



# POLITECNICO MILANO 1863

DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA  
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

---

## TEMPORAL LOGIC AND MODEL CHECKING FOR OPERATOR PRECEDENCE LANGUAGES: THEORY AND APPLICATIONS

Doctoral Dissertation of:  
**Michele Chiari**

Supervisor:

**Prof. Dino Mandrioli**

Co-supervisor:

**Prof. Matteo Pradella**

Tutor:

**Prof. Raffaella Mirandola**

Chair of the Doctoral Program:

**Prof. Barbara Pernici**

Vice-Chair of the Computer Science and Engineering Area:

**Prof. Cristina Silvano**



## Abstract

The growing ubiquity of computer systems in every industrial sector has posed increasingly demanding challenges, one of the most crucial being the verification of adherence of mission- and safety-critical systems to their requirements. One of the most successful and popular techniques that have been developed for this objective is model checking. It consists of the formal specification of the system's requirements by means of a logic formalism, in the generation of a model of the system in exam by using an operational or denotational formalism, and in the subsequent automatic and exhaustive verification of the adherence of the latter to the former.

The capabilities of this process depend on the choice of such formalisms. The most classical thereof are Linear Temporal Logic (LTL), Computation Tree Logic (CTL) and CTL\*. In particular, LTL maintains the same expressive power as First-Order Logic (FOL), while providing a model-checking procedure which is only exponential in specification size, and not non-elementary as FOL. However, LTL only allows to express properties in the realm of regular languages.

Recently, the literature has seen considerable efforts in the direction of extending the expressive power of temporal logic towards larger language classes. The most notable of them is the work on temporal logics based on Visibly Pushdown Languages (VPLs), which led to the introduction of the logic CaRet, and of the First-Order complete NWTL. VPL are a subclass of Deterministic Context-Free Languages (DCFLs), and they are only slightly more expressive than parenthesis languages. The main motivation of this line of research is the verification of procedural programs: the one-to-one correspondence between function calls and returns can only be modeled by a context-free language.

This thesis develops a model-checking framework based on Operator Precedence Languages (OPLs). OPLs, another subclass of DCFLs, are significantly more expressive than VPLs and, consequently, regular languages. Being suitable for describing the syntax of real-world programming languages, they could dramatically extend the properties expressible in system specifications. In particular, they enable verification of programs with exceptions, which VPL-based formalisms could not do. In this thesis, we present two temporal logic formalisms capable of expressing OPL properties. The first one, OPTL, is a first attempt at this task, for which we develop a model-checking procedure. Unfortunately, OPTL is strictly less expressive than FOL. Thus, we introduce a better logic, POTL, for which we prove equivalence to FOL, and develop and implement an automata-theoretic model-checking procedure. We demonstrate the applicability of the resulting tool through case studies, showing promising results.



## Acknowledgments

I would like to express my sincere gratitude to my supervisors, Dino Mandrioli and Matteo Pradella, for their guidance and support throughout this journey and for helping me overcome the difficulties that arose from my work on this thesis.

I am also thankful to Davide Bergamaschi and Francesco Pontiggia for letting me co-supervise their Master’s Theses and helping me develop POMC, the tool I present at the end of this thesis.

My gratitude is also due to all the researchers with whom I collaborated in works that are not strictly related to my thesis, but that formed a considerable contribution to my Ph.D. and my growth as a researcher. So I thank the TAFFO<sup>1</sup> team from the HEAPLab<sup>2</sup> group—namely, Giampaolo Agosta, Daniele Cattaneo, Stefano Cherubin, Nicola Fossati, Gabriele Magnani—and also Roberto Bagnara and Roberta Gori.

Moreover, working in DEEP-SE<sup>3</sup> was undoubtedly helpful, thanks to the lively environment created by its members. So I thank Mehrnoosh Askarpour, Marcello M. Bersani, Hassan Chaudhri, Danilo Filgueira Mendonça, Federica Filippini, Eugenio Gianniti, Michele Guerriero, Bruno Guindani, Marjan Hosseini, Davide Hu, Alireza Javadian Sabet, Safia Kalwar, Livia Lestingi, Francesco Marconi, Alexander Nemirowskiy, Galia Novakova Nedeltcheva, Giovanni E. Quattrocchi, Vincenzo Scotti, Mahsa Shekari, Damian A. Tamburri, Luca Terracciano, Riccardo Tommasini, Alessandra Viale, and Shima Zahmatkesh.

This achievement would not have been possible without the support of my parents and friends, to whom I express my utmost gratitude.

Finally, I am thankful to Manfred Droste and Adriano Peron for their careful review of this thesis.

---

<sup>1</sup><https://taffo-org.github.io/>

<sup>2</sup><https://heaplab.deib.polimi.it/>

<sup>3</sup><https://www.deepse.deib.polimi.it/>



# Contents

<b>Contents</b>	<b>7</b>
<b>1 Introduction</b>	<b>11</b>
1.1 Summary of Contributions . . . . .	15
1.2 Structure of the Thesis . . . . .	15
<b>I Background</b>	<b>17</b>
<b>2 Temporal Logic and Model Checking</b>	<b>21</b>
2.1 Transition Systems . . . . .	21
2.2 Linear Temporal Logic . . . . .	23
2.2.1 Expressive Completeness . . . . .	26
2.2.2 LTL Model Checking . . . . .	27
2.2.2.1 Büchi Automata . . . . .	28
2.2.2.2 NBA Construction . . . . .	30
2.2.2.3 Wrap-Up . . . . .	32
2.3 Branching-Time Logics and Model Checking . . . . .	33
<b>3 Context-Free Model Checking</b>	<b>37</b>
3.1 Context-free models and regular specifications . . . . .	37
3.1.1 Pushdown Systems . . . . .	37
3.1.2 Recursive State Machines . . . . .	38
3.1.3 Boolean Programs . . . . .	39
3.2 Context-free models and context-free specifications . . . . .	39
3.2.1 Process Algebra . . . . .	39
3.2.2 Approaches that specify properties on stack contents . . . . .	40
3.2.3 Other Approaches . . . . .	40
3.2.4 Visibly Pushdown Languages and Nested Words . . . . .	40
3.2.4.1 CaRet . . . . .	41
3.2.4.2 Visibly Pushdown Languages . . . . .	41
3.2.4.3 Nested Words . . . . .	42
3.2.4.4 Nested Words Temporal Logic and Expressive Completeness . . . . .	43
3.2.4.5 More developments on VPLs and Nested Words . . . . .	44
3.2.4.6 Tools . . . . .	45

<b>4</b>	<b>Operator Precedence Languages</b>	<b>47</b>
4.1	Operator Precedence Languages on finite words . . . . .	47
4.2	Operator Precedence $\omega$ -Languages . . . . .	56
4.3	Modeling Programs with OPA . . . . .	58
<b>5</b>	<b>Model-Theoretic Background</b>	<b>61</b>
5.1	Elementary equivalence and its characterizations . . . . .	61
5.2	Ehrenfeucht-Fraïssé Games . . . . .	64
5.3	Composition Arguments . . . . .	66
<b>II</b>	<b>Temporal Logic</b>	<b>69</b>
<b>6</b>	<b>OPTL Syntax and Semantics</b>	<b>73</b>
6.1	Operator Precedence Words . . . . .	73
6.2	Syntax and Informal Semantics . . . . .	76
6.3	Formal Semantics . . . . .	77
6.3.1	OPTL on $\omega$ -Words . . . . .	80
6.4	Examples . . . . .	80
6.5	Model Checking and Satisfiability . . . . .	82
<b>7</b>	<b>OPTL Expressiveness</b>	<b>83</b>
7.1	Relationship with Nested Words . . . . .	83
7.2	Relationship with First-Order Logic . . . . .	87
7.2.1	OPTL's Limitations . . . . .	87
7.2.2	OPTL is not expressively complete . . . . .	88
<b>8</b>	<b>POTL Syntax and Semantics</b>	<b>93</b>
8.1	Syntax and Formal Semantics . . . . .	94
8.1.1	Equivalences . . . . .	97
8.1.1.1	Expansion Laws . . . . .	97
8.1.1.2	Shortcuts . . . . .	99
8.2	POTL on $\omega$ -Words . . . . .	99
8.3	Motivating Examples . . . . .	99
8.4	Comparison with other logics . . . . .	100
8.4.1	Linear Temporal Logic (LTL) . . . . .	100
8.4.2	Logics on Nested Words . . . . .	101
8.4.3	Logics on OPLs . . . . .	101
<b>9</b>	<b>POTL Expressive Completeness</b>	<b>103</b>
9.1	First-Order Completeness on Finite Words . . . . .	103
9.1.1	First-Order Semantics of POTL . . . . .	103
9.1.2	Expressing $\mathcal{X}_{until}$ in POTL . . . . .	105
9.1.2.1	OPM-compatible Unranked Ordered Trees . . . . .	105
9.1.2.2	POTL Translation of $\mathcal{X}_{until}$ . . . . .	107
9.2	First-Order Completeness on $\omega$ -Words . . . . .	112
9.2.1	OPM-compatible $\omega$ -UOTs . . . . .	112
9.2.2	Notation . . . . .	114
9.2.3	RR UOTs . . . . .	115
9.2.4	LR UOTs . . . . .	119

9.2.5	Synthesis . . . . .	122
<b>III</b>	<b>Model Checking</b>	<b>125</b>
<b>10</b>	<b>Model Checking Construction</b>	<b>129</b>
10.1	Finite-Word Model Checking . . . . .	129
10.1.1	Next/Back Operators . . . . .	130
10.1.2	Chain Next Operators . . . . .	130
10.1.3	Chain Back Operators . . . . .	136
10.1.4	Summary Until and Since . . . . .	141
10.1.5	Hierarchical Operators . . . . .	141
10.1.6	Concluding Proof . . . . .	143
10.2	$\omega$ -Word Model Checking . . . . .	144
<b>11</b>	<b>Experimental Evaluation</b>	<b>147</b>
11.1	Implementation . . . . .	147
11.1.1	OPA Emptiness Checking . . . . .	147
11.1.2	$\omega$ OPBA Emptiness Checking . . . . .	148
11.1.3	Modeling Procedural Programs . . . . .	150
11.2	Experimental Evaluation . . . . .	151
11.2.1	Discussion . . . . .	155
<b>IV</b>	<b>Epilogue</b>	<b>159</b>
<b>12</b>	<b>Conclusions and Future Work</b>	<b>161</b>
12.1	Discussion and Contributions . . . . .	161
12.2	Future Work Directions . . . . .	163
	<b>Bibliography</b>	<b>165</b>
	<b>Index of Acronyms</b>	<b>177</b>
	<b>List of Figures</b>	<b>179</b>
	<b>List of Tables</b>	<b>181</b>



# Chapter 1

## Introduction

Functions, or procedures, are arguably the most successful modularization device ever introduced in programming languages. Originally inspired by mathematical functions, they are present in programming languages from almost all paradigms, thanks to their natural abstraction capabilities. Hence, any technique aimed at program verification must be able to not only model them as accurately as possible, but also allow for requirements that take them fully into account. To achieve this, the language used for specifications must explicitly support reasoning on their typical behaviors. Much like what happens with natural languages, if this is not the case, then the resulting sentences—and, consequently, proofs about the program—will necessarily tackle a limited part of the system’s behavior.

In this respect, the current stance of *model checking*, one of the most widely used verification techniques, presents room for improvement. In short, model checking is the name given to a broad range of techniques for the fully automated verification of a system model against a formal specification of its intended behavior. The result of this process is either the assurance that the system satisfies the specification, or a counterexample, i.e., a behavior of the system that does not satisfy such requirements. The system properties that can be verified depend on the choice of the mathematical formalism to be employed for both the model and the specification. For the latter, temporal logic is one of the most successful, thanks to its good balance between expressiveness and computational efficiency of verification algorithms.

Different kinds of temporal logics are usually categorized depending on the way they model time. Linear Temporal Logic (LTL) [138] sees time as a linear sequence of discrete events, each one of them represented by an atomic proposition, as shown in Figure 1.1. This is ideal to express requirements such as “the system will never present unwanted behavior  $A$ ”, “the system will continuously perform task  $B$ ”, or “the system will not have behavior  $A$  until it has done  $B$ ”.

Computation Tree Logic (CTL) [59] instead sees time as an unranked tree, where each node represents a possible system state, and branches starting from it represent its possible future behaviors. This perspective is well-suited to reason on nondeterministic systems, and to express properties such as “the system will always be able to reach a state from where it may do task  $A$ ”, or “there is at least one execution path in which  $B$  happens”.

However, when writing specifications for procedural programs, we may want to express properties such as *Hoare*-style pre- and post-conditions [97] (e.g., “if pre-condition  $\rho$  holds when procedure  $A$  is called, post-condition  $\theta$  will hold when it

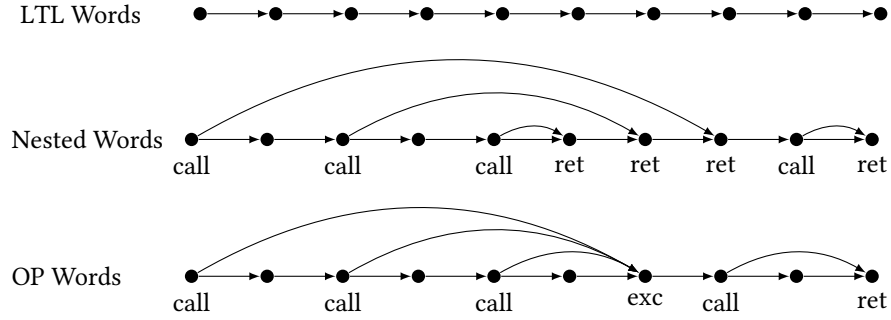


Figure 1.1: Time models of LTL (top), Nested Words (middle) and OP Words (bottom).

returns”); *stack inspection* [102] (e.g., “privileged procedure A cannot be called if unprivileged procedure B is active on the stack”); or *exception safety* [1] (e.g., “procedure A never throws an exception”, and “if procedure B throws an exception, the program remains in a valid state”). These properties refer to a matching between function calls and the returns or exceptions that terminate them. Since the algebraic structures on which the semantics of LTL and CTL are based do not contain such a matching, they are unsuitable to express them.

This limitation is clearer when we analyze such formalisms from the point of view of formal language theory. Indeed, LTL and CTL only represent (a subset of) *regular* properties, i.e., properties expressible as languages generated by a type-3 grammar from the Chomsky hierarchy [93], and accepted by *Finite-State Automata (FSAs)*. Procedural programs, instead, use a LIFO stack to keep track of function activation records: the resulting behaviors can be described by *pushdown automata*, and the corresponding formal language class is *Context-Free Languages (CFLs)*, generated by type-2 grammars in the Chomsky hierarchy.

LTL and CTL model checking have been studied for operational formalisms aimed at modeling procedural programs, such as *Pushdown Systems* [30, 78] and *Recursive State Machines* [9]. However, the limited fitness of LTL and CTL for expressing requirements on them motivated the introduction of temporal logics capable of representing restricted subclasses of CFLs.

CaRet [8] is one of them: the algebraic structure on which its semantics is defined contains a linear ordering, making it a linear-time logic like LTL, equipped with a binary, one-to-one nesting relation. These structures are called *Nested Words* [5], and an example is shown in Figure 1.1. The nesting relation naturally models the matching between function calls and returns, and enables the definition of modal operators that refer to this structure directly. E.g., CaRet contains an *abstract next* operator that states a property of the return associated to the call where it is evaluated, enabling requirements such as Hoare-style pre/post-conditions. Also, it contains a *call-until* operator for expressing stack inspection properties. Another notable logic on nested words is Nested Words Temporal Logic (NWTL) [12], which contains until and since operators based on *summary paths*, that can skip bodies of nested function calls. NWTL has been proven to be equivalent to First-Order Logic (FOL) on nested words, unlike CaRet.

Coming back to the language-theoretic point of view, nested words are an algebraic representation of *Visibly Pushdown Languages (VPLs)* [4], a subclass of Deterministic Context-Free Languages (DCFLs). In VPLs, terminal symbols are partitioned

into *call*, *internal* and *return* symbols. Call and return symbols act much like open and closed parentheses, enclosing internal symbols. VPLs are more general than R. McNaughton’s *Parenthesis Grammars* [128] in that they allow for a set of different call and return symbols, internal symbols, and unmatched calls and returns at the end and at the beginning of strings. While this is enough for representing normally-terminating functions, it shows its limitations when the programming language to be modeled features more advanced constructs like *exceptions* and *continuations*. These flow-control devices cause complex stack behaviors such as the simultaneous termination or instantiation of multiple functions. So, they should be modeled as single events matched with multiple ones (e.g., function calls matched with the exception that terminates them). Unfortunately, the matching relation of nested words is strictly one-to-one, and thus unable to model this kind of behaviors. It is therefore impossible to reason on them with logics based on nested words.

The work presented in this thesis is aimed at closing this gap. In fact, we present temporal logics based on Operator Precedence Languages (OPLs), a language class more powerful than VPLs [121]. OPLs were originally introduced by R.W. Floyd [80] with the purpose of efficient parsing, but were later almost forgotten after D. Knuth introduced LR parsers and proved that they cover the whole class of DCFLs [108]. Our interest in OPLs comes from their nice properties: they are closed by Boolean operations [66], plus concatenation and Kleene \* [64]; language inclusion and of course emptiness are decidable. This makes them a “robust” language class, well-suited for model checking, like regular languages and VPLs, and unlike DCFLs and general CFLs, which are not closed by intersection, and for which inclusion is undecidable [93]. Most importantly, OPLs are a much more expressive language class than VPLs [121]. Since they were introduced with the aim of parsing real-word programming languages, they are not limited to representing parenthetical structures, but also strings whose context-free structure is not visible, such as arithmetic expressions. Thus, when representing their context-free structure as a nesting relation, we obtain a much more general matching, allowing for many-to-one and one-to-many relations. This is ideal to represent, e.g., exceptions, as shown in Figure 1.1. We call *Operator Precedence (OP) words* the algebraic structure made of a linear ordering plus a nesting relation coming from OPLs.

We introduce two temporal logics on OP words. The first one is called Operator Precedence Temporal Logic (OPTL). OPTL features modalities that interact with the multi-matching structure of OP words: there are *matching next/back* operators that state something about the *maximal* (i.e., farthest) position matched with the current one, and *OP-summary until/since* operators that skip positions in between such maximally matched positions. Thus, OPTL can express all properties expressible by logics on nested words such as CaRet and NWTL—and we prove this claim formally—plus more requirements regarding exceptions, such as exception safety. Moreover, OPTL has *hierarchical* operators that express properties on multiple positions in relation with one, that enable requirements on multiple function calls terminated by the same exception (e.g., “only function A may throw an exception, and function B may throw only by a invoking A”).

When investigating the expressiveness of temporal logics, the most common yardstick is FOL. LTL has been proved to be FO-complete (i.e., equivalent to FOL) in [106], and FO-completeness was the motivation for introducing NWTL, and several other temporal logics [12]. Unfortunately, we were not able to prove the FO-completeness of OPTL, but we proved that it is strictly less expressive than FOL. While this is a negative result, we argue that it is still an interesting one. First, while CaRet is con-

jectured not to be FO-complete [12], a proof of such claim has yet to be seen, and the authors of [12] themselves state that it “appears to be difficult”. Second, our proof is based on a novel *Pumping Lemma* for temporal logics, which could be applicable—with appropriate adaptations—to other CFL-based temporal logics.

Thus, we come to the second temporal logic we introduce: Precedence Oriented Temporal Logic (POTL). POTL is equipped with modalities devised with the idea of “navigating” a word’s underlying syntax tree. It features until and since operators that can navigate up or down in the syntax tree, possibly skipping subtrees (which, in the context of procedural program execution traces means skipping function bodies). Also, unlike OPTL, it has chainable hierarchical operators that, as we shall see, allow for expressing more complex properties on e.g. function calls terminated by the same exception, or calls issued by the same function. POTL can express all OPTL-expressible properties, and has some advantages over it. In particular, it can express properties limited to a single subtree in a word’s syntax tree, which allows us to naturally express certain function-local properties that are not expressible in OPTL.

Moreover, we formally prove that POTL is equivalent to FOL on OP words. Proofs of this kind are often very difficult and long: the initial proof of FO-completeness of LTL was the topic of an entire Ph.D. thesis [106], and later proofs are still rather involved [85, 143]; the FO-completeness proofs in [12] use sophisticated model-theoretic techniques such as composition arguments based on Ehrenfeucht-Fraïssé games. For our proof, we follow this last route. We first translate POTL into a FO-complete logic on finite trees,  $\mathcal{X}_{\text{until}}$  [125]. Then, we extend this result to infinite words by means of a composition argument. Our composition argument is somewhat more complex than the ones used in [12], because of the greater variety of tree shapes resulting from OP  $\omega$ -words, which must be combined together.

We remark that equivalence to FOL on context-free languages is an even better assurance of good expressive power than on regular languages (such as for LTL). In fact, the FO-definable fragment of OPLs was recently proved to be equivalent to the class of *non-counting* or *aperiodic* OPLs [122, 123], retracing analogous results on regular languages [129]. With CFLs, aperiodicity manifests by preventing counting properties on the structure of the syntax tree, so that e.g. “procedure A calls itself recursively an even number of times” is not expressible. However, requirements of this kind do not seem to be of great interest, as we shall see when giving examples of useful requirements expressible by OPTL and POTL. Instead, many practically-interesting regular languages are counting.

As a corollary of the FO-completeness proof of POTL, we prove that FOL on OP words has the *three-variable* property. This property holds for a class of algebraic structures if every First-Order (FO) formula on their signature can be written with at most three distinct variable names. It has been linked with expressive completeness of temporal logic [84], and was proved for labeled linear orders (which we also call *LTL words*) and nested words.

We then present an automata-theoretic model checking procedure for POTL based on Operator Precedence Automata (OPAs), the class of automata accepting OPLs. Due to the more complex semantics of OPA, our construction is quite involved, and we formally prove it for each operator. The resulting automata are exponential in the size of the formula, which is in line with the analogous constructions for NWTL [12] and LTL [158]. Moreover, the nice property of LTL model checking of being polynomial in model size is kept, as POTL model checking is also polynomial in model size. POTL model checking and satisfiability are EXPTIME-complete, which is higher than LTL—although the model checking algorithm is still exponential—but the same as NWTL.

We implemented our model checking procedure in an explicit-state model checking tool called Precedence Oriented Model Checker (POMC) [55], for both the finite- and infinite-word cases. To the best of our knowledge, this is the first publicly available tool for model checking a temporal logic with context-free modalities (i.e., including CaRet, NWTL etc.). We tested POMC on several interesting case-studies, checking properties such as exception safety and stack inspection. The procedural programs we checked were both modeled manually and automatically generated from a simple programming language. The results we obtained are promising: we were able to check in a matter of seconds interesting properties on models with up to thousands of states.

## 1.1 Summary of Contributions

Here we briefly summarize the contributions presented in this thesis.

- We introduce two temporal logics on OPLs, OPTL and POTL, and show that they are capable of expressing many useful requirements on procedural programs with exceptions;
- we prove that they are strictly more expressive than state-of-the-art logics on nested words such as CaRet and NWTL;
- we prove that OPTL is not FO-complete in an original way;
- we prove that POTL is FO-complete;
- we prove that OP words enjoy the three-variable property;
- we provide a model-checking procedure for POTL and study its complexity, which is not worse than state-of-the-art logics;
- we implement POTL model checking for both finite and infinite OP words in the open source tool POMC;
- we experimentally evaluate POMC on case studies, obtaining promising results.

## 1.2 Structure of the Thesis

This thesis is divided in four parts. Each part is introduced by a brief note explaining its structure, and the articles in which their contents (if they are original contributions) have been published.

Part I has two aims: introducing background notions that are essential for understanding the rest of the thesis, and giving an overview of the related work in this field. In particular, background is given on model checking in general—and LTL in particular—, on OPLs on both finite and infinite words, and on the model-theoretic techniques used in the FO-completeness proof of POTL.

Part II presents the syntax and semantics of OPTL and POTL, and contains their characterization in terms of expressiveness, i.e., the proofs that OPTL is more expressive than NWTL, and that it is less expressive than FOL, and the FO-completeness proof of POTL.

Part III details the model checking procedures for POTL and their implementation and evaluation.

Finally, Part IV concludes the thesis by providing some directions for future work.



**Part I**

**Background**



As we mentioned in the Introduction, the purpose of this part of the thesis is twofold: introducing background concepts that are essential to understand the rest of the thesis, and surveying related work on the topics we address.

Chapter 2, where we introduce some background on model checking and temporal logic, addresses both such purposes, and is particularly aimed at readers not familiar with such topics.

Chapter 3, instead, gives an overview of related work on the topics of model checking of context-free systems and specifications, with the aim of helping the reader understand the positioning of our work with respect to the state-of-the-art. Thus, we only give a more detailed introduction of previous works that we deem most relevant, and give appropriate references for those not so closely related.

Chapter 4 gives an introduction to OPLs especially aimed at readers not necessarily well-versed in formal language theory. Some knowledge on OPLs is essential for understanding the main topics of the thesis.

Finally, Chapter 5 briefly introduces the model-theoretic techniques used in some of the proofs in Chapter 9. Readers not interested in fully understanding such proofs may skip this chapter.



## Chapter 2

# Temporal Logic and Model Checking

Model checking is a family of techniques for tackling the problem of checking the correctness of software and cyber-physical systems. The main idea that links such techniques—and that considerably contributed to their success—is their being fully automated. This differentiates them from other approaches to verification, such as mechanized theorem proving, that require a variable amount of human interaction. The ease of use gained in this way comes at the cost of restricting the classes of properties that can be checked, which must be decidable.

At the high level, model checking consists of generating a model of the system to be analyzed, coming up with a formal specification of the properties to be assessed, and checking that the former satisfies the latter. If this is not the case, a *counterexample*, i.e. a possible behavior of the system model that violates the specification, is produced. The class of properties that can be verified in this way is determined by the formalisms used for the model and for the specification. The model is often an over-approximation of the actual system, and can represent faithfully only some of its behaviors (especially if the system is a Turing-complete one). Thus, only properties related to such behaviors can be checked. On the other hand, the formalism used for the specification must admit an algorithmic procedure that checks any expressible property with a reasonable temporal and spatial computational complexity (this excludes Turing-complete formalisms, which may represent undecidable properties).

In this section, we give a succinct presentation of classical model checking techniques, focusing on those that model time as a linear succession of discrete events. For a broader presentation of model checking techniques which are out of the scope of this work, cf. [17, 63].

### 2.1 Transition Systems

Discrete systems are most often modeled by means of *transition systems*. A transition system is a directed graph where each vertex represents a system state in a particular moment of its execution, and edges represent actions that cause a possible evolution of the system. Information about the system's state is encoded through *atomic propositions*, which are facts that are true (or false) about the system in one of its states. For

example, if the system is a computer program, atomic propositions encode variable values, such as  $x = 0$ ,  $x > 0$ , and so on. Each action, or transition, can also be labeled.

**Definition 2.1** (Transition System [17]). Given a set of atomic propositions  $AP$  and a set of labels  $\Lambda$ , a transition system is a tuple  $\mathcal{M} = (S, I, L, \delta)$ , where  $S$  is a set of system states,  $I \subseteq S$  is a set of initial states,  $L : S \rightarrow \mathcal{P}(AP)$  is the state-labeling function, and  $\delta \subseteq S \times \Lambda \times S$  is the transition relation.

A transition system is called *finite* if the sets  $AP$ ,  $\Lambda$  and  $S$  are finite.

The  $\delta$  relation describes how the system evolves by linking each state to each possible next one, through actions. Note that  $\delta$  does not have to be a function, i.e. for each state  $s$ , there may be 0 or more states  $s'$  and actions  $a$  such that  $(s, a, s') \in \delta$ . A state with no successors is called *terminal*. The semantics of transition systems is formalized through the notion of *run* or *execution*.

**Definition 2.2.** A finite run of a transition system  $\mathcal{M}$  is a finite sequence  $\rho = s_0 a_1 s_1 a_2 \dots a_n s_n$ , where  $n \geq 0$ , such that  $s_0 \in I$ ,  $s_n$  is a terminal state, and for all  $0 \leq i < n$  we have  $(s_i, a_{i+1}, s_{i+1}) \in \delta$ . An infinite run is defined similarly, except there is no last state or action.

The distinction between finite and infinite runs reflects the need for formalizing both systems whose executions have an end, such as terminating computer programs, and systems that run continuously, such as process controllers and software that runs on servers.

Runs describe the behavior of the systems from an internal point of view, as they show transitions between states explicitly. However, such a level of detail is often not needed or not possible. Thus, we introduce *execution traces*, which describe the system's behavior through the atomic propositions assigned to states by the labeling function. This can be seen as a way of describing the system's evolution only through its *observable* behavior, possibly treating it as a black box, whose internal details are not known (or not interesting).

**Definition 2.3** (Execution Trace). The *trace* of a run  $\rho = s_0 a_1 s_1 a_2 \dots a_n s_n$  is a sequence  $\pi = L(\rho) = L(s_0)L(s_1) \dots L(s_n)$ .

This definition is extended to infinite runs in the obvious way.

We denote as  $L(\mathcal{M}) = \{L(\rho) \mid \rho \text{ is a run of } \mathcal{M}\}$  the set of all traces generated by transition system  $\mathcal{M}$ .

Thus, an execution trace is a sequence, or a *string* or *word*, of sets of atomic propositions from  $\mathcal{P}(AP)$ .

*Example 2.4.* Figure 2.1 shows a simple procedural program, and a possible way of modeling it as a transition system. States are represented with circles, with the atomic propositions holding in them next to them. The only initial state is  $q_0$ . Transitions are represented with arrows, next to their labels. The atomic propositions are *start* and *end*, marking the beginning and end of the program;  $p_A$ , which holds in states that are part of procedure  $p_A$ ;  $x \geq 4$  and  $x < 4$ , which tell us whether the value of variable  $x$  is lower than 4 or not. Since it marks the end of the program, state  $q_5$  is indeed terminal.

Action labels represent program statements, whose execution makes the state of the system to evolve. Additionally, *call* and *ret* mark the invocation and termination of a procedure.

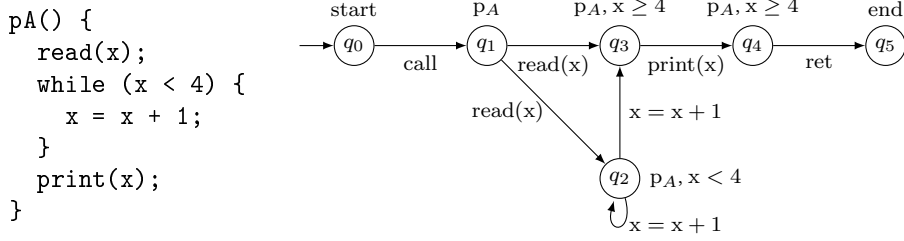


Figure 2.1: A program (left) and the transition system modeling it (right).

Note that this transition system represents the state of the program only partially, since there is no way of knowing the exact value of  $x$  in any state. However, since the program's flow of execution is only sensitive to whether  $x$  is lower than 4 or not, this is enough to represent some of its possible behaviors. One possible run of this system is the following:

$$\rho_{fin} = q_0 \text{ call } q_1 \text{ read}(x) q_3 \text{ print}(x) q_4 \text{ ret } q_5$$

State  $q_5$  is terminal, so the run is finite. This run's execution trace is

$$\{\text{start}\} \{p_A\} \{p_A, x \geq 4\} \{p_A, x \geq 4\} \{\text{end}\}.$$

This transition system also exhibits infinite runs, such as the following:

$$\rho_{inf} = q_0 \text{ call } q_1 \text{ read}(x) q_2 (x = x + 1) q_2 (x = x + 1) q_2 \dots$$

Its execution trace is the following:

$$\{\text{start}\} \{p_A\} \{p_A, x < 4\} \{p_A, x < 4\} \{p_A, x < 4\} \dots$$

The original program, however, is always terminating, since variable  $x$  will necessarily reach the value 4 if it is always incremented. Thus, this transition system *abstracts* from the original program, and represents faithfully only some of its behaviors.

In order to automatically check transition systems against requirement specifications, such properties must be stated through an appropriate formal language, which usually defines a set of system traces that are considered acceptable. Temporal logics are a family of formal languages that are often the premier choice for this purpose.

## 2.2 Linear Temporal Logic

Linear Temporal Logic (LTL) is a modal propositional logic for expressing properties on a linear, usually discrete, timeline. LTL, previously known as *tense logic* [106], was proposed for expressing program requirements by Amir Pnueli in 1977 [138]. LTL is a modal extension of propositional logic, so a finite set of *atomic propositions*  $AP$  needs to be chosen. Then, its syntax is the following:

$$\varphi ::= a \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi \mathcal{U} \varphi \mid \ominus\varphi \mid \varphi \mathcal{S} \varphi$$

where  $a$  can be any element of  $AP$ . The propositional operators can be augmented with  $\top$ , which is always true,  $\wedge$ ,  $\implies$  and  $\iff$  as usual.

Temporal modalities can be divided in *future* ( $\circ$  and  $\mathcal{U}$ ) and *past* ( $\ominus$  and  $\mathcal{S}$ ). A *future* LTL formula contains only future operators, and a *past* one only past operators. LTL is sometimes presented with future operators only (indeed, such operators are expressively complete thanks to the *separation* property [86], as we will see in Section 2.2.1); here we include past operators too, for convenience. The semantics of LTL is based on a discrete<sup>1</sup> linear timeline of events, all labeled with atomic propositions describing them. Each formula is evaluated on one of such time instants. The  $\circ$  operator, called *next*, says something about the immediate future:  $\circ\varphi$  is true in a time instant if and only if the subformula  $\varphi$  is true in the next one. The  $\mathcal{U}$  operator, called *until*, concerns a time interval:  $\varphi\mathcal{U}\psi$  is true at an instant if and only if  $\psi$  is true in a future instant, and  $\varphi$  holds in all instants between them. The *back*  $\ominus$  and *since*  $\mathcal{S}$  operators are the past versions of, respectively,  $\circ$  and  $\mathcal{U}$ .

Formally, the semantics of LTL is based on *words*.

**Definition 2.5 (Word).** A *word* on an alphabet of atomic propositions  $AP$  is a tuple  $w = (U, <, P)$ , where either  $U = \{0, \dots, n\}$ ,  $n \in \mathbb{N}$  (and  $w$  is called a *finite* word), or  $U = \mathbb{N}$  (and  $w$  is an *infinite* or  $\omega$ -word);  $<$  is the usual ordering on  $\mathbb{N}$ ; and  $P : AP \rightarrow \mathcal{P}(U)$  is a *labeling* function associating each atomic proposition to the set of positions where it holds.

LTL formulas are evaluated on a single word position: the notation  $(w, i) \models \varphi$  means that LTL formula  $\varphi$  holds on position  $i$  of word  $w$ . For all words  $w$  and positions  $i$ , we define

- $(w, i) \models a$  iff  $i \in P(a)$ ;
- $(w, i) \models \neg\varphi$  iff  $(w, i) \not\models \varphi$  ( $\varphi$  does not hold in position  $i$  of word  $w$ );
- $(w, i) \models \varphi \vee \psi$  iff  $(w, i) \models \varphi$  or  $(w, i) \models \psi$  or both;
- $(w, i) \models \circ\varphi$  iff  $(w, i + 1) \models \varphi$ ;
- $(w, i) \models \varphi\mathcal{U}\psi$  iff there exists a position  $j \geq i$  such that  $(w, j) \models \psi$  and for all  $i \leq k < j$  we have  $(w, k) \models \varphi$ ;
- $(w, i) \models \ominus\varphi$  iff  $(w, i - 1) \models \varphi$ ;
- $(w, i) \models \varphi\mathcal{S}\psi$  iff there exists a position  $j \leq i$  such that  $(w, j) \models \psi$  and for all  $j < k \leq i$  we have  $(w, k) \models \varphi$ .

We say that a word  $w$  satisfies an LTL formula  $\varphi$  iff  $(w, 0) \models \varphi$ . We can thus see LTL formulas as a way of specifying a formal language: we define the language denoted by  $\varphi$  as

$$L(\varphi) = \{w \mid (w, 0) \models \varphi\}.$$

Until and since operators entail an existential quantification on *linear paths* between the position where they are evaluated and a future (resp. past) one. We call such paths *linear* because they are made of consecutive positions.

*Example 2.6.* We can interpret the execution traces of  $\rho_{fin}$  and  $\rho_{inf}$  from Example 2.4 as  $(\omega)$ -words, respectively called  $w_{fin}$  and  $w_{inf}$ , as shown in Figure 2.2.

Formula  $\circ(x < 4)$  holds in all positions in  $w_{inf}$  except 0, since  $x < 4$  always holds in the next one. Instead,  $\ominus start$  only holds in position 1, because *start* holds in

<sup>1</sup>LTL can be defined on any set equipped with a Dedekind-complete linear order. So, while in principle it could be defined on a continuous set, in this work we are only concerned with discrete timelines.

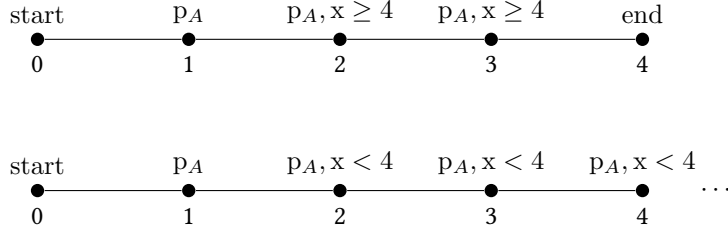


Figure 2.2: Execution traces of runs  $\rho_{fin}$  (top) and  $\rho_{inf}$  (bottom) from Example 2.4 represented as words, respectively  $w_{fin}$  and  $w_{inf}$ . Word positions are displayed as dots with their number below and the atomic propositions holding in them above.

0. Formula  $\alpha := p_A \mathcal{U} \text{end}$  is false in all positions of  $w_{inf}$ , because  $\text{end}$  never holds.  $\alpha$  holds, however, in positions 1, 2 and 3 of  $w_{fin}$ . E.g., in position 1 it is witnessed by the path from 1 to 4, because  $p_A$  holds in 1, 2 and 3, and  $\text{end}$  holds in 4. Thus, formula  $\alpha$  holds in a position if it is part of the execution of procedure  $p_A$ , which must eventually terminate.

Formula  $p_A \mathcal{S} \text{start}$  is true in all positions of  $w_{inf}$ , and in all positions of  $w_{fin}$  except the last one. It means that procedure  $p_A$  has always been active since the start of the program. E.g., it holds in position 3 of  $w_{fin}$  because  $\text{start}$  holds in 0 and  $p_A$  in 1, 2 and 3; it does not hold in position 4 because  $p_A$  does not hold in it.

Until and since operators can be concatenated. For example,  $p_A \mathcal{U} ((x \geq 4) \mathcal{U} \text{end})$  is true in positions where  $p_A$  is active, and after which  $p_A$  terminates with  $x \geq 4$ , such as 1, 2 and 3 in  $w_{fin}$ , but also—trivially—position 4.

In the rest of this thesis, we shall see temporal logics whose until and since operators existentially quantify on different kinds of paths, whose positions may not be consecutive, although they are always linearly ordered.

A few shortcuts are usually defined to make it easier to express properties in LTL. The *eventually* operator is defined as  $\diamond \varphi := \top \mathcal{U} \varphi$ , and it holds in a position  $i$  iff there exists a future position  $j \geq i$  where  $\varphi$  holds. It can be used to say that, in the future, a certain event will occur or a requirement will be satisfied. The *globally* operator is defined as  $\square \varphi := \neg \diamond (\neg \varphi)$  and, due to the double negation, it holds in a position  $i$  iff  $\varphi$  holds in all positions  $j \geq i$ . It can be used to express *safety* requirements, that make sure that some bad behavior will never occur in the system. Another important kind of requirement is obtained by nesting these two operators, with formulas of the form  $\square \diamond \varphi$ . Such requirements are called *liveness* properties, and they ensure that a desired behavior will continuously occur sometime in the future.

*Example 2.6* (continuing from p. 24). Formula  $\diamond \text{end}$  is true in the first (and subsequent) position of words representing execution traces that terminate, such as  $w_{fin}$  from Figure 2.2. If we check  $\diamond \text{end}$  on the transition system of Figure 2.1, we find out that it does not hold, because there are traces such as the one represented by  $w_{inf}$  that do not satisfy it, although the program does.

Formula  $\square \neg (x < 4)$  is true in the first position of  $w_{fin}$ , and it distinguishes cases in which the value entered by the user and assigned to variable  $x$  is greater than or equal to 4.

*Expansion laws* are some of the most important properties of LTL operators: for

any formula  $\varphi$  and  $\psi$ ,

$$\varphi \mathcal{U} \psi \equiv \psi \vee (\varphi \wedge \bigcirc(\varphi \mathcal{U} \psi)), \quad \varphi \mathcal{S} \psi \equiv \psi \vee (\varphi \wedge \ominus(\varphi \mathcal{S} \psi)).$$

Seeing why such laws hold is straightforward:  $\varphi \mathcal{U} \psi$  is true in a position  $i$  if

- $\psi$  holds in  $i$ , so that  $j = i$  and the path witnessing its truth is made on the sole position  $i$ , or
- $\varphi$  holds in  $i$ , and  $\varphi \mathcal{U} \psi$  in  $j$ , which means that there is a path between  $i + 1$  and a position  $j \geq i + 1$  where  $\psi$  holds, and  $\varphi$  holds in all  $i + 1 \leq j' < j$ . If we concatenate  $i$  with this path, we obtain another path that witnesses  $\varphi \mathcal{U} \psi$  in  $i$ .

The explanation for the since case is symmetric. The expansion laws make the non-local requirements entailed by until and since more “local”, which—as we shall see—is useful for LTL model checking.

### 2.2.1 Expressive Completeness

The solidity of LTL’s expressive power is rooted in its equivalence to the first-order theory of the linear order equipped with monadic relations. By this we mean FOL on the signature  $(U, <, (P_a)_{a \in AP})$ , where  $U$  and  $<$  are as in Definition 2.5, and all  $P_a$ ’s are sets (or *monadic relations*) of positions where atomic proposition  $a \in AP$  holds. Formulas in this logic consist of:

- propositional connectives  $\wedge, \vee$ , and  $\neg, \implies, \iff$ ;
- the quantifiers  $\exists x$  and  $\forall x$ , which quantify variables on single positions in  $U$ ;
- monadic predicates  $a(x)$ , which is true iff the value assigned to variable  $x$  is in  $P_a$ , for all  $a \in AP$ ;
- comparisons between variables with  $x < y$  and  $x = y$ .

A variable is said to occur *free* in a formula if it is not quantified.

The extension of first-order logic that allows for quantification over finite sets of word positions is called *weak Monadic Second-Order Logic (MSOL)*. For a more formal and complete introduction to FOL and MSOL, we refer the reader to e.g. [23].

The expressiveness of LTL is characterized by the following seminal result:

**Theorem 2.7** (Kamp’s Theorem [106]). *LTL = FOL with one free variable.*

This means that:

1. for every LTL formula  $\varphi$ , there is an equivalent first-order formula  $\overline{\varphi}(x)$  such that for any word  $w$  and position  $i$  in it, we have  $(w, i) \models \varphi$  iff  $(w, i) \models \overline{\varphi}(x)$ ;
2. for every first-order formula with one free variable  $\overline{\varphi}(x)$  on the above signature, there is an LTL formula  $\varphi$  such that for any word  $w$  and position  $i$  in it, we have  $(w, i) \models \varphi$  iff  $(w, i) \models \overline{\varphi}(x)$ .

Point 1 is quite straightforward, because any LTL formula can be translated into FOL by composing the formal semantics of its operators. The proofs of point 2 are, instead, much more involved. This is justified by a difference in computational complexity: as we will see in Section 2.2.2, LTL satisfiability is PSPACE-complete, while

FOL satisfiability is nonelementary [152]. Thus, translations from FOL to LTL incur in a nonelementary blowup in their length.

After the first proof by Hans Kamp [106], alternative proofs of Theorem 2.7 have been developed. The one from [85] exploits an important property of LTL: the *separation property*. This property says that any LTL formula can be equivalently expressed by a Boolean combination of pure future and past LTL formulas. If such a formula is evaluated on a position  $i$  of a word  $w$ , past subformulas only depend on the prefix of  $w$  from 0 to  $i$ , while future ones only depend on the sub-word starting from  $i$ . Thus, LTL with no past operators is equivalent to FOL too, when its formulas are evaluated in the first position of a word.

Equivalence to FOL, or expressive completeness, is considered a good assurance of a temporal logic's expressiveness. In fact, it has been proved for more expressive variants of LTL [12], which we review in Section 3, and even for temporal logics on trees [126]. Unfortunately, the separation property does not hold for such logics [24], and such proofs use more sophisticated model-theoretic devices.

A consequence of LTL's expressive completeness is the following:

**Theorem 2.8** (Three-variable property [106]). *Every first-order formula on words with at most one free variable is equivalent to one using at most three distinct variables.*

It is proved by showing that the formal semantics of LTL can be expressed using only three variables. Then, arbitrary FO formulas can be translated to LTL thanks to Theorem 2.7, and then back to FOL with three variables by carefully combining LTL formal semantics. The three-variable property for a structure has been linked to the existence of a possibly multi-dimensional expressively-complete modal logic for it [84]. The fragment of FOL with only two variables has also been studied, and proved to be equivalent to a temporal logic with only next, back, and future and past eventually operators [77].

What made LTL successful is not only its ability to express useful temporal requirements, but also the possibility of model checking it automatically and efficiently.

### 2.2.2 LTL Model Checking

Model checking a system  $\mathcal{M}$  against an LTL specification  $\varphi$  means verifying that all execution traces generated by  $\mathcal{M}$  satisfy  $\varphi$ . More formally,

**Definition 2.9** (LTL Model Checking). Given a transition system  $\mathcal{M}$  and an LTL formula  $\varphi$ , the *model checking problem* consists in determining whether

$$L(\mathcal{M}) \subseteq L(\varphi).$$

Here we illustrate the classical automata-theoretic model-checking procedure for LTL introduced by M. Y. Vardi and P. Wolper [158]. The general idea is to exploit the closure properties of a class of automata that have a sufficient expressive power to represent both the system and the specification. For any LTL formula  $\varphi$ , an automaton  $\mathcal{A}_\varphi$  that accepts exactly execution traces satisfying  $\varphi$  can be built. Thus, if we express the system  $\mathcal{M}$  to be checked as one of such automata, then we can build  $\mathcal{A}_\varphi$  and, if the class of automata we use is closed by intersection, we can combine them to obtain an automaton that accepts only execution traces of  $\mathcal{M}$  that satisfy  $\neg\varphi$ , and hence do *not* satisfy  $\varphi$ . If the language accepted by such an automaton is empty, then there exist no such traces, and we have proved that all possible behaviors of  $\mathcal{M}$  satisfy  $\varphi$ . Otherwise,

the resulting automaton is a succinct representation of all traces of  $\mathcal{M}$  that violate  $\varphi$ . Thus, in this case we are able to extract *counterexamples* useful to understand why  $\mathcal{M}$  does not satisfy its specification.

More formally, we have

$$\begin{aligned} L(\mathcal{M}) &\subseteq L(\varphi) \\ \iff L(\mathcal{M}) \cap \overline{L(\varphi)} &= \emptyset \\ \iff L(\mathcal{M}) \cap L(\neg\varphi) &= \emptyset \end{aligned}$$

where by  $\overline{L(\varphi)}$  we mean the set complement of  $L(\varphi)$ . So, if we are able to build an automaton  $\mathcal{A}_{\mathcal{M}}$  such that  $L(\mathcal{A}_{\mathcal{M}}) = L(\mathcal{M})$ , and an automaton  $\mathcal{A}_{\neg\varphi}$  such that  $L(\mathcal{A}_{\neg\varphi}) = L(\neg\varphi)$ , we have

$$L(\mathcal{M}) \subseteq L(\varphi) \iff L(\mathcal{A}_{\mathcal{M}}) \cap L(\mathcal{A}_{\neg\varphi}) = \emptyset.$$

Thus, we have an effective way to solve the model checking problem if the automata class we use admits a practical way of building the intersection automaton for any two automata, and of checking its language emptiness.

This procedure has been originally developed for model checking systems with infinite behaviors, which are usually of greater interest, and uses Büchi automata. We illustrate it in this section, and we assume without loss of generality that  $L(\mathcal{M})$  only contains infinite words.<sup>2</sup> The approach is, however, very general, and can be implemented by means of other automata classes, as we shall see throughout this thesis. For model checking LTL on finite execution traces, in particular, *Finite State Automata* accepting the class of *regular* languages can be used [17].

### 2.2.2.1 Büchi Automata

Büchi automata are acceptors of infinite words. Given a finite alphabet  $\Sigma$ , an *infinite* or  $\omega$ -word is an infinite sequence  $a_0a_1a_2\dots$  with  $a_i \in \Sigma$  for all  $i \in \mathbb{N}$ . We denote as  $\Sigma^\omega$  the set of  $\omega$ -words with characters in  $\Sigma$ .

**Definition 2.10** (Nondeterministic Büchi Automaton (NBA) [42]). A nondeterministic *Büchi* automaton is a tuple  $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ , where  $\Sigma$  is a finite input alphabet,  $Q$  is a finite set of states,  $I \subseteq Q$  is a set of initial states,  $F \subseteq Q$  is a set of final states, and  $\delta \subseteq (Q \times \Sigma) \times Q$  is the transition relation.

An NBA *configuration* is a pair  $\langle w, q \rangle$ , where  $w \in \Sigma^\omega$  is the input word and  $q \in Q$  is the current state. A *run* of an NBA is an infinite sequence of configurations  $\langle w_0, q_0 \rangle \langle w_1, q_1 \rangle \dots$  such that for all  $i \geq 1$  we have  $w_i = a_i w_{i+1}$  for some  $a_i \in \Sigma$ , and  $(q_0, a_i, q_1) \in \delta$ . A run  $\rho$  on  $\omega$ -word  $w$  is a run that starts in a configuration  $\langle w, q_0 \rangle$  for some  $q_0 \in I$ . We define

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

as the set of states that occur infinitely often in  $\rho$ . Run  $\rho$  is *successful* if there is a final state  $q_f \in F$  such that  $q_f \in \text{Inf}(\rho)$ , and an  $\omega$ -word  $w$  is *accepted* by  $\mathcal{A}$  if it performs a successful run on  $w$ . We define the language accepted by  $\mathcal{A}$  as

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

<sup>2</sup>It is always possible to transform a transition system into one that has no finite behaviors by replacing terminal states with *stuttering states*. This is done by simply adding transitions that both start and end in those states, obtaining infinite runs where the stuttering state is repeated infinitely.

The class of languages accepted by NBAs is that of  $\omega$ -regular languages [155].

**Definition 2.11.** A language  $L \subseteq \Sigma^\omega$  is  $\omega$ -regular if there exists an NBA  $\mathcal{A}$  such that  $L(\mathcal{A}) = L$ .

$\omega$ -regular languages have been characterized also by means of  $\omega$ -regular expressions and monadic second-order logic [155].

NBAs have important closure properties, as they form a Boolean algebra [155].

**Theorem 2.12.** Given two NBAs  $\mathcal{A}_1$  and  $\mathcal{A}_2$  on the same alphabet  $\Sigma$ , it is possible to effectively build the following NBAs:

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

Such closure properties are essential for model checking: since LTL contains propositional operators such as  $\vee$ ,  $\wedge$ , and  $\neg$ , any class of automata must be closed by union, intersection and complement in order to recognize languages of traces that satisfy any LTL formula. Closure by intersection is also needed, so we can compute  $L(\mathcal{A}_M) \cap L(\mathcal{A}_{\neg\varphi})$ .

A deterministic version of Büchi automata exists too, but such automata are strictly less powerful than their nondeterministic counterparts [155]. This differs from what happens with regular languages of finite words, as nondeterministic finite-state automata can always be determinized [93].

Before showing the automata construction for model checking, we introduce a variation on Büchi acceptance conditions:

**Definition 2.13** (Generalized NBA). A *generalized NBA* is a tuple  $\mathcal{A} = (\Sigma, Q, I, \mathcal{F}, \delta)$  where  $\Sigma$ ,  $Q$ ,  $I$ , and  $\delta$  are the same as in Definition 2.10, while  $\mathcal{F} \subseteq \mathcal{P}(Q)$  is a set of sets of final states.

The semantics of configurations and runs is the same as for normal NBAs. Instead, the acceptance condition is the conjunction of multiple Büchi acceptance conditions, one for each set in  $\mathcal{F}$ : a run  $\rho$  is *successful* if for each  $F \in \mathcal{F}$  there is a state  $q_f \in F$  such that  $q_f \in \text{Inf}(\rho)$ . The definitions of  $\omega$ -word and language acceptance change consequently.

Generalized NBAs can be translated to normal NBAs polynomially:

**Theorem 2.14.** Let  $\mathcal{A} = (\Sigma, Q, I, \mathcal{F}, \delta)$  be a generalized NBA. It is possible to build an NBA  $\mathcal{A}'$  with  $|Q| \cdot |\mathbf{F}|$  states such that  $L(\mathcal{A}') = L(\mathcal{A})$ .

*Proof.* We use a classical construction based on counters (see e.g. [111]). Let  $k = |\mathbf{F}|$ ; we assign a linear ordering to sets in  $|\mathbf{F}|$ , and call  $F_i$ ,  $0 \leq i \leq k-1$ , the  $i$ -th of such sets. We define  $\mathcal{A}' = (\Sigma, Q', I', F', \delta')$  as follows:

- $Q' = Q \times \{0, \dots, k-1\}$ ;
- $I' = I \times \{0\}$ ;
- $F' = F_1 \times \{0\}$ ;
- if  $(q, a, p) \in \delta$ , then for  $0 \leq i \leq k-1$  we have  $((q, i), a, (p, j)) \in \delta'$ , where  $j = i$  if  $q \notin F_i$ , and  $j = i+1 \pmod k$  otherwise.

$\mathcal{A}'$  contains  $k$  different copies of  $\mathcal{A}$ ; it starts by running the first one, and switches to the next one (modulo  $k$ ) each time it finds an accepting state for the current one. Since the accepting states must be from the first copy, each infinite run must necessarily cycle through all of them, so that each accepting condition is satisfied.  $\square$

Since any normal NBA can be seen as a generalized NBA where  $\mathcal{F}$  is a singleton, we can state the equivalence of the two classes in terms of expressive power. Thus, all properties proved for NBAs in Theorem 2.12 also hold for generalized NBAs.

### 2.2.2.2 NBA Construction

To model check transition systems, we consider NBAs that use  $\Sigma = \mathcal{P}(AP)$  as their alphabet, where  $AP$  is a set of atomic propositions. Then, if a transition system is finite, building an NBA that accepts the language of its traces is quite straightforward. Given  $\mathcal{M} = (S, I, L, \delta)$ , we define  $\mathcal{A}_{\mathcal{M}} = (\mathcal{P}(AP), S, I, S, \delta')$ —so the set of states is the same, and all states are final—where we have  $(q, L(q), q') \in \delta'$  iff  $(q, b, q') \in \delta$  for some  $b \in \Lambda$  (recall that  $\Lambda$  is the set of action labels).

Now, we show how to build  $\mathcal{A}_{\neg\varphi}$ .

**Theorem 2.15** ([158]). *For any LTL formula  $\bar{\varphi}$ —and we take  $\bar{\varphi} = \neg\varphi$  in our case—we can build a generalized NBA  $\mathcal{A}_{\bar{\varphi}}$  such that  $L(\mathcal{A}_{\bar{\varphi}}) = L(\bar{\varphi})$  with  $2^{O(|\bar{\varphi}|)}$  states.*

*Proof.* The idea behind this construction is to make an NBA that has sets of LTL formulas as its states, and its transition relation is such that, at any moment, the formulas contained in the current state are those that hold in the next word position to be read. Thus, in particular, the NBA will only read symbols containing the exact same atomic propositions present non-negated in the current state.

We define the *closure* of a formula  $\bar{\varphi}$ , denoted  $\text{Cl}(\bar{\varphi})$ , as the smallest set such that:

- $\bar{\varphi} \in \text{Cl}(\bar{\varphi})$ ;
- $AP \subseteq \text{Cl}(\bar{\varphi})$ ;
- for any formula  $\psi \in \text{Cl}(\bar{\varphi})$  such that  $\psi \neq \neg\theta$  for any formula  $\theta$ , we have  $\neg\psi \in \text{Cl}(\bar{\varphi})$  (we identify  $\neg\neg\psi$  with  $\psi$ );
- if  $\neg\psi \in \text{Cl}(\bar{\varphi})$  for some formula  $\psi$ , then  $\psi \in \text{Cl}(\bar{\varphi})$ ;
- if any of  $\bigcirc\psi$  or  $\ominus\psi$  is in  $\text{Cl}(\bar{\varphi})$  for some formula  $\psi$ , then  $\psi \in \text{Cl}(\bar{\varphi})$ ;
- if any of  $\psi \wedge \theta$ ,  $\psi \vee \theta$ ,  $\psi \mathcal{U} \theta$ , or  $\psi \mathcal{S} \theta$  is in  $\text{Cl}(\bar{\varphi})$  for some  $\psi$  and  $\theta$ , then  $\psi \in \text{Cl}(\bar{\varphi})$  and  $\theta \in \text{Cl}(\bar{\varphi})$ ;
- if  $\psi \mathcal{U} \theta \in \text{Cl}(\bar{\varphi})$  for some  $\psi$  and  $\theta$ , then  $\bigcirc(\psi \mathcal{U} \theta) \in \text{Cl}(\bar{\varphi})$ ;
- if  $\psi \mathcal{S} \theta \in \text{Cl}(\bar{\varphi})$  for some  $\psi$  and  $\theta$ , then  $\ominus(\psi \mathcal{S} \theta) \in \text{Cl}(\bar{\varphi})$ .

Note that the size of  $\text{Cl}(\bar{\varphi})$  is linear in the length of  $\bar{\varphi}$ .

Now we define the set of  $\text{Atoms}(\bar{\varphi}) \subseteq \mathcal{P}(\text{Cl}(\bar{\varphi}))$ , and we denote its elements with capital letters of the Greek alphabet. We use  $\text{Atoms}(\bar{\varphi})$  as the set of states for  $\mathcal{A}_{\bar{\varphi}}$ : each state is the set of all and only formulas that are true in it. Thus, elements of  $\text{Atoms}(\bar{\varphi})$  must be *consistent* with respect to propositional logic and LTL, which means that they cannot contain any contradiction. In particular,  $\text{Atoms}(\bar{\varphi})$  contains all and only sets  $\Phi \subseteq \text{Cl}(\bar{\varphi})$  such that

- for every  $\psi \in \text{Cl}(\bar{\varphi})$ , we have  $\psi \in \Phi$  iff  $\neg\psi \notin \Phi$ ;
- $\psi \wedge \theta \in \Phi$  iff  $\psi \in \Phi$  and  $\theta \in \Phi$ ;
- $\psi \vee \theta \in \Phi$  iff  $\psi \in \Phi$  or  $\theta \in \Phi$ , or both;
- $\psi \mathcal{U} \theta \in \Phi$  iff one or both of the following conditions hold:
  1.  $\theta \in \Phi$ , or
  2.  $\psi \in \Phi$  and  $\circ(\psi \mathcal{U} \theta) \in \Phi$ ;
- $\psi \mathcal{S} \theta \in \Phi$  iff one or both of the following conditions hold:
  1.  $\theta \in \Phi$ ,
  2.  $\psi \in \Phi$  and  $\ominus(\psi \mathcal{S} \theta) \in \Phi$ .

The last two rules, which are justified by the expansion laws, reduce the satisfaction of the LTL until and since operators to that of the next and back operators.

We define  $\mathcal{A}_{\bar{\varphi}} = (\mathcal{P}(AP), \text{Atoms}(\bar{\varphi}), I_{\bar{\varphi}}, \mathcal{F}_{\bar{\varphi}}, \delta_{\bar{\varphi}})$  with  $I_{\bar{\varphi}} = \{\Phi \in \text{Atoms}(\bar{\varphi}) \mid \bar{\varphi} \in \Phi\}$ . The transition relation  $\delta_{\bar{\varphi}}$  contains all triples  $(\Phi, a, \Psi) \in \text{Atoms}(\bar{\varphi}) \times \mathcal{P}(AP) \times \text{Atoms}(\bar{\varphi})$  such that

- $a = \Phi \cap AP$ ;
- for any  $\circ\psi \in \text{Cl}(\bar{\varphi})$ , we have  $\circ\psi \in \Phi$  iff  $\psi \in \Psi$ ;
- for any  $\ominus\psi \in \text{Cl}(\bar{\varphi})$ , we have  $\ominus\psi \in \Psi$  iff  $\psi \in \Phi$ .

In this way, the transition relation encodes the temporal obligations entailed by the next and back operators.

Notice that, for now, for until operators we only use the consistency requirement on  $\text{Atoms}(\bar{\varphi})$  derived from the expansion law. This requirement can be satisfied even by an infinite succession of states in which  $\psi$  and  $\psi \mathcal{U} \theta$  hold, but  $\theta$  never does. Thus, the generalized Büchi condition is chosen to make sure that the until operators are eventually satisfied, so that for every until formula  $\psi \mathcal{U} \theta \in \text{Cl}(\bar{\varphi})$  that appears in a state,  $\theta$  holds in a subsequent state. Thus, for each  $\psi \mathcal{U} \theta \in \text{Cl}(\bar{\varphi})$  we define

$$F_{\psi \mathcal{U} \theta} = \{\Phi \in \text{Atoms}(\bar{\varphi}) \mid \psi \mathcal{U} \theta \notin \Phi \vee \theta \in \Phi\},$$

and  $\mathcal{F}_{\bar{\varphi}}$  contains all and only such sets. □

*Example 2.16.* Figure 2.3 shows the NBA built for formula  $\diamond \text{end} \equiv \top \mathcal{U} \text{end}$  according to the procedure reported in this section. We only have  $AP = \{\text{end}\}$  as the set of atomic propositions, and the closure is

$$\text{Cl}(\top \mathcal{U} \text{end}) = \{\top \mathcal{U} \text{end}, \top, \text{end}, \circ(\top \mathcal{U} \text{end}), \neg(\top \mathcal{U} \text{end}), \neg \text{end}, \neg \circ(\top \mathcal{U} \text{end})\}.$$

The set of initial states is  $I = \{q_0, q_1, q_3\}$ , as they all contain  $\top \mathcal{U} \text{end}$ . There is only one set of final states,  $F_{\top \mathcal{U} \text{end}} = \{q_1, q_2, q_3\}$ , because the formula contains only one until operator.

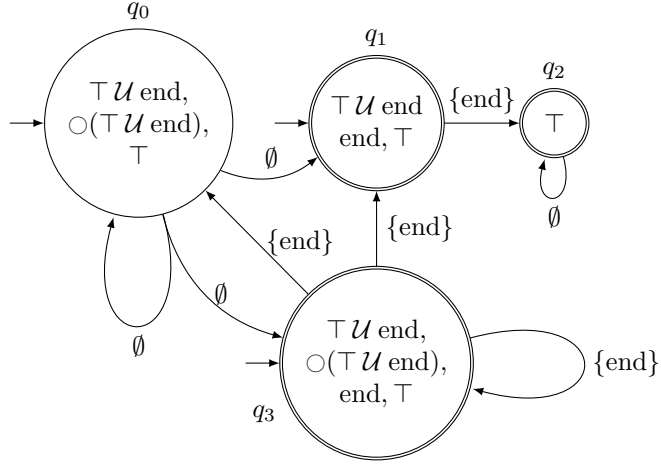


Figure 2.3: Generalized NBA built for formula  $\diamond \text{end}$ . Initial states are marked with an entering arrow, and final ones are drawn with a double line. Negated formulas are omitted for readability: if a formula does not appear in a state, it means it should appear in it negated (e.g.,  $\neg \text{end}$  is in  $q_0$  and  $q_2$ ).

### 2.2.2.3 Wrap-Up

Once we have  $\mathcal{A}_{\mathcal{M}}$  and  $\mathcal{A}_{\neg\varphi}$ , we can compute their intersection according to Theorem 2.12. In practice, there is no need to transform the generalized NBA  $\mathcal{A}_{\neg\varphi}$  into a normal one, because the intersection procedure can be adapted to work with generalized NBAs directly. Actually, an intersection procedure is often defined directly between transition systems and generalized NBAs [17].

To check the intersection for emptiness, graph-theoretic techniques can be used. In particular, an NBA is not empty if there is a reachable cycle in its transition relation that contains a final state. It is easy to see that, if such a cycle exists, the NBA admits at least a run in which a final state is visited infinitely often. This can be checked by using algorithms for finding *strongly connected components*, or by looking for reachable cycles directly, with a procedure called *nested Depth-First Search (DFS)* [17]. The latter consists in launching a first DFS which finds a final state; once such a state is found, another search is launched to see if it is reachable from itself. This approach can perform better in practice, because it does not always require to store the whole NBA in memory. In fact,  $\mathcal{A}_{\mathcal{M}}$ ,  $\mathcal{A}_{\neg\varphi}$  and their intersection can be computed on-the-fly, generating states only when they are visited. The procedure stops as soon as a cycle with a final state is found, terminating early if the NBA is not empty, and showing that  $\mathcal{M}$  violates  $\varphi$ . If no such cycle is found, the whole automaton must be visited to prove that  $\mathcal{M}$  satisfies  $\varphi$ . Hence, the worst-case complexity of this procedure is  $O(n \cdot (n + m))$ , where  $n$  is the number of reachable states and  $m$  the number of transitions connecting them [17]. While  $n$  and  $m$  are linear in  $\mathcal{M}$ , note that the size of  $\mathcal{A}_{\neg\varphi}$  is exponential in the length of  $\varphi$ , according to Theorem 2.15. Thus, the overall temporal complexity is polynomial in model size, but exponential in the size of the specification. However, the LTL formula expressing the specification is (usually) short, while the model can be much larger than  $\mathcal{A}_{\neg\varphi}$ . This makes LTL model checking practically feasible, despite its membership in a complexity class which is usually considered intractable:

**Theorem 2.17** ([150]). *LTL satisfiability and model-checking are PSPACE-complete.*

Thus, LTL model checking is advantageous over that of formalisms such as weak monadic second-order logic, which on one hand is more expressive, as it has the same expressiveness as NBAs [155], but on the other hand its complexity is nonelementary [131]. FOL model checking is PSPACE-complete too [151], but there is no known algorithm whose complexity is favorably split between model and formula size, as for LTL. Indeed, [83] proves that any algorithm for FOL model checking that is polynomial in model size must be nonelementary in formula size.

