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

Temporal Logic for Operator Precedence Words

Candidato:
Michele Chiari
Matricola 881258

Relatore:
Chiar.mo Prof. Dino Mandrioli

Correlatore:
Chiar.mo Prof. Matteo Pradella

Anno Accademico 2017–2018

Prefazione e Ringraziamenti

Il lavoro presentato in questa tesi rappresenta il culmine di un'importante tappa della mia formazione accademica e personale, essendo anche risultato nella pubblicazione di un articolo presso la conferenza GandALF 2018 [CMP18]. Desidero porgere un sentito ringraziamento al Prof. Dino Mandrioli e al Prof. Matteo Pradella per avermi guidato nella realizzazione di questo elaborato e del suo contenuto in modo attento e sapiente. Desidero inoltre ringraziare tutti i docenti che hanno preso parte alla mia formazione scolastica e accademica, e che sono stati anch'essi indispensabili al raggiungimento di questo traguardo. Infine, ringrazio la mia famiglia e in particolare i miei genitori per il sostegno che mi hanno offerto.

Contents

Sommario

Lo scopo del *model checking* consiste nella verifica automatica di proprietà riguardanti il comportamento di un sistema hardware o software. La complessità delle proprietà da verificare e dei comportamenti da modellare dipende dal formalismo utilizzato per esprimere tali proprietà. I più classici di questi formalismi logici, in particolare LTL, CTL e CTL*, possono soltanto rappresentare proprietà riconducibili ai linguaggi regolari. Quindi, possono esprimere la semplice conseguenza temporale, l'eventualità e la necessità passate e future. Comunque, sono limitati alla struttura lineare dei linguaggi regolari. Recentemente, la letteratura in questo campo ha visto il tentativo di superare questa restrizione sfruttando famiglie più vaste di linguaggi strutturati, come i *visibly pushdown languages*, che permettono di modellare strutture annidate. Il lavoro presentato in questa tesi mira ad essere un significativo passo avanti in questa direzione, considerando la classe degli *operator precedence languages*. Essi furono originariamente introdotti da R. W. Floyd con l'intento di generare algoritmi di *parsing* efficienti per i linguaggi di programmazione. Per questo, possono rappresentare una varietà di strutture sintattiche piuttosto vasta, anche se dotate di una forma non immediatamente visibile. Ciononostante, godono delle fondamentali proprietà di chiusura che li rendono appropriati per il *model checking*. Con tale premessa, presentiamo un nuovo formalismo di logica temporale adatto a esprimere e a verificare automaticamente proprietà sugli *operator precedence languages*. Esploriamo la potenza espressiva di questo formalismo dimostrando che esso è almeno espressivo tanto quanto la sua principale controparte sui *visibly pushdown languages*, essendo inoltre basato su una classe di linguaggi ben più vasta. Infine, proponiamo una procedura per il *model checking* basata sulla teoria degli automi, su parole sia finite sia infinite.

Abstract

The aim of model checking consists in the automated verification of properties concerning the behavior of a software or hardware system. The complexity of the properties to be verified and, ultimately, of the behaviors to be modeled, depends on the formalism used to express such properties. The most classical of these logic formalisms, namely LTL, CTL and CTL*, can only tackle properties that can be described as regular languages. As such, they can express simple temporal consequence, past and future eventuality and necessity. However, they are constrained into the linear structure of regular languages. Recently, the literature in the field has seen the attempt at overcoming this restriction by exploiting larger families of structured languages, such as visibly pushdown languages, which enable the modeling of nested structures. The work presented in this thesis aims to represent a significant step forward in this direction, by considering the class of Operator Precedence Languages. They were introduced by R. W. Floyd with the purpose of generating efficient parsing algorithms for programming languages. For this reason, they can represent a rather large variety of syntactic structures, even with a shape not immediately visible. Nonetheless, they enjoy the fundamental closure properties that make them appropriate for model checking. On this premise, we present a novel temporal logic formalism suitable to express and automatically verify properties on operator precedence languages. We explore the expressive power of this formalism by proving that it is at least as expressive as its main counterpart for visibly pushdown languages, while being based on a significantly more powerful language family. Finally, we give an automata theoretic procedure for model checking, for both finite and infinite words.

Chapter 1

Introduction

The formal verification of software and hardware systems is one of the topics that inspired the most intense research efforts in computer science, and yet, it can be still considered an open one, at least up to a considerable extent. One of the historical branches of this research area was the one pioneered by R. W. Floyd and C. A. R. Hoare, which concerned the direct production of formal proofs for programs written in structured programming languages. In particular, Hoare Logic consists of a set of inference rules for program constructs, that can be combined in order to construct a rigorous formal proof of properties of programs containing such constructs. The programming languages targeted by these methods are, however, Turing complete. This, of course, guarantees their full generality, but it sacrifices the decidability of most of the very properties that these formalisms were conceived to prove. Therefore, there exists no complete and fully automated tool capable of building proofs by means of the Hoare method, leading to the need to resort to tools that may sometimes yield inconclusive results or, ultimately, to human ingenuity. While this methods sacrifice decidability to embrace full expressive power, another research branch developed toward the opposite side of this trade-off.

Indeed, using a less expressive formalism such as Finite State Machines (FSM) to model systems allows to exploit its closure and decidability properties, yielding the theoretical possibility of tools capable of automatically proving or disproving properties of the system. An early attempt at pursuing this route was made by J. R. Büchi [Bü60], C. C. Elgot [Elg61] and B. A. Trakhtenbrot [Tra61], who independently gave a logical characterization to regular languages by means of a Monadic Second Order logic (MSO). This logic formalism allowed the expression of properties on FSMs, with the theoretical possibility of proving or disproving them algorithmically. Unfortunately, the computational complexity of such decision algorithms is intractable [FG04], hindering the possibility of their practical use.

A considerable breakthrough in the field was the introduction of *model checking*. While keeping FSMs as the operational formalism to model systems, MSO logic was substituted with other formalisms that are less expressive, but that admit tractable decision procedures. The seminal work of A. Pnueli on Linear Temporal Logic (LTL) [Pnu77] is the first step in this direction. LTL expresses properties with respect to a linear, discrete timeline (linear order), and it has been proved to enjoy the same expressive power of First Order (FO) logic on such an algebraic structure by H. Kamp [Kam68]. Given the FSM model of a system, the model checking procedure for LTL allows to prove or disprove, by generating a counterexample, the truth of an LTL

formula on that system. The required computational complexity is exponential in the length of the LTL formula, which can be considered affordable if such a specification is not too long. Indeed, LTL model checking is PSPACE-complete [SC85]. Subsequently, other temporal formalisms were conceived with the same intent. Computational Tree Logic (CTL), by E. M. Clarke and E. A. Emerson, considers a timeline with branching points, intended to model systems whose behavior can take different execution paths. The computational complexity of the model checking procedure of CTL is linearly proportional to the length of the formula and to the size of the system to be verified. However, its expressiveness is not comparable with LTL, and for LTL properties that have a CTL counterpart, the former is exponentially more succinct. LTL and CTL are subsumed by CTL*, which can express all linear properties of LTL on the branching time that characterizes CTL. CTL* is also PSPACE-complete, as a consequence of including LTL.

LTL, CTL and CTL* are all limited to expressing regular properties, i.e. properties such that there exists a FSM accepting all and only strings conforming to them. This is, indeed, the classical way of performing LTL model checking: given an LTL formula φ, it is possible to build a nondeterministic FSM $\mathcal{A}$ with at most $2^{|\varphi|}$ states accepting all and only strings that satisfy φ [VW86]. Then, due to the closure and decidability properties of FSMs, it is possible to verify whether another system modeled as a FSM $\mathcal{A}'$ satisfies φ by building the automaton recognizing the intersection language between $\mathcal{A}'$ and the complement of $\mathcal{A}$. Finally, the obtained automaton can be tested for emptiness. If it is not empty, then it recognizes the behaviors of the system that do not satisfy φ.

FSMs accept the class of Regular Languages (RL). RL constitute the lower level of the Chomsky Hierarchy, being the least expressive. They have a typical linear structure, their characteristic trait being the repetition of patterns. Despite their simplicity, they found fruitful applications in several fields of computer science. They are, however, unsuitable for representing tree-like and nesting structures. For this reason, in compiler design they can only be used for lexical analysis, being unable to deal with the most common constructs of programming languages, that are typically nested. Context-Free Languages (CFL), situated at the level of the Chomsky Hierarchy just above RLs, are instead characterized by grammars sufficiently complex to describe the syntax of programming languages. This greater generality is achieved to the detriment of several important properties, that are instead enjoyed by RLs. For a start, CFLs lack efficient procedures for recognizing them: the most efficient parsing procedure for the full CFL class is the Earley algorithm [Ear70], with a cubic complexity. The theory of compiler construction overcame this problem by resorting to subclasses of CFLs: Deterministic CFLs (DCFL) can be recognized by LR parsers introduced by D. Knuth [Knu65], with a complexity linear in the length of the input string. A similar issue arises with respect to closure properties: CFLs are not closed by complement and intersection, and containment and equivalence are not decidable for them. These properties, that are all valid for RLs, are a fundamental requirement for model checking, and the lack thereof prevents the extension of model checking to the CFL class. The same can be said for DCFLs: although they are closed under complement, and equivalence is decidable for them, they lack closure under union and intersection, and containment is undecidable. So, it is not possible to follow for model checking the same route that was followed for programming language parsing.

The need of modeling processes more complex than those allowed by FSM, however, remains. Attempts at model checking pushdown systems, i.e. systems that show properties typical of CF languages (indeed, CFLs are recognized by pushdown au-

tomata) have been pursued with a variant of alternation-free modal mu-calculus (a branching time logic) in [BS92], and of LTL and CTL in [BEM97]. These works, however, express the properties to be proved by means of formalisms that are not more powerful than RLs.

A significant step forward was made by rediscovering one of the several subclasses of CFL that were introduced in the past in order to achieve efficient CF parsing procedures. The class of Input-Driven, or Visibly Pushdown Languages (VPL), while being strictly wider than RLs, retains all their important closure and decidability properties: VPLs are closed under the Boolean operations; emptiness, containment and language equivalence are decidable. Thanks to those properties, R. Alur and P. Madhusudan gave to VPLs a MSO logical characterization, followed by the development of temporal logics expressing VPL properties: CaRet and, subsequently, NWTL, the latter being proved to be First-Order complete. Such formalisms were developed, in particular, in order to model the stack of procedural programs. In their execution traces there are matched function calls and returns: this nesting structure could not be tackled by RLs, but VPLs are expressive enough to describe them. Indeed, this nesting structure is *visible* in VPLs: their alphabet is partitioned in three sets, one of which contains the "call" symbols, and another one the "return" symbols, which are matched together by their recognizing automaton, forming an evident nested structure. The fact that the structure of VPLs is immediately visible is also common to other subsets of VPLs that enjoy the same closure properties, such as Parenthesis Languages [McN67], of which VPLs can be considered a generalization.

This peculiarity of VPLs of having an evident structure is in some sense common to another linguistic subfamily of CFLs: Operator Precedence Languages (OPLs). They were introduced by R. W. Floyd [Flo63], also as part of the pursuit of efficient CFL parsing. Although their tree-like structure is not immediately visible as in VPLs, in OPLs it can be easily derived by the precedence relations that hold among terminal symbols. They were, in fact, specifically conceived to represent the typical precedence relations among arithmetic operations in expressions. As a result, they can be used to express the syntax of relevant markup or programming languages, such as JSON, Lua [BCRM+15] and Prolog [DB96]. Also, OPLs strictly include the class of VPLs, being therefore more expressive. Moreover, their structure, while being implicit in precedence relations, allows OPLs to retain all the aforementioned closure properties, namely closure under Boolean operations, but also concatenation and Kleene star, as well as the decidability of several problems [CRMM78, CRM12]. This makes them an ideal candidate for the application of model checking to a wider set of properties. The work presented in this thesis represents an attempt at pursuing this objective. We present a novel temporal logic formalism, called Operator Precedence Temporal Logic (OPTL), conceived specifically to express OPL properties. We start the investigation of its expressive power by comparing it to the logic NWTL, based on VPLs. Moreover, we propose an automata theoretic model checking procedure, that retains the same complexity class of LTL (and NWTL), i.e. it is exponential in the length of the formula.

The thesis is organized as follows. In Chapter 2 we present the language class of VPLs, as well as the temporal logic NWTL, which represents the state of the art in the field. In Chapter 3 we present the class of OPLs, characterizing them by means of generative grammars and a particular class of pushdown automata. Chapter 4 contains the main work of the thesis, since the logic OPTL is presented in this chapter, it is compared to NWTL and a model checking procedure is given for both finite and infinite words. Finally, Chapter 5 contains concluding remarks, as well as directions for further research.

Chapter 2

Visibly Pushdown Languages

Visibly Pushdown Languages (VPL) are a subset of Deterministic Context-Free Languages. They were originally introduced in [Meh80] to support efficient deterministic real-time parsers for a subset of DCFLs. They have the peculiarity of enjoying many of the closure properties of Regular Languages, that are not valid for Context-Free Languages: this is one of the reasons why they were recently revalued in [AM04]. They are, in fact, closed under Boolean operations, concatenation and Kleene star. For this reason they have been employed for model checking NWTL, a formalism that will be presented in Section 2.4. NWTL is the main interesting point of VPLs with respect to the work presented in this thesis. It was proposed in [AAB+08] as a First-Order complete temporal logic formalism based on VPLs. We will prove that the temporal logic we propose, which is presented in Chapter 4, is strictly more expressive than NWTL which, in turn, represents the state of the art in this topic.

The main feature of VPLs is that they are based on a terminal alphabet Σ partitioned into the three sets Σ_c, Σ_r and Σ_i. Σ_c contains *call* symbols, Σ_r *return* symbols, and Σ_i *internal* symbols. Call and return symbols have a role similar to, respectively, open and closed parentheses. In this respect, VPLs can be seen as a generalization of parentheses languages, from which they differ because they also allow unmatched opening and closing symbols. While call and return symbols behave in a typically context-free fashion, internal symbols may only form strings with properties typical of regular languages. This partitioning of the terminal symbols makes the syntactic structure of VPL strings immediately visible, highly facilitating parsing of such strings. These peculiarities of VPLs are better clarified in Section 2.1, where their role in the automata-theoretic characterization of VPLs is shown. Indeed, this particular partition of the alphabet explains the effort put into developing model checking techniques for VPLs: they may be used to model the stack traces of software programs. Call symbols could be associated to procedure calls, and return symbols to procedure returns. In actual stack traces, each function call is matched with the return that terminates the same instance of the function. This is a typically context-free property, which cannot be captured by formalisms based on regular properties, such as LTL, but can be expressed by VPLs.

We present VPLs by means of their automata theoretic characterization in Section 2.1, and we introduce their infinite word counterpart in Section 2.2. In Section 2.3 we present the characterization of VPLs in Monadic Second Order Logic, and in Section 2.4 we describe NWTL, a FO-complete temporal logic formalism on VPLs.

2.1 Visibly Pushdown Automata

We will first characterize VPLs by means of the class of automata recognizing them.

Definition 1 (Visibly Pushdown Automaton (VPA))**.** Let the finite terminal alphabet Σ be partitioned into the three disjoint sets $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_i$. A nondeterministic VPA over $\langle \Sigma_c, \Sigma_r, \Sigma_i \rangle$ is a tuple $\mathcal{A} = \langle Q, I, \Gamma, \delta, F \rangle$ where

- Q is a finite set of states;
- $I \subseteq Q$ is a set of initial states;
- Γ is the set of stack symbol, containing the initial stack symbol $\perp$;
- $\delta = \delta_c \cup \delta_r \cup \delta_i$ is the transition relation, partitioned into the three relations
  - $\delta_c \subseteq Q \times \Sigma_c \times Q \times (\Gamma \setminus \{\perp\})$,
  - $\delta_r \subseteq Q \times \Sigma_r \times \Gamma \times Q$,
  - $\delta_i \subseteq Q \times \Sigma_i \times Q$.

A VPA is *deterministic* iff I is a singleton, and the three transition relations are functions:

$$\delta_c : Q \times \Sigma_c \to Q \times (\Gamma \setminus \{\perp\}), \quad \delta_r : Q \times \Sigma_r \times \Gamma \to Q, \quad \delta_i : Q \times \Sigma_i \to Q.$$

The tripartite transition relation determines three kinds of moves. A transition (p, a, q, γ) with $p, q \in Q$, $a \in \Sigma_c$ and $\gamma \in \Gamma \setminus \{\perp\}$ is called a *push* move, because it pushes symbol γ onto the stack while reading the terminal symbol a, and it causes the automaton to pass from state p to q. A transition (p, a, γ, q) with $p, q \in Q$, $a \in \Sigma_r$ and $\gamma \in \Gamma \setminus \{\perp\}$ is a *pop* move, and it pops γ form the stack, while reading a and bringing the automaton from state p to q. Finally, (p, a, q) with $p, q \in Q$ and $a \in \Sigma_i$ is an internal move, and it just causes a state change when reading a without touching the stack. Internal moves are defined in the same way as the transitions of a finite state automaton. Also, notice that the next move of the automaton is determined only by the type of the terminal symbol being read: call symbols cause push moves, return symbols cause pop moves, internal symbols cause internal moves. Moreover, there are no ε-transitions, and each move consumes exactly one input character: VPA are *real-time*.

More formally, a *configuration* of a VPA is a triple $c = \langle w, q, \Pi \rangle$, where $w \in \Sigma^*$ is the input string, $q \in Q$ is the current state and $\Pi \in \mathcal{S}$ is the content of the stack, with $\mathcal{S} = (\Gamma \setminus \{\perp\})^* \{\perp\}$. Given a string $x = a_0 a_1 \dots a_n \in \Sigma^*$, a *run* of the automaton is a sequence of transitions $c_0 \mapsto c_1 \mapsto \cdots \mapsto c_n$ such that for every $0 \leq i \leq n$ we have

- *push* move: if $a_i \in \Sigma_c$, then there exists a $\gamma \in \Gamma$ such that $\langle a_i a_{i+1} \dots a_n, q_i, \Pi_i \rangle \mapsto \langle a_{i+1} \dots a_n, q_{i+1}, \gamma \Pi_i \rangle$, with $(q_i, a_i, q_{i+1}, \gamma) \in \delta_c$;
- *pop* move: if $a_i \in \Sigma_r$, then there exists a $\gamma \in \Gamma$ such that $\langle a_i a_{i+1} \dots a_n, q_i, \Pi_i \rangle \mapsto \langle a_{i+1} \dots a_n, q_{i+1}, \Pi_{i+1} \rangle$ and either $\gamma \neq \perp$ and $\Pi_{i+1} = \gamma \Pi_i$, or $\gamma = \perp$ and $\Pi_i = \Pi_{i+1} = \perp$, with $(q_i, a_i, \gamma, q_{i+1}) \in \delta_r$;
- *internal* move: if $a_i \in \Sigma_i$, then $\langle a_i a_{i+1} \dots a_n, q_i, \Pi_i \rangle \mapsto \langle a_{i+1} \dots a_n, q_{i+1}, \Pi_i \rangle$ with $(q_i, a_i, q_{i+1}) \in \delta_i$.

A VPA accepts the language

$$L(\mathcal{A}) = \{x \in \Sigma^* \mid \langle x, q_I, \bot\rangle \mapsto^* \langle \epsilon, q_F, \Pi\rangle, \text{with } q_I \in I,\, q_F \in F \text{ and } \Pi \in \mathcal{S}\}.$$

Thus, we can introduce the following:

Definition 2 (Visibly Pushdown Language (VPL))**.** A language $L \subseteq \Sigma^*$ is a VPL if there exists a VPA $\mathcal{A}$ such that $L = L(\mathcal{A})$.

VPLs are a subset of Deterministic Context-Free Languages. Indeed, any nondeterministic VPA can be determinized:

Statement 1 (Theorem 2 of [AM04])**.** *Given a nondeterministic VPA $\mathcal{A}$ with s states, an equivalent deterministic VPA $\tilde{\mathcal{A}}$ can be built with $O(2^{s^2})$ states and $O(2^{s^2} \cdot |\Sigma_c|)$ stack symbols.*

The construction of the deterministic automaton is based on a modified subset construction, that must keep into account the nondeterminism of the stack contents as well. This is achieved by postponing push transitions until the matching return is reached, allowing the automaton to chose the correct push transition. The states of the obtained VPA $\tilde{\mathcal{A}}$ are made of a set S of pairs of states of $\mathcal{A}$, and a set of states R. R keeps track of all possible states $\mathcal{A}$ may nondeterministically be. S contains all possible pairs whose first component is any state of $\mathcal{A}$, and the second is a state reached from the first while reading the subword beginning after the last unmatched push. Push transitions store both S and R on stack, so that pop transitions may use them to determine the correct push transition that would have resulted in a successful run of $\mathcal{A}$.

Most importantly, VPLs with the same alphabet partition are closed under all Boolean operations, enabling model checking by means of VPAs.

Statement 2 (Theorem 1 and Corollary 1 of [AM04])**.** *VPLs on a given alphabet partition are closed under union, complementation and intersection.*

An automaton accepting the union of two VPLs can be built by simply joining the state sets, stack alphabets, initial and final sets of the VPAs accepting them. Closure by complementation is a direct consequence of Statement 1, because deterministic VPAs can be complemented by complementing their set of final states. It is possible to achieve intersection by means of the product construction of both the sets of states and the stack alphabet, because VPA are real-time, and they are synchronized when reading the same word under the same alphabet partition.

Moreover, VPLs have other closure properties that are typical of regular languages:

Statement 3 (Theorem 1 of [AM04])**.** *VPLs on a given alphabet partition are closed by concatenation and Kleene star.*

2.2 Visibly Pushdown ω-Automata

VPLs have been extended to infinite words by defining a class of Büchi automata recognizing them.

Definition 3 (Büchi Visibly Pushdown ω-Automaton (ωVPBA))**.** A nondeterministic ωVPBA over $\langle \Sigma_c, \Sigma_r, \Sigma_i\rangle$ is a tuple $\mathcal{A} = \langle Q, I, \Gamma, \delta, F\rangle$, whose components are the same as those in Definition 1.

The semantics of the automaton is defined in the same way as for finite-word VPAs, except for the fact that computations are infinite. Given a run of the automaton ρ, $\mathrm{Inf}(\rho)$ is the set of states that occur infinitely often in ρ. A run ρ of an ωVPBA is accepting iff at least one final state occurs in it infinitely often, i.e. $\mathrm{Inf}(\rho) \cap F \neq \varnothing$.

A class of VPA for infinite words can also be defined with Muller acceptance conditions:

Definition 4 (Muller Visibly Pushdown ω-Automaton (ωVPMA))**.** A nondeterministic ωVPMA over $\langle \Sigma_c, \Sigma_r, \Sigma_i \rangle$ is a tuple $\mathcal{A} = \langle Q, I, \Gamma, \delta, \mathcal{T} \rangle$, whose components are the same as those in Definition 1, except for $\mathcal{T} \subseteq \mathscr{P}(Q)$, which is called the *table* of the automaton.

The semantics of ωVPMA is also defined in the same way as for VPAs, except for the acceptance condition. A computation ρ of an ωVPMA is accepting iff $\mathrm{Inf}(\rho) \in \mathcal{T}$. For any ωVPBA or ωVPMA $\mathcal{A}$, a string $x \in \Sigma^\omega$ is accepted by $\mathcal{A}$ iff one of the computations of $\mathcal{A}$ on it is accepting. The language accepted by the automaton is

$$L(\mathcal{A}) = \{w \in \Sigma^\omega \mid \mathcal{A} \text{ accepts } w\}.$$

It is possible to prove that ωVPBAs and ωVPMAs accept the same class of ω-languages by extending the classical constructions for their ω-regular counterparts. Differently from what happens for VPAs and ω-regular Muller automata, both ωVPBA and ωVPMA cannot be determinized. ωVPLs retain, however, all the closure properties of VPLs, including being a Boolean algebra:

Statement 4 (Theorems 6 and 8 of [AM04])**.** *ωVPLs on a given alphabet partition are closed under union, complementation, intersection, concatenation and Kleene star.*

2.3 MSO-Logical Characterization of VPLs

The Monadic Second Order Logic characterization of VPLs retraces the classical one for RLs given by [Bü60, Elg61, Tra61]. Given an alphabet Σ and its partition $\langle \Sigma_c, \Sigma_r, \Sigma_i \rangle$, a word w is seen as a structure on the set $U = \{1, \ldots, |w|\}$, with a set of monadic predicates $a(\cdot) \subseteq U$ such that, for any $a \in \Sigma$ and $i \in U$, $a(i)$ holds iff the character in position i in w is an a. In order to reason about word positions, a successor relation $\mathrm{succ} \in U \times U$ is given, so that for any $i, j \in U$ we have $\mathrm{succ}(i, j)$ iff $j = i + 1$. The element that distinguishes the MSO for VPL from the MSO for RL is the binary matching relation $\mu \subseteq (U \cup \{-\infty\}) \times (U \cup \{+\infty\})$: for any $i, j \in U$, we have $\mu(i, j)$ iff the symbol in i is a call, and the one in j is its matching return, i.e. the return that induces the automaton to pop the stack symbol pushed when reading i. An exception to this definition is constituted by pending calls and returns: a position $i \in U$ is a pending call iff $\mu(i, +\infty)$, and it is a pending return iff $\mu(-\infty, i)$. Note that, except for pending calls and returns, the μ relation is one-to-one, and there is no need to quantify it, because it is immediately determined by the word structure. Moreover, a countable infinite set $\mathcal{V}_1$ of first-order variables is given, typeset as lowercase, boldface letters at the end of the alphabet, such as $\mathbf{x}, \mathbf{y}, \ldots$, and interpreted over the single positions in U. Also, the logic uses a set $\mathcal{V}_2$ of second-order (monadic) variables denoted with uppercase, boldface letters at the end of the alphabet, such as $\mathbf{X}, \mathbf{Y}, \ldots$, that represent sets of word positions (subsets of U). The syntax of the MSO logic is the following:

$$\varphi := a(\mathbf{x}) \mid \mathbf{x} \in \mathbf{X} \mid \mathrm{succ}(\mathbf{x}, \mathbf{y}) \mid \mu(\mathbf{x}, \mathbf{y}) \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists \mathbf{x}.\varphi \mid \exists \mathbf{X}.\varphi$$

where $a \in \Sigma$, $\mathbf{x}, \mathbf{y} \in \mathcal{V}_1$ and $\mathbf{X} \in \mathcal{V}_2$. Given a word $w \in \Sigma^+$, its semantics is interpreted with respect to the valuations $v_1 \colon \mathcal{V}_1 \to U$ and $v_2 \colon \mathcal{V}_2 \to \mathscr{P}(U)$:

- $(w, v_1, v_2) \models a(\mathbf{x})$ iff $w = w_1 a w_2$ and $v_1(\mathbf{x}) = |w_1| + 1$;
- $(w, v_1, v_2) \models \mathbf{x} \in \mathbf{X}$ iff $v_1(\mathbf{x}) \in v_2(\mathbf{X})$;
- $(w, v_1, v_2) \models \mathrm{succ}(\mathbf{x}, \mathbf{y})$ iff $v_1(\mathbf{y}) = v_1(\mathbf{x}) + 1$;
- $(w, v_1, v_2) \models \mu(\mathbf{x}, \mathbf{y})$ iff $\mu(v_1(\mathbf{x}), v_1(\mathbf{y}))$, where μ is the matching relation described earlier, which is given;
- $(w, v_1, v_2) \models \neg\varphi$ iff $(w, v_1, v_2) \not\models \varphi$;
- $(w, v_1, v_2) \models \varphi \vee \psi$ iff $(w, v_1, v_2) \models \varphi$ or $(w, v_1, v_2) \models \psi$;
- $(w, v_1, v_2) \models \exists \mathbf{x}.\varphi$ iff there exists a valuation v_1' such that $v_1'(\mathbf{y}) = v_1(\mathbf{y})$ for any $\mathbf{y} \in \mathcal{V}_1 \setminus \{\mathbf{x}\}$ and $(w, v_1', v_2) \models \varphi$;
- $(w, v_1, v_2) \models \exists \mathbf{X}.\varphi$ iff there exists a valuation v_2' such that $v_2'(\mathbf{Y}) = v_2(\mathbf{Y})$ for any $\mathbf{Y} \in \mathcal{V}_2 \setminus \{\mathbf{X}\}$ and $(w, v_1, v_2') \models \varphi$.

To facilitate the comparison of word positions in formulas, the operator $\leq$, whose meaning is the transitive closure of the succ relation, can be used.

A *sentence* is a well-formed formula with no free variables. Given a sentence φ, the language it denoted is defined as

$$L(\varphi) = \{w \in \Sigma^+ \mid w, v_1, v_2 \models \varphi\},$$

where v_1 and v_2 are the valuations described earlier.

For example, given an alphabet $\Sigma = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}\}$ partitioned as $\langle\{\mathrm{a}, \mathrm{b}\}, \{\mathrm{c}\}, \{\mathrm{d}, \mathrm{e}\}\rangle$, formula

$$\forall \mathbf{x} \colon \big(\mathrm{a}(\mathbf{x}) \implies \exists \mathbf{y}.(\mu(\mathbf{x}, \mathbf{y}) \wedge \mathrm{e}(\mathbf{y}))\big)$$

where $\forall \mathbf{x} \colon \varphi$ is a shortcut for $\neg\exists \mathbf{x}.\neg\varphi$, denotes the language of words where all as are matched with a return symbol e. So, the word abece satisfies this formula because a is matched with the last e, while the string abec does not satisfy it, because a is a pending call (in fact, e is the matching return of call symbol b).

The MSO logic described above characterizes VPLs:

Statement 5 (Theorem 4 of [AM04]). *A language L over a partitioned alphabet Σ is a VPL iff there exists a sentence φ in the MSO logic described in this section such that $L = L(\varphi)$.*

The proof of this theorem has been achieved by giving an effective way of translating a VPA into a MSO sentence, and vice versa.

2.4 Nested Words Temporal Logic

Nested Words Temporal Logic (NWTL) is a temporal logic formalism similar to LTL. However, the algebraic structure on which it expresses properties is not purely linear, but contains a matching relation between positions named *calls* and positions named *returns*. In order to present NWTL, we first need to introduce nested words, and the matching relation that characterizes them.

