POLITECNICO
MILANO 1863

Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria
Corso di Laurea Magistrale in Computer Science and Engineering

Tesi di Laurea Magistrale

POMC

Toward a Model Checking Tool for Operator Precedence Languages

Candidato:
Davide Bergamaschi
Matricola 905436

Relatore:
Prof. Matteo Pradella

Correlatore:
Dr. Michele Chiari

Anno Accademico 2019–2020

To my parents.

Abstract

A significant contribution from the field of formal methods is providing software engineers with increasingly powerful and efficient tools to model, analyze and verify the properties and the behavior of computer programs.

Recently a promising line of research has stemmed from Operator Precedence Languages (OPL), a family of formal languages introduced by Robert W. Floyd in the late sixties. In particular, it has been shown how OPL, despite enjoying considerable expressive power, are characterized by algebraic and logic properties that make them particularly suitable for formal verification techniques. A major step in this direction has been taken with the introduction of temporal logics based on OPL, such as POTL (Precedence Oriented Temporal Logic). Procedural programs can be accurately modeled as OPL, and POTL can then be used to formulate specifications on structural elements like procedure calls, exceptions, handlers, etc.

We made a first attempt to reap the practical benefits of this approach with the creation of POMC, a newly born verification tool built in the Haskell programming language. The tool is currently capable of performing runtime verification of POTL formulas on finite strings. In this dissertation, we review the work done to develop POMC up to the latest version, illustrating the main design and architectural choices taken. We also present some concrete usage examples, and discuss the performance of POMC in light of empirical results. We finally suggest how POMC can be extended to implement OPL-based model checking.

Sommario

Un importante contributo da parte del campo dei metodi formali è quello di fornire agli ingegneri del software strumenti sempre più potenti ed efficienti per modellare, analizzare e verificare le proprietà ed il comportamento dei programmi.

Un promettente filone di ricerca si è ultimamente incentrato sugli Operator Precedence Languages (OPL), una famiglia di linguaggi formali introdotta da Robert W. Floyd sul finire degli anni sessanta. In particolare, è stato mostrato come gli OPL, nonostante il loro considerevole potere espressivo, siano caratterizzati da proprietà algebrico-logiche che li rendono particolarmente adatti per le tecniche di verifica formale. Un deciso passo avanti in questa direzione è stato effettuato con l'introduzione di logiche temporali basate sugli OPL, come ad esempio POTL (Precedence Oriented Temporal Logic). I programmi procedurali possono essere fedelmente modellati come OPL, e POTL può poi essere utilizzata per formulare specifiche elaborate su elementi strutturali come chiamate a procedure, eccezioni, *handlers*, ecc.

Un primo tentativo di sfruttare i benefici pratici di questo approccio si è concretizzato nella creazione di POMC, un *tool* di verifica di recente concezione scritto nel linguaggio di programmazione Haskell. Al momento il software è in grado di effettuare la *runtime verification* di formule POTL su stringhe finite. Questa tesi si propone di esporre il lavoro fatto per sviluppare POMC fino alla versione corrente, illustrando le principali scelte progettuali e architetturali fatte. Sono inoltre presentati degli esempi d'uso concreti, e viene discusso il livello di *performance* raggiunto alla luce di una serie di risultati empirici. Si suggerisce infine come POMC possa essere esteso per implementare funzionalità di *model checking* basate sugli OPL.

Acknowledgements

I would like to thank Prof. Matteo Pradella for giving me the opportunity to work on such a fascinating project, as well as Dr. Michele Chiari for his essential assistance throughout the development of POMC. This work owes immensely to their guidance and wisdom.

Furthermore, I would like to thank Prof. Dino Mandrioli for the pleasant discussions and precious suggestions.

Thanks also to my friends at Politecnico di Milano, especially Samuele, for all the philosophical discussions and the good laughs, and Francesco, for having shared with me such a significant part of our university experience.

Special thanks to Camille for proofreading this dissertation, and for having taught me much of what I have learned in these last two years that does not concern computer engineering.

A huge thank you to my longtime friends at home: Alberto, Alessandro & Alessandro, Emanuele, Federica, Flavio, Francesco, Lorenzo, Martina and Mattia, for their patience with me and their unwavering support and affection. Because of them, even when I was alone, I never really felt lonely.

My gratitude also goes to my family, that never stopped believing in me for a second. To my parents I would like to say: thank you for all that you have done for me, including continually supporting me through almost two decades of instruction; while I prepare for the new journey that is ahead of me, I take inspiration from your honesty and consistency.

Contents

Chapter 1

Introduction

Since the beginning of the digital era, humanity and technology have become more and more inextricably intertwined. We rely on computerized, automated systems to work, communicate, travel and even entertain ourselves. Some of these systems are characterized by a huge complexity, and some bear critical responsibilities, with faults and malfunctions potentially having dramatic consequences. The possibility of formally verifying the desired properties of an automated system has therefore become of crucial importance.

In particular, one of the major goals of formal verification is that of providing tools to model and analyze the behavior of software. Nowadays, programs are most commonly expressed in the form of Context-Free Languages (CFL). While traditional model checking formalisms (such as Linear-Time Logic specifications and Büchi automaton models) have been effectively used to verify regular properties of programs, there exist several other interesting properties which are non-regular, in that they are intrinsically tied to the context-free structure of programming languages (e.g. Hoare pre/post condition specifications).

In this regard, Operator Precedence Languages (OPL) [8] are an interesting subfamily of CFL. While OPL still exhibit a strong expressive power (e.g. they are suitable to express the syntax of real-world programming languages), they enjoy several useful properties [14] (such as closure under Boolean operations, concatenation and Kleene $*$, and decidability of language emptiness and inclusion), which do not hold for a generic CFL. Said properties make OPL especially suitable for formal verification.

This motivated the introduction of POTL [5][6], a temporal logic defined on OP languages. It can be used to conveniently reason on procedural programs: one can specify properties on matching pairs of call / return statements (e.g. Hoare-style conditions), but also express specifications on elements which are possibly in a one (many) to many (one) relation, such as handlers and exceptions. This puts POTL on a further level of expressivity with respect to other formalisms which have been used to characterize procedural programs in the past years, such as CaRet [2] and NWTL [1].

Recently, a model checking procedure for POTL on finite OP strings has been theorized [5]. At its heart lies an algorithmic construction for the translation of POTL formulas into equivalent automata. This dissertation introduces POMC, an early-stage verification tool targeted at OPL. Relying on the algorithmic construction mentioned earlier, the current version of the tool is able to perform runtime verification on finite OP strings. As will be discussed, POMC lays the basis for the implementation of a complete model checker based on OPL.

1.1 Outline

The exposition is structured in the following way: chapter 2 introduces the preliminary theoretical notions which underlie POMC; chapter 3 describes POMC, detailing its functionalities and its inner architecture; chapter 4 presents some empirical results concerning POMC, showing some practical use cases and reviewing its performance; chapter 5 summarizes the research contribution stemming from POMC and sketches some possible future developments.

Chapter 2

Preliminary concepts

This chapter covers the theoretical foundations that underpin formal verification techniques relying on Operator Precedence Languages. In particular, section 2.1 introduces OPL and their main formalizations, section 2.2 presents the syntax and semantics of POTL, section 2.3 covers the topic of formal verification of POTL formulas, supplying a fundamental automaton-translation algorithm.

2.1 Operator Precedence Languages

Operator Precedence Languages (OPL) are a subfamily of CFL, characterized by grammars where terminal symbols are enriched with precedence relations between one another.

Precedence relations are leveraged to guide the parsing of OP strings: the right-hand side of a grammar rule is recognized deterministically (it must appear in the form of a specific precedence structure called chain, as will be clarified later), avoiding the risk of having to roll back to a different reduction due to ambiguity. This peculiar feature allows, among other things, to efficiently parallelize the parsing process, thereby taking full advantage of modern parallel architectures [4].

Moreover, OPL share several properties with regular languages, including closure w.r.t. Boolean operations and decidability of the emptiness problem.

As we shall demonstrate later, these are the properties that underlie the proposed formal verification techniques relying on OPL.

The rest of this section will provide some key formal definitions concerning OPL, characterizing them through both their generating grammars (Operator Precedence Grammars, or OPG) and their recognizing automata (Operator Precedence Automata, or OPA). For a more complete introduction to the subject, the reader is referred to [14].

2.1.1 Operator Precedence Grammars

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

A context-free grammar (CF) G is a tuple (V_N, 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 (or l.h.s.) and $\alpha \in V^*$ being the right-hand side (r.h.s.);
- $S \in V_N$ is the start symbol (or axiom).

Figure 2.1 displays the rules of the context-free grammar for a simple arithmetic language supporting addition and multiplication, where numbers are represented with the e terminal symbol. The syntax tree resulting from the parsing of expression $e * e + e$ is shown on the right.

The following naming conventions are adopted, unless otherwise specified: uppercase Latin letters $A, B, \ldots$ denote nonterminal characters; lowercase

$$\begin{aligned}
&S \to E \mid T \mid F\\
&E \to E + T \mid T * F \mid e\\
&T \to T * F \mid e\\
&F \to e
\end{aligned}$$

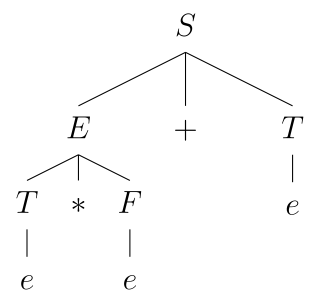


Figure 2.1: A grammar generating arithmetic expressions.

letters $a, b, \dots$ at the beginning of the Latin alphabet denote terminal characters; lowercase letters $x, y, \dots$ at the end of the Latin alphabet denote terminal strings; lowercase Greek letters $\alpha, \beta \dots$ 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 $\alpha_2 \to \alpha'_2 \in P$. Derivation, that is the reflective and transitive closure of the $\Rightarrow$ relation, is denoted by $\overset{*}{\Rightarrow}$. The language generated by a grammar G is thus defined as $L(G) = \{x \in \Sigma^* \mid S \overset{*}{\Rightarrow} x\}$.

Definition (OF,OG)**.** *A grammar rule is in Operator Form (OF) if its r.h.s. has no adjacent nonterminals. An Operator Grammar (OG) is a context-free grammar characterized solely by OF rules.*

Definition (Left/Right Terminal Set)**.** *Let G be an OG, and let A be one of its nonterminal symbols. The left and right terminal sets of A are given by:*

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

where $B \in V_N \cup \{\varepsilon\}$.

Notice how the grammar of Figure 2.1 is an OG. The left and right terminal sets of its nonterminal symbols are: $\mathcal{L}(E) = \{+, *, e\}$, $\mathcal{R}(E) = \{+, *, e\}$, $\mathcal{L}(T) = \{*, e\}$, $\mathcal{R}(T) = \{*, e\}$, $\mathcal{L}(F) = \{e\}$, $\mathcal{R}(F) = \{e\}$.

Let G be an OG, with $\alpha, \beta \in (V_N \cup \Sigma)^*$ and $a, b \in \Sigma$. Three binary precedence relations are defined:

equal in precedence: $a \doteq b \iff \exists\, A \to \alpha aBb\beta$, with $B \in V_N \cup \{\varepsilon\}$;
yields precedence: $a \lessdot b \iff \exists\, A \to \alpha aD\beta$, with $D \in V_N$, $b \in \mathcal{L}_G(D)$;
takes precedence: $a \gtrdot b \iff \exists\, A \to \alpha Db\beta$, with $D \in V_N$, $a \in \mathcal{R}_G(D)$.

For an OG G, the Operator Precedence Matrix (OPM) M of G is defined as a $|\Sigma| \times |\Sigma|$ array that, for each ordered pair (a, b), stores the set M_{ab} of the precedence relations holding between a and b.

Definition (OPG)**.** *An OG G is an Operator Precedence Grammar (OPG) iff the OPM of G is a conflict-free matrix, that is $\forall\, a, b, |M_{ab}| \leq 1$.*

Figure 2.2 contains the OPM for the grammar of arithmetic expressions of Figure 2.1, where the precedence relation holding for the ordered pair (a, b) is represented as the intersection of the a-row and the b-column. Each entry

	+	*	e
+	$\gtrdot$	$\lessdot$	$\lessdot$
*	$\gtrdot$	$\gtrdot$	$\lessdot$
e	$\gtrdot$	$\gtrdot$	

Figure 2.2: OPM for the arithmetic expression grammar.

of the matrix corresponds to at most a single element, making it clear that the grammar is an OPG.

Let Σ be an alphabet and let M be a conflict-free OPM defined on $\Sigma \cup \#$, where the # symbol is used to delimit the beginning and the end of terminal strings. We call the couple $(\Sigma,\ M)$ Operator Precedence Alphabet (OP Alphabet). If $M_{ab} = \pi$, with $\pi \in \{\lessdot, \doteq, \gtrdot\}$, we write $a\,\pi\,b$. By convention, it is established that $\# \doteq \#$ and, $\forall\, a \in \Sigma$, $\# \lessdot a$ and $a \gtrdot \#$.

The fundamental structures to formally characterize the strings of an OP language are chains: recursive structures representing string regions enclosed in $(\lessdot, \gtrdot)$ pairs, that, in parsing, correspond to r.h.s. candidates for reduction.