*Remark 2.18.* The approach described above only works if the transition system  $\mathcal{M}$  to be checked is finite. If it has an infinite number of states, it cannot be translated into an NBA. Even if a direct intersection between transition system and NBA is done, the nested DFS algorithm can only show that the specification is violated by finding an acceptance cycle. It cannot show that  $\mathcal{M}$  satisfies it, because that would require visiting all states of  $\mathcal{M}$ . This is a significant limitation, because many systems of interest are infinite-state: for example, computer programs can be seen as having an unlimited memory (even if a computer’s memory is limited, it is often too big to consider all of its possible configurations explicitly). Indeed, if function `pA` of the example program in Figure 2.1 was recursive, the program stack, which keeps track of active function frames, could potentially grow unbounded, generating an infinite number of configurations. The resulting program would not be representable by a finite transition system.

However, some infinite-state systems admit a finite representation. One notable example, which is the main subject of this thesis, are systems equipped with a stack, such as different kinds of *pushdown* systems. They are interesting because they can model the stack that procedural programs use to keep track of function calls. In the following, we will see how this kind of systems can be modeled and model-checked.

In the last few decades, numerous techniques that improve on the practical feasibility of LTL model checking have been introduced, especially for coping with the large size of models, called the *state-space explosion* problem. An important family of such techniques, called *symbolic* model checking, uses devices such as *Boolean Decision Diagrams (BDDs)* for representing model states more efficiently; others exploit recent advances in Propositional Satisfiability (SAT) and *Satisfiability Modulo Theories (SMT)* solvers to solve the model-checking problem. Such techniques are, however, out of the scope of this work: we refer the interested reader to [63].

## 2.3 Branching-Time Logics and Model Checking

In LTL, time is seen as linear, so that every time instant can be followed by exactly another one. Instead, the family of branching-time logics sees time as a tree, where any time instant can be followed by one among multiple events, each one leading to a different “timeline”, or *execution path*. This represents well the nondeterminism inherent in many systems to be checked, such as cyber-physical systems.

**CTL and CTL\*** One of the first branching-time logics to be introduced is Computation Tree Logic (CTL) [59, 142]. The syntax of CTL is divided in two parts:

$$\begin{aligned}\varphi &::= a \mid \varphi \wedge \varphi \mid \neg\varphi \mid \mathbf{A} \psi \mid \mathbf{E} \psi \\ \psi &::= \bigcirc \varphi \mid \varphi \mathbf{U} \varphi\end{aligned}$$

Formulas generated by  $\varphi$  are *state* formulas, which hold in a state of the system, and those generated by  $\psi$  are *path* formulas, that hold on an execution path starting from a state. State formulas can universally or existentially quantify over paths starting from the state where they hold. Operators in path formulas have the same meaning they have in LTL, where an execution path can be seen as a word made of the subsequent states visited by the run.

For example, formula  $\mathbf{A} \circ a$  says that  $a$  will hold in *all* states that are reachable from the current one in one step, while  $\mathbf{E} \circ a$  says that  $a$  will hold in *at least one* of them. Similarly, formula  $\mathbf{A} \diamond a$ , where  $\diamond$  is the same as in LTL, means that  $a$  will eventually hold in a state in *all* execution paths starting from the current state, and  $\mathbf{E} \diamond a$  that  $a$  will eventually hold in *at least one* of such paths.

LTL formulas can be seen as if they were implicitly universally quantified at the top level—e.g.,  $\circ a$  in LTL is equivalent to  $\mathbf{A} \circ a$  in CTL—but have no way of quantifying other parts of the formula. There are CTL formulas that are not expressible in LTL: for example,  $\mathbf{A} \square \mathbf{E} \diamond a$  means that in every computation, it is always possible to reach a state where  $a$  holds. LTL can express the liveness requirement  $\square \diamond a$ , which however means that  $a$  will be *always* reached in all paths, not just possibly. This requirement, in turn, is not expressible in CTL [17]. Thus, LTL and CTL are incomparable in terms of expressiveness.

The logic CTL\* was introduced to join the expressive power of LTL with that of CTL [59]. It does so by allowing for nested path formulas in CTL, namely replacing the definition of  $\psi$  above with

$$\psi ::= \varphi \mid \circ \psi \mid \psi \mathcal{U} \psi.$$

Thus, in CTL\* it is possible to nest temporal modalities directly, writing formulas such as  $\mathbf{A} \square \diamond a$ , which expresses the LTL liveness requirement. Indeed, CTL\* can express all LTL and CTL properties, and it is strictly more expressive than both of them [17].

In terms of model-checking complexity, one interesting feature is that CTL model checking is in PTIME, and can be implemented with a quite natural graph-theoretic algorithm polynomial in both model and formula length, working directly on transition systems [59]. CTL satisfiability is, instead, in EXPTIME [72]. CTL\* is, of course, computationally costlier: its model checking is PSPACE-complete [60], and satisfiability is in 2EXPTIME [73].

**$\mu$ -calculus** Modal  $\mu$ -calculus [109] is another branching-time logic which has applications in model checking. It features propositional operators and atomic propositions that are evaluated on transition system states, variables that represent sets of states, existential and universal quantification on single transitions, and least and greatest fixpoint operators. The fixpoint operators, combined with those quantifying single transitions, can denote the existence of finite or infinite sequences of transitions, traversing states where certain propositional formulas hold.

The importance of  $\mu$ -calculus derives in part from its expressive power: it is equivalent to monadic second-order logic on deterministic tree transition systems<sup>3</sup> [136]. Thus, it subsumes both LTL, CTL and CTL\* [67]. One relevant characteristic of  $\mu$ -calculus formulas is their *alternation depth*, which is (roughly) the number of alternations among nested least and greatest fixpoints. Alternation-free  $\mu$ -calculus, the fragment of  $\mu$ -calculus with alternation depth 1, is equivalent to weak monadic second-order

<sup>3</sup>A tree transition system is a transition system whose transition graph is a tree, with a root state that has a unique path to every other state.

logic [136]. While a lower bound for the complexity of  $\mu$ -calculus model checking has still to be proved, the currently known algorithms have complexities of the form  $n^{O(d)}$ , where  $n$  is the size of the model and  $d$  is the alternation depth of the formula [40]. Satisfiability, instead, is EXPTIME-complete [73].

For a more complete overview of  $\mu$ -calculus, we refer the reader to [39, 40]. We do not present  $\mu$ -calculus and other branching-time logics in more depth, because they are quite different from the kind of formalisms studied in this thesis.



## Chapter 3

# Context-Free Model Checking

In this section, we survey the related work on model checking of pushdown systems and context-free properties. We describe to some extent the works most relevant with respect to this thesis, and only give an overview of the remaining ones, for which we refer interested readers to the appropriate literature.

Starting from the 1990s and throughout the early 2000s, during a period of intense research on model checking, part of the community investigated ways to broaden the scope of both systems and specifications to be checked. Given the relationship between LTL and regular languages, one natural direction was to go up in the Chomsky hierarchy, by trying to model-check context-free languages. This endeavor was also suggested by practical applications: procedural programming languages use a LIFO stack to keep track of procedure calls, generating inherently context-free behaviors.

### 3.1 Context-free models and regular specifications

Initial efforts were directed towards extending model checking of existing specification formalisms (e.g., LTL, CTL, CTL\* and  $\mu$ -calculus) to context-free system models. We survey this part of the literature in this section.

#### 3.1.1 Pushdown Systems

*Pushdown Systems (PDS's)*, a.k.a. *pushdown processes*, have received considerable attention because of their suitability for abstracting the stack of sequential procedural programs, while remaining much simpler than more general formalisms such as process algebra. PDS's are very close to nondeterministic pushdown automata from formal-language theory and, with a few exceptions (e.g. [45], where transitions are labeled), they differ from them only by their inability to read input symbols. Indeed, in model checking only their stack behaviors are of interest, unlike their language-accepting capabilities. PDS's may also exhibit infinite behaviors.

The literature contains a profligacy of formal definitions for PDS's, all of them being equivalent, or nearly so. Here we give one which should be more familiar for formal-language theorists.

**Definition 3.1** (Pushdown System (PDS)). A PDS is a tuple  $\mathcal{A} = (Q, \Gamma, \delta)$  where  $Q$  is a finite set of control states,  $\Gamma$  is the stack alphabet, and  $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma^*)$

is the set of transition rules. If  $|Q| = 1$ , then  $\mathcal{A}$  is a *context-free* process.<sup>1</sup>

A PDS *configuration* is an element of  $Q \times \Gamma^*$ , and a PDS can transition from a configuration  $(q, A\gamma)$  to  $(p, \alpha\gamma)$  if  $(q, A, p, \alpha) \in \delta$ .

PDS definitions often differ in whether transition rules may pop zero or one symbols, and push zero or more. However, all such definitions are equivalent, because  $\Gamma$  can be expanded to include symbols that encode multiple ones.

One of the formalisms used for model-checking PDS's is *modal  $\mu$ -calculus*. Model checking of the *alternation-free* fragment of modal  $\mu$ -calculus was applied to sequential context-free processes in [44, 100], and extended to the full modal  $\mu$ -calculus in [45, 159], with an exponential computational complexity. [159] also proves that the problem is EXPTIME-complete, even when fixing the specification.

The authors of [30] introduce an algorithm for reachability in alternating PDS's that uses alternating finite-state automata for representing regular sets of configurations. From this, they derive algorithms for model-checking LTL on PDS's that are exponential in formula size and polynomial in model size, and prove the EXPTIME-completeness of the problem. They also introduce a new algorithm for model-checking alternation-free modal  $\mu$ -calculus, and identify a fragment for which model checking is PSPACE-complete. The algorithm for LTL model checking is implemented and experimentally evaluated in [75]. The authors of [78] use a similar symbolic approach, representing stack configurations with finite-state automata to obtain a model checking algorithm for LTL, and sketching one for CTL\*.

Another way of representing PDS's is to view their state-space as an infinite graph, whose nodes are stack contents and arcs are transitions between configurations, and can be represented finitely. In this way, *graph automata* are used in [112] to model-check  $\mu$ -calculus on context-free rewrite systems, which are equivalent to PDS's, and prefix-recognizable rewrite systems, which define transitions through regular expressions on stack contents. This work is extended to LTL in [113], and to *global* model checking<sup>2</sup> in [137].

*Module checking*, i.e., the problem of model checking a system by considering its interactions with the environment, is studied for CTL and CTL\* over PDS's in [35].

### 3.1.2 Recursive State Machines

Recursive State Machines (RSMs) [9] have been introduced as a modeling formalism closer to procedural programs. They consist of a set of *boxes*, which can be seen as procedures, and nodes, which represent control locations in the program. Each box has a set of entry (or *call*), internal, and exit (or *return*) nodes. Boxes are used as stack symbols, and nodes as control states: RSM configurations consist of a stack of boxes and one node; transitions that enter (resp. exit) a box push (resp. pop) the topmost box, and the control state changes accordingly. RSMs are not expressively more powerful than PDS's: each PDS is bisimilar to a RSM and vice versa [9]. They rather introduce a certain degree of algorithmic simplification due to the distinction of entry/exit nodes. In fact, reachability and cycle detection can be solved in time  $O(n\theta^2)$ , where  $n$  is the

<sup>1</sup>Context-free processes are called in this way because they are bisimulation-equivalent to context-free grammars, when comparing transition graphs. There exist PDS's which are bisimulation-equivalent to no context-free processes [47]. Notice that here we are only interested in behaviors of PDS's in the bisimulation context, disregarding accepted language families. The fact that a nondeterministic pushdown automaton can be always transformed into one with only one state accepting the same language is irrelevant.

<sup>2</sup>While *local* model checking aims at verifying that one single initial state satisfies a given property, *global* model checking means finding the set of states that satisfy a property.

number of nodes and  $\theta$  is the maximum between the number of entry and exit nodes. The complexity for the same problems is cubic for PDS's [75], which coincides with that of RSMs if  $\theta = O(n)$ . Algorithms for model checking LTL and CTL\* are derived in [9], while [7] studies subclasses of RSMs for which reachability and cycle detection are linear. More practical algorithms for reachability and cycle-detection for RSMs are given in [10]. RSMs can be seen as a generalization of Hierarchical State Machines, for which model checking is studied in [6].

### 3.1.3 Boolean Programs

*Boolean programs* were introduced with the SLAM toolkit [19], which was developed by Microsoft to model-check control-flow dominated properties of Windows device drivers. Boolean programs can be seen as RSMs augmented with predicates over Boolean variables, which make them more succinct than RSMs. For this reason, they are also called *Extended Recursive State Machines* [89]. C programs are modeled as Boolean programs by abstracting their expressions as Boolean predicates (cf. *predicate abstraction* [104]). Such abstractions can be refined automatically until they are fine-grained enough to prove (or disprove) the desired property, through a process called Counterexample-Guided Abstraction Refinement (CEGAR) [61]. The most important verification problems on Boolean programs are studied in [89]: reachability analysis and LTL model checking are EXPTIME-complete even in model size, while CTL and CTL\* model checking are 2EXPTIME-complete. The same work introduces practically relevant subclasses of Boolean programs for which such problems are in more tractable complexity classes. Several practically efficient ways of verifying Boolean programs have been devised [18, 22, 74, 116].

## 3.2 Context-free models and context-free specifications

Logic formalisms used for specifications in the previous section are, however, restricted to regular-language properties. In particular,  $\mu$ -calculus expresses all regular properties, as it is equivalent to monadic second-order logic [136]. LTL instead corresponds to the first-order definable fragment of regular languages (called *star-free* languages) [106].

While model checking the whole class of context-free specifications is undecidable [99], the problem is decidable for selected fragments thereof. In the rest of this section we review works that address this challenge.

### 3.2.1 Process Algebra

Non-regular behaviors arise from the *Algebra of Communicating Processes (ACP)* [25], a *process algebra* introduced to formalize concurrent processes. ACP expresses choice, sequencing, concurrency and communication among processes in an axiomatic framework. Its fragment called *Basic Process Algebra (BPA)* generates context-free traces [16].

A variant of LTL based on *Presburger arithmetic* was introduced [28] to express counting specifications concerning the number of occurrences of a certain event. It contains a modality that associates a state formula to an integer variable, which is incremented every time the formula is true in a state. For example, it can be used to

require that, in a messaging protocol, the number of requests is the same as the number of answers. The resulting logic, called PLTL, is capable of expressing context-free properties (such as well-balancing of parentheses), but also context-sensitive ones. The model-checking problem for PLTL is, however, undecidable. Verification is decidable for its *positive* fragment  $\text{PLTL}^+$ , which imposes syntactic restrictions on PLTL:  $\text{PLTL}^+$  formulas cannot contain concatenated until operators (i.e., an until cannot appear in the right-hand-side of another until), and formulas such as  $\diamond \square \varphi$  are not expressible. Satisfiability is instead undecidable for both logics. Different fragments of PLTL are presented in [27], in which decidability of model checking is studied for PDS's and *Petri nets*, besides ACP.

PCTL, the branching-time dual of PLTL, is presented in [29]. The results obtained for PLTL are replicated for PCTL: while model checking ACP processes is undecidable for PCTL, it is decidable for its positive fragment  $\text{PCTL}^+$  on *guarded* ACP processes.

### 3.2.2 Approaches that specify properties on stack contents

A formalism for specifying *stack inspection* properties on Java programs is presented in [102], and is motivated by a security framework based on checkpoints that had been introduced in the Java Development Kit at the time. Such properties deal with the sequence of procedures present in the program's stack at any moment, yielding requirements such as "a privileged procedure A cannot be called if another procedure B is present in the stack", and more.

[76] develops model checking for LTL with regular valuations on PDS's, i.e. with atomic propositions that identify a regular language of stack contents, and prove an EXPTIME lower bound. This formalism can express non-regular properties such as stack inspection, but it is more general than previous approaches because it contains the whole LTL.

The problems of determining stack boundedness (whether the stack size remains below a given threshold throughout the execution) and maximum stack size for different classes of *interrupt-driven programs* are studied in [48], yielding complexity bounds from polynomial to PSPACE-hard.

### 3.2.3 Other Approaches

Nondeterministic pushdown parity tree automata are used in [114] to express specifications. Due to their great expressive power, model checking is undecidable on context-free models, but decidable on finite-state ones (in time exponential both in model and specification size).

Propositional Dynamic Logic has also been extended to express some limited classes of context-free languages [92].

### 3.2.4 Visibly Pushdown Languages and Nested Words

A line of research that lead to a more complete framework for expressing context-free specifications was initiated by R. Alur and P. Madhusudan with the introduction of CaRet in [8].

### 3.2.4.1 CaRet

CaRet is a superset of LTL, and includes next and until modalities that interact explicitly with the structure of procedural programs. Given a set of atomic propositions  $AP$ , CaRet's syntax is the following:

$$\varphi ::= a \mid \varphi \vee \varphi \mid \neg \varphi \mid \bigcirc \varphi \mid \varphi \mathcal{U} \varphi \mid \bigcirc^a \varphi \mid \varphi \mathcal{U}^a \varphi \mid \bigcirc^- \varphi \mid \varphi \mathcal{U}^- \varphi$$

where  $a \in AP$ . The semantics of CaRet is based on linear words representing program execution traces, in which each position is either a function *call*, a *return*, or an *internal* position (which represents any instruction that is not a procedure call or return). A matching relation that links each call to the return of the same function frame arises from such partitioning.

The propositional operators and  $\bigcirc$  and  $\mathcal{U}$  have the same meaning as in LTL, with the only addition that **call**, **ret** and **int** can be used as atomic propositions to distinguish the three kinds of positions. Operator  $\bigcirc^a \varphi$  requires  $\varphi$  to hold in the *abstract* successor of the current position  $i$ , which is  $i + 1$  if  $i$  is not a **call** and  $i + 1$  is not a **ret**, and if  $i$  is a **call** it is its matching **ret** (thus the body of such call is skipped).  $\mathcal{U}^a$  is an until on paths made of abstract successors.  $\bigcirc^- \varphi$  requires  $\varphi$  to hold in the **call** of the function containing the current position, while  $\mathcal{U}^-$  is an until on the path made of nested **calls** whose frames contain the current position (note that these are actually past operators).

Abstract operators are useful for expressing properties limited to one function frame. For example  $\Box(p \implies \top \mathcal{U}^a q)$ , where  $\Box$  is the LTL globally, means that if a request  $p$  is made, the answer  $q$  is given at some point in the same function frame. *Call* operators can be used to express stack inspection properties:

$$\Box(\text{call} \wedge p_A \implies \neg p_C \mathcal{U}^- p_B)$$

means that “a module  $A$  should be invoked only within the context of a module  $B$ , with no intervening call to an overriding module  $C$ ”.

[8] also gives a procedure for model-checking CaRet against RSMs, which has complexity polynomial in RSM size and exponential in formula size, and shows that model checking is EXPTIME-complete.

### 3.2.4.2 Visibly Pushdown Languages

*Visibly Pushdown Languages (VPLs)* are the language class that arises from this way of modeling execution traces [4]. VPLs are a strict subset of Deterministic Context-Free Languages, and are very similar to R. McNaughton's *Parenthesis Grammars* [128], where **calls** correspond to open parentheses and **rets** to closed ones, with the addition that VPLs allow for unmatched **calls** and **rets** respectively at the end and at the beginning of a word. We define them through their accepting automata.

**Definition 3.2** (Visibly Pushdown Automaton (VPA) [4]). Let  $\Sigma$  be a finite alphabet, and  $\Sigma_c, \Sigma_r, \Sigma_{int}$  be three disjoint sets such that  $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_{int}$ . A VPA is a tuple  $\mathcal{A} = (Q, I, \Gamma, \delta, F)$  where  $Q$  is a finite set of states,  $I \subseteq Q$  is the set of initial states,  $F \subseteq Q$  is the set of final states,  $\Gamma$  is a stack alphabet with a distinguished member  $\perp \in \Gamma$ , and

$$\delta \subseteq ((Q \times \Sigma_c) \times (Q \times (\Gamma \setminus \{\perp\}))) \cup ((Q \times \Sigma_r) \times (Q \times \Gamma)) \cup ((Q \times \Sigma_{int}) \times Q)$$

is the transition relation.

VPA stack contents are non-empty words over  $\Gamma$  ending in  $\perp$ , configurations are triples  $(x, q, \sigma)$  where  $x \in \Sigma^*$  is the remaining input word,  $q \in Q$  is the current state, and  $\sigma$  are the current stack contents. A run is a finite sequence of configurations  $(x_0, q_0, \sigma_0) \dots (x_n, q_n, \sigma_n)$  such that for all  $0 \leq i \leq n - 1$  we have  $x_i = a_i x_{i+1}$ , and one of three kinds of moves may occur:

**push:** if  $a_i \in \Sigma_c$ , then  $\sigma_{i+1} = \gamma \sigma_i$  for some  $\gamma \in \Gamma$ , and  $(q_i, a_i, q_{i+1}, \gamma) \in \delta$ ;

**pop:** if  $a_i \in \Sigma_r$ , then  $\sigma_i = \gamma \sigma_{i+1}$  or  $\gamma = \sigma_i = \sigma_{i+1} = \perp$ , and  $(q_i, a_i, q_{i+1}, \gamma) \in \delta$ ;

**internal:** if  $a_i \in \Sigma_{int}$ , then  $\sigma_{i+1} = \sigma_i$  and  $(q_i, a_i, q_{i+1}) \in \delta$ .

A word  $w \in \Sigma^*$  is accepted iff there is a run starting in a configuration  $(w, q_0, \perp)$ , for some  $q_0 \in I$ , and ending in  $(\varepsilon, q_f, \gamma)$  for some  $q_f \in F$  and stack contents  $\gamma$ . The language accepted by  $\mathcal{A}$  is the set of strings accepted by  $\mathcal{A}$ .

The main peculiarity of VPAs is that the kind of move (and hence the stack behavior) is completely determined by the current symbol and the alphabet partitioning. Moreover, there are no  $\varepsilon$ -moves, i.e. each move reads an input symbol. For this reason we call VPAs *real-time*. It is also easy to see that the stack of a VPA mimics the one of a procedural programs with respect to function calls and returns, but it has no way of simulating single events that pop multiple stack symbols, such as exceptions.

VPAs had actually already been introduced as *input-driven* pushdown automata in [130], which however mostly studies them for language recognition. In [4], other properties are studied: VPLs sharing the same alphabet partition form a Boolean algebra (they are closed by union, intersection and complement), and are closed by concatenation and Kleene \*. Universality, equivalence and inclusion are decidable and EXPTIME-complete. VPLs are also characterized in terms of MSOL. The same properties are studied for  $\omega$ VPLs, the infinite-word counterpart of VPLs, obtained by adding Büchi acceptance conditions to VPAs.  $\omega$ VPLs form a Boolean algebra, are closed by concatenation between a VPL and an  $\omega$ VPL, and have a MSOL characterization. Universality, equivalence and inclusion remain decidable and EXPTIME-complete. While VPAs are determinizable, their  $\omega$ -counterparts are not. This is unsurprising, because the same happens with  $\omega$ -regular languages (cf. Section 2.2.2.1).

Congruences for VPLs are studied in [11] and, while in general there is no unique minimal VPA for a VPL, VPL subclasses having this property exist [51, 110]. The problem of VPA minimization is NP-complete [88].

### 3.2.4.3 Nested Words

Some early attempts at giving a MSOL characterization to CFLs introduced the idea of a *matching relation* between word positions. A first logic mechanism aimed at “walking through the structure of a context-free sentence” was proposed in [115] and consists in a *matching condition* that relates the two extreme terminals of the right-hand-sides of a context-free grammar in *double Greibach normal form*, i.e. a grammar whose production right-hand-sides exhibit a terminal character at both ends. In a sense, such terminal characters play the role of explicit parentheses. [115] provides a logic language for general CFLs based on such a relation which, however, fails to keep the decidability properties of logics for regular languages due to the lack of closure properties of CFLs.

This matching condition was then resumed to define the algebraic structure behind CaRet, *nested words*, which have been further studied in [5], after being suggested for their possible data-theoretic applications [2].

**Definition 3.3** (Nested Word [5]). A *nested word* over an alphabet  $\Sigma$  is a tuple

$$(w, \mu, \text{call}, \text{ret})$$

where  $w = a_1 \dots a_n$ , with  $a_i \in \Sigma$  for all  $1 \leq i \leq n$ ;  $\mu$  is a binary relation over word positions plus  $\{-\infty, +\infty\}$ , and **call** and **ret** are two sets of word positions, such that for each  $1 \leq i, i', j, j' \leq n$  we have:

- if  $\mu(i, j)$  then **call**( $i$ ), **ret**( $j$ ) and  $i < j$ ;
- if  $\mu(i, j)$  and  $\mu(i, j')$  both hold then  $j = j'$ ; if  $\mu(i, j)$  and  $\mu(i', j)$  both hold then  $i = i'$  ( $\mu$  is one-to-one);
- if  $i \leq j$ , **call**( $i$ ) and **ret**( $j$ ), then either  $\mu(i, k)$  or  $\mu(k, j)$  for some  $i \leq k \leq j$ .

If  $\mu(i, j)$  then  $i$  is the *matching call* of  $j$ , which is the *matching return* of  $i$ .

$-\infty$  and  $+\infty$  are added to model *pending edges*, and we consider  $-\infty < i < +\infty$  for all  $1 \leq i \leq n$ .

A *nested  $\omega$ -word* is defined in the same way, except  $w$  is an  $\omega$ -word.

Regular languages of nested words are equivalent to VPLs [5], and thus they inherit all their properties. They have been characterized in terms of (non)deterministic Nested Words Automata (NWA) and MSOL for both finite and  $\omega$ -words, and retain Boolean and concatenation closure properties and decidability of universality, equivalence and inclusion, which are EXPTIME-complete.

#### 3.2.4.4 Nested Words Temporal Logic and Expressive Completeness

The expressive power of temporal logic on nested words was investigated in [12]. While the stance of CaRet with respect to FOL remains unknown, the authors introduce several kinds of linear-time temporal logics that are provably equivalent to FOL. Here we briefly present one of them, Nested Words Temporal Logic (NWTL), which will be needed later in the thesis. Its syntax is given by

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

with  $a \in AP$ . The semantics of propositional,  $\bigcirc$  and  $\ominus$  operators is the usual one from LTL. For any nested word  $w$  and position  $i$ ,  $(w, i) \models \text{call}$  iff **call**( $i$ ) and similarly for **ret**;  $(w, i) \models \bigcirc_\mu\varphi$  iff there exists a position  $j$  such that  $\mu(i, j)$  and  $(w, j) \models \varphi$ , while the meaning of  $\ominus_\mu\varphi$  is analogous, but it refers to the past. The until and since operators  $\mathcal{U}^\sigma$  and  $\mathcal{S}^\sigma$  are based on *summary paths*. A summary path between positions  $i$  and  $j$ ,  $i < j$ , is a sequence of positions  $i = i_0 < i_1 < \dots < i_n = j$  such that for any  $0 \leq k < n$  we have

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

An example of a property that can be expressed with NWTL is

$$\Box((\text{call} \wedge p \wedge p_A) \implies \bigcirc_\mu q),$$

where  $\Box$  is the LTL globally operator, which means that “if the pre-condition  $p$  holds when procedure  $A$  is invoked, then if the procedure terminates, the post-condition

q is satisfied upon return” [12]. A useful shortcut  $\varphi \mathcal{U}^{\sigma \downarrow} \psi := (\neg \circ \text{ret} \wedge \varphi) \mathcal{U}^{\sigma} \psi$  called *summary-down* until can be defined to only allow for paths that follow *call* edges (from a call  $i$  to  $i + 1$  if it is not a *ret*), *internal* edges (from  $i$  to  $i + 1$  if  $i$  is not a *call* and  $i + 1$  is not a *ret*), and *nesting* edges (from a *call* to its matching *ret*). This operator only goes down in the nesting structure of words, i.e., towards inner function calls, and remains limited to the current function frame and those of the functions it calls. *Since* and *upward* versions of this operator can also be defined. A property that can be expressed with it is

$$\Box(\text{call} \wedge p_A \implies \neg(\top \mathcal{U}^{\sigma \downarrow} \text{wr})),$$

which means that “a procedure  $A$ , and the procedures it calls, do not write to a variable before it returns” [12]. The ability of this operator to remain constrained to one function body is crucial to achieve expressive completeness and, as we shall see in Part II, a similar remark can be made for logics introduced in this thesis. CaRet does not seem to be capable of expressing properties of this kind, which is why it is conjectured not to be expressively complete [12].

The authors of [12] also give two more temporal logics on nested words, which reach expressive completeness through the *within* operator, one of them by adding it to CaRet. For any formula  $\varphi$ , nested word  $w$  and *call* position  $i$ ,  $(w, i) \models \mathcal{W}\varphi$  iff  $(w', i) \models \varphi$ , where  $w'$  is the subword of  $w$  from  $i$  to its matched *ret*  $j$ . Thus, the *within* operator restricts a formula and all its subformulas to one function frame. The addition of the *within* operator to NWTL makes it exponentially more succinct.

This succinctness result has an impact on model checking complexity: NWTL model checking is EXPTIME-complete—and an automata-theoretic procedure exponential in formula length is given in [12]—while model checking logics with the *within* operator is 2EXPTIME-complete. All such complexities are, however, still polynomial in model size.

Finally, [12] identifies temporal logics equivalent to the two-variable fragment of FOL on nested words, i.e., the set of FO formulas using only two distinct variable names. NWTL and the other expressively complete logics are instead equivalent to the three-variable fragment of FOL, which means that any FO formula on nested words can be re-written by using only three distinct variables.

### 3.2.4.5 More developments on VPLs and Nested Words

Nested words have been extended to the branching-time model in [13], which presents NT- $\mu$ , a version of  $\mu$ -calculus on *nested trees*. Model checking of NT- $\mu$  formulas is EXPTIME-complete, while satisfiability is undecidable.

*Visibly Linear Temporal Logic (VLTL)* [33] is a logic that captures the whole class of VPLs, as opposed to the FO-expressible fragment covered by NWTL. VLTL features operators that embed *Visibly Rational Expressions* [32], the VPL version of *regular expressions*, in LTL (similarly to *Regular Linear Temporal Logic* [148], which uses plain regular expressions). VLTL model checking, which is based on a class of two-way alternating automata for VPL [31], surprisingly retains EXPTIME-completeness.

*ConCaRet*, a variant of CaRet with modalities for expressing properties on concurrent programs, is presented in [34]. Its model checking on RSMs augmented with a restricted form of concurrency is EXPTIME-complete.

Two timed versions of CaRet are studied in [37]. One of them is based on *event-clocks* and its satisfiability and model checking problems are EXPTIME-complete. Its model checking algorithm uses VPAs augmented with event-clocks [36]. The other

one is an extension of *Metric Temporal Logic*, and its satisfiability is undecidable even on finite words, although [37] also presents a decidable fragment thereof. Moreover, an extension of *Interval Temporal Logic* for expressing branching-time visibly push-down requirements is presented in [38].

Weighted MSOL for nested words is studied in [127].

*Colored Nested Words* [3] enrich nested words with colors in order to overcome the limitation of their nesting relation to be one-to-one. Since their nesting relation can be many-to-one, they can represent events such as exceptions in procedural programs. This limitation is also addressed by the work presented in this thesis. Notice, however, that our work is based on OPLs, that are more expressive than colored nested words, because they can express one-to-many relations too.

#### 3.2.4.6 Tools

The literature on tools for nested words is not as rich as the theoretical one. Tools and libraries such as VPAlib [133], VPAchecker [154], OpenNWA [69] and SymbolicAutomata [68] only implement operations such as union, intersection, universality/inclusion/emptiness check for Visibly Pushdown or Nested Word Automata, but have no model checking capabilities.

PAL [49] uses nested-word based monitors to express program specifications, and a tool based on BLAST [94] implements its runtime monitoring and model checking. PAL follows the paradigm of program monitors, and is not—strictly speaking—a temporal logic. PTCaRet [146] is a past version of CaRet, and its runtime monitoring has been implemented in JavaMOP [50].

[134, 135] describe a tool for model checking programs against CaRet specifications. Since its purpose is malware detection, it targets program binaries directly by modeling them as PDS's. Unfortunately, this tool does not seem to be available online.

To the best of our knowledge POMC, the tool we present in this thesis, is the only publicly-available tool for model-checking temporal logics capable of expressing context-free properties.



## Chapter 4

# Operator Precedence Languages

Operator Precedence Languages (OPLs) were originally introduced in the context of efficient programming-language parsing. Hence, they were defined through their generating grammars [80], *Operator Precedence Grammars (OPGs)*, which are a special class of Context-Free Grammars (CFGs) for which it is possible to generate efficient deterministic bottom-up parsers. Their algebraic and closure properties [64, 66] recently revived interest in OPLs, leading to their investigation from the logic and automata-theoretic point of view [119].

In Section 4.1, we first present OPLs through their historical grammar-based characterization, and then through the automata-theoretic one, which will be used in the rest of the thesis; in Section 4.2 we extend such definitions to infinite words; in Section 4.3 we illustrate one possible practical application of OPLs through an example.

### 4.1 Operator Precedence Languages on finite words

In the following, by  $\varepsilon$  we denote the empty string; given a set of characters  $\Gamma$ , we denote by  $\Gamma^*$  the set of all finite strings with characters in  $\Gamma$ , and  $\Gamma^+ = \Gamma^* \setminus \{\varepsilon\}$ .

**Definition 4.1** (Context-Free Grammar (CFG)). A CFG is a tuple  $(V, \Sigma, P, S)$  where  $V$  and  $\Sigma$  are finite sets of, respectively, *non-terminal* and *terminal* symbols such that  $V \cap \Sigma = \emptyset$ ;  $P$  is a finite set of *production rules* of the form  $X \rightarrow \alpha$ , where  $X \in V$  is the left-hand side (lhs) and  $\alpha$  is the right-hand side (rhs), a (possibly empty) string of symbols from  $V \cup \Sigma$ ; and  $S \in V$  is called the *axiom*.

A grammar  $G = (V, \Sigma, P, S)$  generates a language by *deriving* its strings. A string  $\beta$  can be derived from  $\alpha$  in one step, written  $\alpha \Rightarrow_G \beta$ , iff  $\alpha = \alpha_1 X \alpha_2$ , with  $X \in V$ ,  $\beta = \alpha_1 \beta' \alpha_2$ , and  $X \rightarrow \beta' \in P$ . The language generated by  $G$  is  $L_G = \{x \in \Sigma^* \mid S \Rightarrow_G^* x\}$ , and  $\Rightarrow_G^*$  is the reflexive and transitive closure of  $\Rightarrow_G$ .

**Definition 4.2.** A production rule is in *operator form* if its rhs has no consecutive non-terminals. An *operator grammar* only contains rules in operator form.

Every CFG can be transformed into an equivalent one in operator form of size polynomial in the original. In particular, if  $(V, \Sigma, P, S)$  is the original grammar, and  $k$



named *yields precedence*, *equal in precedence*, and *takes precedence*. They graphically resemble the traditional arithmetic relations but do not share their typical ordering and equivalence properties. We kept them for historical reasons (they date back to the original introduction of OPLs by R. W. Floyd), but we advise the reader not to be confused by the similarity.

Intuitively, given two input characters  $a, b$  belonging to a grammar's *terminal alphabet* separated by at most one non-terminal,  $a < b$  iff in some grammar derivation  $b$  is the first terminal character of a grammar's rhs following  $a$  whether the grammar's rule contains a non-terminal character before  $b$  or not;  $a \doteq b$  iff  $a$  and  $b$  occur consecutively in some rhs, possibly separated by one non-terminal;  $a > b$  iff  $a$  is the last terminal in a rhs—whether followed or not by a non-terminal—, and  $b$  follows that rhs in some derivation. Formally, PRs are defined as follows:

**Definition 4.4** (Precedence Relations). Let  $G = (V, \Sigma, P, S)$  be an operator grammar, and  $A \in V$ . Its left and right terminal sets are:

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

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

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

- $a < b$  iff  $(A \rightarrow \alpha aD\beta) \in P$  for some  $D \in V$  such that  $b \in \mathcal{L}_G(D)$ ;
- $a \doteq b$  iff  $(A \rightarrow \alpha aBb\beta) \in P$  for some  $B \in V \cup \{\varepsilon\}$ ;
- $a > b$  iff  $(A \rightarrow \alpha Db\beta) \in P$  for some  $D \in V$  such that  $a \in \mathcal{R}_G(D)$ .

Notice that, in the definition, non-terminals are treated in the same way as the empty string: for this reasons we also say that non-terminal characters are “transparent” in OPL parsing.

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

**Definition 4.5** (Operator Precedence Language (OPL)). A grammar  $G$  is an *operator precedence—or Floyd—grammar* (OPG) iff at most one PR holds between any pair of terminals in  $\Sigma$ . Formally, for any  $a, b \in \Sigma$  if  $a \pi_1 b$  and  $a \pi_2 b$  then  $\pi_1 = \pi_2$  (with  $\pi_1, \pi_2 \in \{<, \doteq, >\}$ ). An *operator precedence language* is any language generated by an OPG.

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

**Definition 4.6** (Operator Precedence Matrix (OPM)). An OPM  $M$  over  $\Sigma$  is a partial function  $(\Sigma \cup \{\#\})^2 \rightarrow \{<, \doteq, >\}$ , that, for each ordered pair  $(a, b)$ , defines the PR  $M(a, b)$  holding between  $a$  and  $b$ . If the function is total we say that  $M$  is *complete*. We call the pair  $(\Sigma, M)$  an *operator precedence alphabet*.

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

Note that, in the literature, OPMs may be defined containing multiple PRs in each cell; here we only consider those containing at most one, which are generated by OPLs. Also, we always define  $\#$  to yield precedence to all symbols, if not otherwise specified (as opposed to it yielding precedence to some terminals only).

	call	ret	han	exc
call	<	≐	<	>
ret	>	>	>	>
han	<	>	<	≐
exc	>	>	>	>

Figure 4.2: The OPM  $M_{\text{call}}$ .

0	# < call < han < call < call < <u>call</u> > exc > call ≐ ret > call ≐ ret > ret > #
1	# < call < han < call < <u>call</u> $N$ > exc > call ≐ ret > call ≐ ret > ret > #
2	# < call < han < <u>call</u> $N$ > exc > call ≐ ret > call ≐ ret > ret > #
3	# < call < <u>han</u> ≐ $N$ <u>exc</u> > call ≐ ret > call ≐ ret > ret > #
4	# < call < $N$ <u>call</u> ≐ <u>ret</u> > call ≐ ret > ret > #
5	# < call < $N$ <u>call</u> ≐ <u>ret</u> > ret > #
6	# < <u>call</u> ≐ $N$ <u>ret</u> > #
7	# ≐ $N$ #

Figure 4.3: The sequence of bottom-up reductions during the parsing of  $w_{ex}$ .

*Example 4.3* (continuing from p. 48). Let us list all left and right terminal sets for  $G_{\text{call}}$ :

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

The OPM associated to  $G_{\text{call}}$  is reported in Figure 4.2. In the matrix, the element in row  $i$  and column  $j$  is the PR between the symbol labeling row  $i$  and the one labeling column  $j$ .

The OPM can be used to derive the structure given to a string by an OPG through the operator precedence parsing algorithm. We show how it works by means of the following example, which provides a first intuition of how a set of *unique* PRs drives the parsing of a string of terminal characters in a deterministic way.

*Example 4.3* (continuing from p. 50). Let us see how OPM  $M_{\text{call}}$  drives the construction of the unique ST associated to a string on the alphabet  $\Sigma_{\text{call}}$  through a bottom-up parsing algorithm. The shape of the obtained ST will depend only on the OPM and not on the particular grammar exhibiting it. Consider the sample word  $w_{ex}$ . First, we add the delimiter # at its boundaries and write all precedence relations between consecutive characters, according to  $M_{\text{call}}$ . The result is row 0 of Figure 4.3.

Then, select all innermost patterns of the form  $a < c_1 \doteq \dots \doteq c_\ell > b$ . In row 0 of Figure 4.3 the only such pattern is the underscored **call** enclosed within the pair ( $<$ ,  $>$ ). This means that the ST we are going to build, if it exists, must contain an internal node with the terminal character **call** as its only child. We mark this fact by replacing the pattern  $<\underline{\text{call}}>$  with a dummy non-terminal character, say  $N$ —i.e., we *reduce call* to  $N$ . The result is row 1 of Figure 4.3.

Next, we apply the same labeling to row 1 by simply ignoring the presence of the dummy symbol  $N$  and we find a new candidate for reduction, namely the pattern  $<\underline{\text{call}} N >$ . Notice that there is no ambiguity on whether  $N$  should be considered in a rhs together with the underscored **call**, or the following **exc**: if we reduced just **call** and replaced it by a new  $N$  we would produce two adjacent non-terminals, which is impossible because the ST must be generated by an operator grammar.

The reduction of row 2 is similar, so we come to row 3. This time the terminals to be reduced, again underscored, are two, with an  $\doteq$  and an  $N$  in between. This means

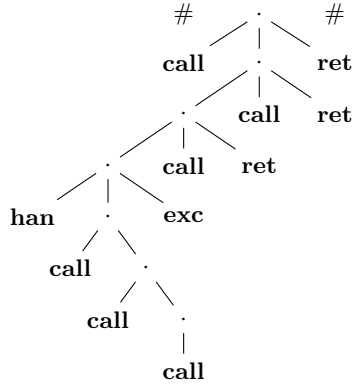


Figure 4.4: The ST corresponding to word  $w_{ex}$ . Dots represent non-terminals.

that they embrace a subtree of the ST whose root is the node represented by  $N$ . By executing the reduction leading from row 3 to 4 we produce a new  $N$  immediately to the left of a **call** which is matched by an equal in precedence **ret**. Then, the procedure is repeated until the final row 7 is obtained, where by convention we state the  $\doteq$  relation between the two delimiters.

Given that each reduction applied in Figure 4.3 corresponds to a grammar derivation step and to the expansion of an internal node of the ST, it is immediate to realize that the ST of  $w_{ex}$  is the one depicted in Figure 4.4, where internal node labels have been omitted. As a side remark we mention that, in general, it may happen that in the same string there are several patterns ready to be reduced; this could enable the implementation of parallel parsing algorithms (see e.g., [21]) which however is not an issue of interest in this thesis.

*Remark 4.7.* The ST of Figure 4.4 shows the main difference between various parenthesis-like language classes, such as VPLs, and OPLs: whereas in the former ones every open parenthesis is consumed by the only corresponding closed one<sup>1</sup>, in our example a **call** can be matched by the appropriate **ret** but also “aborted” by an **exc** which in turn aborts all pending **calls** until its corresponding **han**—if any—is found.

In the example, we derived only a *stencil* of the ST of  $w_{ex}$ . To show how this process can be extended to the parsing of a grammar, we first introduce a normal form for OPGs.

**Definition 4.8** (Fischer Normal Form [79]). An OPG is in Fischer normal form iff

- it is invertible (no two rules share the same rhs),
- none of its rules has the form  $A \rightarrow \varepsilon$  or  $A \rightarrow B$ , where  $A$  is any non-terminal except the axiom, and  $B$  is any non-terminal,
- none of its rules has a rhs containing the axiom.

Any OPG can be automatically transformed into one in Fischer normal form [79]. For an OPG  $G = (V, \Sigma, P, S)$ , such that  $k$  is the maximum length of its rules’ right-hand sides, an equivalent one in Fischer normal form can be built with  $\mathcal{P}(V)$  as the non-terminal set, and with  $O(|P| \cdot 2^{|N| \cdot \lceil \frac{k}{2} \rceil})$  production rules [93].

<sup>1</sup>To be precise, VPLs allow for unmatched closed parentheses but only at the beginning of a string and unmatched open ones at the end.

The operator precedence parsing algorithm reconstructs the syntax tree of a string according to the structure given by an OPG in Fischer normal form, given the corresponding OPM.

*Example 4.3* (continuing from p. 50). Notice how the structure of the ST of Figure 4.4 is similar, but not identical to the one of Figure 4.1: it lacks  $\varepsilon$ -production rules. Indeed, grammar  $G_{\text{call}}$  is not in Fischer normal form, as it contains rules such as  $E \rightarrow \varepsilon$ . However, we can transform it to  $G'_{\text{call}} = (\{S, S', E\}, \Sigma_{\text{call}}, P'_{\text{call}}, S')$ , in Fischer normal form, with the following production rules:

$$\begin{array}{llll}
S' \rightarrow S & S \rightarrow S \text{ call } S \text{ ret} & S \rightarrow S \text{ han } E \text{ exc} & E \rightarrow S \text{ call } E \\
S' \rightarrow \varepsilon & S \rightarrow \text{call } S \text{ ret} & S \rightarrow \text{han } E \text{ exc} & E \rightarrow \text{call } E \\
& S \rightarrow S \text{ call ret} & S \rightarrow S \text{ han exc} & E \rightarrow S \text{ call} \\
& S \rightarrow \text{call ret} & S \rightarrow \text{han exc} & E \rightarrow \text{call}
\end{array}$$

The ST associated to string  $w_{ex}$  by this grammar is isomorphic to the one of Figure 4.4 (excluding the axiom  $S'$ , which is needed to include  $\varepsilon$  in the language). Since the rules in  $G'_{\text{call}}$  are invertible, it is easy to replace each dot in the figure with the appropriate non-terminal, thus obtaining a complete ST of the string generated by the grammar.

In the parsing example, we performed reductions by recognizing patterns of the form  $a < c_1 \doteq \dots \doteq c_\ell > b$ . We call them *chains*, and we formalize them as follows:

**Definition 4.9.** A *simple chain*  ${}^{c_0}[c_1c_2 \dots c_\ell]^{c_{\ell+1}}$  is a string  $c_0c_1c_2 \dots c_\ell c_{\ell+1}$ , such that:  $c_0, c_{\ell+1} \in \Sigma \cup \{\#\}$ ,  $c_i \in \Sigma$  for every  $i = 1, 2, \dots, \ell$  ( $\ell \geq 1$ ), and  $c_0 < c_1 \doteq c_2 \dots c_{\ell-1} \doteq c_\ell > c_{\ell+1}$ .

A *composed chain* is a string  $c_0s_0c_1s_1c_2 \dots c_\ell s_\ell c_{\ell+1}$ , where  ${}^{c_0}[c_1c_2 \dots c_\ell]^{c_{\ell+1}}$  is a simple chain, and  $s_i \in \Sigma^*$  is either the empty string or is such that  ${}^{c_i}[s_i]^{c_{i+1}}$  is a chain (simple or composed), for every  $i = 0, 1, \dots, \ell$  ( $\ell \geq 1$ ). Such a composed chain will be written as  ${}^{c_0}[s_0c_1s_1c_2 \dots c_\ell s_\ell]^{c_{\ell+1}}$ .

In a chain, simple or composed,  $c_0$  (resp.  $c_{\ell+1}$ ) is called its *left* (resp. *right*) *context*; all terminals between them are called its *body*.

A finite word  $w$  over  $\Sigma$  is *compatible* with an OPM  $M$  iff for each pair of letters  $c, d$ , consecutive in  $w$ ,  $M(c, d)$  is defined and, for each substring  $x$  of  $\#w\#$  which is a chain of the form  ${}^a[y]^b$ ,  $M(a, b)$  is defined. For a given operator precedence alphabet  $(\Sigma, M)$  the set of all words compatible with  $M$  is called the *universe* of the operator precedence alphabet and is denoted as  $L(\Sigma, M)$ .

*Example 4.3* (continuing from p. 50). The chain below is the chain defined by the OPM  $M_{\text{call}}$  of Figure 4.2 for the word  $w_{ex}$ . It shows the natural isomorphism between STs with unlabeled internal nodes (see Figure 4.4) and chains.

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

Note that, in composed chains, consecutive inner chains are always separated by an input symbol: this happens because OPL strings are generated by grammars in operator normal form.

Thus, an OPM defines a *universe*, called the *max-language*, of strings on the given alphabet that can be parsed according to it and assigns a unique ST—with unlabeled internal nodes—to each one of them. The grammar generating the max-language of an OPM is called its *max-grammar*. Such a universe is the whole  $\Sigma^*$  iff *the OPM is*

*complete*, i.e. it has no empty cells, including those of the implicit row and column referring to the delimiters. OPGs can be used to carve more interesting sub-languages into this universe. For example, while the string `ret call han` is given the structure  $\#[\text{ret}]\text{call}[\text{han}]\#$  by OPM  $M_{\text{call}}$ , it is not part of the language generated by grammars  $G_{\text{call}}$  and  $G'_{\text{call}}$ . Such a string, in fact, would not be meaningful as a program execution trace.

This fact results in OPLs having some interesting properties, typical of regular languages:

**Theorem 4.10** ([64, 66]). *OPLs compatible with a given OPM  $M$  form a Boolean algebra. I.e., given two OPLs  $L_1$  and  $L_2$  compatible with  $M$ , their complements w.r.t. the max-language of  $M$ ,  $L_1 \cup L_2$ , and  $L_1 \cap L_2$  are also OPLs.*

*OPLs are closed under reversal, prefix, suffix, concatenation and Kleene \* operations.*

OPLs present interesting algebraic closure properties even when they do not share the same OPM. It is possible to define a notion of inclusion between OPMs such that, given OPMs  $M_1$  and  $M_2$  both on  $\Sigma$ , we have  $M_1 \subseteq M_2$  iff for each  $a, b \in \Sigma$  either  $M_1(a, b) = M_2(a, b)$  or  $M_1(a, b)$  is undefined. Then, if we consider an OPM  $M$ , the set  $\mathcal{M}(M) = \{M' \subseteq M \mid M' \text{ is an OPM}\}$  is a poset ordered by  $\subseteq$ , and it forms a lattice whose top and bottom elements are respectively  $M$  and the empty OPM. Moreover, even the max-languages and their max-grammars on such OPMs form a lattice isomorphic to  $\mathcal{M}(M)$ , and if  $M$  is complete, its top and bottom elements are respectively  $\Sigma^*$  and  $\emptyset$  [66].

In the early literature about OPLs, such as [66, 80], OPGs sharing a given OPM were used to define restricted languages with respect to the universe defined by the OPM and to investigate their algebraic properties, with applications to grammar inference [65]. Later on, the same has been done by using different formalisms such as pushdown automata, monadic second order logic, and suitable extensions of regular expressions [119, 122]. In this thesis, we mostly use automata, which are typically applied in logics and model checking.

OPLs are recognized by a class of pushdown automata called *Operator Precedence Automata (OPAs)*. The main feature that distinguished OPAs from classical pushdown automata is that their moves are guided by PRs. Indeed, OPAs store terminals in their stack symbols, and always compare the topmost symbol with the next one to be read. If they are in the  $\prec$  relation, they push another symbol; if they are in the  $\doteq$  relation they update the topmost one; and if they are in the  $\succ$  relation, they pop the topmost symbol. More formally:

**Definition 4.11** (Operator Precedence Automaton (OPA)). An OPA is a tuple  $\mathcal{A} = (\Sigma, M, Q, I, F, \delta)$  where:  $(\Sigma, M)$  is an operator precedence alphabet,  $Q$  is a finite set of states (disjoint from  $\Sigma$ ),  $I \subseteq Q$  is the set of initial states,  $F \subseteq Q$  is the set of final states,  $\delta \subseteq Q \times (\Sigma \cup Q) \times Q$  is the transition relation, which is the union of the three disjoint relations  $\delta_{\text{shift}} \subseteq Q \times \Sigma \times Q$ ,  $\delta_{\text{push}} \subseteq Q \times \Sigma \times Q$ , and  $\delta_{\text{pop}} \subseteq Q \times Q \times Q$ .

An OPA is deterministic iff  $I$  is a singleton, and all three components of  $\delta$  are possibly partial-functions.

To define the semantics of OPAs, we need some new notations. Letters  $p, q, p_i, q_i, \dots$  denote states in  $Q$ . We use  $q_0 \xrightarrow{a} q_1$  for  $(q_0, a, q_1) \in \delta_{\text{push}}$ ,  $q_0 \xrightarrow{-a} q_1$  for  $(q_0, a, q_1) \in \delta_{\text{shift}}$ ,  $q_0 \xrightarrow{q_2} q_1$  for  $(q_0, q_2, q_1) \in \delta_{\text{pop}}$ , and  $q_0 \xrightarrow{w} q_1$ , if the automaton can read  $w \in \Sigma^*$  going from  $q_0$  to  $q_1$ . Let  $\Gamma$  be  $\Sigma \times Q$  and  $\Gamma' = \Gamma \cup \{\perp\}$  be the *stack alphabet*; we denote symbols in  $\Gamma$  as  $[a, q]$ . We set  $\text{smb}([a, q]) = a$ ,  $\text{smb}(\perp) = \#$ ,

and  $st([a, q]) = q$ . For a stack content  $\gamma = \gamma_n \dots \gamma_1 \perp$ , with  $\gamma_i \in \Gamma$ ,  $n \geq 0$ , we set  $smb(\gamma) = smb(\gamma_n)$  if  $n \geq 1$ , and  $smb(\gamma) = \#$  if  $n = 0$ .

A *configuration* of an OPA is a triple  $c = \langle w, q, \gamma \rangle$ , where  $w \in \Sigma^* \#$ ,  $q \in Q$ , and  $\gamma \in \Gamma^* \perp$ . A *computation* or *run* is a finite sequence  $c_0 \vdash c_1 \vdash \dots \vdash c_n$  of *moves* or *transitions*  $c_i \vdash c_{i+1}$ . There are three kinds of moves, depending on the PR between the symbol on top of the stack and the next input symbol:

**push move** if  $smb(\gamma) \leq a$  then  $\langle ax, p, \gamma \rangle \vdash \langle x, q, [a, p]\gamma \rangle$ , with  $(p, a, q) \in \delta_{push}$ ;

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

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

Shift and pop moves are not performed when the stack contains only  $\perp$ . Push moves put a new element on top of the stack consisting of the input symbol together with the current state of the OPA. Shift moves update the top element of the stack by *changing its input symbol only*. Pop moves remove the element on top of the stack, and update the state of the OPA according to  $\delta_{pop}$  on the basis of the current state and the state in the removed stack symbol. They do not consume the input symbol, which is used only as a look-ahead to establish the  $\succ$  relation. The OPA accepts the language  $L(\mathcal{A}) = \{x \in \Sigma^* \mid \langle x\#, q_I, \perp \rangle \vdash^* \langle \#, q_F, \perp \rangle, q_I \in I, q_F \in F\}$ .

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

**Definition 4.12.** Let  $\mathcal{A}$  be an OPA. We call a *support* for simple chain  ${}^{co}[c_1 c_2 \dots c_\ell]^{c_{\ell+1}}$  any path in  $\mathcal{A}$  of the form  $q_0 \xrightarrow{c_1} q_1 \dashrightarrow \dots \dashrightarrow q_{\ell-1} \xrightarrow{c_\ell} q_\ell \xrightarrow{q_0} q_{\ell+1}$ . The label of the last (and only) pop is exactly  $q_0$ , i.e. the first state of the path; this pop is executed because of relation  $c_\ell \succ c_{\ell+1}$ .

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

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

OPAs and OPGs can be used interchangeably to define OPLs:

**Theorem 4.13** ([119]). *Given an OPG  $G$  on an OP alphabet  $(\Sigma, M)$ , it is possible to build an OPA  $\mathcal{A}_G$  such that  $L(G) = L(\mathcal{A}_G)$ . Given an OPA  $\mathcal{A}$  on  $(\Sigma, M)$ , it is possible to build an OPG  $G_{\mathcal{A}}$  such that  $L(\mathcal{A}) = L(G_{\mathcal{A}})$ .*

*Proof (sketch).* Intuitively,  $\mathcal{A}_G$  performs a push move when it reads the first terminal of one of  $G$ 's rhs's, a shift move when reading a terminal from a continuing rhs, and a pop move when reaching past its end. Then, it nondeterministically guesses the non-terminal corresponding to such rhs according to  $G$ 's production rules (there might be more than one).  $\mathcal{A}_G$  accepts a string if it recognizes  $G$ 's axiom after reading it.

The converse construction faces more difficulties and involves more technicalities than the classical one for general CFLs. Essentially, it requires enumerating all

possible chain supports of  $\mathcal{A}$ , and  $G_{\mathcal{A}}$  is built so that it has a non-terminal for each quadruple made of the first and last states of each support and the left and right contexts of the underlying chain. Then, production rules are added that associate such non-terminals to rhs's structured as the underlying chains.  $\square$

Consider the OPA  $\mathcal{A}(\Sigma, M) = (\Sigma, M, \{q\}, \{q\}, \{q\}, \delta_{max})$  where  $\delta_{max}(q, q) = q$ , and  $\delta_{max}(q, c) = q, \forall c \in \Sigma$ . We call it the *OP Max-Automaton* over  $(\Sigma, M)$ . For a max-automaton, each chain has a support; thus, a max-automaton accepts exactly the universe of the OP alphabet. Since there is a chain  $\# [s] \#$  for any string  $s$  compatible with  $M$ , a string is accepted by  $\mathcal{A}(\Sigma, M)$  iff it is compatible with  $M$ . If  $M$  is complete, the language accepted by  $\mathcal{A}(\Sigma, M)$  is  $\Sigma^*$ . With reference to the OPM  $M_{\text{call}}$  of Figure 4.2, the string **ret call han** is accepted by the max-automaton with structure defined by the chain  $\#[\text{ret}]\text{call}[\text{han}]\#$ .

Of course, the results of Theorem 4.10 hold for OPAs too:

**Theorem 4.14.** *Given two OPAs  $\mathcal{A}_1$  and  $\mathcal{A}_2$  on OP alphabet  $(\Sigma, M)$ , it is possible to effectively build the following OPAs:*

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

*Proof.* Since OPA moves are determined by PRs, two OPAs sharing the same OPM are synchronized while reading the same word, and their stacks have the same size at each step. Thus, constructions typical of finite-state automata can be exploited:  $\mathcal{A}_{\cup}$  can be obtained by taking the component-wise set union of  $\mathcal{A}_1$  and  $\mathcal{A}_2$ , and  $\overline{\mathcal{A}_1}$  by determinizing  $\mathcal{A}_1$  and complementing its final set. The main difference is that OPA determinization has a higher complexity—the determinized OPA has  $2^{O(n^2)}$  states, as opposed to only  $2^n$  for finite-state automata—which, however, happens for VPAs too. We show the construction of  $\mathcal{A}_{\cap}$ , which is based on the product automaton, because it is required for model checking.

Let  $\mathcal{A}_1 = (\Sigma, M, Q_1, I_1, F_1, \delta_1)$  and  $\mathcal{A}_2 = (\Sigma, M, Q_2, I_2, F_2, \delta_2)$ . We define  $\mathcal{A}_{\cap} = (\Sigma, M, Q_1 \times Q_2, I_1 \times I_2, F_1 \times F_2, \delta^{\cap})$  where

- if  $(q_1, a, p_1), (q_2, a, p_2) \in \delta_{push}$  then  $((q_1, q_2), a, (p_1, p_2)) \in \delta_{push}^{\cap}$ ;
- if  $(q_1, a, p_1), (q_2, a, p_2) \in \delta_{shift}$  then  $((q_1, q_2), a, (p_1, p_2)) \in \delta_{shift}^{\cap}$ ;
- if  $(q_1, r_1, p_1), (q_2, r_2, p_2) \in \delta_{pop}$  then  $((q_1, q_2), (r_1, r_2), (p_1, p_2)) \in \delta_{pop}^{\cap}$ .  $\square$

Automata-based proofs of the closure properties of OPLs are much simpler than the analogous ones on grammars [66], especially in the case of concatenation and Kleene \* [64]. This further highlights the benefits of using automata instead of grammars for logical characterizations.

Theorem 4.14 naturally suggests explicit-state model-checking procedures similar to those used with regular languages and temporal logics such as LTL.

In conclusion, given an OP alphabet, the OPM assigns a unique structure to any compatible string in  $\Sigma^*$ ; unlike VPLs, such a structure is not visible in the string, and must be built by means of a non-trivial parsing algorithm. An OPG or an OPA defined on the OP alphabet selects an appropriate subset within its universe. OPAs form a Boolean algebra whose universal element is the max-automaton. The language

classes recognized by deterministic and non-deterministic OPAs coincide. For a more complete description of the OPL family and of its relations with other CFLs we refer the reader to [121].

## 4.2 Operator Precedence $\omega$ -Languages

All definitions regarding OPLs are extended to infinite words in the usual way introduced for  $\omega$ -regular languages [155] and extended to VPLs [4], but with a few distinctions [119]. Recall that, given a set of characters  $\Gamma$ , by  $\Gamma^\omega$  we mean the set of all (countably) infinite words made of characters in  $\Gamma$ .

Of course, OP  $\omega$ -words are not terminated by the delimiter  $\#$ . Given an OP alphabet  $(\Sigma, M)$ , an  $\omega$ -word  $w \in \Sigma^\omega$  is compatible with  $M$  if every finite prefix of  $w$  is compatible with  $M$ .

**Definition 4.15** (Open Chain). An  $\omega$ -word may contain never-ending chains of the form  $c_0 \triangleleft c_1 \doteq c_2 \doteq \dots$ , where the  $\triangleleft$  relation between  $c_0$  and  $c_1$  is never closed by a corresponding  $\triangleright$ . Such chains are called *open chains* and may be simple or composed. A composed open chain may contain both open and closed subchains. Of course, a closed chain cannot contain an open one. A terminal symbol  $a \in \Sigma$  is *pending* if it is part of the body of an open chain and of no closed chains.

OPA variants accepting Operator Precedence  $\omega$ -Languages ( $\omega$ OPLs) can be defined by augmenting Definition 4.11 with Büchi or Muller acceptance conditions. With Büchi acceptance conditions, as we saw in Section 2.2.2, a word is accepted if a state from the final set is visited infinitely often. Automata using Muller acceptance conditions are equipped with a set of sets of final states called its *table*, and the set of states occurring infinitely often must be exactly one of them for an  $\omega$ -word to be accepted. In this work, we only consider Büchi conditions, because they are the ones most frequently used in the model-checking literature. Nevertheless, Muller conditions could be used too, since they are equivalent in expressive power when applied to  $\omega$ OPLs [119].

**Definition 4.16** (Operator Precedence Büchi Automaton ( $\omega$ OPBA)). An  $\omega$ OPBA is a tuple  $\mathcal{A} = (\Sigma, M, Q, I, F, \delta)$ , where  $\Sigma, M, Q, I, F, \delta$  are the same as in Definition 4.11.

The semantics of configurations, moves and infinite runs are defined as for OPAs. For the acceptance condition, let  $\rho$  be a run on an  $\omega$ -word  $w$ . We define

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

as the set of states that occur infinitely often in  $\rho$ .  $\rho$  is successful iff there exists a state  $q_f \in F$  such that  $q_f \in \text{Inf}(\rho)$ . An  $\omega$ OPBA  $\mathcal{A}$  accepts  $w \in \Sigma^\omega$  iff there is a successful run of  $\mathcal{A}$  on  $w$ . The  $\omega$ -language recognized by  $\mathcal{A}$  is

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

Unlike OPAs,  $\omega$ OPBAs do not require the stack to be empty for word acceptance: when reading an open chain, the stack symbol pushed when the first character of the body of its underlying simple chain is read remains into the stack forever; it is at most updated by shift moves.

In our model-checking procedures, we will need a slight variation on  $\omega$ OPBAs:

**Definition 4.17** (Generalized  $\omega$ OPBA). A generalized  $\omega$ OPBA is a tuple  $\mathcal{A} = (\Sigma, M, Q, I, \mathbf{F}, \delta)$ , where  $\Sigma, M, Q, I, \delta$  are the same as in Definition 4.11, and  $\mathbf{F} \subseteq \mathcal{P}(Q)$  is the set of sets of Büchi-final states.

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

Generalized  $\omega$ OPBA can be translated to normal  $\omega$ OPBA polynomially:

**Theorem 4.18.** *Let  $\mathcal{A} = (\Sigma, M, Q, I, \mathbf{F}, \delta)$  be a generalized  $\omega$ OPBA. It is possible to build an  $\omega$ OPBA  $\mathcal{A}'$  with  $|Q| \cdot |\mathbf{F}|$  states such that  $L(\mathcal{A}') = L(\mathcal{A})$ .*

*Proof.* We use the construction based on counters we already saw in the proof of Theorem 2.14. Let  $k = |\mathbf{F}|$ ; we assign a linear ordering to sets in  $\mathbf{F}$ , and call  $F_i$ ,  $0 \leq i \leq k-1$ , the  $i$ -th of such sets. We define  $\mathcal{A}' = (\Sigma, M, Q', I', F', \delta')$  as follows:

- $Q' = Q \times \{0, \dots, k-1\}$ ;
- $I' = I \times \{0\}$ ;
- $F' = F_1 \times \{0\}$ ;
- if  $(q, a, p) \in \delta_{push}$ , then for  $0 \leq i \leq k-1$  we have  $((q, i), a, (p, j)) \in \delta'_{push}$ ;
- if  $(q, a, p) \in \delta_{shift}$ , then for  $0 \leq i \leq k-1$  we have  $((q, i), a, (p, j)) \in \delta'_{shift}$ ;
- if  $(q, r, p) \in \delta_{pop}$ , then for  $0 \leq i, h \leq k-1$  we have  $((q, i), (r, h), (p, j)) \in \delta'_{pop}$ ;

where  $j = i$  if  $q \notin F_i$ , and  $j = i+1 \pmod k$  otherwise.

