 Complexity

The complexity claims made for finite-word model checking can be extended to the infinite case, as the presence of the *in-stack* part of states does not cause a further blow-up of their amount. By Theorem 2.8 it is possible to transform the generalized ωOPBA $\mathcal{A}_\varphi^\omega$ into a normal ωOPBA with a size increase proportional to $|\mathbf{F}|$. Since $\mathbf{F}$ contains one set for each occurrence of a chain or hierarchical next or until operator in φ, we have $|\mathbf{F}| = O(|\varphi|)$, which does not change the overall complexity class. Hence,

Theorem 5.6. *Given a POTL formula φ, it is possible to build an ωOPBA accepting the language denoted by φ with at most $2^{O(|\varphi|)}$ states.*

Again, we exploit the complexity lower bounds for NWTL to claim

Theorem 5.7. *POTL model checking and satisfiability on OP ω-words are EXPTIME-complete.*

6 IMPLEMENTATION

We implemented the OPA and ωOPBA constructions of Sections 4 and 5 in an explicit-state model checking tool called POMC [23]. The tool is written in Haskell [61], a purely functional, statically typed programming language with lazy evaluation. In this section, we describe the underlying algorithms together with some remarks on more technical aspects of their implementation.

Given a POTL specification φ and an OPA (resp. ωOPBA) $\mathcal{A}$ to be checked, POMC generates the *product automaton* between $\mathcal{A}$ and the automaton $\mathcal{A}_{\neg\varphi}$ (resp. $\mathcal{A}^{\omega}_{\neg\varphi}$) built according to Section 4 (resp. 5) on-the-fly while executing a reachability algorithm. POMC checks for emptiness of the language accepted by an OPA by checking the reachability of an accepting configuration, by means of a modified Depth-First Search (DFS) of the transition relation. Language emptiness checking for ωOPBA is significantly more involved than the finite-word case, since *fair cycles* must be found in the transition relation. We accomplish this by means of graph-theoretic techniques, with algorithms already sketched in [73] that are similar to the ones developed for Recursive State Machines (RSMs) [5].

6.1 OPA Language Emptiness Checking

The transition system associated to an OPA can be infinite, because the stack may grow unboundedly. However, each transition is determined only by the topmost stack symbol, besides the current state and the input symbol. Intuitively, in runs in which the stack size grows forever, the OPA must visit a configuration featuring the same state and topmost stack symbol infinitely often, forming a cycle. The reachability algorithm exploits this fact to detect cycles in OPA behavior, and thus it does not have to explore an infinite number of configurations.

To formalize the underlying idea, we define *semi-configurations* as tuples whose elements uniquely determine the next move of an OPA, and the *semi-configuration graph*, which is derived from an OPA's transition relation by "projecting" it onto the space of semi-configurations.

Definition 6.1. Given an OP alphabet (Σ, M_Σ), where Σ is a finite input alphabet, let $\mathcal{A} = (\Sigma, M_\Sigma, Q, I, F, \delta)$ be an OPA, with stack symbols in $\Gamma = \Sigma \times Q \cup \{\bot\}$.

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

The *semi-configuration graph* of $\mathcal{A}$ is a pair $(C, \mathcal{E})$ where $\mathcal{E} \subseteq C^2$ is partitioned into the following three disjoint sets:

$$E_{push} = \{((q,g,b),(p,[b,q],\ell)) \in C^2 \mid smb(g) \lessdot b \wedge (q,b,p) \in \delta_{push} \wedge \ell \in \Sigma \cup \{\#\}\}$$

$$E_{shift} = \{((q,[a,r],b),(p,[b,r],\ell)) \in C^2 \mid a \doteq b \wedge (q,b,p) \in \delta_{shift} \wedge \ell \in \Sigma \cup \{\#\}\}$$

$$E_{supp} = \{((q,g,b),(p,g,\ell)) \in C^2 \mid \mathcal{A} \text{ has a support } q \xrightarrow{b} q' \dots q'' \overset{q}{\Longrightarrow} p \text{ and } \ell \in \Sigma \cup \{\#\}\}$$

Elements of semi-configurations represent the current state, the topmost stack symbol, and a look-ahead for the next input symbol, respectively.

The graph has three kinds of edges: push and shift edges, which represent the respective moves in $\mathcal{A}$, and *support* edges,[6] which represent a chain support (cf. Definition 2.5). The need for such edges arises from the fact that, while an OPA may perform push and shift moves freely, pop moves may only occur if a specific stack symbol is present on top of the stack. Thus, cycles of push and shift moves may be followed an arbitrary number of times, while cycles of pop moves are constrained by the number of stack symbols and, consequently, the number of previous push moves. Therefore, we use support edges to represent the whole "life" of a stack symbol, from the push move creating it to the pop move destroying it. Support edges have the additional feature that they "summarize" an entire chain support, so recursively nested chains are replaced by one single edge, producing a finite graph.

[6] Support edges are analogous to *summary* edges in [5], with the usual differences due to the use of our relation χ.

Algorithm 1 OPA semi-configuration reachability

```
1:  function Reach(q, g, c, ℓ)
2:      if (q, g, ℓ) ∈ V ∨ (q, g, ∗) ∈ V then return false
3:      V := V ∪ (q, g, ℓ)
4:      if q ∈ Q_R ∧ g ∈ Γ_R then return true
5:      a := smb(g)
6:      for all (q, b, p) ∈ δ_push s.t. a ⋖ b ∧ (b = ℓ ∨ ℓ = ∗) do
7:          SupportStarts := SupportStarts ∪ {(q, g, c)}
8:          if Reach(p, [b, q], b, ∗) then return true
9:      for all (s, q, c′, ℓ′) ∈ SupportEnds s.t. a ⋖ c′ do
10:         if Reach(s, g, c, ℓ′) then return true
11:     if g ≠ ⊥ then
12:         [a, r] := g
13:         for all (q, b, p) ∈ δ_shift s.t. a ≐ b ∧ (b = ℓ ∨ ℓ = ∗) do
14:             if Reach(p, [b, r], c, ∗) then return true
15:         for all (q, r, p) ∈ δ_pop, b ∈ Σ ∪ {#} s.t. a ⋗ b ∧ (b = ℓ ∨ ℓ = ∗) do
16:             SupportEnds := SupportEnds ∪ {(p, r, c, b)}
17:             for all (r, g′, c′) ∈ SupportStarts s.t. smb(g′) ⋖ c do
18:                 if Reach(p, g′, c′, b) then return true
19:     return false
```

Algorithm 2 OPA emptiness check

```
1:  function IsEmpty(A)
2:      (Σ, M_Σ, Q, I, F, (δ_push, δ_shift, δ_pop)) := A
3:      V := SupportStarts := SupportEnds := ∅
4:      Q_R = F
5:      Γ_R = {⊥}
6:      for all q ∈ I do
7:          if Reach(q, ⊥, #, ∗) then return false
8:      return true
```

It should be now easy to see that a path in the semi-configuration graph represents a run of $\mathcal{A}$ (the stack can be re-constructed by accumulating pushed symbols), and reachability of a node in this graph implies reachability of the semi-configuration in the OPA. Algorithm 1 solves the reachability problem for OPA by performing a DFS on the semi-configuration graph on-the-fly. Each time a chain support is explored, its ending semi-configuration is saved and associated with the starting one. So, the next time the starting semi-configuration is reached, the support does not have to be re-explored.

Function Reach receives as its arguments a state $q \in Q$, a stack symbol $g \in \Gamma$, a character $c \in \Sigma$, and $\ell \in \Sigma \cup \{*\}$. The algorithm searches the semi-configuration graph of the OPA starting from semi-configuration (q, g, ℓ) and stops when it reaches a semi-configuration (q', g', ℓ') with $q' \in Q_R$ and $g' \in \Gamma_R$, where Q_R and Γ_R are the sets of target states and stack symbols. We admit $*$ as a wildcard look-ahead representing all characters in Σ to avoid creating a separate semi-configuration for each input symbol after push and shift moves. The purpose of character c will be explained later.

The algorithm first checks whether the current semi-configuration has already been visited or is a target semi-configuration, and terminates in these cases. Otherwise, it proceeds to examine all transitions that the OPA could perform next.

The loop in line 6 explores push moves. Before analyzing the semi-configuration produced by the push move through a recursive call, it saves the current semi-configuration in the set *SupportStarts*, which contains semi-configurations from which a support begins. The contents of *SupportStarts* will be matched with pop moves to create support edges. The loop in line 13 performs shift moves by updating the input symbol in g and exploring the resulting semi-configuration.

The loop in line 15 performs pop moves: it looks into *SupportStarts* for semi-configurations that led to push moves that could have pushed g onto the stack, and for each one of them (line 17) it explores the semi-configuration resulting by their pop. Each one of the iterations of the internal loop corresponds to the exploration of a support

edge. Here we use character c, called the *latest pushed look-ahead*. We use it to match the state r in the topmost stack symbol g with the character that was pushed with it, in order to uniquely identify the push move that pushed it. We need to store it separately because the character in g could be changed by shift moves.

Until now we have ignored the role of *SupportEnds*. To see why it is needed, suppose the algorithm reaches a semi-configuration (q, g, ℓ) with latest pushed look-ahead c that leads to a push of $[q, \ell]$ to the stack: tuple (q, g, c) is inserted into *SupportStarts*, and the subsequent support explored, starting from $(p, [q, \ell], *)$. Later, $[q, \ell]$ is popped by a move that leads to a semi-configuration (s, g, ℓ') for some $\ell' \in \Sigma$ and latest pushed look-ahead c. Then, suppose a semi-configuration (q, g', ℓ) is reached with $g' \neq g$ that admits the same push move: the next semi-configuration to be explored would be, again, $(p, [q, \ell], *)$. But this semi-configuration is in V, and would not be explored. This would prevent the algorithm from exploring a summary edge that leads to (s, g', ℓ'), thus missing part of the graph.

To solve this issue, when exploring a pop move the algorithm saves into *SupportEnds* a tuple that allows it to reconstruct its target configuration. After each push move, in line 9, the algorithm uses tuples in *SupportEnds* to jump directly from the push move starting a support that has already been explored to the semi-configuration it ends with, effectively following a support edge.

To solve the emptiness problem, as shown in Algorithm 2 we pose $Q_R = F$ and $\Gamma_R = \{\bot\}$, and call $\textsc{Reach}(q, \bot, \#, *)$ for each $q \in I$.

Complexity. Each time an edge in δ is explored, Reach is called at most once for each element in *SupportStarts* and *SupportEnds*, which is bounded by $|\delta_{push}|^2$ (the number of possible push moves times stack symbols, which is also bounded by $|\delta_{push}|$). The space complexity is dominated by the size of V, which in the worst case contains all semi-configurations. Thus, each call to Reach has worst-case time complexity $O(|\delta||\delta_{push}|^2|\Sigma|)$ and space complexity $O(|Q||\delta_{push}||\Sigma|)$. Note that only transitions and states that are actually visited contribute to the complexity, so the above bounds are reached only if the whole OPA is visited. 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.

Remark. When using these algorithms for model checking, states are pairs of states of $\mathcal{A}$ and $\mathcal{A}_{\neg\varphi}$. According to $\mathcal{DR}$ rule (1) from Section 4.1, states of $\mathcal{A}_{\neg\varphi}$ contain exactly the atomic propositions that will be read by subsequent push and shift moves. Thus, it is possible to omit the look-ahead ℓ and the latest pushed look-ahead c, by extracting sets of atomic propositions from states. So, $|\Sigma|$ can be removed from the above complexity bounds.

6.2 ωOPBA Emptiness Checking

The algorithm for checking emptiness of an ωOPBA has been developed in [72]. Due to the Büchi acceptance condition, to check whether an ωOPBA has an accepting run we need to check for reachable cycles containing final states. In Nondeterministic Büchi Automata (NBAs) this is done with a nested DFS, but adapting this algorithm to ωOPBAs is sub-optimal, as was shown for RSMs in [5]. Thus, following the same approach as [5], we use an on-line algorithm to incrementally compute Strongly Connected Components (SCCs) while summary edges are discovered.

We use the path-based algorithm by H.N. Gabow [40], which is well-suited for early-termination and can be easily combined with the transition graph's exploration, because it is based on a DFS. This algorithm performs a DFS on the graph, and contracts SCCs as it finds back-edges. It finds all SCCs in a graph in linear time, by using simple data structures such as arrays and stacks.

The overall fair-cycle detection algorithm works by alternating two phases:

- a *search* phase, in which the transition graph of the ωOPBA is explored and processed by the SCC algorithm without following summary edges (or chain supports), which are stored in a set;

- a *collapse* phase, where summary edges collected in the search phase are added to the graph, and the SCC algorithm only is run once again; resulting new SCCs are collapsed into one single node, if any.

After the collapse phase, a new search phase is launched starting from semi-configurations reached by summary edges, and so on. If a SCC containing final states is detected during any of the two phases, the algorithm terminates, as an accepting run has been found. Otherwise, the algorithm terminates once no more summary edges have been found, which means that the ωOPBA accepts the empty language.

Complexity. This algorithm has a worst-case time complexity of $O(k|\delta||\delta_{push}|^3|\Sigma|)$, where k is the number of SCCs found, and space complexity $O(|\delta||\delta_{push}|^2|\Sigma|)$. k is bounded by $|Q|$, leading to a time bound $O(|Q||\delta||\delta_{push}|^3|\Sigma|)$. We can make the same considerations on the size of Σ as in Section 6.1.

On the choice of Haskell as the implementation language

The model checkers that have obtained most success in the research community are written in imperative programming languages. Just to name a few, the already mentioned SPIN [50] and NuSMV [28] are written in C, and UPPAAL [11] in C++. In this context, the choice of writing a model checker in a purely functional programming language like Haskell deserves some remarks.

The main reason for this choice was that the declarative nature of Haskell and its syntax, which is close to mathematical notation, make it easier to code the numerous rules required by the automaton construction procedure. Moreover, all such rules only need to be activated when the relevant formula is present in the closure. We exploit higher order functions and lazy evaluation to evaluate only rules that are actually needed, something that would require substantial engineering efforts in imperative languages, but that comes naturally with Haskell (in practice, the automaton's transition relation is a thunk that contains only references to functions encoding the relevant rules).

The main drawback of using a functional language is that the reachability algorithms are based on a DFS, which is an inherently sequential algorithm: the global sets and maps used to keep track of visited semi-configurations, as well as *SupportStarts* and *SupportEnds*, do not cope well with referential transparency (in practice, such data structures would need to be partially duplicated at any update, with considerable overhead). Luckily, Haskell offers monads to express sequential computations and, in particular, the `ST` monad implements the *lazy functional state threads* paradigm [56], which allows us to employ mutable data structures embedded in a purely functional context. This represents the standard solution for structuring a DFS search in a lazy functional language [53].

The overall result is a relatively small ($\sim$ 5000 lines of code) and maintainable code base, without sacrificing efficiency.