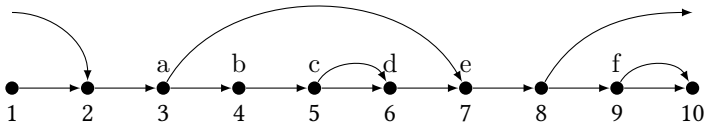


Figure 2.1: Representation of a nested word.

Definition 5 (Matching Relation)**.** Given a set $U \subseteq \mathbb{N}$, a *matching* on U is a triple $\langle \mu, \texttt{call}, \texttt{ret} \rangle$ where $\mu \subseteq U \times U$ is a binary matching relation, and $\texttt{call}, \texttt{ret} \subseteq U$ are two unary relations such that

- if $\mu(i, j)$ then $\texttt{call}(i)$, $\texttt{ret}(j)$ and $i < j$;
- if $\mu(i, j)$ and $\mu(i, j')$ both hold, then $j = j'$, and if $\mu(j, j)$ and $\mu(i', j)$ both hold, then $i = i'$, i.e. the matching relation is one-to-one;
- if $i \leq j$ and $\texttt{call}(i)$ and $\texttt{ret}(j)$, then there exists a position $i \leq k \leq j$ such that either $\mu(i, k)$ or $\mu(k, j)$, i.e. each `call` position is always matched with the closest unmatched `ret`.

Definition 6 (Nested Word)**.** Given a finite alphabet Σ, a nested word is a tuple $w = \langle U, (P_a)_{a \in \Sigma}, <, \mu, \texttt{call}, \texttt{ret} \rangle$, where

- $U = \{1, \ldots, n\}$ if w has a finite length n, and $U = \mathbb{N}$ if w is infinite;
- P_a, is the set of positions labeled with $a \in \Sigma$;
- $<$ is the usual ordering on $\mathbb{N}$;
- $\langle \mu, \texttt{call}, \texttt{ret} \rangle$ form a matching on U.

Word positions $i \in U$ such that $\texttt{call}(i)$ are named *call* positions, those such that $\texttt{ret}(i)$ are *return* positions, and those that are neither a call nor a return are *internal* positions. If $\mu(i, j)$, we say that i is the matching call of j, and j is the matching return of i. If a call has no matching return, it is a *pending* call, and if a return has no matching call, it is a *pending* return.

An example of nested word is shown in Figure 2.1. The set `call` is $\{3, 5, 8, 9\}$, the set `ret` is $\{2, 6, 7, 10\}$ and all other positions are internal. The matching relation is $\mu = \{(3, 7), (5, 6), (9, 10)\}$. Position 2 is a pending return, while position 8 is a pending call.

The subdivision of positions in nested words strictly recalls the partitioned alphabet of VPLs. Indeed, it can be proved [AM09] that regular languages of nested words are equivalent to VPLs. Nested words have also been characterized by means of a class of automata, which can be used for model checking NWTL [AAB+08]. This is, however, out of the scope of this thesis.

NWTL contains all temporal operators from LTL, plus a few operators that express properties regarding the matching relation of nested words. It has the following syntax:

$$\varphi := \top \mid \mathrm{a} \mid \texttt{call} \mid \texttt{ret} \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \bigcirc_\mu \varphi \mid \ominus\varphi \mid \ominus_\mu \varphi \mid \varphi \, \mathcal{U}^\sigma \, \varphi \mid \varphi \, \mathcal{S}^\sigma \, \varphi$$

with $\mathrm{a} \in \Sigma$. The semantics of propositional and LTL operators is the usual one. `call` and `ret` hold in positions that are part of the respective sets. The $\bigcirc_\mu$ (resp. $\ominus_\mu$) operator imposes the truth of its argument in a position that is the matching return (resp. call) of the current position. For example, in the word of Figure 2.1, $\bigcirc_\mu \mathrm{e}$ holds in position 3, and $\ominus_\mu \mathrm{a}$ holds in 7. A call (resp. return) position must have a matching return (resp. call) for $\bigcirc_\mu \varphi$ (resp. $\ominus_\mu \varphi$) to hold, for any φ. For example, $\bigcirc_\mu \top$ does not hold in 8 because it is a pending call. The operators $\mathcal{U}^\sigma$ and $\mathcal{S}^\sigma$ are called *summary* until and since, and they have the usual until and since semantics, except that they consider *summary* paths instead of linear paths. Summary paths can either include consecutive positions, or positions that are in the matching relation, skipping those in between. For example, the summary path between 2 and 8 is made of positions 2, 3, 7 and 8, skipping all positions between 3 and 7, that are a matched pair. The path between 2 and 6 instead all positions between them, because there are no matched call/return pairs between them except 5 and 6, which are consecutive (note that 3 is a call, but its matched return if after 6, so the positions between them cannot be skipped). Formula $\neg \mathrm{b}\, \mathcal{U}^\sigma\, \mathrm{f}$ holds in position 2 because of the summary path made of positions 2, 3, 7, 8, 9, which skips all positions between 3 and 7, and in particular position 4, which contains a b.

More formally, given a nested word w and a position $i \in U$ we write $(w, i) \models \varphi$ to state that NWTL formula φ holds in position i of word w.

- $(w, i) \models \top$ is always true;
- $(w, i) \models \mathrm{a}$, with $\mathrm{a} \in \Sigma$, iff $i \in P_\mathrm{a}$;
- $(w, i) \models$ `call` iff `call`(i) holds;
- $(w, i) \models$ `ret` iff `ret`(i) holds;
- $(w, i) \models \neg\varphi$ iff φ does not hold in i;
- $(w, i) \models \varphi \vee \psi$ iff one or both of φ and ψ hold in i;
- $(w, i) \models \bigcirc\varphi$ iff $(w, i+1) \models \varphi$;
- $(w, i) \models \bigcirc_\mu \varphi$ iff there exists a position $j \in U$ such that $\mu(i, j)$ and $(w, j) \models \varphi$;
- $(w, i) \models \ominus\varphi$ iff $(w, i-1) \models \varphi$;
- $(w, i) \models \ominus_\mu \varphi$ iff there exists a position $j \in U$ such that $\mu(j, i)$ and $(w, j) \models \varphi$;
- $(w, i) \models \varphi\, \mathcal{U}^\sigma\, \psi$ iff there exist a position $j \geq i$ such that $(w, j) \models \psi$ and a summary path $i = i_0 < i_1 < \cdots < i_n = j$ between them such that for any $0 \leq k < n$ we have $(w, i_k) \models \varphi$;
- $(w, i) \models \varphi\, \mathcal{S}^\sigma\, \psi$ iff there exist a position $j \leq i$ such that $(w, j) \models \psi$ ans a summary path $j = i_0 < i_1 < \cdots < i_n = i$ between them such that for any $0 < k \leq n$ we have $(w, i_k) \models \varphi$.

The notion of *summary* path is formalized as follows:

Definition 7 (Summary Path)**.** A summary path between two positions $i, j \in U$, with $i < j$, is a sequence of positions $i = i_0 < i_1 < \cdots < i_n = j$ such that for any $p < n$

$$i_{p+1} = \begin{cases} h & \text{if } \mu(i_p, h) \text{ and } h \leq j, \\ i_p + 1 & \text{otherwise.} \end{cases}$$

Other variants of summary paths have been proposed: *summary-down* paths only follow call edges (from a call to an internal position), internal edges (between two internal positions) and nesting edges (between a call and the matching return). Symmetrically, *summary-up* paths only follow return edges (from an internal position to a return), internal and nesting edges. Until and since operators can be defined based on these paths, without increasing the expressive power of NWTL.

Concerning expressiveness, the following definitive FO-completeness result has been achieved:

Statement 6 (Theorem 4.1 of [AAB+08]). *NWTL = FO over finite and infinite nested words.*

Moreover, a model checking procedure based on automata theory has been given for NWTL, leading to the following:

Statement 7 (Theorem 5.1 of [AAB+08]). *Given a formula φ of NWTL, a nondeterministic Büchi nested word automaton accepting the models of φ can effectively be built with $2^{O(|\varphi|)}$ states.*

A Büchi nested word automaton accepts regular languages of infinite nested words. It is shown in [AM09] that these automata are equivalent to ωVPBA. Therefore, NWTL enables model checking of VPLs.

Bibliographic Notes

The family of VPLs was first introduced by K. Mehlhorn in [Meh80], with the intent of producing a family of automata that could decide their next move only depending on the next character to be read. They were later reintroduced in 2004 by R. Alur and P. Madhusudan, who proved the most important closure properties of VPLs and extended them to infinite words in [AM04]. Such properties made VPLs ideal for model checking the temporal logic CaRet they introduced in [AEM04]. More properties of VPLs were studied in the following years: games with visibly pushdown winning conditions were investigated in [LMS05], in [AKMV05] a syntactic congruence characterizing VPLs was defined, and the problem of minimizing VPAs was studied. In [AM06] (conference version) and [AM09] (journal version), the algebraic structure called *nested word* was presented, and regular languages of nested words were proved to be equivalent to VPLs in linear words. An efficient algorithm for deciding word membership for VPL grammars was given in [LTNP07]. Decisive results regarding temporal logic formalisms based on nested words were reported in [AAB+08]: NWTL was introduced together with an extension of CaRet including the *within* operator, and they were proved to be FO-complete. The closure of VPLs with respect to the quotient operator was proved in [OS17], and to other language operations in [OS18].

Chapter 3

Operator Precedence Languages

The family of OPLs (Operator Precedence Languages) was originally introduced by R. W. Floyd in 1963 [Flo63] with the purpose of supporting the development of efficient deterministic parsers for programming languages. The main idea characterizing them is to assign different precedence relations between terminal characters, similarly to what happens with arithmetic expressions, in which certain operations must be executed before or after the others (e.g. multiplication always precedes addition, etc.). Floyd proposed a shift-reduce bottom-up parsing algorithm that exploits these precedence relations between terminals to identify unambiguously each right-hand side, substituting it with the corresponding left-hand side without the necessity of backtracking due to the choice of a wrong rhs. The algorithm uses a stack to keep track of the previously read terminals and of already reduced rhs. It decides the next move to be performed by comparing the topmost terminal character on the stack with a look-ahead of one character. The way the algorithm uses the stack allows to combine independently parsed substrings to achieve the same result of parsing the whole string, naturally enabling parallel parsing [BVCR+13, BCRM+15].

This is accomplished at the cost of constraining grammar rules to a form that will be detailed later, and that restricts OPLs to be a subset of DCFLs (Deterministic Context-Free Languages). This determined the loss of interest in this language family due to the invention of the more general LR(k) parsers, addressing the whole DCFL family, by D. Knuth [Knu65]. Despite such limitations, the OPL family is wide enough to express the syntactic constructs found in the most common programming languages. Floyd showed himself in [Flo63] that the syntax of the Algol 60 programming language could be adapted to this language family by modifying its context-free grammar. More recently, an OPL-based parser was produced for Prolog [DB96], while the data description language JSON and the programming language Lua were shown to be OPLs in [BCRM+15] by appropriately transforming their grammars.

OPLs will be presented presented from different perspectives in this chapter. First, in Section 3.1 they will be defined by means of the class of grammars that generate them, introduced by Floyd in his seminal paper [Flo63]. The automata-theoretic and MSO characterizations of the OPL family came only decades later, and will be presented in Sections 3.2 and 3.4, respectively. Moreover, in Section 3.3, we present the generalization of OPLs to infinite words. In Sections 3.1 and 3.3, the most important

$$
\begin{aligned}
S &\to G\,\mathbf{throw} \mid S\,\mathbf{call}\,H\,\mathbf{ret} \mid \varepsilon \\
G &\to S\,\mathbf{call}\,C \\
C &\to F\,\mathbf{call}\,C \mid F \\
F &\to F\,\mathbf{call}\,H\,\mathbf{ret} \mid \varepsilon \\
H &\to H\,\mathbf{handle}\,T \mid F \\
T &\to T\,\mathbf{throw} \mid T\,\mathbf{call}\,H\,\mathbf{ret} \mid K\,\mathbf{throw} \mid \varepsilon \\
K &\to T\,\mathbf{call}\,C
\end{aligned}
$$

Figure 3.1: Production rules of grammar $G_{\mathbf{throw}}$, with axiom S. Strings generated by this grammar represent a stack trace, with function calls (**call**) and returns (**ret**). Functions may throw exceptions (**throw**), that unwind the stack until a handler (**handle**) is encountered. The axiom, S, generates a sequence of **call**s either interrupted by a **throw**, or closed by a matching **ret**. In the first case, a sequence of interrupted **call**s is generated by G and C. Otherwise, F generates only matching **call**/**ret** pairs. Inside the first level of such pairs, H generates sequences of other matching **call**/**ret** pairs (through F) or handlers (H), that are be followed by T, which generates either matching **call**/**ret** pairs, or **throw** statements that are caught by this handler, possibly interrupting other **call**s (K). Note that the only difference between S and T is that the latter can generate sequences of **throw**s that do not interrupt any **call**.

closure properties proved for OPLs will be presented. These properties, some of which do not hold for larger language families such as DCFLs and CFLs, are particularly important for the subject of this thesis, because they enable the definition of a sound temporal logic based on OPLs and, most importantly, model checking of properties expressed in such a formalism.

3.1 Operator Precedence Grammars

OPGs (Operator Precedence Grammars) are a family of generative grammars defined by constraining the more general context-free grammars. They are a subset of operator grammars.

Definition 8 (Operator Production Rule)**.** A grammar production rule is in Operator Form if its right-hand side contains no adjacent non-terminal symbols.

Definition 9 (Operator Grammar)**.** An Operator Grammar is a context-free generative grammar that contains only production rules in operator form.

Any context-free grammar may be transformed into Operator Grammar form [Har78].

Figure 3.1 shows the production rules of an operator grammar. Operator grammar form allows us to define precedence relations between pairs of terminal characters. These relations depend on the structure of right hand sides of production rules, and contain information about the structure of the resulting abstract syntax tree. Thus, automata and parsing algorithms for OPLs can infer from them which next move is

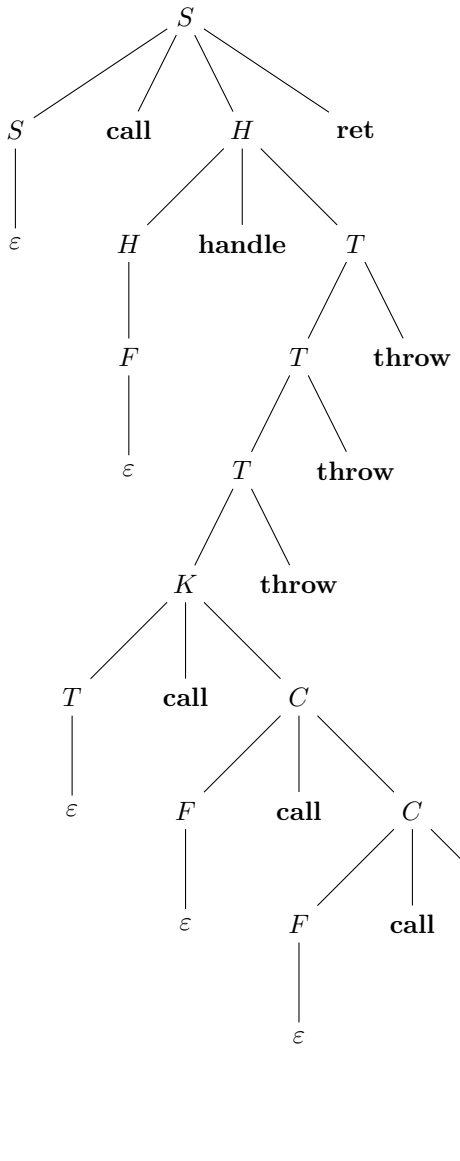


Figure 3.2: Abstract syntax tree of string **call handle call call call throw throw throw ret** according to grammar $G_{\mathbf{throw}}$ of Figure 3.1.

$$
\begin{aligned}
\mathcal{L}(S) &= \{\textbf{call}, \textbf{throw}\} & \mathcal{R}(S) &= \{\textbf{ret}, \textbf{throw}\} \\
\mathcal{L}(G) &= \{\textbf{call}, \textbf{throw}\} & \mathcal{R}(G) &= \{\textbf{call}, \textbf{ret}\} \\
\mathcal{L}(C) &= \{\textbf{call}\} & \mathcal{R}(C) &= \{\textbf{call}, \textbf{ret}\} \\
\mathcal{L}(F) &= \{\textbf{call}\} & \mathcal{R}(F) &= \{\textbf{ret}\} \\
\mathcal{L}(H) &= \{\textbf{call}, \textbf{handle}\} & \mathcal{R}(H) &= \{\textbf{ret}, \textbf{handle}, \textbf{throw}\} \\
\mathcal{L}(T) &= \{\textbf{call}, \textbf{throw}\} & \mathcal{R}(T) &= \{\textbf{ret}, \textbf{throw}\} \\
\mathcal{L}(J) &= \{\textbf{call}, \textbf{throw}\} & \mathcal{R}(J) &= \{\textbf{call}, \textbf{ret}\}
\end{aligned}
$$

Figure 3.3: Left and right terminal sets of grammar $G_{\textbf{throw}}$ of Figure 3.1.

the most appropriate. Let Σ be the set of terminal symbols. We say that two terminals $a, b \in \Sigma$ are *equal in precedence* if they appear in the rhs of a grammar rule consecutively or separated by at most one non-terminal, and we write $a \doteq b$. If two terminals are equal in precedence, they appear at the same level of an abstract syntax tree. If a appears at the immediate left of a subtree such that b is the leftmost leaf of its fringe, or the leftmost terminal leaf after a single non-terminal leaf, we say a *yields precedence* to b, and we write $a \lessdot b$. Finally, if b is at the immediate right of a subtree that has a as either its rightmost leaf or its rightmost terminal leaf after a non-terminal, we say that a *takes precedence* from b, and we write $a \gtrdot b$. Figure 3.2 shows the abstract syntax tree of the grammar of Figure 3.1. It is immediate to note that **call** $\lessdot$ **handle**, because **handle** appears to the left of the subtree generated by B, which follows **call** in the first production. Conversely, **handle** $\gtrdot$ **ret**, because **handle** appears in a subtree to the immediate left of **ret**, with only a non-terminal (I) to its right. Moreover, **call** $\doteq$ **ret**, because they appear in the same rhs, separated only by non-terminal B.

More formally, let V be a set of non-terminal characters.

Definition 10 (Left/Right Terminal Sets)**.** Let G be an operator grammar. The left and right terminal sets of $A \in V$ are

$$
\begin{aligned}
\mathcal{L}_G &= \{a \in \Sigma \mid A \Rightarrow_G^* Ba\alpha\} \\
\mathcal{R}_G &= \{a \in \Sigma \mid A \Rightarrow_G^* \alpha aB\}
\end{aligned}
$$

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

Figure 3.3 shows the left and right terminal sets of the example grammar $G_{\textbf{throw}}$.

Definition 11 (Precedence Relations)**.** Let G be an operator grammar, $\alpha, \beta \in (\Sigma \cup V)^*$ and $a, b \in \Sigma$. The precedence relations $\lessdot$ (*yields precedence*), $\doteq$ (*equal in precedence*) and $\gtrdot$ (*takes precedence*) are defined as follows:

$$
\begin{aligned}
a \lessdot b &\iff \exists A \to \alpha a D \beta \text{ with } D \in V \text{ and } b \in \mathcal{L}_G(D); \\
a \doteq b &\iff \exists A \to \alpha a B b \beta \text{ with } B \in V \cup \{\varepsilon\}; \\
a \gtrdot b &\iff \exists A \to \alpha D b \beta \text{ with } D \in V \text{ and } a \in \mathcal{R}_G(D).
\end{aligned}
$$

Note that the precedence relations enjoy none of the typical properties of order relations, such as reflexivity, transitivity, etc. Given an operator grammar G, it is

	call	**ret**	**handle**	**throw**
call	$\lessdot$	$\doteq$	$\lessdot$	$\gtrdot$
ret	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
handle	$\lessdot$	$\gtrdot$	$\gtrdot$	$\lessdot$
throw	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$

Figure 3.4: OPM $M_{\mathbf{throw}}$ of grammar $G_{\mathbf{throw}}$ of Figure 3.1.

possible to obtain from it the precedence relations that hold for each pair of terminals, defining its Operator Precedence Matrix M^G.

Definition 12 (Operator Precedence Matrix (OPM))**.** Given an Operator Grammar G, its Operator Precedence Matrix M^G is a partial function $M^G : (\Sigma \cup \{\#\})^2 \to \mathscr{P}(\{\lessdot, \doteq, \gtrdot\})$ such that, for each pair $(a, b) \in \Sigma \times \Sigma$, M^G_{ab} is the set of precedence relations that hold between a and b according to G. For any $a \in \Sigma$, $M^G_{a\#} = \{\lessdot\}$ and $M^G_{\#a} = \{\gtrdot\}$, and $M^G_{\#\#} = \{\doteq\}$.

We say M^G is *conflict-free* iff for any $a, b \in \Sigma$ we have $|M^G_{ab}| \leq 1$; and that M^G is *complete* iff the function is total. Two conflict-free OPMs M and M' are *compatible* iff their union is conflict-free, i.e. iff for any $a, b \in \Sigma$, either $M_{ab} = M'_{ab}$, or $M_{ab} = \varnothing$ or $M'_{ab} = \varnothing$.

The special symbol $\#$ is used as a delimiter for strings of OPLs, and its significance will be clearer in the following sections. An *operator precedence alphabet* is a pair (Σ, M) where Σ is a terminal alphabet and M is a conflict-free OPM defined on it. Figure 3.4 shows the OPM of grammar $G_{\mathbf{throw}}$: $(\{\mathbf{call}, \mathbf{ret}, \mathbf{handle}, \mathbf{throw}\}, M_{\mathbf{throw}})$ is an operator precedence alphabet.

The precedence relations strictly depend on the shape of the abstract syntax tree associated to a string by an operator grammar. An OPM summarizes the rules that generate such a structure. Therefore, the presence of a conflict in the OPM results in ambiguity in the interpretation of certain strings with respect to the structure given to them by the grammar. Conversely, a conflict-free OPM can assign a unique structure to each string. This fact is at the base of OPL parsing techniques, and of many of their interesting closure properties.

Definition 13 (Operator Precedence Grammar (OPG))**.** An Operator Grammar G is an Operator Precedence or Floyd Grammar iff the associated OPM M^G is conflict-free.

Definition 14 (Operator Precedence Language (OPL))**.** An Operator Precedence Language is a language generated by an OPG.

Matrix $M_{\mathbf{throw}}$ of Figure 3.4 is conflict-free: this means grammar $G_{\mathbf{throw}}$ of Figure 3.1 is and OPG, and the language it generates is an OPL.

The most important normal form for OPGs is Fischer Normal Form:

Definition 15 (Fischer Normal Form (FNF) [Fis69])**.** An OPG is in FNF iff its production rules are invertible (i.e. no rhs is associated to more than one non-terminal), and it has no renaming rules (i.e. of the form $V \to V'$) and no empty rules (i.e. $V \to \varepsilon$) except those having the axiom as their left hand side.

For any OPG, an equivalent grammar in FNF can be built algorithmically [Har78]. Grammar $G_{\mathbf{throw}}$ of Figure 3.1 can be transformed in FNF by eliminating copy rules and ε-productions, and introducing new non-terminals to factor out common rhs. The fact that grammar rules in FNF are invertible is crucial for the possibility of efficient parsing of OPLs. Although OPL parsing is not the main topic of this thesis, its predisposition to parallelism is remarkable, and it may result to be interesting also in the perspective of parallel model checking. Hence, we report Algorithm 1, which is a parsing algorithm for OPL. It builds bottom-up the syntax tree of the input string by recognizing right-hand sides of production rules as soon as they have been read completely, with the help of a look-ahead of one character when performing reductions. Its moves are completely deterministic, because they are driven by precedence relations, and because the rules of the input grammar are in FNF, and hence when a rhs has been recognized, the non-terminal at its lhs is determined uniquely.

Input: $(\alpha, head, end, \mathcal{S})$.
$\alpha \in \Sigma \cup V$ is the input string, containing both terminal and non-trminal symbols; $head$ and end are integers marking the beginning and the end of the substring of α to be parsed; $\mathcal{S}$ is the initial content of the stack. The stack contains pairs $(Z, p) \in (\Sigma \cup V \cup \{\#\}) \times \{\lessdot, \doteq, \gtrdot, \perp\}$, where $p = \perp$ if $Z \in V$, and it encodes the precedence relation between Z and the previous symbol otherwise. For sequential parsing, initially $\alpha = \beta\#$ for some β, $head = 1$, $end = |\alpha|$, $\mathcal{S} = (\#, \perp)$.
do
  $X := \alpha(head)$;
  $Y :=$ the topmost terminal in $\mathcal{S}$;
  if $Y \lessdot X$ **then** push $(X, \lessdot)$; $head := head + 1$;
  else if $Y \doteq X$ **then** push $(X, \doteq)$; $head := head + 1$;
  else if $X \in V$ **then** push $(X, \perp)$; $head := head + 1$;
  else if $Y \gtrdot X$ **then**
    if $\mathcal{S}$ *does not contain any* $\lessdot$ **then** push $(X, \gtrdot)$; $head := head + 1$;
    else
      Let $\mathcal{S} = (X_n, p_n) \dots (X_i, \lessdot)(X_{i-1}, p_{i-1}) \dots (X_0, p_0)$ where $p_j \neq \lessdot$ with $i < j \leq n$;
      if $X_{i-1} \in V$ *(so* $p_{i-1} = \perp$*) and* $\exists A \to X_{i-1}X_i \dots X_n$ **then**
        in $\mathcal{S}$ replace $(X_n, p_n) \dots (X_i, \lessdot)(X_{i-1}, p_{i-1})$ with $(A, \perp)$
      else if $X_{i-1} \in \Sigma \cup \{\#\}$ *and* $\exists A \to X_i \dots X_n$ **then**
        in $\mathcal{S}$ replace $(X_n, p_n) \dots (X_i, \lessdot)$ with $(A, \perp)$
      else parsing error;
    end
  end
while $head < end$ *or (*$head = end$ *and* $\mathcal{S} \neq (B, \perp)(a, \perp)$ *with* $B \in V$ *and* $a \in \Sigma \cup \{\#\}$*)*;
Output: $\mathcal{S}$

Algorithm 1: Operator Precedence Parsing.

The reason why the stack contents are taken as part of the input, and the input string may contain non-terminals, is the use of this algorithm for parallel parsing. Indeed, it is possible to split the input string, and then execute this algorithm on all parts in parallel. Since not all reductions can be performed if only part of the string is parsed, all instances terminate with a non-empty stack, whose contents can be split

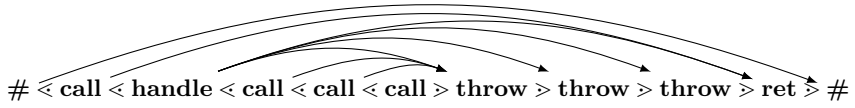


Figure 3.5: The string whose syntax tree is depicted in Figure 3.2, with precedence relations shown and chains highlighted with arrows joining their contexts.

in a part $\mathcal{S}^L$ that does not contain any $\lessdot$ symbol, and a part $\mathcal{S}^R$ that does not contain any $\gtrdot$. In order to join the partially parsed substrings, Algorithm 1 can be launched with $\mathcal{S}^R$ as the initial stack and the sequence of terminal and non-terminal symbols in $\mathcal{S}^L$ as the input string. This process can be repeated until all chunks of the string have been parsed, and the resulting stack content cannot be split.

As we previously remarked, given an OP alphabet, its OPM gives a sort of "hidden" structure to strings on that alphabet. This intuition of structure is formalized by the notion of *chain*.

Definition 16 (Chain)**.** Let (Σ, M) be an OP alphabet.

A *simple chain* is a string $a_0 a_1 \dots a_n a_{n+1}$ such that $a_0, a_{n+1} \in \Sigma \cup \{\#\}$, for any $1 \leq i \leq n$ we have $a_i \in \Sigma$, $M_{a_0 a_{n+1}} \neq \varnothing$, and the precedence relations $a_0 \lessdot a_1 \doteq \dots \doteq a_n \gtrdot a_{n+1}$ hold. We write ${}^{a_0}[a_1 \dots a_n]^{a_{n+1}}$ to mean that such a string is a simple chain.

A *composed chain* is a string $a_0 x_0 a_1 x_1 \dots a_n x_n a_{n+1}$ such that ${}^{a_0}[a_1 \dots a_n]^{a_{n+1}}$ is a simple chain, and for any $0 \leq i \leq n$ we have $x_i \in \Sigma^*$ and either $x_i = \varepsilon$ or ${}^{a_i}[x_i]^{a_{i+1}}$ is a simple or composed chain. We write ${}^{a_0}[x_0 a_1 x_1 \dots a_n x_n]^{a_{n+1}}$, meaning that such a string is a composed chain.

Given a chain ${}^{a}[x]^{b}$, $a, b \in \Sigma \cup \{\#\}$ are called the *context* of the chain, while $x \in \Sigma^+$ is called its *body*.

Figure 3.5 shows an example word, with the chains it contains according to OPM $M_{\mathbf{throw}}$. Notice how bodies of chains are enclosed into the $\lessdot$ and $\gtrdot$ relations. Multiple chains can have the same character as the left or right parts of their contexts. There is a strict relationship between the chains in a string and the production rules of the grammar that generate it, especially if it is in FNF. For example, the body of the chain ${}^{\mathbf{call}}[\mathbf{handle} \dots \mathbf{throw}]^{\mathbf{ret}}$ clearly corresponds to the rhs $\mathbf{handle}\ I$ of grammar $G_{\mathbf{throw}}$, and therefore to non-terminal H. Moreover, I corresponds to the body of the chain ${}^{\mathbf{handle}}[\mathbf{call} \dots \mathbf{throw}]^{\mathbf{ret}}$, generated by $S\ \mathbf{throw}$, and so on. The $\doteq$ relation is strictly connected with the length of right hand sides of production rules. Indeed, if we have $a_0 \doteq a_1 \doteq \dots \doteq a_{n-1} \doteq a_n$ then, according to Definition 11, a grammar generating the same string must contain a rule with rhs $A_0 a_0 A_1 \dots a_{n-1} A_{n-1} a_n A_n$, where A_is, $0 \leq i \leq n$, are possibly missing non-terminals. For this reason, if an OPM contains a "cycle" of terminals in the $\doteq$ relation, no OPG compatible with it can be defined, unless we allow regular expressions in rhs [CP17]. On the other hand, $\doteq$-cyclic OPMs pose no problem with the automata presented in Section 3.2.