We recall that, according to this construction,  $\mathcal{A}'$  contains  $k$  different copies of  $\mathcal{A}$ ; it starts by running the first one, and switches to the next one (modulo  $k$ ) each time it finds an accepting state for the current one. Since the accepting states must be from the first copy, each infinite run must necessarily cycle through all of them, so that each accepting condition is satisfied.  $\square$

The translation from simple to generalized  $\omega$ OPBAs is trivial, hence the two classes are equivalent, and enjoy the same closure properties. Indeed, the most important closure properties of OPLs are preserved by  $\omega$ OPLs:

**Theorem 4.19** ([119]). *Given two  $\omega$ OPBAs  $\mathcal{A}_1$  and  $\mathcal{A}_2$  on OP alphabet  $(\Sigma, M)$ , it is possible to effectively build the following  $\omega$ OPBAs:*

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

*Proof.* The construction of  $\mathcal{A}_\cup$  is similar to the finite-word case. Unfortunately, the one for  $\overline{\mathcal{A}_1}$  cannot rely on equivalence between deterministic and nondeterministic automata (which does not hold for  $\omega$ OPBAs), and is much more involved, so we refer the reader to [119]. Instead, we show the construction for  $\mathcal{A}_\cap$ .

Let  $\mathcal{A}_1 = (\Sigma, M, Q_1, I_1, F_1, \delta^1)$  and  $\mathcal{A}_2 = (\Sigma, M, Q_2, I_2, F_2, \delta^2)$ . As in Theorem 4.18,  $\mathcal{A}_\cap$  alternates the execution of  $\mathcal{A}_1$  and  $\mathcal{A}_2$ , making sure a final state of both appears infinitely often. We define  $\mathcal{A}_\cap = (\Sigma, M, Q_\cap, I_\cap, F_\cap, \delta^\cap)$  where

- $Q_\cap = Q_1 \times Q_2 \times \{0, 1\}$ ;
- $I_\cap = I_1 \times I_2 \times \{0\}$ ;
- $F_\cap = F_1 \times Q_2 \times \{0\}$ ;
- if  $(q_1, a, p_1) \in \delta_{push}^1$  and  $(q_2, a, p_2) \in \delta_{push}^2$  then for all  $i \in \{0, 1\}$  we have  $((q_1, q_2, i), a, (p_1, p_2, j)) \in \delta_{push}^\cap$ ;
- if  $(q_1, a, p_1) \in \delta_{shift}^1$  and  $(q_2, a, p_2) \in \delta_{shift}^2$  then for all  $i \in \{0, 1\}$  we have  $((q_1, q_2, i), a, (p_1, p_2, j)) \in \delta_{shift}^\cap$ ;
- if  $(q_1, r_1, p_1) \in \delta_{pop}^1$  and  $(q_2, r_2, p_2) \in \delta_{pop}^2$  then for all  $i, h \in \{0, 1\}$  we have  $((q_1, q_2, i), (r_1, r_2, h), (p_1, p_2, j)) \in \delta_{pop}^\cap$ ;

with  $j = i$  if  $q_{i+1} \notin F_{i+1}$ , and  $j = i + 1 \pmod 2$  otherwise.  $\square$

$\omega$ OPBAs are also closed under concatenation of an OPL with an  $\omega$ OPL. The equivalence between deterministic and nondeterministic automata is lost in the infinite case, which is unsurprising, since it also happens for regular  $\omega$ -languages and  $\omega$ VPLs. A more complete treatment of  $\omega$ OPLs can be found in [119].

### 4.3 Modeling Programs with OPA

In this section, we show how OPAs can naturally model programming languages such as Java and C++. Given a set  $AP$  of atomic propositions describing events and states of the program, we use  $(\mathcal{P}(AP), M_{AP})$  as the OP alphabet. Note that, when they are used as terminal characters, we refer to elements of  $\mathcal{P}(AP)$  with lowercase letters even if they are sets. For convenience, we consider a partitioning of  $AP$  into a set of normal propositional labels (in round font), and *structural labels* (in bold). Structural labels define the OP structure of the word:  $M_{AP}$  is only defined for subsets of  $AP$  containing exactly one structural label, so that given two structural labels  $\mathbf{l}_1, \mathbf{l}_2$ , for any  $a, a', b, b' \in \mathcal{P}(AP)$  s.t.  $\mathbf{l}_1 \in a, a'$  and  $\mathbf{l}_2 \in b, b'$  we have  $M_{AP}(a, b) = M_{AP}(a', b')$ . In this way, it is possible to define an OPM on the entire  $\mathcal{P}(AP)$  by only giving the relations between structural labels, as we did for  $M_{\text{call}}$ .

Figure 4.5 shows how to model a procedural program with an OPA. The OPA simulates the program's behavior with respect to the stack, by expressing its execution traces with four event kinds: **call** (resp. **ret**) marks a procedure call (resp. return), **han** the installation of an exception handler by a try statement, and **exc** an exception being raised. OPM  $M_{\text{call}}$  defines the context-free structure of the word, which is strictly linked with the programming language semantics: the  $\leq$  PR causes nesting (e.g., **calls** can be nested into other **calls**), and the  $\doteq$  PR implies a one-to-one relation, e.g. between a **call** and the **ret** of the same function, and a **han** and the **exc** it catches.

Each OPA state represents a line in the source code. First, procedure  $p_A$  is called by the program loader ( $M_0$ ), and  $[\{\text{call}, p_A\}, M_0]$  is pushed onto the stack, to track the program state before the **call**. Then, the try statement at line  $A_0$  of  $p_A$  installs a handler. All subsequent calls to  $p_B$  and  $p_C$  push new stack symbols on top of the one pushed with **han**.  $p_C$  may only call itself recursively, or throw an exception, but never return normally. This is reflected by **exc** being the only transition leading from state  $C_0$  to the accepting state  $M_r$ , and  $p_B$  and  $p_C$  having no way to a normal **ret**. The OPA has a look-ahead of one input symbol, so when it encounters **exc**, it must

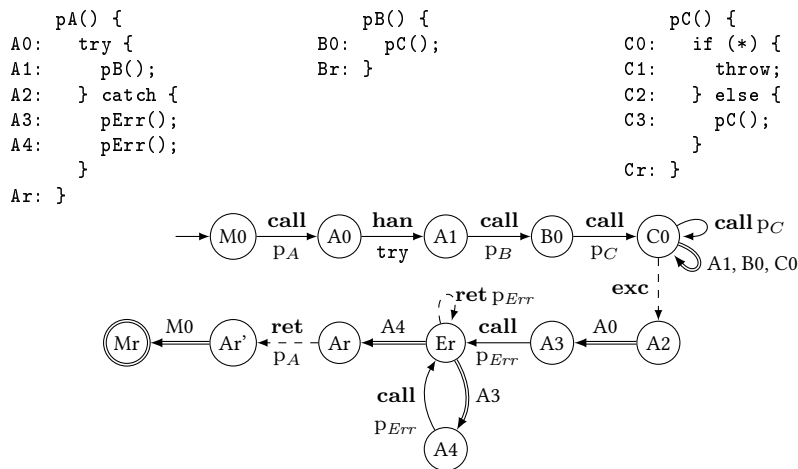


Figure 4.5: Example procedural program (top) and the derived OPA (bottom). Push, shift, pop moves are shown by, resp., solid, dashed and double arrows.

pop all symbols in the stack, corresponding to active function frames, until it finds the one with **han** in it, which cannot be popped because **han**  $\doteq$  **exc**. Notice that such behavior cannot be modeled by VPAs or NWAAs (cf. Section 3.2.4), because they need to read an input symbol for each pop move. Thus, **han** protects the parent function from the exception. Since the state contained in **han**'s stack symbol is A0, the execution resumes in the catch clause of  $p_A$ .  $p_A$  then calls twice the library error-handling function  $p_{Err}$ , which ends regularly both times, and returns. The string of Figure 4.2 is accepted by this OPA.

In this example, we only model the stack behavior for simplicity, but other statements, such as assignments, and other behaviors, such as continuations, could be modeled by a different choice of the OPM, and other aspects of the program's state by appropriate abstractions [104].



## Chapter 5

# Model-Theoretic Background

In this chapter, we give an introduction to the model-theoretic techniques we used for the expressive completeness proof of the logic we present in this thesis.

We start with the seminal work of A. Ehrenfeucht, who introduced a game-theoretic characterization [71] of the definition of elementary equivalence given by R. Fraïssé [82]. The resulting technique, called Ehrenfeucht-Fraïssé (EF) games, has been extensively used to study the expressiveness of first- and second-order logic. EF games are in fact often used to prove that a given property is not first-order definable. They were also used to assess the decidability of logical theories: in particular S. Shelah used them to prove that the theory of monadic second-order logic over linear orderings is decidable [149], providing an alternative to the automata-based proof by J.R. Büchi [42].

The contents of this chapter have been collected mainly from [145, 156]. For a more comprehensive introduction to model theory, we refer the reader to [90, 101].

In the following, we assume some familiarity with the basics of FOL. We refer to FOL as defined in Section 2.2.1, mostly considering the first-order theory of labeled linear orderings in examples. Nevertheless, the techniques presented in this chapter are general enough to be applied to any first-order signature, and we will in fact use them on unranked ordered trees in Chapter 9.

**Notation** We call a *signature* a set of *predicate* symbols  $\Theta = \{R_1, R_2, \dots, R_n\}$ , each one with an arity  $\alpha(R_i)$ ,  $1 \leq i \leq n$ . An *interpretation* or *structure* on  $\Theta$  is a tuple  $\mathcal{A} = (A, R_1^{\mathcal{A}}, \dots, R_n^{\mathcal{A}})$ , where  $A$  is a set called the *universe* or *domain* of  $\mathcal{A}$ , and for  $1 \leq i \leq n$ ,  $R_i^{\mathcal{A}} \subseteq A^{\alpha(R_i)}$  is a relation on  $A$ .

If  $\varphi$  is a FO formula with no free variables on a signature  $\Theta$  and  $\mathcal{A}$  is an interpretation of  $\Theta$ , we write  $\mathcal{A} \models \varphi$  meaning that  $\varphi$  is true on structure  $\mathcal{A}$ , and also say that  $\mathcal{A}$  is a *model* of  $\varphi$ . If  $\varphi(x_1, \dots, x_m)$  is a FO formula with  $m$  free variables on a signature  $\Theta$ ,  $\mathcal{A}$  is an interpretation of  $\Theta$  with domain  $A$ , and  $a_1, \dots, a_m \in A$ , we write  $(\mathcal{A}, a_1, \dots, a_m) \models \varphi(x_1, \dots, x_m)$  meaning that  $\varphi(x_1, \dots, x_m)$  is true on  $\mathcal{A}$  if we assign  $a_i$  to variable  $x_i$  for  $1 \leq i \leq m$ .

### 5.1 Elementary equivalence and its characterizations

We start by saying what we mean by *elementary equivalence*.

**Definition 5.1.** Given two algebraic structures  $\mathcal{A}$  and  $\mathcal{B}$  on the same signature, we say that they are *elementarily* or *first-order equivalent*, written  $\mathcal{A} \equiv \mathcal{B}$  iff for any FO formula  $\varphi$  on that signature we have  $\mathcal{A} \models \varphi$  iff  $\mathcal{B} \models \varphi$ .

The set  $\{\varphi \in \text{FOL} \mid \mathcal{M} \models \varphi\}$  is called the *first-order theory* of a structure  $\mathcal{M}$ .

However, we are actually interested to a weaker form of elementary equivalence, that depends on the nesting of quantifiers we allow in formulas.

**Definition 5.2** (Quantifier Rank). The *quantifier rank* or *depth* of a FOL formula  $\varphi$ , denoted  $\text{qr}(\varphi)$ , is defined by syntactic induction as follows:

- if  $\varphi$  is atomic, then  $\text{qr}(\varphi) = 0$ ;
- if  $\varphi = \neg\psi$  for some  $\psi$ , then  $\text{qr}(\varphi) = \text{qr}(\psi)$ ;
- if  $\varphi$  is of the form  $\psi_1 \vee \psi_2$ ,  $\psi_1 \wedge \psi_2$ ,  $\psi_1 \implies \psi_2$  or  $\psi_1 \iff \psi_2$  for some  $\psi_1$  and  $\psi_2$ , then  $\text{qr}(\varphi) = \max(\text{qr}(\psi_1), \text{qr}(\psi_2))$ ;
- if  $\varphi$  is of the form  $\exists\psi$  or  $\forall\psi$  for some  $\psi$ , then  $\text{qr}(\varphi) = \text{qr}(\psi) + 1$ .

Indeed, we consider equivalence between structures according to formulas of up to a given quantifier rank.

**Definition 5.3** (Rank- $k$  equivalence). Given an integer  $k \geq 0$  and two algebraic structures  $\mathcal{A}$  and  $\mathcal{B}$  on the same signature, we write  $\mathcal{A} \equiv_k \mathcal{B}$  iff for any FOL formula  $\varphi$  such that  $\text{qr}(\varphi) = k$  we have  $\mathcal{A} \models \varphi$  iff  $\mathcal{B} \models \varphi$ .

If  $A$  and  $B$  are respectively the domains of  $\mathcal{A}$  and  $\mathcal{B}$ , and we additionally have the distinguished elements  $a_1, \dots, a_m \in A$  and  $b_1, \dots, b_m \in B$ , we write

$$(\mathcal{A}, a_1, \dots, a_m) \equiv_k (\mathcal{B}, b_1, \dots, b_m)$$

iff for any FOL formula with  $m$  free variables  $\varphi(x_1, \dots, x_m)$  such that  $\text{qr}(\varphi(x_1, \dots, x_m)) = k$  we have

$$(\mathcal{A}, a_1, \dots, a_m) \models \varphi(x_1, \dots, x_m) \quad \text{iff} \quad (\mathcal{B}, b_1, \dots, b_m) \models \varphi(x_1, \dots, x_m).$$

Of course, we have  $\mathcal{A} \equiv \mathcal{B}$  iff  $\mathcal{A} \equiv_k \mathcal{B}$  for all  $k \geq 0$ . Moreover,  $\equiv_k$  is an equivalence relation among structures on the same signature.

This also yields a similarly restricted concept of theory:

**Definition 5.4** (Rank- $k$  type). Given an integer  $k \geq 0$  and a structure  $\mathcal{A}$ , we define the *rank- $k$  type* of  $\mathcal{A}$  as the set

$$\sigma_k(\mathcal{A}) = \{\varphi \in \text{FOL} \mid \mathcal{A} \models \varphi \wedge \text{qr}(\varphi) = k\}.$$

We call  $\Gamma_k$  the set of all rank- $k$  types on a given signature.

If  $A$  is the domain of  $\mathcal{A}$  and  $a_1, \dots, a_m \in A$  are distinguished elements of  $\mathcal{A}$ , we define the *rank- $k$  type* of  $(\mathcal{A}, a_1, \dots, a_m)$  as the set

$$\sigma_k(\mathcal{A}, a_1, \dots, a_m) = \left\{ \varphi(x_1, \dots, x_m) \in \text{FOL} \mid \begin{array}{l} (\mathcal{A}, a_1, \dots, a_m) \models \varphi(x_1, \dots, x_m), \\ \text{qr}(\varphi(x_1, \dots, x_m)) = k \end{array} \right\}.$$

Note that  $\mathcal{A} \equiv_k \mathcal{B}$  iff  $\sigma_k(\mathcal{A}) = \sigma_k(\mathcal{B})$ , so rank- $k$  types can be seen as the equivalence classes of the  $\equiv_k$  relation. As we shall see, the number of such classes for a given  $k$  is actually finite.

Rank- $k$  types can be characterized in a more practical way, actually devised by J. Hintikka [95] before rank- $k$  types were introduced.

**Theorem 5.5** (Hintikka Formulas). *Given an integer  $k \geq 0$ , for any structure  $\mathcal{A}$  there exists a formula  $H_{\mathcal{A}}^k$  such that  $\text{qr}(H_{\mathcal{A}}^k) = k$  and for any structure  $\mathcal{B}$ , we have  $\mathcal{A} \equiv_k \mathcal{B}$  iff  $\mathcal{B} \models H_{\mathcal{A}}^k$ .*

*Proof (sketch).* We actually prove a more general statement by defining formulas

$$H_{(\mathcal{A}, \bar{a})}^k(x_1, \dots, x_m),$$

where  $\bar{a} = (a_1, \dots, a_m) \in A^m$  such that  $(\mathcal{B}, b_1, \dots, b_m) \models H_{(\mathcal{A}, \bar{a})}^k(x_1, \dots, x_m)$  iff  $(\mathcal{A}, a_1, \dots, a_m) \equiv_k (\mathcal{B}, b_1, \dots, b_m)$ . We define such formulas inductively on  $k$ .

If  $k = 0$ , and  $\Phi$  is the set of all possible atomic formulas on  $\mathcal{A}$ 's signature (i.e., all formulas of the form  $x_i = x_j$  and  $R_p(x_{i_1}, \dots, x_{i_{\alpha(R_p)}})$ , where all variable indices range between 1 and  $m$  and  $R_p$  is a predicate symbol), we define

$$H_{(\mathcal{A}, \bar{a})}^0(x_1, \dots, x_m) := \bigwedge_{\substack{\varphi \in \Phi, \\ (\mathcal{A}, \bar{a}) \models \varphi}} \varphi \quad \wedge \quad \bigwedge_{\substack{\varphi \in \Phi, \\ (\mathcal{A}, \bar{a}) \not\models \varphi}} \neg \varphi.$$

Then, we have

$$H_{(\mathcal{A}, \bar{a})}^{k+1}(x_1, \dots, x_m) := \bigwedge \{ \exists y H_{(\mathcal{A}, \bar{a}, a)}^k(x_1, \dots, x_m, y) \mid a \in A \} \\ \wedge \forall y \bigvee \{ H_{(\mathcal{A}, \bar{a}, a)}^k(x_1, \dots, x_m, y) \mid a \in A \}.$$

The fact that the two sets in the formula above are finite can be shown by proving that there are finitely many  $H_{(\mathcal{A}, \bar{a})}^k(x_1, \dots, x_m)$  formulas for any  $\mathcal{A}$ ,  $\bar{a}$  and  $m$  on a given signature, by induction on  $k$ .  $\square$

Thus, given a rank- $k$  type  $\sigma$ , if we take a structure  $\mathcal{A}$  such that  $\sigma_k(\mathcal{A}) = \sigma$ , then for any other structure  $\mathcal{B}$  we have  $\mathcal{B} \models H_{\mathcal{A}}^k$  iff  $\sigma_k(\mathcal{B}) = \sigma$ . Hence, for any rank- $k$  type  $\sigma$  there is a Hintikka formula  $H_{\sigma}^k$  that is true in all structures of rank- $k$  type  $\sigma$ .

Since there are only finitely many Hintikka formulas of quantifier rank  $k$  on a given signature, we can state the following:

**Corollary 5.6.** *There are only finitely many rank- $k$  types on a given signature.*

Actually, in general,

**Corollary 5.7.** *There are only finitely many non-equivalent FO formulas of quantifier rank at most  $k$  on a given signature.*

Thanks to Corollary 5.6, for any formula  $\varphi$  on a given signature with  $\text{qr}(\varphi) = k$  we can collect all rank- $k$  types containing  $\varphi$ , and express  $\varphi$  as the disjunction of all Hintikka formulas characterizing them:

**Lemma 5.8** (Distributive normal form). *Any FO formula  $\varphi$  such that  $\text{qr}(\varphi) = k$  on a given signature can be expressed as*

$$\bar{\varphi} = \bigvee_{\sigma \in \Gamma_{\varphi}} H_{\sigma}^k$$

where  $\Gamma_{\varphi} = \{ \sigma \in \Gamma_k \mid \varphi \in \sigma \}$ .

Lemma 5.8 will be essential for the proofs in Chapter 9.

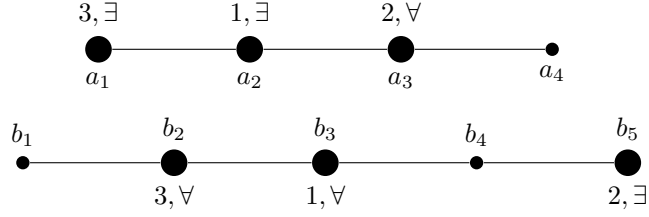


Figure 5.1: A play of game  $G_3(\mathcal{A}, \mathcal{B})$  won by  $\exists$ .  $\mathcal{A}$  is on top, and  $\mathcal{B}$  is below. Position labeled with e.g.  $n, \exists$  has been played by  $\exists$  in round  $n$ .

## 5.2 Ehrenfeucht-Fraïssé Games

We can now introduce EF games as an alternative—and practical—characterization of  $\equiv_k$ -equivalence.

EF games are played between two players,  $\forall$  (the *Spoiler* or *Player I*) and  $\exists$  (the *Duplicator* or *Player II*).<sup>1</sup> A round of an EF game between two structures  $\mathcal{A}$  and  $\mathcal{B}$  with the same signature starts with  $\forall$  picking an element of the domain of either one of  $\mathcal{A}$  and  $\mathcal{B}$ , followed by  $\exists$  answering by picking an element of the other structure.  $\exists$  wins the  $k$ -round game on two structures if the map between the elements picked by  $\forall$  and those picked by  $\exists$  in each of the first  $k$  rounds is a partial isomorphism between  $\mathcal{A}$  and  $\mathcal{B}$ .

*Example 5.9.* Let  $\mathcal{A}$  and  $\mathcal{B}$  be two words (cf. Definition 2.5) with no atomic propositions, respectively of length 4 and 5. Thus,  $\mathcal{A} = (A, \leq_A)$  with  $A = \{a_1, a_2, a_3, a_4\}$  and  $\mathcal{B} = (B, \leq_B)$  with  $B = \{b_1, b_2, b_3, b_4, b_5\}$ . We use  $a_i$  and  $b_i$  instead of natural numbers to emphasize that  $A$  and  $B$  are disjoint, and we have  $a_i \leq_A a_j$  iff  $i \leq j$  and the same for  $\mathcal{B}$ ; we omit the subscript of  $\leq$  when obvious. We call  $a_{i_r}$  and  $b_{j_r}$  the elements chosen by either player respectively from  $A$  and  $B$  at round  $r$ .

Suppose we have a game in which, in the first round,  $\forall$  chooses  $b_3$  and  $\exists$  chooses  $a_2$ ; in the second round  $\forall$  chooses  $a_3$  and  $\exists$  chooses  $b_5$ ; and finally in the third round  $\forall$  chooses  $b_2$  and  $\exists$  chooses  $a_1$ . This game is depicted in Figure 5.1. We have  $(a_{i_1}, a_{i_2}, a_{i_3}) = (a_2, a_3, a_1)$  and  $(b_{j_1}, b_{j_2}, b_{j_3}) = (b_3, b_5, b_2)$  and, thus,  $a_{i_3} \leq a_{i_1} \leq a_{i_2}$  and  $b_{i_3} \leq b_{i_1} \leq b_{i_2}$ . So, the element chosen by  $\exists$  in each round is in the same order as the one chosen by  $\forall$  with respect to the positions previously chosen by either player in the same word. Since the linear orders are the only relations in  $\mathcal{A}$  and  $\mathcal{B}$ , we can say that partial isomorphism is kept throughout the game, and  $\exists$  wins.

Let us consider, instead, the play shown in Figure 5.2. Here  $\forall$  first chooses  $b_3$  and  $\exists$  chooses  $a_2$ ; then  $\forall$  chooses  $b_2$  and  $\exists$  chooses  $a_1$ , and finally  $\forall$  chooses  $b_1$ . Unfortunately, there are no elements of  $A$  before  $a_1$ , which was already chosen, so  $\exists$  is forced to pick  $a_3$  or  $a_4$ , which both break the isomorphism. If  $\exists$  chooses  $a_4$ , we have  $(a_{i_1}, a_{i_2}, a_{i_3}) = (a_2, a_1, a_3)$  and  $(b_{j_1}, b_{j_2}, b_{j_3}) = (b_3, b_2, b_1)$ , so  $a_{i_2} \leq a_{i_1} \leq a_{i_3}$  but  $b_{i_3} \leq b_{i_2} \leq b_{i_1}$ . Thus,  $\forall$  wins this game.

EF games are better formalized as follows:

**Definition 5.10** (Ehrenfeucht-Fraïssé Game). Let  $\mathcal{A} = (A, R_1, \dots, R_p)$  and  $\mathcal{B} = (B, S_1, \dots, S_p)$  be two structures with the same signature, where  $R_i$  and  $S_i$  are re-

<sup>1</sup>We use the notation due to W. Hodges [98], who names the two players  $\forall$  and  $\exists$  after Abelard and (H)Eloise.

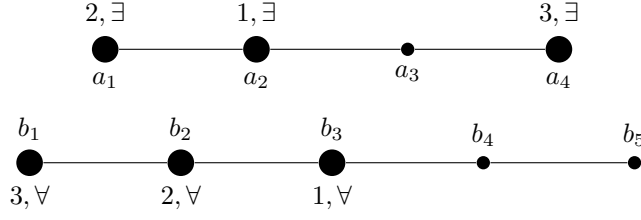


Figure 5.2: A play of game  $G_3(\mathcal{A}, \mathcal{B})$  won by  $\forall$ .  $\mathcal{A}$  is on top, and  $\mathcal{B}$  is below. Position labeled with e.g.  $n, \exists$  has been played by  $\exists$  in round  $n$ .

lations of arity  $\alpha(i)$ . Let  $\bar{a} = (a_1, \dots, a_m) \in A^m$  and  $\bar{b} = (b_1, \dots, b_m) \in B^m$  be sequences of distinguished elements in respectively  $\mathcal{A}$  and  $\mathcal{B}$ . Fix  $k \geq 0$ .

A *play* of the EF game  $G_k((\mathcal{A}, \bar{a}), (\mathcal{B}, \bar{b}))$  is an ordered sequence of  $k$  rounds. In round  $r$ ,  $\forall$  chooses one element, either  $a_{m+r}$  from  $A$  or  $b_{m+r}$  from  $B$ .  $\exists$  responds with one element from the set not picked by  $\forall$ , called respectively  $b_{m+r}$  or  $a_{m+r}$ .

Player  $\exists$  wins a play of the game if she keeps a isomorphism between  $(a_1, \dots, a_{m+k})$  and  $(b_1, \dots, b_{m+k})$ , i.e. if for each  $i$ ,  $1 \leq i \leq p$ , and for each sequence of indices  $j_1, \dots, j_{\alpha(i)}$  between 1 and  $m+k$  we have

$$(a_{j_1}, \dots, a_{j_{\alpha(i)}}) \in R_i \iff (b_{j_1}, \dots, b_{j_{\alpha(i)}}) \in S_i.$$

If this is not the case, the play is won by player  $\forall$ .

Player  $\exists$  has a *winning strategy* in game  $G_k((\mathcal{A}, \bar{a}), (\mathcal{B}, \bar{b}))$  iff there are functions  $f_1, \dots, f_k$  such that for all  $r$ ,  $1 \leq r \leq k$ ,

- $f_r : (A \cup B)^r \rightarrow (A \cup B)$ ;
- if  $c_1, \dots, c_r \in A \cup B$  are the choices of  $\forall$  in the first  $r$  rounds, then
  - $f_r(c_1, \dots, c_r) \in A$  if  $c_r \in B$ , and
  - $f_r(c_1, \dots, c_r) \in B$  if  $c_r \in A$ ;
- if  $c_1, \dots, c_r \in A \cup B$  are the choices of  $\forall$  in the first  $r$  rounds, and

$$a_{m+r} = \begin{cases} c_r & \text{if } c_r \in A \\ f_r(c_1, \dots, c_r) & \text{if } c_r \in B \end{cases} \quad b_{m+r} = \begin{cases} f_r(c_1, \dots, c_r) & \text{if } c_r \in A \\ c_r & \text{if } c_r \in B \end{cases}$$

then for each  $i$ ,  $1 \leq i \leq p$ , and for each sequence of indices  $j_1, \dots, j_{\alpha(i)}$  between 1 and  $m+k$  we have

$$(a_{j_1}, \dots, a_{j_{\alpha(i)}}) \in R_i \iff (b_{j_1}, \dots, b_{j_{\alpha(i)}}) \in S_i.$$

If this is not the case, player  $\forall$  has a winning strategy.

If  $\exists$  has a winning strategy on game  $G_k((\mathcal{A}, \bar{a}), (\mathcal{B}, \bar{b}))$ , we write  $(\mathcal{A}, \bar{a}) \sim_k (\mathcal{B}, \bar{b})$ .

*Example 5.9* (continuing from p. 64). Consider, again, game  $G_3(\mathcal{A}, \mathcal{B})$ . Note that in this game we do not have  $\bar{a}$  and  $\bar{b}$ , which would be pre-defined moves. Does either player have a winning strategy in this game?

The answer is yes, and the player that has a winning strategy is  $\forall$ . In fact,  $\forall$  can choose  $b_3$  in his first move. Then, suppose without loss of generality that  $\exists$  chooses

$a_1$  or  $a_2$  (the strategy if she chooses  $a_2$  or  $a_4$  is symmetric). If  $\exists$  chooses  $a_1$ , in the next move  $\forall$  chooses  $b_1$  or  $b_2$ , and  $\exists$  cannot respond to this move without breaking isomorphism. If  $\exists$  chooses  $a_2$ , in the next move  $\forall$  chooses  $b_2$  and the play goes on as in Figure 5.2, with the inevitable defeat of  $\exists$ .

What about game  $G_2(\mathcal{A}, \mathcal{B})$ ? In this case, the player having a winning strategy is  $\exists$ . Whatever element  $\forall$  picks in the first round,  $\exists$  just needs to answer with an element in the other word that is not an endpoint (i.e., not  $a_1, a_4, b_1$  or  $b_5$ ). Then, whatever element  $\forall$  chooses in the second and last move,  $\exists$  can answer with one in the same order.

Thus,  $\mathcal{A} \sim_2 \mathcal{B}$ , but  $\mathcal{A} \not\sim_3 \mathcal{B}$ .

EF games are linked to FOL by the following theorem:

**Theorem 5.11** (Ehrenfeucht-Fraïssé [71, 82]). *Let  $\mathcal{A}$  and  $\mathcal{B}$  be two structures on the same signature, with domains  $A$  and  $B$  respectively. Given  $k \geq 0$  and  $m \geq 0$ , for any  $\bar{a} \in A^m$  and  $\bar{b} \in B^m$ , we have*

$$(\mathcal{A}, \bar{a}) \equiv_k (\mathcal{B}, \bar{b}) \iff (\mathcal{A}, \bar{a}) \sim_k (\mathcal{B}, \bar{b}).$$

This gives us a practical way of proving that two structures are indistinguishable by FO formulas of quantifier rank at most  $k$ .

*Example 5.9* (continuing from p. 65). According to our analysis of games  $G_2(\mathcal{A}, \mathcal{B})$  and  $G_3(\mathcal{A}, \mathcal{B})$ , we can state that  $\mathcal{A} \equiv_2 \mathcal{B}$ , but  $\mathcal{A} \not\equiv_3 \mathcal{B}$ . Hence, there exists a FO formula of quantifier rank 3 that is true in  $\mathcal{A}$  but not in  $\mathcal{B}$  (or vice-versa), but no formula of quantifier rank 2 has the same property.

In general,

**Corollary 5.12.** *We have  $(\mathcal{A}, \bar{a}) \equiv (\mathcal{B}, \bar{b})$  iff  $(\mathcal{A}, \bar{a}) \sim_k (\mathcal{B}, \bar{b})$  for all  $k \geq 0$ .*

### 5.3 Composition Arguments

One of the most successful uses of EF games is to show that the first-order theory on a class of structures can be determined (and decided) from the theories of parts in which we divide such structures. In particular, S. Shelah used this kind of arguments in [149] to show that the rank- $k$  type of the concatenation (or *sum*) of labeled orderings is determined by the rank- $k$  types of such orderings, and can be computed effectively.

More formally, given structures

$$\mathcal{A}_1 = (A_1, R_{1,1}, \dots, R_{1,p}), \dots, \mathcal{A}_n = (A_n, R_{n,1}, \dots, R_{n,p})$$

on the same signature, with  $A_1, \dots, A_n$  disjoint, these can be composed to create a structure  $\mathcal{A} = (A_1 \cup \dots \cup A_n, S_1, \dots, S_p)$  such that  $R_{i,j} \subseteq S_j$  for all  $1 \leq i \leq n$  and  $1 \leq j \leq p$ .

Then, a *composition argument* (or theorem) on such structures is a proof that, given two structures  $\mathcal{A}$  and  $\mathcal{B}$  that can be divided as above, if for some  $k \geq 0$  we have  $\mathcal{A}_i \equiv_k \mathcal{B}_i$  for all  $1 \leq i \leq n$ , then  $\mathcal{A} \equiv_k \mathcal{B}$ . Such proofs are usually carried out by noting that, thanks to Theorem 5.11, if  $\mathcal{A}_i \equiv_k \mathcal{B}_i$  then  $\mathcal{A}_i \sim_k \mathcal{B}_i$ , and  $\exists$  has a winning strategy on games  $G_k(\mathcal{A}_i, \mathcal{B}_i)$ , for all  $i$ . Thus, if one shows that such strategies can be combined to obtain a winning strategy for game  $G_k(\mathcal{A}, \mathcal{B})$ , another application of Theorem 5.11 leads to  $\mathcal{A} \equiv_k \mathcal{B}$ .

If the final purpose of the composition argument is to prove decidability of a theory as in [149], then one also has to show that the rank- $k$  type of  $\mathcal{A}$  can be effectively computed from those of all  $\mathcal{A}_i$ 's that compose  $\mathcal{A}$ . However, the purpose of our proofs will be to show that a FO formula on  $\mathcal{A}$  can be expressed by combining formulas of the same quantifier rank on  $\mathcal{A}_i$ 's. For this, we will just need the fact that  $\mathcal{A}_i \equiv_k \mathcal{B}_i$  for all  $i$  implies  $\mathcal{A} \equiv_k \mathcal{B}$ . Thus, formula  $\varphi_i$  with  $\text{qr}(\varphi_i)$  holds in both  $\mathcal{A}_i$  and  $\mathcal{B}_i$  for some  $i$  iff a formula  $\varphi$  that contains  $\varphi_i$  but restricts it to quantify on  $\mathcal{A}_i$  only holds on  $\mathcal{A}$ . This is particularly interesting if  $\varphi_i$  is a Hintikka formula: if we gather Hintikka formulas  $H_{\mathcal{A}_i}^k$  for all  $1 \leq i \leq n$  and we combine them appropriately, we obtain a formula that characterizes the rank- $k$  type of  $\mathcal{A}$ , and is equivalent to  $H_{\mathcal{A}}^k$ .

*Example 5.13.* As an example, we give a composition argument for the concatenation of two labeled words.

Let  $AP$  be a set of atomic propositions of size  $p$ , and  $\mathcal{A} = (A = \{a_1, \dots, a_n\}, \leq_{\mathcal{A}}, P_1^{\mathcal{A}}, \dots, P_p^{\mathcal{A}})$  and  $\mathcal{B} = (B = \{b_1, \dots, b_n\}, \leq_{\mathcal{B}}, P_1^{\mathcal{B}}, \dots, P_p^{\mathcal{B}})$  be two words, where  $P_i^{\mathcal{A}}, 1 \leq i \leq p$ , is the monadic relation (i.e., set) containing positions of  $A$  where the  $i$ -th label in  $AP$  holds (and the same for  $P_i^{\mathcal{B}}$ ).

The concatenation of  $\mathcal{A}$  and  $\mathcal{B}$ , denoted  $\mathcal{A} + \mathcal{B}$ , is a word  $\mathcal{C} = (C = A \cup B, \leq_{\mathcal{C}}, P_1^{\mathcal{C}}, \dots, P_p^{\mathcal{C}})$  where, for all  $1 \leq i \leq p$ ,  $P_i^{\mathcal{C}} = P_i^{\mathcal{A}} \cup P_i^{\mathcal{B}}$ , and  $\leq_{\mathcal{C}}$  is such that, for all  $c_1, c_2 \in C$ , we have  $c_1 \leq_{\mathcal{C}} c_2$  iff

- $c_1, c_2 \in A$  and  $c_1 \leq_{\mathcal{A}} c_2$ ; or
- $c_1, c_2 \in B$  and  $c_1 \leq_{\mathcal{B}} c_2$ ; or
- $c_1 \in A$  and  $c_2 \in B$ .

Now, we can prove the following:

*Theorem 5.14.* Let  $\mathcal{A}_1, \mathcal{B}_1, \mathcal{A}_2, \mathcal{B}_2$  be labeled words on the same set of atomic propositions  $AP$ , and let  $k \geq 0$ .

If  $\mathcal{A}_1 \equiv_k \mathcal{A}_2$  and  $\mathcal{B}_1 \equiv_k \mathcal{B}_2$ , then  $(\mathcal{A}_1 + \mathcal{B}_1) \equiv_k (\mathcal{A}_2 + \mathcal{B}_2)$ .

*Proof.* In the following, we call  $\mathcal{C} = \mathcal{A}_1 + \mathcal{B}_1$ , and  $\mathcal{D} = \mathcal{A}_2 + \mathcal{B}_2$ , and we write e.g.  $c \in \mathcal{A}_1$  meaning that  $c$  is in  $\mathcal{A}_1$ 's domain.

We show that if  $\mathcal{A}_1 \equiv_k \mathcal{A}_2$  and  $\mathcal{B}_1 \equiv_k \mathcal{B}_2$ ,  $\exists$  has a winning strategy in  $G_k(\mathcal{C}, \mathcal{D})$ .

The strategy consists in  $\exists$  "simulating" games  $G_k(\mathcal{A}_1, \mathcal{A}_2)$  and  $G_k(\mathcal{B}_1, \mathcal{B}_2)$  with the choices of made by  $\forall$  in the main game. Thus, in each round, if  $\forall$  picks an element from  $\mathcal{A}_i + \mathcal{B}_i$  coming from  $\mathcal{A}_i$ , for  $i \in \{1, 2\}$ ,  $\exists$  simulates the same move in  $G_k(\mathcal{A}_1, \mathcal{A}_2)$ , and she uses her winning strategy on that game to pick an element form  $\mathcal{A}_{(3-i)}$  in response. If, instead,  $\forall$  picks an element from  $\mathcal{A}_i + \mathcal{B}_i$  coming from  $\mathcal{B}_i$ ,  $\exists$  uses her winning strategy on  $G_k(\mathcal{B}_1, \mathcal{B}_2)$  to answer with an element from  $\mathcal{B}_{(3-i)}$ .

We prove by induction on the round number  $r$  that this is a winning strategy for  $\exists$ . We call  $c_t$  the element of  $\mathcal{C}$  picked at round  $t$ , and  $d_t$  the one picked from  $\mathcal{D}$ .

In the base case,  $r = 0$ , no elements have been picked, so partial isomorphism is trivially satisfied.

For  $r > 0$ , suppose that at round  $r - 1$  partial isomorphism is kept, that is:

- there is a permutation of indices  $j_1, \dots, j_{r-1}$  ranging from 1 to  $r - 1$  such that  $c_{j_1} \leq_{\mathcal{C}} \dots \leq_{\mathcal{C}} c_{j_{r-1}}$  and  $d_{j_1} \leq_{\mathcal{D}} \dots \leq_{\mathcal{D}} d_{j_{r-1}}$ , and
- for all  $1 \leq t \leq r - 1$  and  $1 \leq p \leq |AP|$ , we have  $c_t \in P_p^{\mathcal{C}}$  iff  $d_t \in P_p^{\mathcal{D}}$ .

We now assume w.l.o.g. that at round  $r$   $\forall$  picks an element  $c_r$  from  $\mathcal{A}_1$ , as the proof in the other cases is symmetric. Let  $I = \{t \leq r-1 \mid c_t \in \mathcal{A}_1\}$ . Then,  $\exists$  picks an element  $d_r \in \mathcal{A}_2$  from  $\mathcal{D}$  according to her winning strategy in  $G_k(\mathcal{A}_1, \mathcal{A}_2)$ , simulating a play of the latter game in which in the first  $|I|$  moves, for each  $i \in I$ ,  $c_i$  has been picked from  $\mathcal{A}_1$ , and  $d_i$  from  $\mathcal{A}_2$ .

Clearly, we have  $c_r \leq_C c_i$  for each  $c_i \in \mathcal{B}_1$ , and  $d_r \leq_{\mathcal{D}} d_i$  for each  $d_i \in \mathcal{B}_2$ . Moreover, since  $d_r$  was picked according to  $\exists$ 's winning strategy on  $G_k(\mathcal{A}_1, \mathcal{A}_2)$ , for all  $i \in I$  we have  $c_r \leq_{\mathcal{A}_1} c_i$  iff  $d_r \leq_{\mathcal{A}_2} d_i$  and hence  $c_r \leq_C c_i$  iff  $d_r \leq_{\mathcal{D}} d_i$ . For the same reason, we have  $c_r \in P_p^{\mathcal{A}_1}$  iff  $d_r \in P_p^{\mathcal{A}_2}$  and hence  $c_r \in P_p^{\mathcal{C}}$  iff  $d_r \in P_p^{\mathcal{D}}$  for all  $1 \leq p \leq |AP|$ .  $\square$

**Part II**

**Temporal Logic**



In this part of the thesis, we present two temporal logics on OPLs.

OPTL is our first attempt at devising a temporal logic capable of expressing OPL requirements. We present its syntax and semantics in Chapter 6, and study its expressiveness with respect to state-of-the-art logics on nested words and FOL in Chapter 7.

OPTL and its model checking were initially presented in [53] and [52], and later revised in [54]. The contents of Section 7.2 have been introduced in [58].

Then, we present POTL, a temporal logic that captures the FO-definable fragment of OPLs. We introduce its syntax and semantics in Chapter 8, and study its expressiveness in Chapter 9. The questions of model checking and satisfiability are settled in Part III.

POTL and its model checking have been presented in [56], upon which Chapter 8 is partially based; the expressive completeness proof of Chapter 9 is presented in [58].



## Chapter 6

# OPTL Syntax and Semantics

As we saw in Chapter 4, languages recognized by different OPAs on a given OP alphabet form a Boolean algebra. These properties allow us to define OPTL as a sound propositional temporal logic, with logical disjunction, conjunction and negation. Given an OP alphabet, each well-formed OPTL formula characterizes a subset of the universal language based on that alphabet. Due to the closure properties above, for each OPTL formula it is possible to identify an OPA that recognizes the same language denoted by it, paving the way for model checking of OPTL.

Although in Chapter 7 we prove that OPTL is not as expressive as FOL on OP words, we include it in this thesis as an important step in the process of defining temporal logics and model checking for OPLs, as well as for the importance of the negative result on expressiveness itself, because of the original technique used.

Next, in Section 6.1 we present OP words, the algebraic structure upon which OPTL—and also POTL, which we present in Chapter 8—is based. Then, we introduce the syntax of OPTL explaining its meaning by means of simple examples in Section 6.2; we formally define its semantics in Section 6.3, and provide more complex, real-world examples in Section 6.4.

### 6.1 Operator Precedence Words

Operator Precedence Temporal Logic (OPTL) is a linear-time temporal logic, which extends the classical LTL. We recall that the semantics of LTL [138] is defined on a Dedekind-complete set of word positions  $U$  equipped with a total ordering  $<$  and monadic relations, called *atomic propositions* (cf. Section 2.2). In this work, we consider a discrete timeline, hence  $U = \{0, 1, \dots, n\}$ , with  $n \in \mathbb{N}$ , or  $U = \mathbb{N}$ .

The total order, however, is not sufficient to express properties of more complex structures than the linear ones, such as tree-shaped ones, which are the natural domain of CFLs. Other logics on CFLs, such as the one from [115] and those on nested words, use a matching relation between terminal characters that act as parentheses. OPLs are structured but not “visibly structured” as they lack explicit parentheses (cf. Chapter 4). Nevertheless, a more sophisticated notion of matching relation has been introduced in [119] for OPLs by exploiting the fact that OPLs remain input-driven thanks to the OPM. We name the new matching condition *chain relation* and define it below. We fix a finite set of atomic propositions  $AP$ , and an OPM  $M_{AP}$  on  $\mathcal{P}(AP)$ .

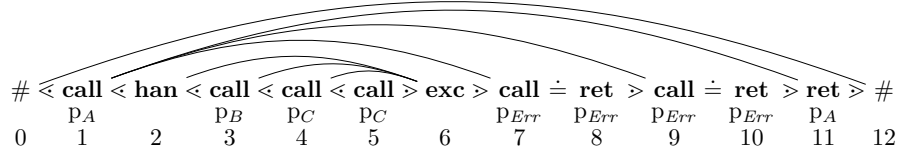


Figure 6.1: The example word  $w_{ex}$  from Chapter 4 as an OP word. Chains are highlighted by arrows joining their contexts; structural labels are in bold, and other atomic propositions are shown below them.  $p_l$  means a **call** or a **ret** is related to procedure  $p_l$ . First, procedure  $p_A$  is called (pos. 1), and it installs an exception handler in pos. 2. Then, three nested procedures are called, and the innermost one ( $p_C$ ) throws an exception, which is caught by the handler. Two more functions are called and, finally,  $p_A$  returns.

**Definition 6.1** (OP word). A *word structure*—also called *OP word* for short—on  $AP$  is the tuple  $\langle U, <, M_{AP}, P \rangle$ , where  $U, <$ , and  $M_{AP}$  are as above, and  $P: AP \rightarrow \mathcal{P}(U)$  is a function associating each atomic proposition with the set of positions where it holds, with  $0, (n+1) \in P(\#)$ .

For the time being, we consider just finite string languages; the necessary extensions needed to deal with  $\omega$ -languages will be introduced in Section 6.3.1.

**Definition 6.2** (Chain relation). The *chain relation*  $\chi(i, j)$  holds between two positions  $i, j \in U$  iff  $i < j - 1$ , and  $i$  and  $j$  are respectively the left and right contexts of the same chain (cf. Definition 4.9), according to  $M_{AP}$  and the labeling induced by  $P$ .

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

The chain relation augments the linear structure of a word with the tree-like structure of OPLs. Figure 6.1 shows word  $w_{ex}$  from Chapter 4 as an OP word and emphasizes the distinguishing feature of the relation, i.e. that, for composed chains, it may not be one-to-one, but also one-to-many or many-to-one. Notice the correspondence between internal nodes in the ST, which we show again in Figure 6.2, and pairs of positions in the  $\chi$  relation. To exemplify the partition into structural and normal labels, in the word of Figure 6.1 position 1 is labeled with the set  $\{\mathbf{call}, p_A\}$ , and position 3 with  $\{\mathbf{call}, p_B\}$ : they both contain **call**, so they are in the  $<$  relation according to matrix  $M_{\mathbf{call}}$  of Figure 4.2.

In the ST, we say that the right context  $j$  of a chain is at the *same level* as the left one  $i$  when  $i \doteq j$  (e.g., in Figure 6.2, pos. 1 and 11), at a *lower level* when  $i < j$  (e.g., pos. 1 with 7, and 9), at a *higher level* if  $i \triangleright j$  (e.g., pos. 3 and 4 with 6).

Furthermore, given  $i, j \in U$ , relation  $\chi$  has the following properties:

1. It never crosses itself: if  $\chi(i, j)$  and  $\chi(h, k)$ , for any  $h, k \in U$ , then we have  $i < h < j \implies k \leq j$  and  $i < k < j \implies i \leq h$ .
2. If  $\chi(i, j)$ , then  $i < i + 1$  and  $j - 1 \triangleright j$ .

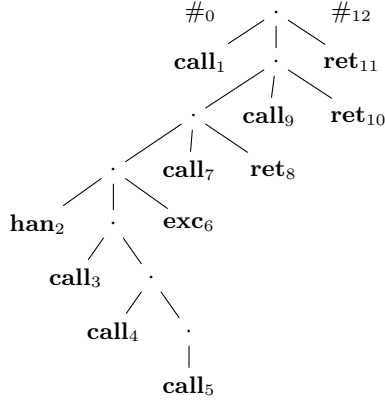


Figure 6.2: The ST of word  $w_{ex}$  (the same as Figure 4.4, but with position numbers). Dots represent non-terminals.

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

Property 4 says that when the chain relation is one-to-many, the contexts of the outermost chain are in the  $\doteq$  or  $>$  relation, while the inner ones are in the  $<$  relation. Property 3 says that contexts of outermost many-to-one chains are in the  $\doteq$  or  $<$  relation, and the inner ones are in the  $>$  relation. We prove such properties below.

**Lemma 6.3.** *Given an OP word  $w$  and positions  $i, j$  in it, properties 1, 2, 3, and 4 hold.*

*Proof.* In the following, we denote by  $c_p$  the character labeling word position  $p$ , and by writing  $c^{-1}[x_0 c_0 x_1 \dots x_n c_n x_{n+1}]^{c^{n+1}}$  we imply  $c_{-1}$  and  $c_{n+1}$  are the contexts of a simple or composed chain, where either  $x_p = \varepsilon$ , or  $c^{p-1}[x_p]^{c^p}$  is a chain, for each  $p$ .

1. Suppose, by contradiction, that  $\chi(i, j)$ ,  $\chi(h, k)$ , and  $i < h < j$ , but  $k > j$ . Consider the case in which  $\chi(i, j)$  is the innermost chain whose body contains  $h$ , so it is of the form  $c^i[x_0 c_0 \dots c_h x_p c_p \dots c_n x_{n+1}]^{c^j}$  or  $c^i[x_0 c_0 \dots c_h x_{n+1}]^{c^j}$ . By the definition of chain, we have either  $c_h \doteq c_p$  or  $c_h > c_j$ , respectively.

Since  $\chi(h, k)$ , this chain must be of the form  $c^h[x_p c_p \dots]^{c^k}$  or  $c^h[x_{n+1} c_j \dots]^{c^k}$ , implying  $c_h < c_p$  or  $c_h < c_j$ , respectively. This means there is a conflict in the OPM, contradicting the hypothesis that  $w$  is an OP word.

In case  $\chi(i, j)$  is not the innermost chain whose body contains  $h$ , we can reach the same contradiction by inductively considering the chain between  $i$  and  $j$  containing  $h$  in its body. Moreover, it is possible to reach a symmetric contradiction with the hypothesis  $\chi(i, j)$ ,  $\chi(h, k)$ , and  $i < k < j$ , but  $i > h$ .

2. Trivially follows from the definition of chain.
3. We prove that only  $i_1$  can be s.t.  $i_1 < j$  or  $i_1 \doteq j$ . Suppose, by contradiction, that for some  $r > 1$  we have  $i_r < j$  or  $i_r \doteq j$ .

If  $i_r < j$ , by the definition of chain,  $j$  must be part of the body of another composed chain whose left context is  $i_r$ . So,  $w$  contains a structure of the form

$c_{i_r}[x_0 c_j \dots]^{c_k}$  where  $|x_0| \geq 1$ ,  $c_{i_r}[x_0]^{c_j}$ , and  $k > j$  is s.t.  $\chi(i_r, k)$ . This contradicts the hypothesis that  $\chi(i_1, j)$ , because such a chain would cross  $\chi(i_r, k)$ , contradicting property (1).

If  $i_r \doteq j$ , then  $w$  contains a structure  $c_{i_r-1}[\dots c_{i_r} x_{i_r}]^{c_j}$ , with  $|x_{i_r}| \geq 1$  and  $c_{i_r}[x_{i_r}]^{c_j}$ . By the definition of chain, we have  $i_r \succ j$ , which contradicts the hypothesis.

Thus, the only remaining alternative for  $r > 1$  is  $i_r \succ j$ .

Similarly, if we had  $i_1 \succ j$ , the definition of chain would lead to the existence of a position  $h < i_1$  s.t.  $\chi(h, j)$ , which contradicts the hypothesis that  $i_1$  is the leftmost of such positions.  $i_1 < j$  and  $i_1 \doteq j$  do not lead to such contradictions.

4. The proof is symmetric to the previous one.  $\square$

While LTL's linear paths only follow the ordering relation  $<$ , paths in OPTL may follow the  $\chi$  relation too. As a result, an OPTL path through a string can simulate paths through the corresponding ST. In the next section, we introduce such kinds of paths informally.

## 6.2 Syntax and Informal Semantics

The syntax of OPTL is given by the following grammar, where  $a$  denotes any symbol in  $AP$ :

$$\begin{aligned} \varphi ::= & a \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid \bigcirc\varphi \mid \bigcirc_\chi\varphi \mid \ominus\varphi \mid \ominus_\chi\varphi \\ & \mid \varphi \mathcal{U}\varphi \mid \varphi \mathcal{U}^\Pi\varphi \mid \varphi \mathcal{S}\varphi \mid \varphi \mathcal{S}^\Pi\varphi \mid \varphi \mathcal{U}^\ominus\varphi \mid \varphi \mathcal{S}^\ominus\varphi \end{aligned}$$

We informally show the meaning of OPTL operators by referring to the word of Figure 6.1, with respect to the OPM of Figure 4.2. The  $\bigcirc$  and  $\ominus$  symbols denote the next and back operators from LTL, while the undecorated  $\mathcal{U}$  and  $\mathcal{S}$  operators are the LTL until and since. These operators have the same semantics as in LTL, and thus they may only express regular properties, although they do so on OP words, whose structure is tree-like, and not only regular. In order to fully exploit the expressive power of OPLs, operators that interact with the peculiarities of their structure are needed.

The  $\bigcirc_\chi$  and  $\ominus_\chi$  operators, which we call *matching next* and *matching back*, express properties on string positions in the maximal chain relation (which will be formally defined later on) with the current one. In Figure 6.1, the chain relation between two positions is shown by an arc joining them. A chain is maximal if it is the outermost one starting or ending in a position: for example, the chain between positions 1 and 11 is maximal, while the ones between 1 and 7, and 1 and 9 are not, because they are contained in the body of the former. For example, formula  $\bigcirc_\chi \text{exc}$ , when evaluated in positions containing a **call**, is true if the corresponding procedure is terminated by an exception thrown by an inner procedure, such as 3 and 4 of Figure 6.1, because position 3 forms a maximal chain with 6, in which **exc** holds, and so on. Formula  $\ominus_\chi \text{han}$ , if evaluated in **exc** positions, is true if the corresponding exception is caught by a **han** statement: it holds in 6, because position 2 forms a chain with it, and **han** holds in 2.

The  $\mathcal{U}^\Pi$  and  $\mathcal{S}^\Pi$  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  $\Pi$  is a placeholder for one or more precedence relations allowed in the path (e.g.  $\mathcal{U}^{\leftarrow \dot{=}}$  or  $\mathcal{U}^{\triangleright \leftarrow}$  and so on).

The presence or absence of these relations influences the ability of the *summary* paths associated with these operators to “cross” chains, entering them from the outside or exiting them from the inside. The  $\triangleright$  relation allows paths starting inside the body of a chain to exit it, and proceed with its right context. Note that the position before the right context of a chain is always in the  $\triangleright$  relation with it. For example, formula  $(\text{call} \vee \text{exc}) \mathcal{U}^{\triangleright} \text{call}$  is true in position 5 because there is a path that links positions 5, 6 and 7, which are in the  $\triangleright$  relation: positions 5 and 6 satisfy  $\text{call} \vee \text{exc}$ , while position 7 satisfies  $\text{call}$ . Since  $\text{call}$  positions yield precedence to other  $\text{call}$  positions, if this operator is evaluated in position 3 or 4, its paths cannot enter the body of an inner procedure call, but they can only jump to the  $\text{exc}$  statement.

The  $\leftarrow$  relation allows paths to enter call bodies: formula  $\top \mathcal{U}^{\leftarrow} p_C$  is true in position 2 because of paths 2-3-4 and 2-3-4-5, which enter all chains they encounter. Such paths could, however, skip the maximal chain starting in 2 and continue with 6. This can be prevented by adding  $(\text{han} \vee \text{call})$ , reducing the scope of the formula to a single stack frame:  $\text{han} \wedge ((\text{han} \vee \text{call}) \mathcal{U}^{\leftarrow} p_C)$  holds in  $\text{han}$  positions that are present in the stack when procedure  $p_C$  is called.

The  $\dot{=}$  relation allows paths to connect consecutive positions that are part of the same right-hand side. This is useful especially in the presence of OPMs with  $\dot{=}$ -circularity, such as the one we will use in Section 7.1. The precedence relations can be combined together to sum these three different behaviors for the summary until and since operators, posing or lifting restrictions on the way these operators navigate the tree-like structure of OP words. In a sense,  $\mathcal{U}^{\leftarrow \dot{=}}$  resembles the summary-down until of NWTL, while  $\mathcal{U}^{\dot{=} \triangleright}$  behaves similarly to the summary-up, with the difference that OPTL operators can interact with multiple chains ending in the same position. Similar considerations can be made for the since versions of these operators.

$\mathcal{U}^{\oplus}$  and  $\mathcal{S}^{\oplus}$ , where  $\oplus$  is a placeholder for  $\uparrow$  or  $\downarrow$ , are called *hierarchical until and since*, and 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,  $\text{call} \mathcal{U}^{\uparrow} p_{Err}$  and  $\text{call} \mathcal{S}^{\downarrow} p_{Err}$  hold in position 2, because there is path 7-9 made of ending positions of chains starting in 2, such that  $\text{call}$  holds until  $p_{Err}$  holds (or  $\text{call}$  has held since  $p_{Err}$  held). Formulas  $\text{call} \mathcal{U}^{\downarrow} p_C$  and  $\text{call} \mathcal{S}^{\uparrow} p_B$  hold in position 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.

### 6.3 Formal Semantics

The semantics of OPTL is based on the OP word structure presented in Definition 6.1, and it deeply relies on the *chain* relation, from Definition 6.2. We additionally define two one-to-one relations, helpful in identifying the largest chain starting or ending in a word position.

**Definition 6.4** (Maximal chains). The *maximal forward* chain relation is defined so that, for any  $i, j \in U$ ,

$$\vec{\chi}(i, j) \iff \chi(i, j) \wedge (i \dot{=} j \vee i \triangleright j);$$

the *maximal backward* chain relation is defined as

$$\overleftarrow{\chi}(i, j) \iff \chi(i, j) \wedge (i \leftarrow j \vee i \dot{=} j).$$

In finite OP words, this implies  $\vec{\chi}(i, j)$  iff  $j = \max\{k \in U \mid \chi(i, k)\}$  and  $\overleftarrow{\chi}(i, j)$  iff  $i = \min\{k \in U \mid \chi(k, j)\}$ . In Section 6.3.1, we shall see that this does not always hold in OP  $\omega$ -words. The maximal forward (resp. backward) chain relation is undefined for a pair of positions either if they are the context of no chain, or if they are the context of a chain which is not forward- (resp. backward-) maximal.

Let  $w$  be an OP word, and  $a \in AP$ . Then, for any position  $i \in U$  of  $w$ , we have  $(w, i) \models a$  if  $a \in P(i)$ . Operators such as  $\wedge$  and  $\neg$  have the usual semantics from propositional logic, while  $\circ$  and  $\ominus$  have the same semantics as in LTL (i.e.  $(w, i) \models \circ \varphi$  iff  $(w, i+1) \models \varphi$ , and similarly for  $\ominus$ ).

The  $\circ_\chi$  and  $\ominus_\chi$  operators express properties regarding the right (resp. left) context of a maximal chain that starts (resp. ends) in the current position:

- $(w, i) \models \circ_\chi \varphi$  iff there exists  $j \in U$  such that  $\vec{\chi}(i, j)$  and  $(w, j) \models \varphi$ ;
- $(w, i) \models \ominus_\chi \varphi$  iff there exists  $j \in U$  such that  $\overleftarrow{\chi}(j, i)$  and  $(w, j) \models \varphi$ .

In Figure 6.1,  $(w, 3) \models \circ_\chi \mathbf{exc}$  because  $\vec{\chi}(3, 6)$  and  $(w, 6) \models \mathbf{exc}$ ;  $(w, 6) \models \ominus_\chi \mathbf{han}$  holds because  $\overleftarrow{\chi}(2, 6)$  and  $(w, 2) \models \mathbf{han}$ , but  $(w, 6) \not\models \ominus_\chi p_B$  because chain  $\chi(3, 6)$  is not backward-maximal (although it is forward-maximal).

We define until and since operators based on the class of paths they consider.

**Definition 6.5** (Paths, until and since). A *path* of length  $n \in \mathbb{N}$  between  $i, j \in U$  is a sequence of positions  $i_1 < i_2 < \dots < i_n$ , with  $i \leq i_1$  and  $i_n \leq j$ .

The *until* operator on a set of paths  $\Gamma$  is defined as follows: for any word  $w$  and position  $i \in U$ , and for any two OPTL formulas  $\varphi$  and  $\psi$ ,  $(w, i) \models \varphi \mathcal{U}(\Gamma) \psi$  iff there exist a position  $j \in U$ ,  $j \geq i$ , and a path  $i_1 < i_2 < \dots < i_n$  between  $i$  and  $j$  in  $\Gamma$  such that  $(w, i_k) \models \varphi$  for any  $1 \leq k < n$ , and  $(w, i_n) \models \psi$ .

The *since* operator is defined symmetrically.

Note that a path from  $i$  to  $j$  does not necessarily start in  $i$  and end in  $j$ , but it may do in positions between them. However, this will only happen with hierarchical paths. We define the different kinds of until/since operators by associating them with suitable sets of paths.

The *linear until* ( $\varphi \mathcal{U} \psi$ ) and *linear since* ( $\varphi \mathcal{S} \psi$ ) operators, based on linear paths, have the same semantics as in LTL. A *linear path* starting in position  $i \in U$  is such that  $i_1 = i$  and, for any  $1 \leq k < n$ , we have  $i_{k+1} = i_k + 1$ .

The *OP-summary until* operator exploits the  $\vec{\chi}$  relation to express properties on paths that skip chain bodies, also keeping precedence relations between consecutive word positions into account.

**Definition 6.6.** Given a set  $\Pi \subseteq \{\ll, \dot{=}, \gg\}$ , the  $\mathcal{U}^\Pi$  operator is based on the class of *forward OP-summary* paths. A path of this class between  $i$  and  $j \in U$  is a sequence of positions  $i = i_1 < i_2 < \dots < i_n = j$  such that, for any  $1 \leq k < n$ ,

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

There exists at most one forward OP-summary path between any two positions. For example, in Figure 6.1, if we take  $\Pi = \{\gg\}$  as in  $(\mathbf{call} \vee \mathbf{exc}) \mathcal{U}^\Pi \mathbf{call}$ , the path between 3 and 9 is made of positions 3-6-7, because  $\vec{\chi}(3, 6)$  and the body of this chain is skipped, and  $6 \gg 7$ . If we add  $\dot{=}$ , and use  $(\mathbf{call} \vee \mathbf{exc}) \mathcal{U}^\Pi \mathbf{ret}$ , the path starting from position 3 can extend to 11, going through 3-6-7-8-9-10-11. If we took

e.g.  $\Pi = \{=, <\}$ , there would be no such path, because consecutive positions in the  $>$  relation are not considered. With  $\Pi = \{>, <\}$ , the path between 1 and 7 does not skip the body of chain  $\chi(2, 6)$ , because it is not forward-maximal: it is the linear path 1-2-3-4-5-6-7.

The *OP-summary since* operator is based on *backward OP-summary* paths, which are symmetric to their until counterparts, relying on the  $\overleftarrow{\chi}$  relation instead of  $\overrightarrow{\chi}$ .

**Definition 6.7.** A *backward OP-summary* path is a sequence of positions  $i = i_1 < i_2 < \dots < i_n = j$  such that, for any  $1 < k \leq n$ ,

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

For example,  $(\text{exc} \vee \text{han}) \mathcal{S}^{<}$  **call** holds in position 6 because of path 1-2-6, that skips the body of chain  $\overleftarrow{\chi}(2, 6)$  and satisfies **exc**  $\vee$  **han** in 2 and 6, and **call** in 1. Again, bodies of chains that are not backward-maximal cannot be skipped.

While summary operators are only aware of the maximal chain relations, hierarchical operators can express properties discriminating between all other chains. The *hierarchical yield-precedence* until and since operators, denoted as  $\mathcal{U}^\uparrow$  and  $\mathcal{S}^\downarrow$  respectively, are based on paths made of the ending positions of non-maximal chains starting in the current position  $i \in U$ .

**Definition 6.8** (Hierarchical yield-precedence path). A *hierarchical yield-precedence path* is a sequence of word positions  $i_1 < i_2 < \dots < i_n$ , with  $i < i_1$ , such that for any  $1 \leq k \leq n$  we have  $i < i_k$  and  $\chi(i, i_k)$ , and, additionally, there is no  $i'_k$  that satisfies these two properties and  $i_{k-1} < i'_k < i_k$ .

Moreover, for the until operator  $i_1$  must be the leftmost position enjoying the above properties (i.e., there is no  $i'_1$  such that  $i < i'_1 < i_1$  enjoying them), and for the since operator  $i_n$  must be the rightmost one.

$\mathcal{S}^\downarrow$  is called a since operator despite being a future modality. We chose this naming because in formulas such as  $\varphi \mathcal{S}^\downarrow \psi$ , the argument  $\psi$  must hold at the beginning of the path, while  $\varphi$  must hold in subsequent positions, which is the typical behavior of since operators. Note that these paths only contain forward non-maximal chain ends, which are in the  $<$  relation with  $i$ . For example, in position 1 formula **call**  $\mathcal{U}^\uparrow$   $\text{p}_{Err}$  holds because of path 7-9, since **call** holds in 7 and  $\text{p}_{Err}$  in 9; the path made only of position 7 satisfies it too. Similarly, **call**  $\mathcal{S}^\downarrow$   $\text{p}_{Err}$  is satisfied by path 7-9, and by the one made of only position 9. Position 11 is not included in these paths, because it does not satisfy the condition  $2 < 11$  (indeed,  $2 > 11$ ).

