POLITECNICO
MILANO 1863

Dipartimento di Elettronica, Informazione e Bioingegneria
Master Thesis in Information Technology

Counterexample extraction in a Context-Free model checker

Master Thesis of:
Giuseppe Scherini

Supervisor:
Prof. Matteo Pradella

Co-supervisor:
Prof. Michele Chiari

2023

Abstract

Model Checking is a formal method to verify the correctness of hardware or software systems. From these a mathematical model is derived and some specifications, expressed in a logical formalism, are checked to be true. The advantage over the canonical testing is that it is possible to have an automatic and exhaustive verification of the adherence of the requirements to the model. Specifications are usually expressed using formulas of temporal logics. In this thesis the Precedence Oriented Temporal Logic (POTL) is discussed as valid means of describing properties for model checking. It is based on Operator Precedence Languages (OPLs), a class of languages studied in the sixties and then recently resumed due to some interesting feautures: the high expressiveness, their monadic second order characterization, the decididability of the emptiness problem, the property of closure under intersection, the suitability to represent the syntax of programming langauges. Then the Precedence Oriented Model Checker (POMC), a model checking tool developed in Politecnico di Milano, is presented and his behavior briefly described. Based on POTL logic and Operator Precedence Automaton (OPA), it receives as input a program and a POTL formula expressing a property and it return an affirmative answer if the property is verified for all the possible executions of the program, a negative one if it is not and eventually a counterexample is returned as proof. Due to the nature of the model checking algorithm of POMC the counterexamples could contain ellipsis or omitted parts. Our work has been that of improving the process of counterexample extraction showing completed traces.

Acknowledgments

Ringrazio Dionzol

Contents

Chapter 1

Introduction

Validation and verification are fundamental parts of the software development cycle: a software not fulfilling his requirements is completely useless and sometimes could reveal also dangerous. That's why we are interested in a method to check the correctness of a program and eventually, if it's not correct, find a proof of the bad behaviour, for example an invalid execution.

In the last years, beside the classic testing, model checking has received a lot of attention because it is showing itself very advantageous respect to the canonical approach. This strategy consists of:

- the conversion of the software to a mathematically unambiguous model
- the description of his properties by a logical formalism
- the exploration of the state-space of the model to confirm correctness or, conversely, find an error

The main benefits are that the exhaustive and complete exploration of the possible state can certify the absence of bugs and not only the presence of them as happens with testing, the process presents an high level of automation and requires the user only to define the model and the properties to check. At first the formalism adopted to implement the validation process was by means of Transition Systems (TSs) and Finite-State Automaton (FSA) for what concerns the model, Linear Temporal Logic (LTL) for what concern properties' definition [11]. The latter operates on a linear sequence of discrete steps that could be seen as a program run.

Anyway with modern programs we usually want to check more complex specifications, such as Hoare style pre/post conditions [7] and stack inspection [8], that can't be formulated with LTL, capable of expressing only the First-Order Logic (FOL) definable fragment of regular languages. As subsequent step the class of Visibly Pushdown Languages (VPLs) [1] is introduced because of his member languages' condition of being a middle ground between context-free and regular languages: their logical characterization could be in terms of Monadic Second-Order (MSO), they are closed under union, intersection, complementation, concatenation and Kleene*, they present a visible structure in the syntax tree. In fact their terminator symbols are partitioned in the three categories call, internal and return. Internal symbols are surrounded by call and return symbols that can be seen like matching parenthesis. This allow within a program to simulate related events like call-return of a function or throw-catch of an exception. Nevertheless also this family of languages has his critical issues: the relation between symbols could be only one-to-one (we wish one-to-many and many-to-one) and the development of a word could be ambigous.

So to cope with these problems the last step is to introduce the Operator Precedence Languages (OPLs) [10], a class of languages more powerful than VPL. It was born to be applied to efficient parsing [6] but recently it revealed robust enough for model checking due to his characteristics: their languages are closed by Boolean operations [5], concatenation and Kleene* [4], language inclusion and emptiness are decidable problems. The peculiarity of these languages is that non-terminal symbols are associated by a binary relation of precedence: we will see that this permits to create a hierarchy and an association between symbols that could be also one-to-many and many-to-one and to develop the Syntax Tree in an unambigous way. OPL can be recognized by a specific variation of Pushdown Automata called Operator Precedence Automaton (OPA), while the logic that could express it is called Precedence Oriented Temporal Logic (POTL).

Both these elements are the basis of the model checker tool Precedence Oriented Model Checker (POMC) [3].The tool acts in this way: it receives a program and a property to check expressed in POTL, it creates the corresponding OPAs and resolve the emptiness problem, trying to search if a word recognizable by the intersection automaton between program and property exists or not. In the case it doesn't, the property is fulfilled, otherwise it's false. It's interesting in the latter case to receive as output the word in the OPL that refutes the property checked: this correspond to an incorrect run of the program and could be useful during the phase of debugging. We can discover under which condition the integrity of our program is compromised and which path violates a property that we want true for all its possible execution, in order to later fix it.

POMC return a useful counterexample in case of violation, but due to the nature of the exploring-state algorithm sometimes it can be shown only a partial trace with some omitted moves replaced by a construct called "Summary". The aim of the work discussed in this thesis has been to review the counterexample system of POMC, implementing new functions and allowing to show a complete counterexample trace substituting summaries with the corresponding real moves.

1.1 Structure of the Thesis

This thesis is structured in six parts:

- Chapter 2 introduces the theoretical notion of Operator Precedence Languages and Operator Precedence Automata that are at the basis of the POMC tool
- Chapter 3 describes POTL logic in his syntax and semantic
- Chapter 4 briefly describes the theory and the implementation of POMC introducing the concepts of counterexample
- Chapter 5 explains the procedure implemented to complete counterexample trace
- Chapter 6 shows the experimental results where the new functionality is applied
- Chapter 7 conclude the thesis giving some cues for the future

Chapter 2

Operator Precedence Languages

In this chapter the main theoretical notion of Operation Precedence Languages is presented.

2.1 Introduction to OPL

Firstly, we recall some notions and notations from formal language theory.

Definition 2.1 (Context-Free Grammar (CFG))**.** A Context-Free Grammar (CFG) G is a tuple (V, Σ, P, S) where:

- V_N (nonterminal alphabet) and Σ (terminal alphabet) are two disjoint finite sets of characters: their union $V_N \cup \Sigma$ is called V
- P is a finite set of rules of the form $A \to \alpha$, with $A \in V_N$ being the left-hand side (lhs) and $\alpha \in V$ being the right-hand side (rhs)
- $S \in V_N$ is the start symbol or axiom

The following naming conventions are adopted, unless otherwise specified: uppercase Latin letters A, B, ... denote nonterminal characters; lowercase letters a, b, ... at the beginning of the Latin alphabet denote terminal characters; lowercase letters x, y, ... at the end of the Latin alphabet denote terminal strings; lowercase Greek letters α, β, ... denote strings over V, with the letter ε indicating the empty string.

Direct derivation is denoted as $\alpha \Rightarrow \beta$, meaning that $\alpha = \alpha_1\alpha_2\alpha_3$, $\beta = \alpha_1\alpha_2'\alpha_3$, and $\alpha \to \alpha_2' \in P$. Derivation, that is the reflective and transitive closure of the $\Rightarrow$ relation, is denoted by $\overset{*}{\Rightarrow}$. The language generated by a a grammar G is thus defined as $L_G = \{x \in \Sigma^* \mid S \overset{*}{\Rightarrow} x\}$

Definition 2.2. A production rule is in *operator form* if its rhs has no consecutive non-terminals. An *operator grammar* only contains rules in operator form.

Let's do an example of a particular Operator Grammar:

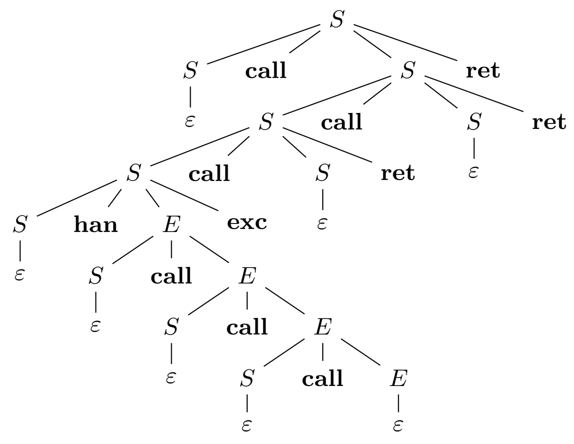


Figure 2.1: Syntax tree of a word derived according to grammar $G_{\mathbf{call}}$.

Example 2.3. Grammar $G_{\mathbf{call}} = (\{S, E\}, \Sigma_{\mathbf{call}}, P_{\mathbf{call}}, S)$, with $\Sigma_{\mathbf{call}} = \{\mathbf{call}, \mathbf{ret}, \mathbf{han}, \mathbf{exc}\}$, has the following production rules:

$$\begin{aligned} &S \to S\,\mathbf{call}\,S\,\mathbf{ret} && E \to S\,\mathbf{call}\,E \\ &S \to S\,\mathbf{han}\,S\,\mathbf{exc} && E \to \varepsilon \\ &S \to \varepsilon && \end{aligned}$$

$G_{\mathbf{call}}$ is made only of rules in operator form, so it's an operator grammar. This grammar ideally can represent the execution trace of a procedural program with exceptions:

call represents the fact that a procedure call occurs

ret represents the fact that a procedure terminates normally and returns to its caller

exc represents the fact that an exception is raised

han represents the fact that an exception handler is installed.

In figure 2.1 it's showed the syntax tree of a possible derivation of the grammar. Every function call is enclosed in the $\mathbf{call} - \mathbf{ret}$ pair of the caller and the calls terminated by an exception are enclosed in its $\mathbf{han} - \mathbf{exc}$ pair.

Input symbols in OPLs are realated by a precedence relation that consists of three possibilities: *yelds* precedence, *equal* in precedence, *takes* precedence. This relation drives the parsing from the final word to the axiom and permit to have an unambiguous syntax tree, so an unique derivation. That's why in the beginning OPLs where used for bottom-up parsing. The three Precedence Relations (PRs) are denoted by the following symbols:

- yelds precedence $\to \lessdot$
- equal in precedence $\to \doteq$
- take precedence $\to \gtrdot$

They must not be confused with the symbols of equality and inequality with which they not share their typical ordering and equivalence properties. 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 the grammar's rule contains a non-terminal character before b or not; $a \doteq b$ 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. Here is the mathematical formulation of precedence relation:

Definition 2.4 (Precedence Relations). Let $G = (V, \Sigma, P, S)$ be an operator grammar, and $A \in V$. Its left and right terminal sets are:

$$\mathcal{L}_G(A) := \{a \in \Sigma \mid A \Rightarrow^*_G Ba\alpha\} \qquad \mathcal{R}_G(A) := \{a \in \Sigma \mid A \Rightarrow^*_G \alpha aB\}$$

with $B \in V \cup \{\varepsilon\}$ and $\alpha \in (V \cup \Sigma)^*$.

For any $a, b \in \Sigma$, and for some $\alpha, \beta \in (V \cup \Sigma)^*$, we have

- $a \lessdot b$ iff $(A \to \alpha a D \beta) \in P$ for some $D \in V$ such that $b \in \mathcal{L}_G(D)$;
- $a \doteq b$ iff $(A \to \alpha a B b \beta) \in P$ for some $B \in V \cup \{\varepsilon\}$;
- $a \gtrdot b$ iff $(A \to \alpha D b \beta) \in P$ for some $D \in V$ such that $a \in \mathcal{R}_G(D)$.

Left and right terminal sets in the example 2.3 for each nonterminal symbol are:

$$\mathcal{L}(S) = \mathcal{L}(E) = \{\mathbf{call}, \mathbf{han}\} \qquad \mathcal{R}(S) = \{\mathbf{ret}, \mathbf{exc}\} \qquad \mathcal{R}(E) = \{\mathbf{call}\}$$

Now, we can give a grammar-based definition of OPL:

Definition 2.5 (Operator Precedence Language (OPL)). A grammar G is an *operator precedence—or Floyd—grammar* (OPG) iff at most one PR holds between any pair of terminals in Σ. Formally, for any $a, b \in \Sigma$ if $a\ \pi_1\ b$ and $a\ \pi_2\ b$ then $\pi_1 = \pi_2$ (with $\pi_1, \pi_2 \in \{\lessdot, \doteq, \gtrdot\}$). An *operator precedence language* is any language generated by an Operator Precedence Grammar (OPG).

PRs between all pairs of terminals can be gathered in a matrix which, as we shall see, contains all the information needed to determine a string's context-free structure.

Definition 2.6 (Operator Precedence Matrix (OPM)). An OPM M over Σ is a partial function $(\Sigma \cup \{\#\})^2 \to \{\lessdot, \doteq, \gtrdot\}$, that, for each ordered pair (a, b), defines the PR $M(a, b)$ holding between a and b. If the function is total we say that M is *complete*. We call the pair (Σ, M) an *operator precedence alphabet*.