Given an OP alphabet (Σ, M) and a string $x \in \Sigma^*$, due to the fact that M is conflict-free there is at most one way to represent $\#x\#$ as a composed chain. More formally:

Definition 17 (Compatible string)**.** A string $x \in \Sigma^*$ over (Σ, M) is *compatible* with M iff:

- for each pair of consecutive terminals a, b in x, we have $M_{ab} \neq \varnothing$;
- for each substring $x' = a_0 x_0 a_1 x_1 \dots a_n x_n a_{n+1}$ of $\#x\#$, if ${}^{a_0}[a_1 \dots a_n]^{a_{n+1}}$ is a simple chain and for any $0 \leq i \leq n$ either $x_i = \varepsilon$ or ${}^{a_i}[x_i]^{a_{i+1}}$ is a chain, then $M_{a_0 a_{n+1}} \neq \varnothing$.

If M is complete, then every string in Σ^* is compatible with it.

OPLs enjoy the most significant closure properties that apply to regular languages, but not to general context-free languages. In particular, the fact that they form a Boolean algebra enables model checking based on this language set, which is the main result of this thesis.

Statement 8 (Corollary 5.7, Theorems 5.8 and 5.9 of [CRMM78]). *The class of OPLs is closed under union, complementation and intersection.*

The above results were proved by defining an algebraic lattice of OPGs and the respective OPLs. From their proofs follows the fact that, given an OP alphabet (Σ, M), the universal language with respect to Boolean closure properties is Σ^* only if M is complete.

Moreover, OPLs enjoy other important closure properties:

Statement 9 (Statement 12 of [CRM12]). *The class of OPLs is closed with respect to string reversal.*

Statement 10 (Statement 13 of [CRM12]). *The class of OPLs is closed with respect to prefix and suffix operations.*

Statement 11 (Theorem 15 of [CRM12]). *OPLs with compatible OPMs are closed under concatenation.*

Statement 12 (Theorem 16 of [CRM12]). *OPLs with OPMs that contain no $\doteq$-cycles are closed under Kleene star.*

Finally, another important property of OPLs is that they strictly include the class of VPLs, and consequently also Parenthesis Languages.

Statement 13 (Theorem 5 of [CRM12]). *For any visibly pushdown automaton $\mathcal{A}$, an OPG G such that $L(G) = L(\mathcal{A})$ can be effectively constructed.*

The resulting grammar is built by pairing each call symbol with a return symbol that can be matched with it according to the transitions of the VPA. Each admissible pair is used as the right hand side of a grammar rule, optionally inserting a non-terminal symbol between them, generating other substrings with balanced call/return pairs. Rules containing only a call or a return symbol, with possibly other non-terminal symbols, are included if the VPA accepts strings with pending calls or returns. The so obtained OPG is characterized by the OPM of Figure 3.6a, which assigns a precedence relation to each pair of terminal symbols depending on which partition of the VPL alphabet they belong to. Call and return symbols are always equal in precedence. Between them, other matched calls and returns can appear: indeed, calls yield precedence to other calls. Conversely, internal symbols take precedence from any other symbol: this way, they form a linear descending branch in the syntax tree, similar to those produced by regular grammars. Finally, returns also take precedence from any other symbol, because they close the right hand sides of the grammar rules. In Section 3.2, we present a class of automata that recognize OPLs. The reader

	Σ_c	Σ_r	Σ_i
Σ_c	$\lessdot$	$\doteq$	$\lessdot$
Σ_r	$\gtrdot$	$\gtrdot$	$\gtrdot$
Σ_i	$\gtrdot$	$\gtrdot$	$\gtrdot$

(a)

	b	**c**	**d**	**e**	**f**
b	$\lessdot$	$\doteq$			$\gtrdot$
c		$\gtrdot$			
d			$\gtrdot$		
e				$\lessdot$	$\doteq$
f	$\doteq$		$\doteq$		$\lessdot$

(b)

Figure 3.6: Figure (a) is the OPM of an OPG obtained by translating a VPA, while (b) is the OPM of language L_{123}.

may compare the moves induced by this OPM in one of such automata with those of VPA, finding out that they are strictly related. For example, the fact that internal symbols take precedence form other symbol makes the automaton pop the corresponding symbol from the stack right after it has been pushed, actually determining a behavior similar to that of FSMs.

It is also possible to prove that this containment relation between VPLs and OPLs is strict.

Statement 14 (Theorem 6 of [CRM12])**.** *The VPL family is strictly included in the OPL family.*

This can easily be proved by noticing that the language $L_{123} = L_1 \cup L_2 \cup L_3$, which is the union of the three languages

$$L_1 = \{\mathbf{b}^n\mathbf{c}^n \mid n \geq 1\}, \quad L_2 = \{\mathbf{f}^n\mathbf{d}^n \mid n \geq 1\}, \quad L_3 = \{\mathbf{e}^n(\mathbf{fb})^n \mid n \geq 1\},$$

is OPL but not VPL. Indeed, strings that are part of L_1 impose that $\mathbf{b}$ is a call and $\mathbf{c}$ is a return; while strings in L_2 impose that $\mathbf{f}$ is a call and $\mathbf{d}$ a return. Conversely, L_3 requires $\mathbf{b}$ or $\mathbf{f}$ to be a return. This is not consistent with the alphabet partition of VPLs, whose sets must be disjoint. Instead, L_{123} can be proved to be an OPL, with the OPM in Figure 3.6b.

3.2 Operator Precedence Automata

Operator Precedence Automata (OPA) are an automata-theoretic formal model that completely characterizes OPLs. They are left-to-right automata based on a pushdown store, but with a predefined stack alphabet.

Definition 18 (Operator Precedence Automaton (OPA))**.** An OPA is a tuple $\mathcal{A} = \langle \Sigma, M, Q, I, F, \delta \rangle$, where:

- (Σ, M) is an operator-precedence alphabet (i.e. Σ is a set of terminals and M is an OPM on Σ);
- Q is a set of states (disjoint from Σ);
- $I \subseteq Q$ is the set of initial states;
- $F \subseteq Q$ is the set of final states;

- $\delta \subseteq Q \times (\Sigma \cup Q) \times Q$ is the transition relation, and it is the union of three disjoint relations

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

An OPA is deterministic iff I is a singleton and the three components of δ are (partial) functions:

$$\delta_{shift} : Q \times \Sigma \to Q, \qquad \delta_{push} \subseteq Q \times \Sigma \to Q, \qquad \delta_{pop} \subseteq Q \times Q \to Q.$$

Despite its importance for the semantics of an OPA, the stack alphabet is not included in the above definition, because it is derived from Σ and Q. Let $\Gamma = \Sigma \times Q$: the stack alphabet is $\Gamma' = \Gamma \cup \{\bot\}$, where $\bot$ is the initial symbol. We denote elements of Q as $p, p_i, q, q_i, \dots$ and elements of Γ as $[a, q]$, with $a \in \Sigma$ and $q \in Q$. We also define the following notations:

- $symbol([a, q]) = a$ and $symbol(\bot) = \#$;
- $state([a, q]) = q$;
- given a stack containing $\Pi = (\pi_n \dots \pi_1 \bot)$ with $\pi_i \in \Gamma$ for $1 \leq i \leq n$ and $n \in \mathbb{N}$,

$$symbol(\Pi) = \begin{cases} symbol(\pi_n) & \text{if } n \geq 1, \\ \# & \text{if } n = 0. \end{cases}$$

A *configuration* of an OPA is a tuple $\langle w, q, \Pi \rangle$ with $w \in \Sigma^* \#$, $q \in Q$ and $\Pi \in \Gamma^* \bot$. A *run* or *computation* of the automaton is a sequence of moves, or transitions between different configurations $c_1 \mapsto c_2$, of the three following kinds:

- *push* move: if $symbol(\Pi) \lessdot a$ then $\langle ax, p, \Pi \rangle \mapsto \langle x, q, [a, p]\Pi \rangle$, with $(p, a, q) \in \delta_{push}$;
- *shift* move: if $a \doteq b$ then $\langle bx, q, [a, p]\Pi \rangle \mapsto \langle x, r, [b, p]\Pi \rangle$, with $(q, b, r) \in \delta_{shift}$;
- *pop* move: if $a \gtrdot b$ then $\langle bx, q, [a, p]\Pi \rangle \mapsto \langle bx, r, \Pi \rangle$, with $(q, p, r) \in \delta_{pop}$.

Shift and pop moves can never occur if the stack only contains the initial symbol $\bot$. Only push and shift rules consume input characters; push moves insert a new symbol on top of the stack, while shift moves only update the topmost symbol. Pop moves remove the topmost stack symbol, updating the current state accordingly; they use the input symbol as a sort of a look-ahead, leaving it to be read by the next push or shift move. Notice the close analogy between grammar rules and stack moves that is determined by precedence relations: the automaton performs a push move if it encounters a character that yields precedence to another one, which means a new rhs starts there, and a new symbol is pushed on stack to keep track of it. A shift move occurs when two consecutive terminals that are equal in precedence are found: since they must be part of the same rhs, the topmost stack symbol is updated. A pop move occurs when the terminal contained in the topmost stack symbol takes precedence from the current input character: this means the computation reached the end of the subtree containing the rhs related to the topmost stack symbol. A pop move is therefore analogous to a non-terminal reduction.

The OPA accepts the language

$$L(\mathcal{A}) = \{x \in \Sigma^* \mid \langle x\#, q_I, \bot \rangle \mapsto^* \langle \#, q_F, \bot \rangle, \text{with } q_I \in I \text{ and } q_F \in F\}.$$

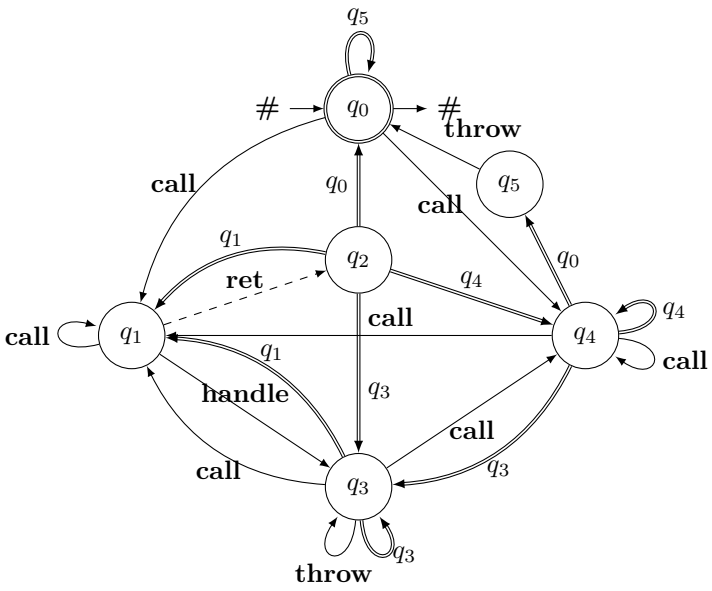


Figure 3.7: OPA based on OPM $M_{\mathbf{throw}}$ of Figure 3.4. The initial state is q_0. When the automaton reads a **call** from this state, it pushes it on stack, and starts two nondeterministic computations: one of them goes in state q_1, and the other one in state q_4. The computation in state q_1 is accepting (i.e. ends in q_0 with an empty stack) only if the next outermost **call** has a matching **ret**, rejecting if it ends with an unhandled **throw**. Inside the outermost **call**/**ret** pair can appear other such pairs (q_1), or **handle** statements (q_3), which can be followed by other **handle**s, **throw**s (q_3), matched **call**/**ret**s (q_1), or **call**s interrupted by a **throw** (q_4). The computation which goes on with state q_4 only accepts **call**s that are interrupted by a **throw**, possibly interleaved with matched **call**/**ret**s (q_1).

step	input	state	stack
1	**call handle call call call throw throw throw ret**#	q_0	$\bot$
2	**handle call call call throw throw throw ret**#	q_1	$[\mathbf{call}, q_0]\bot$
3	**call call call throw throw throw ret**#	q_3	$[\mathbf{handle}, q_1][\mathbf{call}, q_0]\bot$
4	**call call throw throw throw ret**#	q_4	$[\mathbf{call}, q_3][\mathbf{handle}, q_1][\mathbf{call}, q_0]\bot$
5	**call throw throw throw ret**#	q_4	$[\mathbf{call}, q_4][\mathbf{call}, q_3][\mathbf{handle}, q_1][\mathbf{call}, q_0]\bot$
6	**throw throw throw ret**#	q_4	$[\mathbf{call}, q_4][\mathbf{call}, q_4][\mathbf{call}, q_3][\mathbf{handle}, q_1][\mathbf{call}, q_0]\bot$
7	**throw throw throw ret**#	q_4	$[\mathbf{call}, q_4][\mathbf{call}, q_3][\mathbf{handle}, q_1][\mathbf{call}, q_0]\bot$
8	**throw throw throw ret**#	q_4	$[\mathbf{call}, q_3][\mathbf{handle}, q_1][\mathbf{call}, q_0]\bot$
9	**throw throw throw ret**#	q_3	$[\mathbf{handle}, q_1][\mathbf{call}, q_0]\bot$
10	**throw throw ret**#	q_3	$[\mathbf{throw}, q_3][\mathbf{handle}, q_1][\mathbf{call}, q_0]\bot$
11	**throw throw ret**#	q_3	$[\mathbf{handle}, q_1][\mathbf{call}, q_0]\bot$
12	**throw ret**#	q_3	$[\mathbf{throw}, q_3][\mathbf{handle}, q_1][\mathbf{call}, q_0]\bot$
13	**throw ret**#	q_3	$[\mathbf{handle}, q_1][\mathbf{call}, q_0]\bot$
14	**ret**#	q_3	$[\mathbf{throw}, q_3][\mathbf{handle}, q_1][\mathbf{call}, q_0]\bot$
15	**ret**#	q_3	$[\mathbf{handle}, q_1][\mathbf{call}, q_0]\bot$
16	**ret**#	q_1	$[\mathbf{call}, q_0]\bot$
17	#	q_2	$[\mathbf{ret}, q_0]\bot$
18	#	q_0	$\bot$

Figure 3.8: Accepting computation of the OPA of Figure 3.7. (Only the accepting one of the several non-deterministic computations is reported.)

Figure 3.7 shows an example of an OPA based on OPM $M_{\mathbf{throw}}$ of Figure 3.4, while Figure 3.8 shows the computation accepting string **call handle call call call throw throw throw ret**. In Figure 3.7 and the following ones, initial states are represented with an entering arrow, while final states have an exiting arrow and are circled. Push transitions are represented with a solid arc, shift moves with a dashed arc, and pop moves with a double arrow. The arcs of push and shift moves are labeled with the terminal symbol they read, while those of pop moves are labeled with the state they pop from stack.

The concept of chain, presented in Definition 16, has a relevant relation with the moves an OPA preforms while reading it. In general, every time the automaton reads the first terminal of the body of a chain, it does so with a push move; the next terminals of the same body are processed with shift moves, and a pop transition occurs at the end of the body. This behavior is formalized by the notion of *support* of a chain.

Definition 19 (Support of a chain)**.** Le $\mathcal{A}$ be an OPA. A *support* for the simple chain ${}^{a_0}[a_1 \dots a_n]^{a_{n+1}}$ is any computation path of $\mathcal{A}$ of the form

$$q_0 \xrightarrow{a_1} q_1 \overset{a_2}{\dashrightarrow} \dots \overset{a_{n-1}}{\dashrightarrow} q_{n-1} \overset{a_n}{\dashrightarrow} q_n \overset{q_0}{\Longrightarrow} q_{n+1}.$$

A support for the composed chain ${}^{a_0}[x_0 a_1 x_1 \dots a_n x_n]^{a_{n+1}}$ is any computation path of $\mathcal{A}$ of the form

$$q_0 \overset{x_0}{\rightsquigarrow} q'_0 \xrightarrow{a_1} q_1 \overset{x_1}{\rightsquigarrow} q'_1 \overset{a_2}{\dashrightarrow} \dots \overset{a_n}{\dashrightarrow} q_n \overset{x_n}{\rightsquigarrow} q'_n \overset{q'_0}{\Longrightarrow} q_{n+1},$$

where, for any $0 \leq i \leq n$, if $x_i = \varepsilon$ then $q_i = q'_i$ and, otherwise, $q_i \overset{x_i}{\rightsquigarrow} q'_i$ is a support for the simple or composed chain ${}^{a_i}[x_i]^{a_{i+1}}$.

We denote as $q_0 \overset{x}{\rightsquigarrow} q_{n+1}$ the support for a chain whose body is x.

Indeed, given an automaton on an OP alphabet (Σ, M), if it performs a computation $\langle xb, q_i, [a, q_j]\Pi\rangle \mapsto^* \langle b, q_k, \Pi\rangle$, then $q_i \overset{x}{\rightsquigarrow} q_k$, and ${}^a[x]^b$ is a chain on that alphabet.

Given an OPA $\mathcal{A}$, the shape of its transition relation δ and the sets I and Q determine which of the strings on the terminal alphabet Σ compatible with the OPM M are accepted by the automaton, and thus part of $L(\mathcal{A})$. In particular, the following automaton can be introduced:

Definition 20 (Operator Precedence Max-Automaton)**.** The OP Max-Automaton over a given an OP alphabet (Σ, M) is defined as $\mathcal{A}(\Sigma, M) = \langle \Sigma, M, \{q\}, \{q\}, \{q\}, \delta_{max}\rangle$, where $\delta_{max}(q, q) = q$ and $\delta_{max}(q, c) = q$ for any $c \in \Sigma$.

The OP Max-Automaton over (Σ, M) accepts all strings on Σ compatible with OPM M, i.e. it has a support for every chain in such strings. The language it accepts is called the *max-language* or *universal* language of (Σ, M). The importance of this last definition will be clearer when analyzing the logical aspects of OPLs.

In Definition 18 we introduced both nondeterministic and deterministic OPA: these two classes of automata are actually equivalent. Indeed, with respect to nondeterminism, OPLs behave more similarly to regular languages that to other context-free subclasses.

Statement 15 (Theorem 2.4 of [LMPP15])**.** *Given a nondeterministic OPA $\mathcal{A}$ with s states, an equivalent deterministic OPA $\tilde{\mathcal{A}}$ can be built with $2^{O(s^2)}$ states.*

The determinization of an OPA is similar to the classic one for Finite State Automata, because sets are used as the states of $\tilde{\mathcal{A}}$, in order to keep track of nondeterministic computations. However, the state of an OPA consists not only of the actual current state, but also of the stack contents. Indeed, the main issue to be addressed in the construction of $\tilde{\mathcal{A}}$ is to keep track of which push and pop transitions where performed in each execution path. For this reason, the states of $\tilde{\mathcal{A}}$ are sets of pairs of states of $\mathcal{A}$. Each pair keeps track of a different execution path; the first state in the pair is the actual current state of the simulated path, while the second one keeps track of the state of the automaton before the previous push transition in the simulated path. Whenever a pop move occurs, the second component of each pair is retrieved from the corresponding pair stored in the popped stack symbol. This way, the automaton can exploit the stack in order to keep track of the whole sequence of nondeterministic choices made during each computation.

The stack of the automaton is used in a similar way to keep track of previous computations when deriving an OPA equivalent to a given OPG, yielding the following result:

Statement 16 (Theorem 3.1 of [LMPP15]). *Given an OPG G, an OPA $\mathcal{A}$ such that $L(G) = L(\mathcal{A})$ can be built with $O(m^2)$ states, where m is the sum of the lengths of the right-hand sides of the rules of G.*

In order to sketch the construction of an OPA from a given OPG G, we will assume without loss of generality that G has no empty or copy rules, except possibly those whose left-hand side is the axiom. The automaton is built in such a way that its successful computations correspond to reconstructing bottom-up the mirror of the rightmost derivation of the input string according to G. The automaton performs a push move every time it encounters the beginning of a new rhs of a rule in G, a shift move every time it finds a terminal symbol inside a rhs, and multiple nondetermistic pop moves when it reaches the end of one or more rhs, one for each non-terminal the is the lhs of the ending rhs. Non-determinism is used only to address the fact that G may not be in FNF, and its rules may not be invertible, so when a rhs is recognized, the corresponding lhs must be guessed nondeterministically. The states of the automaton are pairs whose left component is the rhs currently being reconstructed, and the right one is the rhs that was being built before the last push move. Every time a push move occurs, the current state is stored on stack, so its contents can be used to reconstruct the whole chain of pending rhs. We will not go into deeper details regarding this construction, referring the interested reader to [LMPP15]. Figure 3.9 shows a successful computation of an automaton derived from a slightly modified version of grammar $G_{\mathbf{throw}}$, where copy and empty rules have been removed. Notice that this run is isomorphic to the one of Figure 3.8.

It is also possible to build an OPG generating the language accepted by a given OPA:

Statement 17 (Theorem 3.5 of [LMPP15]). *Given an OPA $\mathcal{A}$, and OPG G such that $L(G) = L(\mathcal{A})$ can be built.*

For this construction, we will assume that the OPM of $\mathcal{A}$ does not contain $\doteq$-cycles, so that there is an upper bound to the length of the right-hand sides of the resulting grammar G. Otherwise, we could resort to a generalized version of OPGs that allows regular expressions in rhs [CP17]. Let $\mathcal{A} = \langle \Sigma, M, Q, I, F, \delta \rangle$: the equivalent grammar G has tuples $(a, q, p, b) \in \Sigma \times Q \times Q \times \Sigma$ as non-terminals, written as $\langle {}^a p, q^b \rangle$. Then,

step	input	state	stack
1	$\mathbf{call\ han\ call\ call\ call\ thr\ thr\ thr\ ret}\#$	$\langle \varepsilon, \varepsilon \rangle$	$\bot$
2	$\mathbf{han\ call\ call\ call\ thr\ thr\ thr\ ret}\#$	$\langle \mathbf{call}, \varepsilon \rangle$	$[\mathbf{call}, \langle \varepsilon, \varepsilon \rangle]\bot$
3	$\mathbf{call\ call\ call\ thr\ thr\ thr\ ret}\#$	$\langle \mathbf{han}, \mathbf{call} \rangle$	$[\mathbf{han}, \langle \mathbf{call}, \varepsilon \rangle][\mathbf{call}, \langle \varepsilon, \varepsilon \rangle]\bot$
4	$\mathbf{call\ call\ thr\ thr\ thr\ ret}\#$	$\langle \mathbf{call}, \mathbf{han} \rangle$	$[\mathbf{call}, \langle \mathbf{han}, \mathbf{call} \rangle][\mathbf{han}, \langle \mathbf{call}, \varepsilon \rangle][\mathbf{call}, \langle \varepsilon, \varepsilon \rangle]\bot$
5	$\mathbf{call\ thr\ thr\ thr\ ret}\#$	$\langle \mathbf{call}, \mathbf{call} \rangle$	$[\mathbf{call}, \langle \mathbf{call}, \mathbf{han} \rangle][\mathbf{call}, \langle \mathbf{han}, \mathbf{call} \rangle][\mathbf{han}, \langle \mathbf{call}, \varepsilon \rangle][\mathbf{call}, \langle \varepsilon, \varepsilon \rangle]\bot$
6	$\mathbf{thr\ thr\ thr\ ret}\#$	$\langle \mathbf{call}, \mathbf{call} \rangle$	$[\mathbf{call}, \langle \mathbf{call}, \mathbf{call} \rangle][\mathbf{call}, \langle \mathbf{call}, \mathbf{han} \rangle][\mathbf{call}, \langle \mathbf{han}, \mathbf{call} \rangle][\mathbf{han}, \langle \mathbf{call}, \varepsilon \rangle][\mathbf{call}, \langle \varepsilon, \varepsilon \rangle]\bot$
7	$\mathbf{thr\ thr\ thr\ ret}\#$	$\langle C, \mathbf{call} \rangle$	$[\mathbf{call}, \langle \mathbf{call}, \mathbf{han} \rangle][\mathbf{call}, \langle \mathbf{han}, \mathbf{call} \rangle][\mathbf{han}, \langle \mathbf{call}, \varepsilon \rangle][\mathbf{call}, \langle \varepsilon, \varepsilon \rangle]\bot$
8	$\mathbf{thr\ thr\ thr\ ret}\#$	$\langle C, \mathbf{call} \rangle$	$[\mathbf{call}, \langle \mathbf{han}, \mathbf{call} \rangle][\mathbf{han}, \langle \mathbf{call}, \varepsilon \rangle][\mathbf{call}, \langle \varepsilon, \varepsilon \rangle]\bot$
9	$\mathbf{thr\ thr\ thr\ ret}\#$	$\langle K, \mathbf{han} \rangle$	$[\mathbf{han}, \langle \mathbf{call}, \varepsilon \rangle][\mathbf{call}, \langle \varepsilon, \varepsilon \rangle]\bot$
10	$\mathbf{thr\ thr\ ret}\#$	$\langle K\ \mathbf{thr}, \mathbf{han} \rangle$	$[\mathbf{thr}, \langle K, \mathbf{han} \rangle][\mathbf{han}, \langle \mathbf{call}, \varepsilon \rangle][\mathbf{call}, \langle \varepsilon, \varepsilon \rangle]\bot$
11	$\mathbf{thr\ thr\ ret}\#$	$\langle I, \mathbf{han} \rangle$	$[\mathbf{han}, \langle \mathbf{call}, \varepsilon \rangle][\mathbf{call}, \langle \varepsilon, \varepsilon \rangle]\bot$
12	$\mathbf{thr\ ret}\#$	$\langle I\ \mathbf{thr}, \mathbf{han} \rangle$	$[\mathbf{thr}, \langle I, \mathbf{han} \rangle][\mathbf{han}, \langle \mathbf{call}, \varepsilon \rangle][\mathbf{call}, \langle \varepsilon, \varepsilon \rangle]\bot$
13	$\mathbf{thr\ ret}\#$	$\langle I, \mathbf{han} \rangle$	$[\mathbf{han}, \langle \mathbf{call}, \varepsilon \rangle][\mathbf{call}, \langle \varepsilon, \varepsilon \rangle]\bot$
14	$\mathbf{ret}\#$	$\langle I\ \mathbf{thr}, \mathbf{han} \rangle$	$[\mathbf{thr}, \langle I, \mathbf{han} \rangle][\mathbf{han}, \langle \mathbf{call}, \varepsilon \rangle][\mathbf{call}, \langle \varepsilon, \varepsilon \rangle]\bot$
15	$\mathbf{ret}\#$	$\langle I, \mathbf{han} \rangle$	$[\mathbf{han}, \langle \mathbf{call}, \varepsilon \rangle][\mathbf{call}, \langle \varepsilon, \varepsilon \rangle]\bot$
16	$\mathbf{ret}\#$	$\langle H, \mathbf{call} \rangle$	$[\mathbf{call}, \langle \varepsilon, \varepsilon \rangle]\bot$
17	$\#$	$\langle \mathbf{call}\ H\ \mathbf{ret}, \mathbf{call} \rangle$	$[\mathbf{ret}, \langle \varepsilon, \varepsilon \rangle]\bot$
18	$\#$	$\langle S, \varepsilon \rangle$	$\bot$

Figure 3.9: Accepting computation of the OPA deriving from grammar $G_{\mathbf{throw}}$. (**throw** and **handle** have been abbreviated.)

- for every support $q_0 \overset{x}{\rightsquigarrow} q_{n+1}$ of a simple chain ${}^{a_0}[x]^{a_{n+1}}$, with $x = a_1 a_2 \dots a_n$, add to G the rule
$$\langle {}^{a_0} q_0, q_{n+1}{}^{a_{n+1}} \rangle \to a_1 a_2 \dots a_n$$
- for every support $q_0 \overset{x}{\rightsquigarrow} q_{n+1}$ of a composed chain ${}^{a_0}[x]^{a_{n+1}}$, with $x = x_0 a_1 x_1 a_2 \dots a_n x_n$, add to G the rule
$$\langle {}^{a_0} q_0, q_{n+1}{}^{a_{n+1}} \rangle \to \Lambda_0 a_1 \Lambda_1 a_2 \dots a_n \Lambda_n$$
where $\Lambda_i = \langle {}^{a_i} q_i, q_i'{}^{a_{i+1}} \rangle$ if $x_i \neq \varepsilon$, and $\Lambda_i = \varepsilon$ otherwise, for any $0 \leq i \leq n$.

Introduce non-terminal S as the axiom of G. For every obtained non-terminal $\langle {}^{a_0} q_0, q_{n+1}{}^{a_{n+1}} \rangle$, if $a_0 = a_{n+1} = \#$, q_0 is initial and q_{n+1} is final, then add the rule $S \to \langle {}^{a_0} q_0, q_{n+1}{}^{a_{n+1}} \rangle$. If ε is accepted by $\mathcal{A}$, add $S \to \varepsilon$.

Thanks to Statements 16 and 17, we can claim that OPLs are the language class associated to both OPGs and OPAs.

3.3 Operator Precedence ω-Languages and Automata

Languages containing strings of infinite length are useful to describe many processes, such as executions of never-terminating computer programs, which are nowadays widespread in many environments. Languages of infinite words, of ω-languages, were first introduced as an extension of regular languages by Büchi [Büc62]. In [LMPP15], ωOPLs are introduced as the infinite counterpart of OPLs, and different families of automata capable of recognizing them are defined.

As usual, the set of all infinite strings on a set of terminals Σ is denoted as Σ^ω. In order to introduce ωOPLs, we must first redefine the notion of compatibility of a string with an OPM. Given an OP alphabet (Σ, M), we say that a string $w \in \Sigma^\omega$ is compatible with M if and only if all (finite) prefixes of w are compatible with M. Thus, the ω-language made of all infinite words $w \in \Sigma^\omega$ compatible with M is denoted as $L^\omega_M \subseteq \Sigma^\omega$. We can now characterize ωOPLs by means of two different definitions of ωOPAs.

Definition 21 (Büchi Operator Precedence ω-Automaton (ωOPBA)). A nondeterministic ωOPBA is a tuple $\mathcal{A} = \langle \Sigma, M, Q, I, F, \delta \rangle$, whose components are the same as those defined for OPAs in Definition 18.