Conversely, *hierarchical take-precedence* until and since operators ( $\mathcal{U}^\downarrow$  and  $\mathcal{S}^\uparrow$ ) consider non-maximal chains ending in the current position  $j \in U$ .

**Definition 6.9** (Hierarchical take-precedence path). A *hierarchical take-precedence path* is a sequence of word positions  $i_1 < i_2 < \dots < i_n$ , with  $i_n < j$ , such that for any  $1 \leq k \leq n$  we have  $i_k > j$  and  $\chi(i_k, j)$ , and, additionally, there exists no position  $i'_k$  that satisfies these two properties and  $i_k < i'_k < i_{k+1}$ .

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  $i'_n$ ,  $i_n < i'_n < j$ , that satisfies them).

Note that  $\mathcal{U}^\downarrow$  is an until operator despite being a past modality: again, the reason is that  $\varphi \mathcal{U}^\downarrow \psi$  enforces  $\psi$  at the end of the path, and  $\varphi$  in previous positions, making

it more similar to an until operator. In position 6,  $\text{call } \mathcal{U}^\downarrow p_C$  is satisfied by path 3-4 and not by the one made only of position 3, because  $p_C$  does not hold in 3, but it does in 4, and  $\text{call}$  holds in 3. Formula  $\text{call } \mathcal{S}^\uparrow p_B$  is satisfied by path 3-4, and not by the one made only of 4, because  $p_B$  holds in 3 and  $\text{call}$  in 4.

### 6.3.1 OPTL on $\omega$ -Words

In order to extend the semantics of OPTL to the infinite case, it suffices to consider an infinite set of positions  $U = \mathbb{N}$ . The formal definitions of all OPTL operators remain the same we defined above.

Concerning the intuitive meaning, the only change concerns *forward-maximal chains* ( $\vec{\chi}$ ). We formally defined  $\vec{\chi}(i, j)$  to hold if  $\chi(i, j)$  and  $i \doteq j$  or  $i \succ j$ . In finite OP words, one of such chains is also the outermost one starting in a position  $i$ . In  $\omega$ -words, the outermost chain may be *open* (cf. Section 4.2). In this case, according to Definition 6.4,  $\vec{\chi}(i, j)$  does not hold for any  $j$  if  $i$  is the left context of an open chain.

Consequently, property 4 of the  $\chi$  relation does not hold if a position  $i$  is the left context of an open chain. In this case, there may be positions  $j_1 < j_2 < \dots < j_n$  such that  $\chi(i, j_p)$  and  $i < j_p$  for all  $1 \leq p \leq n$ , but no position  $k$  such that  $\chi(i, k)$  and  $i \succ k$  or  $i \doteq k$ .

## 6.4 Examples

Many relevant properties can be expressed in OPTL. We use the standard shortcuts of LTL, such as  $\square$  and  $\diamond$ , extended naturally to OPTL operators. E.g.,  $\diamond^\uparrow \psi := \top \mathcal{U}^\uparrow \psi$ , and  $\square^\uparrow \psi := \neg \diamond^\uparrow \neg \psi$  are defined such that, if evaluated in position  $i$ , they mean that  $\psi$  holds in, respectively, at least one and all positions  $j$  with  $\chi(i, j)$  and not  $\vec{\chi}(i, j)$ .  $\diamond^\downarrow \psi := \top \mathcal{S}^\downarrow \psi$  and  $\square^\downarrow \psi := \neg \diamond^\downarrow \neg \psi$  are symmetric.

**Total/partial correctness** Formula  $\circ^{\text{ret}} \psi := \circ_\chi(\text{ret} \wedge \psi) \vee \circ(\text{ret} \wedge \psi)$ , evaluated in a **call**, states that it is closed by a **ret** in which  $\psi$  holds. Thus, formula

$$\square[\text{call} \implies \circ^{\text{ret}} \top]$$

holds if all procedures terminate, while it is false if there is an uncaught exception.

We can also express Hoare-style pre- and post-conditions [97], which are used in most classical verification techniques. Formula

$$\square[(\text{call} \wedge p_A \wedge \rho) \implies \circ^{\text{ret}} \theta]$$

expresses *total correctness*, i.e. whenever pre-condition  $\rho$  holds when procedure  $p_A$  is called, the latter terminates normally, with post-condition  $\theta$  holding. Instead,

$$\square[(\text{call} \wedge p_A \wedge \rho \wedge \circ^{\text{ret}} \top) \implies \circ^{\text{ret}} \theta]$$

expresses *partial correctness*, i.e. the post-condition has to hold only when the procedure terminates normally.

**Exception Safety** We can do the same with **exc** statements:

$$\circ^{\text{exc}} \psi := \circ_\chi(\text{exc} \wedge \psi) \vee \circ(\text{exc} \wedge \psi),$$

evaluated in a **call**, states that it is terminated by a **exc** in which  $\psi$  holds. Formula  $\Box[(\mathbf{call} \wedge p_A) \implies \neg \circ^{\mathbf{exc}} \top]$  is the requirement that procedure  $p_A$  never throws an exception, also known as the *no-throw guarantee*. When partial correctness is generalized to exceptions, we get *exception safety* [1], an important concept for the correctness of, especially, C++ programs. *Weak* (or *basic*) exception safety requires that, when a C++ class member function terminates exceptionally, all class invariants are preserved (so the class instance is still in a functional state), and no resources are leaked. *Strong* exception safety adds the requirement that, in case of exceptional exit, the operation is aborted, and the state of the instance remains the same as it was before the member function was called. If  $\theta$  is a class invariant, formula  $\Box[(\mathbf{call} \wedge p_A \wedge \theta \wedge \circ^{\mathbf{exc}} \top) \implies \circ^{\mathbf{exc}} \theta]$  expresses weak exception safety for  $p_A$ , and strong exception safety if  $\theta$  represents the whole state of the class instance.

**Function-Local Requirements** With OPM  $M_{\mathbf{call}}$ , the **call** of a procedure is the left context of non-maximal chains whose right contexts are the **calls** and **hans** it issues (except the first one). Thus,

$$\diamond^{loc} \psi := \diamond^\uparrow((\mathbf{call} \wedge \psi) \vee (\mathbf{han} \wedge \Box^\uparrow(\mathbf{call} \implies \psi)))$$

is true if  $\psi$  holds in one of the **calls** issued by the procedure represented by the **call** in which it is evaluated (even if they are guarded by a **han**). A “globally” version of this operator can be defined symmetrically. If we mark with  $wr_X$  the fact that a function writes the program variable  $X$ , then  $\Box[\mathbf{call} \implies (\diamond^{loc} wr_X \implies \circ^{\mathbf{exc}} \top)]$  requires that any function whose sub-calls write to  $X$  is terminated by an exception.

We did not include “internal” positions in OPM  $M_{\mathbf{call}}$  for conciseness of the examples, but they could be defined with an  $\dot{=}$ -circularity as we will do in OPM  $M^{NW}$  from Section 7.1, in order to represent program instructions that do not concern function invocation or termination. Then, function-local requirements on such positions could be easily expressed with a  $\mathcal{U}^{\dot{=}}$  operator.

**Stack Inspection** *Stack Inspection* gathers a wide range of requirements concerning the sequence of function frames that are present on the program’s stack at a certain point of the execution. It is used to enforce security policies in, e.g., Java programs [76, 102]. With the shortcut

$$\varphi \mathcal{S}^{\mathbf{call}} \psi := (\mathbf{call} \implies \varphi) \mathcal{S}^{\leq \dot{=}} (\mathbf{call} \wedge \psi)$$

we get a since operator that only considers **calls** of procedures whose instances are active when the statement at the current word position is executed. This operator is similar to the *call since* of CaRet [8], but it can also work in the presence of exceptions. We also add the related back operator  $\ominus^{\mathbf{call}} \psi := (\neg \mathbf{call}) \mathcal{S}^{\leq \dot{=}} (\mathbf{call} \wedge \psi)$ , which enforces  $\psi$  in the **call** of the function on top of the stack at the point in the execution represented by a word position. Due to the separation property of LTL [86], these operators can express all first-order properties on the stack trace, subsuming the formalism of [102].

A typical requirement of this class is the following: function  $p_A$  can only be called by functions with privilege level  $l_1$ , and not by those with the lower privilege  $l_2$ . We can check this with formula  $\Box[(\mathbf{call} \wedge p_A) \implies (\neg l_2) \mathcal{S}^{\mathbf{call}} l_1]$ . This is also expressible in CaRet, but in OPTL we may additionally state that, if this requirement is violated, an exception is thrown:  $\Box[(\mathbf{call} \wedge p_A \wedge (\neg l_1) \mathcal{S}^{\mathbf{call}} l_2) \implies \circ^{\mathbf{exc}} \top]$ . With a similar

operator, formula  $\Box[\text{call} \wedge p_B \implies \text{call } S^{\leq} \text{han}]$  states that all calls to procedure  $p_B$  must be guarded by a **han** statement catching the exceptions they may throw.

With hierarchical operators, we can formulate requirements limited to the procedures in the stack trace that have been terminated by a single **exc** statement. Shortcut  $\Diamond^\downarrow \psi$  is true in a **exc** position if  $\psi$  holds in at least one of the functions it terminates. So, we may express restrictions on which procedures may raise exceptions. E.g.,  $\Box[\text{exc} \implies \neg \Diamond^\downarrow p_B]$  means that procedure  $p_B$  cannot be terminated by an exception. Formula  $\Box[\text{exc} \implies (\neg l_2) S^\uparrow l_1]$  means that only functions with privilege level  $l_1$  may throw, and those with level  $l_2$  may only do so through an invocation to one of them.

*Remark 6.10.* Those of the requirements listed above that deal with exceptions are not 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. As we noted in Section 3.2.4, VPLs can be used to model function calls and returns, which are in a one-to-one relation, but they cannot model the example of Figure 6.1, because they have no way to express the many-to-one relation that holds between multiple function calls terminated by an exception, and the exception itself.

## 6.5 Model Checking and Satisfiability

A model checking procedure for OPTL has been presented in [54]. It is based on the automata-theoretic framework we presented for LTL in Section 2.2.2, except OPAs (and  $\omega$ OPBAs) are used instead of NBAs. Because of the limitations in OPTL's expressiveness shown in Section 7.2, we did not implement its model checking, and here we only outline the main results.

**Theorem 6.11** ([54]). *For any OPTL formula  $\varphi$ , it is possible to effectively build an OPA (or an  $\omega$ OPBA) that accepts models of  $\varphi$  with at most  $2^{O(|\varphi|)}$  states.*

Thus, the complexity of model checking is not greater than that of competing logics on nested words, such as NWTL.

Since in Section 7.1 we prove that OPTL can express all NWTL formulas, we can use the same lower bounds for the complexity of decision problems [12]. Together with Theorem 6.11 this allows us to state

**Theorem 6.12.** *OPTL model checking and satisfiability are EXPTIME-complete.*

# Chapter 7

## OPTL Expressiveness

In this chapter, we compare OPTL to other logics in order to assess its expressive power. In Section 7.1, we show that it is strictly more expressive than NWTL, which is equivalent to FOL on nested words; in Section 7.2 we compare it to FOL on OP words, and show that, unfortunately, OPTL is not as expressive.

### 7.1 Relationship with Nested Words

We now explore the relationship between OPTL and NWTL [12]. NWTL is based on the VPL family, which is strictly contained in OPLs. In [64, 119, 121] the relations between the two families are discussed in depth both from a mathematical and an application point of view: building OPTL formulas describing OPLs that are not VPLs is a trivial job. This proves that there exist languages not expressible in NWTL that can be expressed in OPTL. To prove that OPTL is more expressive than NWTL, we first show a way to translate a nested word into an “almost isomorphic” OPTL structure; then, we give a translation schema for NWTL formulas into equivalent OPTL ones.

Throughout this section, we refer to Section 3.2.4 for the definitions of nested words (Definition 3.3) and NWTL.

Given a set of atomic propositions  $\Lambda$ , and a nested word  $(w, \mu', \text{call}, \text{ret})$  over  $\mathcal{P}(\Lambda)$ , we consider the equivalent algebraic structure

$$NW = \langle U, (P_a)_{a \in \Lambda}, <, \mu, \text{call}, \text{ret} \rangle$$

where  $U$  is a set of word positions such that  $U = \{1, \dots, |w|\}$  if  $w$  is finite, and  $U = \mathbb{N} \setminus \{0\}$  if it is a nested  $\omega$ -word;  $<$  is the ordering of  $\mathbb{N}$ ;  $P_a$  is the set of positions labeled with  $a \in \Lambda$ . Relations  $\text{call}$  and  $\text{ret}$  are the same, and  $\mu$  is the same as  $\mu'$  except pairs containing  $-\infty$  and  $+\infty$  are removed. Thus, if  $\text{call}(i)$  (resp.  $\text{ret}(j)$ ) but for no  $j \in U$  (resp.  $i \in U$ ) we have  $\mu(i, j)$ , then  $i$  (resp.  $j$ ) is a *pending* call (resp. return).

Given any nested word  $NW$  as defined above, it is possible to build an equivalent algebraic structure for OPTL as  $OW = \langle U', M^{NW}, P' \rangle$ . Given  $U = \{1, \dots, n\}$ , we have  $U' = U \cup \{0, n+1\}$  (remove  $n+1$  if  $NW$  is a nested  $\omega$ -word). The set of propositional letters is  $AP = \Lambda \cup \Sigma$  with  $\Sigma = \{\text{call}, \text{ret}, \text{int}\}$ . For any  $i \in U$  we define  $P'(i) = \{a \in \Lambda \mid i \in P_a\} \cup \sigma(i)$ , where  $\sigma(i) = \{\text{call}\}$  iff  $\text{call}(i)$ ,  $\sigma(i) = \{\text{ret}\}$  iff  $\text{ret}(i)$ , and  $\sigma(i) = \{\text{int}\}$  otherwise. Finally, the OPM  $M^{NW}$  is

	call	ret	int
call	<	≐	<
ret	≐	>	≐
int	≐	>	≐

Figure 7.1: The OPM  $M^{NW}$ .

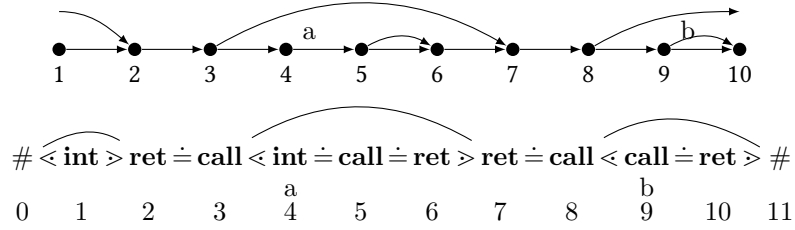


Figure 7.2: The top figure is the representation of a nested word example, and its translation into an OP word is shown below, using OPM  $M^{NW}$ .

shown in Figure 7.1. Note that  $M^{NW}$  is different from the OPM used in [64] to prove that VPL are contained in OPL.

An example nested word is shown in Figure 7.2, along with its translation into an OP word. In the translation, all the call positions form the contexts of a chain with the matched return, except for consecutive positions  $i, i + 1 \in U$  such that  $i$  is a call and  $i + 1$  a return. Therefore, we are able to use the chain relation to translate the matching relation of nested words, except for consecutive call/return positions, which have to be considered separately. Also, unmatched returns and calls form a chain with the first and last  $\#$  positions. This reasoning is formalized by the following lemmas.

**Lemma 7.1.** *For any two distinct positions  $i, j \in U, i < j$ , if  $\chi(i, j)$  holds then  $\text{call} \in P'(i)$  and  $\text{ret} \in P'(j)$ .*

*Proof.*  $\chi(i, j)$  means  $i$  and  $j$  are the context of a chain. According to Definition 4.9, position  $i$  must yield precedence to the next position,  $i + 1: i < i + 1$ . According to matrix  $M^{NW}$ , only call positions can yield precedence to any other position (unless  $\# \in P'(i)$ , which is not the case), therefore  $\text{call} \in P'(i)$ . Similarly, any position can take precedence from a return position only, and since  $j - 1$  must take precedence from  $j$ , we have  $\text{ret} \in P'(j)$ .  $\square$

In Lemma 7.2 we prove that relation  $\chi$  in  $OW$  is one-to-one if restricted to  $U$ . Note that this is not true for positions 0 and  $n + 1$ : 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 7.2.** *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  $i = i'$ .*

*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  $j_1, \dots, j_{n-1}, j_n \in U$  such that  $i < j_1 < \dots < j_{n-1} < j_n$  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 be of the form  $a^i[w_i a_{j_{n-1}} \dots]^{a_{j_n}}$ , hence we have  $a_i \leq a_{j_{n-1}}$ , and  $w_i$  is the body of chain  $a^i[w_i]^{a_{j_{n-1}}}$ . But since  $\chi(i, j_{n-1})$ , and because of Lemma 7.1  $\text{call} \in P'(i)$  and  $\text{ret} \in P'(j_{n-1})$ . Therefore, according to  $M^{NW}$ , we have  $a_i \doteq a_{j_{n-1}}$ , which contradicts our previous claim. The fact that there exists no position  $j \in U$  in which multiple chains starting in  $U$  end can be proved similarly.  $\square$

Lemma 7.3 shows that, when translating a nested word into an OP word, the  $\mu$  relation of the former is reflected into the  $\chi$  relation that holds in the latter. This is, as we observed previously, not true for consecutive positions.

**Lemma 7.3.** *For any  $i, j \in U$  such that  $j > i + 1$ , we have  $\mu(i, j)$  iff  $\chi(i, j)$ .*

*Proof.* The proof is carried out by induction on the nesting depth of the  $\mu/\chi$  relation. Suppose  $i, j \in U$  and  $\mu(i, j)$ . If all positions between  $i$  and  $j$  are internal (i.e. the nesting depth is 0), then  $\text{call} \in P'(i)$ ,  $\text{ret} \in P'(j)$  and  $\text{int} \in P'(k)$  for any  $i < k < j$ . The resulting word structure will be  $i < i + 1 \doteq \dots \doteq j - 1 \succ j$ , and  $\chi(i, j)$ . Now, suppose  $\chi(i, j)$  is a simple chain. According to  $M^{NW}$ , if  $i < i + 1$  then  $\text{call} \in P'(i)$  and  $\text{int} \in P'(i + 1)$ , and similarly  $\text{int} \in P'(k)$  with  $i < k < j$  and  $\text{ret} \in P'(j)$ : in  $NW$   $i$  is a call and  $j$  its matched return, so we have  $\chi(i, j) \implies \mu(i, j)$ .

Suppose  $\mu(i, j)$ , and there are other call and return positions between  $i$  and  $j$ . By the definition of  $\mu$ , such positions are balanced, i.e., each  $\text{call}$  between  $i$  and  $j$  has a matching  $\text{ret}$  before  $j$ . So, the nested word between  $i$  and  $j$  has the form  $c_0 a_1 x_1 \dots x_{n-1} a_{n-1} r_n$  where  $c_0 = i$ ,  $r_n = j$ ,  $\text{call}(c_0)$ ,  $\text{ret}(r_n)$ , and for any  $p$  such that  $0 < p < n$ ,  $a_p = \varepsilon$  or it is an internal position, and either  $x_p = \varepsilon$  or  $x_p$  is a nested word of the form  $x_p = c_p y_p r_p$ , with  $\text{call}(c_p)$  and  $\text{ret}(r_p)$ . Once translated into an OP word, this results in the composed chain  $c^0[a_1 x_1 \dots x_{n-1} a_{n-1}]^{r_n}$ , so  $\chi(i, j)$ . Indeed, according to our translation,  $\text{call} \in P'(i)$  and, if  $a_1 \neq \varepsilon$ , then  $a_1 = i + 1$  and  $\text{int} \in P'(i + 1)$ , which implies  $c_0 \leq a_1$ ; if  $a_1 = \varepsilon$  then  $i$  is followed by  $c_1 = i + 1$  (which is part of  $x_1$ ) and  $\text{call} \in P'(i + 1)$ , so  $c_0 \leq c_1$ . Similarly, either  $a_{n-1} \succ r_n$  or  $r_{n-1} \succ r_n$ . Also, each  $x_p$  forms a chain  $c^p[y_p]^{r_p}$  by itself, while  $c^0[a_1 c_1 r_1 \dots c_{n-1} r_{n-1} a_{n-1}]^{r_n}$  is a simple chain.

The fact that if  $\chi(i, j)$  is a composed chain, then  $\mu(i, j)$  can be proved analogously, keeping into account the results of Lemmas 7.1 and 7.2.  $\square$

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 7.4.** *Given any two word positions  $i, j \in U$ ,  $i \leq j$ , the summary path between  $i$  and  $j$  in  $NW$  coincides with the OP-summary path between the same positions based on precedence relations  $\Pi = \{\leq, \doteq, \succ\}$  in  $OW$ .*

*Proof.* Recall that a summary path between  $i, j \in U$ ,  $i < j$ , is a sequence  $i = i_1 < i_2 < \dots < i_k = j$  such that for all  $p < k$

$$i_{p+1} = \begin{cases} r(i_p) & \text{if } i_p \text{ is a matched call and } j \geq r(i_p); \\ i_p + 1 & \text{otherwise;} \end{cases} \quad (7.1)$$

$$i_{p+1} = \begin{cases} r(i_p) & \text{if } i_p \text{ is a matched call and } j \geq r(i_p); \\ i_p + 1 & \text{otherwise;} \end{cases} \quad (7.2)$$

where  $r(i_p)$  is the only position such that  $\mu(i_p, r(i_p))$ , if it exists.

OP-summary paths are defined in Definitions 6.6 and 6.7. Note that, because of Lemma 7.2, we have  $\chi(i, j) \iff \overleftarrow{\chi}(i, j) \iff \overrightarrow{\chi}(i, j)$  for any  $i, j \in U$ , and forward and backward OP-summary paths coincide. Moreover, due to Lemma 7.3,

$\chi(i, j) \iff \mu(i, j)$  if  $j > i + 1$ : case 7.1 of the definition above coincides with the first case of Definition 6.6. If  $j = i + 1$  and  $\mu(i, j)$ ,  $\chi(i, j)$  does not hold, but this case is subsumed by the second case of Definition 6.6. The latter also incorporates case 7.2 of the definition of summary path in NWTL, because set  $\Pi$  includes all possible precedence relations. Since we have proved that the two definitions coincide for each single step of the path, it is possible to inductively prove that the two kinds of paths actually coincide.  $\square$

After establishing a certain degree of isomorphism between nested words and their OPTL translations, we can give a translation schema from NWTL to OPTL formulas.

**Theorem 7.5** (NWTL  $\subseteq$  OPTL). *Given an NWTL formula  $\varphi$ , it is possible to translate it to an OPTL formula  $\varphi'$  of length linear in  $|\varphi|$  such that, for any nested word  $w$  and position  $i$ , if  $w$  is translated into an OP structure  $w'$  as described at the beginning of Section 7.1, then  $(w, i) \models \varphi$  iff  $(w', i) \models \varphi'$ , with  $i \in U$ .*

*Proof.* Let  $w'$  be an OP word built from  $w$  as described above. For any NWTL formula  $\varphi$  we define  $\varphi' = \alpha(\varphi)$  inductively as follows, for non-trivial operators:

- $\alpha(\bigcirc_{\mu} \varphi) = \bigcirc_{\chi} \alpha(\varphi) \vee (\mathbf{call} \wedge \bigcirc(\mathbf{ret} \wedge \alpha(\varphi)))$ . The validity of this translation trivially follows from Lemma 7.3. Unfortunately, the double repetition of  $\alpha(\varphi)$  may cause an exponential blowup in the worst-case length of the translation.

The following more complex translation does not suffer from this issue:

$$\alpha(\bigcirc_{\mu} \varphi) = \mathbf{call} \wedge \mathbf{call} U^{\doteq} (\mathbf{ret} \wedge \alpha(\varphi)).$$

To better explain it, let  $\gamma := \mathbf{call} U^{\doteq} (\mathbf{ret} \wedge \alpha(\varphi))$ . In NWTL we have  $(w, i) \models \bigcirc_{\mu} \varphi$  iff there exists  $j \in U$  such that  $\mu(i, j)$  and  $(w, j) \models \varphi$ . When referring to the OP structure, we must distinguish between a few mutually exclusive cases:

- *A position  $j$  such that  $\mu(i, j)$  exists and  $j > i + 1$ .* Then, by Lemma 7.3 also  $\chi(i, j)$  holds and, because of Lemma 7.2, we have  $\overrightarrow{\chi}(i, j)$ . 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  $\gamma$  is satisfied. Furthermore, this is the only path in which  $\gamma$  is true; in fact, paths terminating in a position strictly between  $i$  and  $j$  are forbidden by allowing only the  $\doteq$  relation, and all paths surpassing  $j$  must include it, but  $\mathbf{call} \notin P'(j)$  falsifies  $\gamma$ .
  - *A position  $j$  such that  $\mu(i, j)$  exists, but  $j = i + 1$ .* In this case, we have  $\mathbf{call} \in P'(i)$  and  $\mathbf{ret} \in P'(j)$ , so  $i \doteq j$ , and the path made of  $i$  and  $j$  is valid.
  - *$i$  is a pending call.* Then  $\chi(i, n + 1)$ , but  $\mathbf{ret} \notin P'(n + 1)$ , which falsifies  $\gamma$ .
  - *$i$  is an internal position.*  $(w, i) \not\models \bigcirc_{\mu} \varphi$  because position  $i$  is not a matched call, so  $\mathbf{int} \in P'(i)$  and  $\gamma$  is false because  $\mathbf{call}, \mathbf{ret} \notin P'(i)$ .
  - *$i$  is a return.* If  $\mathbf{ret}(i)$ , then  $\mathbf{call}$  in  $\alpha(\bigcirc_{\mu} \varphi)$  is false in  $i$ .
- $\alpha(\ominus_{\mu} \varphi) = \mathbf{ret} \wedge \mathbf{ret} S^{\doteq} (\mathbf{call} \wedge \alpha(\varphi))$ . The argument that justifies this equivalence is similar to the previous one. Again, the more straightforward translation  $\alpha(\ominus_{\mu} \varphi) = \ominus_{\chi} \alpha(\varphi) \vee (\mathbf{ret} \wedge \ominus(\mathbf{call} \wedge \alpha(\varphi)))$  causes an exponential blowup in formula length.

- $\alpha(\varphi \mathcal{U}^\sigma \psi) = \alpha(\varphi) \mathcal{U}^{\geq \dot{=} \leq} \alpha(\psi)$ : from Lemma 7.4 we know that the set of summary paths starting from position  $i$  in  $w$  corresponds to the set of OP-summary paths starting from  $i$  in  $w'$ , which implies the OP-summary operators coincide with their NWTL counterparts.
- $\alpha(\varphi \mathcal{S}^\sigma \psi) = \alpha(\varphi) \mathcal{S}^{\geq \dot{=} \leq} \alpha(\psi)$ : the justification is analogous to the until case.

By induction on the syntactic structure of  $\varphi$ , we can conclude that  $(w, i) \models \varphi$  iff  $(w', i) \models \varphi'$ , and consequently  $\text{NWTL} \subseteq \text{OPTL}$ .  $\square$

For example, consider formula  $\varphi = (\neg a) \mathcal{U}^\sigma b$ : the nested word of Figure 7.2 satisfies  $\varphi$  because of summary path 1-2-3-7-8-9.  $\varphi$  is translated into  $\varphi' = (\neg a) \mathcal{U}^{\geq \dot{=} \leq} b$ , which is satisfied by the OPTL structure of Figure 7.2, where the OP-summary path covering the same positions above witnesses its truth.

The above translation can easily be extended to OP  $\omega$ -words.

It is easy to see that the OPTL semantics of Section 6.3 is expressible in FOL, so  $\text{OPTL} \subseteq \text{FOL}$ . Thus, by the FO-completeness result for NWTL of [12], we conclude

**Corollary 7.6.** *If OPTL is restricted to OPM  $M^{\text{NW}}$ , we have  $\text{OPTL} = \text{NWTL} = \text{FOL}$ .*

This does not mean that OPTL has the same expressive power of NWTL. In fact, the greater expressiveness of OPTL derives from that of OPLs with respect to VPLs, and it consists in using more general OPMs, such as  $M_{\text{call}}$ . In general, OPTL formulae exploiting the fact that the  $\chi$  relation is not exclusively one-to-one, such as those pointed out in Section 6.4, express properties not expressible in NWTL. Thus, also considering that CaRet is expressible in FOL [12], we can state the following:

**Corollary 7.7.**  *$\text{CaRet} \subseteq \text{NWTL} \subset \text{OPTL}$ .*

## 7.2 Relationship with First-Order Logic

After proving that OPTL is more expressive than context-free logics in the state-of-the-art, we point out its limitations by comparing it with FOL on OP words.

### 7.2.1 OPTL's Limitations

We start by pointing out that in OP-summary until and since operators, the precedence relations checked on chain contexts are fixed, so the user can control whether such paths go up or down in a word's syntax tree only partially. This makes it difficult to express function-local properties limited to a single subtree in OPTL.

For example, suppose we want to express the requirement that if an exception is thrown, it is always caught, and procedure  $p_A$  is called at some point inside the resulting **han-exc** block. This is easily expressible in FOL as

$$\alpha := \forall x(\mathbf{exc}(x) \implies \exists y(y < x \wedge \chi(y, x) \wedge \mathbf{han}(y) \wedge \exists z(y \leq z \leq x \wedge p_A(z))).$$

One could try to translate it into OPTL with a formula such as

$$\beta := \square(\mathbf{exc} \implies \ominus_\chi(\mathbf{han} \wedge \top \mathcal{U}^{\leq \dot{=} \geq} p_A)).$$

Consider the OP word of Figure 7.3. When the until in  $\beta$  is evaluated in the **han** of position 3, its paths can only consider positions 4 and 5, because paths touching such

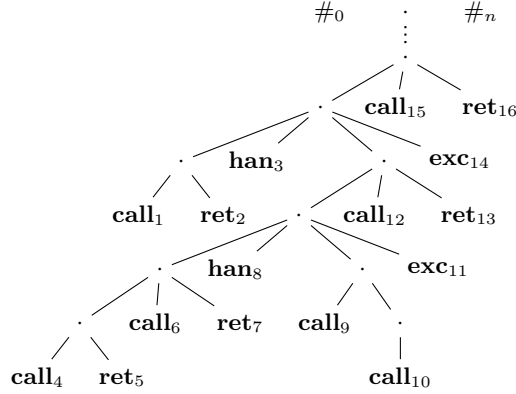


Figure 7.3: Example OP word on OPM  $M_{\text{call}}$ , represented as a syntax tree.

positions cannot pass the  $>$  relation between 5 and 6. Its paths cannot jump between chain contexts in the  $<$  relation, and cannot reach positions 6, 8, and 12 in this way. If  $p_A$  held in positions 6 and 10,  $\alpha$  would be true in the word, but  $\beta$  would be false. Replacing the until with  $\top \mathcal{U}^{< \dot{=} >} p_A$  overcomes such issues, but it introduces another one: its paths would go past position 14, going outside of the subtree. Thus, if  $p_A$  held only in position 15,  $\alpha$  would be false, but this variant of  $\beta$  would hold.

The above intuition about OPTL's weaknesses is made formal in the next section.

## 7.2.2 OPTL is not expressively complete

In this section, we prove that no OPTL formula is equivalent to FOL formula

$$\alpha'(x) := \exists y(\chi(x, y) \wedge \exists z(y \leq z \leq x \wedge p_A(z))).$$

The proof is quite elaborate, which is unsurprising, since the analogous problem of the comparison between CaRet and NWTL is still open.

First, we prove the following

**Lemma 7.8** (Pumping Lemma for OPTL). *Let  $\varphi$  be an OPTL formula and  $L$  an OPL, both defined on a set of atomic propositions  $AP$  and an OPM  $M_{AP}$ . Then, for some positive integer  $n$ , for each  $w \in L$ ,  $|w| \geq n$ , there exist strings  $u, v, x, y, z \in \mathcal{P}(AP)^*$  such that  $w = uvxyz$ ,  $|vy| > 1$ ,  $|vxy| \leq n$  and for any  $k > 0$  we have  $w' = uv^kxy^kz \in L$ ; for any  $0 \leq j \leq k$  and  $0 \leq i < |v|$  we have  $(w, |u|+i) \models \varphi$  iff  $(w', |u|+j|v|+i) \models \varphi$ , and for any  $0 \leq i < |y|$  we have  $(w, |uv^kx|+i) \models \varphi$  iff  $(w', |uv^kx|+j|y|+i) \models \varphi$ .*

*Proof.* Given a word  $w \in L$ , we define  $\lambda(w)$  as the word of length  $|w|$  such that, if position  $i$  of  $w$  is labeled with  $a$ , then the same position in  $\lambda(w)$  is labeled with  $(a, 1)$  if  $(w, i) \models \varphi$ , and with  $(a, 0)$  otherwise. Let  $\lambda(L) = \{\lambda(w) \mid w \in L\}$ , and  $\lambda^{-1}$  is such that  $\lambda^{-1}(\lambda(w)) = w$ . If we prove that  $\lambda(L)$  is context-free, from the classic Pumping Lemma [93] follows that, for some  $n > 0$ , for all  $\hat{w} \in \lambda(L)$  there exist strings  $\hat{u}, \hat{v}, \hat{x}, \hat{y}, \hat{z} \in (\mathcal{P}(AP) \times \{0, 1\})^*$  such that  $\hat{w} = \hat{u}\hat{v}\hat{x}\hat{y}\hat{z}$ ,  $|\hat{v}\hat{y}| > 1$ ,  $|\hat{v}\hat{x}\hat{y}| \leq n$  and for any  $k > 0$  we have  $\hat{w}' = \hat{u}\hat{v}^k\hat{x}\hat{y}^k\hat{z} \in \lambda(L)$ . The claim follows by applying  $\lambda^{-1}$  to such strings, and the word positions in which  $\varphi$  holds in  $\lambda^{-1}(\hat{w})$  are those labeled with 1.

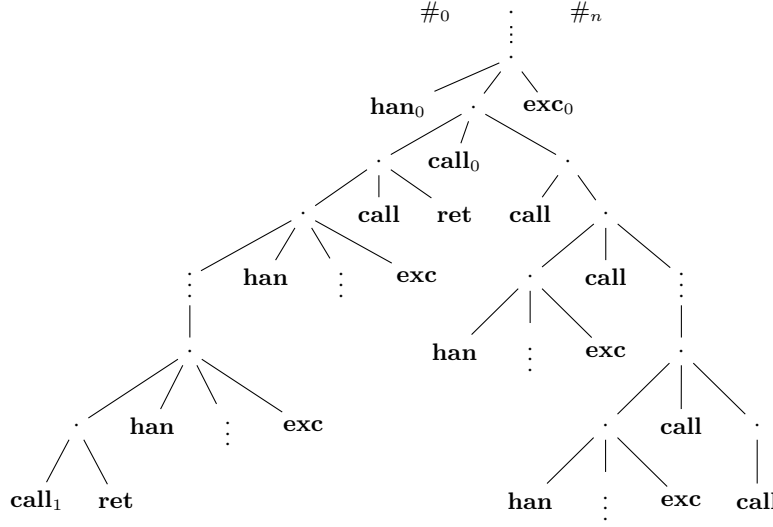


Figure 7.4: Structure of a word in  $L_{\text{call}}$ .

To prove that  $\lambda(L)$  is context-free, we use the OPTL model checking construction given in [54], which yields an OPA  $\mathcal{A}_\varphi = (\mathcal{P}(AP), M_{AP}, Q, I, F, \delta)$  accepting models of  $\varphi$ . The states of  $\mathcal{A}_\varphi$  are elements of the set  $\text{Cl}(\varphi)$ , which contains  $\varphi$  and all its subformulas. Given a word  $w$  compatible with  $M_{AP}$ , the accepting computations of  $\mathcal{A}_\varphi$  are such that, for each  $0 \leq i < |w|$ , the state of  $\mathcal{A}_\varphi$  prior to reading position  $i$  contains  $\psi \in \text{Cl}(\varphi)$  iff  $(w, i) \models \psi$ .

Thus, we build OPA  $\mathcal{A}_{\lambda(\mathcal{P}(AP)^*)} = (\mathcal{P}(AP) \times \{0, 1\}, M_{AP}, Q, I', F, \delta')$  that reads words on  $(\mathcal{P}(AP) \times \{0, 1\})^*$  and accepts  $\lambda(\mathcal{P}(AP)^*)$  as follows:

- $I'$  is the set of all states in  $Q$  not containing past operators (and possibly  $\varphi$ );
- $\delta'_{push}$  is such that if  $(\Phi, a, \Theta) \in \delta_{push}$ , then  $(\Phi, (a, 1), \Theta) \in \delta'_{push}$  if  $\varphi \in \Phi$ , and  $(\Phi, (a, 0), \Theta) \in \delta'_{push}$  otherwise;
- $\delta'_{shift}$  is derived from  $\delta_{shift}$  similarly;
- $\delta'_{pop} = \delta_{pop}$ .

Since  $L$  is an OPL, there exists an OPA  $\mathcal{A}_L$  accepting it.  $\mathcal{A}_L$  can be easily modified to obtain  $\mathcal{A}'_L$ , an OPA accepting all words  $\hat{w} \in (\mathcal{P}(AP) \times \{0, 1\})^*$  such that the underlying word  $w \in \mathcal{P}(AP)^*$  is in  $L$ . Language  $\lambda(L)$  is the intersection between the language accepted by  $\mathcal{A}'_L$ , and  $\lambda(\mathcal{P}(AP)^*)$ . Since OPLs are closed under intersection,  $\lambda(L)$  is also an OPL.  $\square$

Let  $L_{\text{call}}$  be the max-language generated by OPM  $M_{\text{call}}$ , with the addition that  $p_A$  may appear in any word position. We prove the following:

**Theorem 7.9.** *Given the FOL formula  $\alpha'(x)$ , for every OPTL formula  $\varphi$  there exist a word  $w \in L_{\text{call}}$  and an integer  $0 \leq i < |w|$  such that either  $(w, i) \models \alpha'(x)$  and  $(w, i) \not\models \varphi$ , or  $(w, i) \not\models \alpha'(x)$  and  $(w, i) \models \varphi$ .*

*Proof.* Figure 7.4 shows the structure of the syntax trees of words in a subset of  $L_{\text{call}}$ . Dots between **han**–**exc** pairs can be replaced with repetitions of the whole tree structure, and other dots with the repetition of surrounding tree fragments (e.g., **call**–**ret** or **han**–**exc**). **han**<sub>0</sub> is the position in which  $\varphi$  and  $\alpha'(x)$  are evaluated, and **exc**<sub>0</sub> is its matched **exc**. **call**<sub>1</sub> is the **call** right after **han**<sub>0</sub>, and **call**<sub>0</sub> is the one at the highest level of the subtree between **han**<sub>0</sub> and **exc**<sub>0</sub>. **calls** between **call**<sub>0</sub> and **exc**<sub>0</sub> do not have a corresponding **ret**, and are terminated by **exc**<sub>0</sub>. The word delimited by **han**<sub>0</sub> and **exc**<sub>0</sub> is itself part of a larger tree with the same structure. For  $\varphi$  to be equivalent to  $\alpha'(x)$ , it must be able to

1. look for the symbol  $p_A$  in all positions between **han**<sub>0</sub> and **exc**<sub>0</sub>; and
2. not consider any positions before **han**<sub>0</sub> or after **exc**<sub>0</sub>.

In the following, we show that any OPTL formula  $\varphi$  fails to satisfy both requirements:

1. one can hide  $p_A$  in one of the positions not covered, so that  $\varphi$  is false in **han**<sub>0</sub>, but  $\alpha'(x)$  is true; or
2. put  $p_A$  in one of the positions outside **han**<sub>0</sub>–**exc**<sub>0</sub> reached by  $\varphi$ , so that it is true in **han**<sub>0</sub>, but  $\alpha'(x)$  is not.

If  $\varphi$  is evaluated on position **han**<sub>0</sub>, it must contain some modal operator based on paths that reach each position between **han**<sub>0</sub> and **exc**<sub>0</sub>. The length of the word between **han**<sub>0</sub> and **exc**<sub>0</sub> has no limit, so  $\varphi$  must contain at least an until operator, which may be a LTL until, an OPTL hierarchical until, or an OP-summary until  $\mathcal{U}^\Pi$ . For  $\mathcal{U}^\Pi$ , we must have  $\triangleright \in \Pi$ , or the path would not be able to reach past position **ret**<sub>1</sub> (remember that an OP-summary path cannot skip chains with contexts in the  $\triangleleft$  relation). The presence of  $\triangleright$  allows the OP-summary until to reach positions past **exc**<sub>0</sub>: the formula could be true if  $p_A$  appears after **exc**<sub>0</sub>, but not between **han**<sub>0</sub> and **exc**<sub>0</sub>, unlike  $\alpha'(x)$ . To avoid this, the path must be stopped earlier, by embedding an appropriate subformula as the left operand of the until.

Suppose there exists a formula  $\psi$  that is true in **exc**<sub>0</sub>, and false in all positions between **han**<sub>0</sub> and **exc**<sub>0</sub>. By Lemma 7.8, there exists an integer  $n$  such that for any  $w \in L_{\text{call}}$  longer than  $n$  there is  $w' = uv^kxy^kz \in L_{\text{call}}$ , for some  $k > 0$ , such that either (a)  $\psi$  never holds in  $v^kxy^k$ , or (b) it holds at least  $k$  times in there. We can take  $w$  such that **han**<sub>0</sub> and **exc**<sub>0</sub> both appear after position  $n$ , and they contain nested **han**–**exc** pairs. In case (a),  $\psi$  cannot distinguish **exc**<sub>0</sub> from nested **exc**s, so a  $\varphi$  based on  $\psi$  is not equivalent to  $\alpha'(x)$  in **han**<sub>0</sub>. The same can be said in case (b), by evaluating  $\varphi$  in a **han** from  $v^i$  with  $i < k$ . In this case, also chaining multiple untils, each one ending in a position in which  $\psi$  holds, does not work, as  $k$  can be increased beyond the finite length of any OPTL formula.

The above argument holds verbatim for LTL until, and does not change if we prepend LTL or abstract next operators to the until, because the length of the branch between **call**<sub>1</sub> and **call**<sub>0</sub> is unlimited. The argument for using since operators starting from **exc**<sub>0</sub> is symmetric. If both until and since operators are used, it suffices to apply the Pumping Lemma twice (one for until and one for since), and take a value of  $k$  large enough that a part of the string cannot be reached by the number of until and since operators in the formula. If hierarchical operators are used, the argument does not change, as they still need nested until or since operators to cover the whole subtree.

This argument also holds when the formula contains (possibly nested) negated until operators. This is trivial if their paths cannot reach part of the subtree between

**han**<sub>0</sub> and **exc**<sub>0</sub>. If, instead, they can reach positions past **exc**<sub>0</sub>, we can build a word with  $p_A$  in one of such positions, but not between **han**<sub>0</sub> and **exc**<sub>0</sub>. To distinguish it from a word with  $p_A$  between **han**<sub>0</sub> and **exc**<sub>0</sub>, the formula would need a subformula that can distinguish positions between **han**<sub>0</sub> and **exc**<sub>0</sub> from those outside, which would contradict Lemma 7.8.

Until now, we have proved that a formula equivalent to  $\alpha'(x)$  cannot contain until or since operators that stop exactly at **exc**<sub>0</sub>. However, they could be stopped earlier. In the following, we show that any such OPTL formula can only work for words of a limited length. Hence, no OPTL formula is equivalent to  $\alpha'(x)$  on all words in  $L_{\text{call}}$ .

Let  $w \in L_{\text{call}}$ , and  $x = \mathbf{han} y \mathbf{exc}$  a subword of  $w$  with the structure of Figure 7.4, in which each **han** has a matched **exc**, and conversely. We define  $h_{\text{exc}}(x) = 0$  if  $y$  contains no positions labeled with **han** or **exc** (hence, only **calls** and **rets**). Otherwise, let  $x' = \mathbf{han} y' \mathbf{exc}$  be the proper subword of  $y$  with the maximum value of  $h_{\text{exc}}(x')$ : we set  $h_{\text{exc}}(x) = h_{\text{exc}}(x') + 1$ .

We prove by induction on  $h_{\text{exc}}(x)$  that any OPTL formula evaluated in the first position of  $x$  must contain nested until or since operators with a nesting depth of at least  $2 \cdot h_{\text{exc}}(x) + 1$  to be equivalent to  $\alpha'(x)$ .

If  $h_{\text{exc}}(x) = 0$ , at least one until or since operator is needed, as the length of  $x$  is not fixed. E.g., OPTL formula  $\neg \mathbf{exc} \mathcal{U}^{\langle \Leftarrow \rangle} p_A$  suffices.

If  $h_{\text{exc}}(x) = n > 0$ , Figure 7.4 shows a possible structure of  $x$ . Any OP-summary until in the formula must be nested into another operator, or its paths would jump to, and go past, the last position of  $x$  (**exc**<sub>0</sub>). An OP-summary until could be, instead, nested into any number of nested next operators, to be evaluated in one of the positions shown in Figure 7.4 between **han**<sub>0</sub> and **call**<sub>0</sub>. (The tree fragments between **call**<sub>1</sub> and **call**<sub>0</sub> can be repeated enough times so that the next operators alone cannot reach **call**<sub>0</sub>.) As noted earlier, any such summary until must allow for paths with consecutive positions in the  $\succ$  relation. It may also jump to **exc**<sub>0</sub> by following the chain relation, because **call**  $\succ$  **exc**. Hence, the until must be stopped earlier by choosing appropriate operands (e.g.,  $\neg \circ_{\chi} \mathbf{exc}$  as the left operand). However, this leaves the subword between **call**<sub>0</sub> and **exc**<sub>0</sub> unreached, so any of its positions containing (or not)  $p_A$  would be ignored. This can only be solved with another until operator, so at least two are needed. If it is a summary until, then it must not allow the  $\succ$  relation, or it could, again, escape **exc**<sub>0</sub> (e.g. by skipping chains between **calls** and **exc**<sub>0</sub>). The argument can be extended by considering an LTL until which stops anywhere before **exc**<sub>0</sub>, or since operators evaluated in **exc** (e.g., nested in a  $\circ_{\chi}$  operator). The same can be said for hierarchical operators, which can cover only a part of the subtree if used alone.

Let  $x' = \mathbf{han} y' \mathbf{exc}$  be a proper subword of  $y$  with  $h_{\text{exc}}(x') = n - 1$ . Suppose it appears before **call**<sub>0</sub> (the other case is symmetric). It needs at least an until or since operator to be covered, which must not escape **han**<sub>0</sub> or **exc**<sub>0</sub>. The Pumping Lemma can be used to show that no OPTL formula can distinguish positions in  $x$  or  $x'$  from those outside. Thus, a formula with until or since operators that do not exit  $y'$  is needed. By the inductive hypothesis, it consists of at least  $2(n - 1) + 1$  until or since operators, thus  $x$  needs  $2n + 1$  of them.

Note that the argument also holds if the until formulas are negated, because negation cannot change the type of paths considered by an operator, and cannot decrease the number of nested untils needed to cover the whole subtree.  $\square$



## Chapter 8

# POTL Syntax and Semantics

The negative results on OPTL’s expressiveness motivate us to define a more expressive temporal logic on OPLs. The resulting logic is called Precedence Oriented Temporal Logic (POTL), and is expressively complete.

In this chapter, we first give a high-level overview of POTL; then we formally define its syntax and semantics on finite OP words in Section 8.1; and we extend such definitions to OP  $\omega$ -words in Section 8.2; finally, we give a few examples of how to use POTL in practice in Section 8.3.

POTL is based on OP words, the same algebraic structure on which OPTL is based, and for which we refer the reader to Section 6.1. We report the word of Figure 6.1 in Figure 8.1 and the ST of Figure 6.2 in Figure 8.2 for convenience in explaining our examples.

To express properties on OP words, we envisage until and since operators defined on two basic types of path. The first one is that of *summary paths*. By following the chain relation, summary paths may skip chain bodies, which correspond to the fringe of a subtree in the syntax tree. We distinguish between *downward* and *upward* summary paths (respectively DSP and USP). Both kinds can follow both the  $<$  and the  $\chi$  relations; DSPs can enter a chain body but cannot exit it so that they can move only downward in a ST or remain at the same level; conversely, USPs cannot enter one but can move upward by exiting the current one. In other words, if a position

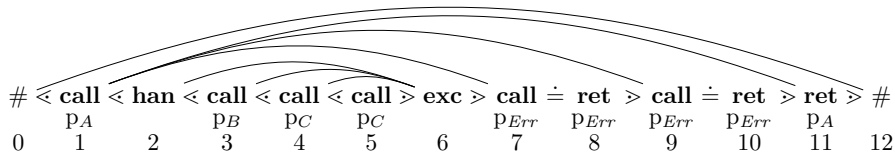


Figure 8.1: The example word  $w_{ex}$  from Chapter 4 as an OP word. Chains are highlighted by arrows joining their contexts; structural labels are in bold, and other atomic propositions are shown below them.  $p_l$  means a **call** or a **ret** is related to procedure  $p_l$ . First, procedure  $p_A$  is called (pos. 1), and it installs an exception handler in pos. 2. Then, three nested procedures are called, and the innermost one ( $p_C$ ) throws an exception, which is caught by the handler. Two more functions are called and, finally,  $p_A$  returns.

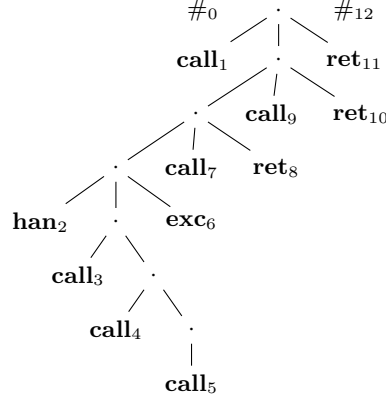


Figure 8.2: The ST of word  $w_{ex}$  (the same as Figure 4.4, but with position numbers). Dots represent non-terminals.

$k$  is part of a DSP, and there are two positions  $i$  and  $j$ , with  $i < k < j$  and  $\chi(i, j)$  holds, the next position in the DSP cannot be  $\geq j$ . E.g., two of the DSPs starting from position 1 in Figure 8.1 are 1-2-3, which enters chain  $\chi(2, 6)$ , and 1-2-6, which skips its body. USPs are symmetric, and some examples thereof are paths 3-6-7 and 4-6-7.

Since the  $\chi$  relation can be many-to-one or one-to-many, it makes sense to write formulas that consider only left contexts of chains that share their right context, or vice versa. Thus, the paths of our second type, named *hierarchical paths*, are made of such positions, but excluding outermost chains. E.g., in Figure 8.1, positions 2, 3 and 4 are all in the  $\chi$  relation with 6, so 3-4 is a hierarchical path ( $\chi(2, 6)$  is the outermost chain). Symmetrically, 7-9 is another hierarchical path. The reason for excluding the outermost chain is that, with most OPMs, such positions have a different semantic role than internal ones. E.g., positions 3 and 4 are both calls terminated by the same exception, while 2 is the handler. Positions 7 and 9 are both calls issued by the same function (the one called in position 1), while 11 is its return. This is a consequence of properties 3 and 4 of the  $\chi$  relation.

In the next section, we describe in a complete and formal way OPTL for finite-word OPLs, while in the subsequent section we briefly describe the necessary changes to deal with  $\omega$ -languages.

## 8.1 Syntax and Formal Semantics

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

$$\begin{aligned} \varphi ::= & a \mid \neg\varphi \mid \varphi \vee \varphi \mid \circ^t \varphi \mid \ominus^t \varphi \mid \chi_F^t \varphi \mid \chi_P^t \varphi \mid \varphi \mathcal{U}_\chi^t \varphi \mid \varphi \mathcal{S}_\chi^t \varphi \\ & \mid \circ_H^t \varphi \mid \ominus_H^t \varphi \mid \varphi \mathcal{U}_H^t \varphi \mid \varphi \mathcal{S}_H^t \varphi \end{aligned}$$

The truth of POTL formulas is defined with respect to a single word position. Let  $w$  be an OP word, and  $a \in AP$ . Then, for any position  $i \in U$  of  $w$ , we have  $(w, i) \models a$  iff  $i \in P(a)$ . Operators such as  $\wedge$  and  $\neg$  have the usual semantics from propositional logic. Next, while giving the formal semantics of POTL operators, we illustrate it by

showing how it can be used to express properties on program execution traces, such as the one of Figure 8.1.

**Next/back operators** The *downward* next and back operators  $\circ^d$  and  $\ominus^d$  are like their LTL counterparts, except they are true only if the next (resp. current) position is at a lower or equal ST level than the current (resp. preceding) one. The *upward* next and back,  $\circ^u$  and  $\ominus^u$ , are symmetric. Formally,

- $(w, i) \models \circ^d \varphi$  iff  $(w, i+1) \models \varphi$  and  $i < (i+1)$  or  $i \doteq (i+1)$ , and
- $(w, i) \models \ominus^d \varphi$  iff  $(w, i-1) \models \varphi$ , and  $(i-1) < i$  or  $(i-1) \doteq i$ .

Substitute  $<$  with  $>$  to obtain the semantics for  $\circ^u$  and  $\ominus^u$ .

E.g., we can write  $\circ^d \text{call}$  to say that the next position is an inner call (it holds in positions 2, 3, 4 of Figure 8.1),  $\ominus^d \text{call}$  to say that the previous position is a **call**, and the current is the first of the body of a function (positions 2,4, 5), or the **ret** of an empty one (positions 8, 10), and  $\ominus^u \text{call}$  to say that the current position terminates an empty function frame (holds in 6, 8, 10). In position 2 formula  $\circ^d p_B$  holds, but  $\circ^u p_B$  does not.

**Chain Next/Back** The *chain* next and back operators  $\chi_F^t$  and  $\chi_P^t$  evaluate their argument respectively on future and past positions in the chain relation with the current one. The *downward* (resp. *upward*) variant only considers chains whose right context goes down (resp. up) or remains at the same level in the ST. Formally,

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

Replace  $<$  with  $>$  for the upward versions.

E.g., in position 1 of Figure 8.1,  $\chi_F^d p_{Err}$  holds because  $\chi(1, 7)$  and  $\chi(1, 9)$ , meaning that  $p_A$  calls  $p_{Err}$  at least once. Also,  $\chi_F^u \text{exc}$  is true in **call** positions whose procedure is terminated by an exception thrown by an inner procedure (e.g., positions 3 and 4).  $\chi_P^u \text{call}$  is true in **exc** statements that terminate at least one procedure other than the one raising it, such as the one in position 6.  $\chi_F^d \text{ret}$  and  $\chi_F^u \text{ret}$  hold in **calls** to non-empty procedures that terminate normally, and not due to an uncaught exception (e.g., position 1).

**(Summary) Until/Since operators** POTL has two kinds of until and since operators. They express properties on paths, which are sequences of positions obtained by iterating the different kinds of next or back operators.

**Definition 8.1** (Paths, until and since). A *path* of length  $n \in \mathbb{N}$  between  $i, j \in U$  is a sequence of positions  $i = i_1 < i_2 < \dots < i_n = j$ .<sup>1</sup>

The *until* operator on a set of paths  $\Gamma$  is defined as follows: for any word  $w$  and position  $i \in U$ , and for any two POTL formulas  $\varphi$  and  $\psi$ ,  $(w, i) \models \varphi \mathcal{U}(\Gamma) \psi$  iff there exist a position  $j \in U$ ,  $j \geq i$ , and a path  $i_1 < i_2 < \dots < i_n$  between  $i$  and  $j$  in  $\Gamma$  such that  $(w, i_k) \models \varphi$  for any  $1 \leq k < n$ , and  $(w, i_n) \models \psi$ .

*Since* operators are defined symmetrically.

<sup>1</sup>Note that this definition is slightly different from Definition 6.5, because in this case paths *must* start in  $i$  and end in  $j$ .

Depending on  $\Gamma$ , a path from  $i$  to  $j$  may not exist. We define until/since operators by associating them with different sets of paths.

The *summary* until  $\psi \mathcal{U}_\chi^t \theta$  (resp. since  $\psi \mathcal{S}_\chi^t \theta$ ) operator is obtained by inductively applying the  $\circ^t$  and  $\chi_F^t$  (resp.  $\ominus^t$  and  $\chi_P^t$ ) operators. It holds in a position in which either  $\theta$  holds, or  $\psi$  holds together with  $\circ^t(\psi \mathcal{U}_\chi^t \theta)$  (resp.  $\ominus^t(\psi \mathcal{S}_\chi^t \theta)$ ) or  $\chi_F^t(\psi \mathcal{U}_\chi^t \theta)$  (resp.  $\chi_P^t(\psi \mathcal{S}_\chi^t \theta)$ ). It is an until operator on paths that can move not only between consecutive positions, but also between contexts of a chain, skipping its body. With the OPM of Figure 4.2, this means skipping function bodies. The downward variants can move between positions at the same level in the ST (i.e., in the same simple chain body), or down in the nested chain structure. The upward ones remain at the same level, or move to higher levels of the ST.

Formula  $\top \mathcal{U}_\chi^u \text{exc}$  is true in positions contained in the frame of a function that is terminated by an exception. It is true in pos. 3 of Figure 8.1 because of path 3-6, and false in pos. 1, because no upward path can enter the chain whose contexts are pos. 1 and 11. Formula  $\top \mathcal{U}_\chi^d \text{exc}$  is true in call positions whose function frame contains excs, but that are not directly terminated by one of them, such as the one in pos. 1 (with path 1-2-6).

We formally define *Downward Summary Paths (DSPs)* as follows. Given an OP word  $w$ , and two positions  $i \leq j$  in  $w$ , the DSP between  $i$  and  $j$ , if it exists, is a sequence of positions  $i = i_1 < i_2 < \dots < i_n = j$  such that, for each  $1 \leq p < n$ ,

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

The Downward Summary (DS) until and since operators  $\mathcal{U}_\chi^d$  and  $\mathcal{S}_\chi^d$  use as  $\Gamma$  the set of DSPs starting in the position in which they are evaluated. The definition for the upward counterparts is, again, obtained by substituting  $\triangleright$  for  $\triangleleft$ . In Figure 8.1,  $\text{call} \mathcal{U}_\chi^d$  ( $\text{ret} \wedge \text{p}_{Err}$ ) holds in pos. 1 because of path 1-7-8 and 1-9-10, ( $\text{call} \vee \text{exc}$ )  $\mathcal{S}_\chi^u \text{p}_B$  in pos. 7 because of path 3-6-7, and ( $\text{call} \vee \text{exc}$ )  $\mathcal{U}_\chi^u \text{ret}$  in 3 because of path 3-6-7-8.

**Hierarchical operators** A single position may be the left or right context of multiple chains. The operators seen so far cannot keep this fact into account, since they “forget” about a left context when they jump to the right one. Thus, we introduce the *hierarchical* next and back operators. The *upward* hierarchical next (resp. back),  $\circ_H^u \psi$  (resp.  $\ominus_H^u \psi$ ), is true iff the current position  $j$  is the right context of a chain whose left context is  $i$ , and  $\psi$  holds in the next (resp. previous) position  $j'$  that is a right context of  $i$ , with  $i < j, j'$ . So,  $\circ_H^u \text{p}_{Err}$  holds in position 7 of Figure 8.1 because  $\text{p}_{Err}$  holds in 9, and  $\ominus_H^u \text{p}_{Err}$  in 9 because  $\text{p}_{Err}$  holds in 7. In the ST,  $\circ_H^u$  goes *up* between *calls* to  $\text{p}_{Err}$ , while  $\ominus_H^u$  goes *down*. Their *downward* counterparts behave symmetrically, and consider multiple inner chains sharing their right context. They are formally defined as:

- $(w, i) \models \circ_H^u \varphi$  iff there exist a position  $h < i$  s.t.  $\chi(h, i)$  and  $h \triangleleft i$  and a position  $j = \min\{k \mid i < k \wedge \chi(h, k) \wedge h \triangleleft k\}$  and  $(w, j) \models \varphi$ ;
- $(w, i) \models \ominus_H^u \varphi$  iff there exist a position  $h < i$  s.t.  $\chi(h, i)$  and  $h \triangleleft i$  and a position  $j = \max\{k \mid k < i \wedge \chi(h, k) \wedge h \triangleleft k\}$  and  $(w, j) \models \varphi$ ;
- $(w, i) \models \circ_H^d \varphi$  iff there exist a position  $h > i$  s.t.  $\chi(i, h)$  and  $i \triangleright h$  and a position  $j = \min\{k \mid i < k \wedge \chi(k, h) \wedge k \triangleright h\}$  and  $(w, j) \models \varphi$ ;

- $(w, i) \models \ominus_H^d \varphi$  iff there exist a position  $h > i$  s.t.  $\chi(i, h)$  and  $i \succ h$  and a position  $j = \max\{k \mid k < i \wedge \chi(k, h) \wedge k \succ h\}$  and  $(w, j) \models \varphi$ .

In the ST of Figure 4.4,  $\circ_H^d$  and  $\ominus_H^d$  go *down* and up among **calls** terminated by the same **exc.** For example, in pos. 3  $\circ_H^d p_C$  holds, because both pos. 3 and 4 are in the chain relation with 6. Similarly, in pos. 4  $\ominus_H^d p_B$  holds. Note that these operators do not consider leftmost/rightmost contexts, so  $\circ_H^u \text{ret}$  is false in pos. 9, as  $\text{call} \doteq \text{ret}$ , and pos. 11 is the rightmost context of pos. 1.

The hierarchical until and since operators are defined by iterating these next and back operators.

**Definition 8.2** (Upward Hierarchical Path (UHP)). The *upward hierarchical path* between  $i$  and  $j$  is a sequence of positions  $i = i_1 < i_2 < \dots < i_n = j$  such that there exists a position  $h < i$  such that for each  $1 \leq p \leq n$  we have  $\chi(h, i_p)$  and  $h \prec i_p$ , and for each  $1 \leq q < n$  there exists no position  $k$  such that  $i_q < k < i_{q+1}$  and  $\chi(h, k)$ .

The until and since operators based on the set of UHPs starting in the position in which they are evaluated are denoted as  $\mathcal{U}_H^u$  and  $\mathcal{S}_H^u$ . E.g.,  $\text{call } \mathcal{U}_H^u p_{Err}$  holds in position 7 because of the singleton path 7 and path 7-9, and  $\text{call } \mathcal{S}_H^u p_{Err}$  in position 9 because of paths 9 and 7-9.

**Definition 8.3** (Downward Hierarchical Path (DHP)). The *downward hierarchical path* between  $i$  and  $j$  is a sequence of positions  $i = i_1 < i_2 < \dots < i_n = j$  such that there exists a position  $h > j$  such that for each  $1 \leq p \leq n$  we have  $\chi(i_p, h)$  and  $i_p \succ h$ , and for each  $1 \leq q < n$  there exists no position  $k$  such that  $i_q < k < i_{q+1}$  and  $\chi(k, h)$ .

The until and since operators based on the set of DHPs starting in the position in which they are evaluated are denoted as  $\mathcal{U}_H^d$  and  $\mathcal{S}_H^d$ . In Figure 8.1,  $\text{call } \mathcal{U}_H^d p_C$  holds in position 3, and  $\text{call } \mathcal{S}_H^d p_B$  in position 4, both because of path 3-4.

## 8.1.1 Equivalences

### 8.1.1.1 Expansion Laws

The POTL until and since operators enjoy expansion laws similar to those of LTL:

$$\varphi \mathcal{U}_X^t \psi \equiv \psi \vee \left( \varphi \wedge \left( \overset{t}{\circ}(\varphi \mathcal{U}_X^t \psi) \vee \chi_F^t(\varphi \mathcal{U}_X^t \psi) \right) \right) \quad (8.1)$$

$$\varphi \mathcal{S}_X^t \psi \equiv \psi \vee \left( \varphi \wedge \left( \overset{t}{\ominus}(\varphi \mathcal{S}_X^t \psi) \vee \chi_P^t(\varphi \mathcal{S}_X^t \psi) \right) \right) \quad (8.2)$$

$$\varphi \mathcal{U}_H^u \psi \equiv (\psi \wedge \chi_P^d \top \wedge \neg \chi_P^u \top) \vee (\varphi \wedge \circ_H^u(\varphi \mathcal{U}_H^u \psi)) \quad (8.3)$$

$$\varphi \mathcal{S}_H^u \psi \equiv (\psi \wedge \chi_P^d \top \wedge \neg \chi_P^u \top) \vee (\varphi \wedge \ominus_H^u(\varphi \mathcal{S}_H^u \psi)) \quad (8.4)$$

$$\varphi \mathcal{U}_H^d \psi \equiv (\psi \wedge \chi_F^u \top \wedge \neg \chi_F^d \top) \vee (\varphi \wedge \circ_H^d(\varphi \mathcal{U}_H^d \psi)) \quad (8.5)$$

$$\varphi \mathcal{S}_H^d \psi \equiv (\psi \wedge \chi_F^u \top \wedge \neg \chi_F^d \top) \vee (\varphi \wedge \ominus_H^d(\varphi \mathcal{S}_H^d \psi)) \quad (8.6)$$

**Lemma 8.4.** Given a word  $w$  on an OP alphabet  $(\mathcal{P}(AP), M_{AP})$ , two POTL formulas  $\varphi$  and  $\psi$ , for any position  $i$  in  $w$  the following equivalence holds:

$$\varphi \mathcal{U}_X^d \psi \equiv \psi \vee \left( \varphi \wedge \left( \circ^d(\varphi \mathcal{U}_X^d \psi) \vee \chi_F^d(\varphi \mathcal{U}_X^d \psi) \right) \right).$$

*Proof.* **[Only if]** Suppose  $\varphi \mathcal{U}_\chi^d \psi$  holds in  $i$ . If  $\psi$  holds in  $i$ , the equivalence is trivially verified. Otherwise,  $\varphi \mathcal{U}_\chi^d \psi$  is verified by a DSP  $i = i_0 < i_1 < \dots < i_n = j$  with  $n \geq 1$ , s.t.  $(w, i_p) \models \varphi$  for  $0 \leq p < n$  and  $(w, i_n) \models \psi$ . Note that, by the definition of DSP, any suffix of that path is also a DSP ending in  $j$ . Consider position  $i_1$ :  $\varphi$  holds in it, and it can be either

- $i_1 = i + 1$ . Then either  $i \triangleleft (i + 1)$  or  $i \doteq (i + 1)$ , and path  $i_1 < i_2 < \dots < i_n = j$  is the DSP between  $i_1$  and  $j$ , and  $\varphi$  holds in all  $i_p$  with  $1 \leq p < n$ , and  $\psi$  in  $j$ . So,  $\varphi \mathcal{U}_\chi^d \psi$  holds in  $i_1$ , and  $\circ^d(\varphi \mathcal{U}_\chi^d \psi)$  holds in  $i$ .
- $i_1 > i + 1$ . Then,  $\chi(i, i_1)$ , and  $i \triangleleft i_1$  or  $i \doteq i_1$ . Since  $i_1 < i_2 < \dots < i_n = j$  is the DSP from  $i_1$  to  $j$ ,  $\varphi \mathcal{U}_\chi^d \psi$  holds in  $i_1$ , and so does  $\chi_F^d(\varphi \mathcal{U}_\chi^d \psi)$  in  $i$ .