7 EXPERIMENTAL EVALUATION

We evaluate POMC on two benchmark suites which we made publicly available [23]. The first one consists of three case studies that were modeled manually as OPAs in [25], and which we now also model as MiniProc programs (Section 7.1), complemented by a systematic verification of the largest one of such programs against a variety of requirements expressed as nontrivial POTL formulas. The main goal of these benchmarks is to stress the key features of OPLs and POTL to evaluate the potential practical application thereof in terms of model checking. The second suite (Section 7.2) comprises different MiniProc implementation variants of the QuickSort algorithm, and continues from the Example 2.10. In this case the accent is on verifying algorithmic correctness and program termination, possibly in the presence of exceptions. While the case studies in Section 7.1 were initially conceived as OPAs on finite-length sentences and subsequently one of them was tested for ω-languages too, in Section 7.2 we use exclusively ωOPBAs, as we need to perform termination analysis.

```
main() {
  pa();
  try {
    pa();
    pb();
  } catch {
    perr();
  }
}
```

```
pa() {
  pc();
  pd();
  if (*) {
    pa();
  } else {}
}
```

```
pb() {
  try {
    pe();
  } catch {
    perr();
  }
}
```

```
pc() {
  if (*) {
    pa();
  } else {
    pe();
  }
}
```

```
pd() {
  pc();
  pa();
}

perr() {}
```

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

Fig. 17. "Basic larger" MiniProc program.

7.1 Basic Case Studies

The first three benchmarks of this section are a simple and a more complex case of stack inspection, and one of exception safety. The fourth one instead is systematic verification of a single program against many different POTL formulas; in this case the verification is carried over both for the OPA and for its ω version.

Simple stack inspection. We checked formula

$$\Box\left((\mathbf{call} \wedge \mathrm{p}_B \wedge \mathit{Scall}(\top, \mathrm{p}_A)) \implies \mathit{CallThr}(\top)\right)$$

from Section 3.3 on two simple MiniProc programs similar to the one of Fig. 4b, named Simple1 and Simple2 in the first rows of Tables 1 and 2 and a third one, called "basic larger" in the third row of the same tables, and shown in Fig. 17.

Java-inspired stack inspection. The security framework of the Java Development Kit (JDK) is based on stack inspection, i.e. the analysis of the program stack contents during 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.

Such needs are explained in [51] through an example Java program for managing a bank account. It allows the user to check the account balance and withdraw money. To perform such tasks, the invoking program must have been granted permissions `CanPay` and `Debit`, respectively. We modeled this program as MiniProc code and as an OPA, both named Java Security in the fourth row of Tables 2 and 1 respectively, and proved that the program enforces security measures effectively by checking it against the formula

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

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. This case study comes from a tutorial on how to make exception-safe generic containers in C++ [75]. It consists of 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.

Table 1. Results of the evaluation of hand-made OPAs. '# states' refers to the OPA to be verified.

#	Benchmark name	# states	Time (ms)	Total Memory (KiB)	MC Memory (KiB)	Result
1	Simple1	12	1,009	73,632	6,096	True
2	Simple2	24	707	73,671	1,911	False
3	Basic larger (Fig. 17)	30	1,214	73,633	9,104	True
4	Java security	42	289	71,504	1,756	True
5	unsafe stack	63	1,332	71,482	21,095	False
6	safe stack	77	596	71,480	3,979	True
7	unsafe stack neutrality	63	4,821	209,981	83,850	True
8	safe stack neutrality	77	787	71,486	8,864	True

While exception safety is undecidable in C++, here we consider the weaker requirement that each modification to the data structure is only committed once no more exceptions can be thrown. We modeled both implementations as OPAs, and checked this requirement against the following formula:

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

POMC successfully found a counterexample for the first implementation named unsafe stack in the fifth row of the two tables, and proved safety of the second one named safe stack in the sixth row.

Additionally, we proved that both implementations are *exception neutral* as reported in rows 7 and 8 of both tables, i.e. `Stack` functions do not block exceptions thrown by the underlying type `T`. This was accomplished by checking the following formula, where `Stack` identifies all methods of the `Stack` class:

$$\Box(\mathbf{exc} \wedge \ominus^u \top \wedge \chi_P^d(\mathbf{han} \wedge \chi_P^d \texttt{Stack}) \implies \chi_P^d \chi_P^d \chi_F^u \mathbf{exc}).$$

Checking the basic larger program against a variety of formulas. To complete the first suite of experiments we performed a systematic check of the program of Fig. 17 against many formulas devised with the purpose of testing all POTL operators. They differ from each other in meaning, length and complexity. Such formulas are explicitly and identically reported in Tables 3 and 4. This experiment has also the goal of comparing the tool performances in the two cases of OPA and ωOPBA. Both automata were automatically generated from the MiniProc code.

Results. These experiments were executed on a laptop with a 2.2 GHz Intel processor and 15 GiB of RAM, running Ubuntu GNU/Linux 20.04.

The results of the first three benchmarks are shown in Tables 1 and 2, reporting, respectively, on the implementations on manually designed OPAs and those automatically generated from MiniProc. Tables 3 and 4 instead list explicitly the —further— POTL formulas used to verify the "basic larger" program and the results obtained for the OPA and the ωOPBA, respectively. Tables 1 through 3 were included in artifact [26].

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" the maximum memory used by model checking as reported by the runtime.

Table 1 shows that model checking on hand-made OPAs runs in at most a few seconds, and with a modest memory occupancy. In Table 2, when checking the same case studies by using automatically-generated OPAs as models, the execution times increase significantly due to the larger size of generated OPAs, which is consistent with the reachability algorithm having a super-linear (but still polynomial) computational complexity. The time and memory requirements remain, however, reasonable. The same can be said about the numerous formulas that

Table 2. Results of the evaluation of MiniProc programs automatically compiled into OPAs. '# states' refers to the OPA to be verified.

#	Benchmark name	# states	Time (ms)	Total Memory (KiB)	MC Memory (KiB)	Result
1	Simple1	19	1,028	71,493	7,009	True
2	Simple2	31	743	71,490	2,138	False
3	Basic larger (Fig. 17)	44	1,315	71,487	8,125	True
4	Java security	1236	1,839	71,489	17,571	True
5	unsafe stack	162	2,869	88,394	33,990	False
6	safe stack	340	11,572	523,531	207,545	True
7	unsafe stack neutrality	162	12,670	468,025	197,892	True
8	safe stack neutrality	340	18,474	760,313	312,682	True

we check in Table 3, most of which take less than one second, except a few outliers, which highlight the fact that the process is exponential in formula length.

Finally, in Table 4 we see the behavior of the ωOPBA emptiness algorithms on the same formulas as in Table 3. In this case, the execution times increase significantly, due to the higher complexity of finding SCCs instead of just checking reachability. However, the time taken by most formulas remains of at most a few seconds, and the same can be said for memory occupancy. There are a few outliers, this time more than in the finite-word case, and one of them even runs out of memory. Again, this is a symptom of the worst-case complexity of the problem, which manifests itself with longer formulas and with hierarchical operators in particular. We did not try the case studies from Table 2 as ωOPBA, because the properties we check do not make sense in the ω-word case.

In conclusion, we can state that the results are promising also in practice, and this opens the way to the use of these techniques for checking more complex systems, such as real-world programs, or parts thereof.

7.2 QuickSort

This benchmark is an adaptation from the suite packaged with the Moped model checker [52]; it consists of a Java implementation of the QuickSort algorithm, which we tailored to exceptions, to show the greater expressive power of POMC to model real-world procedural programs. QuickSort is a well-known sorting algorithm and an ideal case study for the verification of infinite state-space programs, because it admits a naturally recursive implementation. In different variants, it has been targeted in the literature by some state-of-the-art program verifiers (such as DAFNY [21] and STAINLESS [45]).

Two versions of QuickSort are included in Moped: a correct one (*Correct Quicksort*), and a faulty one (called *Buggy Quicksort*, or *Error Quicksort*) which enters an infinite loop for some input arrays, and of which we presented a MiniProc version in Fig. 6. Moped [37, 74] has been used to prove, respectively, the correctness of the former one and the incorrectness of the latter one.

Our first benchmark is an enrichment of the correct QuickSort in Moped, by introducing the management of exceptions. The MiniProc code, called *Semisafe*, is shown in Fig. 18. The QuickSort procedure `qs()` receives as input an array of objects to be sorted. Since the objects are of non-primitive types, the array may contain null references. If one of them is read by the procedure, it throws a `NullPointerException`, potentially terminating the program abnormally. Thus, we devised the following version of the algorithm: procedure `qs()` is first called in a `try-catch` block. If it throws an exception, an input-sanitizing procedure named `parseList()` is called, which removes null references from the array, thus enforcing *void safety* [65]. Then, `qs()` is called on the new array.

Table 3. Results of additional experiments on the program of Fig. 17, automatically translated into an OPA with 44 reachable states. The abbreviations are: M. T. = Total Memory, M. MC = Memory for Model Checking only.

Formula	Time (ms)	M.T. (MiB)	M.MC (KiB)	Results
$\chi_F^d\, \mathrm{p}_{Err}$	0.9	70	160	False
$\bigcirc^d(\bigcirc^d(\mathbf{call} \wedge \chi_F^u\, \mathbf{exc}))$	25.9	70	870	False
$\bigcirc^d(\mathbf{han} \wedge (\chi_F^d(\mathbf{exc} \wedge \chi_P^u\, \mathbf{call})))$	45.6	70	1,354	False
$\Box(\mathbf{exc} \implies \chi_P^u\, \mathbf{call})$	12.1	70	599	True
$\top\, \mathcal{U}_\chi^d\, \mathbf{exc}$	2.0	70	141	False
$\bigcirc^d(\bigcirc^d(\top\, \mathcal{U}_\chi^d\, \mathbf{exc}))$	4.4	70	119	False
$\Box((\mathbf{call} \wedge \mathrm{p}_A \wedge (\neg\mathbf{ret}\, \mathcal{U}_\chi^d\, \mathrm{WRx})) \implies \chi_F^u\, \mathbf{exc})$	5,388.9	121	49,135	True
$\bigcirc^d(\bigcirc^u\, \mathbf{call})$	0.5	70	105	False
$\bigcirc^d(\bigcirc^d(\bigcirc^d(\ominus^u\, \mathbf{call})))$	3.2	70	145	False
$\chi_F^d(\bigcirc^d(\ominus^u\, \mathbf{call}))$	1.4	70	148	False
$\Box((\mathbf{call} \wedge \mathrm{p}_A \wedge \mathit{CallThr}(\top)) \implies \mathit{CallThr}(\mathrm{e}_B))$	13,119.2	200	80,975	False
$\Diamond(\bigcirc_H^d\, \mathrm{p}_B)$	2.4	70	120	False
$\Diamond(\ominus_H^d\, \mathrm{p}_B)$	3.4	70	120	False
$\Diamond(\mathrm{p}_A \wedge (\mathbf{call}\, \mathcal{U}_H^d\, \mathrm{p}_C))$	599.0	70	16,547	True
$\Diamond(\mathrm{p}_C \wedge (\mathbf{call}\, \mathcal{S}_H^d\, \mathrm{p}_A))$	778.6	70	17,305	True
$\Box((\mathrm{p}_C \wedge \chi_F^u\, \mathbf{exc}) \implies (\neg\mathrm{p}_A\, \mathcal{S}_H^d\, \mathrm{p}_B))$	134,494.0	5,920	2,641,030	False
$\Box(\mathbf{call} \wedge \mathrm{p}_B \implies \neg\mathrm{p}_C\, \mathcal{U}_H^u\, \mathrm{p}_{Err})$	175.8	70	7,226	True
$\Diamond(\bigcirc_H^u\, \mathrm{p}_{Err})$	1.3	70	125	False
$\Diamond(\ominus_H^u\, \mathrm{p}_{Err})$	1.4	70	124	False
$\Diamond(\mathrm{p}_A \wedge (\mathbf{call}\, \mathcal{U}_H^u\, \mathrm{p}_B))$	11.2	70	117	False
$\Diamond(\mathrm{p}_B \wedge (\mathbf{call}\, \mathcal{S}_H^u\, \mathrm{p}_A))$	11.9	70	117	False
$\Box(\mathbf{call} \implies \chi_F^d\, \mathbf{ret})$	3.5	70	115	False
$\Box(\mathbf{call} \implies \neg\bigcirc^u\, \mathbf{exc})$	2.5	70	115	False
$\Box(\mathbf{call} \wedge \mathrm{p}_A \implies \neg\mathit{CallThr}(\top))$	150.0	70	2,997	False
$\Box(\mathbf{exc} \implies \neg(\ominus^u(\mathbf{call} \wedge \mathrm{p}_A) \vee \chi_P^u(\mathbf{call} \wedge \mathrm{p}_A)))$	30.7	70	119	False
$\Box((\mathbf{call} \wedge \mathrm{p}_B \wedge (\mathbf{call}\, \mathcal{S}_\chi^d\, (\mathbf{call} \wedge \mathrm{p}_A))) \implies \mathit{CallThr}(\top)$	1,242.5	70	8,143	True
$\Box(\mathbf{han} \implies \chi_F^u\, \mathbf{ret})$	20.4	70	659	True
$\top\, \mathcal{U}_\chi^u\, \mathbf{exc}$	7.0	70	137	True
$\bigcirc^d(\bigcirc^d(\top\, \mathcal{U}_\chi^u\, \mathbf{exc}))$	57.2	70	1,380	True
$\bigcirc^d(\bigcirc^d(\bigcirc^d(\top\, \mathcal{U}_\chi^u\, \mathbf{exc})))$	196.1	70	2,939	True
$\Box(\mathbf{call} \wedge \mathrm{p}_C \implies (\top\, \mathcal{U}_\chi^u\, \mathbf{exc} \wedge \chi_P^d\, \mathbf{han}))$	103.7	70	863	False
$\mathbf{call}\, \mathcal{U}_\chi^d\, (\mathbf{ret} \wedge \mathrm{p}_{Err})$	1.8	70	117	False
$\chi_F^d(\mathbf{call} \wedge ((\mathbf{call} \vee \mathbf{exc})\, \mathcal{S}_\chi^u\, \mathrm{p}_B))$	9.9	70	116	False
$\bigcirc^d(\bigcirc^d((\mathbf{call} \vee \mathbf{exc})\, \mathcal{U}_\chi^u\, \mathbf{ret}))$	6.2	70	116	False

To explore how the model checking implementation scales in practice with respect to the theoretical results, we explored the performances of the same MiniProc model with varying state-space sizes. In particular, two parameters affect the model:

- `M`, the length of the array to be sorted. We considered up to 7 elements in the array ($\mathtt{M} \in [2, 7]$).
- `K`, the number of bits to represent array values ($\mathtt{K} \in [1, 4]$).

Table 4. Results of additional experiments on the program of Fig. 17, but interpreted as a continuously running program. The program has been automatically translated into an ωOPBA with 44 reachable states. The abbreviations are: M.T. = Total Memory, M.MC = Memory for Model Checking only, O.O.M. = Out of memory.