Note that, in the infinite case, OPMs are defined as functions with domain $(\Sigma \cup \{\#\}) \times \Sigma$, because the symbol $\#$ can only appear at the beginning of an infinite string. The transition semantics of ωOPBAs are defined in the same way as for OPAs, as well as the notions of configuration and computation, the latter being infinite in this case. In the following, with "$\exists^\omega i$" we mean "there exist infinitely many values of i". Let $w \in \Sigma^\omega$ be an infinite string and ρ be a computation of a ωOPBA on it. We define $\mathrm{Inf}(\rho) = \{q \in Q \mid \exists^\omega i. \langle w_i, q, \Pi_i \rangle \in \rho\}$ as the set of states appearing infinitely often in configurations in ρ. A computation ρ of an ωOPBA $\mathcal{A}$ is *successful* if there exists a final state $q_f \in F$ such that $q_f \in \mathrm{Inf}(\rho)$. String $x \in \Sigma^\omega$ is *accepted* by $\mathcal{A}$ iff one of its computations on x is successful: the ω-language recognized by $\mathcal{A}$ is the set

$$L(\mathcal{A}) = \{w \in \Sigma^\omega \mid \mathcal{A} \text{ accepts } w\}.$$

Similarly, it is possible to define an automaton for ωOPLs with Muller accepting conditions.

Definition 22 (Muller Operator Precedence ω-Automaton (ωOPMA)**.** A nondeterministic ωOPMA is a tuple $\mathcal{A} = \langle \Sigma, M, Q, I, \mathcal{T}, \delta \rangle$, where Σ, M, Q, I and δ are the same as those defined for OPAs in Definition 18, while $\mathcal{T} \subseteq \mathscr{P}(Q)$ is called the *table* of the automaton.

A computation ρ of an ωOPMA is *successful* iff $\mathrm{Inf}(\rho) \in \mathcal{T}$, i.e. the set of states recurring infinitely often in ρ is part of the table $\mathcal{T}$. The notions of string acceptance and accepted language are defined accordingly, as for ωOPBAs.

ωOPBAs and ωOPMAs have the same expressive power, i.e. they can recognize the same set of language, which coincides with ωOPLs. The deterministic versions of these automaton classes are strictly less expressive than their nondeterministic counterparts. This is true even with the Muller acceptance conditions, that were introduced for regular ω-languages to obtain deterministic automata with the same expressive power of their nondeterministic versions. ω-automata with other accepting conditions can be defined, but the sets of languages they can recognized are more restricted. For this reason, we omit their definitions and refer the reader to [LMPP15] for their complete presentation.

Moreover, we now extend the notion of generalized Büchi accepting conditions [CVWY92] to ωOPBA, since they are needed in Section 4.6.2.

Definition 23 (Generalized Büchi Operator Precedence ω-Automaton (ωOPGBA)**.** A nondeterministic ωOPGBA is a tuple $\mathcal{A} = \langle \Sigma, M, Q, I, \mathcal{F}, \delta \rangle$, where Σ, M, Q, I and δ are the same as those defined for OPAs in Definition 18, while $\mathcal{F} \in \mathscr{P}(Q)$ is a collection of sets of Büchi-final states.

A computation ρ of an ωOPGBA $\mathcal{A}$ is accepting iff for any $F_i \in \mathcal{F}$ there exists a state $q_{F_i} \in F_i$ such that $q_{F_i} \in \mathrm{Inf}(\rho)$, i.e. if at least one state for each one of the final sets in $\mathcal{F}$ appears infinitely often in ρ. The notions of string acceptance and accepted language are defined accordingly.

It is possible to transform an ωOPGBA into an ωOPBA with the classic "counting construction" [BK08]. Otherwise, an equivalent ωOPMA $\mathcal{A}_M$ can be built from any ωOPGBA $\mathcal{A}_G$ by including in the table $\mathcal{T}$ of $\mathcal{A}_M$ all and only those subsets of Q that satisfy the generalized Büchi acceptance condition of $\mathcal{A}_G$, i.e. those containing at least one state from each one of the sets in $\mathcal{F}$. Conversely, an ωOPBA is trivially an ωOPGBA with a single acceptance set. Thus, ωOPGBAs can accept the same class of languages as ωOPBAs and ωOPMAs.

Many of the properties stated in Section 3.1 can be extended to the infinite counterpart of OPLs, the most important one being the fact they form a Boolean algebra:

Statement 18 (Theorems 7.5, 7.6 and 7.7 of [LMPP15])**.** *The class of ωOPLs is closed under union, complementation and intersection.*

Moreover,

Statement 19 (Theorem 7.8 of [LMPP15])**.** *The language obtained by concatenating a finite OPL and an ωOPL remains an ωOPL.*

Finally, the inclusion of VPLs in OPLs holds for the infinite counterparts as well:

Statement 20 (Theorem 6.1 of [LMPP15])**.** *The inclusion relation stating $L(\omega BVPA) \subseteq L(\omega OPBA)$ holds, i.e. $\omega VPL \subseteq \omega OPL$.*

Properties on ωOPLs have been proved by resorting to their automata-theoretic characterizations, differently from those of finite OPLs, which rely on OPG constructions.

3.4 MSO-Logical Characterization of OPLs

A Monadic Second Order Logic has been formulated for OPLs as well. Like the MSO logic for VPLs, it is based on the traditional one for RLs. We will therefore only detail the differences with the one presented in Section 2.3. Indeed, the latter is characterized by the matching relation μ. The MSO for OPLs substitutes it with the *chain* relation M_χ. The main difference between the two is that μ is one-to-one, while M_χ is not. Moreover, μ is given by the VPL alphabet partition, which suffices to define the structure of a word because of the real-time nature of VPA. In OPLs, the word structure is given by an OPM, depending on the characters that surround a given substring, and not by characters of the substring itself, as for VPLs. With respect to the abstract syntax tree of a string, while the μ relation holds between the leftmost and rightmost leaves of a subtree, the M_χ relation holds between the two terminals immediately to the left and to the right of such a subtree, either at the same level ($\doteq$), or at a higher level ($\lessdot$ and $\gtrdot$). Indeed, the M_χ relation holds between the two characters determining the *context* of the subtree whose frontier is delimited by them. As suggested by its name, the M_χ relation is based on the concept of chain presented in Definition 16. More formally, given an OP alphabet (Σ, M), $M_\chi(\mathbf{x}, \mathbf{y})$ holds in a string $\#w\#$ iff $\#w\# = w_1 a w_2 b w_3$, $|w_1| = \mathbf{x}$, $|w_1 a w_2| = \mathbf{y}$, and ${}^a[w_2]^b$ is a simple or composed chain. The word w is surrounded by two $\#$ characters in positions 0 and $|w|+1$ that, as usual, yield precedence to every other character, which in turn takes precedence from $\#$. This way, the whole word w is the body of the chain with the two $\#$ as its context, and $M_\chi(0, |w|+1)$. Again, note that the two positions in the M_χ relation are not part of the body of the chain, but they are its context. For this reason, one of them may be part of the context of multiple chains, which leads to the fact that M_χ is not one-to-one. Notice the resemblance of this relation with the behavior of OPAs and of the OPL parsing algorithm: the structure is inferred by a lookback and lookahead character, which are the context of the chain.

For example, with reference to the word of Figure 3.5, the formula

$$\forall \mathbf{x}\colon \big(\mathbf{throw}(\mathbf{x}) \implies \exists \mathbf{y}.(\mathbf{handle}(\mathbf{y}) \wedge M_\chi(\mathbf{y}, \mathbf{x}))\big)$$

is true for strings where each **throw** is caught by a **handle** statement. A string containing a **throw** statement that forms a chain with the left $\#$ delimiter would not satisfy it.

This MSO logic completely characterizes OPLs:

Statement 21 (Theorem 4.2 of [LMPP15]). *A language L over an OP alphabet (Σ, M) is an OPL iff there exists an MSO sentence φ such that $L = L(\varphi)$.*

Again, the proof of this theorem consists in an effective procedure to translate a given OPA into a MSO formula, and a given MSO formula into an equivalent OPA.

Bibliographic Notes

As we already stated, OPLs were first introduced in [Flo63], in which OPGs were defined and an efficient parsing algorithm presented. In 1969, the Fischer Normal

Form for OPGs was introduced in [Fis69]. Later, in 1978, OPLs were proved to form a Boolean algebra in [CRMM78]. In Section 3.1 we presented the definition of OPGs given in this paper. Then, the interest in OPLs faded away. It was resumed in 2010, when other important closure properties of OPLs (namely closure with respect to reversal, suffix/prefix extraction, concatenation, and Kleene star) and the inclusion of the VPL and BALAN language families were proved (cf. [CRM12]). At the same time, a first attempt at characterizing OPLs by means of a family of pushdown automata was made in [LMP11]. An MSO characterization of OPLs was given in [LMP13], and the ω-version of OPLs was introduced in [PPLM13]. The subjects of the last three mentioned papers are more extensively and exhaustively detailed in [LMPP15]. In particular, OPAs are presented in Section 3.2 as defined in this paper. Moreover, this paper investigates the relationships of various alternative definitions of ω-OPAs with respect to their expressive power, and closure properties of ω-OPLs are proved in it. Concurrently, the benefits introduced by OPLs in parallel parsing were investigated theoretically in [BCRMP13] and practically in [BVCR$^+$13, BCRM$^+$14, BCRM$^+$15]. A weighted version of OPLs was later introduced in [DDMP17], and the more general language family of Higher Order OPLs was introduced in [CP17]. For an extensive and self-contained survey of OPLs and a comparison with other significant subsets of the CFL family, the reader may refer to [MP18] (the presentation of OPLs given in this chapter follows this paper substantially).

Chapter 4

Operator Precedence Temporal Logic

In this chapter we report on Operator Precedence Temporal Logic (OPTL), the main result of this thesis. It is a novel temporal logic formalism based on operator precedence words, an algebraic structure compatible with OPLs. Such a structure consists of a linear sequence of word positions, similarly to other common temporal logics such as LTL, associated to an OPM, that gives a more complex, tree-like structure to such positions. It permits to express a much wider variety of context-free properties, not being limited to those recognized by finite state automata and their infinitary counterparts. The context-free structure of operator precedence words is tackled by means of the chain relation. It is based on the concept of chain (Definition 16), which characterizes the structure given by an OPM to compatible strings. In particular, two word positions are in the chain relation if they are the context of a chain, i.e. if there is a chain body between them. This way, pairs of word positions are put in relation: this somewhat resembles the matching relation of nested words (Definition 6), upon which the logic NWTL is based (cf. Section 2.4). However, the matching relation of nested words has a significant limitation: it is one-to-one, restricting any position to be in relation with only one different position. This is enough for expressing many context-free properties, and in particular those that only put in relation unique pairs of objects: indeed, formalisms such as NWTL have been proposed in order to model the behavior of the stack of a program, where to each procedure call is associated a single return statement. However, there are cases in which it could be useful to consider a single position in relation with many other positions. Sticking to the example of procedures, many instances may be interrupted by an exception or an interrupt, instead of their normal return statement. In this case, it would be useful to be able to put the interrupting event in relation with the call statements of the procedure instances it arrests. This cannot be easily modeled with nested words. Even resorting to pending calls, which do not have a matched return, is limiting, because it does not allow to model the eventual recovery after the interruption. In this context, the main advantage of OPTL over NWTL arises: it naturally allows a single position to be in relation with several others, because the chain relation is not one-to-one. Thus, an exception or interrupt event can be put in relation with multiple call statements, as shown in the examples of Chapter 3, or in those of the following sections.

The peculiarities of OPTL are presented in detail in this chapter. First, the syntax

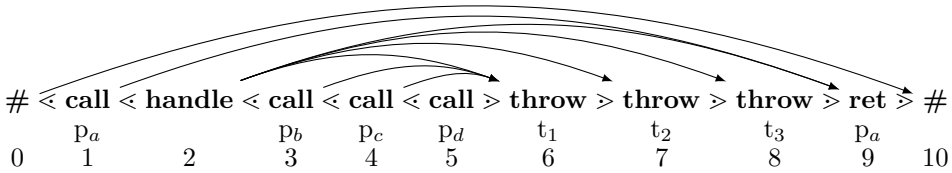


Figure 4.1: An example of execution trace, according to $M_{\mathbf{throw}}$.

of OPTL is presented in Section 4.1, with a few explanatory examples. Then, in Section 4.2, we report its formal semantics. More structured examples and use cases of properties that can be expressed in OPTL are available in Section 4.3. After presenting OPTL as a logic formalism, we proceed by investigating some of its properties. In particular, in Section 4.4 we show that some operators of OPTL are not necessary, because they can be expressed by other operators, and we point out an adequate subset thereof. The relationship between OPTL and NWTL, on which we already informally commented, is thoroughly investigated in Section 4.5. Finally, a fundamental step for the actual utilization of OPTL is made: an automata theoretic model checking procedure is given in Section 4.6 for both finite an infinite words.

4.1 Syntax

The proposed Operator Precedence Temporal Logic (OPTL) is based on the following syntax:

$$\begin{aligned}\varphi := \mathrm{a} \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid \bigcirc\varphi \mid \bigcirc\chi\varphi \mid \ominus\varphi \mid \ominus_\chi\varphi \\ \mid (\varphi\, \mathcal{U}\, \varphi) \mid (\varphi\, \mathcal{U}^{\boxdot}\, \varphi) \mid (\varphi\, \mathcal{S}\, \varphi) \mid (\varphi\, \mathcal{S}^{\boxdot}\, \varphi) \\ \mid (\varphi\, \mathcal{U}^{\Diamond}\, \varphi) \mid (\varphi\, \mathcal{S}^{\Diamond}\, \varphi)\end{aligned}$$

where $\mathrm{a} \in AP$, and AP is the set of atomic propositions. Moreover, $\boxdot$ is one or more of the symbols $\lessdot$, $\doteq$ and $\gtrdot$, and $\Diamond \in \{\uparrow, \downarrow\}$.

Before defining the formal semantics of the operators above, we will provide an intuition of their meaning by means of a few examples based on the OP word of Figure 4.1, interpreted with respect to the OPM of Figure 3.4. As usual, chains are highlighted by arrows joining their context. The position labels typeset in bold are the *structural* labels: the OPM gives a structure to the word based on them, ignoring other labels, which treated as simple atomic propositions. In this case, atomic propositions are used to associate a specific procedure to each statement. First, procedure p_a is called (pos. 1), and it installs an exception handler in pos. 2. Then, three nested procedures are called, and the innermost one (p_d) throws a sequence of exceptions, which are all caught by the handler. Finally, p_a returns, uninstalling the handler.

The syntax of OPTL contains all the familiar operators from LTL: the $\bigcirc$ and $\ominus$ symbols denote the next and back operators, while the undecorated $\mathcal{U}$ and $\mathcal{S}$ operators are LTL until and since. The $\bigcirc\chi$ and $\ominus_\chi$ operators, which we call *matching next* and *matching back*, express properties on string positions in the chain relation (which will be formally defined later on) with the current one. For example, formula $\bigcirc\chi\mathbf{throw}$ is true in positions containing a **call** to a procedure that is terminated by an exception thrown by an inner procedure, such as 3 and 4 of Figure 4.1, because pos. 3 forms

a chain with pos. 6, in which **throw** holds, and so on. Formula $\ominus_\chi$**handle** is true in handled **throw** positions, such as 6, 7 and 8, because e.g. pos. 2 forms a chain with 6, and **handle** holds in 2. The $\mathcal{U}^\boxdot$ and $\mathcal{S}^\boxdot$ operators, called *operator precedence summary until and since*, are inspired to the homonymous $\mathcal{U}^\sigma$ and $\mathcal{S}^\sigma$ operators from NWTL, and are path operators that can "jump" over chain bodies; the symbol $\boxdot$ is a placeholder for one or more precedence relations allowed in the path (e.g. $\mathcal{U}^{\lessdot\doteq}$ or $\mathcal{U}^{\gtrdot\lessdot}$ and so on). Formula (**call** $\vee$ **throw**) $\mathcal{U}^{\gtrdot}$ **ret** is true in pos. 3 because there is a path that jumps over the chain between 3 and 6, and goes on with positions 7, 8 and 9, which are in the $\gtrdot$ relation: pos. 3 and from 6 to 8 satisfy **call** $\vee$ **throw**, while pos. 9 satisfies **ret**. In $\mathcal{U}^{\circledcirc}$ and $\mathcal{S}^{\circledcirc}$, $\circledcirc$ is a placeholder for $\uparrow$ or $\downarrow$; these peculiar path operators are called *hierarchical until and since*, and they express properties about the multiple positions in the chain relation with the current one: their associated paths can dive up and down between such positions. For example, **throw** $\mathcal{U}^\uparrow\ \mathrm{t}_3$ and **throw** $\mathcal{S}^\downarrow\ \mathrm{t}_1$ hold in pos. 2, because there is path 6-7-8 made of ending positions of chains starting in 2, such that **throw** holds until t_3 holds (or **throw** has held since t_1 held). Formulas **call** $\mathcal{U}^\downarrow\ \mathrm{p}_c$ and **call** $\mathcal{S}^\uparrow\ \mathrm{p}_b$ hold in pos. 6, because of path 3-4, made of positions where a chain ending in 6 starts, and whose labels satisfy the appropriate until and since conditions.

Many relevant properties can be expressed in OPTL: formula $\square[$**handle** $\implies \bigcirc\chi$**ret**$]$, where $\square\psi$ is a shortcut for $\neg(\top\ \mathcal{U}\ \neg\psi)$, holds if all exception handlers are properly uninstalled by a return statement. Formula $\square[$**throw** $\implies \neg(\top\ \mathcal{S}^\uparrow\ \mathrm{p}_b)]$ is false if procedure p_b is terminated by an exception; and formula $\neg(\top\ \mathcal{U}^\uparrow$ (**throw** $\wedge$ $\top\ \mathcal{U}^\downarrow$ **call**)) is true in **handle**s catching only throw statements not interrupting any procedure. These properties may not be expressible in NWTL, because its nesting relation is one-to-one, and fails to model situations in which a single entity is in relation with multiple other entities.

4.1.1 Shortcuts

In order to facilitate the composition of complex formulas, we provide shortcuts for the most commonly used constructs. The symbols $\top$ and $\bot$ always stand for a tautology and a contradiction, respectively: $\top \equiv \varphi \vee \neg\varphi$ and $\bot \equiv \varphi \wedge \neg\varphi$, for any formula φ. Moreover, for any two formulas φ and ψ, $\varphi \vee \psi \equiv \neg(\neg\varphi \wedge \neg\psi)$.

The following *globally* operators are defined:

Linear globally: $\square\varphi \equiv \neg(\top\ \mathcal{U}\ \neg\varphi)$;

Operator globally: $\square^\boxdot\varphi \equiv \neg(\top\ \mathcal{U}^\boxdot\ \neg\varphi)$ where $\boxdot$ is replaced with the symbols in O, for any set $O \subseteq \{\lessdot, \doteq, \gtrdot\}$;

Hierarchical globally: $\square^{\circledcirc}\varphi \equiv \neg(\top\ \mathcal{U}^{\circledcirc}\ \neg\varphi)$ where $\circledcirc \in \{\uparrow, \downarrow\}$.

Similarly, the following *eventually* operators can be defined:

Linear eventually: $\Diamond\varphi \equiv \top\ \mathcal{U}\ \varphi$;

Operator eventually: $\Diamond^\boxdot\varphi \equiv \top\ \mathcal{U}^\boxdot\ \varphi$ where $\boxdot$ is replaced with the symbols in O, for any set $O \subseteq \{\lessdot, \doteq, \gtrdot\}$;

Hierarchical eventually: $\Diamond^{\circledcirc}\varphi \equiv \top\ \mathcal{U}^{\circledcirc}\ \varphi$ where $\circledcirc \in \{\uparrow, \downarrow\}$.

In order to obtain clearer formulas, parentheses can be removed by applying the following precedence rules:

- unary operators have higher precedence over all binary operators;

- temporal binary operators ($\mathcal{U}$ and $\mathcal{S}$) have a higher precedence than $\wedge$ and $\vee$;
- temporal binary operators ($\mathcal{U}$ and $\mathcal{S}$) are right-associative, with no distinction based on their type;
- $\wedge$ takes precedence over $\vee$.

4.2 Semantics

4.2.1 Algebraic Structure

The semantics of OPTL is based on the word structure $\langle U, M_{\mathscr{P}(AP)}, P\rangle$ where

- $U = \{0, 1, \ldots, n, n+1\}$, with $n \in \mathbb{N}$, is a sequence of word positions;
- $M_{\mathscr{P}(AP)}$ is an operator precedence matrix on $\mathscr{P}(AP)$;
- $P\colon U \to \mathscr{P}(AP)$ is a function that associates each word position in U with the set of atomic propositions that hold in that position, with $P(0) = P(n+1) = \{\#\}$.

Since the word structure is based upon subsets of AP, in the following we will denote them with lowercase letters such as $a \in \mathscr{P}(AP)$, even though they are sets. For any two positions $i, j \in U$, we will write $i \odot j$, with $\odot \in \{\lessdot, \doteq, \gtrdot\}$, to imply that, given the two sets of propositional letters $a = P(i)$ and $b = P(j)$, the relation $a \odot b$ holds according to the OPM $M_{\mathscr{P}(AP)}$.

Because defining an OPM on $\mathscr{P}(AP)$ is quite impractical, the set of atomic propositions AP will often be partitioned in the two sets Λ and Σ. Λ is a set of generic propositional letters, that can be associated to any word position without restrictions. Σ is the set of structural labels, whose elements will be typeset in **boldface**. These labels define the structure of the word with respect to OP relations: OPMs will be defined on Σ, and the corresponding matrix on $\mathscr{P}(AP)$ will be derived by transposing the relations defined on elements of Σ to subsets of AP containing those elements: if the relation $\mathbf{a} \odot \mathbf{b}$ holds with $\mathbf{a}, \mathbf{b} \in \Sigma$, $\odot \in \{\lessdot, \doteq, \gtrdot\}$ and $\mathbf{a} \in a \subset AP$ and $\mathbf{b} \in b \subset AP$, then $a \odot b$. Of course, each position is associated to only one structural symbol in Σ, while relations between subsets of AP containing more than one of such symbols are left undefined. If M_Σ is the OPM on Σ, it must be complete for OPTL to be able to express the universal language Σ^*.

Another example of OP word is shown in Figure 4.3. In the example, $\Sigma = \{+, *, \mathbf{e}, \#\}$, where $\#$ is an additional symbol that yields precedence to all other elements of Σ, and from which all other structural labels take precedence. The word contains also the propositional letters in $\Lambda = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}, \mathrm{f}\}$, displayed under structural labels in the figure. The OPM is shown in Figure 4.2, and it can be used to infer OP relations between subsets of AP: for example, $\{\mathbf{e}, \mathrm{a}\} \gtrdot \{+\}$.

4.2.2 The Chain Relation

The formal semantics of OPTL is based on the *chain* relation, which allows us to express properties on the structure of OP words by means of the concept of chain presented in Definition 16. Since the context-free structure given to a string by an OPM can be fully characterized by this concept, the chain relation gives to OPTL a

	$+$	$*$	$\mathbf{e}$
$+$	$\gtrdot$	$\lessdot$	$\lessdot$
$*$	$\gtrdot$	$\gtrdot$	$\lessdot$
$\mathbf{e}$	$\gtrdot$	$\gtrdot$	

Figure 4.2: Operator Precedence Matrix of grammar GAE_1 from [MP18].

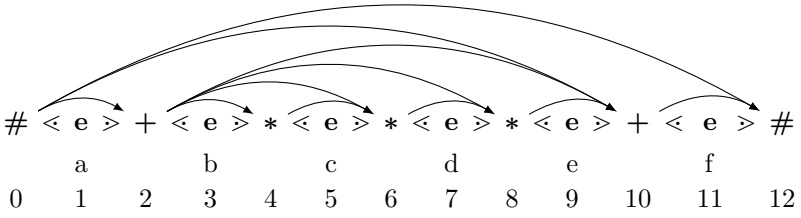


Figure 4.3: Example of an operator precedence word. Structural labels are in bold face, and the structure is defined by the OP Matrix of Figure 4.2. The χ relation is shown by arrows: there is an arrow between positions i and j iff $\chi(i, j)$.

considerable advantage in expressive power over the regular properties expressible in LTL.

Definition 24 (Chain Relation χ)**.** The relation $\chi \subseteq U \times U$ holds between two positions $i < j$ if i and j are the context of a chain. More formally, let $i = i_1 < i_2 < \cdots < i_{n-1} < i_n = j$ be the consecutive positions between i and j, and let $w = a_1 a_2 \cdots a_{n-1} a_n$ be a string on $\mathscr{P}(AP)$ such that for any $k \in \{1, \ldots, n\}$ we have $a_k = P(i_k)$. We write $\chi(i, j)$ if ${}^{a_1}[a_2 \cdots a_{n-1}]^{a_n}$ is a simple or composed chain.

For any two positions $i, j \in U$, $i < j$, the *maximal forward chain* relation $\overrightarrow{\chi} \subseteq U \times U$ is defined as follows:

$$\overrightarrow{\chi}(i, j) \iff j = \max\{k \in U \mid \chi(i, k)\}.$$

Similarly, the *maximal backward chain* relation $\overleftarrow{\chi} \subseteq U \times U$ is defined as

$$\overleftarrow{\chi}(i, j) \iff i = \min\{k \in U \mid \chi(k, j)\}.$$

Note that the two maximal chain relations are one-to-one.

In Figure 4.3, word positions that form the context of a chain are linked by an arrow: for example, positions 2 and 6 are in the chain relation ($\chi(2, 6)$), and the chain they form is not forward-maximal ($\neg\overrightarrow{\chi}(2, 6)$), but it is backward-maximal ($\overleftarrow{\chi}(2, 6)$). Conversely, the chain between positions 2 and 10 is forward-maximal ($\overrightarrow{\chi}(2, 10)$).

4.2.3 Operators

Propositional Operators and Symbols

For any word w and $\mathrm{a} \in AP$, we have $(w, i) \models \mathrm{a}$ iff $\mathrm{a} \in P(i)$. The *not* ($\neg$) and *and* ($\wedge$) symbols have the usual semantics from propositional logic.

Simple Temporal Operators

For any word w and position $i \in U$, and for any well-formed formula φ, the following logical operators are defined:

- $(w, i) \models \bigcirc\varphi$ iff $(w, i+1) \models \varphi$;
- $(w, i) \models \ominus\varphi$ iff $(w, i-1) \models \varphi$;
- $(w, i) \models \bigcirc\chi\varphi$ iff there exists a position $j \in U$ such that $\overrightarrow{\chi}(i, j)$ and $(w, j) \models \varphi$;
- $(w, i) \models \ominus_\chi\varphi$ iff there exists a position $j \in U$ such that $\overleftarrow{\chi}(j, i)$ and $(w, j) \models \varphi$.

In Figure 4.3, $(w, 6) \models \bigcirc\chi *$ because $\overrightarrow{\chi}(6, 8)$ and $(w, 8) \models *$, and $(w, 6) \models \ominus_\chi +$ because $\overleftarrow{\chi}(2, 6)$ and $(w, 2) \models +$.

Path Operators

A *path* of length $n \in \mathbb{N}$ between two string positions $i, j \in U$ is a sequence of positions $i_1 < i_2 < \cdots < i_n$, with $i \leq i_1$ and $i_n \leq j$. Given a set of paths Π, an *until* operator can be defined as follows: for any word w and position $i \in U$, and for any two well-formed formulas φ and ψ, $(w, i) \models \varphi\, \mathcal{U}^\Pi\, \psi$ iff there exist a position $j \in U$, $j \geq i$, and a path $i_1 < i_2 < \cdots < i_n$ between i and j in Π such that $(w, i_k) \models \varphi$ for any $1 \leq k < n$, and $(w, i_n) \models \psi$. A *since* operator can be defined analogously: for any position $j \in U$, $(w, j) \models \varphi\, \mathcal{S}^\Pi\, \psi$ iff there exists a position $i \in U$, $i \leq j$, and a path $i_1 < i_2 < \cdots < i_n$ between i and j in Π such that $(w, i_k) \models \varphi$ for any $1 < k \leq n$, and $(w, i_1) \models \psi$. Note that, with the definitions above, a path from i to j does not necessarily start in i and end in j, but it can also begin and end in positions between them. This, however, will only be true for hierarchical paths.

Three different types of until and since operators are defined, based on the classes of paths described in the following paragraphs.

Linear Operators. A *linear path* of length $n \in \mathbb{N}$ between i and j is a sequence of positions $i = i_1 < i_2 < \cdots < i_n = j$ such that for any $1 \leq k < n$, we have $i_{k+1} = i_k + 1$.

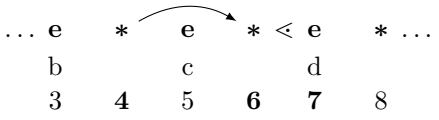
The until and since operators based on linear paths are denoted as $\mathcal{U}$ and $\mathcal{S}$, respectively; they correspond to the linear operators from classic LTL.

Summary Operators. Let $O \subseteq \{\lessdot, \doteq, \gtrdot\}$ be a set of operator-precedence relations. A *forward operator summary* path between positions i and j is a sequence of positions $i = i_1 < i_2 < \cdots < i_n = j$ such that, for any $1 \leq k < n$,

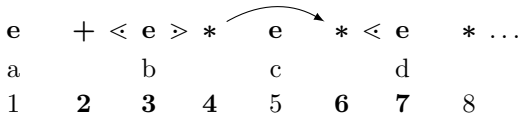
$$i_{k+1} = \begin{cases} h & \text{if } \overrightarrow{\chi}(i_k, h) \text{ and } h \leq j; \\ i_k + 1 & \text{if } i_k \odot i_k + 1 \text{ with } \odot \in O, \text{otherwise.} \end{cases}$$

The *operator-summary until* is defined by considering this class of paths as the set Π, and it is denoted as $\mathcal{U}^{\boxdot}$. Since the existence of these paths depends on the chosen set of OP relations, $\boxdot$ is replaced with the relations contained in O. For example, the until operator on paths allowing only the $\lessdot$ and $\doteq$ relations is denoted as $\mathcal{U}^{\lessdot\doteq}$; and the one that only allows the $\lessdot$ and $\gtrdot$ relations is denoted as $\mathcal{U}^{\gtrdot\lessdot}$.

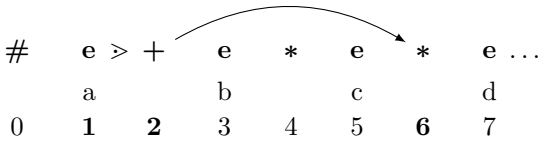
A Forward OP Summary path with $O = \{\lessdot, \gtrdot\}$ between positions 4 and 7 of the OP word of Figure 4.3 is shown in Figure 4.4a. It skips the body of the maximal chain



(a) Example of Forward OP Summary path.