In the following, strings will be surrounded by a pair of $\#$ delimiters. By convention, the initial $\#$ can only yield precedence, and other symbols take precedence on the ending $\#$. If $M(a, b) = \pi$, where $\pi \in \{\lessdot, \doteq, \gtrdot\}$, we write $a\ \pi\ b$. For $u, v \in \Sigma^+$ we write $u\ \pi\ v$ if $u = xa$ and $v = by$ with $a\ \pi\ b$.

Altough OPMs may be defined containing multiple PRs in each cell, here we only consider those containing at most one, which are generated by OPLs.

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

Figure 2.2: The OPM $M_{\mathbf{call}}$.

2.2 Chains

Parsing a OPL word we notice that we can recognize patterns of the form $a \lessdot c_1 \doteq \dots \doteq c_\ell \gtrdot b$ and that we can reduce them to a non-terminal character. We call this type of pattern *chains*, and we formalize them as follows:

Definition 2.7 (Simple chain)**.** 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 $c_0 \lessdot c_1 \doteq c_2 \dots c_{\ell-1} \doteq c_\ell \gtrdot c_{\ell+1}$.

Definition 2.8 (Composed chain)**.** 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 ${}^{c_0}[s_0 c_1 s_1 c_2 \dots c_\ell s_\ell]^{c_{\ell+1}}$. c_0 (resp. $c_{\ell+1}$) is called its *left* (resp. *right*) *context*; all terminals between them are called its *body*.

Definition 2.9 (Compatible words)**.** 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.

Example 2.10. Let's take a word generated by the grammar of example 2.3 and let's add # delimiters:

$$w_{ex} = \#\mathbf{call\ han\ call\ call\ call\ exc\ call\ ret\ call\ ret\ ret}\#$$

Here is the same word but with chains composing it shown in brackets:

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

2.3 OPAs

Operator Precedence Automata (OPAs) are a special kind of Pushdown Automata that recognize OPLs. Every move of this type of automata is driven by the existing precedence relation between topmost stack symbol and next symbol to be read. If they are in the $\lessdot$ relation next input symbol is pushed; if they are in the $\doteq$ relation the topmost one symbol is updated with the new one; if they are in $\gtrdot$ relation the topmost symbol is popped. More formally:

Definition 2.11 (Operator Precedence Automaton (OPA))**.** An OPA is a tuple $\mathcal{A} = (\Sigma, M, Q, I, F, \delta)$ where: (Σ, M) is an operator precedence alphabet, Q is a finite set of states (disjoint from Σ), $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final

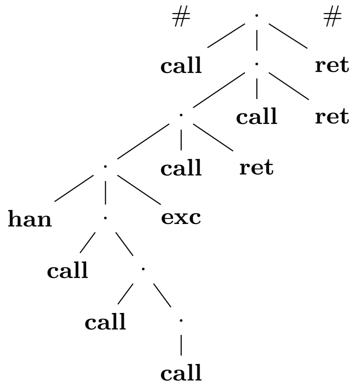


Figure 2.3: The ST corresponding to word w_{ex}. Dots represent non-terminals.

states, $\delta \subseteq Q \times (\Sigma \cup Q) \times Q$ is the transition relation, which is the union of the three disjoint relations $\delta_{shift} \subseteq Q \times \Sigma \times Q$, $\delta_{push} \subseteq Q \times \Sigma \times Q$, and $\delta_{pop} \subseteq Q \times Q \times Q$.

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

To define the semantics of OPAs, we need some new notations. Letters $p, q, p_i, q_i, \dots$ 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 $\gamma = \gamma_n \dots \gamma_1 \bot$, 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 $c_0 \vdash c_1 \vdash \dots \vdash c_n$ of *moves* or *transitions* $c_i \vdash c_{i+1}$. As mentioned previously, 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 $L(\mathcal{A}) = \{x \in \Sigma^* \mid \langle x\#,\ q_I,\ \bot \rangle \vdash^* \langle \#,\ q_F,\ \bot \rangle, q_I \in I, q_F \in F\}$.

OPAs also rely on the OPM to parse words. The relationship between their runs and parsing is highlighted by *supports*:

Definition 2.12. Let $\mathcal{A}$ be an OPA. We call a *support* for simple chain ${}^{c_0}[c_1 c_2 \dots c_\ell]^{c_{\ell+1}}$ any path in $\mathcal{A}$ of the form $q_0 \xrightarrow{c_1} q_1 \dashrightarrow \dots \dashrightarrow q_{\ell-1} \overset{c_\ell}{\dashrightarrow} q_\ell \overset{q_0}{\Longrightarrow} q_{\ell+1}$. The label of the last (and only) pop is exactly q_0, i.e. the first state of the path; this pop is executed because of relation $c_\ell \gtrdot c_{\ell+1}$.

We call a *support for the composed chain* ${}^{c_0}[s_0 c_1 s_1 c_2 \dots c_\ell s_\ell]^{c_{\ell+1}}$ any path in $\mathcal{A}$ of the form $q_0 \overset{s_0}{\leadsto} q'_0 \xrightarrow{c_1} q_1 \overset{s_1}{\leadsto} q'_1 \overset{c_2}{\dashrightarrow} \dots \overset{c_\ell}{\dashrightarrow} q_\ell \overset{s_\ell}{\leadsto} q'_\ell \overset{q'_0}{\Longrightarrow} q_{\ell+1}$ where, for every $i = 0, 1, \dots, \ell$: if $s_i \neq \varepsilon$, then $q_i \overset{s_i}{\leadsto} q'_i$ is a support for the chain ${}^{c_i}[s_i]^{c_{i+1}}$, else $q'_i = q_i$.

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

Theorem 2.13. *Given two OPAs $\mathcal{A}_1$ and $\mathcal{A}_2$ on Operator Precedence (OP) alphabet (Σ, M), it is possible to effectively build the following OPAs:*

- *$\mathcal{A}_\cap$ such that $L(\mathcal{A}_\cap) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$;*
- *$\mathcal{A}_\cup$ such that $L(\mathcal{A}_\cup) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$;*
- *$\overline{\mathcal{A}_1}$ such that $L(\overline{\mathcal{A}_1}) = \Sigma^* \setminus L(\mathcal{A}_1)$.*

The proof is shown in [2] The property about the construction of the intersection OPA is fundamental for what concerns the creation of the exploration-state of the model checker POMC.

Here in this section has been presented the main background of OPAs recognizing finite words i.e. strings with a finite length, that in our context represent finite runs of a programme. This theory can be extended to infinite words, called also ω-words, that represent infinite runs of a programme, but we won't go deeper into this topic. From here on we will treat only the finite case (finite OP words, finite OPAs).

2.4 Modeling Programs with OPA

Let's show how OPAs can naturally model procedural programming languages. The OP alphabet is represented by $(\mathcal{P}(AP), M_{AP})$, where AP is the set of atomic propositions that describe events and states of the program and M_{AP} is the OPM concerning APs. The set of AP could be partitioned into two sets: a set of normal propositional labels (in round font) and a set of *structural labels* (in bold). Structural labels define the OP structure of the word: M_{AP} is only defined for subsets of AP containing exactly one structural label, so that given two structural labels $\mathbf{l}_1, \mathbf{l}_2$, for any $a, a', b, b' \in \mathcal{P}(AP)$ s.t. $\mathbf{l}_1 \in a, a'$ and $\mathbf{l}_2 \in b, b'$ we have $M_{AP}(a, b) = M_{AP}(a', b')$. In this way, it is possible to define an OPM on the entire $\mathcal{P}(AP)$ by only giving the relations between structural labels, as we did for $M_{\mathbf{call}}$.

Figure 2.4 shows how to model a procedural program with an OPA. The OPA simulates the program's behavior with respect to the stack, by expressing its execution traces with four event kinds: **call** corresponds to a function call, **ret** to a return from a function, **han** to the installation of an exception handler by a `try` statement and **exc** to the raising of an exception. OPM $M_{\mathbf{call}}$ defines the context-free structure of the word, which is strictly linked with the programming language semantics: the $\lessdot$ PR

```
    pA() {
A0:   try {
A1:     pB();
A2:   } catch {
A3:     pErr();
A4:     pErr();
      }
Ar: }
```

```
    pB() {
B0:   pC();
Br: }
```

```
    pC() {
C0:   if (*) {
C1:     throw;
C2:   } else {
C3:     pC();
      }
Cr: }
```

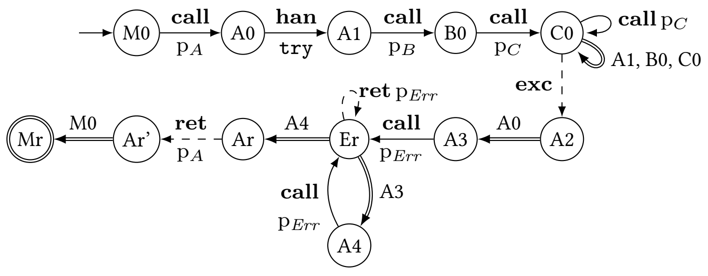


Figure 2.4: Example procedural program (top) and the derived OPA (bottom). Push, shift, pop moves are shown by, resp., solid, dashed and double arrows.

causes nesting (e.g., **call**s can be nested into other **call**s), and the $\doteq$ PR implies a one-to-one relation, e.g. between a **call** and the **ret** of the same function, and a **han** and the **exc** it catches.

Each OPA state represents a line in the source code. First, procedure p_A is called by the program loader (M0), and $[\{\mathbf{call}, \mathrm{p}_A\}, \mathrm{M0}]$ is pushed onto the stack, to track the program state before the **call**. Then, the `try` statement at line `A0` of p_A installs a handler. All subsequent calls to p_B and p_C push new stack symbols on top of the one pushed with **han**. p_C may only call itself recursively, or throw an exception, but never return normally. This is reflected by **exc** being the only transition leading from state C0 to the accepting state Mr, and p_B and p_C having no way to a normal **ret**. The OPA has a look-ahead of one input symbol, so when it encounters **exc**, it must pop all symbols in the stack, corresponding to active function frames, until it finds the one with **han** in it, which cannot be popped because **han** $\doteq$ **exc**. Notice that such behavior cannot be modeled by Visibly Pushdown Automata (VPAs), because they need to read an input symbol for each pop move. Thus, **han** protects the parent function from the exception. Since the state contained in **han**'s stack symbol is A0, the execution resumes in the `catch` clause of p_A. p_A then calls twice the library error-handling function p_{Err}, which ends regularly both times, and returns. The string of Figure 2.2 is accepted by this OPA.

In this example, we only model the stack behavior for simplicity, but other statements, such as assignments, and other behaviors, such as continuations, could be modeled by a different choice of the OPM, and other aspects of the program's state by appropriate abstractions [9].

Chapter 3

Precedence Oriented Temporal Logic (POTL)

In this chapter we describe in a complete and formal way POTL for finite word OPLs

3.1 Some introductory concepts

The peculiarity of POTL is that symbols in a word are adressed by the position defined by the word structure and the positions are bound by a one-to-many or many-to-one relation called chain relation. More formally:

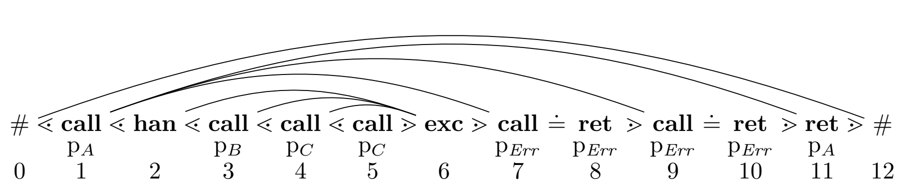
Definition 3.1 (Word Structure). A *word structure* (or OP word) is a tuple $(U, <, M_{AP}, P)$ where

- $U = \{0, 1, \dots, n, n+1\}$, with $n \in \mathbb{N}$ is a set of word positions;
- $<$ is linear order on U;
- M_{AP} is an operator precedence matrix on $\mathcal{P}(AP)$
- $P : AP \to \mathcal{P}(U)$ is a function associating each atomic proposition with the set of positions in which it holds, with $0, (n+1) \in P(\#)$

Given two position i, j and a PR π, we write $i \ \pi \ j$ to say $a \ \pi \ b$, where $a = \{p \mid i \in P(p)\}$, and $b = \{p \mid j \in P(p)\}$

Definition 3.2 (Chain relation). The *chain relation* $\chi \subseteq U \times U$ so that $\chi(i, j)$ holds between two positions i, j iff $i < j - 1$, and i and j are respectively the left and right contexts of the same chain. 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) $i_1 < i_2 < \dots < i_n$ 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 $2 \leq q \leq n$.



# ⋖ call ⋖ han ⋖ call ⋖ call ⋖ call ⋗ exc ⋗ call ≐ ret ⋗ call ≐ ret ⋗ ret ⋗ #
p_A p_B p_C p_C p_Err p_Err p_Err p_Err p_A
0 1 2 3 4 5 6 7 8 9 10 11 12

Figure 3.1: The example word w_{ex} from Chapter 2 as an OP word. Chains are highlighted by arrows joining their contexts.