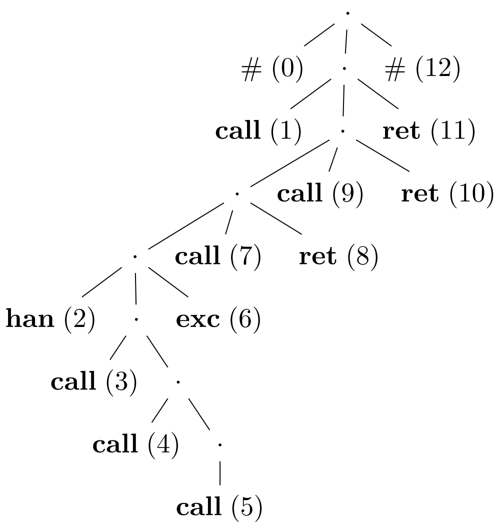
Definition (Simple Chain)**.** *A simple chain* $^{c_0}[c_1c_2\dots c_l]^{c_{l+1}}$ *is a string* $c_0c_1c_2\dots c_lc_{l+1}$*, such that:* $c_0, c_{l+1} \in \Sigma \cup \{\#\}$*,* $c_i \in \Sigma$ *for every* $i = 1, 2, \dots, l$ *(*$l \geq 1$*), and* $c_0 \lessdot c_1 \doteq c_2 \dots c_{l-1} \doteq c_l \gtrdot c_{l+1}$*.*

Definition (Composed Chain)**.** *A composed chain* $^{c_0}[s_0c_1s_1c_2\dots c_ls_l]^{c_{l+1}}$ *is a string* $c_0s_0c_1s_1c_2\dots c_ls_lc_{l+1}$*, with* $s_i \in \Sigma^*$*, where* $^{c_0}[c_1c_2\dots c_l]^{c_{l+1}}$ *is a simple chain, and either* $s_i = \varepsilon$ *or* $^{c_i}[s_i]^{c_{i+1}}$ *is a chain (simple or composed), for every* $i = 0, 1, \dots, l$ *(*$l \geq 1$*).*

For both simple and composed chains, the first and the last symbols are respectively called left and right context.

Definition (Compatible Word)**.** *A word* ω *over the OP alphabet* (Σ, M) *is compatible with* M *iff both the following conditions hold: 1.* $M_{ab} \neq \emptyset$*, for each pair of letters* a, b*, consecutive in* ω*; 2.* $M_{a_0a_{n+1}} \neq \emptyset$ *for each substring* x *of* $\#\omega\#$ *such that* $x = a_0x_0a_1x_1a_2\dots a_nx_na_{n+1}$ *and either* $x_i = \varepsilon$ *or* $^{a}_{i}[x_i]^{a}_{i+1}$ *is a chain (simple or composed) for every* $i \in \{1, \dots, n\}$*.*

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



#[**call**[[[**han**[**call**[**call**[**call**]]]**exc**]**call ret**]**call ret**]**ret**]#

Figure 2.3: OPM for a stack trace language and ST of an example string.

Figure 2.3 displays the OPM for a language of stack traces, where **call** represents procedure calls, **ret** represents regular termination events, **han** represents the instantiation of an exception handler, and **exc** represents the occurrence of an exception. On the right, the figure also shows an example of string compatible with the OPM, with chain bodies represented between square brackets, and the syntax tree isomorphic to its chain structures. Notice that the OPM is defined on all combinations of terminal symbols, therefore all words constructed with said symbols are compatible. Examples of incompatible words can be produced, on the other hand, for the arithmetic expression matrix of Figure 2.2, e.g. $e + e\ e$.

2.1.2 Operator Precedence Automata

For several decades after its introduction, the class of OPL remained characterized solely in terms of grammars, and lacked a family of recognizing automata. Recent research has filled this gap with the definition of Operator Precedence Automata (OPA) [13], a formalism that extends traditional left-to-right pushdown automata with the introduction of token precedence and precedence-guided transitions.

Definition (OPA)**.** *An Operator Precedence Automaton (OPA) is a tuple* $\mathcal{A} = (\Sigma, M, Q, I, F, \delta)$*, where:*

- Σ *is an alphabet*
- M *is an OPM on* Σ
- Q *is a set of states*
- $I \subseteq Q$ *is the set of initial states*
- $F \subseteq Q$ *is the set of final states*
- $\delta \subseteq Q \times (\Sigma \cup Q) \times Q$ *is the transition relation, which is the union of three disjoint relations:*

$$\delta_{shift} \subseteq Q \times \Sigma \times Q, \quad \delta_{push} \subseteq Q \times \Sigma \times Q, \quad \delta_{pop} \subseteq Q \times Q \times Q.$$

We use p, q, ... to denote states.

Let $\Gamma' = \Gamma \cup \{\perp\}$ be the stack alphabet, with $\Gamma = \Sigma \times Q$ and $\perp$ denoting the bottom symbol of the stack. We indicate stack symbols with $\perp$ or $[a, q]$ couples.

We also define predicates *symb* and *state* such that $symb([a, q]) = a$ and $symb(\perp) = \#$, and $state([a, q]) = q$. Furthermore, given stack content $\gamma = \gamma_n \dots \gamma_1 \perp$, with $\gamma_i \in \Gamma$ and $n \geq 0$, we set $symb(\gamma) = symb(\gamma_n)$ if $n \geq 1$ and $symb(\gamma) = \#$ if $n = 0$.

An OPA configuration is a triple $c = \langle w, q, \gamma \rangle$, where $\omega \in \Sigma^* \#$ represents the unread portion of the input symbols, $q \in Q$ is the current state, and $\gamma \in \Gamma^* \perp$ is the current stack content.

A computation is a finite sequence $c_0 \vdash c_1 \vdash \cdots \vdash c_n$ of moves (transitions) $c_i \vdash c_{i+1}$.

Moves are of three kinds, in relation to the precedence relation between the symbol on top of the stack and the next input symbol.

Push Moves

If $symb(\gamma) \lessdot a$, then $\langle ax, p, \gamma \rangle \vdash \langle x, q, [a, p]\gamma \rangle$, with $(p, a, q) \in \delta_{push}$.

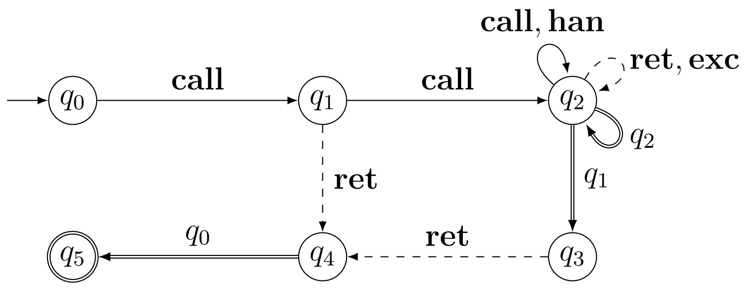
Shift Moves

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 Moves

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

Push moves add a new element on top of the stack, composed by the consumed input symbol and the starting state. Shift moves only update the top of the stack by changing the old input symbol into the consumed input symbol. Pop moves on the other hand do not consume any input, and only remove the top element of the stack. In all three cases, the current state of the automaton is updated according to the respective δ relation. Notice how shift and pop moves are not performed when the stack contains only $\perp$.



Input	State	Stack
call call han call exc ret ret #	q_0	$\perp$
call han call exc ret ret #	q_1	[**call**, q_0] $\perp$
han call exc ret ret #	q_2	[**call**, q_1] [**call**, q_0] $\perp$
call exc ret ret #	q_2	[**han**, q_2] [**call**, q_1] [**call**, q_0] $\perp$
exc ret ret #	q_2	[**call**, q_2] [**han**, q_2] [**call**, q_1] [**call**, q_0] $\perp$
exc ret ret #	q_2	[**han**, q_2] [**call**, q_1] [**call**, q_0] $\perp$
ret ret #	q_2	[**call**, q_1] [**call**, q_0] $\perp$
ret #	q_2	[**ret**, q_1] [**call**, q_0] $\perp$
ret #	q_3	[**call**, q_0] $\perp$
#	q_4	[**ret**, q_0] $\perp$
#	q_5	$\perp$

Figure 2.4: An OPA on the matrix of Figure 2.3, with an example run.

Definition (Accepted Language)**.** *The accepted language of an OPA $\mathcal{A}$ is:*

$$L(\mathcal{A}) = \{x \in \Sigma^* \mid \langle x\#, q_I, \perp\rangle \vdash^* \langle \#, q_F, \perp\rangle, q_I \in I, q_F \in F\}.$$

The automaton of Figure 2.4 displays an OPA accepting a subset of the stack trace language defined with the matrix of Figure 2.3. Following the graphical conventions on OPA, it is represented as a graph having Q as the set of vertices and $\Sigma \cup Q$ as the set of edge labelings, with push, shift and pop transitions denoted respectively with normal, dashed and double arrows. Notice how the automaton, combined with the precedence relations imposed by the OPM, restricts the accepted language to stack traces that: 1. are enclosed in a main ($\mathbf{call}, \mathbf{ret}$) frame; 2. do not contain uncaught exceptions; 3. do not contain return statements without a corresponding call; 4. do not contain non-terminating calls.

2.2 Precedence Oriented Temporal Logic

Temporal logics are logical formalisms that allow reasoning about statements in terms of time. In the field of formal verification, they have assumed a fundamental role for specifying and constraining the evolution of a system in time.

Linear Time Logic (LTL) [16] is one of the most successful among these formalisms. It is based on a linear model of time, characterizing it as a temporally-ordered, possibly non-terminating sequence of events. LTL is able to express several interesting properties, like safety and liveness conditions. Its expressive power is nevertheless confined to the so-called regular properties: the set of prefixes that violate them has to be expressible as a regular language (RL) [3]. This constitutes a relevant limitation when attempting to reason about procedural programs, due to their intrinsically structured nature (that is, context-free rather than regular).

A relevant attempt to overcome these limitations has been done with the introduction of CaRet [2] and NWTL [1]. Procedural programs are modeled as Nested Word structures, consisting in the combination of a linear ordering of elements with a one-to-one matching relation of call and return positions. CaRet and NWTL can then be used to express a new class of properties on such words, directly addressing the nested structure of the program model.

For example, the matching of the start and the end of a procedure allows to express Hoare-style specifications composed of pre/post conditions.

However, some relevant program structures and behaviors are not expressible in terms of a one-to-one matching relation. For example, with modern programming languages, when a procedure raises an exception it not only terminates itself, but also causes the execution environment to backtrack on the call stack terminating all procedures until a suitable handler is found, or the program is terminated. To account for these types of behavior, one-to-many and many-to-one matching relations have to be employed.

Recently, researchers have proposed the use of temporal logics based on OPL to respond to the need of additional expressive power. In particular, it has been shown how OPL strictly include Visibly Pushdown Languages, the family of Nested Words languages [7]. A procedural program can be more closely modeled as OPL, and a suitable temporal logic can be used to express properties on matching elements by leveraging the relation induced by precedence chains.

The most recent and refined example of temporal logic based on OPL is POTL (Precedence Oriented Temporal Logic) [5][6]. POTL allows to express a wide range of specifications on procedural programs, including conditions on the throwing of exceptions and stack inspection properties. In the remainder of this section, we will present a formal definition of its syntax and semantics, as well as some concrete examples on the alphabet of stack traces (Figure 2.3).

Definition (POTL Syntax)**.** *Let AP be a finite set of atomic propositions. The syntax of POTL is given by the following rules:*

$$\begin{aligned}\varphi ::= \mathrm{a} \mid \neg\varphi \quad &\mid \varphi \lor \varphi \\ \mid \bigcirc^t\varphi \quad &\mid \circleddash^t\varphi \\ \mid \chi_F^t\varphi \quad &\mid \chi_P^t\varphi \\ \mid \varphi\, \mathcal{U}_\chi^t\, \varphi &\mid \varphi\, \mathcal{S}_\chi^t\, \varphi \\ \mid \bigcirc_H^t\varphi \quad &\mid \circleddash_H^t\varphi \\ \mid \varphi\, \mathcal{U}_H^t\, \varphi &\mid \varphi\, \mathcal{S}_H^t\, \varphi\end{aligned}$$

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

Before illustrating semantics, some preliminary structural definitions have to be introduced.

Definition (Word Structure)**.** *A word structure (or OP word) is a tuple* $\langle U, M_{\mathcal{P}(AP)}, P\rangle$*, where:*

- $U = \{0, 1, \ldots, n, n+1\}$*, with* $n \in \mathbb{N}$*, is a set of word positions*
- $M_{\mathcal{P}(AP)}$ *is an OPM on* $\mathcal{P}(AP)$
- $P : U \to \mathcal{P}(AP)$ *associates each position with the set of atomic propositions that hold in it, with* $P(0) = P(n+1) = \{\#\}$*.*

Given two word structure positions i and j, we write $i \,\pi\, j$ to indicate that $P(i) \,\pi\, P(j)$.