(b) Example of Forward OP Summary path.



(c) Example of Backward OP Summary path.

Figure 4.4: Examples of OP Summary paths on the OP word of Figure 4.3. OP relations are shown then the path includes the linear edge between consecutive positions, and an arrow is shown when the path skips the body of a chain. Position numbers included in the paths are typeset in boldface.

$\overrightarrow{\chi}(4, 6)$, and, because all chains starting from 6 end after position 7 and $\lessdot \in O$, it ends there. This path witnesses the truth of $(w, 4) \models * \mathcal{U}^{\gtrdot \lessdot}$ d. Note that the bodies of chains that are not maximal are not skipped: Figure 4.4b shows a path of the same kind from 2 to 7, which also includes position 3. Since the chain from 2 to 4 is not forward-maximal, all paths starting in 2 and ending before 10 must include position 3, and $* \mathcal{U}^{\gtrdot \lessdot}$ d does not hold in 2, because it would require $(w, 3) \models *$.

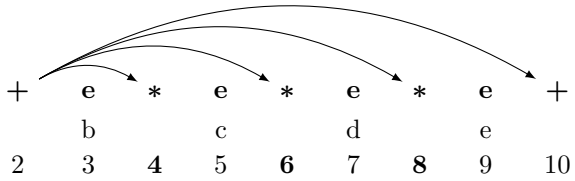
A *backward operator summary* path between positions i and j is a sequence of positions $i = i_1 < i_2 < \cdots < i_n = j$ such that, for any $1 < k \leq n$,

$$i_{k-1} = \begin{cases} h & \text{if } \overleftarrow{\chi}(h, i_k) \text{ and } h \geq i; \\ i_k - 1 & \text{if } i_k - 1 \odot i_k \text{ with } \odot \in O, \text{otherwise.} \end{cases}$$

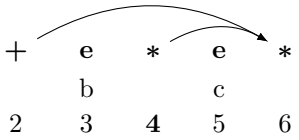
The *operator-summary since* is based on this class of paths and it is denoted as $\mathcal{S}^{\boxdot}$, replacing $\boxdot$ with the precedence relations contained in O. It is defined analogously to the operator-summary until.

An example of Backward OP Summary path with $O = \{\lessdot, \gtrdot\}$ is displayed in Figure 4.4c. It starts in position 1 and, since it is not part of the context of any chain and $\gtrdot \in O$, it continues with position 2; it then skips the body of the backward maximal chain $\overleftarrow{\chi}(2, 6)$. This path is a witness for $(w, 6) \models (+ \vee *)\, \mathcal{S}^{\gtrdot \lessdot}$ a.

Hierarchical Operators. A *yield-precedence hierarchical path* of length $n \in \mathbb{N}$ starting in $i \in U$ is a sequence of word positions $i_1 < i_2 < \cdots < i_n$, with $i < i_1$, such that for any $1 \leq k \leq n$ we have $i \lessdot i_k$ and $\chi(i, i_k)$; moreover, there is no i'_k that satisfies these two properties and $i_{k-1} < i'_k < i_k$.



(a) Example of yield-precedence hierarchical path.



(b) Example of take-precedence hierarchical path.

Figure 4.5: Examples of hierarchical paths on the OP word of Figure 4.3. (The positions that are part of the paths are typeset in boldface.)

The yield-precedence hierarchical until and since operators are based on this class of paths and they are denoted as $\mathcal{U}^{\uparrow}$ and $\mathcal{S}^{\downarrow}$, respectively. Additionally, for the until operator i_1 must be the leftmost position enjoying the above properties (i.e. there is no i'_k s.t. $i < i'_k < i_1$ enjoying them), and for the since operator i_n must be the rightmost.

Figure 4.5a shows an example of a yield-precedence hierarchical path, made of positions 4, 6 and 8. Note that position 10 is not part of the path because, although $\chi(2, 10)$, the symbol at position 2 does not yield precedence to the one at position 10 (in fact $+ \gtrdot +$ according to the OP matrix of Figure 4.3). This path witnesses the fact that $(w, 2) \models * \mathcal{U}^{\uparrow} (\bigcirc \mathrm{e})$, and $(w, 2) \models * \mathcal{S}^{\downarrow} (\ominus \mathrm{b})$.

Similarly, a *take-precedence hierarchical path* of length $n \in \mathbb{N}$ ending in $j \in U$ is a sequence of word positions $i_1 < i_2 < \cdots < i_n$, with $i_n < j$, such that for any $1 \leq k \leq n$ we have $i_k \gtrdot j$ and $\chi(i_k, j)$; there exists no position i'_k that satisfies these two properties and $i_k < i'_k < i_{k+1}$.

The take-precedence until and since operators are based on this class of paths, and they are denoted as $\mathcal{U}^{\downarrow}$ and $\mathcal{S}^{\uparrow}$ respectively. For the until operator, i_1 must be the leftmost position enjoying these properties, and for the since operator i_n must be the rightmost (i.e. there is no i'_k, $i_n < i'_k < j$, that satisfies them).

In Figure 4.5b, the only position that is in the $\gtrdot$ relation with position 6 and that forms a chain with it is 4: the only take-precedence hierarchical path with respect to 6 is made of this sole position. It witnesses, for example, $(w, 6) \models \top \mathcal{U}^{\downarrow} *$.

4.3 Examples

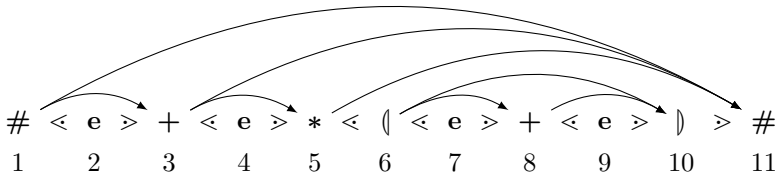
Before presenting further results regarding OPTL, we provide in this section two more thorough examples demonstrating the expressive its capabilities. The reader may find them helpful for better familiarizing with OPTL.

	$+$	$*$	$($	$)$	$\mathbf{e}$
$+$	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$
$*$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$
$($	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$
$)$	$\gtrdot$	$\gtrdot$		$\gtrdot$	
$\mathbf{e}$	$\gtrdot$	$\gtrdot$		$\gtrdot$	

Figure 4.6: Operator Precedence matrix of grammar GAE_P from [MP18].

4.3.1 Parenthesized Expressions

The following example relies on the grammar GAE_P from [MP18], whose OP matrix is shown in Figure 4.6. This OP matrix is an extension of the one in Figure 4.2, as it also includes open and closed parentheses.



1. Parentheses are used only when needed:

$$\square[(\implies (\ominus(* \vee \bigcirc\chi*) \wedge \Diamond^{\uparrow}+)].$$

Every time an open bracket '$($' occurs, either the previous position or the position right after the matched closing bracket must contain a '$*$'. The former requirement is achieved with a simple linear back operator. For the latter, it must be noted that, according to the OP matrix, the open bracket must be preceded by either a '$+$' or another '$($'. Since all terminals yield precedence to $($, and $)$ takes precedence from them all, the position preceding $($ must form the context of a chain along with the next after $)$: an abstract next operator can be used to make sure $)$ is followed by a '$*$'.

Finally, one of the subexpressions inside the brackets must contain a '$+$'. Since '$*$' takes precedence from '$+$', the latter must be at the root of the parsing subtree between $($ and $)$, forming the context of a chain with $($. A hierarchical eventually $\Diamond^{\uparrow}$ can be used to enforce this last requirement.

2. There cannot be more than two consecutive multiplications:

$$\square[* \implies \neg(\bigcirc\chi(* \wedge \bigcirc\chi*))].$$

Since '$+$' yields precedence to '$*$', which yields precedence to '$($' and '$\mathbf{e}$', consecutive occurrences of '$*$' must be sub-expressions of '$+$' and there must be either an '$\mathbf{e}$' or a matched pair of brackets between them. Because '$*$' takes precedence from itself, each occurrence of this operator must be a sub-expression of the one to its right, if present, and form with it the context of the outermost

chain starting from there. It is therefore possible to "hop" from a '$*$' to the next one by iterating the abstract next operator $\bigcirc\chi$. In order to forbid three or more consecutive multiplications, it suffices to state that it must not be possible to do two of such hops, starting from an occurrence of '$*$' and landing on another one of them.

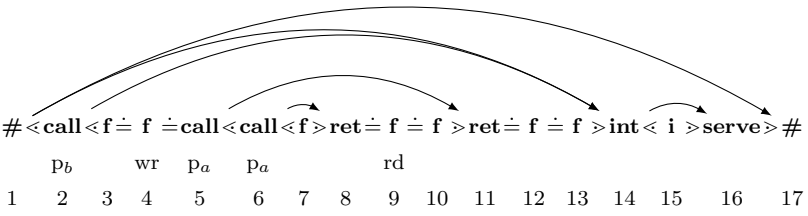
3. Parentheses cannot have more than two nesting levels:

$$\square[(\!(\implies \neg \bigcirc (\neg)\!) \, \mathcal{U}^{\gtrdot\lessdot} ((\!(\wedge \bigcirc (\neg)\!) \, \mathcal{U}^{\gtrdot\lessdot} (\!)))].$$

The above formula works by forbidding all paths that contain three or more nested parentheses. Once an open parenthesis is found, an OP summary operator that allows only the $\lessdot$ and $\gtrdot$ relations is evaluated on the next position. Its paths can both jump over matched pairs of parentheses, ignoring them, and end between them, going through subexpressions. For the whole until to be true, those paths must end when a second open parenthesis is found, that is followed by another path that terminates in a third open parenthesis. The negation in front of the outermost until operator prevents the occurrence of such a scenario, and the whole formula is only true if the expression does not contain more than two nested parentheses. Note that both until operators cannot jump past the closed parenthesis matched with the first '$(\!($', because the $\neg)\!)$ subformula is used as the left-hand side of both the until operators.

4.3.2 Function Calls and Interrupts

The following word represents a program computation, with nested function calls that can be truncated by interrupts. In the first and second rows, the atomic propositions associated to each position are displayed. The ones on the first row, in bold face, denote the structural role of the position, and the precedence relations among them are shown.



The word structure is determined by the following OP Matrix, where the structural labels of Σ are written in bold face:

	call	**ret**	**int**	**serve**	**f**	**i**
call	$\lessdot$	$\doteq$	$\gtrdot$		$\lessdot$	
ret	$\doteq$	$\gtrdot$	$\gtrdot$		$\doteq$	
int			$\lessdot$	$\doteq$		$\lessdot$
serve	$\gtrdot$		$\doteq$	$\gtrdot$		$\doteq$
f	$\doteq$	$\gtrdot$	$\gtrdot$		$\doteq$	
i			$\doteq$	$\gtrdot$		$\doteq$

Assuming that read (rd) and write (wr) labels cannot be associated to **call** or **ret** positions, OPTL allows us to express the following properties on the execution flow.

1. If precondition ρ holds whenever a function is called, then the function returns with postcondition θ holding:

$$\Box[(\mathbf{call} \wedge \rho) \implies \bigcirc\chi(\theta \wedge \mathbf{ret})].$$

   This is a classical procedure pre-/post-condition requirement. Because of the structure given to the word by the OP matrix, every **call** position forms the context of a chain with the **ret** or **int** position that terminates its body. The abstract next operator $\bigcirc\chi$ evaluates its argument in such position. Note that this formula is false whenever a function is terminated by an interrupt, and not a regular return.

2. If precondition ρ holds when a function is called and its execution is stopped by an interrupt, then postcondition θ holds:

$$\Box[(\mathbf{call} \wedge \rho \wedge \bigcirc\chi\mathbf{int}) \implies \bigcirc\chi\theta].$$

   In this case the abstract next operator is used to distinguish whether a function terminates regularly or due to an interrupt.

3. If a procedure writes to a variable, it will read said variable before writing to it again, regardless of what happens in nested function calls:

$$\Box[\mathrm{wr} \implies \bigcirc(\neg\mathrm{wr}\, \mathcal{U}^{\doteq}\, \mathrm{rd}) \wedge \neg \bigcirc (\mathbf{ret} \vee \mathbf{int})],$$

   where wr and rd are atoms associated to write/read positions. This formula uses an OP summary until that only allows the $\doteq$ precedence relation. The paths it considers jump over subcalls by skipping all subchains; none of them can enter the body of a subcall or exit from the current function frame, because any such path would include two position in the $\lessdot$ or $\gtrdot$ relation, respectively. Subformula $\neg\bigcirc(\mathbf{ret} \vee \mathbf{int})$ prevents the until operator from entering the outer function frame in case the write position is immediately followed by a **ret** or an **int**.

4. A procedure and its subcalls can never write to a variable:

$$\Box[\mathbf{call} \implies \Box^{\lessdot\doteq}\neg\mathrm{wr}].$$

   The globally operator is based on a summary until that only allows relations $\lessdot$ and $\doteq$. It evaluates $\neg$wr on paths that can both skip function bodies and enter them, by including **call** positions and the subsequent internal positions, which are in the $\lessdot$ relation with them.

5. If a procedure writes to a variable, it is read by the procedure itself or by any sub-call, and it is not written to before having been read:

$$\Box[\mathbf{call} \implies \bigcirc(\mathbf{ret} \vee \mathbf{int} \vee \Box^{\lessdot\doteq}[\mathrm{wr} \implies \bigcirc(\neg\mathrm{wr} \wedge (\mathbf{ret} \implies \ominus_\chi(\Box^{\lessdot\doteq}\neg\mathrm{wr})))\, \mathcal{U}^{\lessdot\doteq}\, \mathrm{rd}])].$$

The first $\Box^{\lessdot\doteq}$ operator imposes the truth of the subsequent subformula in all positions in the body of the main function call and of all subcalls in which a write action occurs. The outermost until operator asserts that non other write can occur before a read action: for every **ret** in its paths, a $\Box^{\lessdot\doteq}$ globally operator is evaluated starting from the matching **call** (reached by the abstract back operator $\ominus_\chi$), which ensures that no write action is performed in any subcall. Note that a linear next operator is applied to the left-hand side of the outermost until operator, so that the formula is correct even if the position right after the write operation contains a **ret** or an **int**.

Also, the abstract back operator is only applied in **ret** positions whose matching **call** position occurs after the write operation: it is not possible to "jump" back, before the write position. In fact, the paths considered by the outermost $\mathcal{U}^{\lessdot\doteq}$ operator cannot possibly include such a **ret** position, because they would include a $\gtrdot$ relation between the said position and the preceding one.

6. Function p_a can be called recursively at most one time (at most two instances of it can be active at once):

$$\Box[(\mathbf{call} \wedge \mathrm{p}_a) \implies \bigcirc\neg(\Diamond^{\lessdot\doteq}(\mathbf{call} \wedge \mathrm{p}_a \wedge \bigcirc(\Diamond^{\lessdot\doteq}(\mathbf{call} \wedge \mathrm{p}_a))))].$$

The above statement means that each time a **call** to p_a appears, it cannot happen that two nested calls to the same function are performed in its body. The negated formula after the first next operator has in fact the purpose of entering the body of the first instance of the function by means of the $\Diamond^{\lessdot\doteq}$ eventually operator, looking for another function **call**, whose body is searched for a third call by the rightmost eventually operator. If no such call is found, it means that there are no more than two nested calls to p_a.

7. While function p_a is being executed, no interrupt can occur:

$$\Box[\mathbf{int} \implies \neg\Diamond^{\downarrow}\mathrm{p}_a].$$

Due to the structure of the OP matrix, all calls form the context of a chain with the **int** that breaks them. So, in order to assert the above claim, it is sufficient to search all calls linked to an **int** position, looking for proposition p_a. This is done by the hierarchical eventually operator $\Diamond^{\downarrow}$, which considers all chains that end in the current position from the outermost to the innermost, and is true if its operand is true in the starting position of at least one of such chains. Negating this operator means asserting that in none of those calls p_a holds, which can also be expressed with $\Box^{\downarrow}\neg\mathrm{p}_a$.

8. If an interrupt occurs during the execution of procedure p_b, then the innermost instance of p_b must have written to a certain variable:

$$\Box[\mathbf{int} \implies (\Diamond^{\downarrow}\mathrm{p}_b \implies \neg\mathrm{p}_b\, \mathcal{S}^{\uparrow}\, (\mathrm{p}_b \wedge \bigcirc(\Diamond^{\doteq}\mathrm{wr})))].$$

First, subformula $\Diamond^{\downarrow}\mathrm{p}_b$ checks if at least one of the functions interrupted by the current **int** position is an instance of p_b. If this is true, then the hierarchical since operator $\mathcal{S}^{\uparrow}$ finds the innermost **call** to p_b, and it asserts that there must be at least a write in its body. Note that the OP summary eventually only allows the $\doteq$ relation, which prevents its paths from entering subcalls.

9. No interrupt can occur during a recursive call to procedure p_b:

$$\Box[\mathbf{int} \implies \neg\Diamond^{\downarrow}(\mathbf{call} \wedge \mathrm{p}_b \wedge \bigcirc(\Diamond^{\lessdot\doteq}(\mathbf{call} \wedge \mathrm{p}_b)))].$$

Every time there is an **int** position, this formula asserts that the body of none of the calls to p_b it interrupts can contain another call (i.e. a recursive call) to the same procedure. This is achieved with a combination of the hierarchical and OP-summary eventually operators $\Diamond^{\downarrow}$ and $\Diamond^{\lessdot\doteq}$, as explained in the previous examples.

4.4 Equivalence between Operators

In this section, we show that OP summary until operators can be used in place of the hierarchical operators, by providing a translation scheme for them.

First, we define some auxiliary formulas: for any $a \in \mathscr{P}(AP)$, formula

$$\sigma_a \equiv \left(\bigwedge_{\ell \in a} \ell\right) \wedge \left(\bigwedge_{\ell \in (AP \setminus a)} \neg\ell\right)$$

is true only in positions labeled with the set of atomic propositions a. Second, in order to be able to express OP relations in OPTL formulas, we define

$$\xi_{a\lessdot} \equiv \bigvee_{b \in \mathscr{P}(AP) \mid a \lessdot b} \sigma_b$$

which is true only in positions labeled with a set $b \in \mathscr{P}(AP)$ such that $a \lessdot b$. Formula $\xi_{\gtrdot a}$, which identifies positions labeled with a set $b \in \mathscr{P}(AP)$ such that $b \gtrdot a$, with $a \in \mathscr{P}(AP)$, is defined similarly.

The yield-precedence hierarchical until operator $\mathcal{U}^{\uparrow}$ can be translated as follows:

$$\varphi\, \mathcal{U}^{\uparrow}\, \psi \equiv \bigvee_{a \in \mathscr{P}(AP)} \Big[\sigma_a \wedge \bigcirc\Big(\big(\ominus_\chi\, \sigma_a \implies (\varphi \wedge \xi_{a\lessdot})\big)\, \mathcal{U}^{\gtrdot\doteq}\, \big(\ominus_\chi\, \sigma_a \wedge \xi_{a\lessdot} \wedge \psi\big)\Big)\Big].$$

Let us say the formula is evaluated in position $i \in U$, labeled with set $a \in \mathscr{P}(AP)$. The general idea behind this translation is to let an OP summary until run from $i+1$ to a position j such that $\chi(i,j)$ and $i \lessdot j$, in which ψ holds. We use $\ominus_\chi \sigma_a$ and $\xi_{a\lessdot}$ to ensure both requisites, respectively. In all positions between i and j that form the context of a chain with i, φ needs to be true: sub-formula $\ominus_\chi \sigma_a \implies (\varphi \wedge \xi_{a\lessdot})$ is used in the left-hand side of the OP summary until operator for this purpose. $\xi_{a\lessdot}$ ensures that all positions that form a chain with i are in $\lessdot$ relation with it, so that the path cannot pass j and continue with positions that form a chain with other positions labeled with a.

Since the OP relations depend on the sets of atomic propositions labeling i and j, separate instances of this formula are needed for each $a \in \mathscr{P}(AP)$, and they are all or-ed together.

The yield-precedence hierarchical since operator $\mathcal{S}^{\downarrow}$ can be translated as follows:

$$\varphi\, \mathcal{S}^{\downarrow}\, \psi \equiv \bigvee_{a \in \mathscr{P}(AP)} \Big[\sigma_a \wedge \bigcirc\chi\Big(\big((\ominus_\chi \sigma_a \wedge \xi_{a\lessdot}) \implies \varphi\big)\, \mathcal{S}^{\gtrdot\doteq}\, \big(\ominus_\chi\, \sigma_a \wedge \xi_{a\lessdot} \wedge \psi\big)\Big)\Big].$$

The reasoning behind this formula is similar to the previous one. This time we have the OP summary until operator starting from the rightmost position that forms a chain with the current position i, and going backwards until the position right after i, where it must stop because it does not allow the $\lessdot$ OP relation. Since the position that forms a maximal forward chain with i could also be in the $\doteq$ relation with i, formula $\xi_{a\lessdot}$ is needed on both sides of the OP summary until operator.

The translating formulas for $\varphi\, \mathcal{U}^{\downarrow}\, \psi$ and $\varphi\, \mathcal{S}^{\uparrow}\, \psi$ can be obtained in analogous ways.

It must be noted that the size of the translations described in this section is exponential in the length of the initial formula. In fact, let $n = |AP|$: $|\sigma_a| = \Theta(n)$ and consequently $|\xi_{a\lessdot}| = O(n2^n)$. If β is the translation of $\alpha = \varphi\, \mathcal{U}^{\uparrow}\, \psi$, then in the worst-case scenario in which both φ and ψ contain other hierarchical operators, we have $|\beta| = O((2^n)^{|\alpha|}|\xi_{a\lessdot}|) = O(n2^{2n|\alpha|})$.

4.4.1 An adequate set

The above results allow us to state the following:

Theorem 22. *The set of operators* $\{\neg, \wedge, \bigcirc, \ominus, \bigcirc\chi, \ominus_{\chi}, \mathcal{U}, \mathcal{S}, \mathcal{U}^{\boxdot}, \mathcal{S}^{\boxdot}\}$ *is adequate for OPTL.*

We conjecture that a similar result could be obtained also for the linear until and since operators, allowing us to remove $\mathcal{U}$ and $\mathcal{S}$ from this set.

4.5 Relationship with Nested Words

This section explores the relationship between OPTL and NWTL from [AAB+08], presented in Section 2.4. In particular, we will prove in Section 4.5.1 that OPTL is at least as expressively powerful as NWTL, and in Section 4.5.2 that this containment relation is actually strict.

4.5.1 Containment

The purpose of this section is to prove that NWTL $\subseteq$ OPTL. To do so, we will first show a way to translate a nested word into an OPTL word with significantly similar properties, and we will then proceed to devise a translation scheme for NWTL formulae.

Given any nested word $NW = \langle U, (P_{\mathrm{a}})_{\mathrm{a}\in\Lambda}, <, \mu, \mathtt{call}, \mathtt{ret}\rangle$ it is possible to build an equivalent algebraic structure for OPTL as $OW = \langle U', M^{NW}, P'\rangle$. Let $U = \{1, \ldots, n\}$, then if NW is finite, $U' = U \cup \{0, n{+}1\}$; if it is infinite $U' = U \cup \{0\}$. The set of propositional letters is $AP = \Lambda \cup \Sigma$ with $\Sigma = \{\mathbf{call}, \mathbf{ret}, \mathbf{i}, \#\}$; and the P' function is defined for any $i \in U$ as

$$P'(i) = \{\mathrm{a} \in \Lambda \mid i \in P_{\mathrm{a}}\} \cup \begin{cases} \{\mathbf{call}\} & \text{iff } \mathtt{call}(i); \\ \{\mathbf{ret}\} & \text{iff } \mathtt{ret}(i); \\ \{\mathbf{i}\} & \text{otherwise.} \end{cases}$$

Also, $P'(0) = P'(n+1) = \{\#\}$. Finally, the OP-matrix M^{NW} is defined as follows,

	call	**ret**	**i**
call	$\lessdot$	$\doteq$	$\lessdot$
ret	$\doteq$	$\gtrdot$	$\doteq$
i	$\doteq$	$\gtrdot$	$\doteq$

Figure 4.7: Sketch of the OP-matrix M^{NW}: the symbols in Σ are used as representative elements for the subsets of AP.

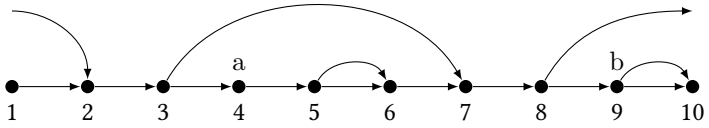


Figure 4.8: Representation of a nested word example.

for any $a, b \in \mathscr{P}(AP)$ such that $\# \notin a \cup b$:

$$M^{NW}_{(a,b)} = \begin{cases} \{\lessdot\} & \text{iff } \mathbf{call} \in a \wedge \mathbf{ret} \notin b; \\ \{\gtrdot\} & \text{iff } \mathbf{call} \notin a \wedge \mathbf{ret} \in b; \\ \{\doteq\} & \text{iff } (\mathbf{call} \notin a \wedge \mathbf{ret} \notin b) \vee (\mathbf{call} \in a \wedge \mathbf{ret} \in b). \end{cases}$$

If $\# \in a$ but $\# \notin b$, then $M^{NW}_{(a,b)} = \{\lessdot\}$; if $\# \in b$ but $\# \notin a$, then $M^{NW}_{(a,b)} = \{\gtrdot\}$. The structure of M^{NW} is shown in Figure 4.7. Note that, since in nested words no position can be both a call and a return, no word position of OW can be labeled with more than a single element of Σ, and we need not care about subsets of AP that contain both **call** and **ret**.

An example nested word is drawn in Figure 4.8, and its translation into an OPTL word is shown in Figure 4.9. In the translation, all the call positions form the context of a chain with the matched return, unless they are consecutive. Therefore, we will be able to use the chain relation to translate the matching relation of nested words, except for consecutive call/return positions, which will have to be considered separately. Also, unmatched returns and calls form a chain with the first and last '#' positions, respectively.

We will now prove a few properties regarding the translating OPTL structure, and the relationship between the NWTL matching relation μ and the OPTL chain relation χ.

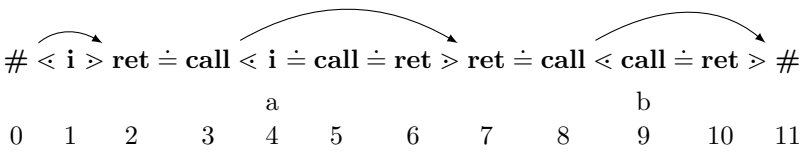


Figure 4.9: Translation into an OPTL structure of the nested word of Figure 4.8.

Lemma 23. *For any two distinct positions $i, j \in U$, $i < j$, if $\chi(i,j)$ holds then* $\mathbf{call} \in P'(i)$ *and* $\mathbf{ret} \in P'(j)$.

Proof. $\chi(i,j)$ means that i and j are the context of a chain, simple or composed. In both cases, according to Definition 16, position i must yield precedence to the next position, $i+1$: $i \lessdot i+1$. According to matrix M^{NW}, only call positions can yield precedence to any other position (unless $\# \in P'(i)$, which is not the case), and therefore $\mathbf{call} \in P'(i)$. Similarly, any position can take precedence from a return position only, and since $j-1$ must take precedence from j, we have $\mathbf{ret} \in P'(j)$. □

In Lemma 24 we will prove that, due to the definition of M^{NW}, relation χ in OW is one-to-one if restricted to U; i.e. for any $i, j, j' \in U$ if $\chi(i,j)$ and $\chi(i,j')$ then $j = j'$, and for any $i, i', j \in U$ if $\chi(i,j)$ and $\chi(i',j)$ then $i = i'$. Note that the above claim is not valid for positions 0 and $n+1$: indeed, we have $\chi(0,j)$ for all $j \in U$ that are unmatched returns, and $\chi(i, n+1)$ for all unmatched calls $i \in U$.

Lemma 24. *Relation χ in OW is one-to-one, if restricted to U.*

Proof. Suppose there exists a position $i \in U$ such that multiple chains start in i and end in distinct positions in U, i.e. there exist positions $j_1, \ldots, j_{n-1}, j_n \in U$ such that $i < j_1 < \cdots < j_{n-1} < j_n$ and $\chi(i, j_k)$ for any $1 \leq k \leq n$. Then, consider the outermost chain, $\chi(i, j_n)$: it must be a composed chain, because it contains the chain $\chi(i, j_{n-1})$. It must therefore be of the form ${}^{a_i}[w_i a_{j_{n-1}} \ldots]^{a_{j_n}}$, where w_i is the body of the chain that has positions i and j_{n-1} as its context: ${}^{a_i}[w_i]^{a_{j_{n-1}}}$ forms a chain. By the definition of composed chain, ${}^{a_i}[a_{j_{n-1}} a_{j_{n-1}+1} \ldots]^{a_{j_n}}$ must be a simple chain, which implies $a_i \lessdot a_{j_{n-1}}$. But we also have $\chi(i, j_{n-1})$, and because of Lemma 23 $\mathbf{call} \in P'(i)$ and $\mathbf{ret} \in P'(j_{n-1})$. Therefore, according to M^{NW}, we have $a_i \doteq a_{j_{n-1}}$, which contradicts our previous claim that $\chi(i, j_n)$.

An analogous reasoning can be applied to prove that there exists no position $j \in U$ such that multiple chains starting in U end in j. □

The following definitions will be useful in the proofs of the next lemmas.

Definition 25 (Nesting Depth)**.** Given a nested word w, based on the set of positions U and nesting relation μ, its nesting depth $\mathrm{nd}_{NW}(w)$ is defined as follows:

$$\mathrm{nd}_{NW}(w) = \begin{cases} 0 & \text{iff } \forall i, j \in U : \neg\mu(i,j); \\ \max\{\, \mathrm{nd}_{NW}(w_{i+1,j-1}) + 1 \\ \qquad \mid i, j \in w \wedge \mu(i,j)\} & \text{otherwise;} \end{cases}$$

where $w_{i+1,j-1}$ is the subword of w delimited by positions $i+1$ and $j-1$, included.

Given an Operator Precedence word x, based on the set of positions U and chain relation χ, its nesting depth $\mathrm{nd}(x)$ is defined as:

$$\mathrm{nd}(x) = \begin{cases} 0 & \text{iff } \forall i, j \in U : \neg\chi(i,j); \\ \max\{\, \mathrm{nd}(y) + 1 \\ \qquad \mid i, j \in x \wedge \chi(i,j) \wedge y \subset x_{i,j}\} & \text{otherwise;} \end{cases}$$

where y is a proper subword of $x_{i,j}$.