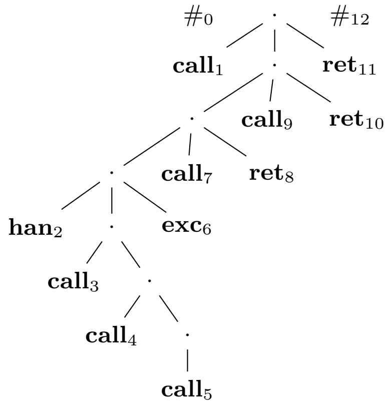


Figure 3.2: The ST of word w_{ex} (the same as Figure 2.3, but with position numbers). Dots represent non-terminals.

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 $1 \leq q \leq n-1$.

Figure 3.1 shows the example string of 2.10 highlighting chains through arrows between their left and right context. Note that Structural Labels, whose precedence with each other is given by the OPM, are in bold, while other atomic propositions are shown below them. When interpreting this string as an execution trace over a generic procedural program, the other atomic proposition denote function names, while structural labels have the usual meaning of function call (**call**), installing an exception handler on the stack (**han**), function returning (**ret**), and exception raising (**exc**). We can see here how the Chain Relation substitutes matching paranthesis of the Paranthesis grammar and augments their expressive power by allowing a many-to-one and one-to-many relation with composed chains. Looking at the Syntax Tree (ST) of figure 3.2, we say that the right context j of a chain is at the same level as the left one i when $i \doteq j$, at a lower level when $i \lessdot j$, at a higher level if $i \gtrdot j$.

3.2 POTL syntax and operators

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 with respect to 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 $i \in P(\mathrm{a})$. Operators such as $\wedge$ and $\neg$ have the usual semantics from propositional logic, $\square$ and $\Diamond$ have the same meaning as in LTL. Next we will give the formal semantics of POTL operators. Let w be a OP word and $i \in U$ a word position in w:

Next/back operators The *downward* next and back operators $\bigcirc^d$ and $\ominus^d$ are like their LTL counterparts, but they impose an additional requirement on the precedence relation between the current (resp. preceding) and the next (resp. current) position.

- $\bigcirc^d$ and $\ominus^d$ require the (resp. current) position to be at a lower or equal ST level than the current (resp. preceding) one.
- $\bigcirc^u$ and $\ominus^u$ require the (resp. current) position to be at a higher or equal ST level than the current (resp. preceding) one.

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 $(i-1) \doteq i$.
- $(w, i) \models \bigcirc^u \varphi$ iff $(w, i+1) \models \varphi$ and $i \gtrdot (i+1)$ or $i \doteq (i+1)$, and
- $(w, i) \models \ominus^u \varphi$ iff $(w, i-1) \models \varphi$, and $(i-1) \gtrdot i$ or $(i-1) \doteq i$.

Chain Next/Back The *chain* next and back operators χ_F^t and χ_P^t are similar to $\bigcirc^t$ and $\ominus^t$, but, instead of the next element, they impose requirements on respectively 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 $(w, j) \models \varphi$.
- $(w, i) \models \chi_F^d \varphi$ iff there exists a position $j > i$ such that $\chi(i, j)$, $i \gtrdot j$ 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 $\chi(j, i)$, $j \gtrdot i$ or $j \doteq i$, and $(w, j) \models \varphi$.

(Summary) Until/Since operators 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. Formally, $(w, i) \models \psi \, \mathcal{U}_\chi^t \, \theta$ iff either:

- $(w, i) \models \theta$ or
- $(w, i) \models \psi$ and either $(w, i) \models \bigcirc^t(\psi \, \mathcal{U}_\chi^t \, \theta)$ or $\chi_F^t(\psi \, \mathcal{U}_\chi^t \, \theta)$

On the other hand $(w, i) \models \psi \, \mathcal{S}_\chi^t \, \theta$ iff either:

- $(w, i) \models \theta$ or
- $(w, i) \models \psi$ and either $(w, i) \models \ominus^t(\psi \, \mathcal{S}_\chi^t \, \theta)$ or $\chi_P^t(\psi \, \mathcal{S}_\chi^t \, \theta)$

Hierarchical next and back operators The same position may be the left or right context of multiple chains, thus having more than one right or left context. Hierarchical next and back operators allow to specify properties on other positions in the chain relation with the same position with respect to the current one. In particular, $\bigcirc^{u}{}_{H}$ and $\ominus^{u}{}_{H}$, holding on the right context of a chain, allow to specify properties on the other positions which are right contexts and have the same left context as the current one. Respectively, $\bigcirc^{d}{}_{H}$ and $\ominus^{d}{}_{H}$ say something on other left contexts which have the same right context as the current one. Formally:

- $(w, i) \models \bigcirc^{u}_{H} \varphi$ 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 $(w, j) \models \varphi$.

Hierarchical until and since operators The $\mathcal{U}^{\downarrow}$ and $\mathcal{S}^{\uparrow}$ are obtained by iterating the $\bigcirc^{\gtrdot}{}_{H}$ and $\ominus^{\gtrdot}{}_{H}$ operators, similarly to how the Summary Until and Since are defined with respect to the Precedence and Chain operators. Formally, $(w, i) \models \psi\, \mathcal{U}^{u}_{H}\, \theta$ if and only if one of the following conditions holds:

- $(w, i) \models \theta$ and there exists a position $h < i$ such that $\chi(h, i)$ and $h \lessdot i$
- $(w, i) \models \psi$ and $(w, i) \models \bigcirc^{u}{}_{H}\, \psi \mathcal{U}^{u}_{H} \theta$

$(w, i) \models \psi\, \mathcal{S}^{u}_{H}\, \theta$ if and only if one of the following conditions holds:

- $(w, i) \models \theta$ and there exists a position $h < i$ such that $\chi(h, i)$ and $h \lessdot i$
- $(w, i) \models \psi$ and $(w, i) \models \ominus^{u}{}_{H}\, \psi \mathcal{S}^{u}_{H} \theta$

$(w, i) \models \psi\, \mathcal{U}^{d}_{H}\, \theta$ if and only if one of the following conditions holds:

- $(w, i) \models \theta$ and there exists a position $h > i$ such that $\chi(i, h)$ and $i \gtrdot h$
- $(w, i) \models \psi$ and $(w, i) \models \bigcirc^{d}{}_{H}\, \psi \mathcal{U}^{d}_{H} \theta$

$(w, i) \models \psi\, \mathcal{S}^{d}_{H}\, \theta$ if and only if one of the following conditions holds:

- $(w, i) \models \theta$ and there exists a position $h > i$ such that $\chi(i, h)$ and $i \gtrdot h$
- $(w, i) \models \psi$ and $(w, i) \models \ominus^{d}{}_{H}\, \psi \mathcal{S}^{d}_{H} \theta$

Chapter 4

POMC, sketch of theory and practical implementation

In this chapter we briefly expose the theory behind POMC, the model checker tools that uses POTL and OPAs, in the context of finite words. Subsequently we explain how POMC is implemented in practice.

4.1 Bulding the OPA

Let's present firstly the formulation of the model checking problem for POTL:

Definition 4.1 (OP Model Checking Problem)**.** Given an OPA A and a POTL formula φ, establish whether $A \models \varphi$.

We define $Words(\varphi)$ as the set of strings that satisfy POTL formula φ, i.e. the language $L(\varphi)$, and AP s the set of Atomic Propositions. Given a OPA A and a POTL formula φ, we can say:

Definition 4.2 (Semantics of POTL over OPAs)**.** $A \models \varphi$ if and only if $L(A) \subseteq Words(\varphi)$

and then we can derive that:

Definition 4.3 (Satisfaction relation)**.** $A \models \varphi$
iif $L(A) \subseteq Words(\varphi)$
iif $L(A) \cap (2^{AP})^* \setminus Words(\varphi) = \varnothing$
iff $L(A) \cap Words(\neg\varphi) = \varnothing$
Hence, for OPA $A_{\neg\varphi}$ with $L(A_{\neg\varphi}) = Words(\neg\varphi)$ we have: $A \models \varphi$ if and only if $L(A) \cap L(A_{\neg\varphi}) = \varnothing$

This means that a POTL formula φ is satisfied and therefore the model checking problem successful if the language accepted by the automaton A intersected with the language expressed by the negated formula φ is the empty set. Furthermore we can say that φ is satisfied if the language accepted by the automaton A intersected with the language accepted by the automaton associated to negated φ is the empty set. So given an OP alphabet $(\mathcal{P}(AP), M_{AP})$, where AP is a finite set of atomic propositions, a formula φ defined on the same alphabet to be checked against an OPA

A, the procedure generates the OPA $A_{\neg\varphi} = \langle \mathcal{P}(AP), M_{AP}, Q, I, F, \delta \rangle$ associated with the negation of φ, builds the Intersection Automaton $A_{\oplus} = A \otimes A_{\neg\varphi}$, and determines whether the language of $A_{\oplus}$ is empty, that is a decidable problem. We will see below in part the construction rules to build the necessary automatons, only to give an idea of it. For the complete explanation see [2].

4.1.1 Closure

We first introduce $\mathrm{Cl}(\varphi)$, the *closure* of φ, containing all subformulas of φ, plus a few auxiliary operators. Initially, $\mathrm{Cl}(\varphi)$ is the smallest set such that

1. $\varphi \in \mathrm{Cl}(\varphi)$,
2. $AP \subseteq \mathrm{Cl}(\varphi)$,
3. 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);
4. if $\neg\psi \in \mathrm{Cl}(\varphi)$, then $\psi \in \mathrm{Cl}(\varphi)$;
5. if any of $\psi \wedge \theta$ or $\psi \vee \theta$ is in $\mathrm{Cl}(\varphi)$, then $\psi \in \mathrm{Cl}(\varphi)$ and $\theta \in \mathrm{Cl}(\varphi)$;
6. if any of the unary temporal operators (such as $\bigcirc^d$, χ_F^d, ...) is in $\mathrm{Cl}(\varphi)$, and ψ is its argument, then $\psi \in \mathrm{Cl}(\varphi)$;
7. if any of the until- and since-like operators is in $\mathrm{Cl}(\varphi)$, and ψ and θ are its operands, then $\psi, \theta \in \mathrm{Cl}(\varphi)$.

4.1.2 Atom and state's form

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

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

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

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

Figure 4.1: Example accepting run of the automaton for $\chi_F^d\,\mathbf{ret}$.

4.1.3 Some example of construction rules

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

Chain Next operator To handle this operator, we add into $\mathrm{Cl}(\varphi)$ the auxiliary symbol χ_L , which forces the current position to be the first one of a chain body. Let the current state of the OPA be $\Phi \in \mathrm{Atoms}(\varphi)^2$: $\chi_L \in \Phi_p$ iff the next transition (i.e. the one reading the current position) is a push. Formally, if $(\Phi, a, \Psi) \in \delta_{shift}$ or $(\Phi, \Theta, \Psi) \in \delta_{pop}$, for any Φ, Θ, Ψ and a, then $\chi_L \notin \Phi_p$. If $(\Phi, a, \Psi) \in \delta_{push}$, then $\chi_L \in \Phi_p$. For any initial state $(\Phi_c, \Phi_p) \in I$, we have $\chi_L \in \Phi_p$ iff $\# \notin \Phi_c$.

If $\chi_F^d\psi \in \mathrm{Cl}(\varphi)$, we add the following constraints on δ.

1. Let $(\Phi, a, \Psi) \in \delta_{push/shift}$: then $\chi_F^d\psi \in \Phi_c$ iff $\chi_F^d\psi, \chi_L \in \Psi_p$;

2. Let $(\Phi, \Theta, \Psi) \in \delta_{pop}$: then $\chi_F^d\psi \notin \Phi_p$, and $\chi_F^d\psi \in \Theta_p$ iff either (a) $\chi_F^d\psi \in \Psi_p$ or (b) $\psi \in \Phi_c$ and $\chi_L\psi \in \Psi_p$;

3. Let $(\Phi, a, \Psi) \in \delta_{shift}$: then $\chi_F^d\psi \in \Phi_p$ iff $\psi \in \Phi_c$

$\chi_F^u\psi$ is allowed in the pending part of initial states.
On the other hand, if $\chi_F^u\psi \in \mathrm{Cl}(\varphi)$, we add the following constraints:

4. Let $(\Phi, a, \Psi) \in \delta_{push/shift}$: then $\chi_F^u\psi \in \Phi_c$ iff $\chi_F^u\psi, \chi_L \in \Psi_p$;

5. Let $(\Phi, \Theta, \Psi) \in \delta_{pop}$: $\chi_F^u\psi \in \Theta_p$ iff $\chi_F^u\psi \in \Psi_p$, and $\chi_F^u\psi \in \Psi_p$ iff $\psi \in \Psi_c$

6. Let Let $(\Phi, a, \Psi) \in \delta_{shift}$: then $\chi_F^u\psi \in \Phi_p$ iff $\psi \in \Phi_c$

This is only a sketch of construction rules to give an idea. For the complete rules consult [2].

4.2 POMC in practice

In this section the main implementations and algorithms being part of POMC are showed, with a focus on the counterexample return system.

```
opa:  initials = 0;
      finals = (4 10);
      deltaPush =
        (0,  (call pa),   1),
        (1,  (han),       2),
        (2,  (call pb),   3),
        (3,  (call pc),   4),
        (4,  (call pc),   4),
      deltaShift =
        (4, (exc),       5),
        (7, (ret perr),  7),
      deltaPop =
        (4, 2, 4),
        (5, 1, 6),
        (7, 8, 9),
        (11, 0, 10),
```

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

Figure 4.2: Explicit definition of OPA (left) and a MiniProc program (right).