Definition (Chain Relation)**.** *The Chain Relation* $\chi \subseteq U \times U$ *is defined so that* $\chi(i, j)$ *holds between positions* i *and* j *iff* $i < j - 1$ *and* i *and* j *are respectively the left and the right contexts of the same chain.*

Figure 2.5 shows the word of Figure 2.3, with the chain relation graphically represented by arcs connecting contexts. The alphabet is extended to be a set of propositional symbol sets, $\mathcal{P}(AP)$, with AP containing **call**, **ret**, **han**, **exc**, and also additional labels having a denotative role (e.g. p_A, p_B, p_C represent the names of different procedures). The OPM is adapted to work on the new alphabet in the following way: the atomic propositions are partitioned into structural labels (or SL, the ones written in bold) and normal labels (the ones in round font). $M_{\mathcal{P}(AP)}$ is then defined only for subsets of AP containing exactly one SL, so that given two SL $\mathbf{l}_1, \mathbf{l}_2$, for any $a, a', b, b' \in \mathcal{P}(AP)$ s.t. $\mathbf{l}_1 \in a, a'$ and $\mathbf{l}_2 \in b, b'$ we have $M_{\mathcal{P}(AP)}(a, b) = M_{\mathcal{P}(AP)}(a', b')$.

The semantics of POTL is defined with respect to single word positions of a word structure. The definitions below, where w is an OP word and $i \in U$ is

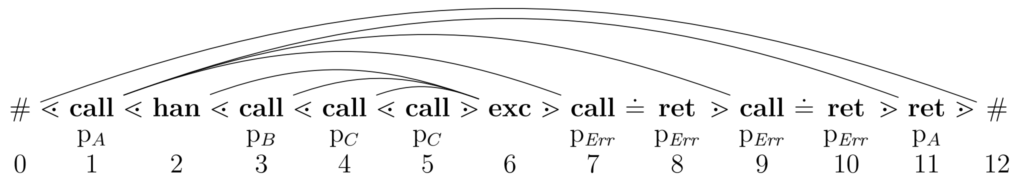

$\# \lessdot \mathbf{call} \lessdot \mathbf{han} \lessdot \mathbf{call} \lessdot \mathbf{call} \lessdot \mathbf{call} \gtrdot \mathbf{exc} \gtrdot \mathbf{call} \doteq \mathbf{ret} \gtrdot \mathbf{call} \doteq \mathbf{ret} \gtrdot \mathbf{ret} \gtrdot \#$
p_A p_B p_C p_C p_{Err} p_{Err} p_{Err} p_{Err} p_A
0 1 2 3 4 5 6 7 8 9 10 11 12

Figure 2.5: The word of Figure 2.3, with additional propositional labels.

a position of w, illustrate the meaning of each syntax element of POTL, and are accompanied by examples on the formula of Figure 2.5.

Atomic Propositions
Let $a \in AP$. Then $(w, i) \models a$, iff $a \in P(i)$. In the example formula, **han** holds in position 3 and p_A holds in positions 1 and 11.

Operators from Propositional Logic and LTL
$\neg$ and $\vee$ (as well as $\wedge, \Box, \Diamond, \dots$) behave like their propositional / LTL counterparts. For example, the formula $\neg(\mathbf{call} \vee \mathbf{han} \vee \mathbf{ret})$ is true in position 6 and false in every other position (except for the start and the end).

Precedence Next and Back
The $\bigcirc^t$ (resp. $\circleddash^t$) operator acts similarly to the Next (resp. Back) operator from LTL, but it imposes a further condition on the subsequent (resp. previous) position: in the d case, said position must be at a lower or equal level in the syntax tree, while in the u case it must be at an equal or higher one.

– $(w, i) \models \bigcirc^d \varphi$ iff $(w, i+1) \models \varphi$, and $i \lessdot (i+1)$ or $i \doteq (i+1)$;

– $(w, i) \models \circleddash^d \varphi$ 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 $i \doteq (i+1)$ or $i \gtrdot (i+1)$;

– $(w, i) \models \circleddash^u \varphi$ iff $(w, i-1) \models \varphi$, and $(i-1) \doteq i$ or $(i-1) \gtrdot i$.

E.g. $\bigcirc^d$**call** imposes that the next position be an inner call (it holds in position 2, 3 and 4 of the example string), and conversely $\circleddash^d$**call** requires that the previous position be a call, meaning that the current one is the first element of the body of a procedure (this is true in positions 2, 4, 5) or the return statement of an empty one (position 8, 10). $\circleddash^u$**call** can instead be used to assert that the current position terminates an empty function frame, as in positions 6, 8 and 10; $\bigcirc^u$**call** also holds in positions 6 and 8.

Chain Next and Back
The operators χ_F^t and χ_P^t impose restrictions in a similar way as $\bigcirc^t$ and $\circleddash^t$, but instead of concerning the subsequent or previous position they move across the chain relation.

- $(w,i) \models \chi_F^d \varphi$ 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 $(w,j) \models \varphi$;
- $(w,i) \models \chi_F^u \varphi$ iff there exists a position $j > i$ such that $\chi(i,j)$, $i \doteq j$ or $i \gtrdot j$, and $(w,j) \models \varphi$;
- $(w,i) \models \chi_P^u \varphi$ iff there exists a position $j < i$ such that $\chi(j,i)$, $j \doteq i$ or $j \gtrdot i$, and $(w,j) \models \varphi$.

In the example string, $\chi_F^d \mathrm{p}_{Err}$ holds in position 1, since p_{Err} holds in 7 and 9 and both $\chi(1,7)$ and $\chi(1,9)$ hold. χ_F^u**exc** can be used to describe **call** positions that correspond to procedures terminated by an exception thrown by an inner procedure (such as pos. 3 and 4). χ_P^u**call** is true in **exc** statements that terminate at least one procedure other than the one raising it, such as the one in pos. 6. χ_F^d**ret** (and equally χ_F^u**ret**) holds in **call** positions initiating a procedure which terminates regularly (with a **ret** statement as opposed to an **exc** one), such as position 1.

Summary Until and Since

The $\mathcal{U}_\chi^t$ and $\mathcal{S}_\chi^t$ are obtained by inductively applying the $\bigcirc^t, \chi_F^t$ and $\ominus^t, \chi_P^t$ operators, respectively.

- $(w,i) \models \psi \, \mathcal{U}_\chi^t \, \theta$ iff one of the following conditions holds:
  - $(w,i) \models \theta$
  - $(w,i) \models \psi$, and $(w,i) \models \bigcirc^t(\psi \, \mathcal{U}_\chi^t \, \theta)$ or $(w,i) \models \chi_F^t(\psi \, \mathcal{U}_\chi^t \, \theta)$;
- $(w,i) \models \psi \, \mathcal{S}_\chi^t \, \theta$ iff one of the following conditions holds:
  - $(w,i) \models \theta$
  - $(w,i) \models \psi$, and $(w,i) \models \ominus^t(\psi \, \mathcal{S}_\chi^t \, \theta)$ or $(w,i) \models \chi_P^t(\psi \, \mathcal{S}_\chi^t \, \theta)$.

For example, $\top \mathcal{U}_\chi^d$**exc** is true in **call** positions corresponding to function frames that contain one or more **exc**, but are not directly terminated by one of them, such as position 1 (due to path 1-2-6). $\top \, \mathcal{U}_\chi^u$ **exc**, on the other hand, holds for positions contained in the frame of a function that is terminated by an exception, such as position 3 (due to path 3-6). Notice also how (**call** $\vee$ **exc**) $\mathcal{S}_\chi^u \, \mathrm{p}_B$ in position 7 because of path 3-6-7.

Hierarchical Next and Back

The $\bigcirc_H^u$ and $\circleddash_H^u$ are next/back operators which allow to "move" between right contexts of chains with the same element as left context. Conversely, $\bigcirc_H^d$ and $\circleddash_H^d$ consider the left contexts of chains sharing their right context.

– $(w,i) \models \bigcirc_H^u \varphi$ iff there exists 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 $(w,j) \models \varphi$;

– $(w,i) \models \circleddash_H^u \varphi$ iff there exists 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 $(w,j) \models \varphi$;

– $(w,i) \models \bigcirc_H^d \varphi$ iff there exists 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 $(w,j) \models \varphi$;

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

In the example word, $\bigcirc_H^u p_{Err}$ holds in position 7 because p_{Err} holds in 9, and $\circleddash_H^u p_{Err}$ in 9 because p_{Err} holds in 7, with the two operators respectively going up and down between calls to p_{Err}. On the other hand, $\bigcirc_H^d$ and $\circleddash_H^d$ can be used to go down and up among **call** positions terminated by the same **exc**: $\bigcirc_H^d p_C$ holds in position 3 (given that both positions 3 and 4 are in the chain relation with position 6), while $\circleddash_H^d p_B$ holds in 4.

Hierarchical Until and Since

The $\mathcal{U}_H^t$ and $\mathcal{S}_H^t$ are obtained by respectively iterating the $\bigcirc_H^t$ and the $\circleddash_H^t$ operators, similarly to how summary until and since are defined.

– $(w,i) \models \psi \,\mathcal{U}_H^u\, \theta$ iff one of the following conditions holds:

  - $(w,i) \models \theta$ and there exists a position $h < i$ s.t. $\chi(h,i)$ and $h \lessdot i$
  - $(w,i) \models \psi$ and $(w,i) \models \bigcirc_H^u(\psi \,\mathcal{U}_H^u\, \theta)$;

– $(w,i) \models \psi \,\mathcal{S}_H^u\, \theta$ iff one of the following conditions holds:

  - $(w,i) \models \theta$ and there exists a position $h < i$ s.t. $\chi(h,i)$ and $h \lessdot i$
  - $(w,i) \models \psi$ and $(w,i) \models \circleddash_H^u(\psi \,\mathcal{S}_H^u\, \theta)$;

– $(w,i) \models \psi \,\mathcal{U}_H^d\, \theta$ iff one of the following conditions holds:

  - $(w,i) \models \theta$ and there exists a position $h > i$ s.t. $\chi(i,h)$ and $i \gtrdot h$

  - $(w, i) \models \psi$ and $(w, i) \models \bigcirc^u_H (\psi \, \mathcal{U}^d_H \, \theta)$;

- $(w, i) \models \psi \, \mathcal{S}^d_H \, \theta$ iff one of the following conditions holds:
  - $(w, i) \models \theta$ and there exists a position $h > i$ s.t. $\chi(i, h)$ and $i \gtrdot h$
  - $(w, i) \models \psi$ and $(w, i) \models \circleddash^u_H (\psi \, \mathcal{S}^d_H \, \theta)$.

  In the example, $\mathbf{call} \, \mathcal{U}^u_H \, p_{Err}$ holds in position 7, while $\mathbf{call} \, \mathcal{S}^u_H \, p_{Err}$ holds in position 9. Furthermore, $\mathbf{call} \mathcal{U}^d_H \mathrm{p}_C$ holds in position 3, and $\mathbf{call} \mathcal{S}^d_H \mathrm{p}_B$ in position 4, both because of path 3-4.

Notice how the *d*-versions of POTL operators specify relations which go downwards with respect to the syntax tree of an OP word, while conversely *u*-versions go upwards.

The distinction above allows the expression of a concise specification for most practical problems, however sometimes one might want to refer to a specific set of precedence relations. To do this conveniently, the following PR-based operators can be used: $\bigcirc^\Pi, \circleddash^\Pi, \chi^\Pi_F, \chi^\Pi_P, \mathcal{U}^\Pi_\chi, \mathcal{S}^\Pi_\chi$, with $\Pi \subseteq \{\lessdot, \doteq, \gtrdot\}, \Pi \neq \emptyset$. Their semantic rules are analogous to the corresponding PR-based operators, the only difference being that the concerned PRs are precisely those contained in Π (instead of just $\{\lessdot, \doteq\}$ or $\{\doteq, \gtrdot\}$). Notice how the two syntax types are equivalent in terms of expressive power (they can be easily translated into one another, e.g. $\chi^d_F = \chi^{\lessdot\doteq}_F$).

2.3 Formal Verification

We restrict our attention to two verification problems for POTL, and we outline a way to solve them algorithmically. The proposed approach is akin to "traditional" automata-theoretic ones [12].

Definition (Runtime Verification Problem)**.** *Given a POTL formula φ and an OP string s, the runtime verification problem consists in establishing whether $(s, 1) \models \varphi$, i.e. verifying that φ holds in the first position of s.*

Definition (Model Checking Problem)**.** *Given an OPA $\mathcal{A}$ and a POTL formula φ, the model checking problem consists in establishing whether $\mathcal{A} \models \varphi$, i.e. verifying that φ holds in the first position of every string accepted by $\mathcal{A}$.*

The runtime verification problem typically arises in the presence of a system producing an execution trace for which some desired properties have to be checked. For our purposes, a trace consists in an OP string s, while the specification is expressed as a POTL formula φ, as described above. The verification question can be answered by constructing an OPA $\mathcal{A}_\varphi$, which accepts all the OP strings that satisfy φ, and then running the automaton with s as input. The trace s satisfies the specification φ iff s is recognized by the automaton.