The nesting depth of the nested word of Figure 4.8 is $\mathrm{nd}_{NW}(w) = 2$, while the nesting depth of the OPTL word of Figure 4.9 is $\mathrm{nd}(x) = 1$.

Lemma 25 shows that, when translating a nested word into an OPTL word, the μ relation of the former is reflected into the χ relation that holds in the latter. This is, as we observed previously, not true for consecutive positions.

Lemma 25. *For any $i, j \in U$ such that $j > i + 1$, we have $\mu(i, j)$ iff $\chi(i, j)$.*

Proof. Given two word positions $i, j \in U$, we will first prove $\mu(i, j) \implies \chi(i, j)$, by induction on the nesting depth $\mathrm{nd}_{NW}(w_{i,j})$ of the subword $w_{i,j}$ delimited by them.

Base case. If $\mathrm{nd}_{NW}(w_{i,j}) = 1$, there cannot be any matched call-return pairs between i and j. Moreover, since $\mu(i, j)$, i is a call and j is a return, the positions between them cannot be unmatched calls or returns, and they must be all internal positions. Therefore in OW we have $\mathbf{call} \in P'(i)$, $\mathbf{ret} \in P'(j)$ and $\mathbf{i} \in P'(k)$ for any $k \in U$ such that $i < k < j$. So, the subword is structured as $i \lessdot (i + 1) \doteq \cdots \doteq (j - 1) \gtrdot j$, and i and j are the context of the simple chain ${}^{i}[(i + 1), \ldots, (j - 1)]^{j}$.

Inductive step. If $\mathrm{nd}_{NW}(w_{i,j}) = n > 1$, suppose that for any $h, k \in U$, $k > h + 1$, if $\mathrm{nd}_{NW}(w_{h,k}) < n$ then $\mu(h, k) \implies \chi(h, k)$. Let $i, j \in U$, $j > i + 1$, be such that $\mathrm{nd}_{NW}(w_{i+1,j-1}) < n$. Then, the word delimited by them is in the form $a_i x_i a_{i+1} x_{i+1} \ldots x_{j-1} a_{j-1} x_j a_j$, where, for any p such that $i \leq p \leq j$, $a_p = \varepsilon$ or it is an internal position, and either $x_p = \varepsilon$ or x_p is a nested word of nesting depth lower than n of the form $x_p = c_p y_p r_p$, with $\mathtt{call}(c_p)$ and $\mathtt{ret}(r_p)$. Also, since by the definition of the μ there cannot be unmatched calls or returns between i and j, we are sure $\mu(c_p, r_p)$ and, since the nesting depth of this subword is lower than n by hypothesis, also $\chi(c_p, r_p)$. Consequently, in OW we have $\mathbf{i} \in P'(a_p)$ (if any), $\mathbf{call} \in P'(c_p)$ and $\mathbf{ret} \in P'(r_p)$, and the subword contains the structure $a_i \lessdot c_i \doteq r_i \doteq a_{i+1} \doteq \cdots \doteq c_j \doteq r_j \gtrdot a_j$, which is a simple chain. Moreover, all substrings $c_p y_p r_p$ also form a chain, which allows us to state that $w_{i,j}$ is a composed chain.

The proof of $\chi(i, j) \implies \mu(i, j)$ relies on the fact that, according to Lemma 23, if $\chi(i, j)$ then $\mathbf{call} \in P'(i)$ and $\mathbf{ret} \in P'(j)$, which, by construction of P', implies $\mathtt{call}(i)$ and $\mathtt{ret}(j)$. The definition of the μ relation in Definition 5 states that, with the above conditions holding on i and j, there exists $i \leq k \leq j$ such that either $\mu(i, k)$ or $\mu(k, j)$. We will now prove by induction on the nesting depth $\mathrm{nd}(w_{i,j})$ of the OP subword delimited by i and j that the said k coincides with i or j.

Base case. If $\mathrm{nd}(w_{i,j}) = 1$, then $\chi(i, j)$ is a simple chain, i.e. between i and j there are no other chains besides $\chi(i, j)$. According to M^{NW}, this can happen only if all positions strictly between i and j are either labeled with $\mathbf{i}$, or if they are positions labeled with $\mathbf{call}$ immediately followed by a position labeled with $\mathbf{ret}$, which must not be j because the χ relation is one-to-one in this case (Lemma 24). In the latter case, since there is surely no other call/return position between such pairs, they must all be in relation μ. Therefore, there is no k between i and j that is either a call or a return and that is not matched with another return or call, respectively. Because the μ relation is one-to-one by definition, surely $\mu(i, j)$.

Inductive step. Suppose $\mathrm{nd}(w_{i,j}) = n > 1$, and for any $h, k \in U$ with $\mathrm{nd}(w_{h,k}) < n$, if $\chi(h, k)$ then $\mu(h, k)$. The body of chain $\chi(i, j)$ can contain positions labeled with $\mathbf{i}$ or consecutive call/return pairs that are in relation μ as we argued before.

Also, the chain is composed, and it contains subchains of depth lower than n. By the inductive hypothesis, the positions that are contexts of such subchains are in relation μ, and none of them coincides with i or j or the context of another chain, because of Lemma 24. Therefore, because the μ relation is one-to-one, we can conclude that there are no unmatched call or return positions between i and j that could be paired with one of them, and $\mu(i, j)$ must hold.

□

The following lemma establishes a correspondence between summary paths in NWTL and OP Summary paths in OPTL, enabling the translation of NWTL summary until operators with their operator-precedence counterparts.

Lemma 26. *Given any two word positions $i, j \in U$, $i \leq j$, the summary path between i and j in NW coincides with the Operator Precedence summary path between the same positions based on precedence relations $O = \{\lessdot, \doteq, \gtrdot\}$ in OW.*

Proof. This proof relies on the definition of summary path in NWTL, given in Definition 7, and of Operator Precedence Summary Paths, whose definition is given in Paragraph 4.2.3. Note that, due to the fact that the chain relation is one-to-one in OW (Lemma 24), we have $\chi(i, j) \iff \overleftarrow{\chi}(i, j) \iff \overrightarrow{\chi}(i, j)$ for any $i, j \in U$. Therefore, OP Forward and Backward Summary paths coincide, and we will simply refer to them as OP Summary Paths in the following. Moreover, since we will consider OP Summary Paths based on the set of all possible OP relations $O = \{\lessdot, \doteq, \gtrdot\}$, the second case of the definitions of OP Summary Paths always applies when the first does not.

We will prove that, given two positions $i, j \in U$, $i \leq j$, the summary path $\pi_{NW} = \{h_1, h_2, \ldots, h_n\}$ with $i = h_1 < h_2 < \cdots < h_n = j$ coincides with the OP summary path $\pi_{OW} = \{k_1, k_2, \ldots, k_m\}$ with $i = k_1 < k_2 < \cdots < k_m = j$, i.e. $n = m$ and $h_p = k_p$ for any $1 \leq p \leq n$. The proof is structured by induction on the distance between i and j.

Base case: $i = j$. Since by the definitions of both classes of paths the first position is always included, we have $\pi_{NW} = \pi_{OW} = \{i\}$.

Inductive step: $i - j > 0$. Suppose that for any $j' < j$, the summary path π'_{NW} between i and j' coincides with the OP summary path π'_{OW} between the same two positions. Let π' be the path between i and $j - 1$: by the inductive hypothesis, it is both a summary path in NW and an OP summary path in OW. The composition of the paths π_{NW} and π_{OW} between i and j depends on how position j is labeled.

ret $\in P'(j) \iff$ `ret`(j) : Suppose j is a matched return in NW, and that its matched call is in position $k \geq i$. In this case, π' is in the form $i = i_1 < i_2 < \cdots < k < \cdots < j - 1$ and, by the first case of Definition 7, π_{NW} is in the form $i = i_1 < i_2 < \cdots < k < j$. By Lemma 25, if $k \neq j - 1$ then $\mu(k, j)$ implies $\chi(k, j)$, which falls in the first cases of the definitions of OP summary paths. Therefore, also π_{OW} is in the form $i = i_1 < i_2 < \cdots < k < j$. If $k = j - 1$ we have $j - 1 \doteq j$ because $j - 1$ is a **call** and j is a **ret**, which falls in the second case of the definition of OP summary path, yielding to the same form of π_{OW}.

If, instead, j is a matched return whose matched call is before i or if it is an unmatched return, the second case of both definitions allows path π' to

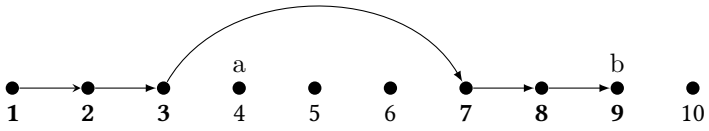


Figure 4.10: Outline of the summary path between positions 1 and 9 of the nested word of Figure 4.8.

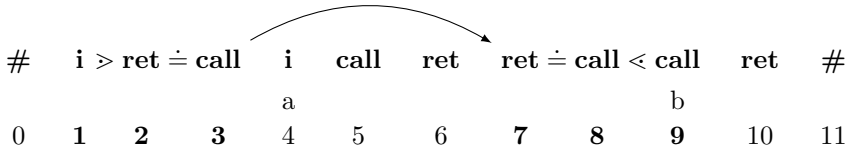


Figure 4.11: The OP summary path between positions 1 and 9 of the OPTL word of Figure 4.9. (Only precedence relations between positions that are part of the path are shown.)

be concatenated with j: the path between i and j is $\pi = \pi_{NW} = \pi_{OW} = \pi' \cup \{j\}$.

call $\in P'(j) \iff \texttt{call}(j)$: If j is a matched call position, the corresponding return must be in a position strictly greater than j, which prevents from applying the first case of the definitions of both summary paths and OP summary paths. If it is an unmatched call, it forms a chain with $n+1$, which is surely beyond j. Therefore, in both cases position j is simply concatenated to the path between i and $j-1$, which is the same for both nested words and OP words.

i $\in P'(j)$: Again, this trivially falls in the second case of the definitions of both classes of paths, and position j is concatenated to π', yielding $\pi = \pi_{NW} = \pi_{OW} = \pi' \cup \{j\}$.

□

Figures 4.10 and 4.11 show the structural analogy between a summary path in a nested word, and the OP summary path between the same positions of its translation into an OPTL word.

Now that we have established a certain degree of isomorphism between nested words and their proposed translation into OPTL structures, we can proceed further by giving a translation scheme between NWTL formulae on nested words and OPTL formulae on the corresponding OPTL structure.

Theorem 27 (NWTL $\subseteq$ OPTL). *Given an NWTL formula φ, it is always possible to translate it to an OPTL formula φ' such that, for any nested word w and position i, if w is translated into an OP word w' as described at the beginning of Section 4.5, then $(w,i) \models \varphi$ iff $(w',i) \models \varphi'$, with $i \in U$.*

Proof. Let w' be an OP word constructed from w as described at the beginning of Section 4.5. For any NWTL formula φ we will denote as $\varphi' = \alpha(\varphi)$ the OPTL formula

that satisfies $(w, i) \models \varphi$ iff $(w', i) \models \varphi'$. The translation function α is defined as follows:

$\alpha(\top) = \top$;

$\alpha(\mathrm{a}) = \mathrm{a}$ for any $a \in \Lambda$. This follows from the construction of w': $\mathrm{a} \in AP$, and P' is defined so that $i \in P_{\mathrm{a}}$ iff $\mathrm{a} \in P'(i)$.

$\alpha(\texttt{call}) = \mathbf{call}$ because, by construction, $\mathbf{call} \in P'(i)$ iff $\texttt{call}(i)$, for any i.

$\alpha(\texttt{ret}) = \mathbf{ret}$;

$\alpha(\neg\varphi) = \neg\alpha(\varphi)$;

$\alpha(\varphi \vee \psi) = \alpha(\varphi) \vee \alpha(\psi)$;

$\alpha(\bigcirc\varphi) = \bigcirc\alpha(\varphi)$: in NWTL $(w, i) \models \bigcirc\varphi$ iff $(w, i+1) \models \varphi$; and since w and w' share the set of word positions U, we have $(w', i) \models \bigcirc\varphi'$ iff $(w', i+1) \models \varphi'$ also in OPTL, with $\varphi' = \alpha(\varphi)$.

$\alpha(\ominus\varphi) = \ominus\alpha(\varphi)$ holds, similarly;

$\alpha(\bigcirc_\mu\varphi) = (\mathbf{call}\,\mathcal{U}^{\doteq}\,(\mathbf{ret} \wedge \alpha(\varphi))) \wedge \neg\mathbf{ret}$. For convenience, let $\gamma \equiv \mathbf{call}\mathcal{U}^{\doteq}(\mathbf{ret} \wedge \alpha(\varphi))$.

In NWTL we have $(w, i) \models \bigcirc_\mu\varphi$ iff there exists a word position j such that $\mu(i, j)$ and $(w, j) \models \varphi$. Because of Lemma 25, if such a position j exists and $j > i + 1$ then also $\chi(i, j)$ holds and, since the χ relation is one-to-one in this case (Lemma 24), we have $\overrightarrow{\chi}(i, j)$.

Now, consider the path only made of i and j. It is an OP summary path, because it falls in the first case of the definition. Since by construction $\mathbf{call} \in P'(i)$ and $\mathbf{ret} \in P'(j)$, if $\alpha(\varphi)$ holds in j, then γ is satisfied.

Note that the OP summary until in γ can only be true on the $\{i, j\}$ path considered above. In fact, none of the paths it allows can terminate in a position strictly between i and j, because it only allows the $\doteq$ OP relation. Moreover, no such path can go beyond j: since the only chain that starts in i has j as its context, j must be always included in the path, and because $\mathbf{call} \notin P'(j)$, the said until operator is only verified if the path ends in j. Therefore, γ is false if $\alpha(\varphi)$ does not hold in j.

If i and j are such that $\mu(i, j)$ but $j = i+1$, then the two positions do not form a chain. However, since $\mathbf{call} \in P'(i)$ and $\mathbf{ret} \in P'(j)$ we have $i \doteq j$, and the path made only of i and j is valid for the $\mathcal{U}^{\doteq}$ operator. Moreover, this path is also the only one allowed by such operator, because longer paths would surely include j, which is not a $\mathbf{call}$. So, γ is equivalent to the given NWTL formula also in this case.

Finally, if $(w, i) \not\models \bigcirc_\mu\varphi$ because position i is not a matched call, then also the translating formula is false. In fact, if $\mathbf{i} \in P'(i)$ then γ is false because $\mathbf{call} \notin P'(i)$ and $\mathbf{ret} \notin P'(i)$. If i is an unmatched call, then $\chi(i, n+1)$, but $\mathbf{ret} \notin P'(n+1)$, so the path $\{i, n+1\}$ cannot be considered by γ. Also, $i+1$ cannot be a return: if it is an internal position, then any path longer than 1 position must include it, which falsifies both arguments of the until operator; if it is another call, then $i \lessdot i+1$, which cannot be part of any path because the

until operator in γ only allows the $\doteq$ relation. If i is a return position, then the translating formula is trivially falsified by the right-hand side of the $\wedge$ operator.

$\alpha(\ominus_\mu \varphi) = (\mathbf{ret}\ \mathcal{S}^{\doteq}\ (\mathbf{call} \wedge \alpha(\varphi))) \wedge \neg\mathbf{call}$: this case is justified by a reasoning similar to the previous one.

$\alpha(\varphi\, \mathcal{U}^\sigma\, \psi) = \alpha(\varphi)\, \mathcal{U}^{\gtrdot\doteq\lessdot}\, \alpha(\psi)$: due to Lemma 26, we know that for any two positions i and j in w, the summary path between them coincides with the OP Summary Path in w' allowing all operator precedence relations. So, the set of summary paths starting from position i in w corresponds to the set of OP summary paths starting from i in w'. The definitions of the OP summary until operator in OPTL and of the summary until operator in NWTL are therefore equivalent, and the latter can be translated into an instance of the former.

$\alpha(\varphi\, \mathcal{S}^\sigma\, \psi) = \alpha(\varphi)\, \mathcal{S}^{\gtrdot\doteq\lessdot}\, \alpha(\psi)$: the correctness of this translation can be shown similarly to that of the summary until operator.

Since we have shown that every NWTL formula φ can be effectively translated into an equivalent OPTL formula $\alpha(\varphi)$, by induction on the syntactic structure of φ we can conclude that NWTL $\subseteq$ OPTL. □

For example, take formula $\varphi = (\neg \mathrm{a})\, \mathcal{U}^\sigma\, \mathrm{b}$. The nested word of Figure 4.8 satisfies φ, and this is witnessed by the summary path outlined in Figure 4.10. According to Theorem 27, φ can be translated into formula

$$\varphi' = (\neg \mathrm{a})\, \mathcal{U}^{\gtrdot\doteq\lessdot}\, \mathrm{b}.$$

Indeed, φ' is satisfied by the OPTL structure of Figure 4.9, where the OP summary path of Figure 4.11, which covers the same positions as the one of Figure 4.10, is a possible witness for its truth.

4.5.2 Strict Containment

In order to prove that OPTL is strictly more expressive than NWTL, we rely on the fact that the family of Operator Precedence Languages strictly contains Visibly Pushdown Languages, and nested words are based upon the latter. This fact was proved in [CRM12], whose authors introduced a language that is OP but not VP (cf. Statement 14). This language is $L_{123} = L_1 \cup L_2 \cup L_3$, and it is the union of the three languages

$$L_1 = \{\mathbf{b}^n \mathbf{c}^n \mid n \geq 1\}, \quad L_2 = \{\mathbf{f}^n \mathbf{d}^n \mid n \geq 1\}, \quad L_3 = \{\mathbf{e}^n (\mathbf{fb})^n \mid n \geq 1\}.$$

First, we need to prove that there exists an OPTL formula that denotes L_{123}, i.e. that is true for all and only the strings that are part of the language. We start by giving a simpler formula that denotes L_1.

Lemma 28. *Language L_1 is denoted by formula*

$$\lambda_1 = \Box\big[(\mathbf{b} \implies (\bigcirc\chi\mathbf{c} \vee \bigcirc\mathbf{c})) \wedge (\mathbf{c} \implies (\ominus_\chi \mathbf{b} \vee \ominus\mathbf{b}))\big],$$

with respect to the OP matrix of Figure 4.12a.

	b	c
b	$\lessdot$	$\doteq$
c		$\gtrdot$

(a)

	b	c	d	e	f
b	$\lessdot$	$\doteq$			$\gtrdot$
c		$\gtrdot$			
d			$\gtrdot$		
e				$\lessdot$	$\doteq$
f	$\doteq$		$\doteq$		$\lessdot$

(b)

Figure 4.12: OP matrices of languages L_1 (4.12a) and L_{123} (4.12b).

Proof. The OP matrix of Figure 4.12a allows **c**s to be followed by other **c**s only, constraining strings to the form $\mathbf{b}^n\mathbf{c}^m$, with $n, m \geq 0$. We must show that λ_1 holds on a string if and only if it is balanced ($n = m$).

If a string is balanced, each **b** forms the context of a chain with a distinct **c**, except for the one that is immediately followed by a **c** (Figure 4.13a). In the former case, $\bigcirc\chi\mathbf{c}$ holds in positions labeled with **b**, while $\ominus_\chi\mathbf{b}$ holds in those labeled with **c**. In the latter, $\bigcirc\mathbf{c}$ and $\ominus\mathbf{b}$ hold, respectively.

On the other hand, unbalanced strings adhere to the pattern shown either in Figure 4.13b or 4.13c. Because **b**s take precedence form # only, those in excess can form a chain with the terminator symbol only, and $\bigcirc\chi\mathbf{c}$ does not hold in such positions. A similar statement can be made for positions in excess labeled with **c**, in which $\ominus_\chi\mathbf{b}$ is false. □

Formulas that denote L_2 and L_3 can be devised similarly. We can therefore obtain a formula that denotes the whole L_{123}:

Lemma 29. *Language L_{123} is denoted by the formula*

$$\lambda_{123} = (\mathbf{b} \vee \mathbf{f} \vee \mathbf{e}) \wedge (\mathbf{b} \implies \lambda_1) \wedge (\mathbf{f} \implies \lambda_2) \wedge (\mathbf{e} \implies \lambda_3),$$

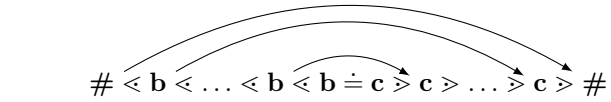
with respect to the OP matrix of Figure 4.12b, where λ_1 is defined in Lemma 28, and

$$\lambda_2 = \Box\big[(\mathbf{f} \implies (\bigcirc\chi\mathbf{d} \vee \bigcirc\mathbf{d})) \wedge (\mathbf{d} \implies (\ominus_\chi\mathbf{f} \vee \ominus\mathbf{f}))\big],$$

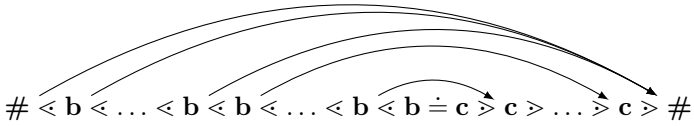
$$\begin{aligned}\lambda_3 = \Box\big[&\neg\mathbf{c} \wedge \neg\mathbf{d} \\ &\wedge (\mathbf{e} \implies (\bigcirc\chi\mathbf{f} \vee \bigcirc\mathbf{f})) && (4.1)\\ &\wedge (\mathbf{f} \implies (\ominus_\chi\mathbf{e} \vee \ominus\mathbf{e}) \wedge \bigcirc\mathbf{b}) && (4.2)\\ &\wedge (\mathbf{b} \implies \ominus\mathbf{f})\big]. && (4.3)\end{aligned}$$

Proof. Formula λ_{123} distinguishes whether the string belongs to L_1, L_2 or L_3 by looking at the label of the first string position, which is then used as the antecedent of implications whose consequent is the λ_i formula that denotes language L_i. This works because each one of those languages contains strings that begin with a letter different from the others, and $\mathbf{b} \vee \mathbf{f} \vee \mathbf{e}$ makes sure that no string starting with a different letter is accepted.

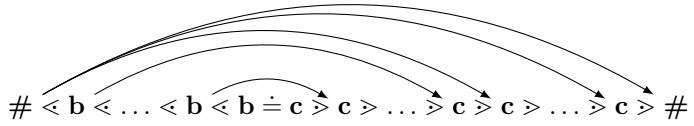
We have already proved in Lemma 28 that λ_1 denotes L_1 under the OP matrix of Figure 4.12a. With respect to **b** and **c**, the OP matrix of Figure 4.12b only differs from



(a)



(b)



(c)

Figure 4.13: Different possible structures for strings according to the OP matrix of Figure 4.12a. The string in (4.13a) is balanced, the one in (4.13b) contains more **b**s than **c**s, and the one in (4.13c) has more **c**s than **b**s.

the latter because **b**s can be followed by **f**s ($\mathbf{b} \gtrdot \mathbf{f}$). However, strings where at least a **b** is followed by an **f** would not satisfy subformula $\bigcirc\mathbf{c}$ that must hold in all positions labeled with **b**, according to λ_1, which is therefore still sufficient to denote L_1.

A similar reasoning proves that λ_2 correctly denotes L_2, taking into account that, despite $\mathbf{f} \doteq \mathbf{b}$, an **f** followed by a **b** would violate $\bigcirc\mathbf{d}$.

Formula λ_3 denotes L_3 in an analogous way: implications (4.1) and (4.2) make sure **e**s are balanced with **f**s, while $\bigcirc\mathbf{b}$ and $\ominus\mathbf{f}$ in (4.2) and (4.3) make sure each **f** is followed by a **b**. Moreover, $\neg\mathbf{c} \wedge \neg\mathbf{d}$ prevent strings containing spurious letters from being accepted. □

Now that we have proved that language L_{123} can be expressed in OPTL, we can state the following:

Theorem 30 (NWTL ⊂ OPTL)**.** *OPTL is strictly more expressive than NWTL.*

Proof. In Theorem 27 we proved that for each NWTL there exists an equivalent OPTL formula. We must now prove that there exists at least an OPTL formula for which no equivalent NWTL formula can be given.

The authors of [AAB+08] proved that NWTL model checking can be performed for any NWTL formula by building a nondeterministic Büchi nested word automaton (BNWA) that only accepts runs satisfying that formula. As shown in [AM09], regular languages of nested words correspond to the family of visibly pushdown languages, and BNWAs can only accept such languages. In particular, language L_{123} was proved not to be a VPL in [CRM12], and therefore it cannot be denoted by any NWTL formula. However, in Lemma 29 we proved that L_{123} can be denoted by OPTL formula λ_{123}: consequently, there exists no equivalent NWTL formula. □

Note that this result follows from the fact that OPTL is based on a wider class of languages than NWTL, and it says nothing about the expressiveness of OPTL when restricted to nested words.

4.6 Model-Checking

In this section, we present a model checking procedure for OPTL, first limited to finite words, and later extended to infinite words. This procedure allows to effectively build a nondeterministic OPA recognizing models of an OPTL formula with at most a number of states exponential in the length of the formula.

4.6.1 Model-Checking for Finite Words

Model-checking of OPTL formulas on finite words is performed by following an automata theoretic approach. Given an OPTL formula φ, we will show how to build a nondeterministic Operator Precedence Automaton $\mathcal{A}$ that only accepts models satisfying φ. Such an automaton is defined as $\mathcal{A} = \langle \mathscr{P}(AP), M_{\mathscr{P}(AP)}, \mathrm{Atoms}(\varphi), I, F, \delta \rangle$, where $\langle \mathscr{P}(AP), M_{\mathscr{P}(AP)} \rangle$ is an operator precedence alphabet made of all subsets of atomic propositions; $\mathrm{Atoms}(\varphi)$ is the set of *atoms* of φ, used as the states of $\mathcal{A}$; I and F are the sets of initial and final states, respectively; and $\delta \subseteq \mathrm{Atoms}(\varphi) \times (\mathscr{P}(AP) \cup \mathrm{Atoms}(\varphi)) \times \mathrm{Atoms}(\varphi)$ is the transition relation.

We will first illustrate the way the automaton checks the satisfaction of linear next and back operators, which is analogous to LTL model checking, while all other operators will be separately treated afterwards. In order to define $\mathrm{Atoms}(\varphi)$, we define $\mathrm{Cl}(\varphi)$, the *closure* of formula φ. $\mathrm{Cl}(\varphi)$ is the smallest set such that:

- $\varphi \in \mathrm{Cl}(\varphi)$;
- $AP \subseteq \mathrm{Cl}(\varphi)$;
- if $\psi \in \mathrm{Cl}(\varphi)$ and $\psi \neq \neg\theta$ for any OPTL formula θ, then $\neg\psi \in \mathrm{Cl}(\varphi)$;
- if any of $\neg\psi$, $\bigcirc\psi$ or $\ominus\psi$ is in $\mathrm{Cl}(\varphi)$, then $\psi \in \mathrm{Cl}(\varphi)$;
- if any of $\psi \wedge \theta$ or $\psi \vee \theta$ is in $\mathrm{Cl}(\varphi)$, then $\psi \in \mathrm{Cl}(\varphi)$ and $\theta \in \mathrm{Cl}(\varphi)$.

The set $\mathrm{Atoms}(\varphi)$ contains all sets $\Phi \subseteq \mathrm{Cl}(\varphi)$ such that

- for every $\psi \in \mathrm{Cl}(\varphi)$, $\psi \in \Phi$ iff $\neg\psi \notin \Phi$;
- if $\psi \wedge \theta \in \Phi$, then $\psi \in \Phi$ and $\theta \in \Phi$;
- if $\psi \vee \theta \in \Phi$, then $\psi \in \Phi$ or $\theta \in \Phi$.

The set of initial states I consists only of those atoms that contain φ, but not containing any $\ominus$ formula.

The way the three components δ_{push}, δ_{shift} and δ_{pop} of the transition relation δ are defined is crucial to enforce the satisfaction of temporal operators.

Push and shift transitions: for any $\Phi, \Theta \in \mathrm{Atoms}(\varphi)$ and $a \in \mathscr{P}(AP)$, $(\Phi, a, \Theta) \in \delta_{push}$ and $(\Phi, a, \Theta) \in \delta_{shift}$ iff

1. for any $\mathrm{p} \in AP$, $\mathrm{p} \in \Phi$ iff $\mathrm{p} \in a$;

2. $\bigcirc\psi \in \Phi$ iff $\psi \in \Theta$;
3. $\ominus\psi \in \Theta$ iff $\psi \in \Phi$;

Pop transitions: for any $\Phi, \Theta, \Psi \in \mathrm{Atoms}(\varphi)$, $(\Phi, \Theta, \Psi) \in \delta_{pop}$ iff Ψ is the smallest set such that

4. $\Phi \subseteq \Psi$;
5. for any $\mathrm{p} \in AP$, $\mathrm{p} \in \Phi$ iff $\mathrm{p} \in \Psi$.

The satisfaction of linear next and back operators is ensured by *shift* and *push* transitions, similarly to regular automaton transitions in LTL.

Note that constraint (4) prevents *pop* transitions from removing formulas inserted by linear next operators. This is required because *pop* transitions do not actually process terminal symbols, but *shift* and *push* transitions do: all formulae that hold in a string position must be present in the state right before one of such transitions consumes the related terminal symbol.

Finally, the set of the final states F must be defined, so that no string is accepted if any of the requirements of temporal operators in formula φ have not been satisfied. So, of course for any $\Phi \in \mathrm{Atoms}(\varphi)$ and $\bigcirc\psi \in \mathrm{Cl}(\varphi)$, $\Phi \notin F$ if $\bigcirc\psi \in \Phi$. Also, we must impose that $AP \cap \Phi = \{\#\}$, because the automaton would otherwise accept strings as if there was any other subset of AP in place of the terminal symbol $\#$.