Formula	Time (ms)	M.T. (MiB)	M.MC (KiB)	Results
$\chi_F^d\, \mathrm{p}_{Err}$	31.7	71	3,717	False
$\bigcirc^d(\bigcirc^d(\mathbf{call} \wedge \chi_F^u\, \mathbf{exc}))$	125.0	71	7,471	False
$\bigcirc^d(\mathbf{han} \wedge (\chi_F^d(\mathbf{exc} \wedge \chi_P^u\, \mathbf{call})))$	231.0	71	16,722	False
$\Box(\mathbf{exc} \implies \chi_P^u\, \mathbf{call})$	8.5	71	864	True
$\top\, \mathcal{U}_\chi^d\, \mathbf{exc}$	10.6	71	1,050	False
$\bigcirc^d(\bigcirc^d(\top\, \mathcal{U}_\chi^d\, \mathbf{exc}))$	23.0	71	1,590	False
$\Box((\mathbf{call} \wedge \mathrm{p}_A \wedge (\neg\mathbf{ret}\, \mathcal{U}_\chi^d\, \mathrm{WRx})) \implies \chi_F^u\, \mathbf{exc})$	39,307.0	2,540	862,164	True
$\bigcirc^d(\bigcirc^u\, \mathbf{call})$	2.1	71	156	False
$\bigcirc^d(\bigcirc^d(\bigcirc^d(\ominus^u\, \mathbf{call})))$	12.9	71	907	False
$\chi_F^d(\bigcirc^d(\ominus^u\, \mathbf{call}))$	46.2	72	2,682	False
$\Box((\mathbf{call} \wedge \mathrm{p}_A \wedge \mathit{CallThr}(\top)) \implies \mathit{CallThr}(\mathrm{e}_B))$	91,806.6	4,137	1,416,790	True
$\Diamond(\bigcirc_H^d\, \mathrm{p}_B)$	26.4	71	3,005	False
$\Diamond(\ominus_H^d\, \mathrm{p}_B)$	22.2	71	2,692	False
$\Diamond(\mathrm{p}_A \wedge (\mathbf{call}\, \mathcal{U}_H^d\, \mathrm{p}_C))$	3,794.6	490	227,858	False
$\Diamond(\mathrm{p}_C \wedge (\mathbf{call}\, \mathcal{S}_H^d\, \mathrm{p}_A))$	3,692.4	415	192,171	False
$\Box((\mathrm{p}_C \wedge \chi_F^u\, \mathbf{exc}) \implies (\neg\mathrm{p}_A\, \mathcal{S}_H^d\, \mathrm{p}_B))$	–	–	–	O.O.M.
$\Box(\mathbf{call} \wedge \mathrm{p}_B \implies \neg\mathrm{p}_C\, \mathcal{U}_H^u\, \mathrm{p}_{Err})$	142.1	71	11,833	True
$\Diamond(\bigcirc_H^u\, \mathrm{p}_{Err})$	5.2	71	167	False
$\Diamond(\ominus_H^u\, \mathrm{p}_{Err})$	14.3	71	992	False
$\Diamond(\mathrm{p}_A \wedge (\mathbf{call}\, \mathcal{U}_H^u\, \mathrm{p}_B))$	29.6	72	2,675	False
$\Diamond(\mathrm{p}_B \wedge (\mathbf{call}\, \mathcal{S}_H^u\, \mathrm{p}_A))$	72.8	72	5,043	False
$\Box(\mathbf{call} \implies \chi_F^d\, \mathbf{ret})$	58.5	72	4,215	False
$\Box(\mathbf{call} \implies \neg\bigcirc^u\, \mathbf{exc})$	6.9	71	116	True
$\Box(\mathbf{call} \wedge \mathrm{p}_A \implies \neg\mathit{CallThr}(\top))$	409.9	71	18,125	True
$\Box(\mathbf{exc} \implies \neg(\ominus^u(\mathbf{call} \wedge \mathrm{p}_A) \vee \chi_P^u(\mathbf{call} \wedge \mathrm{p}_A)))$	32.2	72	1,800	True
$\Box((\mathbf{call} \wedge \mathrm{p}_B \wedge (\mathbf{call}\, \mathcal{S}_\chi^d\, (\mathbf{call} \wedge \mathrm{p}_A))) \implies \mathit{CallThr}(\top)$	1,917.7	130	42,035	True
$\Box(\mathbf{han} \implies \chi_F^u\, \mathbf{ret})$	42.6	71	3,260	True
$\top\, \mathcal{U}_\chi^u\, \mathbf{exc}$	40.2	72	3,190	False
$\bigcirc^d(\bigcirc^d(\top\, \mathcal{U}_\chi^u\, \mathbf{exc}))$	260.5	72	11,556	False
$\bigcirc^d(\bigcirc^d(\bigcirc^d(\top\, \mathcal{U}_\chi^u\, \mathbf{exc})))$	826.6	94	40,479	False
$\Box(\mathbf{call} \wedge \mathrm{p}_C \implies (\top\, \mathcal{U}_\chi^u\, \mathbf{exc} \wedge \chi_P^d\, \mathbf{han}))$	937.7	71	27,683	False
$\mathbf{call}\, \mathcal{U}_\chi^d\, (\mathbf{ret} \wedge \mathrm{p}_{Err})$	25.9	71	2,555	False
$\chi_F^d(\mathbf{call} \wedge ((\mathbf{call} \vee \mathbf{exc})\, \mathcal{S}_\chi^u\, \mathrm{p}_B))$	179.8	71	10,056	False
$\bigcirc^d(\bigcirc^d((\mathbf{call} \vee \mathbf{exc})\, \mathcal{U}_\chi^u\, \mathbf{ret}))$	397.7	72	17,557	False

These two parameters determine the number of bits $\mathtt{G} = \mathtt{K} \cdot \mathtt{M}$ that the model needs for representing the global variables. In the model, array values are chosen nondeterministically before the first call to `qs()`, so that every possible combination is explored by the checker. Hence, the number of initial states is given by $2^{\mathtt{G}}$. As an example, for the model of Fig. 6 we have $(\mathtt{K}, \mathtt{M}) = (3, 4)$. Fig. 18 reports the MiniProc program for $\mathtt{M} = 4$ and $\mathtt{K} = 3$. With

respect to the buggy version, the Semisafe procedure keeps track of the elements equal to the pivot and does not call itself recursively on them.

The benchmark is equipped with ten properties (**Q.1**-**Q.10**), which we describe below. Their POTL formulation, as well as the satisfaction results are listed in Table 5. Two properties are usually of primary interest for such a program:

- The program terminates on any input array, and does it properly. This means that it always reaches a **ret** statement (Property **Q.1**) for the `main` procedure. To verify it, we impose the first position, which is the call of the `main` function, to be in the χ relation with the corresponding return[7].
- Any input array is correctly sorted at the end of the Quicksort procedure (Property **Q.2**). In order to verify it, we introduce the Boolean variable `sorted`. At every swap of values in the array, we check if the array is sorted in ascending order, and update accordingly the value for `sorted`. The verification of this property is somewhat decoupled from the previous one: **Q.2** does imply the termination of the program, but not a proper one, as the matching statement in the χ relation may either be a **ret** or an **exc** statement.

For our benchmark these simple properties are false, because `qs()` might throw other kinds of exceptions even after null references have been removed, so the program may terminate exceptionally even before the array is sorted. We therefore verify the following ones:

- **Q.3** and **Q.4** check whether the `main` and `qs()` procedures satisfy the *no-throw* guarantee, and POMC correctly finds out that they might be terminated abnormally by exceptions.
- **Q.5** verifies that the program can be terminated by an exception only if the second call to `qs()` throws one, which means that the array has been sanitized and the exception is not a `NullPointerException`. In the MiniProc model, `hasParsed` is an atomic proposition that is true only after the execution of the input-sanitizing procedure.
- **Q.6** verifies the property that the array is correctly sorted, hence in the right state, if the program is terminated by an exception. It is false because `qs()` might throw an exception before having finished sorting the array.
- **Q.7** is a stack-inspection property that verifies that whenever an exception is thrown, either there is a handler on the stack (so we are in the first call to `qs()`), or `parseList()` has already been called (so we are in the second call), and hence the exception is, again, not a `NullPointerException`.
- **Q.8** checks that the program always terminates, either properly or by an exception from the second call to `qs()`. It is also called *conditional* proper termination, since it states that the program terminates properly unless an exception is thrown after the input-sanitizing procedure.
- Similarly, **Q.9** checks that the only reason for the array not to be sorted when the program terminates is an exception thrown by the second call to `qs()` (the so-called *conditional* correctness).
- Finally, **Q.10** verifies **Q.8** and **Q.9** together, with a small variant that uses the $\diamond$ operator.

In total, our benchmark is composed of 240 experiments. Fig. 19, 20 and 21 report the performance results for a false formula, **Q.1**, and two valid ones, **Q.7** and **Q.8**. Because of space limits, the graphs for the remaining formulae are left to Appendix B. The graphs show the execution times for the model checking queries for each possible value of `G`. A timeout of one hour is used. All times are in seconds. Experiments that reach this threshold are interrupted, hence not reported in the graphs. Since some values of `G` may be induced by different combinations of `K` and `M`, each dot is also associated with the corresponding (`K`, `M`) pair. For example, for $\texttt{G} = 4$, there are two matching experiments: one for $(\texttt{K}, \texttt{M}) = (1, 4)$ and one for $(\texttt{K}, \texttt{M}) = (2, 2)$. All the experiments were run on a server with a 2.0 GHz AMD CPU and 500 GiB of RAM.

[7]Remember that, when termination is an issue, we must use ωOPBAs. Thus, this whole benchmark suite adopts ωOPBAs.

Table 5. Results of verification of the QuickSort benchmark.

#	Formula	Result
Q.1	$\chi_F^u(\mathbf{ret} \wedge \mathtt{main})$	False
Q.2	$\chi_F^u\,\mathtt{sorted}$	False
Q.3	$\Box((\mathbf{call} \wedge \mathtt{main}) \implies \neg(\bigcirc^u \mathbf{exc} \vee \chi_F^u \mathbf{exc}))$	False
Q.4	$\Box((\mathbf{call} \wedge \mathtt{qs}) \implies \neg(\bigcirc^u \mathbf{exc} \vee \chi_F^u \mathbf{exc}))$	False
Q.5	$(\bigcirc^u \mathbf{exc} \vee \chi_F^u \mathbf{exc}) \implies \bigcirc^u(\mathbf{exc} \wedge \mathtt{hasParsed}) \vee \chi_F^u(\mathbf{exc} \wedge \mathtt{hasParsed})$	True
Q.6	$(\bigcirc^u \mathbf{exc} \vee \chi_F^u \mathbf{exc}) \implies \bigcirc^u(\mathbf{exc} \wedge \mathtt{sorted}) \vee \chi_F^u(\mathbf{exc} \wedge \mathtt{sorted})$	False
Q.7	$\Box((\mathbf{call} \wedge \mathtt{accessValues}) \implies \mathtt{hasParsed} \vee (\top\, \mathcal{S}_\chi^d\, \mathbf{han}))$	True
Q.8	$\chi_F^u(\mathbf{ret} \wedge \mathtt{main}) \vee \chi_F^u(\mathbf{exc} \wedge \mathtt{hasParsed})$	True
Q.9	$\chi_F^u(\mathtt{sorted}) \vee \chi_F^u(\mathbf{exc} \wedge \mathtt{hasParsed})$	True
Q.10	$\Diamond(\mathbf{ret} \wedge \mathtt{main} \wedge \mathtt{sorted}) \vee \chi_F^u(\mathbf{exc} \wedge \mathtt{hasParsed})$	True

For all formulae, the experiments show a normal worst-case exponential behavior with respect to G. The highest value of G handled by POMC within the timeout is reached with a formula that actually holds, whereas typically finding a counterexample should occur earlier on average, as POMC features on-the-fly state exploration and early termination. This peak is 12, given by formula **Q.7**. 12 bits correspond to 4,096 initial states. However, the maximum values before the exponential blow up are on average smaller for valid formulae (7.6) with respect to false ones (8.8), in agreement with the exploration strategy.

When G can be given by different combinations of K and M, M is always dominating. This adheres to the theoretical worst-case complexity of the sorting algorithm $O(\mathtt{M}^2)$, given that the checker has to explore all the recursive calls for every possible initial state ($2^{\mathtt{G}}$), thus yielding an overall complexity of $O(2^{\mathtt{G}} \cdot \mathtt{M}^2)$. Unfortunately, there are low chances to share the search space between different initial states, as different global variables (the array to be sorted, which contains different values in different initial states) induce different search spaces. A partial sharing happens only for those pairs of initial states that correspond to the same set of array values, but with a different ordering. In such a case it may happen that, in the recursive calls, the checker finds itself exploring the sorting of a subarray in an already visited global state, thus aborting the exploration.

Some outliers may blur the inspection of the graphs: as an example, in Fig. 20 experiment $(1, 5)$ overcomes experiment $(3, 2)$. However, the exponential behavior can still be found in this case by considering that experiment $(2, 3)$, which corresponds to $\mathtt{G} = 6$ as well, while not present in the figure because of the timeout, requires more than 3,600 seconds.

Among the ones terminating within the blow up, the highest experiment in memory consumption allocated around 134 GiB. However, most of them do not require such a powerful device: the average amount of memory allocated is 27 GiB.

7.2.1 Iterated QuickSort. The aim of this section is to present a purely "omega" algorithm, that is, a routine that is intentionally meant to execute for an indefinite amount of time. Such routines are usually found in web interfaces or network protocols, where a server keeps listening on an input channel. When a connection is established, the interaction proceeds according to the protocol. However, if something goes wrong, the current session has to be interrupted, and the server gets back to the channel. In such cases, besides investigating the disruption's causes, it is crucial to establish that the disruption of a single session does not affect the operability of the server.

To show the suitability of POMC for the verification of such protocols, in this experiment we analyze another variant of the QuickSort benchmark. A routine continuously generates arrays by choosing their elements nondeterministically, and then sorts them with the correct `qs()` procedure on the generated array. If, however,

```
program:
bool sorted, hasParsed;
u3[4] a;

main() {
    a[0s4] = *;
    a[1s4] = *;
    a[2s4] = *;
    a[3s4] = *;
    sorted = false;
    hasParsed = false;
    try {
        qs(0s4, 3s4);
    } catch {
        hasParsed = true;
        qs(0s4, 3s4);
    }
}

swapElements(s4 swapLeft, s4 swapRight) {
    u3 tmp;

    accessValues();
    tmp = a[swapLeft];
    a[swapLeft] = a[swapRight];
    a[swapRight] = tmp;
    sorted = a[0s4] <= a[1s4]
              && a[1s4] <= a[2s4]
              && a[2s4] <= a[3s4];
}
```

```
qs(s4 left, s4 right) {
    s4 lo, hi, eq;
    u3 piv;

    if (left < right) {
        piv = a[right];
        eq = left;
        lo = eq;
        hi = right;
        while (lo <= hi) {
            if (a[hi] > piv) {
                hi = hi - 1s4;
            } else {
                swapElements(lo, hi);
                if (a[lo] < piv) {
                    swapElements(lo, eq);
                    eq = eq + 1s4;
                } else {}
                lo = lo + 1s4;
            }
        }
        qs(left, eq-1s4);
        qs(lo, right);
    } else {}
}
```

Fig. 18. The “Semisafe” Quicksort algorithm in MiniProc.

the stack size reaches a threshold value, and the allocation of a new frame is required, an exception is thrown, the call stack is emptied and a new array is generated.