With model checking, on the other hand, a system is modeled as an OPA $\mathcal{A}$ and a specification of the desired system behavior is expressed as a POTL formula φ. Then OPA $\mathcal{A}_{\neg\varphi}$ is constructed from the negation of φ, such that $\mathcal{A}_{\neg\varphi}$ accepts all strings that violate φ. Finally, $\mathcal{A}$ is intersected with $\mathcal{A}_{\neg\varphi}$, and the emptiness of the intersection language is checked: if $L(\mathcal{A}) \cap L(\mathcal{A}_{\neg\varphi}) = \emptyset$, then $\mathcal{A}$ meets the specification, otherwise a counterexample can be produced.

It is clear what a crucial role is played by translating POTL formulas into equivalent Operator Precedence Automata in addressing both of the above problems. An algorithmic procedure to do that is provided in the following subsection. The automaton it builds has size of at most $2^{O(|\varphi|)}$ states, where $|\varphi|$ is the length of the input formula [5].

2.3.1 Automaton Construction

Let $(\mathcal{P}(AP), M_{\mathcal{P}(AP)})$ be an OP alphabet, where AP is a finite set of atomic propositions, and let φ be a POTL formula. The following construction shows how to build an OPA $\mathcal{A}_\varphi = \langle \mathcal{P}(AP), M_{\mathcal{P}(AP)}, Q, I, F, \delta \rangle$, that accepts all and only the models of φ.

Let $\mathrm{Cl}(\varphi)$ be the closure of φ, which contains all relevant subformulas of φ. In its basic form (which, as we shall see, is augmented in presence of specific operators), $\mathrm{Cl}(\varphi)$ amounts to the smallest set which respects the following rules:

1. $\varphi \in Cl(\varphi)$;
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 $\bigcirc^{\Pi}$, χ_F^{Π}, ...) 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 $\psi, \theta \in \mathrm{Cl}(\varphi)$.

We also define $\mathrm{Atoms}(\varphi)$ as the set of all consistent subsets of $\mathrm{Cl}(\varphi)$, where $\Phi \subseteq \mathrm{Cl}(\varphi)$ is consistent if all the following conditions hold:

1. $\forall \psi \in \mathrm{Cl}(\varphi)$, $\psi \in \Phi$ 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.

As with the closure, the consistency constraints listed above are enriched when specific operators are present.

The set of states is $Q = \mathrm{Atoms}(\varphi)^2$: it is composed by elements of the form $\Phi = (\Phi_c, \Phi_p)$, where Φ_c (current set) represents the formulas which hold in the current position, and Φ_p (pending set) keeps track of the temporal obligations introduced by certain operators, as will be clarified below.

In a computation of $\mathcal{A}_\varphi$, each state is associated with a word position (and represents the formulas and the obligations which hold in it). The transition function, in its basic form, is thus characterized by the following constraints:

- $\forall(\Phi, a, \Psi) \in \delta_{push/shift}$, $\Phi_c \cap AP = a$;
- $\forall(\Phi, \Theta, \Psi) \in \delta_{pop}$, $\Phi_c = \Psi_c$.

The initial set I is composed by states (Φ_c, Φ_p) such that $\varphi \in \Phi_c$, while the states of final set F are of the form (Ψ_c, Ψ_p), where $\Psi_c \cap AP = \{\#\}$ and Ψ_c contains no future operators. Furthermore, Φ_p and Ψ_p are generally "empty" (contain only negated formulas), with the exception of some operators that are explicitly indicated in the following.

Precedence Next and Back

For the sake of generality, we show the construction for the PR-based version of the Precedence Next and Back operators. The ST-based ones can be easily translated as:

$$\bigcirc^d = \bigcirc^{\lessdot\doteq} \quad \circleddash^d = \circleddash^{\lessdot\doteq} \quad \bigcirc^u = \bigcirc^{\doteq\gtrdot} \quad \circleddash^u = \circleddash^{\doteq\gtrdot}.$$

Let $\bigcirc^{\Pi}\psi \in \mathrm{Cl}(\varphi)$, with $\Pi \subseteq \{\lessdot, \doteq, \gtrdot\}$. Let $(\Phi, a, \Psi) \in \delta_{\mathit{shift}} \cup \delta_{\mathit{push}}$, and $b = \Psi_c \cap AP$. Then, $\bigcirc^{\Pi}\psi \in \Phi_c$ iff $\psi \in \Psi_c$ and $a \, \pi \, b$ for a PR $\pi \in \Pi$.

Let $\circleddash^{\Pi}\psi \in \mathrm{Cl}(\varphi)$, with $\Pi \subseteq \{\lessdot, \doteq, \gtrdot\}$. Let $(\Phi, a, \Psi) \in \delta_{\mathit{shift}} \cup \delta_{\mathit{push}}$, and $b = \Psi_c \cap AP$. Then, $\psi \in \Phi_c$ iff $\circleddash^{\Pi}\psi \in \Psi_c$ and $a \, \pi \, b$ for a PR $\pi \in \Pi$.

Chain Next and Back

For these operators, as for the Precedence Next and Back ones, we will refer to the PR-based form. The ST-based operators can be translated as follows:

$$\chi_F^d = \chi_F^{\lessdot\doteq} \quad \chi_F^u = \chi_F^{\doteq\gtrdot} \quad \chi_P^d = \chi_P^{\lessdot\doteq} \quad \chi_P^u = \chi_P^{\doteq\gtrdot}.$$

Let $\chi_F^{\Pi} \in \mathrm{Cl}(\varphi)$. Then, for each $\pi \in \Pi$, we also add χ_F^{π} to $\mathrm{Cl}(\varphi)$. For every atom Φ, we impose that $\chi_F^{\Pi} \in \Phi$ iff $\chi_F^{\pi'} \in \Phi$ for some $\pi' \in \Pi$.

We also add the auxiliary symbol χ_L in $\mathrm{Cl}(\varphi)$, for which we impose the following rule: if $(\Phi, a, \Psi) \in \delta_{\mathit{push}}$, then $\chi_L \in \Phi_p$; if $(\Phi, a, \Psi) \in \delta_{\mathit{shift}}$ or $(\Phi, \Theta, \Psi) \in \delta_{\mathit{pop}}$, then $\chi_L \notin \Phi_p$.

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

If $\chi_F^{\doteq}\psi \in \mathrm{Cl}(\varphi)$, the following constraints are added:

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

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

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

If $\chi_F^{\gtrdot}\psi \in \mathrm{Cl}(\varphi)$, the following constraints are added::

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

Let $\chi_P^\Pi \in \mathrm{Cl}(\varphi)$. Then, for each $\pi \in \Pi$, we also add χ_P^π to $\mathrm{Cl}(\varphi)$. For every atom Φ, we impose that $\chi_P^\Pi \in \Phi$ iff $\chi_P^{\pi'} \in \Phi$ for some $\pi' \in \Pi$.

We also add the auxiliary symbol χ_R in $\mathrm{Cl}(\varphi)$, for which we impose the following rule: for any $(\Phi, \Theta, \Psi) \in \delta_{pop}$, we have $\chi_R \in \Psi_p$; for any $(\Phi, a, \Psi) \in \delta_{push/shift}$, we have $\chi_R \notin \Psi_p$. χ_R is allowed in the pending part of final states.

If $\chi_P^{\doteq}\psi \in \mathrm{Cl}(\varphi)$, the following constraints are added:

9. let $(\Phi, a, \Psi) \in \delta_{shift}$: then $\chi_P^{\doteq}\psi \in \Phi_c$ iff $\chi_P^{\doteq}\psi, \chi_R \in \Phi_p$;
10. let $(\Phi, a, \Psi) \in \delta_{push}$: then $\chi_P^{\doteq}\psi \notin \Phi_c$;
11. let $(\Phi, \Theta, \Psi) \in \delta_{pop}$: then $\chi_P^{\doteq}\psi \in \Psi_p$ iff $\chi_P^{\doteq}\psi \in \Theta_p$;
12. let $(\Phi, a, \Psi) \in \delta_{push/shift}$: then $\chi_P^{\doteq}\psi \in \Psi_p$ iff $\psi \in \Phi_c$.

If $\chi_P^{\lessdot}\psi \in \mathrm{Cl}(\varphi)$, the following constraints are added:

13. let $(\Phi, a, \Psi) \in \delta_{push}$: then $\chi_P^{\lessdot}\psi \in \Phi_c$ iff $\chi_P^{\lessdot}\psi, \chi_R \in \Phi_p$;
14. let $(\Phi, a, \Psi) \in \delta_{shift}$: then $\chi_P^{\lessdot}\psi \notin \Phi_c$;
15. let $(\Phi, \Theta, \Psi) \in \delta_{pop}$: then $\chi_P^{\lessdot}\psi \in \Psi_p$ iff $\chi_P^{\lessdot}\psi \in \Theta_p$;
16. let $(\Phi, a, \Psi) \in \delta_{push/shift}$: then $\chi_P^{\lessdot}\psi \in \Psi_p$ iff $\psi \in \Phi_c$.

If $\chi_P^{\gtrdot}\psi \in \mathrm{Cl}(\varphi)$, we add the auxiliary symbol $\chi_{\doteq}$ to $\mathrm{Cl}(\varphi)$, for which we impose the following rule: for any $(\Phi, a, \Psi) \in \delta_{shift}$, $\chi_{\doteq} \in \Phi_p$; for any $(\Phi, a, \Psi) \in \delta_{push}$ and $(\Phi, \Theta, \Psi) \in \delta_{pop}$, $\chi_{\doteq} \notin \Phi_p$. $\chi_P^{\gtrdot}\psi$ and $\chi_{\doteq}$ are allowed in the pending part of final states. Additionally, the constraints that follow are imposed.

Let $(\Phi, a, \Psi) \in \delta_{push/shift}$:

17. $\chi_P^{\gtrdot}\psi \notin \Psi_p$;
18. $\chi_P^{\gtrdot}\psi \in \Phi_c$ iff $\chi_P^{\gtrdot}\psi, \chi_R \in \Phi_p$.

Let $(\Phi, \Theta, \Psi) \in \delta_{pop}$:

19. if ($\chi_L \in \Psi_p$ or $\chi_{\doteq} \in \Psi_p$), then $\chi_P^{\gtrdot}\psi \in \Psi_p$ iff $\chi_P^{\gtrdot}\psi \in \Phi_p$;
20. if $\chi_L, \chi_{\doteq} \notin \Psi_p$, then $\chi_P^{\gtrdot}\psi \in \Psi_p$ iff either $\chi_P^{\lessdot}\psi \vee \circleddash^d \psi \in \Theta_c$ or $\chi_P^{\gtrdot}\psi \in \Phi_p$.

Summary Until and Since

For any $\Phi \in \mathrm{Atoms}(\varphi)^2$, we have $\psi \, \mathcal{U}^t \, \theta \in \Phi_c$, with $t \in \{d, u\}$, iff at least one of the following conditions hold:

1. $\theta \in \Phi_c$;
2. $\bigcirc^t(\psi \, \mathcal{U}^t \, \theta), \psi \in \Phi_c$;
3. $\chi_F^t(\psi \, \mathcal{U}^t \, \theta), \psi \in \Phi_c$.

For any $\Phi \in \mathrm{Atoms}(\varphi)^2$, we have $\psi \, \mathcal{S}^t \, \theta \in \Phi_c$, with $t \in \{d, u\}$, iff at least one of the following conditions hold:

4. $\theta \in \Phi_c$,
5. $\ominus^t(\psi \, \mathcal{S}^t \, \theta), \psi \in \Phi_c$, or
6. $\chi_P^t(\psi \, \mathcal{S}^t \, \theta), \psi \in \Phi_c$.

Hierarchical Next and Back

The constructions for these operators rely on the auxiliary symbols χ_L, $\chi_{\doteq}$ and χ_R defined above.

If $\bigcirc_H^d \psi \in \mathrm{Cl}(\varphi)$, we add $\chi_P^{\lessdot} \psi, \ominus^{\lessdot} \psi, \chi_P^{\lessdot}(\bigcirc_H^d \psi), \ominus^{\lessdot}(\bigcirc_H^d \psi)$ to $\mathrm{Cl}(\varphi)$, and impose the constraints that follow.
Let $(\Phi, \Theta, \Psi) \in \delta_{pop}$:

1. if $\chi_L, \chi_{\doteq} \notin \Psi_p$, then $(\chi_P^{\lessdot} \psi \in \Theta_c$ or $\ominus^{\lessdot} \psi \in \Theta_c)$ iff $\bigcirc_H^d \psi \in \Psi_p$;
2. if $\chi_L, \chi_{\doteq} \notin \Psi_p$, then $\bigcirc_H^d \psi \in \Phi_p$ iff $(\chi_P^{\lessdot}(\bigcirc_H^d \psi) \in \Theta_c$ or $\ominus^{\lessdot}(\bigcirc_H^d \psi) \in \Theta_c)$;
3. if $(\chi_P^{\lessdot}(\bigcirc_H^d \psi) \in \Theta_c$ or $\ominus^{\lessdot}(\bigcirc_H^d \psi) \in \Theta_c)$, then $\chi_{\doteq} \notin \Psi_p$.

Let $(\Phi, a, \Psi) \in \delta_{push/shift}$:

4. if $\bigcirc_H^d \psi \in \Phi_c$, then $\chi_L \in \Psi_p$;
5. $\bigcirc_H^d \psi \notin \Psi_p$.