**[If]** Suppose the right-hand side of the equivalence holds in  $i$ . The case  $(w, i) \models \psi$  is trivial, so suppose  $\psi$  does not hold in  $i$ . Then  $\varphi$  holds in  $i$ , and either:

- $\circ^d(\varphi \mathcal{U}_\chi^d \psi)$  holds in  $i$ . Then, we have  $i \triangleleft (i + 1)$  or  $i \doteq (i + 1)$ , and there is a DSP  $i + 1 = i_1 < i_2 < \dots < i_n = j$ , with  $\varphi$  holding in all  $i_p$  with  $1 \leq p < n$ , and  $\psi$  in  $i_n$ .
  - If  $i \doteq (i + 1)$ , it is not the left context of any chain, and  $i = i_0 < i_1 < i_2 < \dots < i_n$  is a DSP satisfying  $\varphi \mathcal{U}_\chi^d \psi$  in  $i$ .
  - Otherwise, let  $k = \min\{h \mid \chi(i, h)\}$ : we have  $k > j$ , because a DSP cannot cross right chain contexts. So, adding  $i$  to the DSP generates another DSP, because there is no position  $h$  s.t.  $\chi(i, h)$  with  $h \leq j$ , and the successor of  $i$  in the path can only be  $i_1 = i + 1$ .
- $\chi_F^d(\varphi \mathcal{U}_\chi^d \psi)$  holds in  $i$ . Then, there exists a position  $k$  s.t.  $\chi(i, k)$  and  $i \triangleleft k$  or  $i \doteq k$  and  $\varphi \mathcal{U}_\chi^d \psi$  holds in  $k$ , because of a DSP  $k = i_1 < i_2 < \dots < i_n = j$ . If  $k = \max\{h \mid h \leq j \wedge \chi(i, h) \wedge (i \triangleleft h \vee i \doteq h)\}$ , then  $i = i_0 < i_1 < i_2 < \dots < i_n$  is a DSP by definition, and since  $\varphi$  holds in  $i$ ,  $\varphi \mathcal{U}_\chi^d \psi$  is satisfied in it. Otherwise, let  $k' = \max\{h \mid h \leq j \wedge \chi(i, h) \wedge (i \triangleleft h \vee i \doteq h)\}$ . Since  $i_1 > i$  and chains cannot cross, there exists a value  $q$ ,  $1 < q \leq n$ , s.t.  $i_q = k'$ . Thus  $i_q < i_{q+1} < \dots < i_n = j$  is a DSP, so  $\varphi \mathcal{U}_\chi^d \psi$  holds in  $i_q$  too. The path  $i < i_q < \dots < i_n$  is a DSP, and  $\varphi \mathcal{U}_\chi^d \psi$  holds in  $i$ .  $\square$

The proofs for the summary since and upward summary until are symmetric.

**Lemma 8.5.** *Given a word  $w$  on an OP alphabet  $(\mathcal{P}(AP), M_{AP})$ , and two POTL formulas  $\varphi$  and  $\psi$ , for any position  $i$  in  $w$  the following equivalence holds:*

$$\varphi \mathcal{U}_H^u \psi \equiv (\psi \wedge \chi_P^d \top \wedge \neg \chi_P^u \top) \vee (\varphi \wedge \circ_H^u(\varphi \mathcal{U}_H^u \psi)).$$

*Proof.* **[Only if]** Suppose  $\varphi \mathcal{U}_H^u \psi$  holds in  $i$ . Then, there exists a path  $i = i_0 < i_1 < \dots < i_n$ ,  $n \geq 0$ , and a position  $h < i$  s.t.  $\chi(h, i_p)$  and  $h \triangleleft i_p$  for each  $0 \leq p \leq n$ ,  $\varphi$  holds in all  $i_q$  for  $0 \leq q < n$ , and  $\psi$  holds in  $i_n$ . If  $n = 0$ ,  $\psi$  holds in  $i = i_0$ , and so does  $\chi_P^d \top$ , but  $\chi_P^u \top$  does not. Otherwise, the path  $i_1 < \dots < i_n$  is also a UHP, so  $\varphi \mathcal{U}_H^u \psi$  is true in  $i_1$ . Therefore,  $\varphi$  holds in  $i$ , and so does  $\circ_H^u(\varphi \mathcal{U}_H^u \psi)$ .

**[If]** If  $\chi_P^d \top$  holds in  $i$  but  $\chi_P^u \top$  does not, then there exists a position  $h < i$  s.t.  $\chi(h, i)$  and  $h \triangleleft i$ . If  $\psi$  also holds in  $i$ , then  $\varphi \mathcal{U}_H^u \psi$  is trivially satisfied in  $i$  by the path made of only  $i$  itself. Otherwise, if  $\circ_H^u(\varphi \mathcal{U}_H^u \psi)$  holds in  $i$ , then there exist a position  $h < i$  s.t.  $\chi(h, i)$  and  $h \triangleleft i$ , and a position  $i_1$  which is the minimum one s.t.

$i_1 > i$ ,  $\chi(h, i_1)$  and  $h \triangleleft i_1$ . In  $i_1$ ,  $\varphi \mathcal{U}_H^u \psi$  holds, so it is the first position of a UHP  $i_1 < i_2 < \dots < i_n$ . Since  $\varphi$  also holds in  $i$ , the path  $i = i_0 < i_1 < \dots < i_n$  is also a UHP, satisfying  $\varphi \mathcal{U}_H^u \psi$  in  $i$ .  $\square$

The proofs for the other hierarchical operators are analogous.

### 8.1.1.2 Shortcuts

As in LTL, it is worth defining some derived operators. For  $t \in \{d, u\}$ , we define the downward/upward summary *eventually* as  $\diamond^t \varphi := \top \mathcal{U}_\chi^t \varphi$ , and the downward/upward summary *globally* as  $\square^t \varphi := \neg \diamond^t (\neg \varphi)$ .  $\diamond^u \varphi$  and  $\square^u \varphi$  respectively say that  $\varphi$  holds in one or all positions in the path from the current position to the root of the ST. Their downward counterparts are more interesting: they consider all positions in the current right-hand-side and its subtrees, starting from the current position.  $\diamond^d \varphi$  says that  $\varphi$  holds in at least one of such positions, and  $\square^d \varphi$  in all of them. E.g., if  $\square^d (\neg p_A)$  holds in a **call**, it means that  $p_A$  never holds in its whole function body, which is the subtree rooted next to the **call**.

We anticipate that preventing downward paths from crossing the boundaries of the current subtrees, and conversely imposing upward ones to exit them without entering any inner ones are the features of POTL that allow its overcoming OPTL.

## 8.2 POTL on $\omega$ -Words

Since applications in model checking usually require temporal logics on infinite words, we now extend POTL to  $\omega$ -words.

To define OP  $\omega$ -words, it suffices to replace the finite set of positions  $U$  with the set of natural numbers  $\mathbb{N}$  in the definition of OP words. Then, the formal semantics of all POTL operators remains the same as in Section 8.1. The only difference in the intuitive meaning of operators occurs in  $\omega$ -words with open chains. In fact, chain next operators ( $\chi_F^d$  and  $\chi_F^u$ ) do not hold on the left contexts of open chains, as the  $\chi$  relation is undefined on them. The same can be said for downward hierarchical operators, when evaluated on left contexts of open chains.

Also recall that property 4 of the  $\chi$  relation does not hold if a position  $i$  is the left context of an open chain. In this case, there may be positions  $j_1 < j_2 < \dots < j_n$  such that  $\chi(i, j_p)$  and  $i \triangleleft j_p$  for all  $1 \leq p \leq n$ , but no position  $k$  such that  $\chi(i, k)$  and  $i > k$  or  $i \dot{=} k$ .

## 8.3 Motivating Examples

POTL can express many useful requirements of procedural programs. To emphasize the potential practical applications in automatic verification, we supply a few examples of typical program properties expressed as POTL formulas.

Let  $\square \psi$  be the LTL *globally* operator, which can be expressed in POTL as in Section 8.4.1. POTL can express Hoare-style pre/post-conditions with formulas such as  $\square(\mathbf{call} \wedge \rho \implies \chi_F^d(\mathbf{ret} \wedge \theta))$ , where  $\rho$  is the pre-condition, and  $\theta$  is the post-condition.

Unlike NWTL, POTL can easily express properties related to exception handling and interrupt management. E.g., the shortcut

$$\mathit{CallThr}(\psi) := \bigcirc^u(\mathbf{exc} \wedge \psi) \vee \chi_F^u(\mathbf{exc} \wedge \psi),$$

evaluated in a **call**, states that the procedure currently started is terminated by an **exc** in which  $\psi$  holds. So,  $\Box(\mathbf{call} \wedge \rho \wedge \mathit{CallThr}(\top) \implies \mathit{CallThr}(\theta))$  means that if precondition  $\rho$  holds when a procedure is called, then postcondition  $\theta$  must hold if that procedure is terminated by an exception. In object oriented programming languages, if  $\rho \equiv \theta$  is a class invariant asserting that a class instance's state is valid, this formula expresses *weak (or basic) exception safety* [1], and *strong exception safety* if  $\rho$  and  $\theta$  express particular states of the class instance. The *no-throw guarantee* can be stated with  $\Box(\mathbf{call} \wedge p_A \implies \neg \mathit{CallThr}(\top))$ , meaning procedure  $p_A$  is never interrupted by an exception.

*Stack inspection* [76, 102], i.e. properties regarding the sequence of procedures active in the program's stack at a certain point of its execution, is an important class of requirements that can be expressed with shortcut  $\mathit{Scall}(\varphi, \psi) := (\mathbf{call} \implies \varphi) \mathcal{S}_\chi^d (\mathbf{call} \wedge \psi)$ , which subsumes the *call since* of CaRet, and works with exceptions too. E.g.,  $\Box((\mathbf{call} \wedge p_B \wedge \mathit{Scall}(\top, p_A)) \implies \mathit{CallThr}(\top))$  means that whenever  $p_B$  is executed and at least one instance of  $p_A$  is on the stack,  $p_B$  is terminated by an exception. The OPA of Figure 4.5 satisfies this formula, because  $p_B$  is always called by  $p_A$ , and  $p_C$  always throws. If the OPA was an  $\omega$ OPBA, it would not satisfy such formula because of computations where  $p_C$  does not terminate.

## 8.4 Comparison with other logics

In this section, we briefly compare POTL with some relevant temporal logics in the state-of-the-art, namely LTL, logics on nested words, and OPTL.

### 8.4.1 Linear Temporal Logic (LTL)

As we already saw with OPTL, the main limitation of LTL is that the algebraic structure it is defined on only contains a linear order on word positions. Thus, it fails to model systems that require an additional binary relation, such as the  $\chi$  relation of POTL. LTL is in fact expressively equivalent to the first-order fragment of regular languages, and it cannot represent context-free languages, as POTL does.

On the other hand, POTL can express all LTL operators, so that POTL is strictly more expressive than LTL. Any LTL next formula  $\circ \varphi$  is in fact equivalent to the POTL formula  $\circ^d \varphi' \vee \circ^u \varphi'$ , where  $\varphi'$  is the translation of  $\varphi$  into POTL, and the LTL back can be translated symmetrically.

The *globally* operator can be translated as  $\Box \psi := \neg \diamond^u (\diamond^d \neg \psi)$ . This formula contains an upward summary eventually followed by a downward one, and it can be explained by thinking to a word's ST. The upward eventually evaluates its argument on all positions from the current one to the root. Its argument considers paths from each one of such positions to all the leaves of the subtrees rooted at their right, in the same rhs. Together, the two eventually operators consider paths from the current position to all subsequent positions in the word. Thus, with the initial negation this formula means that  $\neg \psi$  never holds in such positions, which is the meaning of the LTL globally operator.

The translation for LTL until and since is much more involved. We need to define some shortcuts, that will be used again in Section 9.1.2.2. For any  $a \subseteq AP$ ,  $\sigma_a := \bigwedge_{p \in a} p \wedge \bigwedge_{q \notin a} \neg q$  holds in a position  $i$  iff  $a$  is the set of atomic propositions holding in  $i$ . For any POTL formula  $\gamma$ , let  $\chi_F^{\leq} \gamma := \bigvee_{a, b \subseteq AP, a < b} (\sigma_a \wedge \chi_F^d (\sigma_b \wedge \gamma))$  be the restriction of  $\chi_F^d \gamma$  to chains with contexts in the  $\leq$  PR;  $\chi_P^{\geq} \gamma$  is analogous.

The translation for  $\varphi \mathcal{U} \psi$  follows, that for LTL since being symmetric:

$$\psi' \vee (\varphi' \wedge \alpha(\varphi')) \mathcal{U}_X^u (\psi' \vee (\varphi' \wedge \beta(\varphi'))) \mathcal{U}_X^d (\psi' \wedge \beta(\varphi'))$$

where  $\varphi'$  and  $\psi'$  are the translations of  $\varphi$  and  $\psi$  into POTL, and

$$\begin{aligned} \alpha(\varphi') &:= \chi_F^u \top \implies \neg(\bigcirc^d(\top \mathcal{U}_X^d \neg\varphi') \vee \chi_F^{\leq}(\top \mathcal{U}_X^d \neg\varphi')) \\ \beta(\varphi') &:= \chi_P^d \top \implies \neg(\ominus^u(\top \mathcal{S}_X^u \neg\varphi') \vee \chi_P^{\geq}(\top \mathcal{S}_X^u \neg\varphi')) \end{aligned}$$

The main formula is the concatenation of a Upward Summary (US) until and a DS until, and it can be explained similarly to the translation for LTL globally. Let  $i$  be the word position in which the formula is evaluated, and  $j$  the last one of the linear path, in which  $\psi'$  holds. The outermost US until is witnessed by a path from  $i$  to a position  $i'$  which, in the ST, is part of the rhs which is the closest common ancestor of  $i$  and  $j$ . In all positions  $i < k < i'$  in this path, formula  $\alpha(\varphi)$  holds. It means  $\varphi'$  holds in all positions contained in the subtree rooted at the non-terminal to the immediate right of  $k$  (i.e., in the body of the chain whose left context is  $k$ ).

Then, the DS until is witnessed by a downward path from  $i'$  to  $j$ . Here  $\beta(\varphi)$  has a role symmetric to  $\alpha(\varphi)$ . It forces  $\varphi'$  to hold in all subtrees rooted at the non-terminal to the left of positions in the DS path.

Thus, formulas  $\alpha(\varphi)$  and  $\beta(\varphi)$  make sure  $\varphi'$  holds in all chain bodies skipped by the summary paths, and  $\psi'$  holds in  $j$ .

### 8.4.2 Logics on Nested Words

The first temporal logics with explicit context-free aware modalities were based on nested words (cf. Section 3.2.4).

CaRet was the first temporal logic on nested words to be introduced, and it focuses on expressing properties on procedural programs, which explains its choice of modalities. One of its main limitations is that no CaRet operator allows pure downward movement in the ST, which is needed to express properties limited to a single subtree. While the LTL until can go downward, it can also go beyond the rightmost leaf of a subtree, thus effectively jumping upwards. This seems to be the main expressive limitation of CaRet, which is shared with OPTL, but not by POTL.

Such limitations were overcome with NWTL, which is FO-complete. Later on, in Corollary 9.11, we will show that  $\text{CaRet [8]} \subseteq \text{NWTL [12]} \subset \text{POTL}$ .

### 8.4.3 Logics on OPLs

The only other logic based on OPLs is OPTL, whose limitations were highlighted in Section 7.2. Here we show that POTL does not share such limitations.

As we saw in Section 7.2, the user can control whether OP-summary paths go up or down in the ST only partially. On one hand, OPTL's  $\varphi \mathcal{U}^{\geq} \psi$  is equivalent to POTL's  $\varphi \mathcal{U}_X^u \psi$ , and  $\varphi \mathcal{S}^{\leq} \psi$  to  $\varphi \mathcal{S}_X^d \psi$ : both operators only go upwards in the ST. On the other hand, however, there is no OPTL operator equivalent to POTL's  $\mathcal{U}_X^d$  or  $\mathcal{S}_X^u$ , which go downward.

The FOL formula  $\alpha$  that we introduced in that section to show this limitation is easily expressible in POTL as follows:

$$\Box(\text{exc} \implies \chi_P^d(\mathbf{han} \wedge \diamond^d p_A)).$$

Moreover, formula  $\alpha'(x)$  from the same section can be expressed as  $\chi_F^d \top \wedge \diamond^d p_A$  on OPM  $M_{\text{call}}$ , and as  $\chi_F^d \top \wedge (\odot^d \diamond^d p_A \vee \top \mathcal{U}_H^u (\diamond^d p_A))$  in any other OPM.<sup>2</sup>

OPTL's hierarchical operators have some limitations, too. The paths they are based on do not start in the position where until and since operators are evaluated, but always in a future position. Thus, it is not possible to concatenate them to express complex properties on right (resp. left) contexts of chains sharing their left (resp. right) context, such as several function calls issued by the same function, or multiple function calls terminated by the same exception. POTL has both hierarchical next/back and until/since pairs which are composable, making it expressively complete on such positions. For example, POTL formula

$$\gamma := \Box(\text{call} \wedge p_A \implies (\top \mathcal{S}_H^u (\text{call} \wedge p_B)) \mathcal{S}_X^d (\ominus^d \# \vee \chi_P^d \#))$$

means that whenever procedure  $p_A$  is called, all procedures in the stack have previously invoked  $p_B$  (possibly excluding the one directly calling  $p_A$ ). While  $\mathcal{S}_X^d$  can be replaced with OPTL's  $\mathcal{S}^{\leq \dot{=}}$ , POTL's  $\mathcal{S}_H^u$  cannot be easily translated. In fact, OPTL's hierarchical operators would only allow us to state that  $p_B$  is invoked by the procedures in the stack, but not necessarily before the call to  $p_A$ .

---

<sup>2</sup>The reason for this difference is that with  $M_{\text{call}}$  all right-hand-sides have at most two terminals, and none of them can go on with another terminal after the right context of a chain. On a different OPM,  $\diamond^d$  could go past such a right context, and we have to use a more complex formula which is explicitly limited to a single subtree.

In [58], the proof of Theorem 7.9 is made directly with formula  $\diamond^d p_A$ , which is however harder to express in FOL. For this reason, here we proved that theorem with  $\alpha'(x)$ .

## Chapter 9

# POTL Expressive Completeness

In this chapter, we prove that POTL is equivalent to FOL on OP words. The proof for finite OP words, reported in Section 9.1, is essentially based on syntactic translations, while its extension to OP  $\omega$ -words, in Section 9.2, employs more sophisticated model-theoretic techniques.

### 9.1 First-Order Completeness on Finite Words

To show that  $\text{POTL} \subseteq \text{FOL}$  on finite OP words, we give a direct translation of POTL into FOL. Proving that  $\text{FOL} \subseteq \text{POTL}$  is more involved: we translate  $\mathcal{X}_{\text{until}}$  [125], a logic on trees, into POTL.  $\mathcal{X}_{\text{until}}$  (defined in Section 9.1.2.2) is a logic on trees introduced to prove the expressive completeness of Conditional XPath, and from its being equivalent to FOL on trees [118, 126] we derive a FO-completeness result for POTL.

#### 9.1.1 First-Order Semantics of POTL

We show that POTL can be expressed with FOL equipped with monadic relations for atomic propositions, a total order on positions, and the chain relation between pairs of positions. We define below the translation function  $\nu$ , such that for any POTL formula  $\varphi$ , word  $w$  and position  $i$ ,  $(w, i) \models \nu_\varphi(x)$  iff  $(w, i) \models \varphi$ . The translation for propositional operators is trivial.

For temporal operators, we first need to define a few auxiliary formulas. We define the successor relation as the FO formula

$$\text{succ}(x, y) := x < y \wedge \neg \exists z (x < z \wedge z < y).$$

The PRs between positions can be expressed by means of propositional combinations of monadic atomic relations only. Given a set of atomic propositions  $a \subseteq AP$ , we define formula  $\sigma_a(x)$ , stating that all and only propositions in  $a$  hold in position  $x$ , as follows:

$$\sigma_a(x) := \bigwedge_{p \in a} p(x) \wedge \bigwedge_{p \in AP \setminus a} \neg p(x) \quad (9.1)$$

For any pair of FO variables  $x, y$  and  $\pi \in \{\leq, \dot{=}, \succ\}$ , we can build formula

$$x \pi y := \bigvee_{a, b \subseteq AP \mid a\pi b} (\sigma_a(x) \wedge \sigma_b(y)).$$

The following translations employ the three FO variables  $x, y, z$ , only. This, in addition to the FO-completeness result for POTL, proves that FOL on OP words retains the three-variable property, which holds in regular words.

### Next and Back Operators

$$\nu_{\circ^d} \varphi(x) := \exists y \left( \text{succ}(x, y) \wedge (x \leq y \vee x \dot{=} y) \wedge \exists x (x = y \wedge \nu_\varphi(x)) \right)$$

$\nu_{\circ^d} \varphi(x)$  is defined similarly, and  $\nu_{\circ^u} \varphi(x)$  and  $\nu_{\ominus^u} \varphi(x)$  by replacing  $\leq$  with  $\succ$ .

$$\nu_{\chi_F^d} \varphi(x) := \exists y \left( \chi(x, y) \wedge (x \leq y \vee x \dot{=} y) \wedge \exists x (x = y \wedge \nu_\varphi(x)) \right)$$

$\nu_{\chi_P^d} \varphi(x)$ ,  $\nu_{\chi_F^u} \varphi(x)$  and  $\nu_{\chi_P^u} \varphi(x)$  are defined similarly.

**Downward/Upward Summary Until/Since** The translation for the DS until operator can be obtained by noting that, given two positions  $x$  and  $y$ , the DSP between them, if it exists, is the one that skips all chain bodies entirely contained between them, among those with contexts in the  $\leq$  or  $\dot{=}$  relations. A position  $z$  being part of such a path can be expressed with formula  $\neg\gamma(x, y, z)$  as follows:

$$\begin{aligned} \gamma(x, y, z) &:= \gamma_L(x, z) \wedge \gamma_R(y, z) \\ \gamma_L(x, z) &:= \exists y \left( x \leq y \wedge y < z \wedge \exists x (z < x \wedge \chi(y, x) \wedge (y \leq x \vee y \dot{=} x)) \right) \\ \gamma_R(y, z) &:= \exists x \left( z < x \wedge x \leq y \wedge \exists y (y < z \wedge \chi(y, x) \wedge (y \leq x \vee y \dot{=} x)) \right) \end{aligned}$$

$\gamma(x, y, z)$  is true iff  $z$  is not part of the DSP between  $x$  and  $y$ , while  $x \leq z \leq y$ . In particular,  $\gamma_L(x, z)$  asserts that  $z$  is part of the body of a chain whose left context is after  $x$ , and  $\gamma_R(y, z)$  states that  $z$  is part of the body of a chain whose right context is before  $y$ . Since chain bodies cannot cross, either the two chain bodies are actually the same one, or one of them is a sub-chain nested into the other. In both cases,  $z$  is part of a chain body entirely contained between  $x$  and  $y$ , and is thus not part of the path.

Moreover, for such a path to exist, each one of its positions must be in one of the admitted PRs with the next one. Formula

$$\delta(y, z) := \exists x (z < x \wedge x \leq y \wedge (z \leq x \vee z \dot{=} x) \wedge \neg\gamma(z, y, x) \wedge (\text{succ}(z, x) \vee \chi(z, x)))$$

asserts this for position  $z$ , with the path ending in  $y$ . (Note that by exchanging  $x$  and  $z$  in the definition of  $\gamma(x, y, z)$  above, one can obtain  $\gamma(z, y, x)$  without using any additional variable.) Finally,  $\varphi \mathcal{U}_\chi^d \psi$  can be translated as follows:

$$\begin{aligned} \nu_{\varphi \mathcal{U}_\chi^d \psi}(x) &:= \exists y \left( x \leq y \wedge \exists x (x = y \wedge \nu_\psi(x)) \right. \\ &\quad \left. \wedge \forall z (x \leq z \wedge z < y \wedge \neg\gamma(x, y, z)) \right. \\ &\quad \left. \implies \exists x (x = z \wedge \nu_\varphi(x)) \wedge \delta(y, z) \right) \end{aligned}$$

The translation for the DS since operator is similar:

$$\begin{aligned} \nu_{\varphi S_x^d \psi}(x) &:= \exists y \left( y \leq x \wedge \exists x (x = y \wedge \nu_{\psi}(x)) \right. \\ &\quad \wedge \forall z (y < z \wedge z \leq x \wedge \neg \gamma(y, x, z) \\ &\quad \left. \implies \exists x (x = z \wedge \nu_{\varphi}(x)) \wedge \delta(x, z) \right) \end{aligned}$$

$\nu_{\varphi \mathcal{U}_x^u \psi}(x)$  and  $\nu_{\varphi S_x^u \psi}(x)$  are defined as above, substituting  $\succ$  for  $\leq$ .

**Hierarchical Operators** Finally, below are the translations for two hierarchical operators, the others being symmetric.

$$\begin{aligned} \nu_{\circlearrowleft_H^u \varphi}(x) &:= \exists y \left( y < x \wedge \chi(y, x) \wedge y \leq x \wedge \right. \\ &\quad \exists z \left( x < z \wedge \chi(y, z) \wedge y \leq z \wedge \exists x (x = z \wedge \nu_{\varphi}(x)) \right. \\ &\quad \left. \left. \wedge \forall y (x < y \wedge y < z \implies \forall z (\chi(z, x) \wedge z \leq x \implies \neg \chi(z, y))) \right) \right) \end{aligned}$$

$$\begin{aligned} \nu_{\varphi \mathcal{U}_H^u \psi}(x) &:= \exists z \left( z < x \wedge z \leq x \wedge \chi(z, x) \wedge \right. \\ &\quad \exists y \left( x \leq y \wedge \chi(z, y) \wedge z \leq y \wedge \exists x (x = y \wedge \nu_{\psi}(x)) \wedge \right. \\ &\quad \forall z (x \leq z \wedge z < y \wedge \exists y (y < x \wedge y \leq x \wedge \chi(y, x) \wedge \chi(y, z)) \\ &\quad \left. \left. \implies \exists x (x = z \wedge \nu_{\varphi}(x)) \right) \right) \end{aligned}$$

### 9.1.2 Expressing $\mathcal{X}_{until}$ in POTL

To translate  $\mathcal{X}_{until}$  to POTL, we give an isomorphism between OP words and (a subset of) Unranked Ordered Tree (UOT), the structures on which  $\mathcal{X}_{until}$  is defined. First, we show how to translate OP words into UOTs, and then the reverse.

#### 9.1.2.1 OPM-compatible Unranked Ordered Trees

**Definition 9.1** (Unranked Ordered Trees). A UOT is a tuple  $T = \langle S, R_{\downarrow}, R_{\Rightarrow}, L \rangle$ . Each node is a sequence of child numbers, representing the path from the root to it.  $S$  is a finite set of finite sequences of natural numbers closed under the prefix operation, and for any sequence  $s \in S$ , if  $s \cdot k \in S$ ,  $k \in \mathbb{N}$ , then either  $k = 0$  or  $s \cdot (k-1) \in S$  (by  $\cdot$  we denote concatenation).  $R_{\downarrow}$  and  $R_{\Rightarrow}$  are two binary relations called the *descendant* and *following sibling* relation, respectively. For  $s, t \in S$ ,  $s R_{\downarrow} t$  iff  $t$  is any child of  $s$  ( $t = s \cdot k$ ,  $k \in \mathbb{N}$ , i.e.  $t$  is the  $k$ -th child of  $s$ ), and  $s R_{\Rightarrow} t$  iff  $t$  is the immediate sibling to the right of  $s$  ( $s = r \cdot h$  and  $t = r \cdot (h+1)$ ), for  $r \in S$  and  $h \in \mathbb{N}$ ).  $L: AP \rightarrow \mathcal{P}(S)$  is a function that maps each atomic proposition to the set of nodes labeled with it. We denote as  $\mathcal{T}$  the set of all UOTs.

Given an OP word  $w = \langle U, <, M_{\mathcal{P}(AP)}, P \rangle$ , it is possible to build a UOT  $T_w = \langle S_w, R_{\downarrow}, R_{\Rightarrow}, L_w \rangle \in \mathcal{T}$  with labels in  $\mathcal{P}(AP)$  isomorphic to  $w$ . To do so, we define



**Definition 9.2** (OPM-compatible UOTs). We denote the set of UOTs compatible with an OPM  $M$  as  $\mathcal{T}_M$ . A UOT  $T$  is in  $\mathcal{T}_M$  iff the following properties hold. The root node and its rightmost child are the only ones labeled with  $\#$ . For any node  $s \in T$ , its rightmost child  $r$ , if any, is such that either  $s \prec r$  or  $s \doteq r$ . For any other child  $s' \neq r$  of  $s$ , we have  $s \prec s'$ . If  $\text{Rc}(s)$  exists, then  $s \succ \text{Rc}(s)$ .

Given a tree  $T \in \mathcal{T}_M$  with labels on  $\mathcal{P}(AP)$ , it is possible to build an OP word  $w_T$  isomorphic to  $T$ . Indeed,

**Lemma 9.3.** *Given an OP word  $w$  and the UOT  $T_w = \tau(w)$ , function  $\tau$  is an isomorphism between positions of  $w$  and nodes of  $T_w$ .*

*Proof.* We define function  $\tau_{AP}^{-1} : S \rightarrow \mathcal{P}(AP)^+$ , which maps a UOT node to the subword corresponding to the subtree rooted in it. For any node  $s \in T$ , let its label  $a = \{p \mid s \in L(p)\}$ , and let  $c_0, c_1 \dots c_n$  be its children, if any.  $\tau_{AP}^{-1}(s)$  is defined as  $\tau_{AP}^{-1}(s) = a$  if  $s$  has no children, and  $\tau_{AP}^{-1}(s) = a \cdot \tau_{AP}^{-1}(c_0) \cdot \tau_{AP}^{-1}(c_1) \cdots \tau_{AP}^{-1}(c_n)$  otherwise. We prove  $\tau_{AP}^{-1}(s)$  is an OP word.

We need to prove by induction on the tree structure that for any tree node  $s$ ,  $\tau_{AP}^{-1}(s)$  is of the form  $a_0 x_0 a_1 x_1 \dots a_n x_n$ , with  $n \geq 0$ , and such that for  $0 \leq k < n$ ,  $a_k \doteq a_{k+1}$  and either  $x_k = \varepsilon$  or  $a^k [x_k]^{a_{k+1}}$ . In the following, we denote as  $\text{first}(x)$  the first position of a string  $x$ , and as  $\text{last}(x)$  the last one. Indeed, for each  $0 \leq i < n$  we have  $a \prec \text{first}(\tau_{AP}^{-1}(c_i))$ , and the rightmost leaf  $f_i$  of the tree rooted in  $c_i$  is such that  $\text{Rc}(f_i) = c_{i+1}$ . Since  $f_i = \tau(\text{last}(\tau_{AP}^{-1}(c_i)))$  and  $c_{i+1} = \tau(\text{first}(\tau_{AP}^{-1}(c_{i+1})))$ , we have  $\text{last}(\tau_{AP}^{-1}(c_i)) \succ \text{first}(\tau_{AP}^{-1}(c_{i+1}))$ . So,  $a[\tau_{AP}^{-1}(c_i)]^{\text{first}(\tau_{AP}^{-1}(c_{i+1}))}$ . As for  $\tau_{AP}^{-1}(c_n)$ , if  $a \prec c_n$  then  $\tau_{AP}^{-1}(s) = a_0 x_0$  (and  $a_0 \prec \text{first}(x_0)$ ), with  $a_0 = a$  and  $x_0 = \tau_{AP}^{-1}(c_0) \cdot \tau_{AP}^{-1}(c_1) \cdots \tau_{AP}^{-1}(c_n)$ . If  $a \doteq c_n$ , consider that, by hypothesis,  $\tau_{AP}^{-1}(c_n)$  is of the form  $a_1 x_1 a_2 \dots a_n x_n$ . So  $\tau_{AP}^{-1}(s) = a_0 x_0 a_1 x_1 a_2 \dots a_n x_n$ , with  $a_0 = a$  and  $x_0 = \tau_{AP}^{-1}(c_0) \cdot \tau_{AP}^{-1}(c_1) \cdots \tau_{AP}^{-1}(c_{n-1})$ .

The root  $0$  of  $T$  and its rightmost child  $c_\#$  are labeled with  $\#$ . So,  $\tau_{AP}^{-1}(c_\#) = \#$ , and  $\tau_{AP}^{-1}(0) = \#x_0\#$ , with  $\#[x_0]\#$ , which is a finite OP word.

$\tau^{-1} : S \rightarrow U$  can be derived from  $\tau_{AP}^{-1}$ . By construction, we have  $\tau^{-1}(\tau(i)) = i$  for any word  $w$  and position  $i$ .  $\square$

From Lemma 9.3 follows:

**Proposition 9.4.** *Let  $M_{AP}$  be an OPM on  $\mathcal{P}(AP)$ . For any FO formula  $\varphi(x)$  on OP words compatible with  $M_{AP}$ , there exists a FO formula  $\varphi'(x)$  on trees in  $\mathcal{T}_{M_{AP}}$  such that for any OP word  $w$  and position  $i$  in it,  $w \models \varphi(i)$  iff  $T_w \models \varphi'(\tau(i))$ , with  $T_w = \tau(w)$ .*

### 9.1.2.2 POTL Translation of $\mathcal{X}_{\text{until}}$

We now give the full translation of the logic  $\mathcal{X}_{\text{until}}$  from [125] into POTL.

The syntax of  $\mathcal{X}_{\text{until}}$  formulas is  $\varphi ::= a \mid \top \mid \neg\varphi \mid \varphi \wedge \varphi \mid \rho(\varphi, \varphi)$ , with  $a \in AP$  and  $\rho \in \{\downarrow, \uparrow, \Rightarrow, \Leftarrow\}$ . The semantics of propositional operators is the usual one, while  $\rho(\varphi, \varphi)$  is an until/since operator on the child and sibling relations. Let  $T \in \mathcal{T}$  be a UOT with nodes in  $S$ . For any  $r, s \in S$ ,  $R_\uparrow$  and  $R_\Leftarrow$  are s.t.  $rR_\uparrow s$  iff  $sR_\downarrow r$ , and  $rR_\Leftarrow s$  iff  $sR_\Rightarrow r$ . We denote as  $R_\rho^+$  the transitive (but not reflexive) closure of relation  $R_\rho$ , and by  $R_\rho^*$  its transitive and reflexive closure. For  $s \in S$ ,  $(T, s) \models \rho(\varphi, \psi)$  iff there exists a node  $t \in S$  s.t.  $sR_\rho^+ t$  and  $(T, t) \models \psi$ , and for any  $r \in S$  s.t.  $sR_\rho^+ r$  and  $rR_\rho^+ t$  we have  $(T, r) \models \varphi$ . Notice that  $t \neq s$  and  $r \neq s$ , so  $s$  is not included in the paths: we call this semantics *strict*. Conversely, in POTL paths always start from the position where an until/since operator is evaluated.

[118] proved the equivalence of  $\mathcal{X}_{\text{until}}$  to the logic Conditional XPath, which was proved equivalent to FOL on finite UOTs in [126]. This result is valid for any labeling of tree nodes, and so is on OPM-compatible UOTs.

**Theorem 9.5** ([118, 126]). *Let  $M_{AP}$  be an OPM on AP. For any FO formula  $\varphi(x)$  on trees in  $\mathcal{T}_{M_{AP}}$ , there exists a  $\mathcal{X}_{\text{until}}$  formula  $\varphi'$  such that, for any  $T \in \mathcal{T}_{M_{AP}}$  and node  $t \in T$ , we have  $T \models \varphi(t)$  iff  $(T, t) \models \varphi'$ .*

We define function  $\iota_{\mathcal{X}}$ , which translates any  $\mathcal{X}_{\text{until}}$  formula  $\varphi$  into a POTL formula s.t.  $\varphi$  holds on a UOT  $T$  iff  $\iota_{\mathcal{X}}(\varphi)$  holds on the isomorphic word  $w_T$ .  $\iota_{\mathcal{X}}$  is defined as the identity for the propositional operators, and with the equivalences below for the other  $\mathcal{X}_{\text{until}}$  operators. Recall from Section 8.4.1 that for any  $a \subseteq AP$ ,  $\sigma_a := \bigwedge_{p \in a} p \wedge \bigwedge_{q \notin a} \neg q$  holds in a position  $i$  iff  $a$  is the set of atomic propositions holding in  $i$ . For any POTL formula  $\gamma$ , let  $\chi_F^{\leq} \gamma := \bigvee_{a, b \subseteq AP, a \leq b} (\sigma_a \wedge \chi_F^d(\sigma_b \wedge \gamma))$  be the restriction of  $\chi_F^d \gamma$  to chains with contexts in the  $\leq$  PR; operators  $\chi_{\bar{F}}^{\dot{\leq}} \gamma$ ,  $\chi_P^{\leq} \gamma$ ,  $\chi_{\bar{P}}^{\dot{\leq}} \gamma$ ,  $\circ^{\leq} \gamma$ ,  $\ominus^{\leq} \gamma$  are defined analogously.

For any  $\mathcal{X}_{\text{until}}$  formulas  $\varphi, \psi$ , let  $\varphi' = \iota_{\mathcal{X}}(\varphi)$  and  $\psi' = \iota_{\mathcal{X}}(\psi)$ . We define  $\iota_{\mathcal{X}}$  as follows:

$$\iota_{\mathcal{X}}(\Downarrow(\varphi, \psi)) := \circ^d(\varphi' \mathcal{U}_{\bar{X}}^d \psi') \vee \chi_F^d(\varphi' \mathcal{U}_{\bar{X}}^d \psi') \quad (9.2)$$

$$\iota_{\mathcal{X}}(\Uparrow(\varphi, \psi)) := \ominus^d(\varphi' \mathcal{S}_{\bar{X}}^d \psi') \vee \chi_P^d(\varphi' \mathcal{S}_{\bar{X}}^d \psi') \quad (9.3)$$

$$\iota_{\mathcal{X}}(\Rightarrow(\varphi, \psi)) := \circ_H^u(\varphi' \mathcal{U}_H^u \psi') \quad (9.4)$$

$$\vee (\neg \circ_H^u(\top \mathcal{U}_H^u \neg \varphi') \wedge \chi_{\bar{P}}^{\dot{\leq}}(\chi_{\bar{F}}^{\dot{\leq}} \psi')) \quad (9.5)$$

$$\vee \ominus^{\leq} \left( \chi_{\bar{F}}^{\leq}(\psi' \wedge \neg \ominus_H^u(\top \mathcal{S}_H^u \neg \varphi')) \right) \quad (9.6)$$

$$\vee \ominus^{\leq}(\chi_{\bar{F}}^{\dot{\leq}} \psi' \wedge \neg \chi_{\bar{F}}^{\leq} \neg \varphi') \quad (9.7)$$

$$\iota_{\mathcal{X}}(\Leftarrow(\varphi, \psi)) := \ominus_H^u(\varphi' \mathcal{S}_H^u \psi') \quad (9.8)$$

$$\vee \chi_{\bar{P}}^{\dot{\leq}} \left( \chi_{\bar{F}}^{\leq}(\neg \circ_H^u \top \wedge \varphi' \mathcal{S}_H^u \psi') \right) \quad (9.9)$$

$$\vee \left( \chi_{\bar{P}}^{\leq}(\circ^{\leq} \psi') \wedge \neg \ominus_H^u(\top \mathcal{S}_H^u \neg \varphi') \right) \quad (9.10)$$

$$\vee \chi_{\bar{P}}^{\dot{\leq}}(\circ^{\leq} \psi' \wedge \neg \chi_{\bar{F}}^{\leq} \neg \varphi') \quad (9.11)$$

We prove the correctness of this translation in the following lemmas.

**Lemma 9.6.** *Given an OP alphabet  $(AP, M_{AP})$ , for every  $\mathcal{X}_{\text{until}}$  formula  $\Downarrow(\varphi, \psi)$ , and for any OP word  $w$  and position  $i$  in  $w$ , we have*

$$(T_w, \tau(i)) \models \Downarrow(\varphi, \psi) \text{ iff } (w, i) \models \iota_{\mathcal{X}}(\Downarrow(\varphi, \psi)).$$

$T_w \in \mathcal{T}_{M_{AP}}$  is the UOT obtained by applying function  $\tau$  to every position in  $w$ , such that for any position  $i'$  in  $w$   $(T_w, \tau(i')) \models \varphi$  iff  $(w, i') \models \iota_{\mathcal{X}}(\varphi)$ , and likewise for  $\psi$ .

*Proof.* Let  $\varphi' = \iota_{\mathcal{X}}(\varphi)$  and  $\psi' = \iota_{\mathcal{X}}(\psi)$ .

**[Only if]** Suppose  $(T_w, \tau(i)) \models \Downarrow(\varphi, \psi)$ . Let  $r = \tau(i)$ , and  $s = \tau(j)$  s.t.  $r R_{\Downarrow} s$  and  $s$  is the first tree node of the path witnessing  $\Downarrow(\varphi, \psi)$ .

We inductively prove that  $\varphi' \mathcal{U}_{\bar{X}}^d \psi'$  holds in  $j$ . If  $s$  is the last node of the path, then  $\psi'$  holds in  $j$  and so does, trivially,  $\varphi' \mathcal{U}_{\bar{X}}^d \psi'$ . Otherwise, consider any node  $t = \tau(k)$

in the path, except the last one, and suppose  $\varphi' \mathcal{U}_\chi^d \psi'$  holds in  $k'$  s.t.  $t' = \tau(k')$  is the next node in the path. If  $t'$  is the leftmost child of  $t$ , then  $k' = k + 1$  and either  $k < k'$  or  $k \doteq k'$ : in both cases  $\circ^d(\varphi' \mathcal{U}_\chi^d \psi')$  holds in  $k$ . If  $t'$  is not the leftmost child, then  $\chi(k, k')$  and  $k < k'$  or  $k \doteq k'$ : so  $\chi_F^d(\varphi' \mathcal{U}_\chi^d \psi')$  holds in  $k$ . Thus, by expansion law  $\varphi' \mathcal{U}_\chi^d \psi' \equiv \psi' \vee (\varphi' \wedge (\circ^d(\varphi' \mathcal{U}_\chi^d \psi') \vee \chi_F^d(\varphi' \mathcal{U}_\chi^d \psi')))$ ,  $\varphi' \mathcal{U}_\chi^d \psi'$  holds in  $k$  and, by induction, also in  $j$ .

Suppose  $s$  is the leftmost child of  $r$ :  $j = i + 1$ , and either  $i < j$  or  $i \doteq j$ , so  $\circ^d(\varphi' \mathcal{U}_\chi^d \psi')$  holds in  $i$ . Otherwise,  $\chi(i, j)$  and either  $i < j$  or  $i \doteq j$ . In both cases,  $\chi_F^d(\varphi' \mathcal{U}_\chi^d \psi')$  holds in  $i$ .

**[If]** Suppose (9.2) holds in  $i$ . If  $\circ^d(\varphi' \mathcal{U}_\chi^d \psi')$  holds in  $i$ , then  $\varphi' \mathcal{U}_\chi^d \psi'$  holds in  $j = i + 1$ , and either  $i < j$  or  $i \doteq j$ : then  $s = \tau(j)$  is the leftmost child of  $\tau(i)$ . If  $\chi_F^d(\varphi' \mathcal{U}_\chi^d \psi')$  holds in  $i$ , then  $\varphi' \mathcal{U}_\chi^d \psi'$  holds in  $j$  s.t.  $\chi(i, j)$  and  $i < j$  or  $i \doteq j$ :  $s = \tau(j)$  is a child of  $\tau(i)$  in this case as well.

We prove that if  $\varphi' \mathcal{U}_\chi^d \psi'$  holds in a position  $j$  s.t.  $\tau(i) R_{\downarrow} \tau(j)$ , then  $\downarrow(\varphi, \psi)$  holds in  $\tau(i)$ . If  $\varphi' \mathcal{U}_\chi^d \psi'$  holds in  $j$ , then there exists a DSP of minimal length from  $j$  to  $h > j$  s.t.  $(w, h) \models \psi'$  and  $\varphi'$  holds in all positions  $j \leq k < h$  of the path, and  $(T_w, \tau(k)) \models \varphi$ . In any such  $k$ ,  $\varphi' \mathcal{U}_\chi^d \psi' \equiv \psi' \vee (\varphi' \wedge (\circ^d(\varphi' \mathcal{U}_\chi^d \psi') \vee \chi_F^d(\varphi' \mathcal{U}_\chi^d \psi')))$  holds. Since this DSP is the minimal one,  $\psi'$  does not hold in  $k$ . Either  $\circ^d(\varphi' \mathcal{U}_\chi^d \psi')$  or  $\chi_F^d(\varphi' \mathcal{U}_\chi^d \psi')$  hold in it. Therefore, the next position in the path is  $k'$  s.t. either  $k' = k + 1$  or  $\chi(k, k')$ , and either  $k < k'$  or  $k \doteq k'$ , and  $(w, k') \models \varphi' \mathcal{U}_\chi^d \psi'$ . Therefore,  $\tau(k')$  is a child of  $\tau(k)$ . So, there is a sequence of nodes  $s_0, s_1, \dots, s_n$  in  $T_w$  s.t.  $\tau(i) R_{\downarrow} s_0$ , and  $s_i R_{\downarrow} s_{i+1}$  and  $(T_w, s_i) \models \varphi$  for  $0 \leq i < n$ , and  $(T_w, s_n) \models \psi$ . This is a path making  $\downarrow(\varphi, \psi)$  true in  $\tau(i)$ .  $\square$

The proof for  $\iota_{\mathcal{X}}(\uparrow(\varphi, \psi))$  (9.3) is analogous to Lemma 9.6, and is therefore omitted.

**Lemma 9.7.** *Given an OP alphabet  $(AP, M_{AP})$ , for every  $\mathcal{X}_{\text{until}}$  formula  $\Rightarrow(\varphi, \psi)$ , and for any OP word  $w$  and position  $i$  in  $w$ , we have*

$$(T_w, \tau(i)) \models \Rightarrow(\varphi, \psi) \text{ iff } (w, i) \models \iota_{\mathcal{X}}(\Rightarrow(\varphi, \psi)).$$

$T_w \in \mathcal{T}_{M_{AP}}$  is the UOT obtained by applying function  $\tau$  to every position in  $w$ , such that for any position  $i'$  in  $w$   $(T_w, \tau(i')) \models \varphi$  iff  $(w, i') \models \iota_{\mathcal{X}}(\varphi)$ , and likewise for  $\psi$ .

*Proof.* Let  $\varphi' = \iota_{\mathcal{X}}(\varphi)$  and  $\psi' = \iota_{\mathcal{X}}(\psi)$ .

**[Only if]** Suppose  $\Rightarrow(\varphi, \psi)$  holds in  $s = \tau(i)$ . Then, node  $r = \tau(h)$  s.t.  $r R_{\downarrow} s$  has at least two children, and  $\Rightarrow(\varphi, \psi)$  is witnessed by a path starting in  $t = \tau(j)$  s.t.  $s R_{\Rightarrow} t$ , and ending in  $v = \tau(k)$ . We have the following cases:

1.  $s$  is not the leftmost child of  $r$ .

(a)  $h < k$ . By the construction of  $T_w$ , for any node  $t'$  in the path, there exists a position  $j' \in w$  s.t.  $t' = \tau(j')$ ,  $\chi(h, j')$  and  $h < j'$ . The path made by such positions is a UHP, and  $\varphi' \mathcal{U}_H^u \psi'$  is true in  $j$ . Since  $s$  is not the leftmost child of  $r$ , we have  $\chi(h, i)$ , and  $h < i$ , so (9.4) holds in  $i$ .

(b)  $h \doteq k$ , so  $v$  is the rightmost child of  $r$ .  $\varphi$  holds in all siblings between  $s$  and  $v$  (excluded), and  $\varphi'$  holds in the corresponding positions of  $w$ . All such positions  $j$ , if any, are s.t.  $\chi(h, j)$  and  $h < j$ , and they form a UHP, so  $\circ_H^u(\top \mathcal{U}_H^u \neg \varphi')$  never holds in  $i$ . Moreover, since  $\psi$  holds in  $v$ ,  $\psi'$  holds in  $k$ . Note that  $\chi_P^{\leq}$  in  $i$  uniquely identifies position  $h$ , and  $\chi_F^{\leq}$  evaluated in  $h$  identifies  $k$ . So, (9.5) holds in  $i$ .

2.  $s$  is the leftmost child of  $r$ . In this case, we have  $i = h + 1$  and  $h \triangleleft i$  (if  $h \doteq i$ , then  $r$  would have only one child).

- (a)  $h \triangleleft k$ .  $\ominus^{\triangleleft}$  evaluated in  $i$  identifies position  $h$ .  $\psi'$  holds in  $k$ , and  $\ominus_H^u(\top \mathcal{S}_H^u \neg \varphi')$  does not, because in all positions between  $i$  and  $k$  (excluded) corresponding to children of  $r$ ,  $\varphi'$  holds. Note that all such positions form a UHP, but  $i$  is not part of it ( $i = h + 1$ , so  $\neg \chi(h, i)$ ), and is not considered by  $\top \mathcal{S}_H^u \neg \varphi'$ . So, (9.6) holds in  $i$ .
- (b)  $h \doteq k$ , so  $v$  is the rightmost child of  $r$ .  $\psi$  holds in  $v$ , and  $\varphi$  holds in all children of  $r$ , except possibly the first ( $s$ ) and the last one ( $v$ ). These are exactly all positions s.t.  $\chi(h, j)$  and  $h \triangleleft j$ . Since  $\varphi'$  holds in all of them by hypothesis,  $\neg \chi_F^{\triangleleft} \neg \varphi'$  holds in  $h$ . Since  $\psi$  holds in  $v$ ,  $\psi'$  holds in  $k$ , and  $\chi_F^{\triangleleft} \psi'$  in  $h$ . So, (9.7) holds in  $i$ .

**[If]** We separately consider cases (9.4)–(9.7).

(9.4):  $\circ_H^u(\varphi' \mathcal{U}_H^u \psi')$  holds in a position  $i$  in  $w$ . Then, there exists a position  $h$  s.t.  $\chi(h, i)$  and  $h \triangleleft i$ , and a position  $j$  s.t.  $\chi(h, j)$  and  $h \triangleleft j$  that is the hierarchical successor of  $i$ , and  $\varphi' \mathcal{U}_H^u \psi'$  holds in  $j$ . So,  $i$  and  $j$  are consecutive children of  $r = \tau(h)$ . Moreover, there exists a UHP between  $j$  and a position  $k \geq j$ . The tree nodes corresponding to all positions in the path are consecutive children of  $r$ , so we fall in case 1a of the *only if* part of the proof. In  $T_w$ , a path between  $t = \tau(j)$  and  $v = \tau(k)$  witnesses the truth of  $\Rightarrow (\varphi, \psi)$  in  $s$ .

(9.5):  $\neg \circ_H^u(\top \mathcal{U}_H^u \neg \varphi' \wedge \chi_P^{\triangleleft}(\chi_F^{\triangleleft} \psi'))$  holds in position  $i \in w$  (this corresponds to case 1b). If  $\chi_P^{\triangleleft}(\chi_F^{\triangleleft} \psi')$  holds in  $i$ , then there exists a position  $h$  s.t.  $\chi(h, i)$  and  $h \triangleleft i$ , and a position  $k$  s.t.  $\chi(h, k)$  and  $h \doteq k$ , and  $\psi'$  holds in  $k$ .  $v = \tau(k)$  is the rightmost child of  $r = \tau(h)$ , parent of  $s = \tau(i)$ . Moreover, if  $\neg \circ_H^u(\top \mathcal{U}_H^u \neg \varphi')$  holds in  $i$ , then either:

- $\neg \circ_H^u \top$  holds, i.e. there is no position  $j > i$  s.t.  $\chi(h, j)$  and  $h \triangleleft j$ , so  $v$  is the immediate right sibling of  $s$ . In this case  $\Rightarrow (\varphi, \psi)$  holds in  $s$  because  $\psi$  holds in  $v$ .
- $\neg(\top \mathcal{U}_H^u \neg \varphi')$  holds in  $j > i$ , the first position after  $i$  s.t.  $\chi(h, j)$  and  $h \triangleleft j$ . This means  $\varphi'$  holds in all positions  $j' \geq j$  s.t.  $\chi(h, j')$  and  $h \triangleleft j'$ . Consequently, the tree nodes corresponding to these positions plus  $v = \tau(k)$  form a path witnessing  $\Rightarrow (\varphi, \psi)$ , which holds in  $s = \tau(i)$ .

(9.6):  $\ominus^{\triangleleft} \left( \chi_F^{\triangleleft} (\psi' \wedge \neg \circ_H^u(\top \mathcal{S}_H^u \neg \varphi')) \right)$  holds in  $i$ . Let  $h = i - 1$ , with  $h \triangleleft i$  (it exists because  $\ominus^{\triangleleft}$  is true). There exists a position  $k$ ,  $\chi(h, k)$  and  $h \triangleleft k$ , in which  $\psi'$  holds, so  $\psi$  does in  $v = \tau(k)$ , and  $\ominus_H^u(\top \mathcal{S}_H^u \neg \varphi')$  is false in it. If it is false because  $\neg \circ_H^u \top$  holds, there is no position  $j < k$  s.t.  $\chi(h, j)$  and  $h \triangleleft j$ , so  $v$  is the second child of  $r = \tau(h)$ ,  $s = \tau(i)$  being the first one. So,  $\Rightarrow (\varphi, \psi)$  trivially holds in  $s$  because  $\psi$  holds in the next sibling. Otherwise, let  $j < k$  be the rightmost position lower than  $k$  s.t.  $\chi(h, j)$  and  $h \triangleleft j$ .  $\neg(\top \mathcal{S}_H^u \neg \varphi')$  holds in it, so  $\varphi'$  holds in all positions  $j'$  between  $i$  and  $k$  that are part of the hierarchical path, i.e. s.t.  $\chi(h, j')$  and  $h \triangleleft j'$ . The corresponding tree nodes form a path ending in  $v = \tau(k)$  that witnesses the truth of  $\Rightarrow (\varphi, \psi)$  in  $s$  (case 2a).

(9.7):  $\ominus^{\triangleleft}(\chi_F^{\triangleleft} \psi' \wedge \neg \chi_F^{\triangleleft} \neg \varphi')$  holds in  $i$ . Then let  $h = i - 1$ ,  $h \triangleleft i$ , and  $s = \tau(i)$  is the leftmost child of  $r = \tau(h)$ . Since  $\chi_F^{\triangleleft} \psi'$  holds in  $h$ , there exists a position  $k$ , s.t.  $\chi(h, k)$  and  $h \doteq k$ , in which  $\psi'$  holds. So,  $\psi$  holds in  $v = \tau(k)$ , which is the rightmost child of  $r$ , by construction. Moreover,  $\varphi'$  holds in all positions s.t.  $\chi(h, j)$  and  $h \triangleleft j$ . Hence,

$\varphi$  holds in all corresponding nodes  $t = \tau(j)$ , which are all nodes between  $s$  and  $v$ , excluded. This, together with  $\psi$  holding in  $v$ , makes a path that verifies  $\Rightarrow (\varphi, \psi)$  in  $s$  (case 2b).  $\square$

**Lemma 9.8.** *Given an OP alphabet  $(AP, M_{AP})$ , for every  $\mathcal{X}_{\text{until}}$  formula  $\Leftarrow (\varphi, \psi)$ , and for any OP word  $w$  and position  $i$  in  $w$ , we have*

$$(T_w, \tau(i)) \models \Leftarrow (\varphi, \psi) \text{ iff } (w, i) \models \iota_{\mathcal{X}}(\Leftarrow (\varphi, \psi)).$$

$T_w \in \mathcal{T}_{M_{AP}}$  is the UOT obtained by applying function  $\tau$  to every position in  $w$ , such that for any position  $i'$  in  $w$   $(T_w, \tau(i')) \models \varphi$  iff  $(w, i') \models \iota_{\mathcal{X}}(\varphi)$ , and likewise for  $\psi$ .

*Proof.* Let  $\varphi' = \iota_{\mathcal{X}}(\varphi)$  and  $\psi' = \iota_{\mathcal{X}}(\psi)$ .

**[Only if]** Suppose  $\Leftarrow (\varphi, \psi)$  holds in  $s = \tau(i)$ . Then node  $r = \tau(h)$  s.t.  $rR_{\Downarrow}s$  has at least two children, and  $\Leftarrow (\varphi, \psi)$  is true because of a path starting in  $v = \tau(k)$ , s.t.  $rR_{\Downarrow}v$  and  $(T_w, v) \models \psi$  and ending in  $t = \tau(j)$  s.t.  $tR_{\Rightarrow}s$ . We distinguish between the following cases:

1.  $v$  is not the leftmost child of  $r$ .

- (a)  $h < i$ . By construction, all nodes in the path correspond to positions  $j' \in w$  s.t.  $\chi(h, j')$  and  $h < j'$ , so they form a UHP. Hence,  $\varphi' \mathcal{S}_H^u \psi'$  holds in  $j$ , and (9.8) holds in  $i$ .
- (b)  $h \doteq i$ . In this case,  $s$  is the rightmost child of  $r$ , and  $\chi(h, i)$ . The path made of positions between  $k$  and  $j$  corresponding to nodes between  $v$  and  $t$  (included) is a UHP. So  $\varphi' \mathcal{S}_H^u \psi'$  holds in  $j$ , which is the rightmost position of any possible such UHP: so  $\neg \circ_H^u \top$  also holds in  $j$ . Hence, (9.9) holds in  $i$ .

2.  $v$  is the leftmost child of  $r$ .

- (a)  $h < i$ . In this case,  $k = h + 1$  and  $\psi'$  holds in  $k$ . So,  $\circ^{\leq} \psi'$  holds in  $h$ , and  $\chi_P^{\leq}(\circ^{\leq} \psi')$  holds in  $i$ . Moreover, in all word positions  $j'$  with  $k < j' < j$  corresponding to children of  $r$ ,  $\varphi'$  holds. Such positions form a UHP. So  $\neg \circ_H^u (\top \mathcal{S}_H^u \neg \varphi')$  holds in  $i$ . Note that this is true even if  $s$  is the first right sibling of  $v$ . Thus, (9.10) holds in  $i$ .
- (b)  $h \doteq i$ .  $\psi'$  holds in  $k = h + 1$ , so  $\circ^{\leq} \psi'$  holds in  $h$ . Since  $\chi(h, i)$  and  $h \doteq i$ ,  $\chi_P^{\doteq}(\circ^{\leq} \psi')$  holds in  $i$ . Moreover,  $\varphi$  holds in all children of  $r$  except the first and last one, so  $\varphi'$  holds in all positions  $j'$  s.t.  $\chi(h, j')$  and  $h < j'$ . So  $\neg \chi_F^{\leq} \neg \varphi'$  holds in  $h$ , and (9.11) in  $i$ .

**[If]** We separately consider cases (9.8)–(9.11).

(9.8):  $\circ_H^u(\varphi' \mathcal{S}_H^u \psi')$  holds in  $i$ . Then, there exists a position  $h$  s.t.  $\chi(h, i)$  and  $h < i$ , and a position  $j < i$  s.t.  $\chi(h, j)$  and  $h < j$ . Since  $j \neq h + 1$ , the corresponding tree node is not the leftmost one. So, this corresponds to case 1a, and  $\Leftarrow (\varphi, \psi)$  holds in  $s = \tau(i)$ .

(9.9):  $\chi_P^{\doteq}(\chi_F^{\leq}(\neg \circ_H^u \top \wedge \varphi' \mathcal{S}_H^u \psi'))$  holds in  $i$ . Then, there exists a position  $h$  s.t.  $\chi(h, i)$  and  $h \doteq i$ . Moreover, at least a position  $j'$  s.t.  $\chi(h, j')$  and  $h < j'$  exists. Let  $j$  be the rightmost one, i.e. the only one in which  $\neg \circ_H^u \top$  holds. The corresponding tree node  $t = \tau(j)$  is s.t.  $tR_{\Rightarrow}s$ , with  $s = \tau(i)$ . Since  $\varphi' \mathcal{S}_H^u \psi'$  holds in  $j$ , a UHP starts from it, and  $\psi$  and  $\varphi$  hold in the tree nodes corresponding to, respectively, the first and all other positions in the path. This is case 1b, and  $\Leftarrow (\varphi, \psi)$  holds in  $s$ .

(9.10):  $\chi_P^{\leq}(\circ^{\leq} \psi') \wedge \neg \ominus_H^u(\top \mathcal{S}_H^u \neg \varphi')$  holds in  $i$ . Then, there exists a position  $h$  s.t.  $\chi(h, i)$  and  $h < i$ .  $\psi'$  holds in  $k = h + 1$ , so  $\psi$  holds in the leftmost child of  $r = \tau(h)$ . Moreover,  $\varphi'$  holds in all positions  $j' < i$  s.t.  $\chi(h, j')$  and  $h < j'$ , so  $\varphi$  holds in all children of  $r$  between  $v = \tau(k)$  and  $s = \tau(i)$ , excluded. This is case 2a, and  $\Leftarrow (\varphi, \psi)$  holds in  $s$ .

(9.11):  $\chi_P^{\dot{=}}(\circ^{\leq} \psi' \wedge \neg \chi_F^{\leq} \neg \varphi')$  holds in  $i$ . Then, there exists a position  $h$  s.t.  $\chi(h, i)$  and  $h \dot{=} i$ .  $\circ^{\leq} \psi'$  holds in  $h$ , so  $\psi$  holds in node  $v = \tau(h + 1)$ , which is the leftmost child of  $r = \tau(h)$ . Since  $\neg \chi_F^{\leq} \neg \varphi'$  holds in  $h$ ,  $\psi'$  holds in all positions  $j'$  s.t.  $\chi(h, j')$  and  $h < j'$ . So,  $\psi$  holds in all children of  $r$  except (possibly) the leftmost ( $v$ ) and the rightmost ( $s = \tau(i)$ ) ones. This is case 2b, and  $\Leftarrow (\varphi, \psi)$  holds in  $s$ .  $\square$

It is possible to express all POTL operators in FOL, as per Section 9.1.1. From this, and Lemmas 9.6, 9.7, and 9.8 together with Theorem 9.5, we derive

**Theorem 9.9.** *POTL = FOL with one free variable on finite OP words.*

**Corollary 9.10.** *The propositional operators plus  $\circ^d, \ominus^d, \chi_F^d, \chi_P^d, \mathcal{U}_\chi^d, \mathcal{S}_\chi^d, \circ_H^u, \ominus_H^u, \mathcal{U}_H^u, \mathcal{S}_H^u$  are expressively complete on OP words.*

**Corollary 9.11.** *NWTL  $\subset$  OPTL  $\subset$  POTL over finite OP words.*

**Corollary 9.12.** *Every FO formula with at most one free variable is equivalent to one using at most three distinct variables on finite OP words.*

Corollary 9.10 follows from the definition of  $\iota_{\mathcal{X}}$  and Theorem 9.9 (note that all other operators are shortcuts for formulas expressible with those listed). In Corollary 9.11, NWTL  $\subset$  OPTL was proved in [54], and OPTL  $\subseteq$  POTL comes from Theorem 9.9 and the semantics of OPTL being expressible in FOL similarly to POTL, while OPTL  $\subsetneq$  POTL comes from Theorem 7.9. Corollary 9.12, stating that OP words have the three-variable property, follows from the FOL semantics of POTL being expressible with just three variables.

## 9.2 First-Order Completeness on $\omega$ -Words

To prove the FO-completeness of the translation of  $\mathcal{X}_{until}$  into POTL also on OP  $\omega$ -words, we must prove that  $\mathcal{X}_{until}$  is FO-complete on the OPM-compatible UOTs resulting from  $\omega$ -words. In Section 9.2.1 we show that infinite OPM-compatible UOTs can be divided in two classes, depending on their shape. Then, after recalling some notation in Section 9.2.2, we show how to translate a given FO formula into  $\mathcal{X}_{until}$  separately for each UOT class in Sections 9.2.3 and 9.2.4, and how such translations can be combined to work on any infinite OPM-compatible UOT in Section 9.2.5. Our proofs exploit composition arguments on trees from [91, 117, 132] but introduce new techniques to deal with the peculiarities of UOTs derived from  $\omega$ OPLs.