We consider arrays of 3 elements, whose cells contain 3-bit unsigned integer values. `MAX_STACK`, the threshold, is set at 3. POTL formulas, verification results and execution times are reported in Table 6.

The main property we verify is that the routine never terminates. This can be achieved with different formulas. Namely, formula **IQ.1** checks that the call of the `main` procedure does not have a matching **ret** statement or an exception that aborts it. This represents a variant of the *no-throw* guarantee, previously introduced. Similarly, **IQ.2** checks that every thrown exception has a matching handler that catches it, thus preventing the pop of the initial call from the stack. **IQ.3** is a liveness property that requires that calls to the QuickSort procedure are endlessly pushed onto the stack.

Finally, to verify the QuickSort procedure, we introduce formula **IQ.4**. It expresses conditional correctness and conditional proper termination of procedure `qs()`. Its meaning is that every initial call of `qs()` on an array either terminates properly with a **ret** statement after having correctly sorted the array, or is interrupted by an exception due to having reached the maximum stack size (`maxReached`).

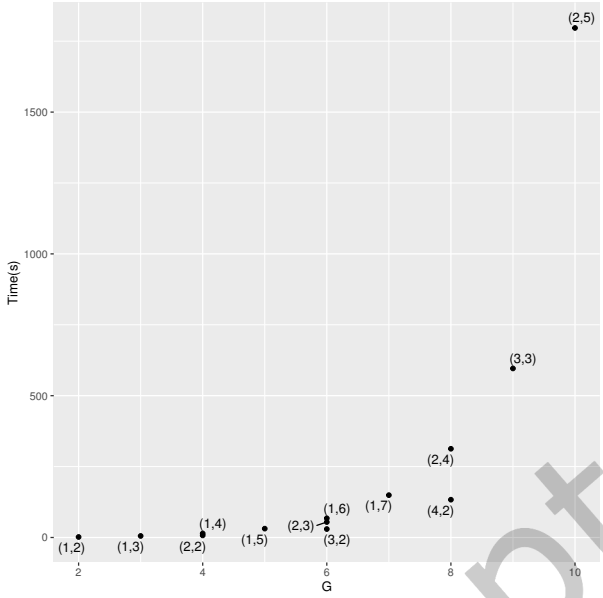


Fig. 19. Experimental results on property **Q.1**

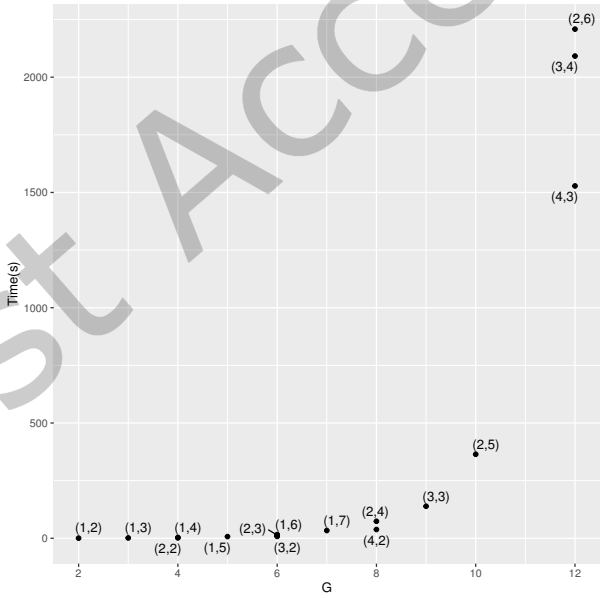


Fig. 20. Experimental results on property **Q.7**

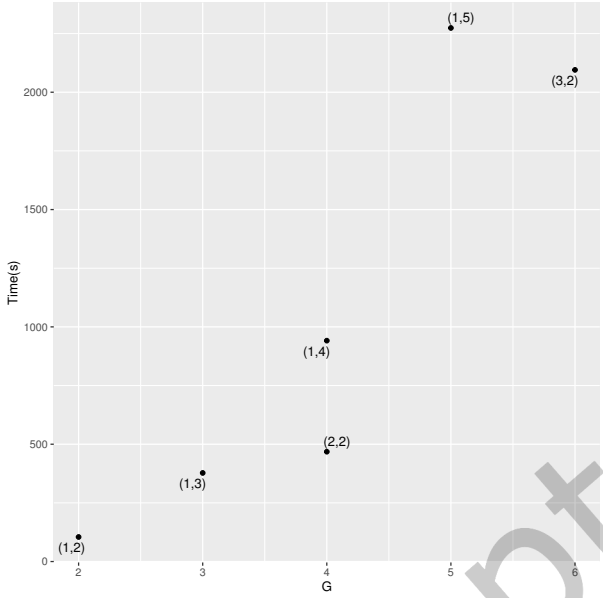


Fig. 21. Experimental results on property **Q.8**

Table 6. Iterated QuickSort Experiment.

#	Formula	Time (s)	Result
IQ.1	$\neg\, \chi_F^u((\mathbf{ret} \wedge \texttt{main}) \vee \mathbf{exc})$	148.19	True
IQ.2	$\Box(\mathbf{exc} \implies \chi_P^u\, \mathbf{han})$	59.78	True
IQ.3	$\Box(\Diamond(\mathbf{call} \wedge \texttt{qs}))$	77.26	True
IQ.4	$\Box((\mathbf{call} \wedge \texttt{qs} \wedge \ominus^d\, \mathbf{han}) \implies (\chi_F^u(\mathbf{ret} \wedge \texttt{qs} \wedge \texttt{sorted}) \vee \chi_F^u(\mathbf{exc} \wedge \texttt{maxReached})))$	> 3,600	True

7.2.2 Buggy Quicksort. For completeness, we show the analysis of the buggy variant performed by POMC. First, we consider the termination check experiment. Following the approach of [37, 74], we abstract away from the array content and just regard the local variables, thus generating a smaller model. With respect to Fig. 6, we replace the comparison `a[hi] > piv` with a nondeterministic choice (except for the first loop iteration, when we know that the outcome is false). The main model parameter is now `N`, the number of bits used to represent the local variables. An example with `N` = 3 is given in Fig. 22. The model does not have global variables, while it has 4 local variables: `left`, `right`, `lo`, `hi`. The values of `left` and `high` are chosen nondeterministically before the first call to `qs()`, hence the number of initial states is $2^{2\texttt{N}}$. The formula we verify is $\Diamond(\mathbf{ret} \wedge \texttt{main})$ (**BQ.1**), which is equivalent to the POTL formula $\chi_F^u(\mathbf{ret} \wedge \texttt{main})$, as there is a single initial call to the `main` procedure. The results, for different values of `N`, are reported in Table 7, left side, where a timeout of 1 h is used.

Secondly, we report on the verification of the model's correctness of sorting. In order to do so, we reintroduce the array values and consider the model of Fig. 6. The formula is $\chi_F^u\, \texttt{sorted}$ (**BQ.2**). As the first position is a call to the `main` procedure, and there are no exceptions in this model, the matching position in the χ relation can only be the corresponding **ret** statement. Then, the previous experiment implies the falsehood of this one. Results are reported in Table 7 (right side), with varying values of K and M.

```
program:

main() {
    u3 left, right;
    left  = *;
    right = *;
    qs(left, right);
}
```

```
qs(u3 left, u3 right) {
    u3 lo, hi;
    bool nondet;

    if (left < right) {
        lo = left; hi = right;
        while (lo <= hi) {
            if (*) { nondet = true; }
            else { nondet = false; }

            if (hi == right || nondet) {
                lo = lo + 1u3;
            } else {
                hi = hi - 1u3;
            }
        }
        qs(left, hi);
        qs(lo, right);
    }
}
```

Fig. 22. An abstract version of the "Buggy" QuickSort algorithm in MiniProc.

Table 7. Results of termination check (**BQ.1**, left) and sorting correctness (**BQ.2**, right) for the (abstracted) Buggy QuickSort.

N	Time (s)	Total memory (KiB)	Result
3	0.011	52,972	False
5	0.022	52,972	False
7	0.072	52,964	False
9	0.320	55,920	False
10	0.649	92,576	False

K	M	Time (s)	Total memory (KiB)	Result
1	2	0.167	58176	False
2	4	0.312	52,932	False
2	7	0.522	66,356	False
3	4	0.323	52,804	False
4	6	0.579	54,856	False

7.3 Related tools

Two well-established formalisms that model recursive programs are *Pushdown Systems (PDS's)* [18, 38] and *(Extended) Recursive State Machines (ERSMs)* [3]. Despite taking different perspectives, they have equivalent expressive power [3].

PDS's are supported by the model checker Moped [37]. Informally, they are Transition Systems equipped with a stack and a set of control locations, and allowing for nondeterministic branches. Variables can be local or global, and supported types are Boolean or bounded-integer, which is simulated with multiple Boolean variables. Arrays of variables are also allowed. PDS's are a lower-level formalism, essentially pushdown automata. Moped can verify on these models LTL properties, and perform reachability queries, implementing the algorithms presented in [34]. A second version of Moped has been developed with a friendlier interface, Remopla, and a Java front-end called jMoped. It features an automatic abstraction loop based on the CEGAR paradigm to target real world Java programs [35], but, unlike the first version, it performs only reachability checks. To cope with the state-space explosion problem, Moped enforces a symbolic representation through Boolean Decision Diagrams (BDDs).

ERSMs adopt a higher-level procedural approach. [3] presents an algorithm for Reachability analysis on ERSMs based on a fixpoint computation which is asymptotically slightly better than translating them into PDS's. To the

best of our knowledge, the only ERSM-based tool that has been developed is VERA [5], which supports only reachability and fair-cycle detection queries, given a monitor for an LTL property or a set of *target states*. In its implementation, it takes a more practical approach and adapts a nested DFS to the presence of summary transitions; moreover, instead of encoding the state space with BDDs, it uses an *on-the-fly* explicit-state automaton representation.

The SLAM toolkit [15] has obtained remarkable results in the verification of procedural programs, through reachability analysis [14]. To model C programs, it uses the formalism of *Boolean Programs*, which are supported by Moped and VERA, too. They are equipped with all the common imperative control-flow structures, as well as recursive procedures with call-by-value parameter passing, and a restricted form of control nondeterminism. In Boolean Programs all variables are of Boolean type. Procedures can return multiple values, and parallel assignment is allowed; however, there are no arrays. Similarly to the second version of Moped, it implements a cyclic abstraction refinement technique, where the first step is to generate an overapproximated model of the C program to be verified. If an error trace for a desired safety property is found to be spurious, this trace is used to refine the overapproximation, and to restart the cycle. In this cycle trace properties are verified by the reachability checker Bebop [13].

A spontaneous question is then how the performances of the above tools compare with each other and with POMC; in fact we purposely took inspiration for our benchmarks from the QuickSort algorithm which was introduced and adopted also to check other tools (i.e., MOPED, but VERA too targeted the buggy QuickSort program [5]). However, VERA is not available online. While the code base of Moped is publicly available[8], the project has not been maintained in recent years, and we did not manage to rerun the experiments on the same device we used for the MiniProc ones, because of the unavailability of some packages. SLAM, instead, is an industrial-level tool which is available within the Microsoft Static Driver Verifier Research Platform, but is highly specialized at checking the correct use of Windows APIs. We figure that it could be adapted to check properties expressible as reachability questions —but only those— in the QuickSort program, and in those cases we expect high-level performances of the tool.

Hence, we do not offer a comparative evaluation in terms of performances in the few cases where programs and specifying formulas could be compared; we just draw some qualitative findings.

Our tool, up to certain limits, is extremely fast in finding a counterexample for a formula that does not hold (with the exception regarding formula **Q.7** shown by Fig. 20). This is in line with the general implementation strategy: the *early termination* and *on-the-fly* properties are met. In this regard POMC shows a similar behavior as VERA, with which it shares the search space exploration approach. Moreover, we observe that the increase in complexity and number of automaton construction rules do not affect the performances when a simple LTL formula is verified, which was another non-trivial requirement for our implementation effort.

We emphasize, however, that the main comparison must be done, rather than in terms of performances on comparable benchmarks —if possible— in terms of the generality of verifiable properties.

None of these formalisms models exceptions and exception-handling constructs, and the only feature on this matter is the `assert` statement of Boolean Programs. More importantly, all tools examined here support (at most) the limited class of LTL specifications, which includes only some simple properties such as the termination guarantee. They cannot investigate the stack-based behavior of these programs, which is the key feature of POTL[9].

[8]http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/moped/

[9]The only reference to an implementation of CaRet model checking we could find is [69]. However, the tool downloadable from the author's website [67] does not seem to accept CaRet specifications.

8 CONCLUSIONS

To the best of our knowledge of the literature, OPLs are the largest family of CFLs that enjoys all algebraic, logic, and decidability properties needed to apply the classical model-checking schema; POTL, being FO-complete and therefore equivalent to aperiodic OPLs [60], is the most expressive temporal logic among the few ones that can express "nested properties" of tree-shaped programs; the object of this paper, the POMC model-checker is the only freely available one performing such a complete model-checking of pushdown automata against properties expressed in a "nested temporal logic".

Thus, on one hand this paper represents the conclusion of a long-standing research aimed at extending the classic properties connected to the finite-state formalism to a suitable subclass of pushdown automata; on the other hand we envision various types of practical tools for automatic verification exploiting the proof of concept obtained from the POMC prototype and the early experimentation with our benchmarks.

For instance, our approach based on on-the-fly state exploration may be counterproductive for applications where many "implementations" must be checked against one or few fixed specifications, possibly for comparative purposes: in such cases pre-computing the complete automaton for the negation of the POTL formula would leave the rest of the verification with a polynomial time complexity.

We plan to explore symbolic techniques that brought considerable performance improvements to model checking of "classical" temporal logics, such as bounded model checking [17] through SAT or SMT encodings, or methods based on tree-shaped tableaux (and encodings thereof) [41]. The recent proposal of exploiting the *antichain approach* to formal verification of OPL properties [48] is also promising.

Finally, it is intriguing to investigate variations of the POTL logic in a parallel way as Computation Tree Logic (CTL), CTL* and other logics for nondeterministic computation have been obtained as variations of LTL[10].

ACKNOWLEDGMENTS

We are grateful to an anonymous reviewer for their valuable suggestions.

This work has been partially funded by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT19018] within project ProbInG and by the EU Commission in the Horizon 2020 research and innovation programme under grant agreement No. 101000162 (PIACERE), and under grant agreement No. 101034440 (Marie Sklodowska-Curie Doctoral Network LogiCS@TU Wien).