If $\ominus_H^d \psi \in \mathrm{Cl}(\varphi)$, we add $\chi_P^{\lessdot} \psi, \ominus^{\lessdot} \psi, \chi_P^{\lessdot}(\ominus_H^d \psi), \ominus^{\lessdot}(\ominus_H^d \psi)$ to $\mathrm{Cl}(\varphi)$, and impose the constraints that follow.
Let $(\Phi, \Theta, \Psi) \in \delta_{pop}$:

6. if $\chi_L, \chi_{\doteq} \notin \Psi_p$, then $(\chi_P^{\lessdot} \psi \in \Theta_c$ or $\ominus^{\lessdot} \psi \in \Theta_c)$ iff $\ominus_H^d \psi \in \Phi_p$;
7. if $\chi_L \notin \Psi_p$, then $\ominus_H^d \psi \in \Psi_p$ iff $(\chi_P^{\lessdot}(\ominus_H^d \psi) \in \Theta_c$ or $\ominus^{\lessdot}(\ominus_H^d \psi) \in \Theta_c)$;

8. if $\ominus_H^d \psi \in \Phi_p$, then $\chi_L, \chi_{\doteq} \notin \Psi_p$.

Let $(\Phi, a, \Psi) \in \delta_{push/shift}$:

9. if $\ominus_H^d \psi \in \Phi_c$, then $\chi_L \in \Psi_p$;
10. $\ominus_H^d \psi \notin \Phi_p$.

If $\bigcirc_H^u \psi \in \mathrm{Cl}(\varphi)$, we add the following rules.
Let $(\Phi, a, \Psi) \in \delta_{push}$:

11. if $\bigcirc_H^u \psi \in \Phi_c$, then $\chi_R \in \Phi_p$;
12. $\bigcirc_H^u \psi \in \Phi_p$ iff $\psi \in \Phi_c$ and $\chi_R \in \Phi_p$.

Let $(\Phi, \Theta, \Psi) \in \delta_{pop}$:

13. if $\chi_R \in \Theta_p$, then $\bigcirc_H^u \psi \in \Theta_c$ iff $\bigcirc_H^u \psi \in \Psi_p$;
14. $\bigcirc_H^u \psi \notin \Phi_p$.

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

15. $\bigcirc_H^u \psi \notin \Phi_p$ and $\bigcirc_H^u \psi \notin \Phi_c$.

If $\ominus_H^u \psi \in \mathrm{Cl}(\varphi)$, we add the following rules.
Let $(\Phi, a, \Psi) \in \delta_{push}$:

16. if $\ominus_H^u \psi \in \Phi_c$, then $\chi_R, \chi_L \in \Phi_p$.

Let $(\Phi, \Theta, \Psi) \in \delta_{pop}$:

17. if $\chi_L \in \Psi_p$, then $\ominus_H^u \psi \in \Psi_c$ iff $\psi \in \Theta_c$ and $\chi_R \in \Theta_p$;

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

18. $\ominus_H^u \psi \notin \Phi_c$.

Hierarchical Until and Since

If $\psi \, \mathcal{U}_H^d \, \theta \in \mathrm{Cl}(\varphi)$, then we add $\psi, \theta, \chi_F^{\gtrdot} \top, \bigcirc_H^d (\psi \, \mathcal{U}_H^d \, \theta)$ to $\mathrm{Cl}(\varphi)$. For any $\Phi \in \mathrm{Atoms}(\varphi)$, $\psi \, \mathcal{U}_H^d \, \theta \in \Phi_c$ iff:

1. $\theta, \chi_F^{\gtrdot} \top \in \Phi_c$ or
2. $\psi, \bigcirc_H^d (\psi \, \mathcal{U}_H^d \, \theta) \in \Phi_c$.

If $\psi \, \mathcal{S}_H^d \, \theta \in \mathrm{Cl}(\varphi)$, then we add $\psi, \theta, \chi_F^{\gtrdot} \top, \ominus_H^d (\psi \, \mathcal{S}_H^d \, \theta)$ to $\mathrm{Cl}(\varphi)$. For any $\Phi \in \mathrm{Atoms}(\varphi)$, $\psi \, \mathcal{S}_H^d \, \theta \in \Phi_c$ iff:

3. $\theta, \chi_F^{\gtrdot}\top \in \Phi_c$ or

4. $\psi, \ominus_H^d(\psi \, \mathcal{S}_H^d \, \theta) \in \Phi_c$.

If $\psi \, \mathcal{U}_H^u \, \theta \in \mathrm{Cl}(\varphi)$, we add $\psi, \theta, \chi_P^{\lessdot}\top, \bigcirc_H^u(\psi \, \mathcal{U}_H^u \, \theta)$ to $\mathrm{Cl}(\varphi)$. For any $\Phi \in \mathrm{Atoms}(\varphi)$, $\psi \, \mathcal{U}_H^u \, \theta \in \Phi_c$ iff:

5. $\theta, \chi_P^{\lessdot}\top \in \Phi_c$ or

6. $\psi, \bigcirc_H^u(\psi \, \mathcal{U}_H^u \, \theta) \in \Phi_c$.

If $\psi \, \mathcal{S}_H^u \, \theta \in \mathrm{Cl}(\varphi)$, then we add $\psi, \theta, \chi_P^{\lessdot}\top, \ominus_H^u(\psi \, \mathcal{S}_H^u \, \theta)$ to $\mathrm{Cl}(\varphi)$. For any $\Phi \in \mathrm{Atoms}(\varphi)$, $\psi \, \mathcal{S}_H^u \, \theta \in \Phi_c$ iff

7. $\theta, \chi_P^{\lessdot}\top \in \Phi_c$ or

8. $\psi, \ominus_H^u(\psi \, \mathcal{S}_H^u \, \theta) \in \Phi_c$.

Chapter 3

POMC

This chapter describes POMC (Precedence Oriented Model Checker), an early-stage software library and application targeted at OPL-based formal verification.

In its current version (`0.1.0.0`), POMC features an optimized implementation of the automata-based runtime verification procedure of POTL formulas on OP strings sketched in section 2.3. As its name suggests though, the development of POMC was carried out with the intention of paving the way for a future extension into a complete OPL model checker as well.

The chapter is organized like so: section 3.1 lists the main functionalities offered by POMC, section 3.2 discusses the fundamental design choices that have characterized its development, section 3.3 describes POMC's architecture and inner workings, section 3.4 discusses the most effective optimizations devised to boost its performance, and section 3.5 describes the format of its input files.

3.1 Functionalities

As will later be discussed in more detail, the POMC library can be used to:

- represent a generic OPA and...
- ...run it against arbitrary input (optionally in parallel);

- represent a generic POTL formula and…
- …check whether it holds for an arbitrary string.

On the other hand, the POMC application takes as input a text file with:

- an OPM,
- a list of POTL formulas,
- a list of text strings,

defined in a high-level domain-specific language, and it determines whether each formula is satisfied or not by each individual string, printing the results as output.

3.2 Design choices

POMC has been developed with the objective of building a useful research prototype, meant to explore the possibilities opened by OPL-based verification. For this reason, generality and extensibility have been regarded as important non-functional goals throughout the entire development process. The ability to handle a wide variety of use cases has been prioritized, even at the cost of some minor performance trade-offs. Furthermore, the architecture of POMC has been designed to be as modular as possible, separating concerns into different self-contained components that can be easily readapted or composed to build future extensions.

A key decision was to develop POMC using Haskell, a purely functional, lazy, statically-typed programming language [15]. While Haskell's high level of abstraction and type safety have been vastly beneficial to address the difficulties and the complexity of creating a verification tool, the maturity of the Haskell compiler machinery made it possible to reach quite satisfying levels of performance in terms of execution time and memory footprint. In light of the above, the trade-off connected to relying on an abstract, garbage-collected language, as opposed to e.g. a lower level imperative language with explicit memory management such as C, seems strongly justified. Furthermore, any hypothetical future reimplementation targeted at maximal efficiency, could still be facilitated by using the existing POMC codebase as a reference.

The purely functional essence of the chosen programming language, has naturally led to a strong adherence to the functional paradigm. Core features have been implemented as higher-order, polymorphic functions, a well-established practice to produce reusable and modular code in the functional realm [10][17]. Thanks to the strictness of the Haskell type system, the API of POMC comes with strong safety guarantees: the vast majority of the functions exported by the library are free of side effects, while the few input/output interactions have been safely confined to the IO monad.

3.3 Architecture

3.3.1 Library

Given the full adherence to the functional paradigm of the POMC library, it is best to describe its architecture through its most relevant data types and function signatures.

The three kinds of precedence relations are defined as a simple enumeration-like type:

```
data Prec = Yield | Equal | Take
```

The cornerstone of the whole architecture is the run function; given the essential elements of an OPA and an input string, it determines whether the string is accepted by the OPA or not. The `Opa.hs` module includes several variants of `run`, which have been used for experimental purposes. For now we focus on the most basic one:

```
run :: (t -> t -> Maybe Prec)
    -> [s]
    -> (s -> bool)
    -> (s -> t -> [s])
    -> (s -> t -> [s])
    -> (s -> s -> [s])
    -> [t]
    -> Bool
```

The input for `run` is: 1. A precedence function, associating two tokens with a nullable precedence; 2. A list of states; 3. A function telling whether a state is final; 4. A δ_{shift} function; 5. A δ_{push} function; 6. A δ_{pop} function; 7. A list of input tokens. The function returns **`True`** if it finds an accepting computation, **`False`** otherwise. Notice how partial OPMs are supported through the `Maybe` type, allowing the precedence function to return `Nothing` where it is not defined: computation paths where an incompatible token is encountered are treated as non-accepting ones.

The definition of `run` is polymorphic in `s` and `t`: the two placeholders respectively represent a state type and a token type; this convention is consistently used for all other type signatures in the library.

The atomic proposition data type follows the principle of generality as well: it wraps an inner, generic proposition type `a` (which will also appear consistently in the following definitions), adding the 0-ary constructor `End` to represent the # OP symbol:

```
data Prop a = Prop a | End
```

Before examining the checking function, it is necessary to introduce a useful typeclass that represents an abstract specification:

```
class Checkable c where
  toReducedPotl :: c a -> RPotl.Formula a
```

What the typeclass asks from a specification type `c` (parametrized with proposition type `a`) is to support a conversion towards the Formula data type contained in the `RPotl.hs` module. It is a tree-like type that represents a POTL formula with a reduced set of ST-based low-level operators, for which a direct checking construction is supported.

Instead of specifying the **`data`** definition of `RPotl.Formula`, we present the definition of `PotlV2.Formula`, a similar but higher-level POTL formula representation located in the `PotlV2.hs` module. It supports all the relevant high-level POTL and LTL operators. Of course, it is an instance of `Checkable`.

```
data Dir = Up | Down

data Formula a = T
               | Atomic (Prop a)
               | Not (Formula a)
               | Or      (Formula a) (Formula a)
               | And     (Formula a) (Formula a)
               | Xor     (Formula a) (Formula a)
               | Implies (Formula a) (Formula a)
               | Iff     (Formula a) (Formula a)
               | PNext Dir (Formula a)
               | PBack Dir (Formula a)
               | XNext Dir (Formula a)
               | XBack Dir (Formula a)
               | HNext Dir (Formula a)
               | HBack Dir (Formula a)
               | Until  Dir (Formula a) (Formula a)
               | Since  Dir (Formula a) (Formula a)
               | HUntil Dir (Formula a) (Formula a)
               | HSince Dir (Formula a) (Formula a)
               | Eventually (Formula a)
               | Always (Formula a)
```

As one might deduce, the first data constructor for Formula represents $\top$, the second is just a wrapper for `Prop`, and the others represent respectively the operators $\neg, \vee, \wedge, \veebar, \Rightarrow, \Leftrightarrow, \bigcirc^t, \circleddash^t, \chi_F^t, \chi_P^t, \bigcirc_H^t, \circleddash_H^t, \mathcal{U}_\chi^t, \mathcal{S}_\chi^t, \mathcal{U}_H^t, \mathcal{S}_H^t, \Diamond, \Box$, with $t \in \{u, d\}$ being represented by the enumeration type `Dir`.

We can now discuss the other central function in the POMC library, that is `check`, from the `Check.hs` module:

```
check :: (Checkable f, Ord a, Show a)
      => f a
      -> (Set (Prop a) -> Set (Prop a) -> Maybe Prec)
      -> [Set (Prop a)]
      -> Bool
```

The function takes three arguments: a `Checkable` specification, a precedence function on sets of APs, and a list (or string) of sets of AP. It returns **`True`** if the formula holds for the input string, **`False`** otherwise. It does so by converting the specification to a `RPotl.Formula`, then it applies the construction described in subsection 2.3.1 to convert the formula to its equivalent OPA, and finally it checks whether running the constructed automaton against the specified token string produces an accepting computation.

To decompose the complexity of `check`, the work has been split in several self-contained auxiliary functions, and the `check` implementation does little more than gluing these building blocks together. It is worth rapidly presenting said helpers, since they will be crucial when describing the applied optimizations in section 3.4.