Matching-Next ($\bigcirc\chi$) Operator

Model checking of this operator works by exploiting the automaton's stack to propagate its requirements. In order to do this, the $\bigcirc\chi^{\mathrm{s}}$ and $\bigcirc\chi^{\mathrm{end}}$ auxiliary operators must be introduced: if $\bigcirc\chi\psi \in \mathrm{Cl}(\varphi)$, then $\psi, \bigcirc\chi^{\mathrm{s}}\psi, \bigcirc\chi^{\mathrm{end}}\psi, \in \mathrm{Cl}(\varphi)$. The satisfaction of formula $\bigcirc\chi\psi$ is ensured by constraints on the automaton's transitions. For any $\Phi, \Theta, \Psi \in \mathrm{Atoms}(\varphi)$ and $a \in \mathscr{P}(AP)$, $(\Phi, a, \Theta) \in \delta_{push}$ iff

1. if $\bigcirc\chi\psi \in \Phi$, then $\bigcirc\chi^{\mathrm{s}}\psi \in \Theta$ and $\bigcirc\chi^{\mathrm{end}}\psi \notin \Theta$;

$(\Phi, a, \Theta) \in \delta_{shift}$ iff

2. if $\bigcirc\chi\psi \in \Phi$, then $\bigcirc\chi^{\mathrm{s}}\psi \in \Theta$ and $\bigcirc\chi^{\mathrm{end}}\psi \notin \Theta$;
3. if $\bigcirc\chi^{\mathrm{s}}\psi \in \Phi$, then also $\psi \in \Phi$, $\bigcirc\chi^{\mathrm{end}}\psi \in \Phi$, and $\bigcirc\chi^{\mathrm{end}}\psi \notin \Theta$;

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

4. if $\bigcirc\chi^{\mathrm{s}}\psi \in \Theta$, then $\bigcirc\chi^{\mathrm{s}}\psi \in \Psi$;
5. if $\bigcirc\chi^{\mathrm{s}}\psi \in \Phi$, then also $\psi \in \Phi$, $\bigcirc\chi^{\mathrm{end}}\psi \in \Phi$, and $\bigcirc\chi^{\mathrm{end}}\psi \notin \Psi$.

In order to better explain how the above rules allow an automaton to check the $\bigcirc\chi$ operator, we will refer to an example automaton built to recognize words that satisfy formula $\bigcirc\chi\mathbf{a}$, with respect to the OP matrix of Figure 4.14. For the sake of simplicity, Figure 4.15 shows only part of this automaton. Figure 4.16 shows an accepting run of the automaton, while Figure 4.17 shows a rejecting one.

In general, the automaton reads subsets of AP with *push* and *shift* transitions. Due to rule 1 from the previous section, when the automaton is in state $\Phi \in \mathrm{Atoms}(\varphi)$, it can only read symbols containing all and only the atomic propositions in Φ: this is why *push* transition arcs in Figure 4.15 are labeled with all and only the atomic

	a	**b**	**c**
a	$\gtrdot$	$\lessdot$	$\lessdot$
b	$\gtrdot$	$\gtrdot$	$\lessdot$
c	$\gtrdot$	$\gtrdot$	$\lessdot$

Figure 4.14: OP Matrix that will be used in the examples of Section 4.6.1.

propositions in the source state. Also, we use the convention stated in Section 4.2.1, so the precedence relations are inferred from the structural labels typeset in boldface, and each state must contain exactly one of them.

To model-check the $\bigcirc\chi$ operator, the automaton proceeds in the following way: suppose it is in state $\Phi \in \mathrm{Atoms}(\varphi)$, with $\bigcirc\chi\psi \in \Phi$, and let $a \in \mathscr{P}(AP)$ be the terminal symbol that is about to be read. Then, the auxiliary operator $\bigcirc\chi^{\mathrm{s}}\psi$ is inserted into the next state thanks to constraint (1) (cf. step 1-2 of Figure 4.16). If a had been read by a *shift* transition, constraint (2) would have ensured this. If the symbol a is in the $\lessdot$ relation with the next one, the latter triggers a *push* transition that stores the next state, containing $\bigcirc\chi^{\mathrm{s}}\psi$, on top of the stack. Then the automaton proceeds normally, until the last symbol b of the context of the innermost chain starting in a is reached. This symbol triggers a *pop* transition that pops the previously stored state containing $\bigcirc\chi^{\mathrm{s}}\psi$ from the stack and, thanks to constraint (4), forces it into the subsequent automaton state (cf. step 3-4 of Figure 4.16). Then, if a *push* transition occurs, it means the current chain is not maximal: the transition simply stores the state, again containing $\bigcirc\chi^{\mathrm{s}}\psi$, onto the stack, and the process is repeated. If, instead, the next transition is a *shift* or a *pop*, then the chain ending in b is maximal, and ψ must be enforced in the state that will hold when b is processed: this is ensured by constraints (3) and (5) (cf. step 6-7 of Figure 4.16, and the last step of Figure 4.17, where this rules cause string rejection). These two rules also enforce $\bigcirc\chi^{\mathrm{end}}\psi$ in the current state, marking the satisfaction of $\bigcirc\chi\psi$: this way, (1) and (2) also block computations in which a chain does not start in a, but ψ holds into the next symbol.

Of course, the auxiliary operators $\bigcirc\chi^{\mathrm{s}}$ and $\bigcirc\chi^{\mathrm{end}}$ must be excluded from the constraint that enforces containment of the previous state in the next one in *pop* transitions, because once the appropriate maximal chain is reached, it must not be propagated further.

As for the final states, note that an accepting run of the operator precedence automaton must end with the stack containing the initial symbol only: this already ensures that the requirements of the $\bigcirc\chi$ operators are satisfied, because the stack symbols containing their auxiliary operators must all be popped from stack. The only exception is the last stack symbol before the initial one: the transition that pops it is not followed by another *pop* or *shift* transition, which would block the computation if the requirements of any $\bigcirc\chi$ operator had not been satisfied. Therefore, for any formula $\bigcirc\chi\psi \in \mathrm{Cl}(\varphi)$, we must exclude from F all atoms containing $\bigcirc\chi^{\mathrm{s}}\psi$ or $\bigcirc\chi\psi$.

Matching-Back ($\ominus_\chi$) Operator

This time, only one auxiliary symbol is needed: if $\ominus_\chi\psi \in \mathrm{Cl}(\varphi)$, then $\psi, \ominus^{\mathrm{s}}_\chi\psi \in \mathrm{Cl}(\varphi)$. For any $\Phi, \Theta, \Psi \in \mathrm{Atoms}(\varphi)$ and $a \in \mathscr{P}(AP)$, $(\Phi, a, \Theta) \in \delta_{push}$ and $(\Phi, a, \Theta) \in \delta_{shift}$ iff

1. if $\ominus_\chi\psi \in \Phi$, then $\ominus^{\mathrm{s}}_\chi\psi \in \Phi$; and

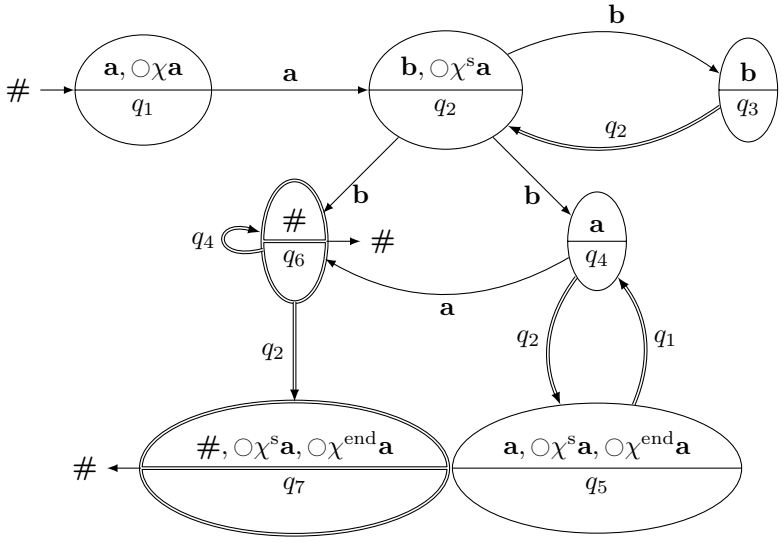


Figure 4.15: Partial representation of the automaton built to recognize words satisfying formula $\bigcirc\chi\mathbf{a}$, with respect to the OP matrix of Figure 4.14. Since its large size would make the whole automaton impractical to be represented, we only show the states and transitions involved in the runs shown in Figures 4.16 and 4.17. Note that the symbols read by the automaton are subsets of AP, but when they are singletons, we represent them with the only proposition they contain.

step	input	state	stack
1	$\mathbf{a} \lessdot \mathbf{b} \gtrdot \mathbf{b} \gtrdot \mathbf{a} \gtrdot \#$	$\{\mathbf{a}, \bigcirc\chi\mathbf{a}\}$	$\bot$
2	$\mathbf{b} \gtrdot \mathbf{b} \gtrdot \mathbf{a} \gtrdot \#$	$\{\mathbf{b}, \bigcirc\chi^{\mathrm{s}}\mathbf{a}\}$	$[\mathbf{a}, \{\mathbf{a}, \bigcirc\chi\mathbf{a}\}]\bot$
3	$\mathbf{b} \gtrdot \mathbf{a} \gtrdot \#$	$\{\mathbf{b}\}$	$[\mathbf{b}, \{\mathbf{b}, \bigcirc\chi^{\mathrm{s}}\mathbf{a}\}][\mathbf{a}, \{\mathbf{a}, \bigcirc\chi\mathbf{a}\}]\bot$
4	$\mathbf{b} \gtrdot \mathbf{a} \gtrdot \#$	$\{\mathbf{b}, \bigcirc\chi^{\mathrm{s}}\mathbf{a}\}$	$[\mathbf{a}, \{\mathbf{a}, \bigcirc\chi\mathbf{a}\}]\bot$
5	$\mathbf{a} \gtrdot \#$	$\{\mathbf{a}\}$	$[\mathbf{b}, \{\mathbf{b}, \bigcirc\chi^{\mathrm{s}}\mathbf{a}\}][\mathbf{a}, \{\mathbf{a}, \bigcirc\chi\mathbf{a}\}]\bot$
6	$\mathbf{a} \gtrdot \#$	$\{\mathbf{a}, \bigcirc\chi^{\mathrm{s}}\mathbf{a}, \bigcirc\chi^{\mathrm{end}}\mathbf{a}\}$	$[\mathbf{a}, \{\mathbf{a}, \bigcirc\chi\mathbf{a}\}]\bot$
7	$\mathbf{a} \gtrdot \#$	$\{\mathbf{a}\}$	$\bot$
8	$\#$	$\{\#\}$	$[\mathbf{a}, \{\mathbf{a}\}]\bot$
9	$\#$	$\{\#\}$	$\bot$

Figure 4.16: A computation of the automaton built for formula $\bigcirc\chi\mathbf{a}$ accepting string "**abba**".

step	input	state	stack
1	$\mathbf{a} \lessdot \mathbf{b} \gtrdot \mathbf{b} \gtrdot \#$	$\{\mathbf{a}, \bigcirc\chi\mathbf{a}\}$	$\bot$
2	$\mathbf{b} \gtrdot \mathbf{b} \gtrdot \#$	$\{\mathbf{b}, \bigcirc\chi^{\mathrm{s}}\mathbf{a}\}$	$[\mathbf{a}, \{\mathbf{a}, \bigcirc\chi\mathbf{a}\}]\bot$
3	$\mathbf{b} \gtrdot \#$	$\{\mathbf{b}\}$	$[\mathbf{b}, \{\mathbf{b}, \bigcirc\chi^{\mathrm{s}}\mathbf{a}\}][\mathbf{a}, \{\mathbf{a}, \bigcirc\chi\mathbf{a}\}]\bot$
4	$\mathbf{b} \gtrdot \#$	$\{\mathbf{b}, \bigcirc\chi^{\mathrm{s}}\mathbf{a}\}$	$[\mathbf{a}, \{\mathbf{a}, \bigcirc\chi\mathbf{a}\}]\bot$
5	$\#$	$\{\#\}$	$[\mathbf{b}, \{\mathbf{b}, \bigcirc\chi^{\mathrm{s}}\mathbf{a}\}][\mathbf{a}, \{\mathbf{a}, \bigcirc\chi\mathbf{a}\}]\bot$
6	$\#$	$\{\#, \bigcirc\chi^{\mathrm{s}}\mathbf{a}, \bigcirc\chi^{\mathrm{end}}\mathbf{a}\}$	$[\mathbf{a}, \{\mathbf{a}, \bigcirc\chi\mathbf{a}\}]\bot$

Figure 4.17: A non-accepting computation of the automaton for formula $\bigcirc\chi\mathbf{a}$ reading string "**abb**". The run cannot proceed further because no pop transition can be performed due to constraint (5), and the stack cannot be emptied.

2. if $\ominus^{\mathrm{s}}_{\chi}\psi \in \Theta$, then $\psi \in \Phi$, unless $\ominus_{\chi}\psi \in \Theta$;

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

3. if $\ominus^{\mathrm{s}}_{\chi}\psi \in \Psi$, then $\ominus^{\mathrm{s}}_{\chi}\psi \in \Theta$.

Also, if $\Phi \in I$, then $\ominus_{\chi}\psi \notin \Phi$.

The way the $\ominus_{\chi}$ operator is processed is similar to that of the $\bigcirc\chi$ operators, except that it works backwards. If $\ominus_{\chi}\psi$ holds in the automaton's state Φ right before a symbol $b \in \mathscr{P}(AP)$ is consumed by a *shift* or *push* transition, then the auxiliary operator $\ominus^{\mathrm{s}}_{\chi}\psi$ is enforced in Φ. If such transition is preceded by a *pop* transition, then at least a chain ends in b, and the maximal chain is the one related to this last *pop* transition: constraint (3) stores $\ominus^{\mathrm{s}}_{\chi}\psi$ into the state popped from stack. This latter state must have been put into the stack by a *push* transition: if this transition is preceded by another *pop* transition, then the automaton is ahead of the beginning symbol $a \in \mathscr{P}(AP)$ of the maximal chain ending in b (there must be another chain in between), and $\ominus_{\chi}\psi$ is again enforced into the previous stack symbol by such *pop* transition. If the *push* transition is preceded by a *shift* or another *push*, then the automaton has just reached a, and ψ must be enforced there (constraint (2)). This last constraint requires that $\ominus_{\chi}\psi$ does not hold in the state where the *shift*/*push* transition brings the automaton, because otherwise $\ominus_{\chi}\psi$ would be satisfied even if ψ held in a word position and $\ominus_{\chi}\psi$ in the consecutive one, which is wrong because there would not be any chain between the two positions.

Path Operators

If formula φ contains any until or since operator, the automaton must be generated with the following additional rules.

Operator Precedence Summary Until. For any $\Phi \in \mathrm{Atoms}(\varphi)$ and $O \subseteq \{\lessdot, \doteq, \gtrdot\}$, if $\psi\,\mathcal{U}^{O}\,\theta \in \Phi$ then $\psi, \theta, \bigcirc\chi(\psi\,\mathcal{U}^{O}\,\theta), \bigcirc(\psi\,\mathcal{U}^{O}\,\theta) \in \mathrm{Cl}(\varphi)$. Also, $\psi\,\mathcal{U}^{O}\,\theta \in \Phi$ iff at least one of the following holds:

1. $\theta \in \Phi$,
2. $\psi \in \Phi$ and $\bigcirc\chi(\psi\,\mathcal{U}^{O}\,\theta) \in \Phi$, or
3. $\psi \in \Phi$ and $\bigcirc(\psi\,\mathcal{U}^{O}\,\theta) \in \Phi$.

If (1) holds, then $\psi\,\mathcal{U}^{O}\,\theta$ is trivially true; if (2) holds, the path skips the body of a chain starting in the current position, where ψ holds; if (3) holds, then ψ is true in the current position and the path continues in the next one. In the latter case, we must make sure the path is followed only if the current position and the next one are in one of the precedence relations contained in O. The following constraint must be added to the definition of the transition relation: if only (3) holds in a state Φ and $\psi\,\mathcal{U}^{O}\,\theta \in \Phi$, then for any $a, b \in \mathscr{P}(AP)$ and $\Theta \in \mathrm{Atoms}(\varphi)$, with $b = \Theta \cap AP$, we have $(\Phi, a, \Theta) \in \delta_{push} \cup \delta_{shift}$ only if $a \odot b$ and $\odot \in O$. We need not impose constraints on pop transitions because they do not consume input symbols, and they preserve the same subset of AP contained in the starting state.

Furthermore, for any $\Phi \in \mathrm{Atoms}(\varphi)$, if $\Phi \in F$ then $\psi\,\mathcal{U}^{O}\,\theta \notin \Phi$, unless $\theta \in \Phi$.

Operator Precedence Summary Since. For any $\Phi \in \mathrm{Atoms}(\varphi)$ and $O \subseteq \{\lessdot, \doteq, \gtrdot\}$, if $\psi \mathcal{S}^O \theta \in \Phi$ then $\psi, \theta, \ominus_\chi(\psi \mathcal{S}^O \theta), \ominus(\psi \mathcal{S}^O \theta) \in \mathrm{Cl}(\varphi)$. Moreover, $\psi \mathcal{S}^O \theta \in \Phi$ iff at least one of the following constraints on Φ holds:

1. $\theta \in \Phi$,
2. $\psi \in \Phi$ and $\ominus_\chi(\psi \mathcal{S}^O \theta) \in \Phi$, or
3. $\psi \in \Phi$ and $\ominus(\psi \mathcal{S}^O \theta) \in \Phi$.

If only (3) holds in a state $\Phi \in \mathrm{Atoms}(\varphi)$ and $\psi \mathcal{S}^O \theta \in \Phi$, then for any $a, b \in \mathscr{P}(AP)$ and $\Theta \in \mathrm{Atoms}(\varphi)$, with $b = \Phi \cap AP$, we have $(\Theta, a, \Phi) \in \delta_{push} \cup \delta_{shift}$ only if $a \odot b$ and $\odot \in O$.

Linear Until. The constraints for the linear until operator are those required by usual LTL model checking. In more details, let $\Phi \in \mathrm{Atoms}(\varphi)$, with $\psi \mathcal{U} \theta \in \Phi$. Then $\psi, \theta, \bigcirc(\psi \mathcal{U} \theta) \in \mathrm{Cl}(\varphi)$, and either $\theta \in \Phi$, or $\psi \in \theta$ and $\bigcirc(\psi \mathcal{U} \theta) \in \Phi$. Also, no final state can contain formula $\psi \mathcal{U} \theta$.

Linear Since. Similarly, let $\Phi \in \mathrm{Atoms}(\varphi)$, with $\psi \mathcal{S} \theta \in \Phi$. Then $\psi, \theta, \ominus(\psi \mathcal{S} \theta) \in \mathrm{Cl}(\varphi)$, and either $\theta \in \Phi$, or $\psi \in \theta$ and $\ominus(\psi \mathcal{S} \theta) \in \Phi$.

Yield-Precedence Hierarchical Until. Let $\Phi \in \mathrm{Atoms}(\varphi)$: if $\psi \mathcal{U}^\uparrow \theta \in \Phi$ we introduce the auxiliary operators $\mathcal{U}^\uparrow_{\mathrm{s}}$ and $\mathcal{U}^\uparrow_{\mathrm{end}}$. We add $\psi \mathcal{U}^\uparrow_{\mathrm{s}} \theta, \bigcirc(\psi \mathcal{U}^\uparrow_{\mathrm{s}} \theta), \psi \mathcal{U}^\uparrow_{\mathrm{end}} \theta \in \mathrm{Cl}(\varphi)$, and $\bigcirc(\psi \mathcal{U}^\uparrow_{\mathrm{s}} \theta) \in \Phi$. The way the automaton deals with this operator is similar to that of the $\bigcirc\chi$ operator, by exploiting the stack of the automaton in order to propagate auxiliary operators. The related constraints are enforced by the δ_{pop} transition relation: given any $\Phi, \Theta, \Psi \in \mathrm{Atoms}(\varphi)$, we have $(\Phi, \Theta, \Psi) \in \delta_{pop}$ iff $\psi \mathcal{U}^\uparrow_{\mathrm{s}} \theta \notin \Phi$ and $\psi \mathcal{U}^\uparrow_{\mathrm{end}} \theta \notin \Phi$, and if $\psi \mathcal{U}^\uparrow_{\mathrm{s}} \theta \in \Theta$, then either

- $\theta \in \Psi$ and $\psi \mathcal{U}^\uparrow_{\mathrm{end}} \theta \in \Psi$, or
- $\psi \in \Psi$ and $\psi \mathcal{U}^\uparrow_{\mathrm{s}} \theta \in \Psi$.

Moreover, for any $\Phi, \Theta \in \mathrm{Atoms}(\varphi)$ and $a \in \mathscr{P}(AP)$, if $(\Phi, a, \Theta) \in \delta_{shift}$ then $\psi \mathcal{U}^\uparrow_{\mathrm{s}} \theta, \psi \mathcal{U}^\uparrow_{\mathrm{end}} \theta \notin \Phi$; and if $(\Phi, a, \Theta) \in \delta_{push}$ then $\psi \mathcal{U}^\uparrow_{\mathrm{end}} \theta \notin \Theta$.

To better illustrate how this definition works, we will refer to the example run of Figure 4.18. Suppose $\Phi \in \mathrm{Atoms}(\varphi)$ is the state of the automaton before reading symbol $a \in \mathscr{P}(AP)$, and $\psi \mathcal{U}^\uparrow \theta \in \Phi$. As we did for $\bigcirc\chi$, $\psi \mathcal{U}^\uparrow_{\mathrm{s}} \theta$ is first stored onto the stack (step 1-2 of Figure 4.18). Then, the automaton proceeds normally, until another symbol b such that $a \lessdot b$ is reached. At this point, state Θ is popped from the stack, and the requirements of the $\psi \mathcal{U}^\uparrow \theta$ operator are enforced: if θ holds in the next state, which contains formulae that are true when b is reached, then the hierarchical until is satisfied, and $\psi \mathcal{U}^\uparrow_{\mathrm{end}} \theta$ is included into the next state in order to mark this condition (cf. step 5-6). Otherwise, ψ must hold, and $\psi \mathcal{U}^\uparrow_{\mathrm{s}} \theta$ is again included into the next state (step 3-4). Then, b is immediately read by a *push* transition, that stores the new state on top of the stack, and prevents the propagation of the $\psi \mathcal{U}^\uparrow_{\mathrm{end}} \theta$ symbol, if present. Once another symbol that yields precedence to a is reached, the process is repeated, unless θ held in b. When the outermost chain having a as the left part of the context is reached, the *pop* transition described above is followed by either another *pop* transition of a symbol c such that $a \gtrdot c$, or a *shift* transition if $a \doteq c$.

step	input	state	stack
1	$\mathbf{a} \lessdot \mathbf{b} \gtrdot \{\mathbf{b}, \mathrm{r}\} \gtrdot \{\mathbf{b}, \mathrm{s}\} \gtrdot \#$	$q_0 = \{\mathbf{a}, \mathrm{r}\, \mathcal{U}^{\uparrow}\, \mathrm{s}, \bigcirc(\mathrm{r}\, \mathcal{U}^{\uparrow}_{\mathrm{s}}\, \mathrm{s})\}$	$\bot$
2	$\mathbf{b} \gtrdot \{\mathbf{b}, \mathrm{r}\} \gtrdot \{\mathbf{b}, \mathrm{s}\} \gtrdot \#$	$q_1 = \{\mathbf{b}, \mathrm{r}\, \mathcal{U}^{\uparrow}_{\mathrm{s}}\, \mathrm{s}\}$	$[\mathbf{a}, q_0]\bot$
3	$\{\mathbf{b}, \mathrm{r}\} \gtrdot \{\mathbf{b}, \mathrm{s}\} \gtrdot \#$	$\{\mathbf{b}, \mathrm{r}\}$	$[\mathbf{b}, q_1][\mathbf{a}, q_0]\bot$
4	$\{\mathbf{b}, \mathrm{r}\} \gtrdot \{\mathbf{b}, \mathrm{s}\} \gtrdot \#$	$q_2 = \{\mathbf{b}, \mathrm{r}, \mathrm{r}\, \mathcal{U}^{\uparrow}_{\mathrm{s}}\, \mathrm{s}\}$	$[\mathbf{a}, q_0]\bot$
5	$\{\mathbf{b}, \mathrm{s}\} \gtrdot \#$	$\{\mathbf{b}, \mathrm{s}\}$	$[\{\mathbf{b}, \mathrm{r}\}, q_2][\mathbf{a}, q_0]\bot$
6	$\{\mathbf{b}, \mathrm{s}\} \gtrdot \#$	$q_3 = \{\mathbf{b}, \mathrm{s}, \mathrm{r}\, \mathcal{U}^{\uparrow}_{\mathrm{end}}\, \mathrm{s}\}$	$[\mathbf{a}, q_0]\bot$
7	$\#$	$\{\#\}$	$[\{\mathbf{b}, \mathrm{s}\}, q_3][\mathbf{a}, q_0]\bot$
8	$\#$	$\{\#\}$	$[\mathbf{a}, q_0]\bot$
9	$\#$	$\{\#\}$	$\bot$

Figure 4.18: A computation of the automaton for formula $\mathrm{r}\, \mathcal{U}^{\uparrow}\, \mathrm{s}$ accepting string "$\mathbf{ab}\{\mathbf{b}, \mathrm{r}\}\{\mathbf{b}, \mathrm{s}\}$". Again, we represent singleton subsets of AP with the only element they contain.

Position c must not be considered part of the hierarchical path: in order to enforce this requirement, we impose that no *pop* or *shift* transition can occur if either $\psi\, \mathcal{U}^{\uparrow}_{\mathrm{s}}\, \theta$ or $\psi\, \mathcal{U}^{\uparrow}_{\mathrm{end}}\, \theta$ hold in the current state, thus blocking the computation.

Again, the fact that an accepting run must end with an empty stack ensures that all requirements are satisfied, except those related to the last symbol popped from stack. For this reason, states containing $\psi \mathcal{U}^{\uparrow}_{\mathrm{s}}\, \theta$ or $\psi \mathcal{U}^{\uparrow}_{\mathrm{end}}\, \theta$ are excluded from the final set F.

Yield-Precedence Hierarchical Since. If $\psi\, \mathcal{S}^{\downarrow}\, \theta \in \Phi$, with $\Phi \in \mathrm{Atoms}(\varphi)$, then $\psi\, \mathcal{S}^{\downarrow}_{\mathrm{s}}\, \theta, \psi\, \mathcal{S}^{\downarrow}_{\mathrm{end}}\, \theta, \bigcirc\chi(\psi\, \mathcal{S}^{\downarrow}_{\mathrm{end}}\, \theta) \in \mathrm{Cl}(\varphi)$. Also, $\bigcirc\chi(\psi\, \mathcal{S}^{\downarrow}_{\mathrm{end}}\, \theta) \in \Phi$ must hold, and the following constraints on the transition relation must be enforced: let $(\Phi, \Theta, \Psi) \in \delta_{pop}$, with $\Phi, \Theta, \Psi \in \mathrm{Atoms}(\varphi)$;

1. if $\psi\, \mathcal{S}^{\downarrow}_{\mathrm{end}}\, \theta \in \Psi$, then $\psi\, \mathcal{S}^{\downarrow}_{\mathrm{s}}\, \theta \in \Theta$;
2. if $\psi\, \mathcal{S}^{\downarrow}_{\mathrm{s}}\, \theta \in \Psi$ and $\theta \notin \Psi$, then $\psi \in \Psi$ and $\psi\, \mathcal{S}^{\downarrow}_{\mathrm{s}}\, \theta \in \Theta$.

For any $\Phi, \Theta \in \mathrm{Atoms}(\varphi)$ and $a \in \mathscr{P}(AP)$, if $(\Phi, a, \Theta) \in \delta_{shift}$ or $(\Phi, a, \Theta) \in \delta_{push}$ then $\psi\, \mathcal{S}^{\downarrow}_{\mathrm{s}}\, \theta \notin \Theta$.

The rationale behind this definition is similar to that of the $\ominus_{\chi}$ operator. Suppose $\psi\, \mathcal{S}^{\downarrow}\, \theta$ holds in a state $\Phi \in \mathrm{Atoms}(\varphi)$, associated to the terminal symbol $a \in \mathscr{P}(\varphi)$. Then the auxiliary operator $\psi\, \mathcal{S}^{\downarrow}_{\mathrm{end}}\, \theta$ is enforced through a $\bigcirc\chi$ operator into the state that holds right after the stack symbol related to the outermost chain starting with a is popped. Constraint (1) then makes sure the other auxiliary operator $\psi\, \mathcal{S}^{\downarrow}_{\mathrm{s}}\, \theta$ is stored into the stack symbol mentioned above. Every time the stack symbol related to a position that is part of the yield-precedence hierarchical path is popped from stack, constraint (2) ensures that either θ holds in the next position in the path, which ends there, or ψ holds in there, along with $\psi\, \mathcal{S}^{\downarrow}_{\mathrm{s}}\, \theta$, which ensures the prosecution of the path. The last constraint on *push* and *shift* transitions prevents the first position right after a from being considered part of the path.

Note that the acceptance conditions of $\bigcirc\chi(\psi\, \mathcal{S}^{\downarrow}_{\mathrm{end}}\, \theta)$ already ensure the satisfaction of formula $\psi\, \mathcal{S}^{\downarrow}\, \theta$ before the end of the string.