### 9.2.1 OPM-compatible $\omega$ -UOTs

The application of function  $\tau$  from Section 9.1 to OP  $\omega$ -words results in two classes of infinite UOTs, depending on the shape of the underlying ST. In both cases, in the UOT the rightmost child of the root is not labeled with  $\#$ , and nodes in the rightmost branch do not have a *right context*.  $\tau^{-1}$ , which can be defined in the same way, converts such UOTs into words with open chains.



reaches the same size infinitely many times. The rightmost branch of  $T_w = \tau(w)$  ends with a node  $r_\infty$  with an infinite number of children (cf. Figure 9.3). Node  $r_\infty$  is in the  $<$  relation with all of its children, otherwise it would violate property 3 of the  $\chi$  relation. This is the Left-Recursive (LR) class of UOTs.

An exception to the above classification may occur if the OPM is such that the transitive closure of the  $\dot{=}$  relation is reflexive—in other words the OPM contains  $\dot{=}$ -circularities—. In this case the ST *may* contain just one node with an infinite number of children, all in the  $\dot{=}$  PR. As a result, such nodes form *a unique infinite branch* in the corresponding UOT whose nodes are in the  $\dot{=}$  relation unlike the case of Figure 9.2. This is the distinguishing feature of RR UOTs, despite the fact that in this case the stack of the  $\omega$ OPBA remains “ultimately bounded” as in the case of LR UOTs. Thus, this exceptional case is attributed to the RR class.

In the following, by RR (resp. LR) word or ST we mean an OP  $\omega$ -word or ST that translates to a RR (resp. LR) UOT. Next, we separately give a translation of FOL into  $\mathcal{X}_{\text{until}}$  for RR and LR UOTs, and then show how to combine them to obtain completeness.

*Remark 9.13.* There cannot be RR UOTs containing a node with infinitely many children, or LR UOTs where more than one node has infinite children. If this was the case, then the infinite children would appear as consecutive infinite subsets of positions in the OP  $\omega$ -word isomorphic to the UOT. But this is impossible, because the set of word positions is  $\mathbb{N}$ , which is not dense and does not contain dense subsets.

*Remark 9.14.* The FO-completeness proof of the logic NWTL [12] is also based on a translation of Nested Words to UOTs. However, Nested Words result in only one kind of UOT, because VPL grammars can be transformed so that words grow in only one direction. Thus, that proof does not deal with the issue of combining two separate translations.

## 9.2.2 Notation

Throughout the rest of this section, we use the model theoretic notions and notation that we introduced in Chapter 5.

We refer to the syntax and semantics of  $\mathcal{X}_{\text{until}}$  presented in Section 9.1.2.2. The semantics of the until and since operators is strict, i.e. the position where they are evaluated is not part of their paths, which start with the next one. Thus, next and back operators are not needed, but we define them as shortcuts, for any  $\mathcal{X}_{\text{until}}$  formula  $\varphi$ :

$$\begin{aligned} \circ\Downarrow\varphi &:= \Downarrow(\neg\top, \varphi) & \circ\Uparrow\varphi &:= \Uparrow(\neg\top, \varphi) \\ \circ\Rightarrow\varphi &:= \Rightarrow(\neg\top, \varphi) & \circ\Leftarrow\varphi &:= \Leftarrow(\neg\top, \varphi) \end{aligned}$$

We also use LTL, but with strict semantics, which is slightly different from what we presented in Chapter 2. Hence, we re-define it formally below. We call the structure  $\langle U, <, P \rangle$  a finite LTL word if  $U \subseteq \mathbb{N}$  is finite, and an LTL  $\omega$ -word if  $U = \mathbb{N}$ .  $<$  is a linear order on  $U$ , and  $P: AP \rightarrow \mathcal{P}(U)$  is a labeling function. The syntax of an LTL formula  $\varphi$  is, for  $a \in AP$ ,  $\varphi ::= a \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathcal{U} \varphi \mid \varphi \mathcal{S} \varphi$ , where propositional operators have the usual meaning. Given an LTL word  $w$ , and a position  $i$  in it,

- $(w, i) \models a$  iff  $i \in P(a)$ ;
- $(w, i) \models \psi \mathcal{U} \theta$  iff there is  $j > i$  s.t.  $(w, j) \models \theta$  and for any  $i < j' < j$  we have  $(w, j') \models \psi$ ;

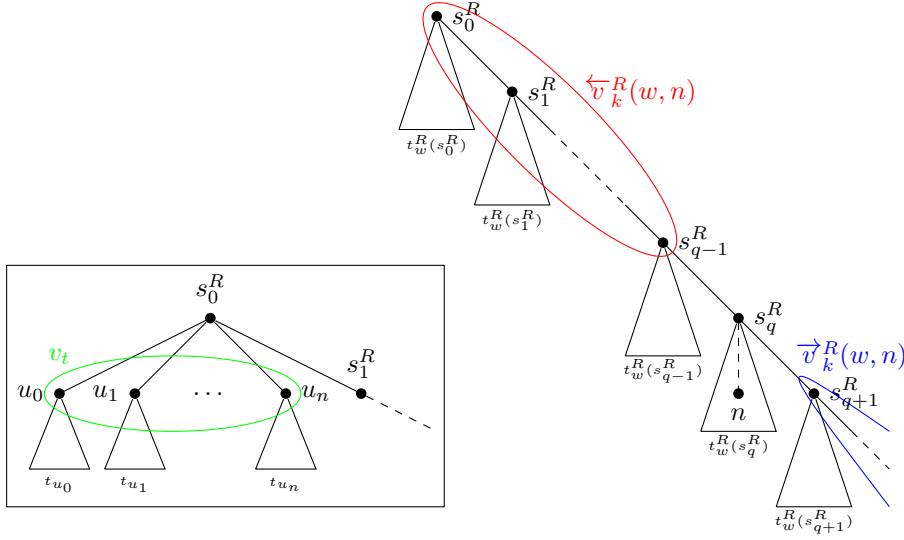


Figure 9.4: Parts in which we divide a RR UOT for Lemma 9.16 (right) and Lemma 9.17 (left).

- $(w, i) \models \psi \mathcal{S} \theta$  iff there is  $j < i$  s.t.  $(w, j) \models \theta$  and for any  $j < j' < i$  we have  $(w, j') \models \psi$ .

A *future* LTL formula only contains the  $\mathcal{U}$  modality, and a *past* one only contains  $\mathcal{S}$ .

### 9.2.3 RR UOTs

We prove the following:

**Lemma 9.15.** *Given a FO formula on UOTs  $\bar{\varphi}(x)$  of quantifier rank  $k \geq 1$ , there exists a  $\mathcal{X}_{\text{until}}$  formula  $\varphi_R$  s.t. for any OP  $\omega$ -word  $w$  s.t.  $T_w = \tau(w)$  is a RR UOT, and for any node  $n \in T_w$  we have  $(T_w, n) \models \bar{\varphi}(x)$  iff  $(T_w, n) \models \varphi_R$ .*

To prove Lemma 9.15, we show that  $\varphi_R$  can be built by combining formulas describing only parts of a UOT, as hinted in Figure 9.4. First, we prove that the rank- $k$  type of such parts determines the rank- $k$  type of the whole tree (Lemma 9.16); then, since such subdivision includes finite subtrees, we show how to express their rank- $k$  types in  $\mathcal{X}_{\text{until}}$  (Lemma 9.17); finally we use such results to translate  $\bar{\varphi}(x)$ .

This part of the proof partially resembles the one for the FO-completeness of NWTL in [12], because of the similarity between the shape of RR UOTs and those resulting from nested  $\omega$ -words. However, the two proofs diverge significantly, because Nested Words are isomorphic to binary trees, while RR UOTs are unranked.

Consider the RR UOT  $T_w$  of the statement. We denote with  $s_0^R, s_1^R, \dots$  the infinite sequence of *pending* nodes obtained by starting from the root of  $T_w$ , and always descending through the rightmost child. We call  $t_w^R(s_p^R)$  the finite subtree obtained by removing the rightmost child of  $s_p^R$  and its descendants from the subtree rooted in  $s_p^R$ . If  $s_p^R$  has one single child,  $t_w^R(s_p^R)$  is made of  $s_p^R$  only. Let  $n$  be any node in  $T_w$ , and let  $s_q^R$  be s.t.  $n$  is part of  $t_w^R(s_q^R)$ , and  $s_q^R = n$  if  $n$  is a pending node. Let  $\Gamma_k$  be the (finite) set of all rank- $k$  types of finite UOTs, and  $\sigma_k^R(w, n) \in \Gamma_k$  be the rank- $k$

type of  $t_w^R(s_q^R)$ . We define  $\overleftarrow{v}_k^R(w, n)$  as a finite LTL word of length  $q$  on alphabet  $\Gamma_k$ , s.t. each position  $p$ ,  $0 \leq p \leq q-1$ , is labeled with  $\sigma_k^R(w, s_p^R)$ . Also, let  $\overrightarrow{v}_k^R(w, n)$  be a LTL  $\omega$ -word on  $\Gamma_k$  having each position labeled with  $\sigma_k^R(w, s_{q+j+1}^R)$  for all  $j \geq 0$ . We give the following composition argument:

**Lemma 9.16.** *Let  $w_1$  and  $w_2$  be two OP  $\omega$ -words, such that  $T_{w_1} = \tau(w_1)$  and  $T_{w_2} = \tau(w_2)$  are two RR UOTs. Let  $i_1$  and  $i_2$  be two positions in, resp.,  $w_1$  and  $w_2$ , and let  $s_1 = \tau(i_1)$  and  $s_2 = \tau(i_2)$ . For any  $k \geq 1$ , if*

1.  $\overleftarrow{v}_k^R(w_1, s_1) \equiv_k \overleftarrow{v}_k^R(w_2, s_2)$ ,
2.  $\overrightarrow{v}_k^R(w_1, s_1) \equiv_k \overrightarrow{v}_k^R(w_2, s_2)$  and
3.  $\sigma_k^R(w_1, s_1) = \sigma_k^R(w_2, s_2)$ ,

then  $(T_{w_1}, s_1) \equiv_k (T_{w_2}, s_2)$ .

*Proof.* By Theorem 5.11, equivalences 1–2–3 imply that the respective games have a winning strategy; we refer to them by game 1, 2 and 3. We prove that  $(T_{w_1}, s_1) \sim_k (T_{w_2}, s_2)$ , i.e. that  $\exists$  has a winning strategy in the EF game on  $(T_{w_1}, s_1)$  and  $(T_{w_2}, s_2)$  so that the thesis follows. In round 0 of the game, partial isomorphism is ensured by  $s_1$  and  $s_2$  having the same labels, due to hypothesis 3. In any subsequent round, suppose w.l.o.g. that  $\forall$  picks a node  $s^{\forall}$  from  $T_{w_1}$  (the converse is symmetric). Let  $s_{q_1}^R$  be the pending node s.t.  $s_1$  is part of  $t_w^R(s_{q_1}^R)$ , and  $s_{q_2}^R$  be the pending node s.t.  $s_2$  is part of  $t_w^R(s_{q_2}^R)$ . We have the following cases:

- $s^{\forall} = s_{q^{\forall}}^R$  is one of the pending nodes that are ancestors of  $s_{q_1}^R$ , and  $q^{\forall}$  is the corresponding position in  $\overleftarrow{v}_k^R(w_1, s_1)$ . Then,  $\exists$  selects  $q^{\exists}$  in  $\overleftarrow{v}_k^R(w_2, s_2)$  in response to  $q^{\forall}$  according to her winning strategy for game 1. Her answer to  $s^{\forall}$  is  $s^{\exists} = s_{q^{\exists}}^R$  (i.e. the pending node with the same index).
- $s^{\forall}$  is part of a subtree  $t_{w_1}^R(s_{q^{\forall}}^R)$  such that  $s_{q^{\forall}}^R$  is an ancestor of  $s_{q_1}^R$ . Then  $\exists$  chooses position  $q^{\exists}$  in  $\overleftarrow{v}_k^R(w_2, s_2)$  as before. According to game 1,  $q^{\forall}$  and  $q^{\exists}$  must be labeled with the same rank- $k$  type of a subtree (i.e.  $\sigma_k^R(w_1, s_{q^{\forall}}^R) = \sigma_k^R(w_2, s_{q^{\exists}}^R)$ ). Hence, game  $t_{w_1}^R(s_{q^{\forall}}^R) \sim_k t_{w_2}^R(s_{q^{\exists}}^R)$  has a winning strategy, which  $\exists$  can use to pick  $s^{\exists}$  in  $t_{w_2}^R(s_{q^{\exists}}^R)$ .
- If the node picked by  $\forall$  is the rightmost child of  $s_{q_1}^R$  or one of its descendants, then  $\exists$  proceeds symmetrically, but using her winning strategy on game 2.
- Finally, if  $\forall$  picks a node in  $t_{w_1}^R(s_{q_1}^R)$ , then by 3 we have  $t_{w_1}^R(s_{q_1}^R) \sim_k t_{w_2}^R(s_{q_2}^R)$ , and  $\exists$  answers according to her winning strategy in this game.

This strategy preserves the partial isomorphism w.r.t. the child and sibling relations and monadic predicates, as a direct consequence of rank- $k$  type equivalences in the hypotheses.  $\square$

Lemma 9.16 shows that the rank- $k$  type of an RR OPM-compatible UOT is determined by the rank- $k$  types of the parts in which we divide it. Given FO formula  $\varphi(x)$ , consider the set of all tuples made of 1. the rank- $k$  type of  $\overleftarrow{v}_k^R(w, s)$ , 2. the rank- $k$  type of  $\overrightarrow{v}_k^R(w, s)$ , and 3. the type  $\sigma_k^R(w, s)$ , such that  $(T_w, s) \models \varphi(x)$  for any RR UOT  $T_w$  and  $s \in T_w$ . This set is finite, because there are only finitely many rank- $k$  types

of each component. So we can translate the Hintikka formulas expressing the types in each tuple into  $\mathcal{X}_{\text{until}}$  separately, and combine them to obtain one for the whole tree. Then,  $\mathcal{X}_{\text{until}}$  formula  $\varphi$  is a disjunction of the resulting translated formulas, one for each tuple, similarly to Lemma 5.8.

Before proceeding with our translation, we need to show how to express in  $\mathcal{X}_{\text{until}}$  the rank- $k$  type of a finite UOT such as  $t_w^R(s_p^R)$ , for some  $w$  and  $p$ , in the context of  $T_w$ . Since the rank- $k$  type of  $t_w^R(s_p^R)$  only contains information about that subtree, we need to restrict the formula expressing it to such nodes. In the following lemma we show how to do this, thanks to a formula  $\alpha^*$  which holds in the root of the subtree (but may hold in other parts of  $T_w$ ), and allows us to restrict  $\mathcal{X}_{\text{until}}$  operators so that they do not exit the subtree.

**Lemma 9.17.** *Let  $\sigma_k(t)$ , with  $k \geq 1$ , be the rank- $k$  type of a finite OPM-compatible UOT  $t$ . Let  $r$  be a node of a RR or LR OPM-compatible UOT  $T_w$  with a finite number of children. Let  $\alpha^*$  be a  $\mathcal{X}_{\text{until}}$  formula that holds in  $r$ , and does not hold in subtrees rooted at children of  $r$ , except possibly the rightmost one. Then, there exists a  $\mathcal{X}_{\text{until}}$  formula  $\beta(t)$  that, if evaluated in  $r$ , is true iff  $r$  is the root of a subtree of  $T_w$  with rank- $k$  type  $\sigma_k(t)$ , from which the subtree rooted in  $r$ 's rightmost child has been erased in case  $\alpha^*$  holds in that child.*

*Proof.* Let  $t_w$  be the subtree rooted at  $r$ , excluding  $r$ 's rightmost child and its descendants, if  $\alpha^*$  holds in it. We provide a formula  $\beta(t)$  that holds in  $r$  iff  $t_w \equiv_k t$ . If  $t$  has only one node  $s$  with no children, which can be determined with a FO formula with one quantifier, then  $\beta(t) := \beta_{AP}(s) \wedge \neg \bigcirc_{\Downarrow} (\neg \alpha^*)$ , where  $\beta_{AP}(s)$  is a Boolean combination of the atomic propositions holding in  $s$ .

Otherwise, the rank- $k$  type of  $t$  is fully determined by

1. the propositional symbols holding in its root  $s$ ;
2. the rank- $k$  type of the finite LTL word  $v_t$ , whose positions  $0 \leq p \leq m$  are labeled with the rank- $k$  types of the subtrees  $t_{u_p}$  rooted at the children  $u_p$  of  $s$  (including the rightmost one).

This can be proved with a simple composition argument.

Thus, we define

$$\beta(t) := \bigcirc_{\Downarrow} (\neg \bigcirc_{\Leftarrow} \top \wedge \beta'(t)) \wedge \beta_{AP}(s),$$

where  $\beta_{AP}(s)$  is a Boolean combination of the atomic propositions holding in  $s$ , and  $\beta'(t)$  characterizes  $t_{u_0}, \dots, t_{u_m}$ . By this we mean that  $\beta'(t)$  is such that when  $\beta(t)$  holds on  $r$ , its children (except the rightmost one if  $\alpha^*$  holds in it) must be roots of subtrees isomorphic to  $t_{u_0}, \dots, t_{u_m}$ . Note that formula  $\beta'(t)$  is enforced in the leftmost child (where  $\bigcirc_{\Leftarrow} \top$  is false) of the node where  $\beta(t)$  is evaluated.

We now show how to obtain  $\beta'(t)$ . Due to Kamp's Theorem [106] and the separation property of LTL [86], there exists a future LTL formula  $\beta''(t)$  that, evaluated in the first position of  $v_t$ , completely determines its rank- $k$  type (it can be obtained by translating into LTL the Hintikka formula equivalent to the rank- $k$  type of  $v_t$ ).

We now show how to express the rank- $k$  types of the subtrees  $t_{u_p}$ . Since they are finite, by Marx's Theorem [126], there exists a  $\mathcal{X}_{\text{until}}$  formula  $\gamma_p$  that, evaluated in  $u_p$ , fully determines the rank- $k$  type of  $t_{u_p}$  ( $\gamma_p$  can be obtained by translating the Hintikka formula for the rank- $k$  type of  $t_{u_p}$  into  $\mathcal{X}_{\text{until}}$ ). Unfortunately, the separation

property does not hold for  $\mathcal{X}_{\text{until}}$  [24], and  $\gamma_p$  may contain  $\uparrow$  operators that, in the context of  $T_w$ , consider nodes that are not part of  $t_{u_p}$ .

There is, however, a way of syntactically transforming  $\gamma_p$  so that, if evaluated on a child  $r'$  of  $r$ , its paths remain constrained to  $t_{r'}$ , the subtree rooted in  $r'$ . Given a  $\mathcal{X}_{\text{until}}$  formula  $\psi$ , it can be written as a Boolean combination of atomic propositions and until/since operators (possibly nested). We denote by  $\psi^\downarrow$  the formula obtained by replacing all subformulas of the form  $\uparrow(\varphi, \varphi')$ ,  $\Rightarrow(\varphi, \varphi')$  and  $\Leftarrow(\varphi, \varphi')$  at the topmost level with  $\neg\top$ . If  $\gamma_p$  is evaluated in the root of  $t_{u_p}$  outside of  $t$ , all such operators evaluate to false. So,  $\gamma_p^\downarrow$  in  $r'$  agrees with  $\gamma_p$  in the root of  $t_{u_p}$  on such subformulas. Now, take  $\gamma_p^\downarrow$ , and recursively replace all subformulas of the form  $\uparrow(\varphi, \varphi')$  with the following:

$$\uparrow(\varphi \wedge \neg \circ_{\uparrow} \alpha^*, (\neg \circ_{\uparrow} \alpha^* \wedge \varphi') \vee (\circ_{\uparrow} \alpha^* \wedge \varphi'^{\downarrow})).$$

We call  $\gamma'_p$  the obtained formula. Note that, since  $\alpha^*$  holds in  $r$  and at most in its rightmost child,  $\circ_{\uparrow} \alpha^*$  only holds in nodes  $u_p$ ,  $0 \leq p \leq m$ . This way, we can prove that  $\uparrow$  paths in  $\gamma'_p$  cannot continue past  $u_p$ , and those ending in  $u_p$  depend on a formula that does not consider nodes outside the subtree rooted at  $u_p$ . Thus, when  $\gamma'_p$  is evaluated in  $u_p$  in the context of  $T_w$ , all its until/since operators consider exactly the same positions as the corresponding ones in  $\gamma_p$  evaluated outside of  $T_w$ . Hence, the two formulas are equivalent in their respective contexts, and  $\gamma'_p$  is true in positions that are the root of a subtree equivalent to  $t_{u_p}$ .

We now show that such transformations do not change the formula's meaning in unwanted ways. Let  $t_1$  be a UOT, and let  $t_2$  be a subtree of a larger UOT  $T$ , such that  $t_1$  and  $t_2$  are identical, and  $\alpha^*$  holds in the parent of  $r_2$ , the root of  $t_2$ . We prove that  $\gamma'_p$  holds on  $r_2$  iff  $\gamma_p$  holds on  $r_1$ , the root of  $t_1$ . For any subformula  $\psi$  of  $\gamma_p$  not at the topmost nesting level, let  $\psi'$  be the corresponding subformula in  $\gamma'_p$ . Then, we prove by structural induction on  $\psi$  that, for any node  $s \neq r_1$  in  $t_1$  and  $t_2$ , we have  $(t_1, s) \models \psi$  iff  $(T, s) \models \psi'$ .

The base case, where  $\psi$  is an atomic proposition, is trivial. The composition by Boolean operators is also straightforward.

Paths considered by formulas of the form  $\Rightarrow(\varphi_1, \varphi_2)$ ,  $\Leftarrow(\varphi_1, \varphi_2)$ , and  $\Downarrow(\varphi_1, \varphi_2)$  cannot get out of  $t_2$ , when evaluated in one of its nodes that is not  $r_2$ . Thus, we have e.g.  $(t_1, s) \models \Rightarrow(\varphi_1, \varphi_2)$  iff  $(T, s) \models \Rightarrow(\varphi'_1, \varphi'_2)$  directly from the fact that  $(t_1, s') \models \varphi_1$  iff  $(T, s') \models \varphi'_1$  for any  $s' \in t_1$  (and the same for  $\varphi_2$ ).

Finally, let  $\psi = \uparrow(\varphi_1, \varphi_2)$ . Suppose  $\psi$  is witnessed by a path in  $t_1$  that does not include its root  $r_1$ . Then,  $\neg \circ_{\uparrow} \alpha^*$  holds in all positions in such path inside  $t_2$ , and  $\psi'$  is satisfied by the same path, thanks to the inductive hypothesis. Otherwise,  $\psi$  may be witnessed by a path ending in  $r_1$ . In this case,  $\neg \circ_{\uparrow} \alpha^*$  holds in all positions in the corresponding path in  $t_2$  except the last one. There,  $(\circ_{\uparrow} \alpha^* \wedge \varphi'^{\downarrow})$  holds. In fact,  $\varphi$  holds in the root of  $t_1$  where any  $\uparrow$ ,  $\Rightarrow$  and  $\Leftarrow$  operator are false, so they can correctly be replaced with  $\neg\top$  in  $\varphi'^{\downarrow}$ . Moreover,  $\Downarrow$  operators are strict, so they must be witnessed by paths completely contained in  $t_1$ , excluding its root. Hence, by the inductive hypothesis they witness the corresponding operators in  $\varphi'^{\downarrow}$ . Conversely, any path that witnesses  $\psi'$  in  $s \in t_2$  must be, by construction, entirely inside  $t_2$ , so it witnesses  $\psi$  in  $t_1$  as well. Finally,  $\gamma'_p$  holding on  $r_2$  can be justified by an argument similar to the one for  $\varphi'^{\downarrow}$ .

Now, we can go on with the definition of  $\beta(t)$ : we set  $\beta'(t) := \neg\alpha^* \wedge \beta'''(t)$ , where  $\beta'''(t)$  is obtained from  $\beta''(t)$  by

- recursively replacing all subformulas  $\varphi \mathcal{U} \varphi'$  with  $\Rightarrow(\neg\alpha^* \wedge \varphi, \neg\alpha^* \wedge \varphi')$ ;

- replacing all labels, which are rank- $k$  types of UOTs  $t_{u_p}$ , with their corresponding  $\gamma'_p$ .

Thus,  $\beta'(t)$  is a  $\mathcal{X}_{until}$  formula that characterizes children of  $r$ , without considering the one where  $\alpha^*$  holds (if any).  $\square$

Lemma 9.17 allows us to express in  $\mathcal{X}_{until}$  the rank- $k$  type of any subtree  $t = t_w^R(s_p^R)$ , for any pending node  $s_p^R$  in  $T_w$ . The root of  $t$  is a pending node, and so is its rightmost child  $s_{p+1}^R$ , which is not part of  $t$ . So, we use  $\alpha_\infty^R := \neg \uparrow (\top, \circ \Rightarrow \top)$ , which is true in pending nodes, where the path from the current node to the root is made of rightmost nodes only. Thus, formula  $\beta(t)$  from Lemma 9.17 is true in a pending node iff it is the root of a subtree equivalent to  $t$ , excluding its rightmost child.

Let  $n$  be the node where the translated formula is evaluated (i.e. the one corresponding to the free variable in the FO formula  $\bar{\varphi}(x)$  to be translated). Let  $s_{p_n}^R$  be its closest pending ancestor. According to Lemma 9.16, we need to express the rank- $k$  types of  $t_w^R(s_{p_n}^R)$ ,  $\vec{v}_k^R(w, n)$ , and  $\overleftarrow{v}_k^R(w, n)$ .

For  $t_w^R(s_{p_n}^R)$ , simply take formula  $\beta(t_w^R(s_{p_n}^R))$ .

By Kamp's Theorem and the separation property of LTL, the rank- $k$  type of word  $\vec{v}_k^R(w, n)$  can be expressed by a future LTL formula  $\vec{\psi}$  to be evaluated in its first position. First, we recursively take the conjunction of all subformulas of  $\vec{\psi}$  with  $\alpha_\infty^R$ . Then, we recursively substitute each LTL operator  $\varphi \mathcal{U} \varphi'$  with  $\Downarrow(\varphi, \varphi')$ , obtaining a  $\mathcal{X}_{until}$  formula  $\vec{\psi}'$  that evaluates its paths only on pending positions in  $T_w$ . We can prove that equivalence is kept despite such transformations by induction on the formula's structure, as we did in Lemma 9.17. Since  $\vec{v}_k^R(w, n)$  is labeled with rank- $k$  types of UOTs, we substitute any such atomic proposition  $\sigma_k(t)$  in  $\vec{\psi}'$  with  $\beta(t)$ , obtaining formula  $\vec{\psi}''$ , that captures the part of the tree rooted at the rightmost child of  $s_{p_n}^R$ .

A formula  $\overleftarrow{\psi}''$  for  $\overleftarrow{v}_k^R(w, n)$  can be obtained similarly, but using a past LTL formula. The Since modality can be replaced with  $\uparrow$ , while other transformations remain the same.

All formulas we built so far are meant to be evaluated in a pending node. If we have  $n = s_{p_n}^R$ , then we are done. Otherwise, we evaluate in  $n$  an appropriate  $\uparrow$  formula, that can only be witnessed by a path ending in  $s_{p_n}^R$ . Thus, the final translation is

$$\varphi_R := (\alpha_\infty^R \wedge \varphi'_R) \vee \uparrow (\neg \alpha_\infty^R, \alpha_\infty^R \wedge \varphi'_R)$$

where

$$\varphi'_R := \beta(t_w^R(s_{p_n}^R)) \wedge \circ \Downarrow (\neg \circ \Rightarrow \top \wedge \vec{\psi}'') \wedge \circ \uparrow \overleftarrow{\psi}''.$$

This concludes the proof of Lemma 9.15.  $\square$

## 9.2.4 LR UOTs

Let  $w$  be an OP  $\omega$ -word, and  $T_w = \tau(w)$  a LR UOT. We name  $r_\infty$  the node with infinite children, and denote as  $t_f(w)$  the finite UOT obtained by removing all children of  $r_\infty$  from  $T_w$ . We prove the following:

**Lemma 9.18.** *Given a FO formula on UOTs  $\bar{\varphi}(x)$  of quantifier rank  $k \geq 1$ , there are two  $\mathcal{X}_{until}$  formulas  $\varphi_{L1}$  and  $\varphi_{L2}$  s.t. for any OP  $\omega$ -word s.t.  $T_w = \tau(w)$  is a LR UOT, and for any node  $n \in T_w$  we have*

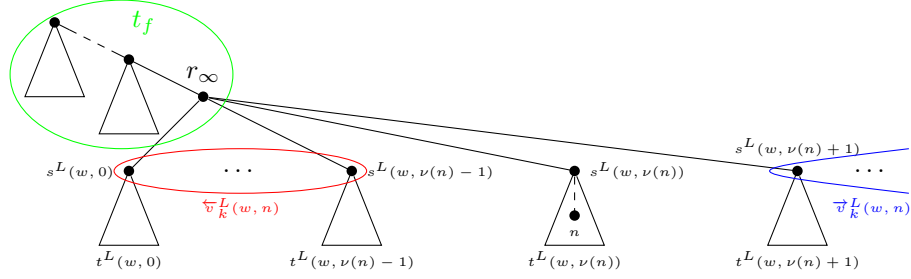


Figure 9.5: Parts in which we divide a LR UOT for Lemma 9.19.

- $(T_w, n) \models \bar{\varphi}(x)$  iff  $(T_w, n) \models \varphi_{L1}$  when  $n \notin t_f(w)$ , and
- $(T_w, n) \models \bar{\varphi}(x)$  iff  $(T_w, n) \models \varphi_{L2}$  when  $n \in t_f(w)$ .

The proof is structured in a way similar to that of Lemma 9.15, but differs in the parts in which we divide the tree. In fact, we divide a LR UOT into its finite part, and two LTL words which make up the infinite children of  $r_\infty$  (see Figure 9.5).

We name  $s^L(w, 0), s^L(w, 1), \dots$  the children of  $r_\infty$ , and we denote as  $t^L(w, 0), t^L(w, 1), \dots$  the finite subtrees rooted at, resp.,  $s^L(w, 0), s^L(w, 1), \dots$  and  $\sigma_k^L(w, 0), \sigma_k^L(w, 1), \dots$  their rank- $k$  types. For all  $p \geq 0$ , for any node  $m$  in  $t^L(w, p)$ , we define the map  $\nu$  so that  $\nu(m) = p$ . Now, let  $n$  be any node in  $T_w$ . We define  $\overrightarrow{v}_k^L(w, n)$  as the LTL  $\omega$ -word on the alphabet of rank- $k$  types of finite OPM-compatible UOTs, such that each of its positions  $i_q, q \geq 0$ , is labeled with  $\sigma_k^L(w, \nu(n) + q + 1)$  if  $n \notin t_f(w)$ . If  $n$  is in  $t_f(w)$ , then each position  $i_q$  is labeled with  $\sigma_k^L(w, q)$ . We further define  $\overleftarrow{v}_k^L(w, n)$  as the finite LTL word of length  $\nu(n)$  on the same alphabet, such that each of its positions  $i_q, 0 \leq q \leq \nu(n) - 1$ , is labeled with  $\sigma_k^L(w, q)$ . If  $\nu(n) = 0$ , or  $n$  is part of  $t_f(w)$ , then  $\overleftarrow{v}_k^L(w, n)$  is the empty word. We now prove the following composition argument:

**Lemma 9.19.** *Let  $w_1$  and  $w_2$  be two OP  $\omega$ -words, such that  $T_{w_1} = \tau(w_1)$  and  $T_{w_2} = \tau(w_2)$  are two LR UOTs, and  $r_\infty^1$  and  $r_\infty^2$  are their nodes with infinitely many children. Let  $i_1$  and  $i_2$  be two positions in  $w_1$  and  $w_2$ , such that, by letting  $n_1 = \tau(i_1)$  and  $n_2 = \tau(i_2)$ ,  $n_1 \in t_f(w_1)$  iff  $n_2 \in t_f(w_2)$ . If*

1.  $(t_f(w_1), n_1) \equiv_k (t_f(w_2), n_2)$  if  $n_1 \in t_f(w_1)$ ;  $(t_f(w_1), r_\infty^1) \equiv_k (t_f(w_2), r_\infty^2)$  otherwise;
2.  $\overleftarrow{v}_k^L(w_1, n_1) \equiv_k \overleftarrow{v}_k^L(w_2, n_2)$ ;
3.  $\overrightarrow{v}_k^L(w_1, n_1) \equiv_k \overrightarrow{v}_k^L(w_2, n_2)$ ;
4.  $n_1 \notin t_f(w_1)$  implies  $t^L(w_1, \nu(n_1)) \equiv_k t^L(w_2, \nu(n_2))$

then  $(T_{w_1}, n_1) \equiv_k (T_{w_2}, n_2)$ .

*Proof.* The proof is carried out by means of a standard composition argument. We give a winning strategy for  $\exists$  in the  $k$ -round EF game between  $(T_{w_1}, n_1)$  and  $(T_{w_2}, n_2)$ . Suppose w.l.o.g. that  $\forall$  picks a node  $n^\forall$  from  $T_{w_1}$ .  $n^\forall$  may be part of, either,  $t_f(w_1)$  or a subtree  $t^L(w_1, \nu(n^\forall))$ . In the former case,  $\exists$  answers with a node in  $t_f(w_2)$ , according to her winning strategy in game 1. For the latter case, let us first suppose  $n_1$  and  $n_2$  are not in, resp.,  $t_f(w_1)$  and  $t_f(w_2)$ . Then,  $\exists$  proceeds as follows:

- (a)  $\nu(n^\forall) < \nu(n_1)$ . In this case,  $\exists$  plays her game 2 as if  $\forall$  had picked position  $\nu(n^\forall)$  in  $\overleftarrow{v}_k^L(w_1, n_1)$ . Let  $q^\exists$  be the position in  $\overleftarrow{v}_k^L(w_2, n_2)$  chosen according to such strategy. Since  $\overleftarrow{v}_k^L(w_1, n_1) \equiv_k \overleftarrow{v}_k^L(w_2, n_2)$  and  $k \geq 1$ ,  $\nu(n^\forall)$  and  $q^\exists$  must have the same label, so we have  $\sigma_k^L(w_1, \nu(n^\forall)) = \sigma_k^L(w_2, q^\exists)$ . So,  $t^L(w_1, \nu(n^\forall)) \equiv_k t^L(w_2, q^\exists)$ , and  $\exists$  may pick a node  $n^\exists$  in  $t^L(w_2, q^\exists)$  according to her winning strategy for the  $k$ -round game on  $t^L(w_1, \nu(n^\forall))$  and  $t^L(w_2, q^\exists)$ , considering  $n^\forall$  as  $\forall$ 's move.
- (b)  $\nu(n^\forall) = \nu(n_1)$ . In this case,  $\exists$  picks  $n^\exists$  in  $T_{w_2}$  according to her winning strategy on the EF game for equivalence 4.
- (c)  $\nu(n^\forall) > \nu(n_1)$ .  $\exists$  may proceed as in case (a), but picking  $q^\exists$  from  $\overrightarrow{v}_k^L(w_2, n_2)$  according to her winning strategy on game 3.

If, instead,  $n_1$  and  $n_2$  are in  $t_f(w_1)$  and  $t_f(w_2)$ , then word  $\overrightarrow{v}_k^L(w_1, n_1)$  (resp.  $\overrightarrow{v}_k^L(w_2, n_2)$ ) represents all of the infinite siblings in  $T_{w_1}$  (resp.  $T_{w_2}$ ), and  $\overleftarrow{v}_k^L(w_1, n_1)$  and  $\overleftarrow{v}_k^L(w_2, n_2)$  are the empty word. So,  $\exists$  may proceed as in case (a), but picking  $q^\exists$  from  $\overrightarrow{v}_k^L(w_2, n_2)$  according to her winning strategy on game 3.  $\square$

We now show how to express a FO formula with one free variable with a  $\mathcal{X}_{\text{until}}$  formula equivalent to it on a LR UOT. As in the RR case, we show how to represent in  $\mathcal{X}_{\text{until}}$  the rank- $k$  types of all parts in which we divide the tree in Lemma 9.19. Let  $n$  be the node in which the  $\mathcal{X}_{\text{until}}$  formula is evaluated. We need to distinguish whether  $n \in t_f(w)$  or not. This is more conveniently done while also discerning such cases from the RR one. Thus, we now treat the two LR cases separately, and show how to combine them with the RR case in Section 9.2.5.

Suppose  $n \notin t_f(w)$ . Let  $s_n = s^L(w, \nu(n))$  be the root of the subtree  $t_n = t^L(w, \nu(n))$  containing  $n$ . Children of  $r_\infty$  can be identified by the  $\mathcal{X}_{\text{until}}$  formula  $\alpha_\infty^L := \bigcirc \Rightarrow \top \wedge \neg \Rightarrow (\top, \neg \bigcirc \Rightarrow \top)$ , saying that no right sibling without right siblings is reachable from the current node (i.e. there exists no rightmost sibling). By Lemma 9.17 with  $\alpha^* = \alpha_\infty^L$ , there exists a  $\mathcal{X}_{\text{until}}$  formula  $\beta(t_n)$  that fully identifies the rank- $k$  type of  $t_n$  if evaluated in  $s_n$ .

Moreover,  $\overleftarrow{v}_k^L(w, n)$  and  $\overrightarrow{v}_k^L(w, n)$  can be expressed similarly to  $\overleftarrow{v}_k^R(w, n)$  and  $\overrightarrow{v}_k^R(w, n)$  for RR UOTs. By Kamp's Theorem, there exists a future LTL formula  $\overrightarrow{\psi}_k(w, n)$  that, evaluated in the first position of  $\overrightarrow{v}_k^L(w, n)$ , fully identifies its rank- $k$  type. Recursively substitute  $\varphi \mathcal{U} \varphi'$  subformulas with  $\Rightarrow (\varphi, \varphi')$  in  $\overrightarrow{\psi}_k(w, n)$ , obtaining  $\overrightarrow{\psi}'_k(w, n)$ . Then, replace all rank- $k$  types of UOTs in  $\overrightarrow{\psi}'_k(w, n)$  with the respective formulas obtained from Lemma 9.17, with  $\alpha^* = \alpha_\infty^L$ , thus obtaining  $\overrightarrow{\psi}''_k(w, n)$ . Now,  $\bigcirc \Rightarrow \overrightarrow{\psi}''_k(w, n)$ , evaluated in  $s_n$ , fully describes the rank- $k$  type of all right siblings of  $s_n$ , and of the subtrees rooted in them.

Formula  $\bigcirc \Leftarrow \overleftarrow{\psi}''_k(w, n)$ , which describes left siblings of  $s_n$  if evaluated in it, can be obtained symmetrically, but replacing  $\mathcal{S}$  with  $\Leftarrow$  in the LTL formula.

Finally, we need to describe the rank- $k$  type of  $t_f(w)$ . By Marx's Theorem, there exists a formula  $\psi_{f1}$  that, evaluated in  $r_\infty$ , describes the rank- $k$  type of  $t_f(w)$ . Take the conjunction of each subformula of  $\psi_{f1}$  with  $\neg \alpha_\infty^L$ , and call the obtained formula  $\psi'_{f1}$ . Thus, the rank- $k$  type of  $t_f(w)$  is described by  $\bigcirc \uparrow \psi'_{f1}$ , evaluated in  $s_n$ . Finally, the rank- $k$  type of the whole tree is described by the following formula, evaluated in  $n$ :

$$\varphi_{L1} := (\alpha_\infty^L \wedge \varphi'_{L1}) \vee \uparrow (\neg \alpha_\infty^L, \alpha_\infty^L \wedge \varphi'_{L1}),$$

where the  $\uparrow$  operator is needed to reach  $s_n$  from  $n$  if  $s_n \neq n$ , and

$$\varphi'_{L1} := \circ_{\uparrow} \psi'_{f1} \wedge \circ_{\leftarrow} \overleftarrow{\psi}''_k(w, n) \wedge \circ_{\Rightarrow} \overrightarrow{\psi}''_k(w, n) \wedge \beta(t_n).$$

Suppose, instead,  $n \in t_f(w)$ . Then, we express the rank- $k$  type of  $t_f(w)$  by means of a  $\mathcal{X}_{until}$  formula  $\psi_{f2}$ , evaluated in  $n$ , which exists by Marx's Theorem. In it, we recursively take the conjunction of subformulas with  $\neg\alpha_{\infty}^L$ , thus obtaining  $\psi'_{f2}$ .

Next, we need to describe the rank- $k$  type of word  $\overrightarrow{v}^L_k(w, s^L(w, 0))$  (recall that  $\overleftarrow{v}^L_k(w, s^L(w, 0))$  is the empty word). This can be done as in case  $n \notin t_f(w)$ , thus obtaining formula  $\overrightarrow{\psi}''_k(w, s^L(w, 0))$  which, evaluated in  $s^L(w, 0)$ , fully identifies the children of  $r_{\infty}$ . The latter can be identified by formula  $\circ_{\Downarrow} \alpha_{\infty}^L$ . Thus, to describe its children from  $n$ , we use formula

$$\varphi'_{L2} := \uparrow \left( \top, \neg \circ_{\uparrow} \top \wedge \downarrow \left( \neg \circ_{\Rightarrow} \top, \circ_{\Downarrow} (\alpha_{\infty}^L \wedge \neg \circ_{\leftarrow} \top \wedge \overrightarrow{\psi}''_k(w, s^L(w, 0))) \right) \right),$$

where the outermost  $\uparrow$  reaches the root of  $T_w$  (identified by  $\neg \circ_{\uparrow} \top$ ), the inner  $\downarrow$  reaches  $r_{\infty}$  (identified by  $\circ_{\Downarrow} \alpha_{\infty}^L$ ) by descending through rightmost children ( $r_{\infty}$  is the left context of an open chain, hence a pending position), and  $\neg \circ_{\leftarrow} \top$  identifies  $s^L(w, 0)$ . The final formula is

$$\varphi_{L2} := \varphi'_{L2} \wedge \psi'_{f2}.$$

This concludes the proof of Lemma 9.18.  $\square$

## 9.2.5 Synthesis

To finish the proof, we must combine the previous cases to obtain a single  $\mathcal{X}_{until}$  formula. First, note that it is possible to discern the type of UOT by means of FO formulas of quantifier rank at most 5. The following formula identifies RR UOTs:

$$\gamma_R := \forall x [(0R_{\Downarrow}^* x \wedge \forall y (0R_{\Downarrow}^* y \wedge yR_{\Downarrow}^* x) \implies \mu(y)) \implies \exists y (xR_{\Downarrow} y \wedge \mu(y))],$$

where  $\mu(y) := \neg \exists z (yR_{\Rightarrow} z)$ .  $\gamma_R$  means that any node  $x$  which is on the rightmost branch from the root must have a rightmost child, i.e. the rightmost branch has no end. LR UOTs are identified by the following formulas:

$$\gamma_{L1}(n) := \exists x [xR_{\Downarrow}^+ n \wedge \exists y (xR_{\Downarrow} y) \wedge \forall y (xR_{\Downarrow} y \implies \exists z (yR_{\Rightarrow} z))]$$

$$\gamma_{L2}(n) := \exists x [\neg (xR_{\Downarrow}^+ n) \wedge \exists y (xR_{\Downarrow} y) \wedge \forall y (xR_{\Downarrow} y \implies \exists z (yR_{\Rightarrow} z))]$$

Both  $\gamma_{L1}(n)$  and  $\gamma_{L2}(n)$  say there exists a node  $x = r_{\infty}$  that has infinite children, but  $\gamma_{L2}(n)$  is true iff the free variable  $n$  is in  $t_f(w)$ , while  $\gamma_{L1}(n)$  is true otherwise.

Given a FO formula of quantifier rank  $m$  with one free variable  $\bar{\varphi}(x)$ , let  $k = \max(m, 5)$ . We use Lemma 5.8 to express  $\bar{\varphi}(x)$ . Consider the finite set

$$\Gamma_k = \{\sigma_k(T_w, n) \mid (T_w, n) \models \bar{\varphi}(x), n \in T_w\}$$

of the rank- $k$  types of OPM-compatible UOTs satisfying  $\bar{\varphi}(x)$ , with  $n$  as a distinguished node. For each one of them, we take the corresponding Hintikka formula  $H^k(w, n)$ . Since  $k \geq 5$ , and the UOT type is distinguishable by formulas of quantifier rank at most 5, for each  $\sigma_k(T_w, n)$  it is possible to tell whether it describes RR or LR UOTs. For each type, by Lemmas 9.15 and 9.18, it is possible to express  $H^k(w, n)$  through formulas describing the rank- $k$  types of the substructures given by the composition arguments, and translate them into  $\mathcal{X}_{until}$  accordingly. Then, the translated formula  $\varphi$  is the XOR of one of the following formulas, for each type  $\sigma_k(T_w, n) \in \Gamma_k$ .

- If  $T_w$  is RR, then  $\xi_R \wedge \varphi_R(\sigma_k(T_w, n))$ .
- If  $T_w$  is LR, and  $n \notin t_f(w)$ :  $\xi_L \wedge (\alpha_\infty^L \vee \uparrow(\top, \alpha_\infty^L)) \wedge \varphi_{L1}(\sigma_k(T_w, n))$ , where we assert one of the infinite siblings is reachable going upwards from  $n$ , and so  $n$  is not in  $t_f(w)$ .
- If  $T_w$  is LR, and  $n \in t_f(w)$ :  $\xi_L \wedge \neg(\alpha_\infty^L \vee \uparrow(\top, \alpha_\infty^L)) \wedge \varphi_{L2}(\sigma_k(T_w, n))$ .

In the above formulas,  $\varphi_R(\sigma_k(T_w, n))$ ,  $\varphi_{L1}(\sigma_k(T_w, n))$ , and  $\varphi_{L2}(\sigma_k(T_w, n))$  are the formulas expressing  $\sigma_k(T_w, n)$ , obtained as described in the previous paragraphs. Moreover, we have

$$\begin{aligned}\xi_R &:= \uparrow(\top, \neg \circ_{\uparrow} \top \wedge \neg \downarrow(\neg \circ_{\Rightarrow} \top, \neg \circ_{\Rightarrow} \top \wedge \neg \circ_{\downarrow} \top)), \\ \xi_L &:= \uparrow(\top, \neg \circ_{\uparrow} \top \wedge \downarrow(\neg \circ_{\Rightarrow} \top, \neg \circ_{\Rightarrow} \top \wedge \circ_{\downarrow} \alpha_\infty^L)).\end{aligned}$$

$\xi_R$  identifies RR UOTs. In it, the outermost  $\uparrow$  reaches the root of the tree, and the inner  $\downarrow$  imposes that the rightmost branch has no end.  $\xi_L$  identifies LR UOTs. The outermost  $\uparrow$  also reaches the root of the tree, and the inner  $\downarrow$  is verified by a path in which all nodes are on the rightmost branch, except the last one, which is one of the infinite siblings.

Boolean queries can be expressed as Boolean combinations of formulas of the form  $\bar{\varphi} := \exists x(\varphi'(x))$ . Then, it is possible to translate  $\bar{\varphi}'(x)$  as above, to obtain  $\mathcal{X}_{\text{until}}$  formula  $\varphi'$ , and  $\downarrow(\top, \varphi')$  is such that  $(T_w, 0) \models \downarrow(\top, \varphi')$  iff  $T_w \models \bar{\varphi}$ , for any OPM-compatible UOT  $T_w$ .

Thus, we can state

**Theorem 9.20.**  $\mathcal{X}_{\text{until}} = \text{FOL}$  with one free variable on OPM-compatible  $\omega$ -UOTs.

Thanks to Theorem 9.20, we can extend the results entailed by the translation of Section 9.1.2.2 to  $\omega$ -words.

**Theorem 9.21.**  $\text{POTL} = \text{FOL}$  with one free variable on OP  $\omega$ -words.

**Corollary 9.22.** The propositional operators plus  $\circ^d, \ominus^d, \chi_F^d, \chi_P^d, \mathcal{U}_\chi^d, \mathcal{S}_\chi^d, \circ_H^u, \ominus_H^u, \mathcal{U}_H^u, \mathcal{S}_H^u$  are expressively complete on OP  $\omega$ -words.

The containment relations in Corollary 9.11 are easily extended to  $\omega$ -words:

**Corollary 9.23.**  $\text{NWTL} \subset \text{OPTL} \subset \text{POTL}$  over OP  $\omega$ -words.

Moreover,

**Corollary 9.24.** Every FO formula with at most one free variable is equivalent to one using at most three distinct variables on OP  $\omega$ -words.



**Part III**

**Model Checking**



In this part, we study model checking for OPLs. In Chapter 10, we describe the model checking procedure for POTL, which we choose instead of that for OPTL because of the greater expressiveness of the former. In Chapter 11, we report on its implementation and on the experimental results of its evaluation.

The model checking construction for POTL and its implementation were presented in [56], and partially in [140].



# Chapter 10

## Model Checking Construction

The model checking procedure we give for POTL follows the automata-theoretic approach we presented for LTL in Chapter 2, adapting it to work with OPA for the finite-word case, and  $\omega$ OPBA for the infinite-word case. Thus, we define a construction for automata that accept models of an arbitrary POTL formula, and prove its correctness. This construction is significantly more involved than the one for LTL, although the final size of the automaton is still singly exponential in formula length.

We first treat the finite-word construction in Section 10.1, and then we adapt it to  $\omega$ -words in Section 10.2.

### 10.1 Finite-Word Model Checking

We present an automata-theoretic model checking procedure for POTL based on OPA. Given an OP alphabet  $(\mathcal{P}(AP), M_{AP})$ , where  $AP$  is a finite set of atomic propositions, and a formula  $\varphi$ , let  $\mathcal{A}_\varphi = \langle \mathcal{P}(AP), M_{AP}, Q, I, F, \delta \rangle$  be an OPA. The construction of  $\mathcal{A}_\varphi$  resembles the classical one for LTL and the ones for NWTL and OPTL, diverging from them significantly when dealing with temporal obligations involving positions in the  $\chi$  relation.

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

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

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

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

The consistency constraints on  $\text{Atoms}(\varphi)$  will be augmented incrementally in the following, for each operator.

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

### 10.1.1 Next/Back Operators

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

### 10.1.2 Chain Next Operators

Operators  $\chi_F^\pi, \chi_P^\pi$ , with  $\pi \in \{<, \doteq\}$ , restrict their downward counterparts to a single PR. Their semantics can be defined as follows: given an OP word  $w$  and a position  $i$ , we have  $(w, i) \models \chi_F^\pi \psi$  iff there exists a position  $j > i$  such that  $\chi(i, j)$  and  $i \pi j$ , and  $(w, j) \models \psi$ . Since they are needed for model-checking hierarchical operators, we include them in the construction. We also use them to model check downward/upward chain next and back operators.

If  $\chi_F^d \psi \in \text{Cl}(\varphi)$ , we add  $\chi_F^< \psi, \chi_F^{\doteq} \psi \in \text{Cl}(\varphi)$ , and for each  $\Phi \in \text{Atoms}(\varphi)^2$  we impose that  $\chi_F^d \psi \in \Phi_c$ , iff  $\chi_F^< \psi \in \Phi_c$  or  $\chi_F^{\doteq} \psi \in \Phi_c$ . To model check  $\chi_F^u \psi$ , we add the consistency constraint that, for any  $\Phi \in \text{Atoms}(\varphi)$ ,  $\chi_F^u \psi \in \Phi_c$  iff either  $\chi_F^{\doteq} \psi \in \Phi_c$ ,  $\chi_F^> \psi \in \Phi_c$ , or both.

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

If  $\chi_F^{\doteq} \psi \in \text{Cl}(\varphi)$ , its satisfaction is ensured by the following constraints on  $\delta$ .

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

	input	state	stack	PR	move
1	<b>call han exc ret #</b>	$\Phi_c^0 = \{\text{call}, \chi_F^d \text{ret}, \chi_F^{\dot{\dot{c}}} \text{ret}\},$ $\Phi_p^0 = \{\chi_L\}$	$\perp$	$\# < \text{call}$	push
2	<b>han exc ret #</b>	$\Phi^1 = (\{\text{han}\}, \{\chi_F^{\dot{\dot{c}}} \text{ret}, \chi_L\})$	$[\text{call}, \Phi^0] \perp$	$\text{call} < \text{han}$	push
3	<b>exc ret #</b>	$\Phi^2 = (\{\text{exc}\}, \emptyset)$	$[\text{han}, \Phi^1] [\text{call}, \Phi^0] \perp$	$\text{han} \dot{=} \text{exc}$	shift
4	<b>ret #</b>	$\Phi^3 = (\{\text{ret}\}, \emptyset)$	$[\text{exc}, \Phi^1] [\text{call}, \Phi^0] \perp$	$\text{exc} > \text{ret}$	pop
5	<b>ret #</b>	$\Phi^4 = (\{\text{ret}\}, \{\chi_F^{\dot{\dot{c}}} \text{ret}\})$	$[\text{call}, \Phi^0] \perp$	$\text{call} \dot{=} \text{ret}$	shift
6	<b>#</b>	$\Phi^5 = (\{\#\}, \emptyset)$	$[\text{ret}, \Phi^0] \perp$	$\text{ret} > \#$	pop
7	<b>#</b>	$\Phi^5 = (\{\#\}, \emptyset)$	$\perp$	-	-

Figure 10.1: Example accepting run of the automaton for  $\chi_F^d \text{ret}$ .

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

If  $\chi_F^{\dot{\dot{c}}} \psi \in \text{Cl}(\varphi)$ ,  $\chi_F^{\dot{\dot{c}}} \psi$  is allowed in the pending part of initial states, and we add the following constraints.

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

The rules for  $\chi_F^{\dot{\dot{c}}} \psi$  only differ in  $\psi$  being enforced by a pop transition, triggered by the  $>$  relation between the left and right contexts of the chain on which  $\chi_F^{\dot{\dot{c}}} \psi$  holds. Thus, if  $\chi_F^{\dot{\dot{c}}} \psi \in \text{Cl}(\varphi)$  we have:

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

We illustrate how the construction works for  $\chi_F^{\dot{\dot{c}}}$  with the example of Figure 10.1. The OPA starts in state  $\Phi^0$ , with  $\chi_F^d \text{ret} \in \Phi_c^0$ , and guesses that  $\chi_F^d$  will be fulfilled by  $\chi_F^{\dot{\dot{c}}}$ , so  $\chi_F^{\dot{\dot{c}}} \text{ret} \in \Phi_c^0$ . **call** is read by a push move, resulting in state  $\Phi^1$ . The OPA guesses the next move will be a push, so  $\chi_L \in \Phi_p^1$ . By rule 1, we have  $\chi_F^{\dot{\dot{c}}} \text{ret} \in \Phi_p^1$ . The last guess is immediately verified by the next push (step 2-3). Thus, the pending obligation for  $\chi_F^{\dot{\dot{c}}} \text{ret}$  is stored onto the stack in  $\Phi^1$ . The OPA, then, reads **exc** with a shift, and pops the stack symbol containing  $\Phi^1$  (step 4-5). By rule 2, the temporal obligation is resumed in the next state  $\Phi^4$ , so  $\chi_F^{\dot{\dot{c}}} \text{ret} \in \Phi_p^4$ . Finally, **ret** is read by a shift which, by rule 3, may occur only if **ret**  $\in \Phi_c^4$ . Rule 3 verifies the guess that  $\chi_F^{\dot{\dot{c}}} \text{ret}$  holds in  $\Phi_0$ , and fulfills the temporal obligation contained in  $\Phi_p^4$ , by preventing computations in which **ret**  $\notin \Phi_c^4$  from continuing. Had the next transition been a pop (e.g. because there was no **ret** and **call**  $>$  **#**), the run would have been blocked by rule 2, preventing the OPA from reaching an accepting state, and from emptying the stack.

We now prove the correctness of this construction. For each operator, we show that in all accepting computations it appears in an OPA state iff it holds in the corresponding word position. While doing so, we assume that the construction is correct for the operands of each operator. This allows us to prove the correctness of the whole

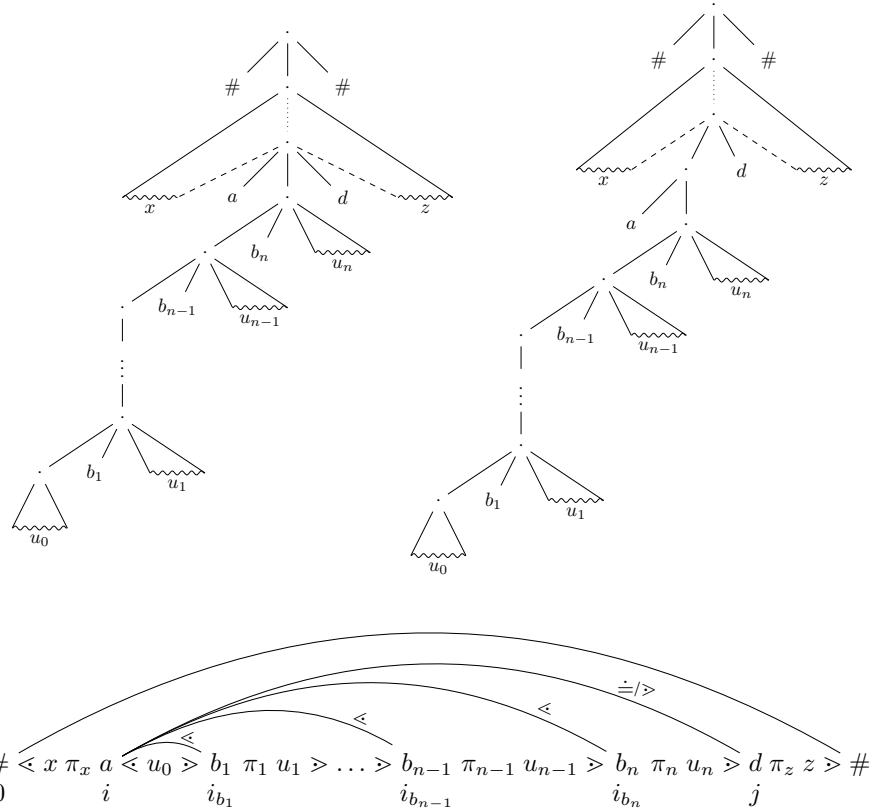


Figure 10.2: The two possible STs of a generic OP word  $w = xyz$  (top), and its flat representation with chains (bottom). Wavy lines are placeholders for subtree frontiers. We have either  $a \doteq d$  (top left), or  $a > d$  (top right). In both trees,  $a \ll b_k$  for  $1 \leq k \leq n$ , and the corresponding word positions are in the chain relation. For  $1 \leq k \leq n$ ,  $u_k$  is the word generated by the right part of the rhs whose first terminal is  $b_k$ . So, either  ${}^{b_k}[u_k]^{b_{k+1}}$ , or  $u_k$  is of the form  $v_0^k c_0^k v_1^k c_1^k \dots c_{m_k}^k v_{m_k+1}^k$ , where  $c_p^k \doteq c_{p+1}^k$  for  $0 \leq p < m_k$ ,  $b_k \doteq c_0^k$ , and resp.  $c_{m_k}^k \gg b_{k+1}$  and  $c_{m_n}^n \gg d$  (cf. Figure 10.3). Moreover, for each  $0 \leq p < m_k$ , either  $v_{p+1}^k = \varepsilon$  or  $c_p^k [v_{p+1}^k]^{c_{p+1}^k}$ ; either  $v_0^k = \varepsilon$  or  ${}^{b_k}[v_0^k]^{c_0^k}$ , and either  $v_{m_k+1}^k = \varepsilon$  or  $c_{m_k}^k [v_{m_k+1}^k]^{b_{k+1}}$  (resp.  $c_{m_n}^n [v_{m_n+1}^n]^d$ ).  $u_0$  has this latter form, except  $v_0^0 = \varepsilon$  and  $a \ll c_0^0$ . In the bottom representation, the  $\pi_k$ s are placeholders for precedence relations, that depend on the surrounding characters.

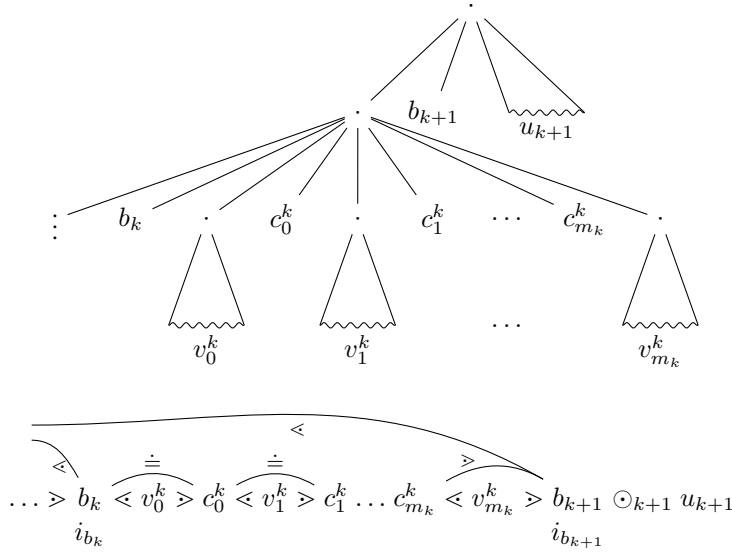


Figure 10.3: The structure of  $u_k$  in the word of Figure 10.2.

construction inductively on the formula's structure, in Section 10.1.6. In the following, we denote as  $\text{first}(w)$  the first position of a word  $w$ . We also use Figure 10.3, which represents the generic structure of any composed chain.

**Lemma 10.1.** *Given a finite set of atomic propositions  $AP$ , an OP alphabet  $(\mathcal{P}(AP), M_{AP})$ , a word  $w = \#xyz\#$  on it, and a position  $i = |x| + 1$  in  $w$ , we have*

$$(w, i) \models \chi_F^{\dot{}} \psi$$

if and only if all accepting computations of an OPA satisfying rules 1-3 bring it from a configuration  $\langle yz, \Phi, \alpha \gamma \rangle$  with  $\chi_F^{\dot{}} \psi \in \Phi_c$  to a configuration  $\langle z, \Phi', \alpha' \gamma \rangle$  such that  $\chi_F^{\dot{}} \psi \notin \Phi'_p$ ,  $|\alpha| = 1$  and  $|\alpha'| = 1$  if  $\text{first}(y)$  is read by a shift move,  $|\alpha'| = 2$  if it is read by a push move. If  $\chi_F^{\dot{}} \psi \notin \Phi_p$  and it is not in the pending part of the state in  $\alpha$ , then it is not in the pending parts of states in  $\alpha'$ . If no other rules constrain the transition relation, at least one computation is accepting.

*Proof.* In the following, we denote by  $\Phi^a$  the state of the automaton before reading symbol  $a$ , so  $\Phi^a \cap AP = a$ , for any  $a \subseteq AP$ .

[ $\Rightarrow$ ] Suppose  $\chi_F^{\dot{}} \psi$  holds in position  $i$ , corresponding to terminal symbol  $a$ . In all accepting computations, the OPA reaches configuration  $\langle a \dots z, \Phi^a, [f, \Phi^f] \gamma \rangle$ , where  $\alpha = [f, \Phi^f]$ , and guesses that  $\chi_F^{\dot{}} \psi$  holds in  $i$ , so  $\chi_F^{\dot{}} \psi \in \Phi_c^a$ . We show later in the proof that all accepting computations must make this guess.  $a$  is read by either a push or a shift transition, leading the OPA to configuration  $\langle c_0^0 \dots z, \Phi^{c_0^0}, \delta \rangle$ , with either  $\delta = [a, \Phi^a][f, \Phi^f] \gamma$  or  $\delta = [a, \Phi^f] \gamma$ , respectively. Moreover,  $\chi_F^{\dot{}} \psi \in \Phi_p^{c_0^0}$  and  $\chi_L \in \Phi_p^{c_0^0}$  due to rule (1). Since  $\chi_F^{\dot{}} \psi$  holds in  $i$ ,  $a$  is the left context of a chain, so the next transition is a push, satisfying the requirement for  $\chi_L$ . This also means  $w$  has the form of Figure 10.2, possibly with  $n = 0$  (cf. the caption for notation). Any accepting computation must go through the support for this chain. The next configuration is

$\langle v_1^0 \dots z, \Phi^{v_1^0}, [c_0^0, \Phi^{c_0^0}] \delta \rangle$ , with  $\chi_{\bar{F}}^{\dot{\psi}} \psi \in \Phi_p^{c_0^0}$ . Then, the computation goes on normally. Note that, when reading an inner chain body such as  $v_1^0$ , the automaton does not touch the stack symbol containing  $\Phi^{c_0^0}$ , and other symbols in the body of the same simple chain, i.e.  $c_1^0, c_2^0 \dots$ , are read with shift moves that update the topmost stack symbol with the new terminal, leaving state  $\Phi^{c_0^0}$  untouched.