REFERENCES

[1] David Abrahams. 1998. Exception-Safety in Generic Components. In *Generic Programming (LNCS, Vol. 1766)*. Springer, Berlin, Heidelberg, 69–79. https://doi.org/10.1007/3-540-39953-4_6

[2] Rajeev Alur, Marcelo Arenas, Pablo Barceló, Kousha Etessami, Neil Immerman, and Leonid Libkin. 2008. First-Order and Temporal Logics for Nested Words. *LMCS* 4, 4 (2008), 44 pages. https://doi.org/10.2168/LMCS-4(4:11)2008

[3] Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, and Mihalis Yannakakis. 2005. Analysis of Recursive State Machines. *ACM Trans. Program. Lang. Syst.* 27, 4 (2005), 786–818. https://doi.org/10.1145/1075382.1075387

[4] Rajeev Alur, Ahmed Bouajjani, and Javier Esparza. 2018. Model Checking Procedural Programs. In *Handbook of Model Checking*. Springer, 541–572. https://doi.org/10.1007/978-3-319-10575-8_17

[5] Rajeev Alur, Swarat Chaudhuri, Kousha Etessami, and Parthasarathy Madhusudan. 2005. On-the-Fly Reachability and Cycle Detection for Recursive State Machines. In *TACAS '05 (LNCS, Vol. 3440)*. Springer, Berlin, Heidelberg, 61–76. https://doi.org/10.1007/978-3-540-31980-1_5

[6] Rajeev Alur, Swarat Chaudhuri, and Parthasarathy Madhusudan. 2011. Software model checking using languages of nested trees. *ACM Trans. Program. Lang. Syst.* 33, 5 (2011), 15:1–15:45. https://doi.org/10.1145/2039346.2039347

[7] Rajeev Alur, Kousha Etessami, and P. Madhusudan. 2004. A Temporal Logic of Nested Calls and Returns. In *TACAS '04 (LNCS, Vol. 2988)*. Springer, Berlin, Heidelberg, 467–481. https://doi.org/10.1007/978-3-540-24730-2_35

[8] Rajeev Alur and Dana Fisman. 2016. Colored Nested Words. In *LATA 2016 (LNCS, Vol. 9618)*. Springer, 143–155. https://doi.org/10.1007/978-3-319-30000-9_11

[10] A few steps of such a path have been done with reference to the VPA formalism [6].

[9] Rajeev Alur and Parthasarathy Madhusudan. 2004. Visibly Pushdown Languages. In *STOC '04*. ACM, 202–211. https://doi.org/10.1145/1007352.1007390

[10] Rajeev Alur and Parthasarathy Madhusudan. 2009. Adding nesting structure to words. *J. ACM* 56, 3 (2009), 16:1–16:43. https://doi.org/10.1145/1516512.1516518

[11] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim Guldstrand Larsen, M. Oliver Möller, Paul Pettersson, Carsten Weise, and Wang Yi. 2000. UPPAAL - Now, Next, and Future. In *MOVEP'00 (LNCS, Vol. 2067)*. Springer, 99–124. https://doi.org/10.1007/3-540-45510-8_4

[12] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. MIT Press.

[13] Thomas Ball and Sriram K. Rajamani. 2000. Bebop: A Symbolic Model Checker for Boolean Programs. In *SPIN 2000 (LNCS, Vol. 1885)*. Springer, 113–130. https://doi.org/10.1007/10722468_7

[14] Thomas Ball and Sriram K. Rajamani. 2001. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN'01 (LNCS, Vol. 2057)*. Springer, 103–122. https://doi.org/10.1007/3-540-45139-0_7

[15] Thomas Ball and Sriram K. Rajamani. 2001. The SLAM Toolkit. In *CAV '01 (LNCS, Vol. 2102)*. Springer, 260–264. https://doi.org/10.1007/3-540-44585-4_25

[16] Alessandro Barenghi, Stefano Crespi Reghizzi, Dino Mandrioli, Federica Panella, and Matteo Pradella. 2015. Parallel parsing made practical. *Sci. Comput. Program.* 112 (2015), 195–226. https://doi.org/10.1016/j.scico.2015.09.002

[17] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *TACAS'99 (LNCS, Vol. 1579)*. Springer, 193–207. https://doi.org/10.1007/3-540-49059-0_14

[18] Ahmed Bouajjani, Javier Esparza, and Oded Maler. 1997. Reachability analysis of pushdown automata: application to model-checking. In *CONCUR '97 (LNCS, Vol. 1243)*. Springer, 135–150. https://doi.org/10.1007/3-540-63141-0_10

[19] Laura Bozzelli and César Sánchez. 2014. Visibly Linear Temporal Logic. In *Automated Reasoning*. Springer, 418–433. https://doi.org/10.1007/978-3-319-08587-6_33

[20] Olaf Burkart and Bernhard Steffen. 1999. Model Checking the Full Modal mu-Calculus for Infinite Sequential Processes. *Theor. Comput. Sci.* 221, 1-2 (1999), 251–270. https://doi.org/10.1016/S0304-3975(99)00034-1

[21] Razvan Certezeanu, Sophia Drossopoulou, Benjamin Egelund-Müller, K. Rustan M. Leino, Sinduran Sivarajan, and Mark J. Wheelhouse. 2016. Quicksort Revisited — Verifying Alternative Versions of Quicksort. In *Theory and Practice of Formal Methods (LNCS, Vol. 9660)*. Springer, 407–426.

[22] Swarat Chaudhuri and Rajeev Alur. 2007. Instrumenting C Programs with Nested Word Monitors. In *SPIN '07 (LNCS, Vol. 4595)*. Springer, 279–283. https://doi.org/10.1007/978-3-540-73370-6_20

[23] Michele Chiari, Davide Bergamaschi, and Francesco Pontiggia. 2021. POMC. https://github.com/michiari/POMC

[24] Michele Chiari, Dino Mandrioli, and Matteo Pradella. 2020. Operator precedence temporal logic and model checking. *Theor. Comput. Sci.* 848 (2020), 47–81. https://doi.org/10.1016/j.tcs.2020.08.034

[25] Michele Chiari, Dino Mandrioli, and Matteo Pradella. 2021. Model-Checking Structured Context-Free Languages. In *CAV '21 (LNCS, Vol. 12760)*. Springer, 387–410. https://doi.org/10.1007/978-3-030-81688-9_18

[26] Michele Chiari, Dino Mandrioli, and Matteo Pradella. 2021. Model-Checking Structured Context-Free Languages (Artifact). Zenodo. https://doi.org/10.5281/zenodo.4723740

[27] Michele Chiari, Dino Mandrioli, and Matteo Pradella. 2022. A First-Order Complete Temporal Logic for Structured Context-Free Languages. *Log. Methods Comput. Sci.* 18:3 (2022), 49 pages. https://doi.org/10.46298/LMCS-18(3:11)2022

[28] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *CAV'02 (LNCS, Vol. 2404)*. Springer, 359–364. https://doi.org/10.1007/3-540-45657-0_29

[29] Stefano Crespi Reghizzi and Dino Mandrioli. 2012. Operator Precedence and the Visibly Pushdown Property. *J. Comput. Syst. Sci.* 78, 6 (2012), 1837–1867. https://doi.org/10.1016/j.jcss.2011.12.006

[30] Stefano Crespi Reghizzi, Dino Mandrioli, and Daniel F. Martin. 1978. Algebraic Properties of Operator Precedence Languages. *Information and Control* 37, 2 (1978), 115–133. https://doi.org/10.1016/S0019-9958(78)90474-6

[31] Loris D'Antoni. 2014. A symbolic automata library. https://github.com/lorisdanto/symbolicautomata

[32] Koen De Bosschere. 1996. An Operator Precedence Parser for Standard Prolog Text. *Softw., Pract. Exper.* 26, 7 (1996), 763–779.

[33] Evan Driscoll, Aditya V. Thakur, and Thomas W. Reps. 2012. OpenNWA: A Nested-Word Automaton Library. In *CAV '12 (LNCS, Vol. 7358)*. Springer, 665–671.

[34] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. 2000. Efficient Algorithms for Model Checking Pushdown Systems. In *CAV 2000 (LNCS, Vol. 1855)*. Springer, 232–247. https://doi.org/10.1007/10722167_20

[35] Javier Esparza, Stefan Kiefer, and Stefan Schwoon. 2006. Abstraction Refinement with Craig Interpolation and Symbolic Pushdown Systems. In *TACAS'06 (LNCS, Vol. 3920)*. Springer, 489–503. https://doi.org/10.1007/11691372_35

[36] Javier Esparza, Antonín Kučera, and Stefan Schwoon. 2003. Model checking LTL with regular valuations for pushdown systems. *Information and Computation* 186, 2 (2003), 355–376. https://doi.org/10.1016/S0890-5401(03)00139-1

[37] Javier Esparza and Stefan Schwoon. 2001. A BDD-Based Model Checker for Recursive Programs. In *CAV '01 (LNCS, Vol. 2102)*. Springer, Berlin, Heidelberg, 324–336. https://doi.org/10.1007/3-540-44585-4_30
[38] Alain Finkel, Bernard Willems, and Pierre Wolper. 1997. A direct symbolic approach to model checking pushdown systems. In *Infinity '97 (ENTCS, Vol. 9)*. Elsevier, 27–37. https://doi.org/10.1016/S1571-0661(05)80426-8
[39] Robert W. Floyd. 1963. Syntactic Analysis and Operator Precedence. *J. ACM* 10, 3 (1963), 316–333. https://doi.org/10.1145/321172.321179
[40] Harold N. Gabow. 2000. Path-based depth-first search for strong and biconnected components. *Inf. Process. Lett.* 74, 3-4 (2000), 107–114. https://doi.org/10.1016/S0020-0190(00)00051-X
[41] Luca Geatti, Nicola Gigante, Angelo Montanari, and Mark Reynolds. 2021. One-pass and tree-shaped tableau systems for TPTL and $\mathrm{TPTL_b}$+Past. *Inf. Comput.* 278 (2021), 104599. https://doi.org/10.1016/j.ic.2020.104599
[42] Patrice Godefroid and Mihalis Yannakakis. 2013. Analysis of Boolean Programs. In *TACAS '13 (LNCS, Vol. 7795)*. Springer, 214–229. https://doi.org/10.1007/978-3-642-36742-7_16
[43] Dick Grune and Ceriel J. Jacobs. 2008. *Parsing techniques: a practical guide*. Springer, New York. 664 pages. https://doi.org/10.1007/978-0-387-68954-8
[44] Matthew Hague. 2013. Saturation of Concurrent Collapsible Pushdown Systems. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2013, December 12-14, 2013, Guwahati, India (LIPIcs, Vol. 24)*, Anil Seth and Nisheeth K. Vishnoi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 313–325. https://doi.org/10.4230/LIPIcs.FSTTCS.2013.313
[45] Jad Hamza, Nicolas Voirol, and Viktor Kunčak. 2019. System FR: Formalized Foundations for the Stainless Verifier. *PACMPL* 3, OOPSLA, Article 166 (oct 2019), 30 pages. https://doi.org/10.1145/3360592
[46] Michael A. Harrison. 1978. *Introduction to Formal Language Theory*. Addison Wesley, Boston, MA, USA.
[47] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2003. Software Verification with BLAST. In *SPIN 2003 (LNCS, Vol. 2648)*. Springer, 235–239. https://doi.org/10.1007/3-540-44829-2_17
[48] Thomas A. Henzinger, Pavol Kebis, Nicolas Mazzocchi, and N. Ege Saraç. 2023. Regular Methods for Operator Precedence Languages. In *ICALP'23*.
[49] Uschi Heuter. 1991. First-order properties of trees, star-free expressions, and aperiodicity. *ITA* 25 (1991), 125–145. https://doi.org/10.1051/ita/1991250201251
[50] Gerard J. Holzmann. 2004. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley.
[51] Thomas P. Jensen, Daniel Le Métayer, and Tommy Thorn. 1999. Verification of Control Flow based Security Properties. In *Proc. '99 IEEE Symp. Secur. Privacy*. IEEE Computer Society, Oakland, California, USA, 89–103. https://doi.org/10.1109/SECPRI.1999.766902
[52] Stefan Kiefer, Stefan Schwoon, and Dejvuth Suwimonteerabuth. 2010. Moped, version 1.0.16. http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/moped/
[53] David J. King and John Launchbury. 1995. Structuring Depth-First Search Algorithms in Haskell. In *POPL*. ACM Press, 344–354.
[54] Orna Kupferman. 2018. Automata Theory and Model Checking. In *Handbook of Model Checking*. Springer, 107–151. https://doi.org/10.1007/978-3-319-10575-8_4
[55] Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. 2002. Model Checking Linear Properties of Prefix-Recognizable Systems. In *CAV '02 (LNCS, Vol. 2404)*. Springer, 371–385. https://doi.org/10.1007/3-540-45657-0_31
[56] John Launchbury and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. In *PLDI'94*. ACM, 24–35. https://doi.org/10.1145/178243.178246
[57] Clemens Lautemann, Thomas Schwentick, and Denis Thérien. 1994. Logics For Context-Free Languages. In *CSL'94 (LNCS, Vol. 933)*. Springer, Berlin, Heidelberg, 205–216. https://doi.org/10.1007/BFb0022257
[58] Violetta Lonati, Dino Mandrioli, Federica Panella, and Matteo Pradella. 2015. Operator Precedence Languages: Their Automata-Theoretic and Logic Characterization. *SIAM J. Comput.* 44, 4 (2015), 1026–1088. https://doi.org/10.1137/140978818
[59] Dino Mandrioli and Matteo Pradella. 2018. Generalizing input-driven languages: Theoretical and practical benefits. *Computer Science Review* 27 (2018), 61–87. https://doi.org/10.1016/j.cosrev.2017.12.001
[60] Dino Mandrioli, Matteo Pradella, and Stefano Crespi Reghizzi. 2020. Aperiodicity, Star-freeness, and First-order Definability of Structured Context-Free Languages. *CoRR* abs/2006.01236 (2020), 53 pages. arXiv:2006.01236 https://arxiv.org/abs/2006.01236
[61] Simon Marlow. 2010. Haskell 2010 Language Report. https://www.haskell.org/onlinereport/haskell2010/
[62] Robert McNaughton. 1967. Parenthesis Grammars. *J. ACM* 14, 3 (1967), 490–500. https://doi.org/10.1145/321406.321411
[63] Robert McNaughton and Seymour Papert. 1971. *Counter-free Automata*. MIT Press, Cambridge, USA.
[64] Kurt Mehlhorn. 1980. Pebbling Mountain Ranges and its Application of DCFL-Recognition. In *ICALP '80 (LNCS, Vol. 85)*. 422–435. https://doi.org/10.1007/3-540-10003-2_89
[65] Bertrand Meyer. 2005. Attached Types and Their Application to Three Open Problems of Object-Oriented Programming. In *ECOOP'05 (LNCS, Vol. 3586)*. Springer, 1–32. https://doi.org/10.1007/11531142_1
[66] Ha Nguyen. 2006. Visibly Pushdown Automata Library.
[67] Huu-Vu Nguyen and Tayssir Touili. 2013. PoMMaDe – A PushdOwn Model-checker for MAlware DEtection. https://lipn.univ-paris13.fr/~touili/pommade/ Accessed on December 2nd, 2022.