4.2.1 Overview

The tool has been developed in Haskell: an advanced, purely functional, statically typed programming language with lazy evaluation. This last feature means that the evaluation of an expression is deferred until his result is needed by other computations. In this way arguments are not evaluated before they are passed to a function, but only when their values are actually used. Laziness permits so to save computational power and memory, in our case it's useful during the construction and the state exploration of the OPA.

POMC requires as input an OPA and a POTL specification, his negation is transformed into OPA and intersected by cartesian product method with the input OPA. At this point the emptiness algorithm is applied to find if there exists an accepted word or not. The emptiness check exploits a reachability algorithm that check if a particular configuration is reachable by means of a modified Depth-First Search (DFS) of the transition relation. The creation of the state-space is on-the-fly, that is to say that a state is generated only before it is visited, coping the problem of space-state explosion.

Another artifice to improve performance is that of early termination: if during the DFS the emptiness of the OPA is refuted because a counterexample is found, the visit of the states is interrupted because it is no more necessary to continue. Only when the emptiness problem is true we have a complete exploration of states.

As we said earlier, POMC tool requires as input:

- an Operator Precedence Matrix
- a POTL formula
- an OPA

The last one could be derived from two possible source, chosen by the user:

- explicit definition: transitions of the OPA are defined in an explicit way
- miniproc: a simple C-like language to model little programs with only global variables, function without parameters and throw-catch exception

4.2.2 Reachability Algorithm

The reachability algorithm finds if a semiconfiguration is reachable from a specific state and stack symbol. For semiconfiguration of an OPA we mean a pair (q, g), including a state of the OPA and a stack symbol. So formally a semiconfiguration is an element of $C = Q \times \Gamma$ where Q is the set of states and $\Gamma = \Sigma \times Q \cup \{\bot\}$ the cartesian product between the stack alphabet and the set of states, plus the bottom symbol that means empty stack. To solve the emptiness problem wecall the reachability algorithm passing as parameters the set of initial states and we check if from each of them it is possible to reach the final semiconfiguration $Q_R = F$ and $\Gamma_R = \{\bot\}$, where F is the set of final states and Γ_R represent the empty stack. The algorithm halts when the final semiconfiguration is found to be reachable or all the state-space has been explored. To avoid infinite loop and improve performances the procedure makes use of Summaries: during the exploration it could happen that a semiconfiguration at the beginning of a chain that has been already visited is met again. Then knowing that a chain brings always to the same semiconfiguration we can completely skip its body and jump to the corresponding exit state and stack condition. Virtually we add additional edges in the OPA's transition graph, which we call summary edges.

Initial and final semiconfigurations of a chain, when visited, are stored respectively in SupportStart and SupportEnd with all the information to skip the exploration and resuming to the end of chain. Here below the pseudocode for the reachability algorithm is presented (Algorithm 1). Its arguments are a state $q \in Q$, a stack symbol $g \in \Gamma$, a character $c \in \Sigma$ and a look-ahead $l \in \Sigma \cup \{*\}$. If $l = *$, then any character in Σ may be used as a look-ahead. Algorithm 2 shows how to solve the emptiness problem, posing $Q_R = F$ and $\Gamma_R = \{\bot\}$, and calling $Reach(q, \bot, \#, *)$ for each $q \in I$.

Algorithm 1 OPA semi-configuration reachability

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

Algorithm 2 OPA emptiness check

```
1: function IsEmpty($\mathcal{A}$)
2:     $(\Sigma, M_\Sigma, Q, I, F, (\delta_{push}, \delta_{shift}, \delta_{pop})) := \mathcal{A}$
3:     $V := SupportStarts := SupportEnds := \emptyset$
4:     $Q_R = F$
5:     $\Gamma_R = \{\bot\}$
6:     for all $q \in I$ do
7:         if Reach($q, \bot, \#, *$) then return false
8:     return true
```

In practice these algorithms are a little more complex and they are implemented in the `Satisfiability.hs` module of the program.

4.2.3 Sketch of OPA practical construction

Before introducing counterexample traces let's see very sketchily how an OPA can model a Miniproc program. Every state of the OPA represent a state of the program to which POMC randomly assigns an integer Id. Every transition from a state to another is labeled with boolean expression guards that must be true to be performed. Anyway every instruction or operational block within the Miniproc program has a corresponding translation as transition of the OPA: a function call corresponds to a push move with pushing the relative atomic propositon onto the stack, a return from that function corresponds to a pop move popping the relative Atomic Proposition (AP) from the stack. An exception thrown in the same way corresponds to a pop move, while an handler that catches an exception is represented with shift moves. Figure 4.3 shows an example of Miniproc with the corresponding derived OPA.

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

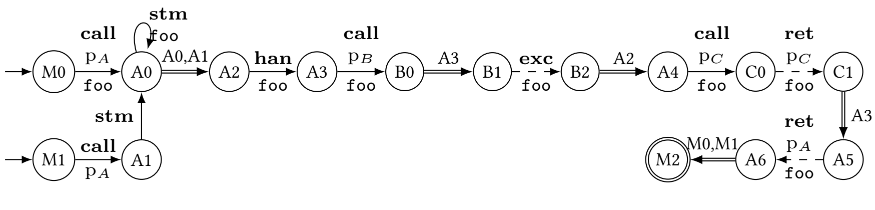


Figure 4.3: MiniProc toy program (top) and the OPA derived by it (bottom).

4.2.4 Counterexamples

Let us remark that model checking an OPA means to verify if the emptiness problem is true or not, namely to check if the intersection of the program's OPA and the negated POTL formula's OPA has at least a word in his language. POMC returns TRUE if emptiness problem is true, false and a counterexample trace if an accepted word is found. It's interesting to have a proof of a path in the OPA that does not comply with the specification, and the first found by the DFS is returned.

A trace in POMC, for the finite case, is in the form:

$$trace = t_0 t_1 t_2 t_3 \dots t_n$$

where every component corresponds to a performed transition relation in the DFS. The frame of a trace component is the following: $(q, [ap_1, ap_2 \dots])$ where $q \in Q$ represent the starting state of the transition and $ap_n \in 2^{AP}$ represent the atomic propositions true on that transition.

Example 4.4. Here the trace from the example *37-generic-medium*, a test from the generic-medium set evaluated in section 6.1:

$[(0, [''call'']), (1, [''call'']), (2, [''...'']), (13, [''ret'']), (15, [''han'']), (19, [''call'']),$
$(3, [''...'']), (20, [''exc'']), (23, [''call'']), (10, [''...'']), (12, [''ret'']), (21, [''ret'']),$
$(8, [''\#''])]$

We can notice for example that the first element $(0, [''call''])$ means that the first transaction performed started from 0 state and *call* was an active AP (a push move has been performed), in the last one $(8, [''\#''])$ 8 is a final state and the active symbol $\#$ means that the stack is empty and we reached the end of the word, in $(2, [''\dots''])$ means that the algorithm has skipped an entire part of a chain starting from state 2. This is an element of type *Summary* and we will come back on it later. This final trace is reworked from another trace that corresponds to the recording of all the transition moves performed during the DFS. This is implemented in the `Satisfiability.hs` module within the *Reachability* function.

Everytime a move is performed, before the algorithm passes to examine a new state, the move is chained to the previous ones and so on until the final move. The frame of an element of this support trace is (moveType, state, stack symbol) where:

- state: is the state from where the transition move started
- stack symbol: is the combination of the active AP in the state just visited and the last state pushed. It represents the head of the stack in the moment of transition.
- movetype: is the type of transition move performed. It could be:
  - *Found* if we are arrived at the end of the DFS, and so in a final state with the empty stack.
  - *Push* if a push move is performed and so a new stack symbol is pushed in the stack
  - *Shift* if a shift move is performed
  - *Pop* if a pop move is performed and so a stack symbol is popped from the stack

  - *Summary* if the algorithm as explained in section 4.2.2 has to skip a part of visit because it has already visited a state or it is falling in an infinite loop.

This support trace when completed is manipulated firstly by removing Pop moves because they do not contribute to the final OP word and secondly extracting the atomic formula valid for each move, obtaining so the final form of example 4.4. We can say that we don't have always a complete trace because of the presence of *Summary* movetype: it introduces omissions that suggests us only the starting state of the chain skipped, but what's inside is hidden: it could be only a pair of push-pop moves or a very long and deep serie of moves. The aim of the work done has been to unwrap the summary moves to obtain the corresponding coherent serie of moves performed by the reachability algorithm and so to output finally a complete trace without any omission. The procedure is presented in the next chapter of this paper.

Chapter 5

Completing the counterexamples

In this chapter we explain the procedure to complete the counterexample trace returned after checking a property on an OPA. To do this the module `Satisfiability.hs` has been slightly modified and the new module `Trace.hs` has been created to contain all data types and functions to handle trace elements.

5.1 Recap of support trace in practical details

As previously explained in addition to the final track, real counterexample of the checked property, POMC makes use of a support trace whose element are in the form (moveType, starting state, stack symbol). Everytime during the DFS a transition is performed the move is recorded and attached to the serie of the previous ones. This happen in every subroutine of the reachability algorithm:

reachPush where push are performed

reachShift where shift are performed

reachPop where pop are performed

During the state-space exploration every visited state is marked and whenever it is encountered again the research stops on that path. Meeting a previously visited state means we can fall in a potential endless loop. Nevertheless it could happen that another path of exploration pass directly through that previously visited state: if it arrive there from another semiconfiguration, with another symbol in head of stack. To avoid that this potential valid path interrupts, the algorithm permits to jump directly to the end of the route where the visited node of the OPA would have brought. Since from every already found state a push type transition is performed, we can say that jumping to the end of the path means skipping the entire chain body starting from that state and arriving to the state subsequent to the pop.

A chain has a fixed outcome i.e. after his examination we arrive always in the same state and with the same head of the stack as before the initial push. That's why reachability algorithm stores the OPA's semi-configuration whenever it enters or exits a chain support, respectively in the sets *SupportStarts* and *SupportEnds*. SupportStarts

1. Push $\rightarrow$ Pop
2. Push $\rightarrow$ Shift $\rightarrow$ Shift $\rightarrow$ Pop
3. Push $\rightarrow$ Push $\rightarrow$ Pop $\rightarrow$ Pop
4. Push $\rightarrow$ Push $\rightarrow$ Shift $\rightarrow$ Pop $\rightarrow$ Shift $\rightarrow$ Pop
5. Push $\rightarrow$ Push $\rightarrow$ Push $\rightarrow$ Shift $\rightarrow$ Pop $\rightarrow$ Shift $\rightarrow$ Pop $\rightarrow$ Pop

Figure 5.1: Possible series of move that a Summary can contain.

is used to shorten the process of exploration of chains in reachPop function. SupportEnds instead contains, for every push state, the possible arrival states at the end of the chain. The algorithm so, when it finds a visited push state, jumps directly to the arrival corresponding states maintaining the previous stack head and resumes the exploration from that point. This operation is performed in the reachPush function and this special transition move generate an element to be attached to the support trace that has Summary as movetype.

The frame of a summary element of support trace is (Summary, q, g), where Summary obviously denote the Summary type of move, q denote the state where the first push of the chain is performed and g denote the symbol on top of the stack. From the new arrival state from the pop move of the chain and the symbol g the exploration will be resumed. From here on the word summary could denote either the element of trace of type summary or the entire unknown body of a chain, the meaning could be inferred by the context.

When the final counterexample trace present a summary element we know for sure that the OPA path corresponding to that summary passes through a starting push from the state of the summary, where that state is pushed into the stack, and a final pop where that state is popped from the stack. What is inside these two extremes is unknown. We don't know if the summary chain is composed only by a single push-pop couple: if it contains also shift moves, if it has inside other subordinate push-pop couples. In the next section we will see the procedure to resolve the summary elements in the trace substituting them with the corresponding paths that the automaton actually travelled.

5.2 Resolving summaries

During the state-space exploration of the OPA, when a summary move is performed the entire body of a chain already explored is skipped and the exploration is resumed at the end of it. Resolving a summary means to get the series of moves performed when that chain was visited the first time. To achieve this result we implement the procedure in two phases:

1. *Recording phase*: all the moves performed during the exploration of the automaton are registered in a specific data structure

2. *Reconstruction phase*: all the previously registered moves are assembled in a consistent way to find the actual trace of a summary

Let's introduce some preparatory concept and subsequently let's explain how the two phases are implemented.

5.2.1 Preliminary concepts

To achieve summary resolution we created a new module called `Trace.hs`. It contains all functions and data types that concerns trace implementation, visualization and debugging. Due to the fact that it is necessary to save data in some specific structures and update them we need an artifice to transcend the pure functional code of Haskell. Fortunately the *ST* monad can help us: it's a monad that permits to write code in a procedural way chaining sequentially some actions that are seen by the rest of the code if they were functional. Almost all the functions in `Trace.hs` module make use of ST monad, returning an object of type ST.ST s (Object to return).