The `closure` function implements the closure operator Cl from the automaton construction procedure. It takes a `RPotl` formula, and a list of additional APs (these are, in the case of `check`, the APs contained in the input string, which must also be present in the closure). By traversing the input, it produces the set of all formulas which are relevant for the construction.

```
closure :: Ord a
        => Formula a
        -> [Prop a]
        -> Set (Formula a)
```

The `atoms` function implements instead the Atoms operator from the automaton construction.

```
atoms :: Ord a
      => Set (Formula a)
      -> Set (Set (Prop a))
      -> [Atom a]
```

What this function fundamentally does is taking the closure as input, and returning the list of its subsets which respect the validity constraints imposed by the rules on Atoms. These sets of formulas will then be used to construct the automaton states. For now, let us momentarily ignore the precise definition of `Atom` as well as the role of the second argument: these details will be clarified in section 3.4.

The `initials` function handles the creation of initial states.

```
initials :: (Ord a)
         => Formula a
         -> Set (Formula a)
         -> [Atom a]
         -> [State a]
```

The arguments are the formula to be checked, its closure and its atoms. The function returns a list of states, constructed by filtering the atoms according to the rules concerning initial states (e.g. the checked formula must be present). Here too we leave the details of the `State` type for the next section.

The final interesting building block is the `delta` function. We will omit the full signature since it is fairly long. However, its most relevant argument is a `RuleGroup`, a type containing all the transition rules that the caller wants `delta` to enforce. These are in the form of boolean predicates that, given the relevant context as input, return `True` if the rule they represent is satisfied and `False` otherwise. δ_{shift}, δ_{push} and δ_{pop} are simply implemented by currying `delta` with the necessary parameters (among which, the different rule groups), and are first-order Haskell functions themselves (as opposed to association maps stored in memory).

Going back to the `check` function, the auxiliary procedures are composed according to the following algorithm:

```
compute the closure
compute the atoms
compute the initial states
compute the curried delta functions
run the automaton on the input string
return the result
```

Notice how, thanks to the generality of `run` and to the way the δ functions are implemented, `check` does not need to build the actual automaton in advance in order to run it against a string. Rather, the exploration performed by `run` generates new states on the fly, while old ones are garbage collected as soon as possible.

The `Check.hs` module also provides another check function, `fastcheck`, that works in an analogous way, but that is more optimized and should be preferred over `check`. It will be discussed more thoroughly in section 3.4.

3.3.2 Application

The architecture of the POMC application is very straightforward, revolving around a simple sequential interaction.

At first, the user specifies the name of an input file as a command-line argument. The file must be written in the POMC input language (see section 3.5 for a precise specification), and it must contain three declarations: 1. a list of precedence relations on sets of APs; 2. a list of POTL formulas; 3. a list of input strings, composed of sets of APs.

They key element for the parsing part is the function `checkRequestP` from module `Parse.hs`. In adherence with traditional functional-style parsing, the function is built through the combination of simpler parser functions (with the help of the monadic parser combinator library Megaparsec [11]).

The result of the parsing operation is stored as a `CheckRequest`:

```
data CheckRequest = CheckRequest {
  creqPrecRels :: [(Set (Prop Text), Set (Prop Text), Prec)],
  creqFormulas :: [PotlV2.Formula Text],
  creqStrings  :: [[Set (Prop Text)]]
}
```

with AP being interpreted as simple Unicode strings of the `Text` type. Notice also how formulas are conveniently represented using the `PotlV2.Formula` high-level type.

At this point an actual precedence function still has to be built from the precedence relations. This is done using the `fromRelations` helper contained in the `Prec.hs` module. It builds a new precedence function that, given two tokens `t1` and `t2`, establishes their precedence by iteratively trying to match against the `(ts1, ts2, p)` relations specified by the user, where a relation matches if `ts1` is a subset of `t1` and `ts2` is a subset of `t2`; if no match is found, then **`Nothing`** is returned.

Lastly, each (formula, input string) couple is evaluated with the `fastcheck` function, measuring the execution time. Results are printed as output.

3.4 Optimizations

Atom encoding. When looking at the first general rule imposed on the Atoms operator in subsection 2.3.1, it is straightforward to think of a binary encoding to represent atoms, or at least to generate them efficiently. Let v be an array containing all the non-negated formulas in the closure. Then an atom can be represented by a binary vector of length size(v), where if the i-th value is 1, then the i-th formula in v is present, whereas if the i-th value is 0, then the i-th formula in v is present in its negated form.

Notice that multiple negations are not handled directly by this encoding. The adopted solution consists in a normalization of the input formula, before doing any other processing: the syntax tree of the formula is traversed, iteratively removing double negations. The obtained formula is equivalent to the input one, since, of course, $\neg\neg\psi \equiv \psi$ for every ψ.

The `atoms` function generates the relevant bit vectors one by one, decoding them and performing the necessary consistency checks. If the checks fail, the atom is discarded immediately along with its binary encoding. Otherwise it is kept in memory as `Atom`, a type which stores both the decoded and encoded form for convenience:

```
data Atom a = Atom { atomFormulaSet :: Set (Formula a)
                   , atomEncodedSet :: Vector Bit
                   }
```

State format. In the automaton construction of subsection 2.3.1, a state Φ corresponds to a couple $(\Phi_c, \Phi_p) \in \text{Atoms}(\varphi)^2$. While Φ_c must really be an atom, by looking at the construction it is clear that only a few operators are optionally needed in Φ_p, that is χ_F^π, χ_P^π with $\pi \in \{\lessdot, \doteq, \gtrdot\}$ and $\bigcirc_H^u, \bigcirc_H^d, \ominus_H^d$. Furthermore, χ_L, $\chi_{\doteq}$ and χ_R can only be in Φ_p. For this reason we can drop atom consistency rules for pending sets, and reduce them to the power set of the formulas starting with the aforementioned operators, combined with all possible configurations of the auxiliary symbols. The auxiliary function called by `check` which, given the closure as input, handles the creation of said combinations, is:

```
pendCombs :: (Ord a)
          => Set (Formula a)
          -> Set ((Set (Formula a), Bool, Bool, Bool))
```

Auxiliary symbols are treated as external boolean values, to further reduce the size of pending sets. This way, there is no need of inserting them in the closure. In light of this, we present the data type representing a state:

```
data State a = State
    { current   :: Atom a
    , pending   :: Set (Formula a)
    , mustPush  :: Bool
    , mustShift :: Bool
    , afterPop  :: Bool
    }
```

where the three boolean flags represent respectively χ_L, $\chi_{\doteq}$ and χ_R.

AP-based filtering. A big advantage for the `check` function comes from the fact that its input consists of finite strings which are known a priori. Given how the automaton construction works, an atom can be possibly useful only if the APs it contains represent the AP set corresponding to a position of the input string, or if it is one the "final" atoms which contain # as their only AP. This opens the possibility for a major optimization: the set of possible AP sets is passed as the second argument of the `atoms` function; this way, atoms characterized by an AP configuration which is useless with respect to the input AP sets, can be immediately discarded. This results in a significant shrinkage of the atom space, noticeably improving performance.

Laziness. A quite significant optimization comes almost "for free" thanks to the laziness of the Haskell language: the auxiliary functions of `check`, most importantly `atoms` and `initials`, are structured so that they never enforce the sequential generation of the whole atom set (with some precautions taken to prevent the phenomenon of "thunking", that is the accumulation of significantly memory-heavy amount of unexpanded expressions). This results in atoms being generated as needed during the exploration performed by the run function. This way, an accepting run can terminate without generating the entire state space.

Plug-in rules. Almost all rules involved in the delta transitions are only useful in the presence of the operators they concern. The function `deltaRules` takes advantage of this fact:

```
deltaRules :: (Ord a)
           => Set (Formula a)
           -> (RuleGroup a, RuleGroup a, RuleGroup a)
```

it takes the closure as input, and, based on which operators are present, produces minimal sets of rules for δ_{shift}, δ_{push} and δ_{pop}. A similar mechanism is applied in the `atoms` function, so that only the relevant consistency rules on the Atoms operator are checked at run-time.

Token lookahead. This is another improvement based on the input string. By looking at a log of the execution of the `check` function, one realizes that many execution paths are terminated by the main rule on atomic propositions: $\forall(\Phi, a, \Psi) \in \delta_{push/shift}$, $\Phi_c \cap AP = a$. This is because the push and the shift transition functions do not know in advance the AP set corresponding to the following input token, and therefore cannot use this information to discriminate future states. To address this shortcoming, an alternative run function has been implemented, which exposes a lookahead of a single token to δ functions:

```
augRun :: (t -> t -> Maybe Prec)
       -> [s]
       -> (s -> Bool)
       -> (Maybe t -> s -> t -> [s])
       -> (Maybe t -> s -> t -> [s])
       -> (Maybe t -> s -> s -> [s])
       -> [t]
       -> Bool
```

The lookahead is wrapped in a `Maybe`, with `Nothing` values signaling that there will be no following tokens. A new checking function named `fastcheck` has then been created: it has the same signature of `check` but relies on `augRun` instead of `run`. It adds an additional rule to shift and push rules provided by `deltaRules`, which imposes that the next state must have the same APs as the lookahead token if it is not empty, # otherwise. This improvement is quite effective in terms of performance, reducing the non-determinism of the transition functions, avoiding lengthy sequences of useless moves.

Fast atom comparisons. As shown above, the `Atom` structure carries a binary representation. Whenever an equality check among two atoms has to be performed, it is sufficient to check if the respective binary representations are the same. This optimization is exploited to speed up the enforcement of the following generic rule on pop moves: $\forall(\Phi, \Theta, \Psi) \in \delta_{pop}$, $\Phi_c = \Psi_c$. Obviously, this is one of the most frequently checked rules, since it applies to all pop transitions. The performance gain is therefore non-negligible.

AP mapping. This is a minor optimization concerning the POMC application. Once the `CheckRequest` is parsed, its precedence function, formulas and strings are expressed in form of `Text` (Unicode strings). While `Text` is a reasonably efficient way of handling character strings (as opposed to the `String` type), operations like comparison, which in POMC can occur frequently (e.g. while comparing formulas), are still more efficient for a type like `Int`. For this reason, before calling the `check` function, the `CheckRequest` elements are parsed to collect all APs in a set; then the set is enumerated, creating a bijective mapping between `Text` APs and `Int`; finally the mapping is applied to the `CheckRequest` elements, to obtain a more compact and optimized version ready for the checking operation.

3.5 Input Language

Listing 3.1 shows an example of well-formed POMC input file, based on the OP alphabet of Figure 2.3.

Comments are lines prefixed with `//` or enclosed in `/* ... */`, as in C-style programming languages.

```
/* Example POMC file */

// Defining precedences for a stack trace OPL.
prec = call < call, call = ret, call < han, call > exc,
        ret > call,  ret > ret,  ret > han,  ret > exc,
        han < call,  han > ret,  han < han,  han = exc,
        exc > call,  exc > ret,  exc > han,  exc > exc, * > #;

// Defining POTL formulas.
formulas = pa,
           F (call And (XNu exc)),
           PNd (PNd pb),
           G (exc --> XBu call);

// Defining input strings.
strings = call han call call call exc call ret call ret ret,
          (call pa) han (call pb) exc ret,
          call han call exc call ret call ret ret;
```

Listing 3.1: An example of POMC input file.

A generic set of atomic propositions is expressed as a comma-separated list of strings enclosed by round brackets. Conveniently, brackets can be omitted with singleton sets. So the set $\{\mathbf{call}, p_A\}$ can be written as `(call, pa)`, while the singleton set $\{\mathbf{han}\}$ can be written as just `han`. The special character `#` can be used to indicate the $\#$ atomic proposition.

The **`prec`** definition establishes a set of precedence relations among sets of atomic propositions. The precedence keywords `<`, `=` and `>` correspond respectively to the token precedence relations $\lessdot, \doteq, \gtrdot$. After **`prec`** `=` the user has to supply a comma-separated semicolon-terminated list of precedence relations between atomic proposition sets. `*` is a wildcard matching every set. It is recommended to always specify a rule such as `* > #`, so that formula sets containing the $\#$ proposition are handled as expected.

The **`formulas`** definition has to be followed by a comma-separated semicolon-terminated list of POTL formulas. The syntax and precedence rules for operators is specified in Table 3.1; note how all binary operators are expressed in infix notation. Strings that are not operators are interpreted as atomic propositions. As customary in mathematical notation, round brackets are used to induce a different formula evaluation order.

The **`strings`** definition consists of a comma-separated semicolon-terminated list of input string. Each string is expressed as a whitespace-separated sequence of atomic proposition sets.

All three main definitions must of course be present and non-empty, and their order has to be that shown by the example.