If  $a$  is the left context of more than one chain (i.e.  $n > 0$  in the figure), the OPA then reaches configuration  $\langle b_1 \dots z, \Phi^{b_1}, [c_{m_0}^0, \Phi^{c_0^0}] \delta \rangle$ . Since  $c_{m_0}^0 \succ b_1$ , the next transition is a pop.  $\chi_{\bar{F}}^{\dot{\psi}} \psi \in \Phi^{c_0^0}$ , so by rule (2), the automaton reaches configuration  $\langle b_1 \dots z, \Phi^{b_1}, \delta \rangle$  with  $\chi_{\bar{F}}^{\dot{\psi}} \psi \in \Phi_p^{b_1}$ . Then, since  $a$  is contained in the topmost stack symbol and  $a < b_1$ , the next move is a push, leading to  $\langle v_0^1 \dots z, \Phi^{v_0^1}, [b_1, \Phi^{b_1}] \delta \rangle$ . Notice how  $\chi_{\bar{F}}^{\dot{\psi}} \psi$  is again stored as a pending obligation in the topmost stack symbol. The OPA run goes on in the same way for each terminal  $b_p$ ,  $1 \leq p \leq n$ , until the automaton reaches configuration  $\langle d \dots z, \Phi^d, [c_{m_n}^n, \Phi^{b_n}] \delta \rangle$  with  $\chi_{\bar{F}}^{\dot{\psi}} \psi \in \Phi_p^{b_n}$ . If  $a$  was the left context of one chain only, this is the configuration reached after reading the body of such chain, with  $n = 0$ . Since  $c_{m_n}^n \succ d$ , a pop transition leads to  $\langle d \dots z, \Phi^d, \delta \rangle$ , with  $\chi_{\bar{F}}^{\dot{\psi}} \psi \in \Phi_p^d$ , by rule (2). Note that there exists a computation in which  $\chi_{\bar{F}}^{\dot{\psi}} \psi \notin \Phi_p^d$ , so rule (2) applies. The fact that a computation with  $\chi_{\bar{F}}^{\dot{\psi}} \psi \notin \Phi_p^d$  is blocked by rule (2) is correct, because this implies  $\chi_{\bar{F}}^{\dot{\psi}} \psi$  holds in the position preceding  $d$ . This would be wrong, because such a position is in the  $\succ$  relation with  $d$ , and it cannot be the left context of a chain, so  $\chi_{\bar{F}}^{\dot{\psi}} \psi$  must be false in it. Then, if  $\chi_{\bar{F}}^{\dot{\psi}} \psi$  holds in  $i$ , since  $a$  is the terminal in the topmost stack symbol, we must have  $a \dot{=} d$ . So  $d$  is read by a shift move, leading to  $\langle z, \Phi^z, \alpha' \gamma \rangle$  with  $\alpha' = [d, \Phi^d][f, \Phi^f]$  or  $\alpha' = [d, \Phi^d]$ , depending on which kind of move previously read  $a$ . Note that if  $\chi_{\bar{F}}^{\dot{\psi}} \psi \notin \Phi_p^a, \Phi_p^f$ , the claim about states in  $\alpha'$  is satisfied. Since  $\chi_{\bar{F}}^{\dot{\psi}} \psi$  holds in  $i$ ,  $\psi$  holds in  $j$  (the position corresponding to  $d$ ), and  $\psi \in \Phi_c^d$ , because we assume the correctness of the construction for all other operators. This satisfies rule (3), and verifies the initial guess that  $\chi_{\bar{F}}^{\dot{\psi}} \psi$  holds in  $i$ . By rule (3), any computation in which  $\psi$  holds in  $j$  must have  $\chi_{\bar{F}}^{\dot{\psi}} \psi \in \Phi_p^d$ , which is only the case if the OPA makes such initial guess. Finally, there exists a computation in which  $\chi_{\bar{F}}^{\dot{\psi}} \psi \notin \Phi^z$ , satisfying the thesis statement. Note that all computations of this form may then proceed normally until acceptance, if they are not blocked by rules other than 1-3.

[ $\Leftarrow$ ] Suppose that an accepting computation starts from configuration  $\langle a \dots z, \Phi^a, [f, \Phi^f] \gamma \rangle$ , with  $\chi_{\bar{F}}^{\dot{\psi}} \psi \in \Phi_c^a$ ,  $\alpha = [f, \Phi^f]$ , and  $f < a$  (the case  $f \dot{=} a$  is analogous).  $a$  is read by a push move in this case, which leads the OPA to configuration  $\langle c_0^0 \dots z, \Phi^{c_0^0}, [a, \Phi^a][f, \Phi^f] \gamma \rangle$ , with  $\chi_{\bar{F}}^{\dot{\psi}} \psi, \chi_L \in \Phi_p^{c_0^0}$ . Since  $\chi_L \in \Phi_p^{c_0^0}$ , the next transition must be a push, so  $a < c_0^0$ ,  $a$  is the left context of a chain and  $w$  is of the form of Figure 10.2. The push move brings the OPA to configuration  $\langle v_0^0 \dots z, \Phi^{v_0^0}, [c_0^0, \Phi^{c_0^0}][a, \Phi^a][f, \Phi^f] \gamma \rangle$ . Notice that the stack size is now  $|\gamma| + 3$ . By the thesis, the automaton eventually reaches a configuration in which the stack size is  $|\gamma| + 2$ . This can be achieved if  $[c_0^0, \Phi^{c_0^0}]$  is popped, so  $\alpha' = [a, \Phi^a][f, \Phi^f]$ . In a generic word such as the one of Figure 10.2, this happens only before reading  $b_p$ ,  $1 \leq i \leq n$ , or  $d$ .

In both cases, let  $[c_{m_k}^k, \Phi^{b_k}]$  be the popped stack symbol. We have  $\chi_{\bar{F}}^{\dot{\psi}} \psi \in \Phi_p^{b_k}$ . By rule (2), if  $\Phi'$  is the destination state of the pop move,  $\chi_{\bar{F}}^{\dot{\psi}} \psi \in \Phi'_p$ , which does not satisfy the thesis statement. If the next move is a push (such as when reading any  $b_p$ ,  $1 \leq p \leq n$ ), the stack length increases again, which also does not satisfy the thesis. If the next move is a pop, rule (2) blocks the computation. So, the next move must be

a shift, updating symbol  $[a, \Phi^a]$  to  $[d, \Phi^a]$ , where  $d$  is the just-read terminal symbol. This means the OPA reached the right context of the chain whose left context is  $i$  (i.e.  $a$ ), and the two positions are in the  $\dot{=}$  relation. By rule (3),  $\psi$  is part of the starting state of this move, so  $\psi$  holds in this position, satisfying  $\chi_F^{\dot{=}} \psi$  in  $i$ . The state resulting from the shift move may not contain  $\chi_F^{\dot{=}} \psi$  as a pending obligation, thus satisfying the thesis.  $\square$

The proof for  $\chi_F^{\dot{>}}$  is very similar to Lemma 10.1, and is therefore omitted.

**Lemma 10.2.** *Given a finite set of atomic propositions  $AP$ , an OP alphabet  $(\mathcal{P}(AP), M_{AP})$ , a word  $w = \#xyz\#$  on it, and a position  $i = |x| + 1$  in  $w$ , we have*

$$(w, i) \models \chi_F^{\leq} \psi$$

*if and only if all accepting computations of an OPA satisfying rules 4-5 bring it from configuration  $\langle yz, \Phi, \alpha \gamma \rangle$  with  $\chi_F^{\leq} \psi \in \Phi_c$  to a configuration  $\langle z, \Phi', \alpha' \gamma \rangle$  such that  $\chi_F^{\leq} \psi \notin \Phi'_p$ ,  $|\alpha| = 1$  and  $|\alpha'| = 1$  if  $\text{first}(y)$  is read by a shift move,  $|\alpha'| = 2$  if it is read by a push move. If  $\chi_F^{\leq} \psi \notin \Phi_p$  and it is not in the pending part of the state in  $\alpha$ , then it is not in the pending parts of states in  $\alpha'$ . If no other rules constrain the transition relation, at least one computation is accepting.*

*Proof.*  $[\Rightarrow]$  Suppose  $\chi_F^{\leq} \psi$  holds in position  $i$ , corresponding to terminal  $a$ . Then,  $a$  must be the left context of more than one chain (by property 4 of the  $\chi$  relation), and the word being read is of the form of Figure 10.2, with  $n \geq 1$ . Let us call  $b_p$ ,  $1 \leq p \leq n$ , the right contexts of those of these chains that are s.t.  $a \leq b_p$  (i.e., all except the rightmost context of  $i$ ). There exists an index  $q$ ,  $1 \leq q \leq n$ , such that  $\psi$  holds in  $i_{b_q}$ , the word position labeled with  $b_q$ . All accepting computations reach a configuration  $\langle a \dots z, \Phi^a, [f, \Phi^f] \gamma \rangle$ , where  $\alpha = [f, \Phi^f]$ , and  $\chi_F^{\leq} \psi \in \Phi_c^a$ , because the OPA guesses that  $\chi_F^{\leq} \psi$  holds in  $i$ .  $a$  is read by a shift or a push transition, which leads the OPA to configuration  $\langle c_0^0 \dots z, \Phi^{c_0^0}, \delta \rangle$ , with  $\delta = \alpha' \gamma$ , and either  $\alpha' = [a, \Phi^a][f, \Phi^f]$  or  $\alpha' = [a, \Phi^f]$ , respectively. The claim on the pending part of states in  $\alpha'$  is trivially satisfied. Due to rule (4), we have  $\chi_F^{\leq} \psi \in \Phi_p^{c_0^0}$  and  $\chi_L \in \Phi_p^{c_0^0}$ . As a result, the next move must be a push, consistently with the hypothesis implying  $a$  is the left context of a chain. Then, starting with  $c_0^0$ , the OPA reads the body of the innermost chain whose left context is  $a$ , until it reaches its right context  $b_1$ . In this process, the topmost stack symbol  $[c_0^0, \Phi^{c_0^0}]$  may be updated by shift transitions reading other terminals  $c_p^0$ ,  $1 \leq p \leq m_0$ , that are part of the same simple chain as  $c_0^0$ . However, it is never popped until  $b_1$  is reached, since subchains cause the OPA to only push, pop and update new stack symbols, but not existing ones. So, the OPA reaches configuration  $\langle b_1 \dots z, \Phi^{b_1}, [c_{m_0}^0, \Phi^{c_{m_0}^0}] \delta \rangle$ , with  $\chi_F^{\leq} \psi \in \Phi_p^{c_{m_0}^0}$ .

Suppose  $q \neq 1$ , so  $\psi$  does not hold in  $b_1$ . Since  $c_{m_0}^0 \dot{>} b_1$ , the next transition is a pop. Due to rule (5), it leads the OPA to configuration  $\langle b_1 \dots z, \Phi^{b_1}, \delta \rangle$  with  $\chi_F^{\leq} \psi \in \Phi_p^{b_1}$  and  $\chi_L \in \Phi_p^{b_1}$ . The presence of  $\chi_L$  implies the next move is a push, a requirement that is satisfied because  $a \leq b_1$ . So, the OPA transitions to configuration  $\langle v_0^1 \dots z, \Phi^{v_0^1}, [b_1, \Phi^{b_1}] \delta \rangle$ . The computation, then, goes on in the same way for each  $b_p$ ,  $1 \leq p < q$ . Before  $b_q$  is read, (and possibly  $q = 1$ ), the OPA is in configuration  $\langle b_q \dots z, \Phi^{b_q}, [c_{m_q-1}^{q-1}, \Phi^{c_{m_q-1}^{q-1}}] \delta \rangle$ , with  $\chi_F^{\leq} \psi \in \Phi_p^{b_q}$ . Since  $c_{m_q-1}^{q-1} \dot{>} b_q$ , a pop transition brings the OPA to  $\langle b_q \dots z, \Phi^{b_q}, \delta \rangle$ . Since by hypothesis  $\psi \in \Phi_c^{b_q}$ , by rule (5) we just have  $\chi_L \in \Phi_p^{b_q}$ , and the initial guess is verified. Since the topmost stack symbol contains

$a$ , and  $a \triangleleft b_q$ , the next transition is a push, which satisfies the requirement of  $\chi_L$ . Note that  $\chi_F^{\triangleleft} \psi \notin \Phi_p^{b_q}$ , and the current stack is  $\delta$ , which satisfies the thesis statement, also ensuring that a computation of this form may be finally accepting.

[ $\Leftarrow$ ] Suppose that during an accepting computation the OPA reaches configuration  $\langle a \dots z, \Phi^a, [f, \Phi^f] \gamma \rangle$ , with  $\chi_F^{\triangleleft} \psi \in \Phi_c^a$ . Again,  $a$  must be read by either a push or a shift move. Since  $\chi_L$  is inserted as a pending requirement into the state resulting from this move, the next transition must be a push, so  $a$  is the left context of at least a chain. This chain has the form of Figure 10.2. By rule (4), the OPA reaches configuration  $\langle c_0^0 \dots z, \Phi^{c_0^0}, \delta \rangle$ , with  $\chi_F^{\triangleleft} \psi, \chi_L \in \Phi_p^{c_0^0}$ , and  $\delta$  as in the [ $\Rightarrow$ ] part after reading  $a$ . Let  $[c_0^0, \Phi^{c_0^0}]$  be the stack symbol pushed with  $c_0^0$ . The stack size at this time is greater by one w.r.t. what is required by the thesis statement, so  $[c_0^0, \Phi^{c_0^0}]$  must be popped.

This happens when the OPA reaches a symbol  $e$  s.t. the terminal symbol in the topmost stack symbol takes precedence from  $e$ .  $e$  must be s.t.  $a \triangleleft e$  (and  $e = b_1$  in Figure 10.2). Otherwise, suppose by contradiction that  $a \triangleright e$  or  $a \doteq e$  (so  $e = d$  in Figure 10.2, in which  $n = 0$  and  $c_{m_0}^0$  precedes  $d$ ). In this case, after popping  $[c_{m_0}^0, \Phi^{c_{m_0}^0}]$ , the automaton reaches configuration  $\langle dz, \Phi'^d, \delta' \rangle$ . Since  $\chi_F^{\triangleleft} \psi \in \Phi_p^{c_{m_0}^0}$ , by rule (5) we have  $\chi_F^{\triangleleft} \psi \in \Phi_p'^d$ , so this configuration does not satisfy the thesis statement. Moreover,  $\chi_L \in \Phi_p'^d$ , which requires the next transition to be a push. But  $a \doteq d$  or  $a \triangleright d$ , and  $a$  is the topmost stack symbol, so such a computation is blocked by  $\chi_L$ , never reaching a configuration complying with the thesis statement.

So,  $e = b_1$ , and the OPA reaches configuration  $\langle b_1 \dots z, \Phi^{b_1}, [c_{m_0}^0, \Phi^{c_{m_0}^0}] \delta \rangle$ . The subsequent pop move leads to  $\langle b_1 \dots z, \Phi'^{b_1}, \delta \rangle$ . Suppose  $\psi \in \Phi_c^{b_1}$ . Then, by rule (5) we only have  $\chi_L \in \Phi_p'^{b_1}$ , and  $\chi_F^{\triangleleft} \psi \notin \Phi_p'^{b_1}$ . This configuration satisfies the thesis statement, and since  $a \triangleleft b_1$ ,  $a$  and  $b_1$  are the context of a chain, and  $\psi$  holds in  $b_1$ , we can conclude that  $\chi_F^{\triangleleft} \psi$  holds in  $a$ .

Otherwise, if  $\psi \notin \Phi_c^{b_1}$ , by rule (5) we have  $\chi_F^{\triangleleft} \psi, \chi_L \in \Phi_p'^{b_1}$ . The next transition will therefore push the symbol  $[b_1, \Phi'^{b_1}]$  onto the stack, again with  $\chi_F^{\triangleleft} \psi$  as a pending obligation in it. Then, the same reasoning done with  $[c_0^0, \Phi^{c_0^0}]$  (and its subsequent updates) can be repeated. The only way the thesis statement can be satisfied is by reading a position  $b_q$ , s.t.  $a \triangleleft b_q$ , the terminal in the topmost stack symbol takes precedence from  $b_q$  (so  $a$  and  $b_q$  are the context of a chain), and  $\psi \in \Phi_c^{b_q}$ , so  $\psi$  holds in  $b_q$ . This implies  $\chi_F^{\triangleleft} \psi$  holds in  $a$ .  $\square$

### 10.1.3 Chain Back Operators

We now give the construction for the chain back operators, and their proofs.

To model check the  $\chi_P^d \psi$  and  $\chi_P^u \psi$  operators, we employ the auxiliary operator  $\chi_P^\pi \psi$ , with  $\pi \in \{\triangleleft, \doteq, \triangleright\}$ . Given an OP word  $w$  and a position  $i$  in it, we have  $(w, i) \models \chi_P^\pi \psi$  iff there exists a position  $j < i$  such that  $\chi(j, i)$  and  $j \pi i$ , and  $(w, j) \models \psi$ . For any  $\Phi \in \text{Atoms}(\varphi)$ , we have  $\chi_P^d \psi \in \Phi_c$  iff either  $\chi_P^{\triangleleft} \psi \in \Phi_c$ ,  $\chi_P^{\doteq} \psi \in \Phi_c$ , or both;  $\chi_P^u \psi \in \Phi_c$  iff either  $\chi_P^{\doteq} \psi \in \Phi_c$ ,  $\chi_P^{\triangleright} \psi \in \Phi_c$ , or both.

We add symbol  $\chi_R$ , which lets the computation go on only if the previous transition was a pop, and the position associated with the current state is the right context of a chain. So, for any  $(\Phi, a, \Psi) \in \delta_{push/shift}$ , we have  $\chi_R \notin \Psi_p$ ; for any  $(\Phi, \Theta, \Psi) \in \delta_{pop}$ , we have  $\chi_R \in \Psi_p$ .  $\chi_R$  is allowed in the pending part of final states.

If  $\chi_P^{\doteq} \psi \in \text{Cl}(\varphi)$ , we add the following constraints on the transition relation.

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

The constraints added if  $\chi_{\bar{P}}^{\dot{<}} \psi \in \text{Cl}(\varphi)$  now follow.

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

Finally, when  $\chi_{\bar{P}}^{\dot{>}} \psi \in \text{Cl}(\varphi)$ , we add symbol  $\chi_{\bar{=}}$ , which appears in a state iff the next transition will be a shift: for any  $(\Phi, a, \Psi) \in \delta_{push}$  and  $(\Phi, \Theta, \Psi) \in \delta_{pop}$ ,  $\chi_{\bar{=}} \notin \Phi_p$ , and for any  $(\Phi, a, \Psi) \in \delta_{shift}$ ,  $\chi_{\bar{=}} \in \Phi_p$ .  $\chi_{\bar{P}}^{\dot{>}} \psi$  and  $\chi_{\bar{=}}$  are allowed in the pending part of final states. We also add the constraints below.

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

17.  $\chi_{\bar{P}}^{\dot{>}} \psi \notin \Psi_p$ ;
18.  $\chi_{\bar{P}}^{\dot{>}} \psi \in \Phi_c$  iff  $\chi_{\bar{P}}^{\dot{>}} \psi, \chi_R \in \Phi_p$ ;

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

19. if  $(\chi_L \in \Psi_p \text{ or } \chi_{\bar{=}} \in \Psi_p)$ , then  $\chi_{\bar{P}}^{\dot{>}} \psi \in \Psi_p$  iff  $\chi_{\bar{P}}^{\dot{>}} \psi \in \Phi_p$ ;
20. if  $\chi_L, \chi_{\bar{=}} \notin \Psi_p$ , then  $\chi_{\bar{P}}^{\dot{>}} \psi \in \Psi_p$  iff either  $\chi_{\bar{P}}^{\dot{<}} \psi \vee \ominus^d \psi \in \Theta_c$  or  $\chi_{\bar{P}}^{\dot{>}} \psi \in \Phi_p$ .

We proceed by proving the correctness of the construction for each operator, as we did for their future counterparts.

**Lemma 10.3.** *Given a finite set of atomic propositions  $AP$ , an OP alphabet  $(\mathcal{P}(AP), M_{AP})$ , a word  $w = \#xyz\#$  on it, and a position  $j = |xy|$  in  $w$ , we have*

$$(w, j) \models \chi_{\bar{P}}^{\dot{=}} \psi$$

*if and only if all accepting computations of an OPA satisfying rules 9-12 bring it from configuration  $\langle yz, \Phi, \alpha\gamma \rangle$  to a configuration  $\langle z, \Phi', \alpha'\gamma \rangle$  such that  $|\alpha| = 1, |\alpha'| = 1$  if  $\text{first}(y)$  is read by a shift move,  $|\alpha'| = 2$  if it is read by a push move, and  $\chi_{\bar{P}}^{\dot{=}} \psi \in \Phi_c^j$ , where  $\Phi^j$  is the state of the OPA before reading  $j$ , the last position of  $y$ . If no other rules constrain the transition relation, at least one computation is accepting.*

*Proof.*  $[\Rightarrow]$  Suppose  $\chi_{\bar{P}}^{\dot{=}} \psi$  holds in position  $i$ , corresponding to terminal symbol  $a$ . Then, there exists a position  $j$ , labeled with terminal  $d$ , s.t.  $\chi(i, j), a \dot{=} d$ , and  $\psi$  holds in  $i$ . Since  $a$  and  $d$  are the context of a chain, the input word must have the form of Figure 10.2. All accepting computations of the OPA reach configuration  $\langle a \dots z, \Phi^a, [f, \Phi^f]\gamma \rangle$  before reading  $a$ . By the inductive assumption, we have  $\psi \in \Phi^a$ .  $a$  is read by a shift or a push move, bringing the OPA to  $\langle c_0^0 \dots z, \Phi^{c_0^0}, \delta \rangle$ , with  $\delta = \alpha'\gamma$ , and either  $\alpha' = [a, \Phi^a][f, \Phi^f]$  or  $\alpha' = [a, \Phi^f]$ , respectively. Due to rule (12), we

have  $\chi_{\bar{P}}^{\dot{a}} \psi \in \Phi_p^{c_0^0}$ . After reading  $c_0^0$ , the OPA reaches configuration  $\langle v_0^0 \dots z, \Phi^{v_0^0}, [c_0^0, \Phi^{c_0^0}] \delta \rangle$ . Then, the automaton proceeds to read the rest of the body of chain  $\chi(i, j)$ . If  $i$  is the left context of multiple chains, the stack symbol  $[c_0^0, \Phi^{c_0^0}]$ , containing  $\chi_{\bar{P}}^{\dot{a}} \psi$  as a pending obligation, is popped before reaching  $d$ . Let  $b_p$ ,  $1 \leq p \leq n$ , be all labels of positions  $i_{b_p}$  s.t.  $\chi(i, i_{b_p})$  and  $a < b_p$ . It can be proved inductively that, before reading any of such positions, the OPA is in a configuration  $\langle b_p \dots z, \Phi^{b_p}, [c_{m_{p-1}}^{p-1}, \Phi^{b_{p-1}}] \delta \rangle$ , with  $\chi_{\bar{P}}^{\dot{a}} \psi \in \Phi^{b_{p-1}}$ . Since  $c_{m_{p-1}}^{p-1} > b_p$ , the next move is a pop, leading to a configuration  $\langle b_p \dots z, \Phi^{b_p}, \delta \rangle$ , with  $\chi_{\bar{P}}^{\dot{a}} \psi \in \Phi^{b_p}$ , due to rule (11). Then,  $b_p$  is read by a push move because  $a < b_p$ , so  $\chi_{\bar{P}}^{\dot{a}} \psi$  is again stored in the topmost stack symbol as a pending obligation, in a configuration  $\langle v_0^p \dots z, \Phi^{v_1^p}, [b_p, \Phi^{b_p}] \delta \rangle$ . The stack symbol containing  $\chi_{\bar{P}}^{\dot{a}} \psi$  is only popped in positions  $b_p$ , or when reaching  $d$ , since subchains only cause the OPA to push and pop new symbols.

So, configuration  $\langle dz, \Phi^d, [c_{m_n}^n, \Phi^{b_n}] \delta \rangle$  is reached, with  $\chi_{\bar{P}}^{\dot{a}} \psi \in \Phi^{b_n}$  (note that  $d$  labels the last position of  $y$ ). Due to rule (11), a pop move leads the OPA to  $\langle dz, \Phi^d, \delta \rangle$ , with  $\chi_{\bar{P}}^{\dot{a}} \psi \in \Phi^{b_n}$ . Then, since by hypothesis  $a \doteq d$ , and  $a$  is contained in the topmost stack symbol,  $d$  is read by a shift move. Since this transition is preceded by a pop, we have a computation in which  $\chi_R \in \Phi^{b_n}$ . So, by rule (9), since  $\chi_{\bar{P}}^{\dot{a}} \psi, \chi_R \in \Phi^{b_n}$ , we have  $\chi_{\bar{P}}^{\dot{a}} \psi \in \Phi_c^d$ , with the stack equal to  $\delta$ , satisfying the thesis statement. Computations of this form can proceed until acceptance, if not blocked by rules other than 1-3.

[ $\Leftarrow$ ] Suppose that, while reading  $w$ , an accepting computation of the OPA arrives at a configuration  $\langle dz, \Phi^d, \delta \rangle$ , where  $d$  is the last character of  $y$ , and  $\chi_{\bar{P}}^{\dot{a}} \psi \in \Phi_c^d$ . By rule (9), we have  $\chi_{\bar{P}}^{\dot{a}} \psi, \chi_R \in \Phi^{b_n}$ .  $\chi_R \in \Phi^{b_n}$  requires the previous transition to be a pop, so  $d$  is the right context of a chain. Let  $a$  be its left context. By hypothesis, the computation proceeds reading  $d$ , and by rule (10) it must be read by a shift transition. So, we have  $a \doteq d$ , and  $w$  must be of the form of Figure 10.2. Going back to  $\langle dz, \Phi^d, \delta \rangle$ , consider the pop move leading to this configuration. It starts from configuration  $\langle dz, \Phi^d, [c_{m_n}^n, \Phi^{b_n}] \delta \rangle$ , and by rule (11) we have  $\chi_{\bar{P}}^{\dot{a}} \psi \in \Phi^{b_n}$ .

Consider the move that pushed  $\Phi^{b_n}$  onto the stack. Suppose it was preceded by a pop move. Since  $\Phi^{b_n}$  is the target state of this transition, and  $\chi_{\bar{P}}^{\dot{a}} \psi \in \Phi^{b_n}$ , by rule (11)  $\chi_{\bar{P}}^{\dot{a}} \psi$  must be contained as a pending obligation in the popped state as well. So, this obligation is propagated backwards every time the automaton encounters a position that is the left context of a chain, i.e. positions  $b_p$ ,  $1 \leq p \leq n$ , in Figure 10.2. In order to stop the propagation, a push of a state with  $\chi_{\bar{P}}^{\dot{a}} \psi$  as a pending obligation, preceded by another push or shift move must be encountered. Such a transition pushes or updates the stack symbol under the one containing  $\chi_{\bar{P}}^{\dot{a}} \psi$ , which means the left context  $a$  s.t.  $a \doteq d$  of a chain whose right context is  $d$  has been reached. In both cases, the target state of the push/shift transitions contains  $\chi_{\bar{P}}^{\dot{a}} \psi$  as a pending obligation, so by rule (12) we have  $\psi \in \Phi_c^a$ . Hence, by the inductive assumption,  $\psi$  holds in position  $i$  (corresponding to  $a$ ), we have  $i \doteq j$  and  $\chi(i, j)$ , which implies  $\chi_{\bar{P}}^{\dot{a}} \psi$  holds in  $j$ .  $\square$

The proof of the model checking rules of  $\chi_{\bar{P}}^{\leq} \psi$  is similar to the one of Lemma 10.3, and is therefore omitted.

**Lemma 10.4.** *Given a finite set of atomic propositions  $AP$ , an OP alphabet  $(\mathcal{P}(AP), M_{AP})$ , a word  $w = \#xyz\#$  on it, and a position  $j = |xy|$  in  $w$ , we have*

$$(w, j) \models \chi_{\bar{P}}^{\geq} \psi$$

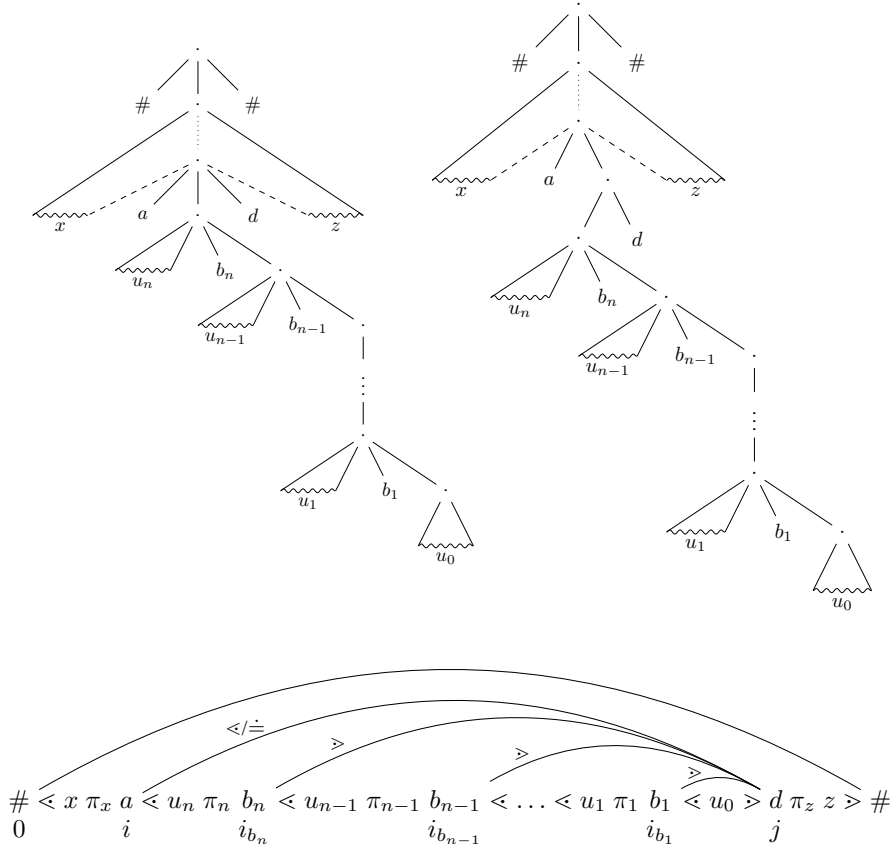


Figure 10.4: The two possible STs of a generic OP word  $w = xyz$  (top) expanded on the rightmost non-terminal, and its flat representation with chains (bottom). Wavy lines are placeholders for subtree frontiers. We have either  $a \doteq d$  (top left) or  $a \leq d$  (top right), and  $b_k \geq d$  for  $1 \leq k \leq n$ . For  $1 \leq k \leq n$ , we either have  $b_{k+1}[u_k]^{b_k}$ , or  $u_k$  is of the form  $v_0^k c_0^k v_1^k c_1^k \dots c_{m_k}^k v_{m_k+1}^k$ , where  $c_p^k \doteq c_{p+1}^k$  for  $0 \leq p < m_k$ ,  $c_{m_k}^k \doteq b_k$ , and resp.  $a \leq c_0^n$  and  $b_{k+1} \leq c_0^k$ . Moreover, for each  $0 \leq p < m_k$ , either  $v_{p+1}^k = \varepsilon$  or  $c_p^k [v_{p+1}^k]^{c_{p+1}^k}$ ; either  $v_{m_k+1}^k = \varepsilon$  or  $c_{m_k}^k [v_{m_k+1}^k]^{b_k}$ , and either  $v_0^k = \varepsilon$  or  $b_{k+1}[v_0^k]^{c_0^k}$  (resp.  ${}^a[v_0^n]^{c_0^n}$ ).  $u_0$  has the same form, except  $v_{m_0}^0 = \varepsilon$  and  $c_{m_0}^0 \geq d$ . The  $\pi_i$ s are placeholders for precedence relations, and they vary depending on the surrounding terminal characters.

if and only if all accepting computations of an OPA satisfying rules 17-20 bring it from configuration  $\langle yz, \Phi, \alpha\gamma \rangle$  to a configuration  $\langle z, \Phi', \alpha'\gamma \rangle$  such that  $|\alpha| = 1$ ,  $|\alpha'| = 1$  if  $\text{first}(y)$  is read by a shift move,  $|\alpha'| = 2$  if it is read by a push move, and  $\chi_P^{\triangleright} \psi \in \Phi_c^j$ , where  $\Phi^j$  is the state of the OPA before reading  $j$ , the last position of  $y$ . If no other rules constrain the transition relation, at least one computation is accepting.

*Proof.*  $[\Rightarrow]$  Suppose  $\chi_P^{\triangleright} \psi$  holds in position  $j$ . Then,  $j$  is the right context of at least two chains, and the word  $w$  has the form of Figure 10.4, with  $i$  being the left context of the outermost chain whose right context is  $j$ . Let positions  $i_{b_p}$ , labeled with  $b_p$ ,  $1 \leq p \leq n$ , be all other left contexts of chains sharing  $j$  as their right context. There exists a value  $q$ ,  $\leq q \leq n$ , s.t.  $\psi$  holds in  $i_{b_q}$ .

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

$$\langle b_q \dots z, \Phi^{b_q}, [c_{m_q}^q, \Phi^{c_0^q}][b_{q+1}, \Phi^{c_0^{q+1}}] \dots \delta \rangle,$$

with  $\psi \in \Phi_c^{b_q}$ ,  $\delta = \alpha'\gamma$ , and either  $\alpha' = [a, \Phi^a][f, \Phi^f]$ , if  $a$  (the label of  $i$ ) was read by a push move, or  $\alpha' = [a, \Phi^f]$  if it was read by a shift. Note that if  $b_q$  is the only character in its simple chain body ( $u_q = \varepsilon$  in Figure 10.4)  $[c_{m_q}^q, \Phi^{c_0^q}]$  is not present on the stack. In this case,  $b_q$  is read by a push move instead of a shift. Suppose  $b_q$  is the left context of one or more chains, besides the one whose right context is  $j$ . In Figure 10.4, this means  $v_0^{q-1} \neq \varepsilon$ . Consider the right context of the outermost of such chains: w.l.o.g. we call it  $c_0^{q-1}$  (it may as well be  $b_{q-1}$ ). Since  $\psi$  holds in  $i_{b_q}$ ,  $\chi_P^{\triangleright} \psi$  holds in  $c_0^{q-1}$ . If, instead,  $v_0^{q-1} = \varepsilon$ , then  $c_0^{q-1}$  is the successor of  $b_q$ , and  $\ominus^{\leq} \psi$  holds in it. In both cases,  $\chi_P^{\leq} \psi \vee \ominus^{\leq} \psi$  holds in  $c_0^{q-1}$ . Since  $b_q < c_0^{q-1}$ , the latter is read by a push transition, pushing stack symbol  $[c_0^{q-1}, \Phi^{c_0^{q-1}}]$ , with  $\chi_P^{\leq} \psi \vee \ominus^{\leq} \psi \in \Phi_c^{c_0^{q-1}}$ . This symbol remains on stack until  $d$  is reached, although its terminal symbol may be updated. The computation then proceeds normally, until configuration  $\langle dz, \Phi^{(q-2)d}, [b_{q-1}, \Phi^{c_0^{q-1}}] \dots \delta \rangle$  is reached.

Since  $\chi_P^{\leq} \psi \vee \ominus^{\leq} \psi \in \Phi_c^{c_0^{q-1}}$ , by rule (20), the OPA transitions to configuration  $\langle dz, \Phi^{(q)d}, [b_q, \Phi^{c_0^q}] \dots \delta \rangle$  with  $\chi_P^{\triangleright} \psi \in \Phi_p^{(q)d}$  and  $\chi_L, \chi_{\pm} \notin \Phi_p^{(q)d}$ . (Note that the next transition must be a pop, since the topmost stack symbol is  $b_q$ , and  $b_q \triangleright d$ .) Then, by rule (20), all subsequent pop transitions propagate  $\chi_P^{\triangleright} \psi$  as a pending obligation in the OPA state, until configuration  $\langle dz, \Phi^{(n-1)d}, \delta \rangle$ , with  $\chi_P^{\triangleright} \psi \in \Phi_p^{(n-1)d}$ . Now, the automaton guesses that this is the last pop move, and the next one will be a push or a shift. So, it transitions to  $\langle dz, \Phi^{(n)d}, \delta \rangle$ , with  $\chi_L \in \Phi_p^{(n)d}$  or  $\chi_{\pm} \in \Phi_p^{(n)d}$ , and  $\chi_P^{\triangleright} \psi \in \Phi_p^{(n)d}$ , according to rule (19). Also,  $\chi_R \in \Phi_p^{(n)d}$ , because the previous move was a pop. At this point,  $d$  is read with either a shift or a push transition. According to rule (18),  $\chi_P^{\triangleright} \psi \in \Phi_c^{(n)d}$ , which satisfies the thesis statement.

$[\Leftarrow]$  Suppose the automaton reaches a state  $\Phi^j = \Phi^{(n)d}$  s.t.  $\chi_P^{\triangleright} \psi \in \Phi_c^j$  during an accepting computation.  $j$  has to be read by either a push or a shift move, so either  $\chi_L \in \Phi_p^j$  or  $\chi_{\pm} \in \Phi_p^j$ . By rule (18), for the computation to continue, we have  $\chi_R \in \Phi_p^j$ . So, the transition leading to state  $\Phi_p^j$  must be a pop, and the related word position  $d$  is the right context of a chain. Let  $\Phi'^j$  be the starting state of this transition. Since  $\chi_P^{\triangleright} \psi \in \Phi_p^j$ , by rule (19) we have  $\chi_P^{\triangleright} \psi \in \Phi'^j$ . By rule (17), this transition must be preceded by another pop, so  $d$  is the right context of at least two chains, and the word being read is of the form of Figure 10.4, with  $n \geq 1$ .

So, before reading  $d$ , the OPA performs a pop transition for each inner chain having  $d$  as a right context, i.e. those having  $b_p$ ,  $1 \leq p \leq n$ , as left contexts in Figure 10.4,

plus one for the outermost chain (whose left context is  $a$ ). By rule (20),  $\chi_P^{\triangleright} \psi$  is propagated backwards through such transitions from the one before  $d$  is read, to one in which  $\chi_P^{\leq} \psi \vee \ominus^{\leq} \psi$  is contained into the popped state.

By rule (17), for the computation to reach such pop transitions, the propagation of  $\chi_P^{\triangleright} \psi$  as a pending obligation must stop. So, the OPA must reach a configuration  $\langle dz, \Phi^{(q)d}, [b_q, \Phi^{c_0^q}] \dots \delta \rangle$  with  $\chi_P^{\leq} \psi \vee \ominus^{\leq} \psi \in \Phi_c^{c_0^q}$ . Note that the following reasoning also applies to the case in which, in Figure 10.4,  $u_q = \varepsilon$ , by substituting  $b_q$  to  $c_0^q$ . The top-most stack symbol was pushed after configuration  $\langle c_0^q \dots z, \Phi^{c_0^q}, [b_{q-1}, \Phi^{c_0^{q-1}}] \dots \delta \rangle$ . We have  $b_{q-1} \leq c_0^q$ . If  $v_0^q = \varepsilon$ , and  $c_0^q$  is in the position next to  $b_{q-1}$ ,  $\ominus^{\leq} \psi$  holds, while if  $v_0^q \neq \varepsilon$ , since  $b_{q-1} [v_0^q] c_0^q$  is a chain,  $\chi_P^{\leq} \psi$  holds. Therefore,  $\psi$  holds in  $b_{q-1}$ . Since  $b_{q-1} \triangleright d$  and  $\chi(i_{b_{q-1}}, j)$ ,  $\chi_P^{\triangleright} \psi$  holds in  $j$ .  $\square$

### 10.1.4 Summary Until and Since

The construction for these operators is based on their expansion laws. The rules for until follow, those of since being symmetric. For any  $\Phi \in \text{Atoms}(\varphi)^2$ , we have  $\psi \mathcal{U}^t \theta \in \Phi_c$ , with  $t \in \{d, u\}$  being a direction, iff either: 1.  $\theta \in \Phi_c$ , 2.  $\circ^t(\psi \mathcal{U}^t \theta)$ ,  $\psi \in \Phi_c$ , or 3.  $\chi_P^t(\psi \mathcal{U}^t \theta)$ ,  $\psi \in \Phi_c$ .

### 10.1.5 Hierarchical Operators

For the hierarchical operators, we do not give an explicit OPA construction, but we rely on a translation into other POTL operands. For each hierarchical operator  $\eta$  in  $\varphi$ , we add a propositional symbol  $q_{(\eta)}$ . The upward hierarchical operators consider the right contexts of chains sharing the same left context. To distinguish such positions, we define formula  $\gamma_{L,\eta} := \chi_P^{\leq} (q_{(\eta)} \wedge \circ(\Box \neg q_{(\eta)}) \wedge \ominus(\Box \neg q_{(\eta)}))$ , where  $\Box \psi := \neg(\top \mathcal{U}_\chi^u (\top \mathcal{U}_\chi^d \neg \psi))$ , and  $\Box$  is symmetric.  $\circ$  and  $\ominus$  are the LTL next and back operators, for which model checking can be done as for  $\circ^d$  and  $\ominus^d$ , but removing the restrictions on PR. They could be replaced with  $\circ \psi := \circ^d \psi \vee \circ^u \psi$ , but this would cause an exponential blowup in the following equivalences, which can be used for model checking upwards hierarchical operators.  $\gamma_{L,\eta}$ , evaluated in one of the right contexts, asserts that  $q_{(\eta)}$  holds in the unique left context of the same chain, only.

$$\circ_H^u \psi := \gamma_{L,\circ_H^u} \psi \wedge \circ((\neg \chi_P^{\leq} q_{(\circ_H^u \psi)}) \mathcal{U}_\chi^u (\chi_P^{\leq} q_{(\circ_H^u \psi)} \wedge \psi)) \quad (10.1)$$

$$\ominus_H^u \psi := \gamma_{L,\ominus_H^u} \psi \wedge \ominus((\neg \chi_P^{\leq} q_{(\ominus_H^u \psi)}) \mathcal{S}_\chi^u (\chi_P^{\leq} q_{(\ominus_H^u \psi)} \wedge \psi)) \quad (10.2)$$

$$\psi \mathcal{U}_H^u \theta := \gamma_{L,\psi \mathcal{U}_H^u \theta} \wedge (\chi_P^{\leq} q_{(\psi \mathcal{U}_H^u \theta)} \implies \psi) \mathcal{U}_\chi^u (\chi_P^{\leq} q_{(\psi \mathcal{U}_H^u \theta)} \wedge \theta) \quad (10.3)$$

$$\psi \mathcal{S}_H^u \theta := \gamma_{L,\psi \mathcal{S}_H^u \theta} \wedge (\chi_P^{\leq} q_{(\psi \mathcal{S}_H^u \theta)} \implies \psi) \mathcal{S}_\chi^u (\chi_P^{\leq} q_{(\psi \mathcal{S}_H^u \theta)} \wedge \theta) \quad (10.4)$$

We only prove equivalence (10.3), as the others are essentially analogous.

**Lemma 10.5** (Equivalence (10.3)). *Let  $w$  be an OP word based on an alphabet of atomic propositions  $\mathcal{P}(AP)$ , and  $i$  a position in  $w$ , and let  $q_{(\psi \mathcal{U}_H^u \theta)} \notin AP$ ,  $\psi$  and  $\theta$  being two POTL formulas on  $AP$ . Let  $w'$  be a word on alphabet  $\mathcal{P}(AP \cup \{q_{(\psi \mathcal{U}_H^u \theta)}\})$  identical to  $w$ , except  $q_{(\psi \mathcal{U}_H^u \theta)}$  holds in position  $h < i$  s.t.  $\chi(h, i)$  and  $h \leq i$ .*

*Then,  $(w, i) \models \psi \mathcal{U}_H^u \theta$  iff  $(w', i) \models \Upsilon(\psi, \theta)$ , with*

$$\begin{aligned} \Upsilon(\psi, \theta) &:= \gamma_{L,\psi \mathcal{U}_H^u \theta} \wedge (\chi_P^{\leq} q_{(\psi \mathcal{U}_H^u \theta)} \implies \psi) \mathcal{U}_\chi^u (\chi_P^{\leq} q_{(\psi \mathcal{U}_H^u \theta)} \wedge \theta), \\ \gamma_{L,\eta} &:= \chi_P^{\leq} (q_{(\eta)} \wedge \circ(\Box \neg q_{(\eta)}) \wedge \ominus(\Box \neg q_{(\eta)})). \end{aligned}$$

*Proof.*  $[\Rightarrow]$  Suppose  $\psi \mathcal{U}_H^u \theta$  holds in position  $i$  in word  $w$ . Then, by its semantics, there exists a UHP  $i = i_0 < i_1 < \dots < i_n$ , with  $n \geq 0$ , and a position  $h < i$  s.t. for each  $i_p$ ,  $0 \leq p \leq n$ , we have  $\chi(h, i_p)$  and  $h < i_p$ , and for  $0 \leq q < n$ ,  $\psi$  holds in  $i_q$ , and  $\theta$  holds in  $i_n$ . We show that in  $\Upsilon(\psi, \theta)$  holds in  $i$  in  $w'$ . By construction, in  $w'$ ,  $\mathsf{q}_{(\psi \mathcal{U}_H^u \theta)}$  holds in  $h$  only. So,  $\mathsf{q}_{(\psi \mathcal{U}_H^u \theta)} \wedge \circ(\Box \neg \mathsf{q}_{(\psi \mathcal{U}_H^u \theta)}) \wedge \ominus(\Box \neg \mathsf{q}_{(\psi \mathcal{U}_H^u \theta)})$  holds in  $h$ , and  $\gamma_{L, \psi \mathcal{U}_H^u \theta}$  holds in  $i$ .

For  $(\chi_P^{\leq} \mathsf{q}_{(\psi \mathcal{U}_H^u \theta)} \implies \psi) \mathcal{U}_\chi^u (\chi_P^{\leq} \mathsf{q}_{(\psi \mathcal{U}_H^u \theta)} \wedge \theta)$  to hold in  $i$ , there must exist a USP between  $i = i_0$  and  $i_n$ . Suppose, by contradiction, that no such path exists. This implies there exist two positions  $r, s$ , with  $i \leq r < s \leq i_n$ ,  $r < s$ , either  $s = r + 1$  or  $\chi(r, s)$ , s.t. no USP can skip them. So, there exist no positions  $r', s'$  s.t.  $i \leq r' < s < s' \leq i_n$  s.t.  $\chi(r', s')$  and either  $r' \doteq s'$  or  $r' > s'$ . Since  $r < s$ ,  $r$  is the left context of a chain. Let  $k$  be the maximal (i.e. rightmost) position s.t.  $\chi(r, k)$ . There are three cases:

- $k > i_n$ . In this case,  $i_n$  is part of the body of the chain  $\chi(r, k)$ . However, by hypothesis,  $\chi(h, i_n)$ , and  $h < i \leq r < i_n < k$ . These two chains cross each other, which is impossible by the definition of chain.
- $k = i_n$ . If  $r \doteq i_n$  or  $r > i_n$ , then  $i_n$  is reachable by the USP. Otherwise, we would have  $\chi(h, i_n)$  and  $\chi(r, i_n)$ ,  $h < i_n$  and  $r < i_n$  with  $h \neq r$ , which is impossible because of property (3) of Lemma 6.3.
- $k < i_n$ . If  $r \doteq k$  or  $r > k$ , then  $r$  and  $k$  can be part of an USP reaching  $i_n$ . If  $r < k$ , then  $k$  is the first position of the body of another chain having  $r$  as its left context, which contradicts the assumption that  $k$  is maximal.

By hypothesis,  $\mathsf{q}_{(\psi \mathcal{U}_H^u \theta)}$  holds in  $h$ , so  $\chi_P^{\leq} \mathsf{q}_{(\psi \mathcal{U}_H^u \theta)}$  holds in all positions  $i_p$ ,  $0 \leq p \leq n$ , in the UHP. Since  $\theta$  holds in  $i_n$ ,  $\chi_P^{\leq} \mathsf{q}_{(\psi \mathcal{U}_H^u \theta)} \wedge \theta$  holds in it. Moreover, since  $\psi$  holds in all  $i_q$ ,  $0 \leq q \leq n$ ,  $\chi_P^{\leq} \mathsf{q}_{(\psi \mathcal{U}_H^u \theta)} \implies \psi$  holds in all positions in the USP between  $i_0$  and  $i_n$ .

$[\Leftarrow]$  Suppose  $(w', i) \models \Upsilon(\psi, \theta)$ . Then,  $\gamma_{L, \psi \mathcal{U}_H^u \theta}$  holds in  $i$ . This implies there exists a position  $h$  s.t.  $\chi(h, i)$  and  $h < i$ , which is unique by Lemma 6.3. By  $\gamma_{L, \psi \mathcal{U}_H^u \theta}$ ,  $\mathsf{q}_{(\psi \mathcal{U}_H^u \theta)}$  holds in  $h$  and in no other position. Moreover,  $(\chi_P^{\leq} \mathsf{q}_{(\psi \mathcal{U}_H^u \theta)} \implies \psi) \mathcal{U}_\chi^u (\chi_P^{\leq} \mathsf{q}_{(\psi \mathcal{U}_H^u \theta)} \wedge \theta)$  holds in  $i$ , so there exists an USP  $i = j_0 < j_1 < \dots < j_m$ . We show that there exists a sequence of indices  $0 = p_0 < p_1 < \dots < p_n = m$  s.t.  $j_{p_0}, j_{p_1}, \dots, j_{p_n}$  is a UHP satisfying  $\psi \mathcal{U}_H^u \theta$  in  $i$  in  $w$ .

First, note that  $\theta$  holds in  $j_{p_n}$ , and since  $h$  is the only position in which  $\mathsf{q}_{(\psi \mathcal{U}_H^u \theta)}$  holds, we have  $\chi(h, j_{p_n})$  and  $h < j_{p_n}$ . So,  $j_{p_n}$  is the last position of a UHP starting in  $i$ . For each position  $j$  s.t.  $i < j < j_{p_n}$ ,  $\chi(h, j)$  and  $h < j$ , there exists an index  $1 \leq q \leq n - 1$  s.t.  $j_{p_q} = j$ . Since all such positions  $j$  are between  $j_0$  and  $j_m$ , the USP could skip them only if they were part of the body of a chain, i.e. if there exist two positions  $j_0 \leq r < s \leq j_m$  s.t.  $\chi(j_0, j_m)$  and either  $r \doteq s$  or  $r > s$ . Such a chain would, however, cross with  $\chi(h, j)$ , which contradicts the definition of chain.

Because  $\mathsf{q}_{(\psi \mathcal{U}_H^u \theta)}$  only holds in  $h$ , the fact that  $\chi_P^{\leq} \mathsf{q}_{(\psi \mathcal{U}_H^u \theta)} \implies \psi$  holds in all positions  $j_0, j_1, \dots, j_{m-1}$  implies  $\psi$  holds in all of  $j_{p_0}, j_{p_1}, \dots, j_{p_{n-1}}$ . So, by construction of  $w'$ ,  $j_{p_0}, j_{p_1}, \dots, j_{p_n}$  is a UHP satisfying  $\psi \mathcal{U}_H^u \theta$  in position  $i$  in  $w$ .  $\square$

We now give the equivalences for downward hierarchical operators. The following formula, when evaluated in the left context of a chain, forces symbol  $p_\eta$  in the right context. Note that if the left context is in the  $>$  relation with the right one, the

latter is uniquely identified.

$$\gamma_{R,\eta} := \chi_F^{\succ} (\mathfrak{q}(\eta) \wedge \circ(\Box \neg \mathfrak{q}(\eta)) \wedge \ominus(\Box \neg \mathfrak{q}(\eta)))$$

$$\circ_H^d \psi := \gamma_{R,\circ_H^d \psi} \wedge \circ((\neg \chi_F^{\succ} \mathfrak{q}(\circ_H^d \psi)) \mathcal{U}_\chi^d (\chi_F^{\succ} \mathfrak{q}(\circ_H^d \psi) \wedge \psi)) \quad (10.5)$$

$$\ominus_H^d \psi := \gamma_{R,\ominus_H^d \psi} \wedge \ominus((\neg \chi_F^{\succ} \mathfrak{q}(\ominus_H^d \psi)) \mathcal{S}_\chi^d (\chi_F^{\succ} \mathfrak{q}(\ominus_H^d \psi) \wedge \psi)) \quad (10.6)$$

$$\psi \mathcal{U}_H^d \theta := \gamma_{R,\psi \mathcal{U}_H^d \theta} \wedge (\chi_F^{\succ} \mathfrak{q}(\psi \mathcal{U}_H^d \theta) \implies \psi) \mathcal{U}_\chi^d (\chi_F^{\succ} \mathfrak{q}(\psi \mathcal{U}_H^d \theta) \wedge \theta) \quad (10.7)$$

$$\psi \mathcal{S}_H^d \theta := \gamma_{R,\psi \mathcal{S}_H^d \theta} \wedge (\chi_F^{\succ} \mathfrak{q}(\psi \mathcal{S}_H^d \theta) \implies \psi) \mathcal{S}_\chi^d (\chi_F^{\succ} \mathfrak{q}(\psi \mathcal{S}_H^d \theta) \wedge \theta) \quad (10.8)$$

### 10.1.6 Concluding Proof

**Theorem 10.6** (Correctness of Finite Model Checking.). *Given a finite set of atomic propositions  $AP$ , an OP alphabet  $(\mathcal{P}(AP), M_{AP})$ , a word  $w$  on it, and an POTL formula  $\varphi$ , the automaton built according to the procedure in this section is such that we have*

$$(w, 1) \models \varphi$$

*if and only if it performs at least one accepting computation on a word  $w'$  equal to  $w$ , except for the presence of one more propositional symbol for each hierarchical operator in  $\varphi$ .*

*Proof.* We proved that all chain next/back operators hold in a position in  $w$  iff in all accepting computations, after reading a subword of  $w$ , the OPA is left in a state not containing any pending obligation related to that instance of the operator (cf. Lemmas 10.1, 10.2, 10.3, 10.4). While the correctness of the upward/downward next/back operators is trivial, that of summary until/since operators is due to the correctness of the respective expansion laws, proved in Lemma 8.4. Moreover, in Lemma 10.5 we proved the correctness of the equivalences for the hierarchical operators.

The results above allow us to prove that, by structural induction on the syntax of  $\varphi$ , if  $\varphi$  holds in position 1 of  $w$ , there exists a word  $w'$  identical to  $w$ , except for the propositional symbols needed for the hierarchical operators, such that the OPA performs at least a computation reaching the end of  $w$  in a state containing no future operators and no temporal obligations. By the definition of the set of final states  $F$ , such a computation is accepting.

Conversely, suppose there exists a word  $w'$  with the described features on which the OPA performs at least one accepting computation starting from a state containing  $\varphi$ . Then  $\varphi$  holds in the first position of a word  $w$  built by removing the propositional symbols introduced by equivalence formulas for hierarchical operators. Indeed, such a computation ends with an empty stack, and a state containing no future operators or temporal obligations which, by the lemmas listed above, implies all temporal obligations have been satisfied, and  $w$  is a model for  $\varphi$ .  $\square$

**Complexity** The set  $\text{Cl}(\varphi)$  is linear in  $|\varphi|$ , the length of  $\varphi$ .  $\text{Atoms}(\varphi)$  has size at most  $2^{|\text{Cl}(\varphi)|} = 2^{O(|\varphi|)}$ , and the size of the set of states is the square of that. Moreover, the use of the equivalences for the hierarchical operators causes only a linear increase in the length of  $\varphi$ . Therefore,

**Theorem 10.7.** *Given a POTL formula  $\varphi$ , it is possible to build an OPA  $\mathcal{A}_\varphi$  accepting the language denoted by  $\varphi$  with at most  $2^{O(|\varphi|)}$  states.*

$\mathcal{A}_\varphi$  can then be intersected with an OPA modeling a program (e.g. Figure 4.5), and emptiness can be decided with polynomial-time reachability algorithms that we will present in Section 11.1.

Since it is possible to linearly translate NWTL into POTL in a way similar to what we did with OPTL in Section 7.1, we can exploit the same lower bounds for decision problems:

**Theorem 10.8.** *POTL model checking and satisfiability on finite OP words are EXP-TIME-complete.*

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

## 10.2 $\omega$ -Word Model Checking

To perform model checking of an OPTL formula  $\varphi$  on OP  $\omega$ -words, we use the same approach as in [54]. We build a generalized  $\omega$ OPBA (cf. Definition 4.17)

$$\mathcal{A}_\varphi^\omega = \langle \mathcal{P}(AP), M_{AP}, Q_\omega, I, \mathbf{F}, \delta \rangle,$$

where  $Q_\omega = \text{Atoms}(\varphi) \times \text{Atoms}(\varphi) \times \mathcal{P}(\text{Cl}_{stack}(\varphi))$ .

In finite words, the stack is empty at the end of every accepting computation, which implies the satisfaction of all temporal constraints tracked by the pending part of stack symbols. In  $\omega$ OPBAs, the stack may never be empty, and symbols with a non-empty pending part may remain in it indefinitely, never enforcing the satisfaction of the respective formulas. To overcome this issue, we use  $\text{Atoms}(\varphi) \times \text{Atoms}(\varphi) \times \mathcal{P}(\text{Cl}_{stack}(\varphi))$ , with  $\text{Cl}_{stack}(\varphi) \subseteq \text{Cl}(\varphi)$ , as the state set of the  $\omega$ OPBA. Such states have the form  $\Phi = (\Phi_c, \Phi_p, \Phi_s)$ , where  $\Phi_c$  and  $\Phi_p$  have the same role as in the finite-word case, and  $\Phi_s$  is the *in-stack* part of  $\Phi$ . All rules defined in Section 10.1 for  $\Phi_c$  and  $\Phi_p$  remain the same.  $\Phi_s$  contains elements of  $\text{Cl}_{stack}(\varphi)$  contained in any symbol currently on the stack.  $\text{Cl}_{stack}(\varphi)$  contains formulas in  $\text{Cl}(\varphi)$  that use the stack to ensure the satisfaction of future temporal requirements, namely all  $\chi_F^\pi \psi \in \text{Cl}(\varphi)$ , with  $\pi \in \{<, \doteq, >\}$ . Thus, pending temporal obligations are moved from the stack to the  $\omega$ OPBA state, and they can be considered by the Büchi acceptance condition.

Suppose we want to model check  $\chi_F^\doteq \psi$ . Formula  $\chi_F^\doteq \psi$  must be inserted in the in-stack part of the current state whenever a stack symbol containing it in its pending part is pushed. It must be kept in the in-stack part of the current state until the last stack symbol containing it in its pending part is popped, marking the satisfaction of its temporal requirement. Then, it is possible to define an acceptance set  $F_{\chi_F^\doteq \psi} \in \mathbf{F}$ , as the set of states not containing  $\chi_F^\doteq \psi$  in any part.

This construction is formalized as follows. Let  $\psi \in \text{Cl}_{stack}(\varphi)$ . We add a few constraints on the transition relations. For any  $\Phi, \Theta, \Psi \in Q_\omega$  and  $a \in \mathcal{P}(AP)$ , let  $(\Phi, a, \Theta) \in \delta_{push}$ :

21. if  $\psi \in \Phi_p$ , then  $\psi \in \Theta_s$ ;

let  $(\Phi, a, \Theta) \in \delta_{push}$  or  $(\Phi, a, \Theta) \in \delta_{shift}$ :

22. if  $\psi \in \Phi_s$ , then  $\psi \in \Theta_s$ ;

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

23. if  $\psi \in \Phi_s$  and  $\psi \in \Theta_s$ , then  $\psi \in \Psi_s$ .

Thus, we can state the following:

**Lemma 10.9.** *Let  $AP$  be a finite set of atomic propositions,  $(\mathcal{P}(AP), M_{AP})$  an OP alphabet,  $\psi \in \text{Cl}_{\text{stack}}(\varphi)$ , and  $\mathcal{A}$  an  $\omega$ OPBA satisfying rules 21-23 above. For any  $\omega$ -word  $w = \#xy$  on  $(\mathcal{P}(AP), M_{AP})$ , let  $\langle y, \Phi, \gamma \rangle$  be  $\mathcal{A}$ 's configuration after reading  $x$ .*

*If there exists a stack symbol  $[a, \Theta] \in \gamma$  such that  $\psi \in \Theta_p$ , then  $\psi \in \Phi_s$ .*

We omit the proof, as it is substantially equivalent to the one of Lemma 6.1 in [54].

The only operators needing to be in  $\text{Cl}_{\text{stack}}(\varphi)$  are  $\chi_F^\pi \psi$ , with  $\pi \in \{\prec, \dot{=}, \succ\}$ , since the satisfaction of all other future operators depends on them.

**Lemma 10.10.** *Given a finite set of atomic propositions  $AP$ , an OP alphabet  $(\mathcal{P}(AP), M_{AP})$ , an  $\omega$ -word  $w = \#xy$  on it, a position  $i = |x| + 1$  in  $w$ , and a PR  $\pi \in \{\prec, \dot{=}, \succ\}$ , we have*

$$(w, i) \models \chi_F^\pi \psi$$

*if and only if an OPA satisfying rules 1-8 and 21-23 performs an accepting computation passing through a configuration  $\langle y, \Phi, \gamma \rangle$  with  $\chi_F^\pi \psi \in \Phi_c$ .*

*Proof.* [ $\Rightarrow$ ] Suppose  $\chi_F^\pi \psi$  does not appear in any pending part of the states in  $\gamma$ , and  $\chi_F^\pi \psi \notin \Phi_p$ . Then, by Lemmas 10.1-10.2, after reading the body of the chain after  $i$ , the OPA arrives to a configuration  $\langle z, \Phi', \gamma' \rangle$  where  $\chi_F^\pi \psi$  does not appear in any pending part of the states in  $\gamma'$ . Hence, by Lemma 10.9,  $\chi_F^\pi \psi \notin \Phi'_s$ . Thus, even if  $\chi_F^\pi \psi$  holds infinitely often in  $w$ , accepting states in which it does not appear in the in-stack part occur infinitely often.

If  $\chi_F^\pi \psi$  was present in any pending part of the states in  $\gamma$ , or  $\chi_F^\pi \psi \in \Phi_p$ , then a pending instance of  $\chi_F^\pi \psi$  appeared previously in the computation. If such instance is the result of a wrong guess, then a computation without such guess exists. Otherwise,  $\chi_F^\pi \psi$  holds in a position  $i'$  of  $x$ , and the above reasoning can be applied to it, proving that an accepting state is finally reached.

[ $\Leftarrow$ ] If an accepting computation exists, then a state  $\Phi'$  with  $\chi_F^\pi \psi \notin \Phi'_s$  occurs infinitely often. By Lemma 10.9, in such configurations  $\chi_F^\pi \psi$  is not present in pending parts of stack states. Thus, the stack symbol pushed after reading  $i$ , which by rules 1, 4 and 6 contains  $\chi_F^\pi \psi$  in its pending part, is finally popped. Thus, the right part of the implications of Lemmas 10.1-10.2 applies, and  $\chi_F^\pi \psi$  holds in  $i$ .  $\square$

An acceptance condition for summary until operators is also needed. For  $\psi \mathcal{U}_X^d \theta \in \text{Cl}(\varphi)$ , we add an acceptance set  $\mathbf{F}_{\psi \mathcal{U}_X^d \theta}$  such that for any  $\Phi$  in it we have  $\chi_F^\prec(\psi \mathcal{U}_X^d \theta)$ ,  $\chi_F^{\dot{=}}(\psi \mathcal{U}_X^d \theta) \notin \Phi_s$ , and either  $\psi \mathcal{U}_X^d \theta \notin \Phi_c$  or  $\theta \in \Phi_c$ . The condition for  $\psi \mathcal{U}_X^u \theta$  is symmetric.

We can now conclude the proof:

**Theorem 10.11** (Correctness of  $\omega$  Model Checking). *Given a finite set of atomic propositions  $AP$ , an OP alphabet  $(\mathcal{P}(AP), M_{AP})$ , an  $\omega$ -word  $w$  on it, and an POTL formula  $\varphi$ , the automaton built according to the procedure in this section is such that we have*

$$(w, 1) \models \varphi$$

*if and only if it performs at least one accepting computation on a word  $w'$  equal to  $w$ , except for the presence of one more propositional symbol for each hierarchical operator in  $\varphi$ .*

*Proof.* The claim follows from Theorem 10.6 for past operators, and from Lemma 10.10 for the future ones.  $\square$

**Complexity** The complexity claims made for finite-word model checking can be easily extended to the infinite case, as the presence of the *in-stack* part of states does not cause a further blow-up of their amount.

**Theorem 10.12.** *Given a POTL formula  $\varphi$ , it is possible to build an  $\omega$ OPBA  $\mathcal{A}_\varphi$  accepting the language denoted by  $\varphi$  with at most  $2^{O(|\varphi|)}$  states.*

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

**Theorem 10.13.** *POTL model checking and satisfiability on OP  $\omega$ -words are EXPTIME-complete.*

# Chapter 11

## Experimental Evaluation

### 11.1 Implementation

We implemented the OPA and  $\omega$ OPBA constructions of Chapter 10 in an explicit-state model checking tool called POMC [55]. The tool is written in Haskell [124], a purely functional, statically typed programming language with lazy evaluation. The declarative nature of Haskell makes it easier to code the numerous rules required by the construction. Lazy evaluation is exploited to only evaluate rules that are actually needed, depending on operators in the closure, and to generate OPA states on-the-fly.

Given a POTL specification  $\varphi$  and an OPA (resp.  $\omega$ OPBA)  $\mathcal{A}$  to be checked, POMC executes the reachability algorithm, generating the product between  $\mathcal{A}$  and the OPA (resp.  $\omega$ OPBA) for  $\neg\varphi$  on-the-fly.

POMC checks OPA for emptiness by checking the reachability of an accepting configuration, by means of a modified DFS of the transition relation. Emptiness checking for  $\omega$ OPBA is significantly more involved, since fairness cycles must be found in the transition relation. This is done by means of graph theoretic techniques. Such algorithms are similar to the ones in [10].

#### 11.1.1 OPA Emptiness Checking

The reachability algorithm we use exploits the fact that all transitions only consider the topmost stack symbol, so reachability is actually computed only for *semi-configurations* made of one stack symbol and one state. Each time a chain support is explored, its ending semi-configuration is saved and associated with the starting one, so the next time the latter is reached, the support does not have to be re-explored. This allows the algorithm to exploit the cyclicities of OPA to terminate after having explored the whole transition relation.

Given an OP alphabet  $(\Sigma, M_\Sigma)$ , where  $\Sigma$  is a finite input alphabet, let  $\mathcal{A} = \langle \Sigma, M_\Sigma, Q, I, F, \delta \rangle$  be an OPA. Let  $\Gamma = \Sigma \times Q \cup \{\perp\}$  be the set of stack symbols.