The data types created to represent the trace are the following:

- `data TraceType = Push | Shift | Pop | Summary | Found deriving (Eq, Ord, Show)`: it's the data type representing the type of move that the OPA can perform in a transition. The meaning of each one has been explained in section 4.2.4.
- `type TraceId state = [(TraceType, StateId state, Stack state)]`: it's the data type representing the support trace, it consists of a list of elements whose meaning has been explained in section 4.2.4.
- `type TraceChunk state = [(TraceType, StateId state, Stack state, StateId state)]`: it's the data type representing in his entirety a transaction. It consists of a list of element of this type: (move type, current state, stack symbol, future state) where the first three elements are the same of `TraceId`, while the fourth represent the future state, i.e. the state of arrival of a the performed transaction. It will be useful to link consistent element of the trace when reconstructing it.

Elements of traceId and traceChunk from here on will be called also tuples, denoting the fundamental unit of the trace.

The data structure created to store the tuples registered in the first phase is called `TraceMap` and has the following header: `type TraceMap s state = MV.MVector s (TraceChunk state, TraceChunk state, TraceChunk state)`. `TraceMap` is built on a `MutableVector`, an array with not fixed size but expandable length running in the ST monad. Each element of the mutable vector, indexed by an integer, contains a triple of `traceChunk` (we will explain during recording phase explanation how they have to be interpreted). Let's now explain the two phases of the procedure to resolve summaries.

5.2.2 Recording phase

Firstly we have, during the exploration of the OPA, to register all the transitions that are performed in the context of each chain encountered. Every chain and every simple move within it can be indexed by the starting state of the chain i.e. the state from where the push, the initial move, is performed: indeed when a push is executed the starting state is pushed into the head of the stack and all the subsequent shift and pop transactions will have that state in the top stack symbol. Also a Summary move could be indexed by the starting state of the corresponding chain and can be considered as a Shift move: both of these transaction change the state from the starting to the arrival one maintaining the symbol on top of the stack.

The implementation of the pratically recording is the following: the reachability function of POMC has been modified so that, during the visit of the states of the OPA, every tuple representing the transaction performed is inserted into the `TraceMap` data structure indexed in this way:

- by the current state if the algorithm is performing a Push inside the reachPush subroutine
- by the state at top of the stack if the algorithm is performing a Shift inside the reachShift subroutine, a Pop inside the reachPop subroutine or a Summary inside the reachPush subroutine

When we say that a state indexes a memory space in the `TraceMap` data structure we mean that the memory space is referred by the integer Id of that state, accessible by the function `getId`. The tuples saved representing the transaction are elements of `TraceChunk` type, so they are in the form (move type, current state, stack symbol, future state) where move type is the action performed by the transition, current state is the state from which the transition started, stack symbol is the head of the stack before the transition composed by (qProps, stack state), respectively the atomic propositions active in the current state and the last state pushed, and future state is the state of arrival of the transition.

Let's recall that an element of `TraceMap` is of type `(TraceChunk state, TraceChunk state, TraceChunk state)`, so it's a triple of list of tuple of actions performed. We name these list respectively (*push*, *shift*, *pop*). So when an action is performed it is inserted into `TraceMap` indexed by the right state, being attached to the list of the right category according to the move type:

push for Push move type

shift for Shift and Summary move type (as previously depicted this two moves could be seen similarly)

pop for Pop move type

A tuple is inserted in `TraceMap` only once, there is a check that prevents duplicates.

Example 5.1. Let's have these plausible tuples:

- *(Push, 58, (qProps, 30), 67)*: the move type Push suggests to save in the push `TraceChunk` indexed by the current state 58, so 58 → *((Push, 58, (qProps, 30), 67) + push ,shift,pop)*
- *(Shift, 67, (qProps, 58), 75)*: the move type Shift suggests to save in the shift `TraceChunk` indexed by the state on top of the stack 58, so 58 → *(push ,(Shift, 67, (qProps, 58), 75) + shift,pop)*
- *(Summary, 75, (qProps, 58), 89)*: the move type Push suggests to save in the shift `TraceChunk` indexed by the state on top of the stack 58, so 58 → *(push ,(Shift, 75, (qProps, 58), 89) + shift,pop)*
- *(Pop, 89, (qProps, 58), 34)*: the move type Push suggests to save in the pop `TraceChunk` indexed by the state on top of the stack 58, so 58 → *(push , shift, (Pop, 89, (qProps, 58), 34) + pop)*

The recording phase is followed by the reconstruction that is decribed in the next section.

5.2.3 Reconstruction phase

The reconstruction phase consists of, taken the counterexample processed by the reachability algorithm, substituting all his summary tuples with coherent trace chunks obtained assembling the move tuples saved in `TraceMap` data structure during the OPA's exploration. Indeed the trace returned by the reachability algorithm, that from here on we will call *OPA trace*, is of `TraceId` type (see section 5.2.1) and could contain summary moves. Simply stated, the algorithm, by means of the function `unrollTrace`, scans the OPA trace, transformed into `traceChunk` trace, and tries to solve each summary that it finds one by one, possibly recursively if in turn it contains other summaries. Each summary is resolved by the function `resolveSummary` that, consulting `TraceMap`, tries to find out all the possible corrisponding sequences of tuples attaching them like domino cards. The correspondence happens matching the future state of the previous tuple with the current state of the next one. All the discovered eligible sequences are examined to find if there is one closed i.e. that do not contains other summary tuples. If it's found it's the real substitute of the summary and than it's returned, if it's not every trace is in turn tried to be solved until a closed sequence is achieved.

The procedure's pseudocode is presented below:

Algorithm 3 Function unrollTrace

```
1: function UNROLLTRACE(opaTrace)
2:     tcTrace := traceToChunk (opaTrace)
3:     solvedTrace := browseTrace (tcTrace, $\emptyset$)
4:     return solvedTrace
```

Algorithm 4 Function browseTrace

```
1: function BROWSETRACE(tcTrace, visitedTpl)
2:     realTrace := $\emptyset$
3:     for tuple@(moveType, current_state, stack_symbol, future_state) in tcTrace
4:     if moveType = Summary then
5:         solvedSummary := resolveSummary (tuple, visitedTpl)
6:         if solvedSummary = $\emptyset$ then return $\emptyset$
7:         realTrace := realTrace $\oplus$ solvedSummary
8:     else
9:         realTrace := realTrace $\oplus$ tuple
10:    return realTrace
```

where the operation $\oplus$ represents string concatenation.

`ResolveSummary` finds a set of possible replacement trace chunks of a specific Summary tuple and it returns the first closed one that it finds if present. If it's not present it means that all the trace chunks found contain other Summary tuples that must be solved. `BrowseTrace` is so executed on all the elements of the set of *consistentCombination*. Let's note so that the research of the solution is *depth-first*: the first element

Algorithm 5 Function resolveSummary

```
function RESOLVESUMMARY(tuple, visitedTpl)
    (moveType, current_state, stack_symbol, future_state) := tuple
    if current_state $\in$ visitedTpl then return $\emptyset$
    else
        push, shift, pop) := getRecordedTransition (current_state, TraceMap)
        allCombination := findCompletion ((push, shift, pop),1)
        consistentCombination := filterConsistent (allCombination)
        if consistentCombination = $\emptyset$ then return $\emptyset$
        else
            closedTrace := takeClosedTrace (consistentCombination)
            if closedTrace = $\emptyset$ then return $\emptyset$
            else
                traces = $\emptyset$
                for traces in consistentCombination
                    traces = traces $\cup$ browseTrace (trace, tuple $\cup$ visitedTpl)
                return shortestTrace (traces)
```

of *consistentCombination* is expanded with all his children and grandchildren and so on exploring possible chunk trace until the presence of a solution or the absence of a valid one is assessed, then it's the turn of the second one, then the third…
The algorithm indeed goes always deeper and interrupts only if it finds a valid replacement trace, if it tries to resolve a summary already encountered (going ahead exploring would mean an infinite loop) or if it finds an empty set of consistent possible replacement traces. When all the elements of *consistentCombination* have been explored the shortest trace chunk is taken as valid trace. The function `filterConsistent` eliminates from the rough set of candidate trace chunks those that aren't consistent, namely those that don't respect the following rules:

- every tuple must have a current state equal to the future state of the previous tuple
- if a tuple has moveType = Push its current state must be equal to the state of stack symbol of the successor
- if a tuple has moveType = Shift or moveType = Summary its state of stack symbol must be equal to the state of stack symbol of the successor

Let's now explain how the construction of all the possible trace chunks is performed. We have to compose in a consistent way the transitions that we have recorded during the OPA exploration. Every chain start with a push move and terminate with a pop move. The most simple chain that we can have is of the form:
Push $\rightarrow$ (Shift | Summary)* $\rightarrow$ Pop
with only an initial push and a final pop. More complex chains can have nested Push-Pop couples in the form:
Push^n $\rightarrow$ (Shift | Summary)* $\rightarrow$ Pop^n or also
Push^n $\rightarrow$ (Shift | Summary)* $\rightarrow$ Pop$_1$ $\rightarrow$ (Shift | Summary)* $\rightarrow$ Pop$_2$ $\rightarrow \cdots \rightarrow$ (Shift | Summary)* $\rightarrow$ pop$_n$.
While the simple chain can be closed consulting `TraceMap` only in the context of the current state of the initial push, for the other chain the procedure is more complex. To

construct the possible trace chunk, tuples recorded during exploration are attached like domino cards: a tuple can be attached to another one if its current state is equal to the future state of the latter. So starting from the initial push move different trace chunks are created with the compatible recorded tuples extracted from `TraceMap`. Being the OPA non deterministic a tuple can be associated to more subsequent tuples. All the moves combinations discovered are closed by a pop, found in `TraceMap` in the context of the corresponding push state. The procedure to construct all the possible trace chunks is implemented on progressive levels: before are found trace chunks of the form Push $\rightarrow$ (Shift | Summary)* $\rightarrow$ Pop (level 1), if there isn't anyone consistent it proceeds to find that of the form Push$_1$ $\rightarrow$ Push$_2$ $\rightarrow$ (Shift | Summary)* $\rightarrow$ Pop$_1$ $\rightarrow$ Pop$_2$ (level 2), then Push$_1$ $\rightarrow$ Push$_2$ $\rightarrow$ Push$_3$ $\rightarrow$ (Shift | Summary)* $\rightarrow$ Pop$_1$ $\rightarrow$ (Shift | Summary)* $\rightarrow$ Pop$_2$ $\rightarrow$ (Shift | Summary)* $\rightarrow$ Pop$_3$ (level 3) and so on. To every level corresponds a number of pushes and pops. In this way we evaluate before the most simple trace chunks constituted by one push and then eventually the more complex ones. This allows us to save memory and computational power due to the fact that most of the summaries to solve are composed by a simple chain, so exploring only at level 1 is really more convenient than creating all the possible combinations in one shot.

Here follows the pseudocode of the procedure:

Algorithm 6 Function findCompletion

```
1: function findCompletion((push, shift, pop), level)
2:     pushSet := $\emptyset$
3:     for pushTuple in push
4:         newTrChunk := completePush (pushTuple, level)
5:         pushSet = pushSet $\cup$ newTrChunk
6:     shiftSet := $\emptyset$
7:     for pushTrace in pushSet
8:         newTrChunk := completeShift (pushTrace)
9:         shiftSet := shiftSet $\cup$ newTrChunk
10:    popSet := $\emptyset$
11:    for shiftTrace in shiftSet
12:        newTrChunk := completePop (shiftTrace)
13:        popSet := popSet $\cup$ newTrChunk
14:    consistentCombination := filterConsistent (popSet)
15:    if consistentCombination = $\emptyset$ then
16:        return findCompletion ((push,shift,pop), level+1)
17:    else
18:        return consistentCombination
```

Algorithm 7 Function completePush

```
function completePush(pushTuple, level)
    if level $= 1$ then return pushTuple
    else
        (moveType, current_state, stack_symbol, future_state) := pushTuple
        (push, _, _) := getRecordedTransition (future_state, TraceMap)
        if push $= \emptyset$ then return pushTuple
        else
            pushSet := $\emptyset$
            for nextpushTuple in push
                newTrChunk := completePush (nextpushTuple, level $-1$)
                pushSet = pushSet $\cup$ (pushTuple $\oplus$ newTrChunk)
            return pushTuple
```

Algorithm 8 Function completeShift

```
function completeShift(pushTrace)
    (moveType, push_current_state, push_stack_symbol, push_future_state) := last (pushTrace)
    (_, shift, _) := getRecordedTransition (push_current_state, TraceMap)
    if shift $= \emptyset$ then return pushTrace
    else
        shiftSet := $\emptyset$
        for shiftTuple in shift
            newTrChunk := completeSingleShift (shiftTuple, shift)
            shiftSet := shiftSet $\cup$ (shiftTuple $\oplus$ newTrChunk)
    return shiftSet
function completeSingleShift(shiftTuple, shift)
    (_, _, _, shiftTuple_future_state) := shiftTuple
    matchingTuples := searchTupleByState (shift, shiftTuple_future_state)
    if matchingTuples $= \emptyset$ then return shiftTuple
    else
        shiftSet := $\emptyset$
        for shiftTuple in matchingTuples
            newTrChunk := completeSingleShift (shiftTuple, shift)
            shiftSet := shiftSet $\cup$ (shiftTuple $\oplus$ newTrChunk)
    return shiftSet
```

Algorithm 9 Function completePop

```
function completePop(shiftTrace)
    pushAcc := reverse (takePush (shiftTrace))
    (_ , push_current_state, _, _) := head (pushAcc)
    (_, _, _, shift_future_state) := last (shiftTrace)
    (_, _, pops) := getRecordedTransition (push_current_state, TraceMap)
    matchingTuples := searchTupleByState (pops, shift_future_state)
    popSet := $\emptyset$
    for popTuple in matchingTuples
        newTrChunk := completePopFindShift (popTuple, tail (pushAcc))
        popSet := popSet $\cup$ (shiftTrace $\oplus$ newTrChunk)
    return popSet
function completePopFindShift(popTuple, pushAcc)
    if pushAcc = $\emptyset$ then return popTuple
    else
        popShift := completeShift(popTuple)
        popShiftSet := $\emptyset$
        for popShiftTrace in popShift
            newTrChunk := findShiftPopComb (popShiftTrace, pushAcc)
            popShiftSet := popShiftSet $\cup$ newTrChunk
        return popShiftSet
function findShiftPopComb(popShiftTrace, pushAcc)
    (_ , push_current_state, _, _) := head (pushAcc)
    (_, _, _, shift_future_state) := last (shiftTrace)
    (_, _, pops) := getRecordedTransition (push_current_state, TraceMap)
    matchingTuples := searchTupleByState (pops, shift_future_state)
    if matchingTuples = $\emptyset$ then return $\emptyset$
    else
        popSet := $\emptyset$
        for popTuple in matchingTuples
            newTrChunk := completePopFindShift (popTuple, tail (pushAcc))
            popSet := popSet $\cup$ (popShiftTrace $\oplus$ newTrChunk)
        return popSet
```

The function `searchtuplebystate`, given a set of `traceChunk` elements and a future state, finds all the tuples within the set that have the current state corresponding to the future state.