[68] Huu-Vu Nguyen and Tayssir Touili. 2017. CARET model checking for malware detection. In *SPIN 2017*. ACM, 152–161. https://doi.org/10.1145/3092282.3092301
[69] Huu-Vu Nguyen and Tayssir Touili. 2017. CARET model checking for pushdown systems. In *SAC 2017*. ACM, 1393–1400. https://doi.org/10.1145/3019612.3019829
[70] Nir Piterman and Moshe Y. Vardi. 2004. Global Model-Checking of Infinite-State Systems. In *CAV '04 (LNCS, Vol. 3114)*. Springer, 387–400. https://doi.org/10.1007/978-3-540-27813-9_30
[71] Amir Pnueli. 1977. The Temporal Logic of Programs. In *FOCS '77*. IEEE Computer Society, 46–57. https://doi.org/10.1109/SFCS.1977.32
[72] Francesco Pontiggia. 2021. *POMC. A model checking tool for operator precedence languages on omega-words.* Master's thesis. Politecnico di Milano. http://hdl.handle.net/10589/176028
[73] Francesco Pontiggia, Michele Chiari, and Matteo Pradella. 2021. Verification of Programs with Exceptions Through Operator Precedence Automata. In *SEFM'21 (LNCS, Vol. 13085)*. Springer, Berlin, Heidelberg, 293–311. https://doi.org/10.1007/978-3-030-92124-8_17
[74] Stefan Schwoon. 2002. *Model checking pushdown systems.* Ph. D. Dissertation. Technical University Munich, Germany. http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/schwoon.html
[75] Herb Sutter. 1997. Exception-Safe Generic Containers. *C++ Report* 9 (1997). https://ptgmedia.pearsoncmg.com/imprint_downloads/informit/aw/meyerscddemo/DEMO/MAGAZINE/SU_FRAME.HTM
[76] Nguyen Van Tang and Hitoshi Ohsaki. 2011. Checking On-the-Fly Universality and Inclusion Problems of Visibly Pushdown Automata. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.* 94-A, 12 (2011), 2794–2801. https://doi.org/10.1587/transfun.E94.A.2794
[77] James W. Thatcher. 1967. Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. *Journ. of Comp. and Syst. Sc.* 1 (1967), 317–322. https://doi.org/10.1016/S0022-0000(67)80022-9
[78] Wolfgang Thomas. 1984. Logical Aspects in the Study of Tree Languages. In *CAAP'84*. Cambridge University Press, 31–50.
[79] Moshe Y. Vardi and Pierre Wolper. 1986. An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report). In *LICS '86*. IEEE Computer Society, 332–344.
[80] Igor Walukiewicz. 2001. Pushdown Processes: Games and Model-Checking. *Inf. Comput.* 164, 2 (2001), 234–263. https://doi.org/10.1006/inco.2000.2894

A OMITTED CORRECTNESS PROOFS OF MODEL CHECKING

Here we report the remaining lemmas and related proofs of the correctness of the model checking construction of Section 4.

A.1 Chain Next Operators

Lemma A.1 proves the correctness of $\mathscr{DR}$ rules for the $\chi_F^{\lessdot}$ operator, while the proof for $\chi_F^{\gtrdot}$ is omitted altogether, because it is very similar to the one for $\chi_F^{\doteq}$ in Lemma 4.2.

Lemma A.1 ($\chi_F^{\lessdot}$ operator). *Given a finite set of atomic propositions AP, an OP alphabet $(\mathcal{P}(AP), M_{AP})$, and a formula $\chi_F^{\lessdot}\psi$, let $\mathcal{A}_\varphi$ be the OPA built for a formula φ such that $\chi_F^{\lessdot}\psi \in \mathrm{Cl}(\varphi)$; and let $\mathcal{A}_{\varphi - \chi_F^{\lessdot}\psi}$ be the OPA built as $\mathcal{A}_\varphi$ but using $\mathscr{DR} \setminus \mathscr{DR}(\chi_F^{\lessdot}\psi)$ for δ.*
Inductive assumption: *in all accepting computations of $\mathcal{A}_\varphi$ for each position i in the input word w and for each sub-formula $\psi' \in \mathrm{ssubf}(\chi_F^{\lessdot}\psi)$ we have $(w,i) \models \psi'$ iff $\psi' \in \Phi_c^g(i)$.*
***Inductive claim** I:* $[I_1]$ *A computation ρ of $\mathcal{A}_\varphi$ is accepting if and only if $[I_2]$ ρ is accepting for $\mathcal{A}_{\varphi - \chi_F^{\lessdot}\psi}$ and for each position i in the input word w we have $(w,i) \models \chi_F^{\lessdot}\psi$ iff $\chi_F^{\lessdot}\psi \in \Phi_c^g(i)$.*

Proof. We prove two **auxiliary claims**, based on the following assertions:

- let $[A_1]$ be: $(w,i) \models \chi_F^{\lessdot}\psi$;
- let $[A_2]$ be: all accepting computations of $\mathcal{A}_\varphi$ bring it from configuration $\langle yz, \Phi^g(i), \alpha\gamma\rangle$ with $\chi_F^{\lessdot}\psi \in \Phi_c^g(i)$ to a configuration $\langle z, \Phi^g(i_z), \alpha'\gamma\rangle$ such that $\chi_F^{\lessdot}\psi \notin \Phi_p^g(i_z)$, $|\alpha| = 1$ and $|\alpha'| = 1$ if $\mathrm{first}(y)$ is read by a shift move, $|\alpha'| = 2$ if it is read by a push move.

We prove that for any word $w = \#xyz\#$ and positions $i = |x| + 1$, $i_z = |xy| + 1$ in w, $A_1 \iff A_2$.

$[A_1 \implies A_2]$ Suppose $\chi_F^{\lessdot}\psi$ holds in position i, labeled a. Then, a must be the left context of more than one chain,[11] and the word being read must have one of the structures of Fig. 12, with $n \geq 1$. Let us call b_p, $1 \leq p \leq n$, the right contexts of those chains that are s.t. $a \lessdot b_p$ (i.e., all except the rightmost context of i). There exists an index q, $1 \leq q \leq n$, such that ψ holds in i_{b_q}, the word position labeled with b_q, and it does not hold in any other $i_{b_{q'}}$, for $q < q' \leq n$.

All accepting computations reach a configuration $\langle a \dots z, \Phi^g(i), [f, \Phi^g(k)]\gamma\rangle$, where $k < i$ and $\alpha = [f, \Phi^g(k)]$, and $\chi_F^{\lessdot}\psi \in \Phi_c^g(i)$, because the OPA guesses that $\chi_F^{\lessdot}\psi$ holds in i. Symbol a is read by a shift or a push transition, which leads the OPA to configuration $\langle c_0^0 \dots z, \Phi^g(i_{c_0^0}), \mu'\rangle$, with $\mu' = \alpha'\gamma$ (as in claim A_2), and either $\alpha' = [a, \Phi^g(i)][f, \Phi^g(k)]$ or $\alpha' = [a, \Phi^g(k)]$, respectively. Due to rule (9), we have $\chi_F^{\lessdot}\psi \in \Phi_p^g(i_{c_0^0})$ and $\zeta_L \in \Phi_p^g(i_{c_0^0})$. As a result, the next move must be a push, consistently with the hypothesis implying that a is the left context of a chain. Then, starting with c_0^0, the OPA reads the body of the innermost chain whose left context is a, until it reaches its right context b_1. In this process, the topmost stack symbol $[c_0^0, \Phi^g(i_{c_0^0})]$ may be updated by shift transitions reading other terminals c_p^0, $1 \leq p \leq m_0$, that are part of the same simple chain as c_0^0. However, it is not popped until b_1 is reached, since subchains cause the OPA to only push, pop and update new stack symbols, but not existing ones. So, the OPA reaches configuration $\langle b_1 \dots z, \Phi(i_{b_1}), [c_{m_0}^0, \Phi^g(i_{c_0^0})]\mu'\rangle$, with $\chi_F^{\lessdot}\psi \in \Phi_p^g(i_{c_0^0})$.

Suppose, first, $q > 1$, and that ψ does not hold in b_1. Since $c_{m_0}^0 \gtrdot b_1$, the next transition is a pop. Due to rule (10), it leads the OPA to configuration $\langle b_1 \dots z, \Phi'(i_{b_1}), \mu'\rangle$ with $\chi_F^{\lessdot}\psi \in \Phi'_p(i_{b_1})$ and $\zeta_L \in \Phi'_p(i_{b_1})$. If, instead, ψ holds in b_1, the OPA guesses that it will also hold in a future position i_{b_p} (possibly $p = q$), and puts $\chi_F^{\lessdot}\psi \in \Phi'_p(i_{b_1})$ anyways. The presence of ζ_L implies the next move is a push, a requirement that is satisfied because $a \lessdot b_1$. So, the OPA transitions to configuration $\langle v_0^1 \dots z, \Phi(i_{v_0^1}), [b_1, \Phi'(i_{b_1})]\mu'\rangle$. The computation, then, goes on in the same way for each b_p, $1 \leq p < q$. Before b_q is read, (and possibly $q = 1$), the OPA is in configuration $\langle b_q \dots z, \Phi(i_{b_q}),$

[11] By property 4 of the χ relation, a chain with contexts in the $\lessdot$ relation must be embedded in a composed chain with contexts in another PR.

$[c^{q-1}_{m_{q-1}}, \Phi^g(i_{b_{q-1}})]\mu'\rangle$, with $\chi^{\lessdot}_F \psi \in \Phi^g_p(i_{b_{q-1}})$. Since $c^{q-1}_{m_{q-1}} \gtrdot b_q$, a pop transition brings the OPA to $\langle b_q \dots z, \Phi'(i_{b_q}), \mu'\rangle$. Since by hypothesis $\psi \in \Phi_c(i_{b_q})$, by rule (10) we just have $\zeta_L \in \Phi'_p(i_{b_q})$, and the initial guess is verified. Since the topmost stack symbol contains a, and $a \lessdot b_q$, the next transition is a push, which satisfies the requirement of ζ_L. Note that $\chi^{\lessdot}_F \psi \notin \Phi'_p(i_{b_q})$, and the stack is μ', which satisfies A_2.

$[A_2 \implies A_1]$ Suppose that during an accepting computation the OPA reaches configuration $\langle a \dots z, \Phi(i), [f, \Phi^g(k)]\gamma\rangle$, with $k < i$ and $\chi^{\lessdot}_F \psi \in \Phi_c(i)$. Again, a must be read by either a push or a shift move. Since ζ_L is inserted as a pending requirement into the state resulting from this move, the next transition must be a push, so a is the left context of at least a chain. This chain has the form of Fig. 12. By rule (9), the OPA reaches configuration $\langle c^0_0 \dots z, \Phi^g(i_{c^0_0}), \delta\rangle$, with $\chi^{\lessdot}_F \psi, \zeta_L \in \Phi^g_p(i_{c^0_0})$, and μ' as in the $[A_1 \implies A_2]$ part after reading a. The stack symbol pushed while reading c^0_0 is $[c^0_0, \Phi^g(i_{c^0_0})]$. The stack size is now greater by one w.r.t. what is required by assertion A_2, so $[c^0_0, \Phi^g(i_{c^0_0})]$ must be popped.

This happens when the OPA reaches a symbol e s.t. the terminal symbol in the topmost stack symbol takes precedence over e. We claim that e must be s.t. $a \lessdot e$ and, according to the notation of Fig. 12, $e = b_1$. Suppose by contradiction that, on the contrary, $a \gtrdot e$ or $a \doteq e$ (so $e = d$ in Fig. 12, in which $n = 0$ and $c^0_{m_0}$ precedes d). In this case, after popping $[c^0_{m_0}, \Phi^g(i_{c^0_0})]$, the automaton reaches configuration $\langle dz, \Phi'(i_d), \mu''\rangle$. Since $\chi^{\lessdot}_F \psi \in \Phi^g_p(i_{c^0_0})$, by rule (10) we have $\chi^{\lessdot}_F \psi \in \Phi'_p(i_d)$, so this configuration does not satisfy the thesis statement. Moreover, $\zeta_L \in \Phi'_p(i_d)$, which requires the next transition to be a push. But $a \doteq d$ or $a \gtrdot d$, and a is the topmost stack symbol, so such a computation is blocked by rule (4), never reaching a configuration complying with the thesis statement.

So, $e = b_1$, and the OPA reaches configuration $\langle b_1 \dots z, \Phi(i_{b_1}), [c^0_{m_0}, \Phi^g(i_{c^0_0})]\mu'\rangle$. The subsequent pop move leads to $\langle b_1 \dots z, \Phi'(i_{b_1}), \mu'\rangle$.

Suppose $\psi \in \Phi_c(i_{b_1})$. Then, by rule (10) we only have $\zeta_L \in \Phi'_p(i_{b_1})$, and $\chi^{\lessdot}_F \psi \notin \Phi'_p(i_{b_1})$. This configuration satisfies claim A_2, and since $a \lessdot b_1$, a and b_1 are the context of a chain, and ψ holds in b_1, we can conclude that $\chi^{\lessdot}_F \psi$ holds in a.

Otherwise, if $\psi \notin \Phi_c(i_{b_1})$, by rule (10) we have $\chi^{\lessdot}_F \psi, \zeta_L \in \Phi'_p(i_{b_1})$. The next transition will therefore push $[b_1, \Phi'(i_{b_1})]$ onto the stack, again with $\chi^{\lessdot}_F \psi$ as a pending obligation in it. Then, the same argument done with $[c^0_0, \Phi^g(i_{c^0_0})]$ (and its subsequent updates) can be repeated. The only way the target configuration of claim A_2 can be reached is by reading a position b_q, s.t. $a \lessdot b_q$, the terminal in the topmost stack symbol takes precedence from b_q (so a and b_q are the context of a chain), and $\psi \in \Phi_c(i_{b_q})$, so ψ holds in b_q. This implies $\chi^{\lessdot}_F \psi$ holds in a.

$[I_1 \implies I_2]$ follows from $A_1 \implies A_2$ and $\mathcal{A}_{\varphi - \chi^{\doteq}_F \psi}$'s $\mathcal{DR}$ rules being a strict subset of $\mathcal{A}_\varphi$'s. $[I_2 \implies I_1]$ again follows from $A_2 \implies A_1$, and the fact that $\Phi^g(i_z)$ may not contain $\chi^{\lessdot}_F \psi$, nor do states in α', so rules (9) and (10) may not prevent the computation from reaching a final state. □

A.2 Chain Back Operators

Now, we prove the correctness of rules given in Section 4.1.3 for the $\chi^{\doteq}_P$ and $\chi^{\gtrdot}_P$ operators. The proof for $\chi^{\lessdot}_P$ is very similar to the one for $\chi^{\doteq}_P$ and is therefore omitted. The proof for $\chi^{\doteq}_P$ uses, again, the left tree of Fig. 12, whereas the one for $\chi^{\lessdot}_P$ would use both trees. The proof for $\chi^{\gtrdot}_P$, instead, uses Fig. 23, which represents many-to-one chains, with the outermost one expanded on the rightmost non-terminal in its body.