Take-Precedence Hierarchical Until. Let $\Phi \in \mathrm{Atoms}(\varphi)$, with $\psi \mathcal{U}^{\downarrow} \theta \in \Phi$: then $\psi\, \mathcal{U}^{\downarrow}_{\mathrm{s}}\, \theta, \psi\, \mathcal{U}^{\downarrow}_{\mathrm{end}}\, \theta, \ominus\psi, \ominus\theta \in \mathrm{Cl}(\varphi)$. For any $\Phi, \Theta \in \mathrm{Atoms}(\varphi)$ and $a \in \mathscr{P}(AP)$, if $(\Phi, a, \Theta) \in \delta_{shift}$ or $(\Phi, a, \Theta) \in \delta_{push}$ then

step	input	state	stack
1	$\{\mathbf{c},\mathrm{r}\} \lessdot \{\mathbf{c},\mathrm{s}\} \lessdot \mathbf{c} \gtrdot \mathbf{a} \gtrdot \#$	$q_0 = \{\mathbf{c},\mathrm{r},\bigcirc\chi(\mathrm{r}\,\mathcal{U}^{\downarrow}\,\mathrm{s})\}$	$\bot$
2	$\{\mathbf{c},\mathrm{s}\} \lessdot \mathbf{c} \gtrdot \mathbf{a} \gtrdot \#$	$q_1 = \{\mathbf{c},\mathrm{s},\bigcirc\chi^{\mathrm{s}}(\mathrm{r}\,\mathcal{U}^{\downarrow}_{\mathrm{s}}\,\mathrm{s}),\ominus\mathrm{r}\}$	$[\{\mathbf{c},\mathrm{r}\},q_0]\bot$
3	$\mathbf{c} \gtrdot \mathbf{a} \gtrdot \#$	$q_2 = \{\mathbf{c},\ominus\mathrm{s}\}$	$[\{\mathbf{c},\mathrm{s}\},q_1][\{\mathbf{c},\mathrm{r}\},q_0]\bot$
4	$\mathbf{a} \gtrdot \#$	$\{\mathbf{a},\mathrm{r}\,\mathcal{U}^{\downarrow}\,\mathrm{s}\}$	$[\mathbf{c},q_2][\{\mathbf{c},\mathrm{s}\},q_1][\{\mathbf{c},\mathrm{r}\},q_0]\bot$
5	$\mathbf{a} \gtrdot \#$	$\{\mathbf{a},\mathrm{r}\,\mathcal{U}^{\downarrow}\,\mathrm{s},\mathrm{r}\,\mathcal{U}^{\downarrow}_{\mathrm{s}}\,\mathrm{s}\}$	$[\{\mathbf{c},\mathrm{s}\},q_1][\{\mathbf{c},\mathrm{r}\},q_0]\bot$
6	$\mathbf{a} \gtrdot \#$	$\left\{\begin{array}{r}\mathbf{a},\mathrm{r}\,\mathcal{U}^{\downarrow}\,\mathrm{s},\mathrm{r}\,\mathcal{U}^{\downarrow}_{\mathrm{s}}\,\mathrm{s},\\ \bigcirc\chi^{\mathrm{s}}(\mathrm{r}\,\mathcal{U}^{\downarrow}\,\mathrm{s}),\\ \bigcirc\chi^{\mathrm{end}}(\mathrm{r}\,\mathcal{U}^{\downarrow}\,\mathrm{s})\end{array}\right\}$	$[\{\mathbf{c},\mathrm{r}\},q_0]\bot$
7	$\mathbf{a} \gtrdot \#$	$q_3 = \{\mathbf{a},\mathrm{r}\,\mathcal{U}^{\downarrow}\,\mathrm{s},\mathrm{r}\,\mathcal{U}^{\downarrow}_{\mathrm{end}}\,\mathrm{s}\}$	$\bot$
8	$\#$	$\{\#\}$	$[\mathbf{a},q_3]\bot$
9	$\#$	$\{\#\}\,\bot$	

Figure 4.19: A run of the automaton for formula $\bigcirc\chi(\mathrm{r}\ \mathcal{U}^{\downarrow}\ \mathrm{s})$ accepting string "$\{\mathbf{c},\mathrm{r}\}\{\mathbf{c},\mathrm{s}\}\mathbf{ca}$".

1. $\psi\,\mathcal{U}^{\downarrow}_{\mathrm{s}}\,\theta \notin \Theta$, and
2. if $\psi\,\mathcal{U}^{\downarrow}\,\theta \in \Phi$ then $\psi\,\mathcal{U}^{\downarrow}_{\mathrm{end}}\,\theta \in \Phi$.

Moreover, for any $\Phi,\Theta,\Psi \in \mathrm{Atoms}(\varphi)$, if $(\Phi,\Theta,\Psi) \in \delta_{pop}$ then

3. if $\psi\,\mathcal{U}^{\downarrow}_{\mathrm{end}}\,\theta \in \Psi$ then $\psi\,\mathcal{U}^{\downarrow}_{\mathrm{s}}\,\theta \in \Phi$;
4. if $\psi\,\mathcal{U}^{\downarrow}_{\mathrm{s}}\,\theta \in \Psi$ then either $\ominus\theta \in \Theta$, or $\ominus\psi \in \Theta$ and $\psi\,\mathcal{U}^{\downarrow}_{\mathrm{s}}\,\theta \in \Phi$.

Of course, if $\psi\,\mathcal{U}^{\downarrow}\,\theta \in \Phi$ then $\Phi \notin I$.

Figure 4.19 shows an example accepting run for a formula containing this operator. Let $\Phi \in \mathrm{Atoms}(\varphi)$ be the state of the automaton just before processing $a \in \mathscr{P}(AP)$. The consistency of $\psi\mathcal{U}^{\downarrow}\theta \in \Phi$ is checked during the *pop* transitions that remove from the automaton's stack the symbols related to all chains ending with a. First, constraint (2) introduces the auxiliary operator $\psi\mathcal{U}^{\downarrow}_{\mathrm{end}}\,\theta$ into the state of the automaton after the last *pop* transition (cf. step 6-7 of Figure 4.19). Then, this last transition introduces $\psi\ \mathcal{U}^{\downarrow}_{\mathrm{s}}\ \theta$ into the previous state (constraint (3)): this way the position in which the outermost chain begins is not affected, as it is not part of the take-precedence path (it does not take precedence from a). Thanks to constraint (4), all subsequent *pop* transitions must either enforce θ in the position b in which the chain starts, by having $\ominus\theta$ into the popped stack symbol, and letting the path end (cf. step 3-4), or they enforce ψ in b, and let the path continue by leaving $\psi\,\mathcal{U}^{\downarrow}_{\mathrm{s}}\,\theta$ in the previous state (cf. step 2-3). In order to disallow paths that do not end with a position in which θ holds, $\psi\mathcal{U}^{\downarrow}_{\mathrm{s}}\,\theta$ cannot be the final state of any *push* or *shift* transition, and in particular of the one the consumes the terminal symbol right before a, which is not part of the path.

Take-Precedence Hierarchical Since. Let $\Phi \in \mathrm{Atoms}(\varphi)$ be a state in which $\psi\ \mathcal{S}^{\uparrow}\ \theta$ holds. Then we add $\psi\ \mathcal{S}^{\uparrow}_{\mathrm{s}}\ \theta, \psi\ \mathcal{S}^{\uparrow}_{\mathrm{end}}\ \theta \in \mathrm{Cl}(\varphi)$, and we impose the following constraints on the transition function. For any $\Phi,\Theta \in \mathrm{Atoms}(\varphi)$, and $a \in \mathscr{P}(AP)$, if $(\Phi,a,\Theta) \in \delta_{push}$ or $(\Phi,a,\Theta) \in \delta_{shift}$ then

1. if $\psi\,\mathcal{S}^{\uparrow}\,\theta \in \Theta$, then $\psi\,\mathcal{S}^{\uparrow}_{\mathrm{s}}\,\theta \in \Theta$;
2. $\psi\,\mathcal{S}^{\uparrow}_{\mathrm{s}}\,\theta, \psi\,\mathcal{S}^{\uparrow}_{\mathrm{end}}\,\theta \notin \Phi$.

For any $\Phi, \Theta, \Psi \in \mathrm{Atoms}(\varphi)$, if $(\Phi, \Theta, \Psi) \in \delta_{pop}$ we have

3. if $\psi\, \mathcal{S}^{\uparrow}\, \theta \in \Psi$ then $\psi\, \mathcal{S}^{\uparrow}\, \theta \in \Phi$;
4. if $\psi\, \mathcal{S}^{\uparrow}_{\mathrm{s}}\, \theta \in \Phi$ and $\ominus\theta \in \Theta$, then $\psi\, \mathcal{S}^{\uparrow}_{\mathrm{end}}\, \theta \in \Psi$;
5. if $\psi\, \mathcal{S}^{\uparrow}_{\mathrm{s}}\, \theta \in \Phi$ and $\ominus\theta \notin \Theta$ then $\ominus\psi \in \Theta$ and $\psi\, \mathcal{S}^{\uparrow}_{\mathrm{s}}\, \theta \in \Psi$.

Finally, if $\psi\, \mathcal{S}^{\uparrow}\, \theta \in \Phi$, then $\Phi \notin I$.

The constraints for this operator work in a way similar to those of the $\mathcal{U}^{\downarrow}$ operator, except for the fact that here the auxiliary operator $\mathcal{S}^{\uparrow}_{\mathrm{end}}$ is used to mark the chain in which the take-precedence hierarchical path ends with θ, thus preventing it from being the outermost chain with constraint (2). Moreover, constraint (3) forces formula $\psi\, \mathcal{S}^{\uparrow}\, \theta \in \Psi$ to hold since the first *pop* transition, so that (1) always puts $\psi\, \mathcal{S}^{\uparrow}_{\mathrm{s}}\, \theta$ into the state of the automaton just before such transition.

Complexity

The construction procedure described above yields the following result:

Theorem 31. *For any OPTL formula φ, it is possible to build an OPA of size $2^{O(|\varphi|)}$ that accepts models satisfying φ.*

4.6.2 Model-Checking for Infinite Words

Model-checking of OPTL can be performed on ω-words by constructing a generalized nondeterministic Operator Precedence Automaton (cf. ωOPBA, [LMPP15]) $\mathcal{A}_\omega = \langle \mathscr{P}(AP), M_{\mathscr{P}(AP)}, \mathrm{Atoms}(\varphi), I, \mathbf{F}, \delta \rangle$, where all components are the same as those of the finite-word counterpart $\mathcal{A}$, except for $\mathbf{F}$, which is a set of sets of Büchi-final states. An ω-word is accepted by the automaton if and only if at least one state from each one of the sets in $\mathbf{F}$ appears infinitely often in a run. The way the automaton enforces the requisites entailed by temporal operators is the same as described for the finite-word case: the only differences are in fact the acceptance conditions, and we will only illustrate the modifications needed in order to adapt a finite-word automaton to use Büchi acceptance conditions.

The main problem that arises when doing so, is the fact that ωOPBA acceptance conditions only depend on the automaton's infinitely recurring states, and say nothing about the stack, which may never be empty. This forces us to introduce new auxiliary operators that keep track of the requirements of temporal operators such as the matching-next and yield-precedence hierarchical until, because the previously introduced auxiliary operators may remain buried into the stack, never being popped and having a chance to enforce the requirements they hold.

Matching-Next ($\bigcirc\chi$) Operator

In order to define a Büchi-acceptance condition for this operator, we need to introduce the auxiliary operators $\bigcirc\chi^{\mathrm{p}}$ and $\bigcirc\chi^{\omega\mathrm{end}}$. For any formula $\bigcirc\chi\psi \in \mathrm{Cl}(\varphi)$, we add $\bigcirc\chi^{\mathrm{p}}\psi, \bigcirc\chi^{\omega\mathrm{end}}\psi \in \mathrm{Cl}(\varphi)$. The $\bigcirc\chi^{\mathrm{p}}$ operator will be inserted into the next automaton's state whenever $\bigcirc\chi\psi$ holds, and it will continue being propagated into subsequent states until the outermost occurrence of $\bigcirc\chi\psi$ is satisfied, which implies all pending occurrences of $\bigcirc\chi\psi$ have also been satisfied. To mark this event, $\bigcirc\chi^{\omega\mathrm{end}}\psi$ is also inserted into the current state. This is accomplished by adding the following

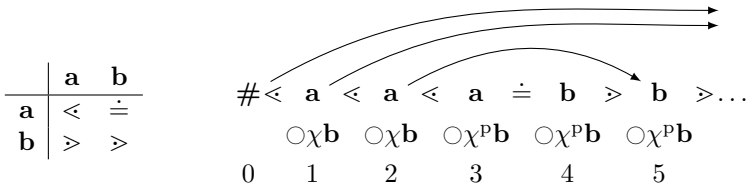


Figure 4.20: An example of OP ω-word with never-ending chains.

constraints on the transition relation: for any $\Phi, \Theta, \Psi \in \mathrm{Atoms}(\varphi)$ and $a \in \mathscr{P}(AP)$, $(\Phi, a, \Theta) \in \delta_{push}$ and $(\Phi, a, \Theta) \in \delta_{shift}$ iff

1. if $\bigcirc\chi\psi \in \Phi$, then $\bigcirc(\bigcirc\chi^{\mathrm{p}}\psi \in \Theta)$;
2. if $\bigcirc\chi^{\mathrm{p}}\psi \in \Phi$, then $\bigcirc\chi^{\mathrm{p}}\psi \in \Theta$;

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

3. $\bigcirc\chi^{\omega\mathrm{end}}\psi \in \Psi$ iff $\bigcirc\chi^{\mathrm{end}}\psi \in \Psi$ and $\bigcirc\chi^{\mathrm{p}}\psi \notin \Theta$;
4. if $\bigcirc\chi^{\mathrm{p}}\psi \in \Phi$, then $\bigcirc\chi^{\mathrm{p}}\psi \in \Psi$, unless $\bigcirc\chi^{\omega\mathrm{end}}\psi \in \Psi$.

Also, if $(\Phi, a, \Theta) \in \delta_{push}$ then $\bigcirc\chi^{\omega\mathrm{end}}\psi \notin \Phi$ (so we are sure $\bigcirc\chi^{\omega\mathrm{end}}\psi$ only appears in maximal chains).

In this way, the presence of $\bigcirc\chi^{\mathrm{p}}\psi$ in a state reveals the fact that a stack symbol contains $\bigcirc\chi^{\mathrm{s}}\psi$, and the requirement of at least a $\bigcirc\chi\psi$ formula is pending. When all such requirements are satisfied, formula $\bigcirc\chi^{\omega\mathrm{end}}\psi$ appears.

It may happen that multiple instances of the same $\bigcirc\chi\psi$ formula are pending at the same time. This is what happens in the example of Figure 4.20, where formula $\bigcirc\chi\mathbf{b}$ is required to hold in positions 1 and 2, the instance in position 1 being the outermost. In this case, when an inner instance of $\bigcirc\chi\psi$ is encountered, then $\bigcirc\chi^{\mathrm{p}}\psi$ will be already present in the automaton's state thanks to constraint (1), and it will be pushed on stack with $\bigcirc\chi^{\mathrm{s}}\psi$. Therefore, these inner instances of $\bigcirc\chi$ can be recognized because the stack symbol that carries their auxiliary operator $\bigcirc\chi^{\mathrm{s}}\psi$ also carries $\bigcirc\chi^{\mathrm{p}}\psi$. So, we insert the $\bigcirc\chi^{\omega\mathrm{end}}\psi$ operator only when $\bigcirc\chi^{\mathrm{p}}\psi$ is not present in the popped stack symbol, so the automaton does not stop propagating $\bigcirc\chi^{\mathrm{p}}\psi$ if the outermost instance of $\bigcirc\chi\psi$ has not been satisfied yet. Indeed, in the above example the instance of $\bigcirc\chi\mathbf{b}$ of position 2 is satisfied in position 5, but since the one of position 1 is not, $\bigcirc\chi^{\mathrm{p}}\mathbf{b}$ does not stop being propagated, and unless the chain starting in 1 is closed with a b, this word is rejected by the automaton, because $\bigcirc\chi^{\mathrm{p}}\mathbf{b}$ occurs in infinitely many states, and $\bigcirc\chi^{\omega\mathrm{end}}\mathbf{b}$ does not.

Therefore, we may define the set $F_{\bigcirc\chi\psi} \in \mathbf{F}$ so that for any $\Phi \in \mathrm{Atoms}(\varphi)$, $\Phi \in F_{\bigcirc\chi\psi}$ iff either $\bigcirc\chi^{\mathrm{p}}\psi \notin \Phi$ or $\bigcirc\chi^{\omega\mathrm{end}}\psi \in \Phi$. This makes sure that every $\bigcirc\chi$ operator is satisfied infinitely often, if it is required to hold infinitely often.

As for the $\ominus_\chi$ operator, the rules given for the finite case suffice to ensure its satisfaction in the infinite case, because its requirements are in the past, and must be already satisfied when it is encountered.

Path Operators

Operator Precedence Summary Until. For any formula $\psi\,\mathcal{U}^O\,\theta$ and set $O \subseteq \{\lessdot, \doteq, \gtrdot\}$, we introduce the Büchi acceptance set $F_{\psi\mathcal{U}^O\theta} \in \mathbf{F}$, such that $\Phi \in F_{\psi\mathcal{U}^O\theta}$ if either

- $\psi \, \mathcal{U}^O \, \theta \notin \Phi$ or
- $\theta \in \Phi$ and $\bigcirc\chi^{\mathrm{p}}(\psi \, \mathcal{U}^O \, \theta) \notin \Phi$.

Note that also the acceptance conditions for the $\bigcirc\chi(\psi \, \mathcal{U}^O \, \theta)$ operator are involved, and they prevent words with never-ending chains that delay the satisfaction of $\psi \mathcal{U}^O \theta$ from being mistakenly accepted.

Other Summary/Linear Operators. As for the linear until, the acceptance conditions are very similar to this last case, and they are exactly the same used in LTL model checking. Also, the since counterparts of these operators do not require Büchi acceptance conditions, because they regard the past.

Yield-Precedence Hierarchical Until. The acceptance conditions for this operator are very similar to those for the $\bigcirc\chi$ operator, since they both rely on stack symbols to propagate their requirements. We will therefore introduce the additional auxiliary operators $\mathcal{U}^{\uparrow}_{\mathrm{p}}$ and $\mathcal{U}^{\uparrow}_{\omega\mathrm{end}}$, of which the first is kept into the automaton's states until all instances of $\mathcal{U}^{\uparrow}$ with the same arguments are satisfied, marking the fact that their requirements are still pending, and the second marks the satisfaction of all such pending requirements.

So, if $\psi \, \mathcal{U}^{\uparrow} \, \theta \in \mathrm{Cl}(\varphi)$, then we add $\psi \, \mathcal{U}^{\uparrow}_{\mathrm{p}} \, \theta, \psi \, \mathcal{U}^{\uparrow}_{\omega\mathrm{end}} \, \theta, \bigcirc\bigcirc(\psi \, \mathcal{U}^{\uparrow}_{\mathrm{p}} \, \theta) \in \mathrm{Cl}(\varphi)$, and for any $\Phi \in \mathrm{Atoms}(\varphi)$, if $\psi \, \mathcal{U}^{\uparrow} \, \theta \in \Phi$ then $\bigcirc\bigcirc(\psi \, \mathcal{U}^{\uparrow}_{\mathrm{p}} \, \theta) \in \Phi$. The following constraints on the transition relation are also needed: for any $\Phi, \Theta, \Psi \in \mathrm{Atoms}(\varphi)$ and $a \in \mathscr{P}(AP)$,

- if $(\Phi, a, \Theta) \in \delta_{shift}$ and $\psi \, \mathcal{U}^{\uparrow}_{\mathrm{p}} \, \theta \in \Phi$, then $\psi \, \mathcal{U}^{\uparrow}_{\mathrm{p}} \, \theta \in \Theta$;
- if $(\Phi, a, \Theta) \in \delta_{push}$ and $\psi \, \mathcal{U}^{\uparrow}_{\mathrm{p}} \, \theta \in \Phi$, then $\psi \, \mathcal{U}^{\uparrow}_{\mathrm{p}} \, \theta \in \Theta$, unless $\psi \, \mathcal{U}^{\uparrow}_{\omega\mathrm{end}} \, \theta \in \Phi$.

Moreover, $(\Phi, \Theta, \Psi) \in \delta_{pop}$ iff

- $\psi \, \mathcal{U}^{\uparrow}_{\omega\mathrm{end}} \, \theta \in \Psi$ iff $\psi \, \mathcal{U}^{\uparrow}_{\mathrm{end}} \, \theta \in \Psi$ and $\psi \, \mathcal{U}^{\uparrow}_{\mathrm{p}} \, \theta \notin \Theta$;
- if $\psi \, \mathcal{U}^{\uparrow}_{\mathrm{p}} \, \theta \in \Phi$, then $\psi \, \mathcal{U}^{\uparrow}_{\mathrm{p}} \, \theta \in \Psi$.

Additionally, we need another constraint that prevents $\psi \, \mathcal{U}^{\uparrow}_{\mathrm{end}} \, \theta$ from appearing in states where it should not: if $(\Phi, \Theta, \Psi) \in \delta_{pop}$ and $\psi \, \mathcal{U}^{\uparrow}_{\mathrm{end}} \, \theta \in \Psi$, then $\psi \, \mathcal{U}^{\uparrow}_{\mathrm{s}} \, \theta \in \Theta$ and $\theta \in \Psi$.

Again, these rules exploit the fact that $\psi \, \mathcal{U}^{\uparrow}_{\mathrm{p}} \, \theta$ is included into the stack symbols related to inner instances of $\psi \mathcal{U}^{\uparrow} \theta$, and $\psi \mathcal{U}^{\uparrow}_{\omega\mathrm{end}} \theta$ is only inserted when such auxiliary operator is not present, thus marking the states in which all instances are satisfied.

Finally, we can define the set $F_{\psi \mathcal{U}^{\uparrow} \theta} \in \mathbf{F}$ so that for any $\Phi \in F_{\psi \mathcal{U}^{\uparrow} \theta}$ either $\psi \mathcal{U}^{\uparrow}_{\mathrm{p}} \theta \notin \Phi$ or $\psi \, \mathcal{U}^{\uparrow}_{\omega\mathrm{end}} \, \theta \in \Phi$.

Other Hierarchical Operators. Regarding the yield-precedence hierarchical since ($\mathcal{S}^{\downarrow}$), the acceptance conditions related to $\bigcirc\chi(\psi \, \mathcal{S}^{\downarrow}_{\mathrm{end}} \, \theta)$ suffice to ensure its satisfaction. Moreover, no further acceptance conditions are needed for the take-precedence hierarchical operators, since their requirements must be satisfied in the past.

Complexity

The auxiliary operators added in the construction of the ωOPBA only cause a linear increase in the size of the set $\mathrm{Cl}(\varphi)$, thus leaving the overall complexity unchanged:

Theorem 32. *For any OPTL formula φ, it is possible to build an ωOPBA of size $2^{O(|\varphi|)}$ that accepts models satisfying φ.*

Chapter 5

Conclusion

In this thesis, we have introduced OPTL, a novel temporal logic formalism, and proved that it is strictly more expressive than the formalisms constituting the state of the art. OPTL allows to express properties that could not be tackled by the main preexisting temporal logics, namely LTL and the already more expressive NWTL. We started investigating its expressive power by comparing it to NWTL, and by finding an initial subset of operators that is expressively adequate. However, concerning expressive power, a crucial issue remains open: the comparison with First Order Logic. A FO-completeness result, which was previously achieved in the literature for both LTL and NWTL, will certainly be essential for the success of OPTL in the field of model checking, because it would be the assurance that OPTL is capable of expressing a significant portion of OPL-expressible properties.

After achieving such a result, the actual use of OPTL for model checking is the next natural question to be tackled. In this thesis, we gave an automata theoretic procedure similar to the classic one for LTL [VW86]. It allows to build a nondeterministic OPA of size exponential in the length of the specification, accepting only models of this specification. The determinization of such an automaton would imply a further exponential "jump" in complexity. Therefore, one of the next research steps would be to study the actual implementation of OPTL model checking. In this respect, the results obtained for OPL parallel parsing [BCRMP13], due to the peculiar properties of OPLs, are promising for the possible exploitation of parallelism for OPTL model checking.

Finally, we acknowledge that the complexity and the consequent steep learning curve of OPTL and similar temporal formalisms (such as LTL, CTL, CTL* and NWTL), could be an obstacle to their actual utilization in the industry. This problem could be tackled by introducing a more higher-level and informal interface for OPTL, requiring a significantly less sharp mathematical skill than writing a well-formed OPTL formula expressing an informal requirement.

Bibliography

[AAB+08] Rajeev Alur, Marcelo Arenas, Pablo Barcelo, Kousha Etessami, Neil Immerman, and Leonid Libkin. First-Order and Temporal Logics for Nested Words. *Logical Methods in Computer Science*, Volume 4, Issue 4, November 2008.

[AEM04] Rajeev Alur, Kousha Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 467–481, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[AKMV05] Rajeev Alur, Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Congruences for visibly pushdown languages. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Automata, Languages and Programming*, pages 1102–1114, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[AM04] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*, STOC '04, pages 202–211, New York, NY, USA, 2004. ACM.

[AM06] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. In Oscar H. Ibarra and Zhe Dang, editors, *Developments in Language Theory*, pages 1–13, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[AM09] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3):16:1–16:43, May 2009.

[BCRM+14] Alessandro Barenghi, Stefano Crespi-Reghizzi, Dino Mandrioli, Federica Panella, and Matteo Pradella. The PAPAGENO Parallel-Parser Generator. In Albert Cohen, editor, *Compiler Construction*, pages 192–196, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[BCRM+15] Alessandro Barenghi, Stefano Crespi-Reghizzi, Dino Mandrioli, Federica Panella, and Matteo Pradella. Parallel parsing made practical. *Science of Computer Programming*, 112:195–226, 2015.

[BCRMP13] Alessandro Barenghi, Stefano Crespi-Reghizzi, Dino Mandrioli, and Matteo Pradella. Parallel parsing of operator precedence grammars. *Information Processing Letters*, 113(7):245–249, 2013.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR '97:*

Concurrency Theory, pages 135–150, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[BS92] O. Burkart and B. Steffen. Model checking for context-free processes. In *CONCUR '92*, volume 630 of *LNCS*, pages 123–137. Springer Berlin Heidelberg, 1992.

[Büc62] J. Richard Büchi. On a decision method in restricted second order arithmetic. In *International Congress in Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.

[BVCR+13] Alessandro Barenghi, Ermes Viviani, Stefano Crespi-Reghizzi, Dino Mandrioli, and Matteo Pradella. PAPAGENO: A parallel parser generator for operator precedence grammars. In *Software Language Engineering*, pages 264–274, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Bü60] J. Richard Büchi. Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly*, 6(1-6):66–92, 1960.

[CMP18] M. Chiari, D. Mandrioli, and M. Pradella. Temporal Logic and Model Checking for Operator Precedence Languages. *ArXiv e-prints*, September 2018. arXiv:1809.03100.

[CP17] Stefano Crespi-Reghizzi and Matteo Pradella. Higher-order operator precedence languages. In Erzsébet Csuhaj-Varjú, Pál Dömösi, and György Vaszil, editors, *Proceedings 15th International Conference on Automata and Formal Languages, AFL 2017, Debrecen, Hungary, September 4-6, 2017.*, volume 252 of *EPTCS*, pages 86–100, 2017.

[CRM12] Stefano Crespi-Reghizzi and Dino Mandrioli. Operator precedence and the visibly pushdown property. *Journal of Computer and System Sciences*, 78(6):1837–1867, 2012. JCSS Multidisciplinary Emerging Networks and Systems (MENS).

[CRMM78] Stefano Crespi-Reghizzi, Dino Mandrioli, and David F. Martin. Algebraic properties of operator precedence languages. *Information and Control*, 37(2):115–133, 1978.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2):275–288, Oct 1992.

[DB96] Koen De Bosschere. An Operator Precedence Parser for Standard Prolog Text. *Software: Practice and Experience*, 26(7):763–779, 1996.

[DDMP17] Manfred Droste, Stefan Dück, Dino Mandrioli, and Matteo Pradella. Weighted Operator Precedence Languages. In Kim G. Larsen, Hans L. Bodlaender, and Jean-Francois Raskin, editors, *42nd International Symposium on Mathematical Foundations of Computer Science (MFCS 2017)*, volume 83 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 31:1–31:15, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Ear70] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, February 1970.

[Elg61] Calvin C. Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98(1):21–51, 1961.

[FG04] Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. *Annals of Pure and Applied Logic*, 130(1):3–31, 2004. Papers presented at the 2002 IEEE Symposium on Logic in Computer Science (LICS).

[Fis69] Michael J. Fischer. Some properties of precedence languages. In *Proceedings of the First Annual ACM Symposium on Theory of Computing*, STOC '69, pages 181–190, New York, NY, USA, 1969. ACM.

[Flo63] Robert W. Floyd. Syntactic analysis and operator precedence. *J. ACM*, 10(3):316–333, July 1963.

[Har78] Michael A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1978.

[Kam68] Hans Kamp. *Tense logic and the theory of linear order*. PhD thesis, University of California, Los Angeles, 1968.

[Knu65] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[LMP11] Violetta Lonati, Dino Mandrioli, and Matteo Pradella. Precedence automata and languages. In Alexander Kulikov and Nikolay Vereshchagin, editors, *Computer Science – Theory and Applications*, pages 291–304, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[LMP13] Violetta Lonati, Dino Mandrioli, and Matteo Pradella. Logic characterization of invisibly structured languages: The case of Floyd languages. In Peter van Emde Boas, Frans C. A. Groen, Giuseppe F. Italiano, Jerzy Nawrocki, and Harald Sack, editors, *SOFSEM 2013: Theory and Practice of Computer Science*, pages 307–318, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[LMPP15] Violetta Lonati, Dino Mandrioli, Federica Panella, and Matteo Pradella. Operator precedence languages: Their automata-theoretic and logic characterization. *SIAM Journal on Computing*, 44(4):1026–1088, 2015.

[LMS05] Christof Löding, P. Madhusudan, and Olivier Serre. Visibly pushdown games. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science*, pages 408–420, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[LTNP07] Salvatore La Torre, Margherita Napoli, and Mimmo Parente. The word problem for visibly pushdown languages described by grammars. *Formal Methods in System Design*, 31(3):265, Oct 2007.

[McN67] Robert McNaughton. Parenthesis grammars. *J. ACM*, 14(3):490–500, July 1967.

[Meh80] Kurt Mehlhorn. Pebbling mountain ranges and its application to dcfl-recognition. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, pages 422–435, Berlin, Heidelberg, 1980. Springer Berlin Heidelberg.

[MP18] Dino Mandrioli and Matteo Pradella. Generalizing input-driven languages: Theoretical and practical benefits. *Computer Science Review*, 27:61–87, 2018.

[OS17] Alexander Okhotin and Kai Salomaa. The quotient operation on input-driven pushdown automata. In Giovanni Pighizzini and Cezar Câmpeanu, editors, *Descriptional Complexity of Formal Systems*, pages 299–310, Cham, 2017. Springer International Publishing.

[OS18] Alexander Okhotin and Kai Salomaa. Further closure properties of input-driven pushdown automata. In Stavros Konstantinidis and Giovanni Pighizzini, editors, *Descriptional Complexity of Formal Systems*, pages 224–236, Cham, 2018. Springer International Publishing.

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (SFCS 1977)*, pages 46–57, Oct 1977.

[PPLM13] Federica Panella, Matteo Pradella, Violetta Lonati, and Dino Mandrioli. Operator precedence ω-languages. In Marie-Pierre Béal and Olivier Carton, editors, *Developments in Language Theory*, pages 396–408, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, July 1985.

[Tra61] Boris A. Trakhtenbrot. Finite automata and logic of monadic predicates. *Doklady Akademii Nauk SSSR*, 140:326–329, 1961.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331. IEEE Computer Society, 1986.