Group	POTL Operator	POMC Operator	Notation	Associativity
Unary	$\neg$	`~`, `Not`	Prefix	–
	$\bigcirc^d$	`PNd`	Prefix	–
	$\bigcirc^u$	`PNu`	Prefix	–
	$\ominus^d$	`PBd`	Prefix	–
	$\ominus^u$	`PBu`	Prefix	–
	χ_F^d	`XNd`	Prefix	–
	χ_F^u	`XNu`	Prefix	–
	χ_P^d	`XBd`	Prefix	–
	χ_P^u	`XBu`	Prefix	–
	$\bigcirc_H^d$	`HNd`	Prefix	–
	$\bigcirc_H^u$	`HNu`	Prefix	–
	$\ominus_H^d$	`HBd`	Prefix	–
	$\ominus_H^u$	`HBu`	Prefix	–
	$\Diamond$	`F`, `Eventually`	Prefix	–
	$\Box$	`G`, `Always`	Prefix	–
POTL Binary	$\mathcal{U}_\chi^d$	`Ud`	Infix	Right
	$\mathcal{U}_\chi^u$	`Uu`	Infix	Right
	$\mathcal{S}_\chi^d$	`Sd`	Infix	Right
	$\mathcal{S}_\chi^u$	`Su`	Infix	Right
	$\mathcal{U}_H^d$	`HUd`	Infix	Right
	$\mathcal{U}_H^u$	`HUu`	Infix	Right
	$\mathcal{S}_H^d$	`HSd`	Infix	Right
	$\mathcal{S}_H^u$	`HSu`	Infix	Right
Prop. Binary	$\wedge$	`And`, `&&`	Infix	Left
	$\vee$	`Or`, `||`	Infix	Left
	$\veebar$	`Xor`	Infix	Left
	$\Rightarrow$	`Implies`, `-->`	Infix	Right
	$\Leftrightarrow$	`Iff`, `<-->`	Infix	Right

Table 3.1: POTL operators supported by POMC, in descending order of precedence. Operators listed on the same line are synonyms. Operators in the same group have the same precedence.

Chapter 4

Experiments

This chapter will discuss some experimental results concerning POMC. In particular, section 4.1 reports the results of benchmarks concerning the verification of individual POTL operators, while section 4.2 presents some more complex verification examples regarding relevant properties of stack traces.

All tests have been conducted on the same machine, mounting an AMD A8-5600K processor (3.9 GHz) with 8 GiB of RAM (DDR3, 667 MHz), running a GNU/Linux operating system.

4.1 Operator Benchmarks

Some valuable insights concerning the performance of POMC can be obtained by analyzing the performance of the verification of each individual POTL operator expressible with the input language.

Below, the results of a batch of operator benchmarks are reported, where the `fastcheck` function has been executed on specific formulas, selected to isolate the execution time and memory footprint connected to every POTL-specific fundamental operator (until/since-like operators have been omitted, since their performance depends directly on the operators they are expanded as).

Table 4.1 reports the running times of the operator benchmarks, while Table 4.2 reports their memory allocation. Note that the reported memory values describe the cumulative amount of memory allocated by the `fastcheck` function throughout its entire execution, and **not** the peak amount of allocated physical memory, which has been well under 1 GiB during all benchmark cases, even accounting for the whole POMC process. Tables present results with respect to the targeted operator and length ℓ of the input string of each case. For each operator, both an accepting and a rejecting case are provided, with results laid out on the same row.

All benchmark cases are based on the OP alphabet of stack traces (Figure 2.3). The input strings vary according to the operator type, and are designed to stress-test the checking procedure as much as possible:

$\bigcirc^t, \ominus^t$. `call han call han ... (call x) (ret y) ... exc ret exc ret`

χ_F^t, χ_P^t. `(call x) call han call han ... exc ret exc ret (ret y)`

$\bigcirc_H^d, \ominus_H^d$. `(call x) (call y) call call han call han ... exc ret exc ret exc`

$\bigcirc_H^u, \ominus_H^u$. `call call han call han ... exc ret exc ret (call x) ret (call y)`

Every case formula relates the `x` and the `y` propositions using the specific operator to be assessed, e.g. $\Diamond(y \wedge \chi_P^u x)$ for χ_P^u. The $\Diamond$ operator is prepended to all formulas, to make sure that they can "reach" the relevant position of the input string.

By looking at the gathered data, one can immediately notice how, by virtue of the nature of the algorithm, accepting runs are consistently faster than rejecting ones: rejecting implies having explored the entire space of execution paths, while accepting is possible as soon as a terminating path is found. As discussed in section 3.4, some of the applied optimizations contribute to making this time gap even more pronounced.

Interestingly, the measurements highlight non-negligible performance differences between operators, with some of them, namely $\chi_F^d, \chi_F^u, \bigcirc_H^d, \ominus_H^d$, looking markedly less efficient than the others. In particular this "slow group" of operators exhibits an exponential increase in execution time and allocated memory with respect to the length ℓ of the input string, despite the cardinality of the closure (and, by consequence, of the state set) staying constant. On the contrary, the performance of "fast" operators is much less sensitive to the size of the input: at increase of ℓ, benchmark data show a growth trend for both time and memory which is only linear, rather than exponential. We speculate that this difference is due to the very nature of the automaton construction, possibly because the propagation of the checks connected with the "slow operators" introduces more non-determinism. The performance gap can be clearly observed by comparing Figure 4.1 with Figure 4.2: the former contains the time and memory plots the $\bigcirc_H^u$ case, while the latter displays those of the $\bigcirc_H^d$ operator. Notice how with the $\bigcirc_H^u$ operator, despite the input string reaching a length $\ell > 200$, the execution time remains in the order of a few milliseconds, while in the $\bigcirc_H^d$ case the execution time becomes significant much more quickly. Notice also how, for both the operators, the time graph is highly resemblant to the memory one, suggesting a tight link between these two quantities.

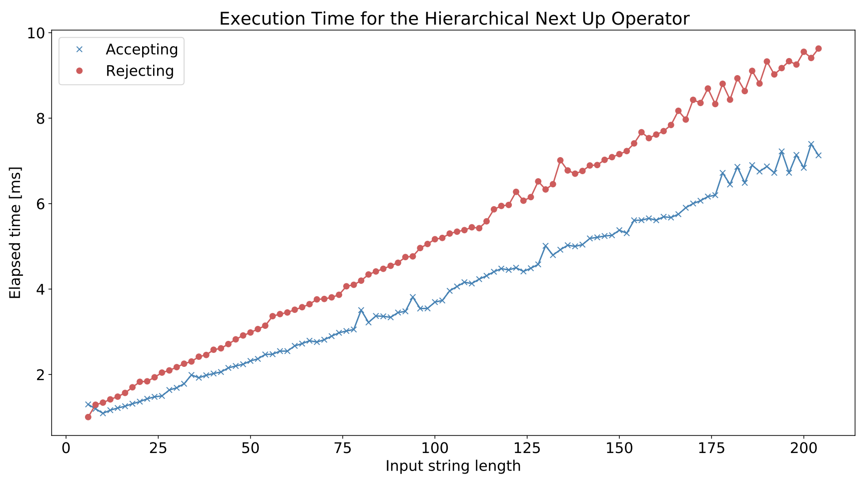
In any case, it must be noted that the performance of POMC can be considerably sensitive to the shape of the input strings. For example, the χ_F^d acceptance cases execute in a dramatically smaller amount of time if the supplied input strings are of the type: `call han ... (call x) han (ret y) ... exc ret`, characterized by `x` and `y` being at a fixed, close distance. While no significant improvement is observed with rejecting strings, the speed of accepting runs is hugely boosted (for $\ell = 17$, execution time is around $6ms$) and scales much better with the length of the input. In light of the above, one should keep in mind that the performance results displayed in this section attempt to address the performance of the worst case rather than the average one, striving to represent the most "problematic" inputs.

Operator	$\ell = 8$	$\ell = 10$	$\ell = 12$	$\ell = 14$	$\ell = 16$
$\bigcirc^d$ (acc.)	979.7 μs	1.009 ms	1.082 ms	1.085 ms	1.218 ms
$\bigcirc^d$ (rej.)	1.204 ms	1.328 ms	1.412 ms	1.467 ms	1.626 ms
$\bigcirc^u$ (acc.)	973.1 μs	994.3 μs	1.066 ms	1.125 ms	1.202 ms
$\bigcirc^u$ (rej.)	1.206 ms	1.256 ms	1.426 ms	1.459 ms	1.620 ms
$\ominus^d$ (acc.)	970.0 μs	1.015 ms	1.082 ms	1.157 ms	1.196 ms
$\ominus^d$ (rej.)	1.017 ms	1.063 ms	1.100 ms	1.198 ms	1.203 ms
$\ominus^u$ (acc.)	945.5 μs	1.012 ms	1.103 ms	1.118 ms	1.127 ms
$\ominus^u$ (rej.)	991.2 μs	1.071 ms	1.125 ms	1.185 ms	1.218 ms
χ_F^d (acc.)	87.90 ms	426.4 ms	2.109 s	9.692 s	44.33 s
χ_F^d (rej.)	661.3 ms	3.306 s	15.94 s	76.05 s	340.2 s
χ_F^u (acc.)	25.58 ms	93.45 ms	363.7 ms	1.442 s	5.768 s
χ_F^u (rej.)	85.34 ms	335.0 ms	1.341 s	5.310 s	21.26 s
χ_P^d (acc.)	3.109 ms	3.239 ms	3.384 ms	3.504 ms	3.683 ms
χ_P^d (rej.)	3.327 ms	3.576 ms	3.704 ms	3.935 ms	4.113 ms
χ_P^u (acc.)	12.43 ms	12.97 ms	13.45 ms	14.05 ms	14.89 ms
χ_P^u (rej.)	13.73 ms	14.50 ms	16.06 ms	16.32 ms	17.24 ms
$\bigcirc_H^d$ (acc.)	16.89 ms	38.45 ms	89.50 ms	204.5 ms	465.1 ms
$\bigcirc_H^d$ (rej.)	46.00 ms	112.0 ms	273.0 ms	643.4 ms	1.467 s
$\bigcirc_H^u$ (acc.)	990.1 μs	1.061 ms	1.095 ms	1.128 ms	1.198 ms
$\bigcirc_H^u$ (rej.)	1.228 ms	1.297 ms	1.361 ms	1.436 ms	1.501 ms
$\ominus_H^d$ (acc.)	10.11 ms	18.01 ms	33.64 ms	63.36 ms	124.4 ms
$\ominus_H^d$ (rej.)	45.41 ms	91.32 ms	183.9 ms	371.4 ms	740.2 ms
$\ominus_H^u$ (acc.)	883.3 μs	960.7 μs	998.4 μs	1.044 ms	1.115 ms
$\ominus_H^u$ (rej.)	946.6 μs	1.028 ms	1.072 ms	1.121 ms	1.212 ms

Table 4.1: Execution time for the verification of different POTL operators.

Operator	$\ell = 8$	$\ell = 10$	$\ell = 12$	$\ell = 14$	$\ell = 16$
$\bigcirc^d$ (acc.)	1.17MiB	1.24MiB	1.31MiB	1.38MiB	1.44MiB
$\bigcirc^d$ (rej.)	1.47MiB	1.6MiB	1.72MiB	1.85MiB	1.97MiB
$\bigcirc^u$ (acc.)	1.17MiB	1.24MiB	1.3MiB	1.37MiB	1.44MiB
$\bigcirc^u$ (rej.)	1.47MiB	1.6MiB	1.73MiB	1.85MiB	1.98MiB
$\ominus^d$ (acc.)	1.16MiB	1.24MiB	1.31MiB	1.39MiB	1.46MiB
$\ominus^d$ (rej.)	1.21MiB	1.3MiB	1.38MiB	1.46MiB	1.54MiB
$\ominus^u$ (acc.)	1.16MiB	1.24MiB	1.31MiB	1.39MiB	1.46MiB
$\ominus^u$ (rej.)	1.22MiB	1.3MiB	1.38MiB	1.46MiB	1.54MiB
χ_F^d (acc.)	123.7MiB	612.53MiB	2.89GiB	13.59GiB	62.53GiB
χ_F^d (rej.)	969.88MiB	4.69GiB	22.37GiB	104.02GiB	474.2GiB
χ_F^u (acc.)	36.01MiB	131.48MiB	512.82MiB	1.99GiB	7.95GiB
χ_F^u (rej.)	118.33MiB	462.42MiB	1.8GiB	7.17GiB	28.68GiB
χ_P^d (acc.)	4.57MiB	4.77MiB	4.98MiB	5.19MiB	5.41MiB
χ_P^d (rej.)	4.96MiB	5.23MiB	5.5MiB	5.77MiB	6.05MiB
χ_P^u (acc.)	19.61MiB	20.43MiB	21.25MiB	22.07MiB	22.9MiB
χ_P^u (rej.)	21.82MiB	23.01MiB	24.19MiB	25.38MiB	26.56MiB
$\bigcirc_H^d$ (acc.)	27.83MiB	65.3MiB	154.12MiB	359.82MiB	827.82MiB
$\bigcirc_H^d$ (rej.)	78.82MiB	198.85MiB	483.21MiB	1.11GiB	2.57GiB
$\bigcirc_H^u$ (acc.)	1.27MiB	1.34MiB	1.42MiB	1.49MiB	1.56MiB
$\bigcirc_H^u$ (rej.)	1.59MiB	1.69MiB	1.78MiB	1.89MiB	1.98MiB
$\ominus_H^d$ (acc.)	16.81MiB	30.22MiB	56.64MiB	109.03MiB	213.41MiB
$\ominus_H^d$ (rej.)	78.12MiB	158.59MiB	319.54MiB	641.43MiB	1.26GiB
$\ominus_H^u$ (acc.)	1.12MiB	1.18MiB	1.25MiB	1.32MiB	1.39MiB
$\ominus_H^u$ (rej.)	1.21MiB	1.3MiB	1.37MiB	1.45MiB	1.54MiB

Table 4.2: Cumulative amount of allocated memory for the verification of different POTL operators.