Lemma A.2 ($\chi^{\doteq}_P$ operator). *Given a finite set of atomic propositions AP, an OP alphabet $(\mathcal{P}(AP), M_{AP})$, and a formula $\chi^{\doteq}_P \psi$, let $\mathcal{A}_\varphi$ be the OPA built for a formula φ such that $\chi^{\doteq}_P \psi \in \mathrm{Cl}(\varphi)$; and let $\mathcal{A}_{\varphi - \chi^{\doteq}_P \psi}$ be the OPA built as $\mathcal{A}_\varphi$ but using $\mathcal{DR} \setminus \mathcal{DR}(\chi^{\doteq}_P \psi)$ for δ.*
Inductive assumption: *in all accepting computations of $\mathcal{A}_\varphi$ for each position i in the input word w and for each sub-formula $\psi' \in \mathrm{ssubf}(\chi^{\doteq}_P \psi)$ we have $(w, i) \models \psi'$ iff $\psi' \in \Phi^g_c(i)$.*

Inductive claim I: $[I_1]$ *A computation ρ of $\mathcal{A}_\varphi$ is accepting if and only if* $[I_2]$ *ρ is accepting for $\mathcal{A}_{\varphi - \chi_P^{\doteq} \psi}$ and for each position j in the input word w we have $(w, j) \models \chi_P^{\doteq} \psi$ iff $\chi_P^{\doteq} \psi \in \Phi_c^g(j)$.*

Proof. We first prove an **auxiliary claim** based on the following assertions:

- let $[A_1]$ be: $(w, j) \models \chi_P^{\doteq} \psi$;
- let $[A_2]$ be: all accepting computations of $\mathcal{A}_\varphi$ bring it from configuration $\langle yz, \Phi^g(i), \alpha\gamma\rangle$ to a configuration $\langle z, \Phi^g(i_z), \beta\gamma\rangle$ such that $|\alpha| = 1$, $|\beta| = 1$ if first(y) is read by a shift move, $|\beta| = 2$ if it is read by a push move, and $\chi_P^{\doteq} \psi \in \Phi_c^g(j)$.

We prove that for any word $w = \#uz\#$ and position $j = |u|$ in w there exists a partition of $u = xy$, with y not empty and $|x| = i - 1 > 0$, such that $A_1 \iff A_2$.

$[A_1 \Rightarrow A_2]$ Suppose $\chi_P^{\doteq} \psi$ holds in position j, labeled with d. Then, there exists a position i, labeled with a, s.t. $\chi(i, j)$, $a \doteq d$, and ψ holds in i. Since a and d are the contexts of a chain, w must have the form of Fig. 12 (left). All accepting computations of the OPA reach configuration $\langle a \dots z, \Phi^g(i), [f, \Phi^g(k)]\gamma\rangle$ with $k < i$ before reading a. By the assumption on $\mathcal{A}_\varphi$, we have $\psi \in \Phi_c^g(i)$. a is read by a shift or a push move, bringing the OPA to $\langle c_0^0 \dots z$, $\Phi^g(i_{c_0^0}), \mu\rangle$, with $\mu = \alpha'\gamma$, and either $\alpha' = [a, \Phi^g(i)][f, \Phi^g(k)]$ or $\alpha' = [a, \Phi^g(k)]$, respectively. Due to rule (19), we have $\chi_P^{\doteq} \psi \in \Phi_p^g(i_{c_0^0})$. After reading c_0^0, the OPA reaches configuration $\langle v_0^0 \dots z, \Phi(i_{v_0^0}), [c_0^0, \Phi^g(i_{c_0^0})]\mu\rangle$. Then, the automaton proceeds to read the rest of the body delimited by relation $\chi(i, j)$. If i is the left context of multiple chains, the stack symbol $[c_0^0, \Phi^g(i_{c_0^0})]$, containing $\chi_P^{\doteq} \psi$ as a pending obligation, is popped before reaching d. Let b_p, $1 \le p \le n$, be all labels of positions i_{b_p} s.t. $\chi(i, i_{b_p})$ and $a \lessdot b_p$. It can be proved inductively that, before reading any of such positions, the OPA is in a configuration $\langle b_p \dots z, \Phi(i_{b_p}), [c_{m_{p-1}}^{p-1}, \Phi^g(i_{b_{p-1}})]\mu\rangle$, with $\chi_P^{\doteq} \psi \in \Phi_p^g(i_{b_{p-1}})$. Since $c_{m_{p-1}}^{p-1} \gtrdot b_p$, the next move is a pop, leading to a configuration $\langle b_p \dots z, \Phi'(i_{b_p}), \mu\rangle$, with $\chi_P^{\doteq} \psi \in \Phi'_p(i_{b_p})$, due to rule (18). Then, b_p is read by a push move because $a \lessdot b_p$, so $\chi_P^{\doteq} \psi$ is again stored in the topmost stack symbol as a pending obligation, in a configuration $\langle v_0^p \dots z, \Phi(i_{v_1^p}), [b_p, \Phi'(i_{b_p})]\mu\rangle$. The stack symbol containing $\chi_P^{\doteq} \psi$ is only popped in positions b_p, or when reaching d, since subchains only cause the OPA to push and pop new symbols.

So, configuration $\langle dz, \Phi(j), [c_{m_n}^n, \Phi'(i_{b_p})]\mu\rangle$ is reached, with $\chi_P^{\doteq} \psi \in \Phi'_p(i_{b_p})$ (recall that d labels j, the last position of y). Due to rule (18), a pop move leads the OPA to $\langle dz, \Phi'(j), \mu\rangle$, with $\chi_P^{\doteq} \psi \in \Phi'_p(j)$. Then, since by hypothesis $a \doteq d$, and a is contained in the topmost stack symbol, d is read by a shift move. Since this transition is preceded by a pop, we have $\zeta_R \in \Phi'_p(j)$. So, by rule (16), since $\chi_P^{\doteq} \psi, \zeta_R \in \Phi'_p(j)$, we have $\chi_P^{\doteq} \psi \in \Phi'_c(j)$. Finally, the shift move reads d and leads the OPA to $\langle z, \Phi(i_z), \beta\gamma\rangle$, with either $\beta = [d, \Phi^g(i)][f, \Phi^g(k)]$ or $\beta = [d, \Phi^g(k)]$, satisfying claim A_1.

$[A_2 \Rightarrow A_1]$ Suppose that, while reading w, an accepting computation of the OPA reaches a configuration $\langle dz$, $\Phi^g(j), \mu\rangle$, where j is the last position of y, labeled with d, and $\chi_P^{\doteq} \psi \in \Phi_c^g(j)$. By rule (16), we have $\chi_P^{\doteq} \psi, \zeta_R \in \Phi_p^g(j)$. The presence of ζ_R in $\Phi_p^g(j)$ requires the previous transition to be a pop, so d is the right context of a chain. Let a, in position i, be its left context. By hypothesis, the computation proceeds reading d, and by rule (17) it must be read by a shift transition. So, we have $a \doteq d$, and w must be of the form of Fig. 12 (left). Going back to $\langle dz, \Phi^g(j)$, $\mu\rangle$, consider the pop move leading to this configuration. It starts from configuration $\langle dz, \Phi(j), [c_{m_n}^n, \Phi^g(i_{b_n})]\mu\rangle$, and by rule (18) we have $\chi_P^{\doteq} \psi \in \Phi_p^g(i_{b_n})$.

Consider the move that pushed $\Phi^g(i_{b_n})$ onto the stack. Suppose it was preceded by a pop move. Since $\Phi^g(i_{b_n})$ is the target state of this transition, and $\chi_P^{\doteq} \psi \in \Phi_p^g(i_{b_n})$, by rule (18) $\chi_P^{\doteq} \psi$ must be contained as a pending obligation in the popped state as well. So, this obligation is propagated backwards every time the automaton encounters a position that is the left context of a chain, i.e. positions b_p, $1 \le p \le n$, in Fig. 12. In order to stop the propagation, a push of a state with $\chi_P^{\doteq} \psi$ as a pending obligation, preceded by another push or shift move must be encountered. Such a transition pushes or updates the stack symbol under the one containing $\chi_P^{\doteq} \psi$, which means the left context

a s.t. $a \doteq d$ of a chain whose right context is d has been reached. In both cases, the target state of the push/shift transitions contains $\chi_P^{\doteq}\,\psi$ as a pending obligation, so by rule (19) we have $\psi \in \Phi_c^g(i)$. Hence, by the inductive assumption, ψ holds in position i (corresponding to a), we have $i \doteq j$ and $\chi(i, j)$, which implies $\chi_P^{\doteq}\,\psi$ holds in j.

$[I_1 \Rightarrow I_2]$ follows from $A_1 \Rightarrow A_2$ and $\mathcal{A}_{\varphi - \chi_P^{\doteq}\psi}$'s $\mathcal{DR}$ rules being a strict subset of $\mathcal{A}_\varphi$'s. $[I_2 \Rightarrow I_1]$ again follows from $A_2 \Rightarrow A_1$ and the fact that, after j is read, the next state $\Phi^g(i_z)$ may not contain $\chi_P^{\doteq}\,\psi$ (unless ψ holds in j, which however cannot be required by $\chi_P^{\doteq}\,\psi$ holding in j), so rules 16-19 may not prevent the computation from reaching a final state. □

Lemma A.3 ($\chi_P^{>}$ operator). *Given a finite set of atomic propositions AP, an OP alphabet $(\mathcal{P}(AP), M_{AP})$, and a formula $\chi_P^{>}\,\psi$, let $\mathcal{A}_\varphi$ be the OPA built for a formula φ such that $\chi_P^{>}\,\psi \in \mathrm{Cl}(\varphi)$; and let $\mathcal{A}_{\varphi - \chi_P^{>}\psi}$ be the OPA built as $\mathcal{A}_\varphi$ but using $\mathcal{DR} \setminus \mathcal{DR}(\chi_P^{>}\,\psi)$ for δ.*

Inductive assumption: *in all accepting computations of $\mathcal{A}_\varphi$ for each position i in the input word w and for each sub-formula $\psi' \in \mathrm{ssubf}(\chi_P^{>}\,\psi)$ we have $(w, i) \models \psi'$ iff $\psi' \in \Phi_c^g(i)$.*

***Inductive claim** I:* $[I_1]$ *A computation ρ of $\mathcal{A}_\varphi$ is accepting if and only if $[I_2]$ ρ is accepting for $\mathcal{A}_{\varphi - \chi_P^{>}\psi}$ and for each position j in the input word w we have $(w, j) \models \chi_P^{>}\,\psi$ iff $\chi_P^{>}\,\psi \in \Phi_c^g(j)$.*

Proof. We first prove an **auxiliary claim** based on the following assertions:

- let $[A_1]$ be: $(w, j) \models \chi_P^{>}\,\psi$;
- let $[A_2]$ be: all accepting computations of $\mathcal{A}_\varphi$ bring it from configuration $\langle yz, \Phi^g(i), \alpha\gamma\rangle$ to a configuration $\langle z, \Phi^g(i_z), \beta\gamma\rangle$ such that $|\alpha| = 1$, $|\beta| = 1$ if $\mathrm{first}(y)$ is read by a shift move, $|\beta| = 2$ if it is read by a push move, and $\chi_P^{>}\,\psi \in \Phi_c^g(j)$, where j is the last position of y.

We prove that for any word $w = \#xyz\#$ and position $j = |xy|$ in w, $A_1 \iff A_2$, where $i = |x| + 1$.

$[A_1 \Rightarrow A_2]$ Suppose $\chi_P^{>}\,\psi$ holds in position j. Then, j is the right context of at least two chains, and the word w has one of the structures of Fig. 23, with i being the leftmost context of j. Let positions i_{b_p}, labeled with b_p, $1 \le p \le n$, be all other left contexts of chains sharing j as their right context. There exists an index q, $1 \le q \le n$, s.t. ψ holds in i_{b_q}.

During an accepting run, the OPA reads w normally, until it reaches b_q, with configuration

$$\langle b_q \dots z, \Phi(i_{b_q}), [c_{m_q}^q, \Phi^g(i_{c_0^q})][b_{q+1}, \Phi^g(i_{c_0^{q+1}})] \dots \mu\rangle,$$

with $\psi \in \Phi_c(i_{b_q})$, $\mu = [a, \Phi^g(k)]\gamma$, with $k \le i$ depending on whether a (the label of i) was read by a push or a shift move. Note that if b_q is the only character in its simple chain body ($u_q = \varepsilon$ in Fig. 23), then $[c_{m_q}^q, \Phi^g(i_{c_0^q})]$ is not present on the stack. In this case, b_q is read by a push move instead of a shift. Suppose b_q is the left context of one or more chains, besides the one whose right context is j. In Fig. 23, this means $v_0^{q-1} \neq \varepsilon$. Consider the right context of the outermost of such chains: assume, w.l.o.g., that it is c_0^{q-1} (it may as well be b_{q-1}). Since ψ holds in i_{b_q}, $\chi_P^{\lessdot}\,\psi$ holds in $i_{c_0^{q-1}}$. If, instead, $v_0^{q-1} = \varepsilon$, then c_0^{q-1} is the successor of b_q, and $\ominus^d\,\psi$ holds in it. In both cases, $\chi_P^{\lessdot}\,\psi \vee \ominus^d\,\psi$ holds in $i_{c_0^{q-1}}$. Since $b_q \lessdot c_0^{q-1}$, the latter is read by a push transition, pushing stack symbol $[c_0^{q-1}, \Phi^g(i_{c_0^{q-1}})]$, with $\chi_P^{\lessdot}\,\psi \vee \ominus^d\,\psi \in \Phi_c^g(i_{c_0^{q-1}})$. This symbol remains on stack until d is reached, although its terminal symbol may be updated. The computation then proceeds normally, until configuration $\langle dz, \Phi^{(q-1)}(j), [b_{q-1}, \Phi^g(i_{c_0^{q-1}})] \dots \mu\rangle$ is reached.