So summarizing, function `completePop` finds the first part of Summary trace aggregating push tuples working on a level system: on level 1 will return only piece of trace with one push (push), on level 2 with the combination of two pushes (push-push) and so on. Every push tuple will be extracted from `TraceMap` indexed by the forward state of the previous considered tuple. Function `completeShift` takes as input a first piece of trace composed by only push moves and the shift set obtained consulting `TraceMap` indexed by the current state of the last push (the most internal). It tries, taken a trace, to add at every step a compatible shift tuple until no more additions are possible. `CompleteShift` returns a piece of trace composed by push and shift moves. Function `completePop` closes the game finding for every partial trace the corresponding pop moves. Every pop move to find has a corresponding push move in

```
opa:
  initials = 0;
  finals = (2 8 10);
```

```
deltaPush =
    (0,  (call pa),    1),
    (1,  (call pb),    2),
    (2,  (call pc),    3),
    (2,  (han),        4),
    (3,  (call pb),    2),
    (4,  (call pc),    3),
    (7,  (exc eb),     8),
    (9,  (call perr),  10),
    (10, (call perr),  10),
    (15, (han),        19),
    (19, (call pc),    3),
    (23, (call perr),  10),
    (8,  (call),       8);
```

```
deltaPop =
    (3,   2,  5),
    (3,   4,  9),
    (3,  19, 20),
    (5,   1,  6),
    (5,   3, 18),
    (6,   0,  7),
    (8,   7,  8),
    (9,   2,  9),
    (11, 10, 12),
    (11,  9, 13),
    (11, 23, 21),
    (14,  1, 15),
    (14,  3, 16),
    (17,  4, 17),
    (17,  2, 13),
    (17, 19, 21),
    (18,  2,  5),
    (18,  4,  9),
    (18, 19, 20),
    (22,  0,  8),
    (23, 15, 23),
    (8,   8,  8);
```

```
deltaShift =
    (9,  (exc),      9),
    (10, (ret perr), 11),
    (12, (ret perr), 11),
    (13, (ret pb),   14),
    (16, (ret pc),   17),
    (20, (exc),      23),
    (21, (ret pa),   22),
    (8,  (ret),      8);
```

Figure 5.2: Explicit automaton of 32-generic-medium example

the partial trace, so they must be the same number, and every pop move is extracted from `TraceMap` in the context of the corresponding push move. `CompletePop` includes the function `completeShift` to discover the shift moves interleaved between pop moves, for example in the situation Push $\rightarrow$ Push $\rightarrow$ Shift $\rightarrow$ Pop $\rightarrow$ Shift $\rightarrow$ Shift $\rightarrow$ Pop.

5.2.4 A practical example

Let's use POMC on *32-generic-medium.pomc* example to verify a formula. The automaton expressed in explicit way is shown in figure 5.2. The POTL formula to be checked is the following:

$$\square((\mathbf{call} \land pa) \Rightarrow \neg\,(\bigcirc^u\,\mathbf{exc} \lor \chi_F^u\,\mathbf{exc}))$$

Let's launch the console command: stack exec – pomc 32-generic-medium.pomc –finite

Model Checking
Formula: G (("call" And "pa") –> (((PNu "exc") Or (XNu "exc"))))
Input OPA state count: 24
Result: False
Counterexample: $[(0, [''call'', ''pa'']), (1, [''call'']), (2, [''...'']), (7, [''exc'']), (8, [''\#''])]$
Elapsed time: 253.4 ms

We see that our trace is $[(0, [''call'', ''pa'']), (1, [''call'']), (2, [''...'']), (7, [''exc'']), (8, [''\#''])]$ and there is Summary = $(2, [''\ldots''])$ to solve. The support trace of counterexample is the following:

(Push,StateId getId = 26, getState = MCState 0,Nothing,StateId getId = 2307, getState = MCState 1),
(Push,StateId getId = 2307, getState = MCState 1,(EncodedAtom [12]34,StateId getId = 26, getState = MCState 0),StateId getId = 2415, getState = MCState 2),
(Summary,StateId getId = 2415, getState = MCState 2,(EncodedAtom [12]2,StateId getId = 2307, getState = MCState 1),StateId getId = 3078, getState = MCState 5),
(Pop,StateId getId = 3078, getState = MCState 5,(EncodedAtom [12]2,StateId getId = 2307, getState = MCState 1),StateId getId = 4397, getState = MCState 6),
(Pop,StateId getId = 4397, getState = MCState 6,(EncodedAtom [12]34,StateId getId = 26, getState = MCState 0),StateId getId = 4402, getState = MCState 7),
(Push,StateId getId = 4402, getState = MCState 7,Nothing,StateId getId = 4403, getState = MCState 8),
(Pop,StateId getId = 4403, getState = MCState 8,(EncodedAtom [12]16,StateId getId = 4402, getState = MCState 7),StateId getId = 4451, getState = MCState 8),
(Found,StateId getId = 4451, getState = MCState 8,Nothing,StateId getId = 4451, getState = MCState 8)

where the tuples in this example are represented in the usual way: (moveType, id of current state, (encoded propositions, state of stack), id of future state)

The id of the summary's state is 2415. The scan of the trace by `browseTrace` spots the summary tuple and launches the function `resolveSummary` on it. `ResolveSummary` first checks that the state of the summary has been already visited. In this case not because it's the first time we launch `resolveSummary` and the set of bad tuples is void. Now the algorithm consults the `TraceMap` with id = 2415 to find all the push, shift, summary and pop tuples to create all the possible replacement traces. Here is the list of all pushes recorded during the OPA exploration performed from state 2415, each one of these is a possible starting point for a solving trace:

(Push,StateId getId = 2415, getState = MCState 2, (EncodedAtom [12]2,StateId getId = 2307, getState = MCState 1),StateId getId = 2471, getState = MCState 3)
(Push,StateId getId = 2415, getState = MCState 2, (EncodedAtom [12]2,StateId getId = 2307, getState = MCState 1),StateId getId = 2470, getState = MCState 3)
(Push,StateId getId = 2415, getState = MCState 2, (EncodedAtom [12]2,StateId getId = 2307, getState = MCState 1),StateId getId = 2469, getState = MCState 3)

Subsequently `findCompletion` is launched, the procedure combines all the tuples to obtain a set of consistent traces. The procedure return only one possible replacement trace after findCompletion and filtering , that is shown here:

Trace 1: **(Push,StateId getId = 2415, getState = MCState 2,(EncodedAtom [12]2,StateId getId = 2307, getState = MCState 1),StateId getId = 2469, getState = MCState 3), (Summary,StateId getId = 2469, getState = MCState 3,(EncodedAtom [12]2,StateId getId = 2415, getState = MCState 2),StateId getId = 3102, getState = MCState 18), (Pop,StateId getId = 3102, getState = MCState 18,(EncodedAtom [12]2,StateId getId = 2415, getState = MCState 2),StateId getId = 3078, getState = MCState 5)**

The trace is checked: it is not a closed trace i.e. it contains other Summary tuples. It must be solved too. `browseTrace` is launched on it again and this time we obtain the following two coherent traces:

Trace 1.1: **(Push,StateId getId = 2469, getState = MCState 3,(EncodedAtom [12]2,StateId getId = 2415, getState = MCState 2),StateId getId = 2417, getState = MCState 2),(Summary,StateId getId = 2417, getState = MCState 2,(EncodedAtom [12]2,StateId getId = 2469, getState = MCState 3),StateId getId = 3078, getState = MCState 5),(Pop,StateId getId = 3078, getState = MCState 5,Just (EncodedAtom [12]2,StateId getId = 2469, getState = MCState 3),StateId getId = 3102, getState = MCState 18)**
Trace 1.2: **(Push,StateId getId = 2469, getState = MCState 3,(EncodedAtom [12]2,StateId getId = 2415, getState = MCState 2),StateId getId = 2415, getState = MCState 2),(Summary,StateId getId = 2415, getState = MCState 2,(EncodedAtom [12]2,StateId getId = 2469, getState = MCState 3),StateId getId = 3078, getState = MCState 5),(Pop,StateId getId = 3078, getState = MCState 5,(EncodedAtom [12]2,StateId getId = 2469, getState = MCState 3),StateId getId = 3102, getState = MCState 18)**

Now both of the traces are checked: none of them is closed i.e. they both contains other Summary tuples. They must be solved too. Due to the fact that the algorithm proceedes in a depth-first manner, the first trace is tried to be solved with eventually its subtraces if they contains summary tuples, and then the second trace. `BrowseTrace` is launched on the Trace 1.1 to spot the summary that has state id = 2417 and then `resolveSummary`. The set of already visited state is updated adding state = 2415 (the external summary). After `findCompletion` we obtain the following two traces:

Trace 1.1.1: **(Push,StateId getId = 2469, getState = MCState 3,(EncodedAtom [12]2,StateId getId = 2415, getState = MCState 2),StateId getId = 2417, getState = MCState 2),(Summary,StateId getId = 2417, getState = MCState 2,(EncodedAtom [12]2,StateId getId = 2469, getState = MCState 3),StateId getId = 3078, getState = MCState 5),(Pop,StateId getId = 3078, getState = MCState 5,(EncodedAtom [12]2,StateId getId = 2469, getState = MCState 3),StateId getId = 3102, getState = MCState 18)**
Trace 1.1.2: **(Push,StateId getId = 2469, getState = MCState 3,(EncodedAtom [12]2,StateId getId = 2415, getState = MCState 2),StateId getId = 2415, getState = MCState 2),(Summary,StateId getId = 2415, getState = MCState 2,(EncodedAtom [12]2,StateId getId = 2469, getState = MCState 3),StateId getId = 3078, getState = MCState 5),(Pop,StateId getId = 3078, getState = MCState 5,(EncodedAtom [12]2,StateId getId = 2469, getState = MCState 3),StateId getId = 3102, getState = MCState 18)**

Both the traces contain summary tuples, so the procedure goes on: Trace 1.1.1 is resolved. After `findCompletion` we find only one eligible trace:

Trace 1.1.1.1: **(Push,StateId getId = 2417, getState = MCState 2,(EncodedAtom [12]2,StateId getId = 2469, getState = MCState 3),StateId getId = 2553, getState = MCState 3),(Pop,StateId getId = 2553, getState = MCState 3,Just (EncodedAtom [12]2,StateId getId = 2417, getState = MCState 2),StateId getId = 3078, getState = MCState 5)**

It's consistent and it doesn't contain summaries, so it's returned. Although we have probably found the right replacement trace, the procedure doesn't stop here because there is pending the resolution of Trace 1.1.2 and then of Trace 1.2. `browseTrace` is launched on Trace 1.1.2 and then `resolveSummary`. Firstly it checks that the state of the summary is a new state never visited. The state id of the summary tuple is 2415, the same of the state id of original trace's summary. The function doesn't go on and returns void. Also Trace 1.2 has a summary with state id equal to 2415. For the same reason void is returned. There are no more traces to examine and from the procedure we obtained a coherent trace that is Trace 1.1.1.1. It is inserted into the original trace replacing the summary. The new trace is returned and the procedure stops.