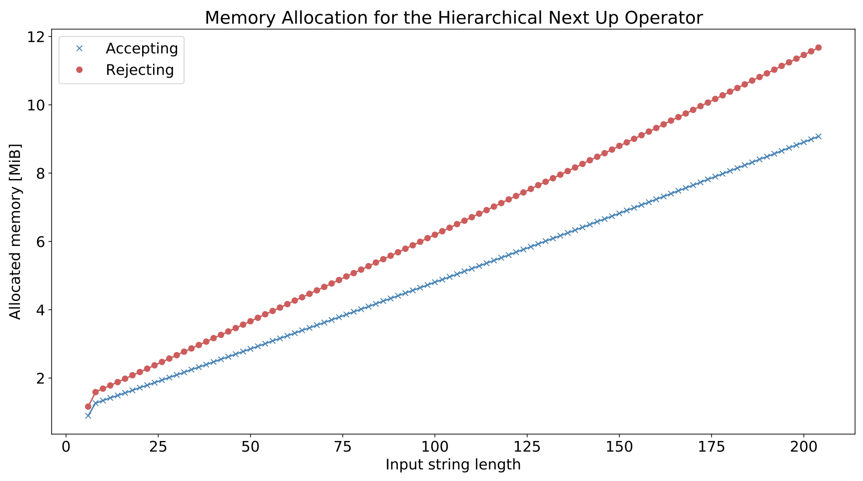
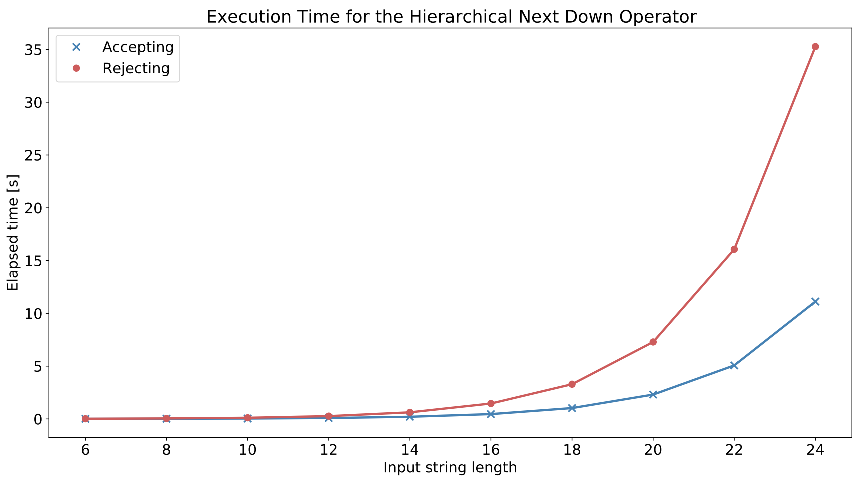


Figure 4.1: Graphs representing the execution time and the memory allocation of the $\bigcirc_H^u$ operator benchmarks.



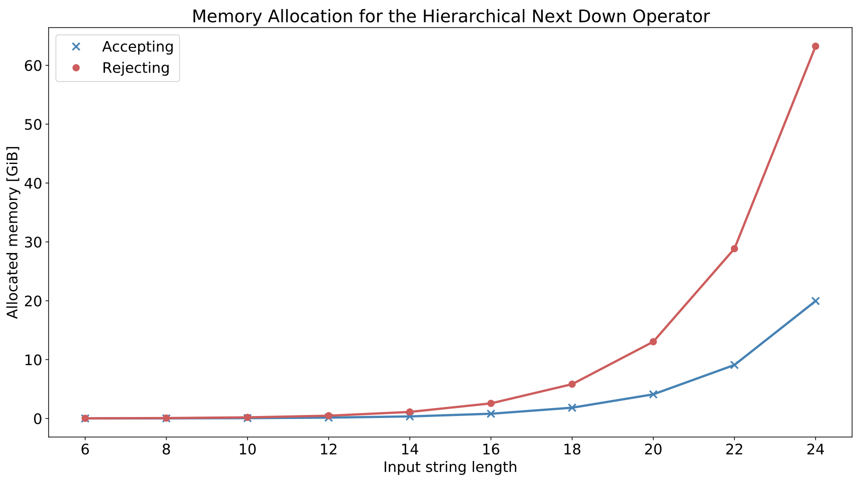


Figure 4.2: Graphs representing the execution time and the memory allocation of the $\bigcirc_H^d$ operator benchmarks.

4.2 Checking Stack Trace Properties

Thanks to the expressivity of POTL, POMC can be used to verify a wide range of specifications. This section provides some examples of the stack trace properties that can be enforced with POMC, relying on the OP alphabet of Figure 2.3. Each one has been tested against a few input strings using the POMC application, and results are reported.

Hoare-style pre/post conditions

A way to formally specify the behavior of a procedure is to specify a set of preconditions, that must hold when it is called, and a set of postconditions, that must hold after the function has terminated [9]. This type of specification can be expressed with a formula such as:

$$\square(\mathbf{call} \wedge \mathrm{p}_A \wedge \mathrm{pre} \Rightarrow \chi_F^d(\mathbf{ret} \wedge \mathrm{post}) \vee \bigcirc^d(\mathbf{ret} \wedge \mathrm{post})),$$

requesting that for every call to a procedure of type A for which the preconditions hold, postconditions must hold at termination. Notice how pre and post conditions are represented as simple atomic propositions in this case, but they could be substituted with any subformula.

Input string	Result	Elapsed
`(call pa pre) han (ret post)`	True	45.71 ms
`(call pa pre) han (ret)`	False	49.90 ms
`(call pa pre) han (call pa) han exc ret exc (ret post)`	True	66.13 ms
`(call pa pre) han (call pa pre) han exc ret exc (ret post)`	False	336.4 ms

Exception specification

It is possible to enforce the type of exception that can terminate a specific procedure. For example, with the formula:

$$\square(\mathbf{call} \wedge \mathrm{p}_A \wedge (\bigcirc^u\mathbf{exc} \vee \chi_F^u\mathbf{exc}) \Rightarrow (\bigcirc^u\mathrm{e}_B \vee \chi_F^u\mathrm{e}_B)),$$

one can specify that, whenever a procedure of type A is terminated by an exception, that exception must be of type B.

Input string	Result	Elapsed
`(call pa) (exc eb)`	True	856.7 ms
`(call pa) exc`	False	868.7 ms
`call (call pa) (exc eb) (call pa) han (call pa) call (exc eb) ret`	True	15.73 s
`call (call pa) (exc eb) (call pa) han (call pa) call exc ret`	False	212.8 s

Data access

By relying on properly annotating traces, one can impose data consistency conditions. For example, formula:

$$\square(\mathbf{call} \wedge \mathrm{p}_A \wedge (\neg\mathbf{ret}\, \mathcal{U}^d_\chi\, \mathrm{wr}_x) \Rightarrow \chi^d_F \mathbf{exc})$$

imposes that, if a procedure of type A or its subprocedures write to a variable x (an operation symbolized by the wr_x atomic proposition), then they are terminated by an exception.

Input string	Result	Elapsed
`(call pa) (call pb WRx) exc`	True	262.2 ms
`(call pa) (call pb WRx) (ret pb) (ret pa)`	False	1.665 s
`call han (call pa) (call pb) (call WRx) (call pc) exc ret`	True	161.9 s
`call han (call pa) han (call pb WRx) (ret pb) (ret pa) ret`	False	123.8 min

Notice how the execution time is quite high the last rejecting case. This is due to the lengthiness of the input string combined with a quite large closure and the presence of a compound operator, the $\mathcal{U}^d_\chi$.

Regular termination

The formula:

$$\square(\mathbf{call} \wedge \mathrm{p}_A \Rightarrow \neg(\bigcirc^u \mathbf{exc} \vee \chi^u_F \mathbf{exc}))$$

states that procedures of type A must not be terminated by an exception.

Input string	Result	Elapsed
`(call pa) (call pb) (ret pb) (ret pa)`	True	17.87 ms
`(call pa) (call pb) (ret pb) exc`	False	18.8 ms
`(call pa) han (call pb) (call pb) (call pc) call exc (ret pa)`	True	248.5 ms
`(call pa) han (call pb) (call pa) (call pc) call exc (ret pa)`	False	122.3 ms

Stack inspection

An interesting possibility is assessing properties on the stack content at a certain point of execution. For example, with formula:

$$\square(\mathbf{call} \land \mathrm{p}_B \land (\top \, \mathcal{S}_\chi^d \, (\mathbf{call} \land \mathrm{p}_A)) \Rightarrow \bigcirc^u \mathbf{exc} \lor \chi_F^u \mathbf{exc}),$$

it is requested that, if a procedure of type A is present in the stack when a procedure of type B is called, then the latter is terminated by an exception.

Input string	Result	Elapsed
`(call pa) (call pb) exc`	True	3.102 s
`(call pa) (call pb) (ret pb) (ret pa)`	False	3.586 s
`(call pa) han (call pb) call han ret call exc (ret pa)`	True	5.430 s
`(call pa) han (call pb) call ret call ret (ret pb) (ret pa)`	False	6.210 s

Chapter 5

Conclusions

This dissertation has presented POMC: an early-stage verification tool for Operator Precedence Languages.

The main achievement of POMC is the implementation of a recently devised algorithm to translate Precedence Oriented Temporal Logic formulas into equivalent Operator Precedence Automata.

It has been shown how this is leveraged by POMC to carry out runtime verification of POTL specifications on finite input strings. The technologies and the internal architecture that make this possible have been thoroughly discussed. Furthermore, with the assistance of empirical results, light has been shed on both the practical uses and the limitations of the POMC tool in its current version.

Finally, it has been suggested how POMC constitutes a first step towards the implementation of a complete model checker based on OP languages.

5.1 Future Work

The future developments for POMC are varied and interesting. The following list comprises some of the open possibilities.

- Relying on a generic binary encoding of formula sets contained in the closure for faster comparisons and membership checks, in the context of the automaton construction. Most transition rules could be updated to take advantage of this optimization, obtaining a particularly relevant performance boost in relation to checks that have to be performed frequently throughout the verification process.

- Extending POMC with actual model checking capabilities. In particular, to support a traditional automata-theoretic model checking procedure, intersection and emptiness check for Operator Precedence Automata should be implemented. These additions are currently under active development.
- Extending POMC to handle Operator Precedence ω-Languages [13], that is OP languages characterized by infinite strings. This would facilitate the formal modeling of systems whose behavior must be specified with respect to non-terminating sequences of events.
- Creating high-level simplified tools leaning on the POMC library to address domain-specific verification tasks. This way users could enjoy the benefits of powerful formal methods techniques, without having to delve into the theoretical complexity behind them.

Bibliography

[1] R. Alur, M. Arenas, P. Barceló, K. Etessami, N. Immerman, and L. Libkin. First-order and temporal logics for nested words. *LMCS*, 4(4), 2008.

[2] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS 2004*, pages 467–481. Springer, 2004.

[3] C. Baier, J.P. Katoen, and K.G. Larsen. *Principles of Model Checking*. Mit Press. MIT Press, 2008.

[4] A. Barenghi, S. Crespi Reghizzi, D. Mandrioli, F. Panella, and M. Pradella. Parallel parsing made practical. *Sci. Comput. Program.*, 112:195–226, 2015.

[5] M. Chiari, D. Mandrioli, and M. Pradella. A first-order complete temporal logic for model-checking structured context-free languages. Submitted, 2020.

[6] Michele Chiari, Dino Mandrioli, and Matteo Pradella. Potl: A first-order complete temporal logic for operator precedence languages. *CoRR*, abs/1910.09327, 2019.

[7] S. Crespi Reghizzi and D. Mandrioli. Operator Precedence and the Visibly Pushdown Property. *JCSS*, 78(6):1837–1867, 2012.

[8] R. W. Floyd. Syntactic Analysis and Operator Precedence. *JACM*, 10(3):316–333, 1963.

[9] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 26(1):53–56, January 1983.

[10] John Hughes. Why functional programming matters. *The Computer Journal*, 32:98–107, 1989.

[11] M. Karpov, P. Martini, D. Leijen, et al. Megaparsec library. Available online at https://github.com/mrkkrp/megaparsec.

[12] Orna Kupferman. Automata theory and model checking. In *Handbook of Model Checking*, pages 107–151. Springer, 2018.

[13] V. Lonati, D. Mandrioli, F. Panella, and M. Pradella. Operator precedence languages: Their automata-theoretic and logic characterization. *SIAM J. Comput.*, 44(4):1026–1088, 2015.

[14] D. Mandrioli and M. Pradella. Generalizing input-driven languages: Theoretical and practical benefits. *Computer Science Review*, 27:61–87, 2018.

[15] Simon Marlow et al. Haskell 2010 language report. 2010. Available online at https://www.haskell.org/onlinereport/haskell2010.

[16] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, page 46–57, USA, 1977. IEEE Computer Society.

[17] Simon Thompson. Higher-order + polymorphic = reusable, 1997. Available online at https://kar.kent.ac.uk/21504.