Since $\chi_P^{\lessdot}\,\psi \vee \ominus^d\,\psi \in \Phi_c^g(i_{c_0^{q-1}})$, by rule (29), the OPA transitions to configuration $\langle dz, \Phi^{(q)}(j), [b_q, \Phi^g(i_{c_0^q})] \dots \mu\rangle$ with $\chi_P^{>}\,\psi \in \Phi_p^{(q)}(j)$ and $\zeta_L, \zeta_{\doteq} \notin \Phi_p^{(q)}(j)$. (Note that the next transition must be a pop, since the topmost stack symbol is b_q, and $b_q \gtrdot d$.) Then, by rule (29), all subsequent pop transitions propagate $\chi_P^{>}\,\psi$ as a pending obligation in the OPA state, until configuration $\langle dz, \Phi^{(n-1)}(j), \mu\rangle$, with $\chi_P^{>}\,\psi \in \Phi_p^{(n-1)}(j)$. Now, the automaton guesses that

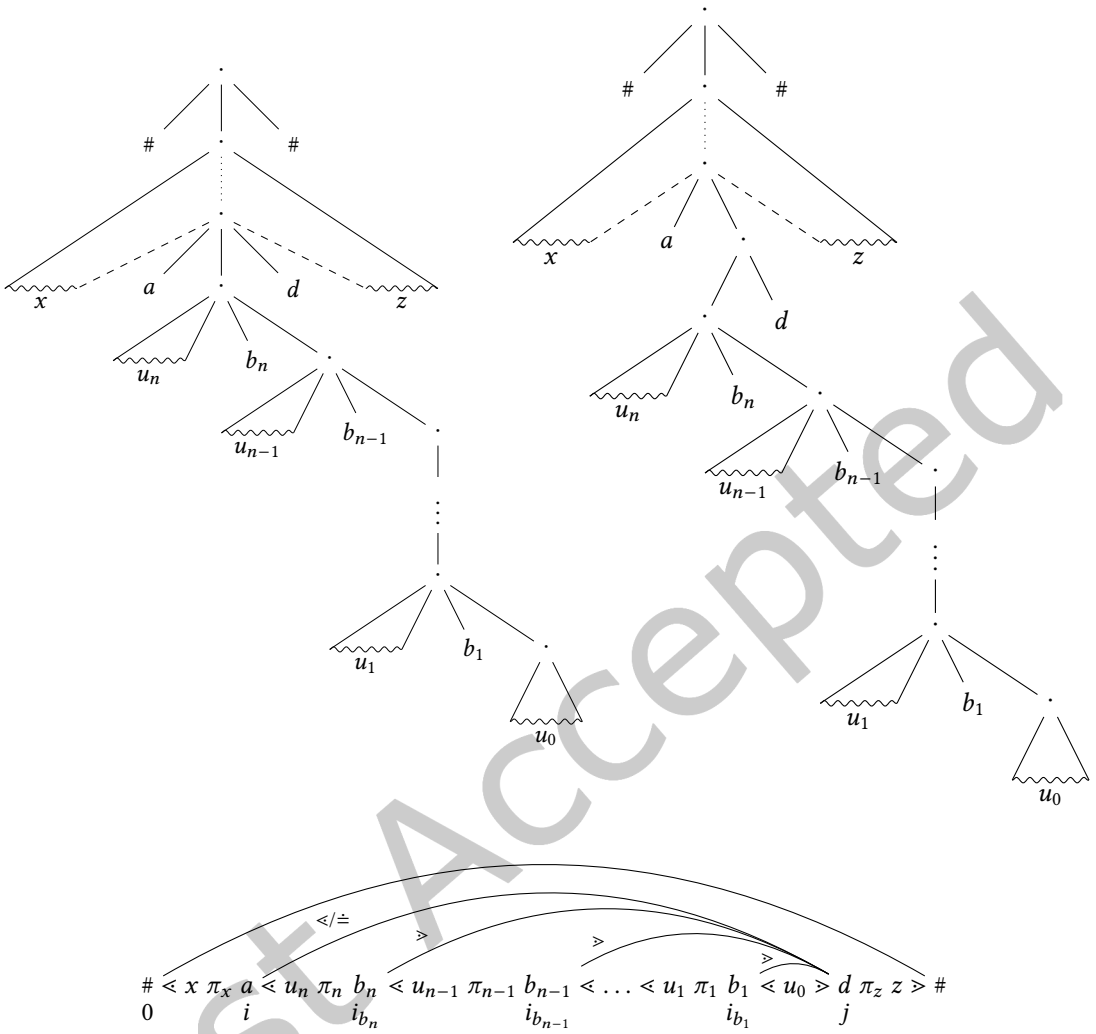


Fig. 23. The two possible STs of a generic OP word $w = xyz$ (top) expanded on the rightmost non-terminal, and its flat representation with chains (bottom). Wavy lines are placeholders for frontiers of subtrees or parts thereof. We have either $a \doteq d$ (top left) or $a \lessdot d$ (top right), and $b_k \gtrdot d$ for $1 \le k \le n$. For $1 \le k \le n$, we either have ${}^{b_{k+1}}[u_k]^{b_k}$, or u_k is of the form $v_0^k c_0^k v_1^k c_1^k \dots c_{m_k}^k v_{m_k+1}^k$, where $c_p^k \doteq c_{p+1}^k$ for $0 \le p < m_k$, $c_{m_k}^k \doteq b_k$, and resp. $a \lessdot c_0^n$ and $b_{k+1} \lessdot c_0^k$. Moreover, for each $0 \le p < m_k$, either $v_{p+1}^k = \varepsilon$ or ${}^{c_p^k}[v_{p+1}^k]^{c_{p+1}^k}$; either $v_{m_k+1}^k = \varepsilon$ or ${}^{c_{m_k}^k}[v_{m_k+1}^k]^{b_k}$, and either $v_0^k = \varepsilon$ or ${}^{b_{k+1}}[v_0^k]^{c_0^k}$ (resp. ${}^{a}[v_0^n]^{c_0^n}$). u_0 has the same form, except $v_{m_0}^0 = \varepsilon$ and $c_{m_0}^0 \gtrdot d$. The π_is are placeholders for precedence relations, and they vary depending on the surrounding terminal characters. Chains that may or may not exist depending on the form of each u_k are not shown by edges (e.g., between a and b_n).

this is the last pop move, and the next one will be a push or a shift. So, it transitions to $\langle dz, \Phi^{(n)}(j), \mu\rangle$, with and $\zeta_L \in \Phi_p^{(n)}(j)$ or $\zeta_{\doteq} \in \Phi_p^{(n)}(j)$, and $\chi_P^{\gtrdot}\psi \in \Phi_p^{(n)}(j)$, according to rule (28). Also, $\zeta_R \in \Phi_p^{(n)}(j)$, because the previous move was a pop. At this point, d is read by a shift or a push transition, so $\Phi^g(j) = \Phi^{(n)}(j)$, and the new stack is $\beta\gamma$ with either $\beta = [d, \Phi^g(k)]$ or $\beta = [d, \Phi^g(j)][a, \Phi^g(k)]$. According to rule (27), $\chi_P^{\gtrdot}\psi \in \Phi_c^{(n)}(j)$, which satisfies claim A_1.

$[A_2 \Rightarrow A_1]$ Suppose the automaton reaches a state $\Phi^g(j) = \Phi^{(n)}(j)$ s.t. $\chi_P^{\gtrdot}\psi \in \Phi_c^g(j)$ during an accepting computation. Position j must be read by either a push or a shift move, so either $\zeta_L \in \Phi_p^g(j)$ or $\zeta_{\doteq} \in \Phi_p^g(j)$. By rule (27), for the computation to continue, we have $\zeta_R \in \Phi_p^g(j)$. So, the transition leading to state $\Phi_p^g(j)$ must be a pop, and the related word position d is the right context of a chain. Let $\Phi^{(n-1)}(j)$ be the starting state of this transition. Since $\chi_P^{\gtrdot}\psi \in \Phi_p^g(j)$, by rule (28) we have $\chi_P^{\gtrdot}\psi \in \Phi_p^{(n-1)}(j)$. By rule (26), this transition must be preceded by another pop, so d is the right context of at least two chains, and the word being read has one of the forms of Fig. 23, with $n \geq 1$.

So, before reading d, the OPA performs a pop transition for each inner chain having d as a right context, i.e. those having b_p, $1 \leq p \leq n$, as left contexts in Fig. 23, plus one for the outermost chain (whose left context is a). By rule (29), $\chi_P^{\gtrdot}\psi$ is propagated backwards through such transitions from the one before d is read, to one in which $\chi_P^{\lessdot}\psi \vee \ominus^{\lessdot}\psi$ is contained into the popped state.

By rule (26), for the computation to reach such pop transitions, the propagation of $\chi_P^{\gtrdot}\psi$ as a pending obligation must stop. So, the OPA must reach a configuration $\langle dz, \Phi^{(q)}(j), [b_q, \Phi^g(i_{c_0^q})] \dots \mu\rangle$ with $\chi_P^{\lessdot}\psi \vee \ominus^{\lessdot}\psi \in \Phi_c^g(i_{c_0^q})$. Note that the following argument also applies to the case in which, in Fig. 23, $u_q = \varepsilon$, by substituting b_q for c_0^q. The topmost stack symbol was pushed after configuration $\langle c_0^q \dots z, \Phi^g(i_{c_0^q}), [b_{q-1}, \Phi^g(i_{c_0^{q-1}})] \dots \mu\rangle$. We have $b_{q-1} \lessdot c_0^q$. If $v_0^q = \varepsilon$, and c_0^q is in the position next to b_{q-1}, $\ominus^{\lessdot}\psi$ holds, while if $v_0^q \neq \varepsilon$, since ${}^{b_{q-1}}[v_0^q]^{c_0^q}$ is a chain, $\chi_P^{\lessdot}\psi$ holds. Therefore, ψ holds in b_{q-1}. Since $b_{q-1} \gtrdot d$ and $\chi(i_{b_{q-1}}, j)$, $\chi_P^{\gtrdot}\psi$ holds in j.

$[I_1 \Rightarrow I_2]$ follows from $A_1 \Rightarrow A_2$ and $\mathcal{A}_{\varphi - \chi_P^{\lessdot}\psi}$'s $\mathscr{DR}$ rules being a strict subset of $\mathcal{A}_\varphi$'s. $[I_2 \Rightarrow I_1]$ again follows from $A_2 \Rightarrow A_1$ and the fact that rules (26) and (27) do not propagate $\chi_P^{\gtrdot}\psi$ further during the transition that reads j, so rules 26-29 may not prevent the computation from reaching a final state. □

B OMITTED GRAPHS FROM THE EXPERIMENTAL EVALUATION

Here we report graphs from the experimental evaluation described in Section 7.2 for formulas **Q.2**–**Q.6** and formulas **Q.8**–**Q.10** that have been omitted from the main text.

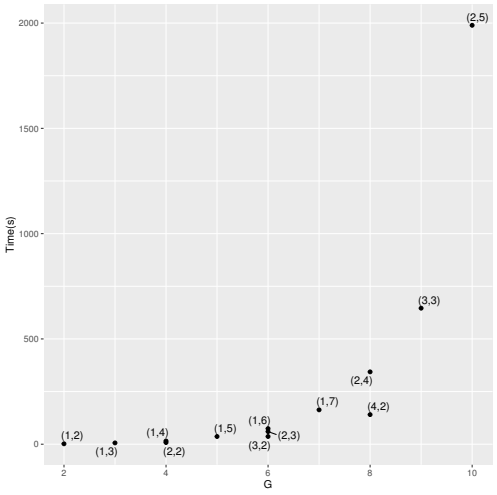


Fig. 24. Experimental results on property **Q.2**

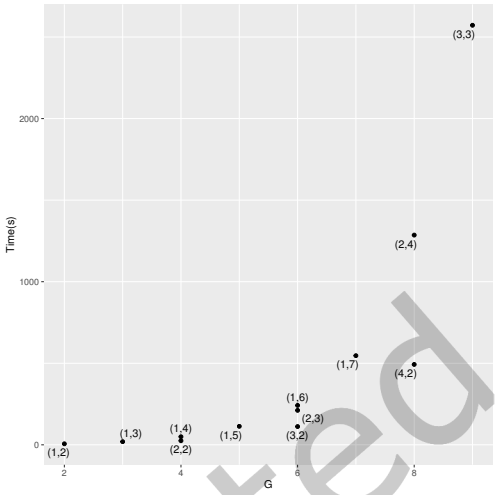


Fig. 25. Experimental results on property **Q.3**

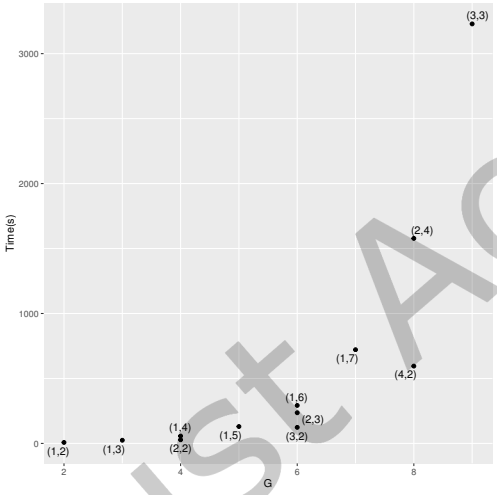


Fig. 26. Experimental results on property **Q.4**

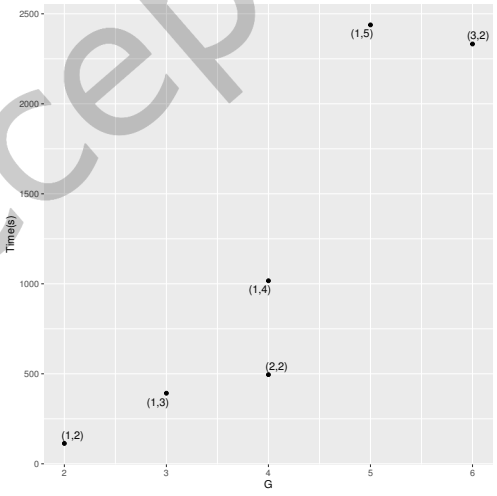


Fig. 27. Experimental results on property **Q.5**

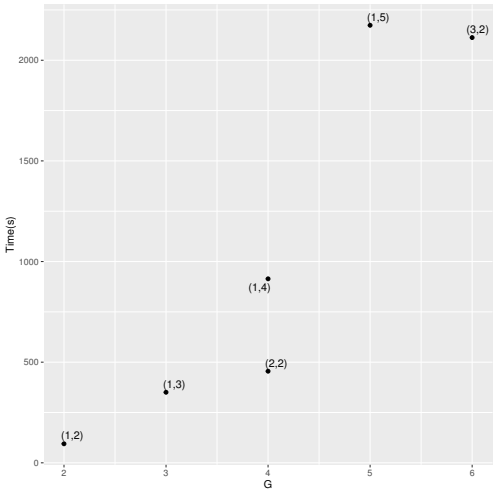


Fig. 28. Experimental results on property **Q.6**

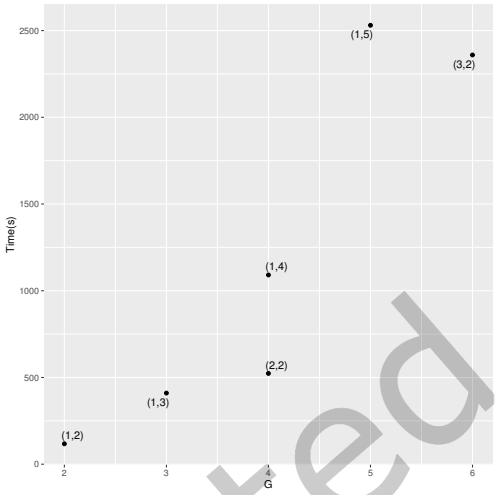


Fig. 29. Experimental results on property **Q.9**

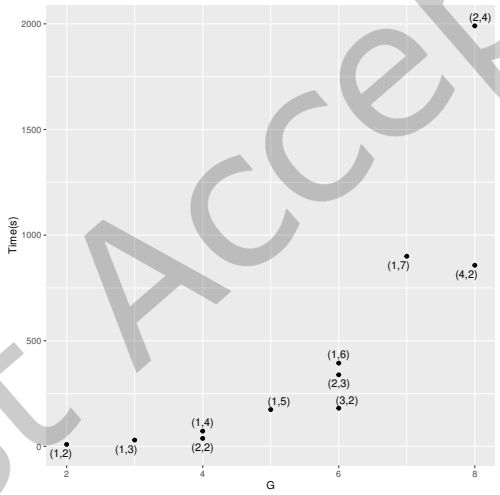


Fig. 30. Experimental results on property **Q.10**