New trace: **(Push,StateId getId = 26, getState = MCState 0,Nothing,StateId getId = 2307, getState = MCState 1),(Push,StateId getId = 2307, getState = MCState 1,(EncodedAtom [12]34,StateId getId = 26, getState = MCState 0),StateId getId = 2415, getState = MCState 2),(Push,StateId getId = 2469, getState = MCState 3,(EncodedAtom [12]2,StateId getId = 2415, getState = MCState 2),StateId getId = 2417, getState = MCState 2),(Push,StateId getId = 2469, getState = MCState 3,(EncodedAtom [12]2,StateId getId = 2415, getState = MCState 2),StateId getId = 2417, getState = MCState 2),(Push,StateId getId = 2417, getState = MCState 2,(EncodedAtom [12]2,StateId getId = 2469, getState = MCState 3),StateId getId = 2553, getState = MCState 3),(Pop,StateId getId = 2553, getState = MCState 3,Just (EncodedAtom [12]2,StateId getId = 2417, getState = MCState 2),StateId getId = 3078, getState = MCState 5),(Pop,StateId getId = 3078, getState = MC-State 5,(EncodedAtom [12]2,StateId getId = 2469, getState = MCState 3),StateId getId = 3102, getState = MCState 18),(Pop,StateId getId = 3078, getState = MC-State 5,Just (EncodedAtom [12]2,StateId getId = 2469, getState = MCState 3),StateId getId = 3102, getState = MCState 18),(Pop,StateId getId = 3078, getState = MC-State 5,(EncodedAtom [12]2,StateId getId = 2307, getState = MCState 1),StateId getId = 4397, getState = MCState 6),(Pop,StateId getId = 4397, getState = MC-State 6,(EncodedAtom [12]34,StateId getId = 26, getState = MCState 0),StateId getId = 4402, getState = MCState 7),(Push,StateId getId = 4402, getState = MC-State 7,Nothing,StateId getId = 4403, getState = MCState 8),(Pop,StateId getId = 4403, getState = MCState 8,(EncodedAtom [12]16,StateId getId = 4402, getState = MCState 7),StateId getId = 4451, getState = MCState 8),(Found,StateId getId = 4451, getState = MCState 8,Nothing,StateId getId = 4451, getState = MCState 8)**

that is presented to the final user in this way:
Counterexample: $[(0, ["call", "pa"]), (1, ["call"]), (2, ["call"]), (3, ["call"]), (2, ["call"]), (7, ["exc"]), (8, ["\#"])]$

Chapter 6

Experimental results

6.1 Experimental Evaluation

The new functionality resolving summaries has been tried on a benchmark formulated for the test of the original POMC tool. The tests have been executed on laptop with processor 2.7 GHz AMD® A6-4400m, 8 GiB of RAM running Ubuntu 22.04.2 LTS. Here follows the table with the comparisons, the part of trace found by the new algorithm is highlighted in bold. We selected only tests whose results are false and which contain summaries in their counterexample traces.

Generic-medium series

Generic-medium benchmark is a set of 44 tests evaluated on the same medium-sized OPA on different formulas. The explicit OPA it is shown in figure 5.2: it's the same used for the example in Subsection 5.2.4. The formulas checked for every test are shown in the numbered list below, while in table 6.1 is shown for every test the execution time and trace before and after the summary resolution.

1. $\chi_F^d \, perr$
2. $\mathrm{T} \, \mathcal{U}_\chi^d \, \mathbf{exc}$
3. $\bigcirc^d (\bigcirc^d (\mathrm{T} \, \mathcal{U}_\chi^d \, \mathbf{exc}))$
4. $\neg \, (\mathrm{T} \, \mathcal{U}_\chi^d \, \mathbf{exc})$
5. $\neg \, (\mathrm{T} \, \mathcal{U}_\chi^u \, \mathbf{exc})$
6. $\neg \, (\chi_F^d \, (\bigcirc^d \, (\circleddash^u \; \mathbf{call})))$
7. $\diamond \, (\bigcirc_H^d \, pb)$
8. $\diamond \, (pa \wedge (\mathbf{call} \, \mathcal{U}_H^d \, pc))$
9. $\diamond \, (pc \wedge (\mathbf{call} \, \mathcal{S}_H^d \, pa))$
10. $\square \, ((pc \wedge (\chi_F^u \, \mathbf{exc})) \Rightarrow ((\neg \, pa) \, \mathcal{S}_H^d \, pb))$
11. $\square \, ((\mathbf{call} \wedge pa) \Rightarrow \neg \, (\bigcirc^u \, \mathbf{exc} \vee \chi_F^u \, \mathbf{exc}))$

12. $\square\,((\mathbf{call} \wedge pb \wedge (\mathrm{T}\,\mathcal{S}^d_H\,(\mathbf{call} \wedge pa))) \Rightarrow (\bigcirc^u\ \mathbf{exc} \vee \chi^u_F\,\mathbf{exc}))$

13. $\square\,(\mathbf{han} \Rightarrow \chi^u_F\,\mathbf{ret})$

14. $\mathrm{T}\,\mathcal{U}^d_H\,\mathbf{exc}$

15. $\bigcirc^d\,(\bigcirc^d\,(\bigcirc^d\,(\mathrm{T}\,\mathcal{U}^u_\chi\,\mathbf{exc}$

16. $\square\,(\mathbf{call} \wedge pc \Rightarrow (\mathrm{T}\,\mathcal{U}^u_\chi\,(\mathbf{exc} \wedge \chi^d_P\,\mathbf{han})))$

17. $\bigcirc^d\,(\bigcirc^d\,((\mathbf{call} \vee \mathbf{exc})\mathcal{U}^u_\chi\,\mathbf{ret}))$

We can notice analyzing summary solving POMC's times that they are proportional to those of original POMC. This is due to the fact that in original POMC times grow obviously proportionally to the size of state-space, but primarily with the capillarity of the exploration of the state-space and so in new POMC: more state are visited more transitions are registered in `TraceMap` and have to be combined to find the true summary replacement. Anyway for most of the tests in this set, which they share the same medium-sized OPA, exploration is not so spread in the state-space and so also trace reconstruction is easy with low times. Most of the times summary solving needs to aggregate tuples at level 1. Only generic-medium-14 requires level 2 of aggregation but time anyway doesn't rise significantly. Generic-medium-19 and generic-medium-20 are an exception: in this two tests state-space exploration is very large, so times bump up a lot. Summary solving can take up to seven times the property check because of the huge amount of transitions' tuples registered: they could be more than two thousand for state and so combining them all to find the right trace chunk it's very costly.

Generic-small series

Generic-medium benchmark is a set of 44 tests evaluated on the same small-sized OPA on different formulas. The explicit OPA it is shown in figure 6.1. The formulas checked for every test are shown in the numbered list below, while in table 6.2 is shown for every test the execution time and trace before and after the summary resolution.

```
opa:
    initials = 0;
    finals = (4 10);


deltaPop =
     (4, 2, 4),
     (4, 3, 4),
     (4, 4, 4),
     (5, 1, 6),
     (7, 6, 8),
     (7, 8, 9),
     (11, 0, 10),
     (10, 10, 10);
```

```
deltaPush =
    (0,  (call pa),   1),
    (1,  (han),       2),
    (2,  (call pb),   3),
    (3,  (call pc),   4),
    (4,  (call pc),   4),
    (6,  (call perr), 7),
    (8,  (call perr), 7),
    (10, (call),     10);


deltaShift =
     (4, (exc),       5),
     (7, (ret perr),  7),
     (9, (ret pa),   11),
     (10, (ret),     10);
```

Figure 6.1: Explicit automaton of generic-small set

1. $\neg\,(\mathrm{T}\,\mathcal{U}^{d}_{\chi}\,\mathbf{exc})$
2. $\mathrm{T}\,\mathcal{U}^{d}_{\chi}\,\mathbf{exc}$
3. $\diamond\,(pa \wedge (\mathbf{call}\,\mathcal{U}^{d}_{H}\,pc))$
4. $\diamond\,(pc \wedge (\mathbf{call}\,\mathcal{S}^{d}_{H}\,pa))$
5. $\square\,(\mathbf{call} \Rightarrow \chi^{d}_{F}\,\mathbf{ret})$
6. $\square\,(\mathbf{han} \Rightarrow \chi^{u}_{F}\,\mathbf{ret})$
7. $\mathrm{T}\,\mathcal{U}^{u}_{\chi}\,\mathbf{exc}$

Here the results are coherent with the previous ones: times are low and tests are easy-solvable due to the small size of the OPA.

Table 6.1: Results of the evaluation on the benchmark *generic-medium*.

#	Benchmark name	Original POMC trace	Time (ms)	New POMC trace	Time (ms)
1	2-generic-medium	[(0,["call"]),(1,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(9,["exc"]),(9,["call"]),(10,["call"]),(10,["call"]),(10,["ret"]),(12,["ret"]),(13,["ret"]),(16,["ret"]),(13,["ret"]),(15,["han"]),(19,["call"]),(20,["exc"]),(23,["call"]),(10,["..."]),(12,["ret"]),(21,["ret"]),(8,["#"])]	69,45	[(0,["call"]),(1,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(9,["exc"]),(9,["call"]),(10,["call"]),(10,["call"]),(10,["ret"]),(12,["ret"]),(13,["ret"]),(16,["ret"]),(13,["ret"]),(15,["han"]),(19,["call"]),(20,["exc"]),(23,["call"]),**(10,["call"]),(10,["ret"])**,(12,["ret"]),(21,["ret"]),(8,["#"])]	93,32
2	6-generic-medium	[(0,["call"]),(1,["..."]),(7,["exc"]),(8,["#"])]	2,977	[(0,["call"]),**(1,["call"]),(2,["call"])**,(7,["exc"]),(8,["#"])]	6,186
3	7-generic medium	[(0,["call"]),(1,["..."]),(7,["exc"]),(8,["#"])]	5,186	[(0,["call"]),**(1,["call"]),(2,["call"])**,(7,["exc"]),(8,["#"])]	11,08
4	8-generic-medium	(0,["call"]),(1,["call"]),(2,["..."]),(9,["..."]),(13,["ret"]),(15,["han"]),(19,["call"]),(3,["..."]),(20,["exc"]),(23,["call"]),(10,["..."]),(12,["ret"]),(21,["ret"]),(8,["#"])]	39,11	[(0,["call"]),(1,["call"]),**(2,["han"]),(4,["call"]),(9,["exc"]),(9,["call"]),(10,["ret"])**,(13,["ret"]),(15,["han"]),(19,["call"]),**(3,["call"]),(2,["call"])**,(20,["exc"]),(23,["call"]),**(10,["call"]),(10,["ret"])**,(12,["ret"]),(21,["ret"]),(8,["#"])]	63,00
5	9-generic-medium	[(0,["call"]),(1,["call"]),(2,["..."]),(7,["exc"]),(8,["#"])]	60,02	[(0,["call"]),(1,["call"]),**(2,["call"]),(3,["call"]),(2,["call"])**,(7,["exc"]),(8,["#"])]	68,93
6	14-generic-medium	[(0,["call"]),(1,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(9,["exc"]),(9,["call"]),(10,["call"]),(10,["call"]),(10,["call"]),(10,["call"]),(10,["call"]),(10,["call"]),(10,["ret"]),(12,["ret"]),(12,["ret"]),(13,["ret"]),(16,["ret"]),(13,["ret"]),(15,["han"]),(19,["call"]),(3,["..."]),(20,["exc"]),(23,["call"]),(10,["ret"]),(21,["ret"]),(8,["#"])]	36,44	[(0,["call"]),(1,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(3,["call"]),(2,["call"]),(3,["call"]),(2,["call"]),(9,["exc"]),(9,["call"]),(10,["call"]),(10,["call"]),(10,["call"]),(10,["call"]),(10,["call"]),(10,["call"]),(10,["ret"]),(12,["ret"]),(12,["ret"]),(13,["ret"]),(16,["ret"]),(13,["ret"]),(15,["han"]),(19,["call"]),**(3,["call"]),(2,["call"])**,(20,["exc"]),(23,["call"]),(10,["ret"]),(21,["ret"]),(8,["#"])]	50,74

7	16-generic-medium	[(0,["call"]),(1,["call","pb"]),(2,["call"]),(3,["call","pb"]),(2,["han"]),(4,["call"]),(9,["exc"]),(9,["call"]),(10,["call"]),(10,["call"]),(10,["ret"]),(12,["ret"]),(13,["pb","ret"]),(16,["ret"]),(13,["pb","ret"]),(15,["han"]),(19,["call"]),(20,["exc"]),(23,["call"]),(10,["..."]),(12,["ret"]),(21,["ret"]),(8,["#"])]	17,05	[(0,["call"]),(1,["call","pb"]),(2,["call"]),(3,["call","pb"]),(2,["han"]),(4,["call"]),(9,["exc"]),(9,["call"]),(10,["call"]),(10,["call"]),(10,["ret"]),(12,["ret"]),(13,["pb","ret"]),(16,["ret"]),(13,["pb","ret"]),(15,["han"]),(19,["call"]),(20,["exc"]),(23,["call"]),**(10,["call"]),(10,["ret"])**,(12,["ret"]),(21,["ret"]),(8,["#"])]	63,00
8	18-generic-medium	[(0,["call","pa"]),(1,["call"]),(2,["..."]),(7,["exc"]),(8,["#"])]	1943	[(0,["call","pa"]),(1,["call"]),**(2,["call","pc"])**,(7,["exc"]),(8,["#"])]	2372
9	19-generic-medium	[(0,["call","pa"]),(1,["call"]),(2,["..."]),(13,["ret"]),(15,["han"]),(19,["..."]),(20,["exc"]),(23,["call"]),(10,["ret"]),(21,["pa","ret"]),(8,["#"])]	2006	[(0,["call","pa"]),(1,["call"]),**(2,["call","pc"]),(3,["call"]),(2,["han"]),(4,["call","pc"]),(9,["exc"]),(9,["call"]),(10,["ret"]),(13,["ret"]),(16,["pc","ret"])**,(13,["ret"]),(15,["han"]),**(19,["call","pc"])**,(20,["exc"]),(23,["call"]),(10,["ret"]),(21,["pa","ret"]),(8,["#"])]	15980
10	20-generic-medium	[(0,["call","pa"]),(1,["call","pb"]),(2,["..."]),(13,["pb","ret"]),(15,["han"]),(19,["..."]),(20,["exc"]),(23,["call"]),(10,["ret"]),(21,["pa","ret"]),(8,["#"])]	64250	[(0,["call","pa"]),(1,["call","pb"]),**(2,["call","pc"]),(3,["call","pb"]),(2,["han"]),(4,["call","pc"]),(3,["call","pb"]),(2,["call","pc"]),(9,["exc"]),(9,["call"]),(10,["ret"])**,(13,["pb","ret"]),(16,["pc","ret"]),(13,["pb","ret"]),(15,["han"]),**(19,["call","pc"])**,(20,["exc"]),(23,["call"]),(10,["ret"]),(21,["pa","ret"]),(8,["#"])]	332700
11	32-generic-medium	[(0,["call","pa"]),(1,["call"]),(2,["..."]),(7,["exc"]),(8,["#"])]	179,7	[(0,["call","pa"]),(1,["call"]),**(2,["call"]),(3,["call"]),(2,["call"])**,(7,["exc"]),(8,["#"])]	217,8
12	34-generic-medium	[(0,["call","pa"]),(1,["call","pb"]),(2,["call"]),(3,["call","pb"]),(2,["call"]),(3,["call","pb"]),(2,["call"]),(3,["call","pb"]),(2,["han"]),(4,["call"]),(3,["call","pb"]),(2,["han"]),(4,["..."]),(9,["exc"]),(9,["call"]),(10,["..."]),(12,["ret"]),(13,["pb","ret"]),(16,["ret"]),(13,["pb","ret"]),(15,["han"]),(19,["call"]),(3,["..."]),(20,["exc"]),(23,["call"]),(10,["..."]),(12,["ret"]),(21,["pa","ret"]),(8,["#"])]	165,6	[(0,["call","pa"]),(1,["call","pb"]),(2,["call"]),(3,["call","pb"]),(2,["call"]),(3,["call","pb"]),(2,["call"]),(3,["call","pb"]),(2,["han"]),**(4,["call"]),(3,["call","pb"]),(2,["han"])**,(4,["call"]),(3,["call","pb"]),(2,["call"]),(9,["exc"]),(9,["call"]),**(10,["call"]),(10,["ret"])**,(12,["ret"]),(13,["pb","ret"]),(16,["ret"]),(13,["pb","ret"]),(15,["han"]),(19,["call"]),**(3,["call","pb"]),(2,["call"])**,(20,["exc"]),(23,["call"]),**(10,["call"]),(10,["ret"])**,(12,["ret"]),(21,["pa","ret"]),(8,["#"])]	245,3
13	36-generic-medium	[(0,["call"]),(1,["call"]),(2,["..."]),(9,["..."]),(13,["ret"]),(15,["han"]),(19,["..."]),(20,["exc"]),(23,["call"]),(10,["ret"]),(21,["ret"]),(8,["#"])]	56,39	[(0,["call"]),(1,["call"]),**(2,["han"]),(4,["call"]),(9,["exc"]),(9,["call"]),(10,["ret"])**,(13,["ret"]),(15,["han"]),**(19,["call"])**,(20,["exc"]),(23,["call"]),(10,["ret"]),(21,["ret"]),(8,["#"])]	72,96
14	37-generic-medium	[(0,["call"]),(1,["call"]),(2,["..."]),(13,["ret"]),(15,["han"]),(19,["call"]),(3,["..."]),(20,["exc"]),(23,["call"]),(10,["..."]),(12,["ret"]),(21,["ret"]),(8,["#"])]	42,74	[(0,["call"]),(1,["call"]),**(2,["call"]),(3,["call"]),(2,["han"]),(4,["call"]),(9,["exc"]),(9,["call"]),(10,["ret"]),(13,["ret"]),(16,["ret"])**,(13,["ret"]),(15,["han"]),(19,["call"]),**(3,["call"]),(2,["call"])**,(20,["exc"]),(23,["call"]),**(10,["call"]),(10,["ret"])**,(12,["ret"]),(21,["ret"]),(8,["#"])]	91,32

15	39-generic-medium	[(0,["call"]),(1,["call"]),(2,["..."]),(7,["exc"]),(8,["#"])]	798,9	[(0,["call"]),(1,["call"]), **(2,["call"])**,(7,["exc"]),(8,["#"])] 71,486	883,5
16	40-generic-medium	[(0,["call"]),(1,["call"]),(2,["..."]),(7,["exc"]),(8,["#"])]	255,7	[(0,["call"]),(1,["call"]), **(2,["call","pc"])**,(7,["exc"]),(8,["#"])]	323,1
17	43-generic-medium	[(0,["call"]),(1,["call"]),(2,["..."]),(7,["exc"]),(8,["#"])]	224,3	[(0,["call"]),(1,["call"]), **(2,["call"])**,(7,["exc"]),(8,["#"])]	254,6

Table 6.2: Results of the evaluation on the benchmark *generic-small*

#	Benchmark name	Original POMC trace	Time (ms)	New POMC trace	Time (ms)
1	7-generic-small	[(0,["call"]),(1,["han"]),(2,["call"]),(3,["..."]),(4,["exc"]),(6,["call"]),(7,["ret"]),(8,["call"]),(7,["ret"]),(9,["ret"]),(10,["#"])]	7,404	[(0,["call"]),(1,["han"]),(2,["call"]), **(3,["call"]),(4,["exc"])**, (6,["call"]),(7,["ret"]),(8,["call"]),(7,["ret"]),(9,["ret"]),(10,["#"])]	13,06
2	8-generic-small	[(0,["call"]),(1,["han"]),(2,["call"]),(3,["..."]),(4,["exc"]),(6,["call"]),(7,["ret"]),(8,["call"]),(7,["ret"]),(9,["ret"]),(10,["#"])]	3,720	[(0,["call"]),(1,["han"]),(2,["call"]), **(3,["call"]),(4,["exc"])**, (6,["call"]),(7,["ret"]),(8,["call"]),(7,["ret"]),(9,["ret"]),(10,["#"])]	7,291
3	18-generic-small	[(0,["call"]),(1,["han"]),(2,["call"]),(3,["..."]),(4,["exc"]),(6,["call"]),(7,["ret"]),(8,["call"]),(7,["ret"]),(9,["ret"]),(10,["#"])]	233,8	[(0,["call"]),(1,["han"]),(2,["call"]), **(3,["call"]),(4,["exc"])**, (6,["call"]),(7,["ret"]),(8,["call"]),(7,["ret"]),(9,["ret"]),(10,["#"])]	319,5
4	19-generic-small	[(0,["call","pa"]),(1,["han"]),(2,["..."]),(4,["exc"]),(6,["call"]),(7,["ret"]),(8,["call"]),(7,["ret"]),(9,["pa","ret"]),(10,["#"])]	304,3	[(0,["call","pa"]),(1,["han"]), **(2,["call"]),(3,["call","pc"])**, (4,["exc"]),(6,["call"]),(7,["ret"]),(8,["call"]),(7,["ret"]),(9,["pa","ret"]),(10,["#"])]	884,5
5	30-generic-small	[(0,["call"]),(1,["han"]),(2,["..."]),(4,["exc"]),(6,["call"]),(7,["ret"]),(8,["call"]),(7,["ret"]),(9,["ret"]),(10,["#"])]	12,53	[(0,["call"]),(1,["han"]),**(2,["call"]), (3,["call"])**,(4,["exc"]),(6,["call"]),(7,["ret"]),(8,["call"]),(7,["ret"]),(9,["ret"]),(10,["#"])]	18,06
6	36-generic-small	[(0,["call"]),(1,["han"]),(2,["..."]),(4,["exc"]),(6,["call"]),(7,["ret"]),(8,["call"]),(7,["ret"]),(9,["ret"]),(10,["#"])]	9,631	[(0,["call"]),(1,["han"]),**(2,["call"]), (3,["call"])**,(4,["exc"]),(6,["call"]),(7,["ret"]),(8,["call"]),(7,["ret"]),(9,["ret"]),(10,["#"])]	14,83
7	37-generic-small	[(0,["call"]),(1,["han"]),(2,["call"]),(3,["..."]),(4,["exc"]),(6,["call"]),(7,["ret"]),(8,["call"]),(7,["ret"]),(9,["ret"]),(10,["#"])]	6,800	[(0,["call"]),(1,["han"]),(2,["call"]), **(3,["call"]),(4,["call"])**,(4,["exc"]),(6,["call"]),(7,["ret"]),(8,["call"]),(7,["ret"]),(9,["ret"]),(10,["#"])]	13,63

Chapter 7

Conclusions

In this paper we have discussed how OPL and POTL can represent a program and a formal property to verify and we have presented the realization of a model checker based on these two elements, POMC, in a new guise. The improvement of the counterexample return system permits to have a more complete vision of violations of the property checked, helping in troubleshooting phase and debugging. The counterexample procedure implemented is at its first version and it could be improved: though most of traces could be solved easily, sometimes finding all the possible combination matching push, shift and pop tuples could be very costly from the point of view of memory and computational power. This is due to the exponential complexity of the algorithm. Some benchmark test showed that the time for the solving of summary tuples could be up to three times that of the resolution of emptiness problem that has exponential complexity too. So the first work to do in the future will be that of lighten the procedure.

POMC now can resolve problems in the context of infinite runs of a program managing omega-words, but for the moment it doesn't return counterexamples. The next step is therefore the implementation of the counterexample system in omega-context that consist in finding an acceptance path that is repeated infinitely often. The implementation of the summary solving module has been absolutely not trivial, that's why we decided to stop here and leave this ideas as suggests for a future work.

Bibliography

[1] Rajeev Alur and Parthasarathy Madhusudan. Visibly pushdown languages. In *STOC '04*, pages 202–211. ACM, 2004. doi:10.1145/1007352.1007390.

[2] Michele Chiari, Dino Mandrioli, and Matteo Pradella. Temporal logic and model checking for operator precedence languages. In *GandALF 2018*, volume 277 of *EPTCS*, pages 161–175. Open Publishing Association, 2018. doi:10.4204/EPTCS.277.12.

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

[4] Stefano Crespi Reghizzi and Dino Mandrioli. Operator precedence and the visibly pushdown property. *J. Comput. Syst. Sci.*, 78(6):1837–1867, 2012. doi:10.1016/j.jcss.2011.12.006.

[5] Stefano Crespi Reghizzi, Dino Mandrioli, and Daniel F. Martin. Algebraic properties of operator precedence languages. *Information and Control*, 37(2):115–133, 1978. doi:10.1016/S0019-9958(78)90474-6.

[6] Robert W. Floyd. Syntactic analysis and operator precedence. *J. ACM*, 10(3): 316–333, 1963. doi:10.1145/321172.321179.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi:10.1145/363235.363259.

[8] Thomas Jensen, Daniel Le Metayer, and Tommy Thorn. Verification of control flow based security properties. In *Proc. '99 IEEE Symp. on Security and Privacy*, pages 89–103, 1999. doi:10.1109/SECPRI.1999.766902.

[9] Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. Predicate abstraction for program verification. In *Handbook of Model Checking*, pages 447–491. Springer, 2018. doi:10.1007/978-3-319-10575-8_15.

[10] Dino Mandrioli. On the heroism of really pursuing formal methods. In *FormaliSE '15*, pages 1–5. IEEE Computer Society, 2015. doi:10.1109/FormaliSE.2015.8.

[11] Amir Pnueli. The temporal logic of programs. In *FOCS '77*, pages 46–57. IEEE Computer Society, 1977. doi:10.1109/SFCS.1977.32.

Index of Acronyms

AP	Atomic Proposition
CFG	Context-Free Grammar
CFL	Context-Free Language
DFS	Depth-First Search
FOL	First-Order Logic
FO	First-Order
FSA	Finite-State Automaton
lhs	left-hand side
LTL	Linear Temporal Logic
MC	Model Checking
MSOL	Monadic Second-Order Logic
MSO	Monadic Second-Order
OPA	Operator Precedence Automaton
OPG	Operator Precedence Grammar
OPL	Operator Precedence Language
OPM	Operator Precedence Matrix
OP	Operator Precedence
PDS	Pushdown System
POMC	Precedence Oriented Model Checker
POTL	Precedence Oriented Temporal Logic
PR	Precedence Relation
rhs	right-hand side
ST	Syntax Tree
TS	Transition System

VPA	Visibly Pushdown Automaton
VPL	Visibly Pushdown Language

List of Figures

List of Tables