**Definition 11.1.** A *semi-configuration* of  $\mathcal{A}$  is an element of  $\mathcal{C} = Q \times \Gamma$ .

Algorithm 1 solves the reachability problem for OPA, by adapting a DFS to the use of summaries. Function REACH receives as its arguments a state  $q \in Q$ , a stack symbol  $g \in \Gamma$ , a character  $c \in \Sigma$ , and a look-ahead  $\ell \in \Sigma \cup \{*\}$ . If  $\ell = *$ , then any character in  $\Sigma$  may be used as a look-ahead. The algorithm searches the transition graph of the OPA, and it stops when it finds out that a semi-configuration  $(q, g)$  is reachable. To

solve the emptiness problem, it suffices to pose  $Q_R = F$  and  $\Gamma_R = \{\perp\}$ , and to call  $\text{REACH}(q, \perp, \#, *)$  for each  $q \in I$ , as shown in Algorithm 2.

The algorithm stores the OPA's semi-configuration whenever it enters or exits a chain support, respectively in *SupportStarts* and *SupportEnds*. Thus, whenever it finds the beginning of a support in a semi-configuration that has already been visited, it just uses this pre-computed information to jump to the corresponding pop move directly. We can see this as the insertion of additional edges in the OPA's transition graph, which we call *summary* edges. This is not just a performance optimization, but it is essential for the algorithm's correctness. In fact, due to the context-free nature of OPLs, an OPA may have to go through nested chain supports that start with the same semi-configuration.

Suppose one of such supports starts with a push transition  $(q, b, p)$ , coming from semi-configuration  $(q, g)$ . When  $\text{REACH}$  meets this move a second time, it returns false to avoid an infinite loop. However, this stops the exploration of the chain support. Thus,  $(q, g)$  is saved in *SupportStarts*. If an accepting run containing  $(q, g)$  exists, then it must eventually lead to a chain support that is not cyclic, so that the run terminates. When such a chain support is visited, the resulting pop transition finds  $(q, g)$  in *SupportStarts* and resumes the initial computation.

On the other hand, the algorithm may reach the same support again while reading a different character from the same semi-configuration. To prevent it from stopping because a semi-configuration in the support has already been visited, pop transitions are saved in *SupportEnds* when they are encountered, so that when a semi-configuration leading to them is found, the exploration jumps directly to the end of the support.

**Complexity** Each call to  $\text{REACH}$  has worst-case time complexity  $O(|\delta| |\delta_{push}|^2 |\Sigma|)$  and space complexity  $O(|\delta| |\delta_{push}| |\Sigma|)$ . Note that only transitions and states that are actually visited contribute to the complexity, so the above bounds are reached only if the whole OPA is visited. Also, if  $\Sigma$  contains sets of atomic propositions, we consider only those on which the OPM is defined. E.g., with  $M_{\text{call}}$  we use only elements of  $\Sigma_{\text{call}}$  as look-aheads, and  $|\Sigma_{\text{call}}|$  is a small constant.

### 11.1.2 $\omega$ OPBA Emptiness Checking

The algorithm for checking emptiness of an  $\omega$ OPBA has been developed in [139]. Due to the Büchi acceptance condition, to check whether an  $\omega$ OPBA has an accepting run we need to check for reachable cycles containing final states. In NBAs this is done with a nested DFS, but adapting this algorithm to  $\omega$ OPBAs is sub-optimal [10], because the dynamic discovery of summary edges may add cycles in parts of the transition graph that have already been explored. Thus, we use an on-line algorithm to incrementally compute Strongly Connected Components (SCCs) while summary edges are discovered.

We use the path-based algorithm by H.N. Gabow [87], which is well-suited for early-termination, because it is based on a DFS. This algorithm works by exploring the graph with a DFS, and contracting SCCs as it finds back-edges. It finds all SCCs in a graph in linear time, by using simple data structures such as arrays and stacks.

We use it in an algorithm that works by alternating two phases:

- a *search* phase, in which the transition graph of the  $\omega$ OPBA is explored without following summary edges (or chain supports), which are stored in a set;

---

**Algorithm 1** OPA semi-configuration reachability

---

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

---

---

**Algorithm 2** OPA emptiness check

---

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

---

<pre> PROGRAM = [DECLS] FUNCTION [FUNCTION ...] DECLS = var IDENTIFIER [, IDENTIFIER ...] ; FUNCTION = IDENTIFIER ( ) { STMT; [STMT; ...] } STMT = IDENTIFIER := BEXPR         while (BEXPR) { [STMT; ...] }         if (BEXPR) { [STMT; ...] } else { [STMT; ...] }         try { [STMT; ...] } catch { [STMT; ...] }         IDENTIFIER ( )         throw BEXPR = BEXPR &amp;&amp; BDISJ   BDISJ BDISJ = BDISJ    BTERM   BTERM BTERM = !BTERM   (BEXPR)   IDENTIFIER   true   false </pre>	<pre> program: var foo; pa() {   foo = false;   try { pb(); }   catch { pc(); } } pb() {   if (foo) { throw; }   else {} } pc() { } </pre>
---	--

Figure 11.1: MiniProc syntax (left) and a MiniProc program (right). Non-terminals are uppercase, and keywords lowercase. Parts in square brackets are optional, and ellipses mean that the enclosing group can be repeated zero or more times. An IDENTIFIER is any sequence of letters, numbers, or characters ‘.’, ‘:’ and ‘\_’, starting with a letter or an underscore.

- a *collapse* phase, where summary edges collected in the search phase are added to the graph and the resulting new SCCs are collapsed, if any.

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

**Complexity** This algorithm has a worst-case time complexity of  $O(k|\delta||\delta_{push}|^3|\Sigma|)$  and space complexity  $O(|\delta||\delta_{push}|^2|\Sigma|)$ , where  $k$  is the number of SCCs found. We can make the same considerations on the size of  $\Sigma$  as in Section 11.1.1.

### 11.1.3 Modeling Procedural Programs

We use a simple procedural programming language with exceptions called MiniProc, which only admits Boolean variables. Its syntax is shown in Figure 11.1.

A program starts with a variable declaration, which must include all variables used in the program. Then, a sequence of functions are defined, the first one being the entry-point to the program. Function bodies consist of semicolon-separated statements. Assignments, while loops and ifs have the usual semantics. The try-catch statement executes the catch block whenever an exception is thrown by any statement in the try block (or any function it calls). Exceptions are thrown by the throw statement, and they are not typed (i.e., there is no way to distinguish different kinds of exceptions). Functions can be called by prepending their name to the ( ) token (they do not admit arguments, as all variables are global). Since all variables are Boolean, expressions can be composed with the logical and (&&), or (||) and negation (!) operators.

OPA and  $\omega$ OPBA semantically equivalent to a MiniProc program can be generated automatically, both based on OPM  $M_{call}$ . We illustrate their construction through examples. First, an *extended* OPA is generated, in which every state corresponds to some program state, and transitions can be labeled with Boolean expression guards that must be true for them to be performed, or variable assignments. Figure 11.2

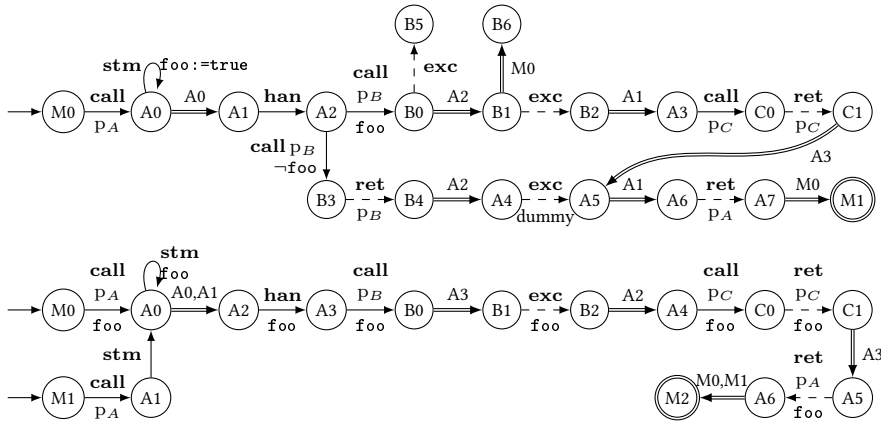


Figure 11.2: Extended OPA (top) and OPA (bottom) generated from the code of Figure 11.1.

shows the extended OPA from the code in Figure 11.1. The stack semantics of the two models coincide: a symbol is pushed for every function call, and popped after the corresponding return (or exception). Handlers are paired to the exception they catch by a shift move updating the same symbol; a dummy exception is placed after the try body to uninstall the handler. The model-checking procedures of the previous sections do not take guards into account, so the extended OPA must be transformed into a normal one. This is done by enumerating all possible Boolean variable assignments for each state, leaving only those that are actually reachable. The resulting OPA for our example is in Figure 11.2.

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

When an OPA is generated, the set of final states only contains the last state of the “main” module (M2) in the example. When an  $\omega$ OPBA is generated, all entry states of functions and loops are marked as final. If the MiniProc programs contains an actual infinite loop, this will result in an accepting loop in the  $\omega$ OPBA. A stuttering state is also added at the end of the  $\omega$ OPBA, so that finite behaviors can be modeled too.

## 11.2 Experimental Evaluation

We checked with POMC several requirements on three case studies which we modeled both manually as OPA and in MiniProc. We report the results on the former in Table 11.1, and on the latter in Table 11.2. Some additional formulas we checked on a MiniProc program are in Table 11.4. Since POMC can generate both OPA and  $\omega$ OPBA from MiniProc, we repeated the experiments from Table 11.4 for the  $\omega$ -word case and we report them respectively in Table 11.5. We also checked a MiniProc benchmark targeted at  $\omega$ OPBA, and its results are in Table 11.3. The results from Table 11.1 can be reproduced through a publicly available artifact [57].

All experiments except those in Table 11.3 were executed on a laptop with a 2.2 GHz Intel processor and 15 GiB of RAM, running Ubuntu GNU/Linux 20.04. Those in Table 11.3 were run on a server with a 2.0 GHz AMD CPU and 500 GiB of RAM. In

Table 11.1: Results of the evaluation of hand-made OPAs. ‘# states’ refers to the OPA to be verified.

#	Benchmark name	# states	Time (ms)	Memory (KiB)		Result
				Total	MC only	
1	generic (Fig. 4.5)	12	1,009	73,632	6,096	True
2	generic medium	24	707	73,671	1,911	False
3	gen. larger (Fig. 11.3)	30	1,214	73,633	9,104	True
4	Jensen	42	289	71,504	1,756	True
5	unsafe stack	63	1,332	71,482	21,095	False
6	safe stack	77	596	71,480	3,979	True
7	unsafe stack neutrality	63	4,821	209,981	83,850	True
8	safe stack neutrality	77	787	71,486	8,864	True

Table 11.2: Results of the evaluation of MiniProc programs, automatically transformed into OPAs. ‘# states’ refers to the OPA to be verified.

#	Benchmark name	# states	Time (ms)	Memory (KiB)		Result
				Total	MC only	
1	generic (Fig. 4.5)	19	1,028	71,493	7,009	True
2	generic medium	31	743	71,490	2,138	False
3	gen. larger (Fig. 11.3)	44	1,315	71,487	8,125	True
4	Jensen	1236	1,839	71,489	17,571	True
5	unsafe stack	162	2,869	88,394	33,990	False
6	safe stack	340	11,572	523,531	207,545	True
7	unsafe stack neutrality	162	12,670	468,025	197,892	True
8	safe stack neutrality	340	18,474	760,313	312,682	True

```

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

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

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

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

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

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

perr() {}

```

Figure 11.3: “Generic larger” MiniProc program.

the tables, by “Total” memory we mean the maximum resident memory including the Haskell runtime (which allocates 70 MiB by default), and by “MC only” the maximum memory used by model checking as reported by the runtime. Since model checking is polynomial in OPA size and exponential in formula length, we focus on checking a variety of requirements, rather than large OPA.

**Generic procedural program** We checked formula

$$\square ((\text{call} \wedge p_B \wedge \text{Scall}(\top, p_A)) \implies \text{CallThr}(\top))$$

from Section 8.3 on the OPA of Figure 4.5 (benchmark 1), and also against two larger OPA (2, where the property does not hold, and 3, where it holds).

We also checked the largest of such MiniProc programs, shown in Figure 11.3, against a set of formulas devised with the purpose of testing all POTL operators. The results are reported in Table 11.4 for the finite-word case, and in Table 11.5 for the  $\omega$ -word case.

**Stack Inspection** The security framework of the Java Development Kit (JDK) is based on stack inspection, i.e. the analysis of the contents of the program’s stack during the execution. The JDK provides method `checkPermission(perm)` from class `AccessController`, which searches the stack for frames of functions that have not been granted permission `perm`. If any are found, an exception is thrown. Such permission checks prevent the execution of privileged code by unauthorized parts of the program, but they must be placed in sensitive points manually. Failure to place them appropriately may cause the unauthorized execution of privileged code. An automated tool to check that no code can escape such checks is thus desirable. Any such tool would need the ability to model exceptions, as they are used to avoid code execution in case of security violations.

[102] explains such needs by providing an example Java program for managing a bank account. It allows the user to check the account balance, and to withdraw money. To perform such tasks, the invoking program must have been granted permissions

CanPay and Debit, respectively. We modeled such program as an OPA (4), and proved that the program enforces such security measures effectively by checking it against the formula

$$\Box(\text{call} \wedge \text{read} \implies \neg(\top \mathcal{S}_X^d(\text{call} \wedge \neg\text{CanPay} \wedge \neg\text{read})))$$

meaning that the account balance cannot be read if some function in the stack lacks the CanPay permission (a similar formula checks the Debit permission).

**Exception Safety** [153] is a tutorial on how to make exception-safe generic containers in C++. It presents two implementations of a generic stack data structure, parametric on the element type T. The first one is not exception-safe: if the constructor of T throws an exception during a pop action, the topmost element is removed, but it is not returned, and it is lost. This violates the strong exception safety requirement that each operation is rolled back if an exception is thrown. The second version of the data structure instead satisfies such requirement.

While exception safety is, in general, undecidable, we can prove the stronger requirement that each modification to the data structure is only committed once no more exceptions can be thrown. We modeled both versions as OPA, and checked such requirement with the following formula:

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

POMC successfully found a counterexample for the first implementation (5), and proved the safety of the second one (6).

Additionally, we proved that both implementations are *exception neutral* (7, 8), i.e. Stack functions do not block exceptions thrown by the underlying type T. This was accomplished by checking the following formula:

$$\Box(\text{exc} \wedge \ominus^u T \wedge \chi_P^d(\text{han} \wedge \chi_P^d \text{Stack}) \implies \chi_P^d \chi_P^d \chi_F^u \text{exc}).$$

**QuickSort** To test the  $\omega$ -word model checking algorithms in particular, we adapted a benchmark from the suite packaged with Moped [74, 107], a tool for LTL model checking of PDS's and Boolean programs. This benchmark consists of an implementation of the QuickSort sorting algorithm [96] in Java, which we adapted to exceptions. The QuickSort procedure `qs()` receives an array of objects to be sorted in input, which may contain null references. If one of them is read by the procedure, it throws a `NullPointerException`, potentially terminating the program. Thus, we modified this benchmark as follows: procedure `qs()` is first called in a `try-catch` block. If it throws an exception, an input-sanitizing procedure named `parseList()` is called, which removes null references from the array. Then, `qs()` is called on the new array.

We checked several properties, and we report the results in Table 11.3.

- Property Q.1 states that the algorithm always terminates normally and Q.2 that the array is correctly sorted at the end. They are false because `qs()` might throw other kinds of exceptions even after null references have been removed, so the program may terminate exceptionally before the array is sorted.

Table 11.3: Results of verification of the QuickSort benchmark (188456  $\omega$ OPBA states). The abbreviations are: R = Results, T = True, F = False.

#	Formula	Time (s)	R
Q.1	$\chi_F^u(\mathbf{ret} \wedge \mathbf{main})$	289	F
Q.2	$\chi_F^u(\neg(\mathbf{aleftGTaright}))$	277	F
Q.3	$\Box((\mathbf{call} \wedge \mathbf{main}) \implies \neg(\bigcirc^u \mathbf{exc} \vee \chi_F^u \mathbf{exc}))$	246	F
Q.4	$\Box((\mathbf{call} \wedge \mathbf{qs}) \implies \neg(\bigcirc^u \mathbf{exc} \vee \chi_F^u \mathbf{exc}))$	247	F
Q.5	$(\bigcirc^u \mathbf{exc} \vee \chi_F^u \mathbf{exc}) \implies$ $(\bigcirc^u \mathbf{exc} \wedge \mathbf{hasParsed}) \vee (\chi_F^u \mathbf{exc} \wedge \mathbf{hasParsed})$	19,617	T
Q.6	$(\bigcirc^u \mathbf{exc} \vee \chi_F^u \mathbf{exc}) \implies$ $(\bigcirc^u \mathbf{exc} \wedge \neg \mathbf{aleftGTaright}) \vee (\chi_F^u \mathbf{exc} \wedge \neg \mathbf{aleftGTaright})$	387	F
Q.7	$\Box((\mathbf{call} \wedge \mathbf{accessValues}) \implies \mathbf{hasParsed} \vee (\top \mathcal{S}_\chi^d \mathbf{han}))$	446	T
Q.8	$(\diamond(\mathbf{ret} \wedge \mathbf{main})) \vee (\chi_F^u(\mathbf{exc} \wedge \mathbf{hasParsed}))$	1,124	T
Q.9	$(\chi_F^u(\mathbf{ret} \wedge \mathbf{main})) \vee (\chi_F^u(\mathbf{exc} \wedge \mathbf{hasParsed}))$	12,809	T
Q.10	$(\diamond(\Box \neg \mathbf{aleftGTaright})) \vee (\chi_F^u(\mathbf{exc} \wedge \mathbf{hasParsed}))$	1,615	T
Q.11	$(\chi_F^u(\neg \mathbf{aleftGTaright})) \vee (\chi_F^u(\mathbf{exc} \wedge \mathbf{hasParsed}))$	12,736	T
Q.12	$(\diamond(\mathbf{ret} \wedge \mathbf{main} \wedge \neg \mathbf{aleftGTaright})) \vee (\chi_F^u(\mathbf{exc} \wedge \mathbf{hasParsed}))$	2,247	T

- Q.3 and Q.4 check whether the `main` and `qs()` functions satisfy the *no-throw* guarantee, and POMC correctly finds out that they might be terminated by exceptions.
- Q.5 verifies that the program can be terminated by an exception only if the second call to `qs()` throws, which means that the array has been sanitized and the exception is not a `NullPointerException`.
- Q.6 verifies whether the array is sorted if the program is terminated by an exception, which is false because if `qs()` throws, it might not have finished sorting the array.
- Q.7 is a stack-inspection property that verifies that whenever an exception is thrown, either there is a handler in the stack (so we are in the first call to `qs()`), or `parseList()` has already been called (so we are in the second call), and hence the exception is, again, not a `NullPointerException`.
- Q.8 and Q.9 check in two equivalent ways that the program always terminates, either normally or by an exception from the second call to `qs()`.
- Similarly, Q.10 and Q.11 check that the only reason for the array not to be sorted when the program terminates is an exception thrown by the second call to `qs()`.
- Finally, Q.12 verifies Q.10 and Q.12 together.

These—and more—results have been published in [140].

### 11.2.1 Discussion

We ran POMC on several case studies and examples. Model checking on hand-made OPAs runs in at most a few seconds, and with a modest memory occupancy, as shown in Table 11.1. In Table 11.2, when checking the same case studies by using automatically-generated OPAs as models, the execution times increase significantly due to

the larger size of generated OPAs, which is consistent with the reachability algorithm having a super-linear (but still polynomial) computational complexity. The time and memory requirements remain, however, reasonable. The same can be said about the numerous formulas that we check in Table 11.4, most of which take less than one second, except a few outliers, which highlight the fact that the process is exponential in formula length.

Finally, in Table 11.5 we saw the behavior of the  $\omega$ OPBA emptiness algorithms on the same formulas as in Table 11.4. In this case, the execution times increase significantly, due to the higher complexity of finding SCCs instead of just checking reachability. However, the time taken by most formulas remains of at most a few seconds, and the same can be said for memory occupancy. There are a few outliers, this time more than in the finite-word case, and one of them even runs out of memory. Again, this is a symptom of the worst-case complexity of the problem, which manifests itself with longer formulas and with hierarchical operators in particular. We did not try the case studies from Table 11.2 as  $\omega$ OPBA, because the properties we check do not make sense in the  $\omega$ -word case.

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

Table 11.4: Results of the additional experiments on a MiniProc program equivalent to OPA “generic larger”. The program has been automatically translated into an OPA with 44 states. The abbreviations are: M. T. = Total Memory, M. MC = Memory for Model Checking only, R = Results, T = True, F = False.

Formula	Time (ms)	M. T. (MiB)	M. MC (KiB)	R
$\chi_F^d p_{Err}$	0.9	70	160	F
$\circ^d(\circ^d(\mathbf{call} \wedge \chi_F^u \mathbf{exc}))$	25.9	70	870	F
$\circ^d(\mathbf{han} \wedge (\chi_F^d(\mathbf{exc} \wedge \chi_P^u \mathbf{call})))$	45.6	70	1,354	F
$\square(\mathbf{exc} \implies \chi_P^u \mathbf{call})$	12.1	70	599	T
$\top \mathcal{U}_X^d \mathbf{exc}$	2.0	70	141	F
$\circ^d(\circ^d(\top \mathcal{U}_X^d \mathbf{exc}))$	4.4	70	119	F
$\square((\mathbf{call} \wedge p_A \wedge (\neg \mathbf{ret} \mathcal{U}_X^d \text{WRx})) \implies \chi_F^u \mathbf{exc})$	5,388.9	121	49,135	T
$\circ^d(\circ^u \mathbf{call})$	0.5	70	105	F
$\circ^d(\circ^d(\circ^d(\circ^u \mathbf{call})))$	3.2	70	145	F
$\chi_F^d(\circ^d(\circ^u \mathbf{call}))$	1.4	70	148	F
$\square((\mathbf{call} \wedge p_A \wedge \mathit{CallThr}(\top)) \implies \mathit{CallThr}(e_B))$	13,119.2	200	80,975	F
$\diamond(\circ_H^d p_B)$	2.4	70	120	F
$\diamond(\circ_H^u p_B)$	3.4	70	120	F
$\diamond(p_A \wedge (\mathbf{call} \mathcal{U}_H^d p_C))$	599.0	70	16,547	T
$\diamond(p_C \wedge (\mathbf{call} \mathcal{S}_H^d p_A))$	778.6	70	17,305	T
$\square((p_C \wedge \chi_F^u \mathbf{exc}) \implies (\neg p_A \mathcal{S}_H^d p_B))$	134,494.0	5,920	2,641,030	F
$\square(\mathbf{call} \wedge p_B \implies \neg p_C \mathcal{U}_H^u p_{Err})$	175.8	70	7,226	T
$\diamond(\circ_H^u p_{Err})$	1.3	70	125	F
$\diamond(\circ_H^u p_{Err})$	1.4	70	124	F
$\diamond(p_A \wedge (\mathbf{call} \mathcal{U}_H^u p_B))$	11.2	70	117	F
$\diamond(p_B \wedge (\mathbf{call} \mathcal{S}_H^u p_A))$	11.9	70	117	F
$\square(\mathbf{call} \implies \chi_F^d \mathbf{ret})$	3.5	70	115	F
$\square(\mathbf{call} \implies \neg \circ^u \mathbf{exc})$	2.5	70	115	F
$\square(\mathbf{call} \wedge p_A \implies \neg \mathit{CallThr}(\top))$	150.0	70	2,997	F
$\square(\mathbf{exc} \implies \neg(\circ^u(\mathbf{call} \wedge p_A) \vee \chi_P^u(\mathbf{call} \wedge p_A)))$	30.7	70	119	F
$\square((\mathbf{call} \wedge p_B \wedge (\mathbf{call} \mathcal{S}_X^d(\mathbf{call} \wedge p_A))) \implies \mathit{CallThr}(\top))$	1,242.5	70	8,143	T
$\square(\mathbf{han} \implies \chi_F^u \mathbf{ret})$	20.4	70	659	T
$\top \mathcal{U}_X^u \mathbf{exc}$	7.0	70	137	T
$\circ^d(\circ^d(\top \mathcal{U}_X^u \mathbf{exc}))$	57.2	70	1,380	T
$\circ^d(\circ^d(\circ^d(\top \mathcal{U}_X^u \mathbf{exc})))$	196.1	70	2,939	T
$\square(\mathbf{call} \wedge p_C \implies (\top \mathcal{U}_X^u \mathbf{exc} \wedge \chi_P^d \mathbf{han}))$	103.7	70	863	F
$\mathbf{call} \mathcal{U}_X^d(\mathbf{ret} \wedge p_{Err})$	1.8	70	117	F
$\chi_F^d(\mathbf{call} \wedge ((\mathbf{call} \vee \mathbf{exc}) \mathcal{S}_X^u p_B))$	9.9	70	116	F
$\circ^d(\circ^d((\mathbf{call} \vee \mathbf{exc}) \mathcal{U}_X^u \mathbf{ret}))$	6.2	70	116	F

Table 11.5: Results of the additional experiments on the same MiniProc program of Table 11.4, but interpreted as a continuously running program. The program has been automatically translated into an  $\omega$ OPBA with 44 states. The abbreviations are: M. T. = Total Memory, M. MC = Memory for Model Checking only, R = Results, T = True, F = False, O = Out of memory.

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

**Part IV**

**Epilogue**



## Chapter 12

# Conclusions and Future Work

### 12.1 Discussion and Contributions

Temporal logic was identified in the 1980s as the formalism of choice for expressing system specifications and requirements with the aim of automated verification. The main advantage of temporal logic is the good balance between expressiveness and efficiency of verification algorithms. Indeed, logics such as LTL and CTL formalize naturally concepts related to the flow of time and events, as well as the evolution of a system as a succession of discrete states. Also, they admit relatively efficient verification algorithms, as we saw in Chapter 2. However, the increasing variety of systems and environments in which verification has been applied have often highlighted the expressive limitations of those logics, motivating efforts aimed at defining more expressive versions of them. The work presented in this thesis is part of such an effort, and it is mainly motivated by the verification of procedural programs.

As we highlighted in Chapter 3, the fact that procedures are managed by means of a stack causes programs to present inherently context-free behaviors. The need of specifying complex properties that take into account the nesting structure of procedure calls and returns was tackled by logics on VPLs and nested words, namely CaRet and NWTL. However, nested words only offer a one-to-one matching relation, which is not enough to model behaviors of certain programming constructs, such as exceptions, that are considerably widespread in modern programming languages, and that create the need for specifying requirements that rule out some of their unwanted side-effects.

We closed this gap by presenting temporal logics featuring temporal modalities that explicitly consider such context-free behaviors, and by basing their semantic on OPLs, a subclass of DCFLs as robust as VPLs, but of much wider expressiveness.

**OPTL** The first logic that we introduced is OPTL. OPTL is capable of expressing, among others, requirements on total and partial correctness (i.e., Hoare-style pre- and post-conditions), exception safety, and stack inspection in the presence of exceptions—thus, stating requirements that forbid or enforce exceptions when certain procedures are active. Moreover, we proved that OPTL is strictly more expressive than logics on VPLs such as NWTL thanks to its being based on OPLs.

However, the motivations for introducing OPTL were not only related to its possible practical applications, but also to the more theoretical question of how far it

is possible to push temporal logics in terms of expressive power with respect to the Chomsky hierarchy. When it comes to measuring the expressiveness of temporal logics, the most conventional yardstick is “classical” logics such as FOL and MSOL. Unfortunately, when investigating the stance of OPTL against FOL, we found out that OPTL is strictly *less* expressive: there are FO-expressible properties on OP words that cannot be expressed by means of an OPTL formula. This might seem surprising, as we proved that OPTL can express all properties that can be stated in NWTL, which in turn is FO-complete. Indeed, this is another symptom of the greater expressive power of OPLs with respect to VPLs, and consequently of OP words with respect to nested words. Thus, OPTL is expressive enough to represent all FO-expressible properties on nested words, but not enough to tackle the greater generality of OPLs.

This is one of the reasons why we decided to present OPTL in this thesis, as we consider it an important step towards the development of expressively complete logics for OPLs. The other reason is its incompleteness proof. Indeed, most proofs concerning the expressiveness of temporal logics that appear in the literature usually state their equivalence to FOL, MSOL or a fragment thereof—see e.g. the seminal [106], and also [12, 24, 132]. Ours is instead a proof that no formula expressible in OPTL is equivalent to a specific one in FOL. This is a non-trivial endeavor when it comes to logics with a more complex syntax. In fact, the same question regarding the relationship between CaRet and NWTL has yet to be answered, although CaRet is conjectured not to be as expressive as FOL on nested words [12]. Our proof uses a novel technique, based on a *Pumping Lemma* for temporal logics, which may be general enough to be applicable to other contexts, such as the aforementioned CaRet vs. NWTL issue.

**POTL** We introduced POTL with the main intent of having a FO-complete temporal logic based on OPLs. We reached this goal, and the resulting proof is non-trivial, as it uses rather sophisticated techniques from model theory. The complexity of this proof further highlights the generality of OPLs, because it stems from the need for combining the various shapes that OP  $\omega$ -words can exhibit, which we achieved in a non-trivial way. As a corollary, we proved that OP words and  $\omega$ -words have the three-variable property.

Note that the very fact that POTL is based on OPLs makes this FO-completeness result even more important than the analogous one for LTL [106]. Indeed, the FO-definable subset of OPLs is equivalent to non-counting or aperiodic OPLs [122, 123], as well as the FO-definable subset of regular languages is equivalent to their non-counting subset [129]. The definition of aperiodicity for OPLs excludes languages that pose counting requirements on the nesting structure of words. In the context of program verification, this would result in the inexpressibility of requirements such as “procedure A must be present on the stack in an even number of instances”, which however seem to be of scarce practical interest. On the contrary, many FSAs formalize counting devices.

POTL’s greater expressiveness is, nonetheless, not only theoretical, but also arises when trying to express requirements. POTL is in fact capable of expressing all OPTL-expressible properties, often in an easier and more natural way. In particular, the ability of its operators to constrain their semantics to a single subtree in a word’s syntax tree—i.e., on a single function frame in the program verification context—enables the specification of many requirements in which OPTL fails. For example, function-local requirements are easily expressible in POTL and, as an exercise, we also succeeded in giving a direct POTL translation of LTL.

**Model Checking** We studied POTL model checking both theoretically and practically. We developed an automata-theoretic model checking procedure for POTL, which involves the construction of automata that accept models of POTL formulas. Such a construction is much more involved than that of simpler logics such as LTL, mainly because of the higher complexity of the underlying automata: OPAs have a stack, which must be used carefully to keep track of temporal obligations. From this, we derive bounds on the computational complexity of POTL model checking and satisfiability, which are both EXPTIME-complete. While such bounds are higher than that of LTL model checking, which is PSPACE-complete, they match those of NWTTL [12]. Thus, POTL does not pay its greater expressiveness with higher computational complexity. Overall, the model checking algorithm has time exponential in formula length, and polynomial in model size, just like LTL. The only drawback comes from the more complex modeling formalism. While model checking on transition systems can be linear in model size, when using pushdown automata such as OPAs, NWAs or RSMs the complexity of the emptiness-checking algorithms becomes cubic in (parts of) model size.

To see how such algorithms behave in practice, we implemented a POTL model checker, called POMC. We ran it on several case studies, checking interesting properties for real-world scenarios on models of varying size, both hand-made and automatically generated from a simplified programming language. Most formulas are checked in just a few seconds, with only a few outliers. The presence of such outliers was mostly related to formula length, and not to the model to be checked. This is unsurprising, considering the worst-case complexity of the procedure. We tried model checking both on finite and  $\omega$ -words, finding out that the increased overhead of the more complex emptiness-checking algorithms for the infinite-word case is in most cases acceptable or negligible. Overall, we can state that the results we obtained are promising, and pave the way for further applications of POTL model checking.

## 12.2 Future Work Directions

Here we survey some of the directions for future research that are opened by the work in this thesis.

**Application domains** Throughout this thesis, we mostly focused on the verification of procedural programs with exceptions, both in the examples we used to illustrate the concepts we introduced, and in the experimental part. However, the generality of OPLs and POTL is much greater: it is possible to define a virtually infinite number of verification frameworks by simply picking a different choice of the OPM. Thus, POTL could be used to specify requirements in the presence of other programming constructs, such as continuations. Moreover, any system involving a stack may exhibit context-free behaviors that may benefit from the use of POTL for formal verification. Just to give some examples, we mention system interrupts, log-based systems such as databases, and version control systems.

**Better program modeling** In Chapter 11, we modeled programs both by hand and by generating them from a very simple programming language. In both cases, we only admitted Boolean variables. Of course, however, real-world programs come with much richer type systems and data representations. Thus, to automatically verify real-world programs more sophisticated techniques are needed. One possibility would

be to apply already-existing abstraction techniques that have already been applied successfully to software model checking. We could employ *predicate abstraction* [103, 104] and related *abstraction refinement* techniques [62], and possibly adapt them for the kind of requirements expressible by POTL.

**More efficient model representation** While explicit-state model checking has its own merits, it suffers from the *state-space explosion problem* [17], which in our case is exacerbated by the non-linear complexity of emptiness-checking algorithms for pushdown-based systems. In the context of model checking logics such as LTL, this problem has been tackled by introducing more compact and efficient model representations, such as *binary decision diagrams* [105]. The resulting verification technique is called *symbolic model checking* [43]. Thus, the development of similar techniques could be explored for OPAs and other stack-based automata, although the presence of the stack could make this rather difficult.

**Bounded Model Checking** Bounded model checking [26] is an efficient model checking technique that exploits the recent advances in SAT and SMT solvers. It consists in encoding the model checking problem as formulas in propositional logic (in the SAT case) possibly enriched with decidable theories (in the SMT case), and then using an efficient solver to check satisfiability of the resulting formula. This approach has the drawback that properties can be checked only up to a certain temporal bound (although this is not always the case [15]), but its notable efficiency has determined its success. A propositional or SMT-based encoding of OP words could be attempted, to investigate the possible benefits of this approach to POTL model checking.

**User-Friendliness** Even if we extensively illustrated POTL's capabilities in terms of ease of expressing requirements, we admit that its full comprehension requires a somewhat solid mathematical background, which is not always common in end-users, and could discourage its adoption. We note, however, that this issue is not restricted to POTL in particular, but it is shared with temporal logics and formal methods in general [120], including LTL [14] and NWTL. One way to tackle this problem would be to offer simple, graphical-based frameworks for expressing domain-specific properties, possibly based on UML [81]. An orthogonal solution would be to gather a catalog of useful requirements for a specific domain (e.g., as we did up to a limited extent in Sections 6.4 and 8.3), that use POTL formulas only as a backend.

**Extensions** Although we argue that the FO-expressible fragment of OPLs is enough for most applications, several ways of further extending POTL's expressive power could be investigated.

A logic capturing the whole OPLs could be devised by embedding regular-expression-like constructs in it, as was done with LTL [148] and logics on VPLs [33]. In this respect, a variant of regular expressions for OPLs has been recently introduced [122].

Timed or metric variants of POTL could be devised, following the steps already done for VPL logics [37]. Weighted variants could also be an option, fostered by the recent introduction of weighted OPLs [70].

Finally, ways of modeling and checking concurrent procedural programs could be investigated. This would most likely require multi-stack automata for model checking. While multi-pushdown automata tend to have decidability issues [144], model checking has been studied for stack-bounded variants [20, 41, 46, 141, 157].

# Bibliography

- [1] David Abrahams. Exception-safety in generic components. In *Generic Programming*, volume 1766 of *LNCS*, pages 69–79. Springer, 1998. doi:10.1007/3-540-39953-4\_6.
- [2] Rajeev Alur. Marrying words and trees. In *PODS '07*, pages 233–242. ACM, 2007. doi:10.1145/1265530.1265564.
- [3] Rajeev Alur and Dana Fisman. Colored nested words. In *LATA 2016*, volume 9618 of *LNCS*, pages 143–155. Springer, 2016. doi:10.1007/978-3-319-30000-9\_11.
- [4] Rajeev Alur and Parthasarathy Madhusudan. Visibly pushdown languages. In *STOC '04*, pages 202–211. ACM, 2004. doi:10.1145/1007352.1007390.
- [5] Rajeev Alur and Parthasarathy Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3), 2009. doi:10.1145/1516512.1516518.
- [6] Rajeev Alur and Mihalis Yannakakis. Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst.*, 23(3):273–303, 2001. doi:10.1145/503502.503503.
- [7] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Analysis of recursive state machines. In *CAV '01*, volume 2102 of *LNCS*, pages 207–220. Springer, 2001. doi:10.1007/3-540-44585-4\_18.
- [8] Rajeev Alur, Kousha Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS '04*, volume 2988 of *LNCS*, pages 467–481. Springer, 2004. doi:10.1007/978-3-540-24730-2\_35.
- [9] Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, and Mihalis Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005. doi:10.1145/1075382.1075387.
- [10] Rajeev Alur, Swarat Chaudhuri, Kousha Etessami, and Parthasarathy Madhusudan. On-the-fly reachability and cycle detection for recursive state machines. In *TACAS '05*, volume 3440 of *LNCS*, pages 61–76. Springer, 2005. doi:10.1007/978-3-540-31980-1\_5.
- [11] Rajeev Alur, Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Congruences for visibly pushdown languages. In *ICALP '05*, volume 3580 of *LNCS*, pages 1102–1114. Springer, 2005. doi:10.1007/11523468\_89.

- [12] Rajeev Alur, Marcelo Arenas, Pablo Barceló, Kousha Etessami, Neil Immerman, and Leonid Libkin. First-order and temporal logics for nested words. *LMCS*, 4(4), 2008. doi:10.2168/LMCS-4(4:11)2008.
- [13] Rajeev Alur, Swarat Chaudhuri, and Parthasarathy Madhusudan. Software model checking using languages of nested trees. *ACM Trans. Program. Lang. Syst.*, 33(5):15:1–15:45, 2011. doi:10.1145/2039346.2039347.
- [14] Mehrnoosh Askarpour and Marcello M. Bersani. Teaching formal methods: An experience report. In *FISEE 2019*, volume 12271 of *LNCS*, pages 3–18. Springer, 2019. doi:10.1007/978-3-030-57663-9\_1.
- [15] Mohammad Awedh and Fabio Somenzi. Termination criteria for bounded model checking: Extensions and comparison. In *BMC '05*, volume 144(1) of *ENTCS*, pages 51–66. Elsevier, 2006. doi:10.1016/j.entcs.2005.07.019.
- [16] Jos C. M. Baeten, Jan A. Bergstra, and Jan Willem Klop. Decidability of bisimulation equivalence for processes generating context-free languages. *J. ACM*, 40(3):653–682, 1993. doi:10.1145/174130.174141.
- [17] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008. ISBN 978-0-262-02649-9.
- [18] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 2000*, volume 1885 of *LNCS*, pages 113–130. Springer, 2000. doi:10.1007/10722468\_7.
- [19] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *CAV '01*, volume 2102 of *LNCS*, pages 260–264. Springer, 2001. doi:10.1007/3-540-44585-4\_25.
- [20] Kshitij Bansal and Stéphane Demri. Model-checking bounded multi-pushdown systems. In *CSR '13*, volume 7913 of *LNCS*, pages 405–417. Springer, 2013. doi:10.1007/978-3-642-38536-0\_35.
- [21] Alessandro Barenghi, Stefano Crespi Reghizzi, Dino Mandrioli, Federica Panella, and Matteo Pradella. Parallel parsing made practical. *Sci. Comput. Program.*, 112:195–226, 2015. doi:10.1016/j.scico.2015.09.002.
- [22] Gérard Basler, Daniel Kroening, and Georg Weissenbacher. SAT-based summarization for Boolean programs. In *SPIN '07*, volume 4595 of *LNCS*, pages 131–148. Springer, 2007. doi:10.1007/978-3-540-73370-6\_10.
- [23] Mordechai Ben-Ari. *Mathematical Logic for Computer Science, 3rd Edition*. Springer, London, UK, 2012. ISBN 978-1-4471-4129-7. doi:10.1007/978-1-4471-4129-7.
- [24] Michael Benedikt and Clemens Ley. Limiting until in ordered tree query languages. *ACM Trans. Comput. Log.*, 17(2):14:1–14:34, 2016. doi:10.1145/2856104.
- [25] Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Inf. Control.*, 60(1-3):109–137, 1984. doi:10.1016/S0019-9958(84)80025-X.

- [26] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS '99*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999. doi:10.1007/3-540-49059-0\_14.
- [27] Ahmed Bouajjani and Peter Habermehl. Constrained properties, semilinear systems, and Petri nets. In *CONCUR '96*, volume 1119 of *LNCS*, pages 481–497. Springer, 1996. doi:10.1007/3-540-61604-7\_71.
- [28] Ahmed Bouajjani, Rachid Echahed, and Peter Habermehl. On the verification problem of nonregular properties for nonregular processes. In *LICS '95*, pages 123–133. IEEE Computer Society, 1995. doi:10.1109/LICS.1995.523250.
- [29] Ahmed Bouajjani, Rachid Echahed, and Peter Habermehl. Verifying infinite state processes with sequential and parallel composition. In *POPL '95*, pages 95–106. ACM Press, 1995. doi:10.1145/199448.199470.
- [30] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: application to model-checking. In *CONCUR '97*, volume 1243 of *LNCS*, pages 135–150. Springer, 1997. doi:10.1007/3-540-63141-0\_10.
- [31] Laura Bozzelli. Alternating automata and a temporal fixpoint calculus for visibly pushdown languages. In *CONCUR '07*, volume 4703 of *LNCS*, pages 476–491. Springer, 2007. doi:10.1007/978-3-540-74407-8\_32.
- [32] Laura Bozzelli and César Sánchez. Visibly rational expressions. *Acta Informatica*, 51(1):25–49, 2014. doi:10.1007/s00236-013-0190-6.
- [33] Laura Bozzelli and César Sánchez. Visibly linear temporal logic. *J. Autom. Reason.*, 60(2):177–220, 2018. doi:10.1007/s10817-017-9410-z.
- [34] Laura Bozzelli, Salvatore La Torre, and Adriano Peron. Verification of well-formed communicating recursive state machines. *Theor. Comput. Sci.*, 403(2-3): 382–405, 2008. doi:10.1016/j.tcs.2008.06.012.
- [35] Laura Bozzelli, Aniello Murano, and Adriano Peron. Pushdown module checking. *Formal Methods Syst. Des.*, 36(1):65–95, 2010. doi:10.1007/s10703-010-0093-x.
- [36] Laura Bozzelli, Aniello Murano, and Adriano Peron. Event-clock nested automata. In *LATA '18*, volume 10792 of *LNCS*, pages 80–92. Springer, 2018. doi:10.1007/978-3-319-77313-1\_6.
- [37] Laura Bozzelli, Aniello Murano, and Adriano Peron. Timed context-free temporal logics. In *GandALF '18*, volume 277 of *EPTCS*, pages 235–249. Open Publishing Association, 2018. doi:10.4204/EPTCS.277.17.
- [38] Laura Bozzelli, Angelo Montanari, and Adriano Peron. Interval temporal logic for visibly pushdown systems. In *FSTTCS '19*, volume 150 of *LIPICs*, pages 33:1–33:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.FSTTCS.2019.33.
- [39] Julian C. Bradfield and Colin Stirling. Modal mu-calculi. In *Handbook of Modal Logic*, volume 3 of *Studies in logic and practical reasoning*, pages 721–756. North-Holland, 2007. doi:10.1016/s1570-2464(07)80015-2.

- [40] Julian C. Bradfield and Igor Walukiewicz. The mu-calculus and model checking. In *Handbook of Model Checking*, pages 871–919. Springer, 2018. doi:10.1007/978-3-319-10575-8\_26.
- [41] Luca Breveglieri, Alessandra Cherubini, Claudio Citrini, and Stefano Crespi-Reghizzi. Multi-push-down languages and grammars. *Int. J. Found. Comput. Sci.*, 7(3):253–292, 1996. doi:10.1142/S0129054196000191.
- [42] J. Richard Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science (LMPS '60)*, pages 1–11. Stanford University Press, 1962.
- [43] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking:  $10^{20}$  states and beyond. *Inf. Comput.*, 98(2):142–170, 1992. doi:10.1016/0890-5401(92)90017-A.
- [44] Olaf Burkart and Bernhard Steffen. Model checking for context-free processes. In *CONCUR '92*, volume 630 of *LNCS*, pages 123–137. Springer, 1992. doi:10.1007/BFb0084787.
- [45] Olaf Burkart and Bernhard Steffen. Model checking the full modal mu-calculus for infinite sequential processes. *Theor. Comput. Sci.*, 221(1-2):251–270, 1999. doi:10.1016/S0304-3975(99)00034-1.
- [46] Dario Carotenuto, Aniello Murano, and Adriano Peron. Ordered multi-stack visibly pushdown automata. *Theor. Comput. Sci.*, 656:1–26, 2016. doi:10.1016/j.tcs.2016.08.012.
- [47] Didier Caucal and Roland Monfort. On the transition graphs of automata and grammars. In *WG '90*, volume 484 of *LNCS*, pages 311–337. Springer, 1990. doi:10.1007/3-540-53832-1\_51.
- [48] Krishnendu Chatterjee, Di Ma, Rupak Majumdar, Tian Zhao, Thomas A. Henzinger, and Jens Palsberg. Stack size analysis for interrupt-driven programs. *Inf. Comput.*, 194(2):144–174, 2004. doi:10.1016/j.ic.2004.06.001.
- [49] Swarat Chaudhuri and Rajeev Alur. Instrumenting C programs with nested word monitors. In *SPIN '07*, volume 4595 of *LNCS*, pages 279–283. Springer, 2007. doi:10.1007/978-3-540-73370-6\_20.
- [50] Feng Chen and Grigore Rosu. Java-MOP: A monitoring oriented programming environment for Java. In *TACAS '05*, volume 3440 of *LNCS*, pages 546–550. Springer, 2005. doi:10.1007/978-3-540-31980-1\_36.
- [51] Patrick Chervet and Igor Walukiewicz. Minimizing variants of visibly push-down automata. In *MFCS '07*, volume 4708 of *LNCS*, pages 135–146. Springer, 2007. doi:10.1007/978-3-540-74456-6\_14.
- [52] Michele Chiari. Temporal logic and model checking for operator precedence words. Master's thesis, Politecnico di Milano, 2018. URL <http://hdl.handle.net/10589/142923>.

- [53] Michele Chiari, Dino Mandrioli, and Matteo Pradella. Temporal logic and model checking for operator precedence languages. In *GandALF 2018*, volume 277 of *EPTCS*, pages 161–175. Open Publishing Association, 2018. doi:10.4204/EPTCS.277.12.
- [54] Michele Chiari, Dino Mandrioli, and Matteo Pradella. Operator precedence temporal logic and model checking. *Theor. Comput. Sci.*, 848:47–81, 2020. doi:10.1016/j.tcs.2020.08.034.
- [55] Michele Chiari, Davide Bergamaschi, and Francesco Pontiggia. POMC, 2021. URL <https://github.com/michiari/POMC>.
- [56] Michele Chiari, Dino Mandrioli, and Matteo Pradella. Model-checking structured context-free languages. In *CAV '21*, volume 12760 of *LNCS*, page 387–410. Springer, 2021. doi:10.1007/978-3-030-81688-9\_18.
- [57] Michele Chiari, Dino Mandrioli, and Matteo Pradella. Model-checking structured context-free languages (artifact), 2021. URL <https://doi.org/10.5281/zenodo.4723740>.
- [58] Michele Chiari, Dino Mandrioli, and Matteo Pradella. A first-order complete temporal logic for structured context-free languages. *CoRR*, abs/2105.10740, 2021. URL <https://arxiv.org/abs/2105.10740>.
- [59] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981. doi:10.1007/BFb0025774.
- [60] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986. doi:10.1145/5397.5399.
- [61] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV 2000*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000. doi:10.1007/10722167\_15.
- [62] Edmund M. Clarke, Robert P. Kurshan, and Helmut Veith. The localization reduction and counterexample-guided abstraction refinement. In *Time for Verification, Essays in Memory of Amir Pnueli*, volume 6200 of *LNCS*, pages 61–71. Springer, 2010. doi:10.1007/978-3-642-13754-9\_4.
- [63] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018. ISBN 978-3-319-10574-1. doi:10.1007/978-3-319-10575-8.
- [64] Stefano Crespi Reghizzi and Dino Mandrioli. Operator precedence and the visibly pushdown property. *J. Comput. Syst. Sci.*, 78(6):1837–1867, 2012. doi:10.1016/j.jcss.2011.12.006.
- [65] Stefano Crespi Reghizzi, Michel A. Melkanoff, and Larry Lichten. The use of grammatical inference for designing programming languages. *Commun. ACM*, 16(2):83–90, 1973. doi:10.1145/361952.361958.

- [66] Stefano Crespi Reghizzi, Dino Mandrioli, and Daniel F. Martin. Algebraic properties of operator precedence languages. *Information and Control*, 37(2):115–133, 1978. doi:10.1016/S0019-9958(78)90474-6.
- [67] Mads Dam. CTL\* and ECTL\* as fragments of the modal mu-calculus. *Theor. Comput. Sci.*, 126(1):77–96, 1994. doi:10.1016/0304-3975(94)90269-0.
- [68] Loris D’Antoni. A symbolic automata library, 2014. URL <https://github.com/lorisdanto/symbolicautomata>.
- [69] Evan Driscoll, Aditya V. Thakur, and Thomas W. Reps. OpenNWA: A nested-word automaton library. In *CAV ’12*, volume 7358 of *LNCS*, pages 665–671. Springer, 2012.
- [70] Manfred Droste, Stefan Dück, Dino Mandrioli, and Matteo Pradella. Weighted operator precedence languages. *Inf. Comput.*, page 104658, 2020. doi:10.1016/j.ic.2020.104658.
- [71] Andrzej Ehrenfeucht. An application of games to the completeness problem for formalized theories. *Fund. Math.*, 49:129–141, 1961. doi:10.4064/fm-49-2-129-141.
- [72] E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *J. Comput. Syst. Sci.*, 30(1):1–24, 1985. doi:10.1016/0022-0000(85)90001-7.
- [73] E. Allen Emerson and Charanjit S. Jutla. The complexity of tree automata and logics of programs. *SIAM J. Comput.*, 29(1):132–158, 1999. doi:10.1137/S0097539793304741.
- [74] Javier Esparza and Stefan Schwoon. A BDD-based model checker for recursive programs. In *CAV ’01*, volume 2102 of *LNCS*, pages 324–336. Springer, 2001. doi:10.1007/3-540-44585-4\_30.
- [75] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV 2000*, volume 1855 of *LNCS*, pages 232–247. Springer, 2000. doi:10.1007/10722167\_20.
- [76] Javier Esparza, Antonín Kučera, and Stefan Schwoon. Model checking LTL with regular valuations for pushdown systems. *Information and Computation*, 186(2):355–376, 2003. doi:10.1016/S0890-5401(03)00139-1.
- [77] Kousha Etessami, Moshe Y. Vardi, and Thomas Wilke. First-order logic with two variables and unary temporal logic. *Inf. Comput.*, 179(2):279–295, 2002. doi:10.1006/inco.2001.2953.
- [78] Alain Finkel, Bernard Willems, and Pierre Wolper. A direct symbolic approach to model checking pushdown systems. In *Infinity ’97*, volume 9 of *ENTCS*, pages 27–37. Elsevier, 1997. doi:10.1016/S1571-0661(05)80426-8.
- [79] Michael J. Fischer. Some properties of precedence languages. In *STOC ’69*, pages 181–190. ACM, 1969. doi:10.1145/800169.805432.
- [80] Robert W. Floyd. Syntactic analysis and operator precedence. *J. ACM*, 10(3):316–333, 1963. doi:10.1145/321172.321179.

- [81] International Organization for Standardization. ISO/IEC 15926:2005, Information technology—Open Distributed Processing—Unified Modeling Language (UML) version 1.4.2, 2005. URL <https://www.iso.org/standard/32620.html>.
- [82] Roland Fraïssé. Sur quelques classifications des systèmes de relations. *Publ. Sci. Univ. Alger*, A(1):35–182, 1954.
- [83] Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. *Ann. Pure Appl. Log.*, 130(1-3):3–31, 2004. doi:10.1016/j.apal.2004.01.007.
- [84] Dov M. Gabbay. Expressive functional completeness in tense logic. In *Aspects of Philosophical Logic*, pages 91–117. Springer, 1981. doi:10.1007/978-94-009-8384-7\_4.
- [85] Dov M. Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *POPL '80*, pages 163–173. ACM Press, 1980. doi:10.1145/567446.567462.
- [86] Dov M. Gabbay, Ian Hodkinson, and Mark Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects*, volume 1. Clarendon Press, Oxford, UK, 1994. ISBN 0-19-853769-7.
- [87] Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Inf. Process. Lett.*, 74(3-4):107–114, 2000. doi:10.1016/S0020-0190(00)00051-X.
- [88] Olivier Gauwin, Anca Muscholl, and Michael Raskin. Minimization of visibly pushdown automata is NP-complete. *LMCS*, 16(1), 2020. doi:10.23638/LMCS-16(1:14)2020.
- [89] Patrice Godefroid and Mihalis Yannakakis. Analysis of Boolean programs. In *TACAS '13*, volume 7795 of *LNCS*, pages 214–229. Springer, 2013. doi:10.1007/978-3-642-36742-7\_16.
- [90] Erich Grädel, Phokion G. Kolaitis, Leonid Libkin, Maarten Marx, Joel Spencer, Moshe Y. Vardi, Yde Venema, and Scott Weinstein. *Finite Model Theory and its applications*. Springer, 2007. doi:10.1007/3-540-68804-8.
- [91] Thilo Hafer and Wolfgang Thomas. Computation tree logic CTL\* and path quantifiers in the monadic theory of the binary tree. In Thomas Ottmann, editor, *Automata, Languages and Programming*, pages 269–279. Springer, 1987. ISBN 978-3-540-47747-1.
- [92] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*, pages 99–217. Springer, 2002. ISBN 978-94-017-0456-4. doi:10.1007/978-94-017-0456-4\_2.
- [93] Michael A. Harrison. *Introduction to Formal Language Theory*. Addison Wesley, 1978.
- [94] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *SPIN 2003*, volume 2648 of *LNCS*, pages 235–239. Springer, 2003. doi:10.1007/3-540-44829-2\_17.

- [95] Jaakko Hintikka. Distributive normal forms in the calculus of predicates. *Acta Philosophica Fennica*, 6:71, 1953.
- [96] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321, 1961. doi:10.1145/366622.366644.
- [97] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi:10.1145/363235.363259.
- [98] Wilfrid Hodges. *Model Theory*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, Cambridge, UK, 1993. doi:10.1017/CBO9780511551574.
- [99] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata and Formal Languages*. Addison-Wesley, Reading, MA, 1979.
- [100] Hardi Hungar and Bernhard Steffen. Local model checking for context-free processes. In *ICALP '93*, volume 700 of *LNCS*, pages 593–605. Springer, 1993. doi:10.1007/3-540-56939-1\_105.
- [101] Neil Immerman. *Descriptive complexity*. Graduate texts in computer science. Springer, 1999. ISBN 978-1-4612-6809-3. doi:10.1007/978-1-4612-0539-5.
- [102] Thomas Jensen, Daniel Le Metayer, and Tommy Thorn. Verification of control flow based security properties. In *Proc. '99 IEEE Symp. on Security and Privacy*, pages 89–103, 1999. doi:10.1109/SECPRI.1999.766902.
- [103] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, 2009. doi:10.1145/1592434.1592438.
- [104] Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. Predicate abstraction for program verification. In *Handbook of Model Checking*, pages 447–491. Springer, 2018. doi:10.1007/978-3-319-10575-8\_15.
- [105] Sheldon B. Akers Jr. Binary decision diagrams. *IEEE Trans. Computers*, 27(6): 509–516, 1978. doi:10.1109/TC.1978.1675141.
- [106] Hans Kamp. *Tense logic and the theory of linear order*. PhD thesis, University of California, Los Angeles, 1968.
- [107] Stefan Kiefer, Stefan Schwoon, and Dejavuth Suwimonteerabuth. Moped, version 1.0.16, 2010. URL <http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/moped/>.
- [108] Donald E. Knuth. On the translation of languages from left to right. *Inf. Control.*, 8(6):607–639, 1965. doi:10.1016/S0019-9958(65)90426-2.
- [109] Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983. doi:10.1016/0304-3975(82)90125-6.
- [110] Viraj Kumar, Parthasarathy Madhusudan, and Mahesh Viswanathan. Minimization, learning, and conformance testing of Boolean programs. In *CONCUR '06*, volume 4137 of *LNCS*, pages 203–217. Springer, 2006. doi:10.1007/11817949\_14.

- [111] Orna Kupferman. Automata theory and model checking. In *Handbook of Model Checking*, pages 107–151. Springer, 2018. doi:10.1007/978-3-319-10575-8\_4.
- [112] Orna Kupferman and Moshe Y. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In *CAV 2000*, volume 1855 of *LNCS*, pages 36–52. Springer, 2000. doi:10.1007/10722167\_7.
- [113] Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. Model checking linear properties of prefix-recognizable systems. In *CAV '02*, volume 2404 of *LNCS*, pages 371–385. Springer, 2002. doi:10.1007/3-540-45657-0\_31.
- [114] Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. Pushdown specifications. In *LPAR '02*, volume 2514 of *LNCS*, pages 262–277. Springer, 2002. doi:10.1007/3-540-36078-6\_18.
- [115] Clemens Lautemann, Thomas Schwentick, and Denis Thérien. Logics for context-free languages. In *CSL '94*, pages 205–216, 1994. doi:10.1007/BFb0022257.
- [116] K. Rustan M. Leino. A SAT characterization of Boolean-program correctness. In *SPIN '03*, volume 2648 of *LNCS*, pages 104–120. Springer, 2003. doi:10.1007/3-540-44829-2\_7.
- [117] Leonid Libkin. The finite model theory toolbox of a database theoretician. In *PODS '09*, pages 65–76. ACM, 2009. doi:10.1145/1559795.1559807.
- [118] Leonid Libkin and Cristina Sirangelo. Reasoning about XML with temporal logics and automata. *J. Appl. Log.*, 8(2):210–232, 2010. doi:10.1016/j.jal.2009.09.005.
- [119] Violetta Lonati, Dino Mandrioli, Federica Panella, and Matteo Pradella. Operator precedence languages: Their automata-theoretic and logic characterization. *SIAM J. Comput.*, 44(4):1026–1088, 2015. doi:10.1137/140978818.
- [120] Dino Mandrioli. On the heroism of really pursuing formal methods. In *FormaliSE '15*, pages 1–5. IEEE Computer Society, 2015. doi:10.1109/FormaliSE.2015.8.
- [121] Dino Mandrioli and Matteo Pradella. Generalizing input-driven languages: Theoretical and practical benefits. *Computer Science Review*, 27:61–87, 2018. doi:10.1016/j.cosrev.2017.12.001.
- [122] Dino Mandrioli, Matteo Pradella, and Stefano Crespi Reghizzi. Star-freeness, first-order definability and aperiodicity of structured context-free languages. In *ICTAC '20*, volume 12545, pages 161–180. Springer, 2020. doi:10.1007/978-3-030-64276-1\_9.
- [123] Dino Mandrioli, Matteo Pradella, and Stefano Crespi Reghizzi. Aperiodicity, star-freeness, and first-order definability of structured context-free languages. *CoRR*, abs/2006.01236, 2020. URL <https://arxiv.org/abs/2006.01236>.
- [124] Simon Marlow. Haskell 2010 language report, 2010. URL <https://www.haskell.org/onlinereport/haskell2010/>.
- [125] Maarten Marx. Conditional XPath, the first order complete XPath dialect. In *PODS '04*, page 13. ACM Press, 2004. doi:10.1145/1055558.1055562.

- [126] Maarten Marx. Conditional XPath. *ACM Trans. Database Syst.*, 30(4):929–959, 2005. doi:10.1145/1114244.1114247.
- [127] Christian Mathissen. Weighted logics for nested words and algebraic formal power series. *LMCS*, 6(1), 2010. doi:10.2168/LMCS-6(1:5)2010.
- [128] Robert McNaughton. Parenthesis grammars. *J. ACM*, 14(3):490–500, 1967. doi:10.1145/321406.321411.
- [129] Robert McNaughton and Seymour Papert. *Counter-free Automata*. MIT Press, Cambridge, USA, 1971. ISBN 9780262130769.
- [130] Kurt Mehlhorn. Pebbling mountain ranges and its application of DCFL-recognition. In *ICALP '80*, volume 85 of *LNCS*, pages 422–435, 1980. doi:10.1007/3-540-10003-2\_89.
- [131] Albert R. Meyer. Weak monadic second order theory of successor is not elementary-recursive. In *Boston Logic Colloquium*, volume 453 of *LNM*, pages 132–154. Springer, 1975. doi:10.1007/BFb0064872.
- [132] Faron Moller and Alexander Moshe Rabinovich. On the expressive power of CTL\*. In *LICS '99*, pages 360–368. IEEE Computer Society, 1999. doi:10.1109/LICS.1999.782631.
- [133] Ha Nguyen. Visibly pushdown automata library, 2006.
- [134] Huu-Vu Nguyen and Tayssir Touili. CARET model checking for malware detection. In *SPIN 2017*, pages 152–161. ACM, 2017. doi:10.1145/3092282.3092301.
- [135] Huu-Vu Nguyen and Tayssir Touili. CARET model checking for pushdown systems. In *SAC 2017*, pages 1393–1400. ACM, 2017. doi:10.1145/3019612.3019829.
- [136] Damian Niwinski. Fixed points vs. infinite generation. In *LICS '88*, pages 402–409. IEEE Computer Society, 1988. doi:10.1109/LICS.1988.5137.
- [137] Nir Piterman and Moshe Y. Vardi. Global model-checking of infinite-state systems. In *CAV '04*, volume 3114 of *LNCS*, pages 387–400. Springer, 2004. doi:10.1007/978-3-540-27813-9\_30.
- [138] Amir Pnueli. The temporal logic of programs. In *FOCS '77*, pages 46–57. IEEE Computer Society, 1977. doi:10.1109/SFCS.1977.32.
- [139] Francesco Pontiggia. POMC. A model checking tool for operator precedence languages on omega-words. Master's thesis, Politecnico di Milano, 2021. URL <http://hdl.handle.net/10589/176028>.
- [140] Francesco Pontiggia, Michele Chiari, and Matteo Pradella. Verification of programs with exceptions through operator-precedence automata. In *SEFM '21*, LNCS. Springer, 2021. To appear.
- [141] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *TACAS '05*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005. doi:10.1007/978-3-540-31980-1\_7.

- [142] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming, 5th Colloquium*, volume 137 of LNCS, pages 337–351. Springer, 1982. doi:10.1007/3-540-11494-7\_22.
- [143] Alexander Rabinovich. A proof of Kamp’s theorem. *Log. Methods Comput. Sci.*, 10(1), 2014. doi:10.2168/LMCS-10(1:14)2014.
- [144] Ganesan Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000. doi:10.1145/349214.349241.
- [145] Joseph G. Rosenstein. *Linear Orderings*. Number 98 in Pure and Applied Mathematics. Academic Press, 1982.
- [146] Grigore Rosu, Feng Chen, and Thomas Ball. Synthesizing monitors for safety properties: This time with calls and returns. In *RV ’08*, volume 5289 of LNCS, pages 51–68. Springer, 2008. doi:10.1007/978-3-540-89247-2\_4.
- [147] Arto K. Salomaa. *Formal Languages*. Academic Press, New York, NY, 1973.
- [148] César Sánchez and Martin Leucker. Regular linear temporal logic with past. In *VMCAI ’10*, volume 5944 of LNCS, pages 295–311. Springer, 2010. doi:10.1007/978-3-642-11319-2\_22.
- [149] Saharon Shelah. The monadic theory of order. *Ann. Math.*, 102(3):379–419, 1975. doi:10.2307/1971037.
- [150] A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985. doi:10.1145/3828.3837.
- [151] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theor. Comput. Sci.*, 3(1): 1–22, 1976. doi:10.1016/0304-3975(76)90061-X.
- [152] Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time: Preliminary report. In *STOC ’73*, pages 1–9. ACM, 1973. doi:10.1145/800125.804029.
- [153] Herb Sutter. Exception-safe generic containers. *C++ Report*, 1997. URL [https://ptgmedia.pearsoncmg.com/imprint\\_downloads/informit/aw/meyerscddemo/DEMO/MAGAZINE/SU\\_FRAME.HTM](https://ptgmedia.pearsoncmg.com/imprint_downloads/informit/aw/meyerscddemo/DEMO/MAGAZINE/SU_FRAME.HTM).
- [154] Nguyen Van Tang and Hitoshi Ohsaki. Checking on-the-fly universality and inclusion problems of visibly pushdown automata. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 94-A(12):2794–2801, 2011. doi:10.1587/transfun.E94.A.2794.
- [155] Wolfgang Thomas. Automata on infinite objects. In Jan van Leeuwen, editor, *Handbook of theoretical computer science (vol. B)*, pages 133–191. MIT Press, Cambridge, MA, USA, 1991. ISBN 0-444-88074-7.
- [156] Wolfgang Thomas. Ehrenfeucht games, the composition method, and the monadic theory of ordinal words. In *Structures in Logic and Computer Science, A Selection of Essays in Honor of Andrzej Ehrenfeucht*, volume 1261 of LNCS, pages 118–143. Springer, 1997. doi:10.1007/3-540-63246-8\_8.

- [157] Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. A robust class of context-sensitive languages. In *LICS '07*, pages 161–170. IEEE Computer Society, 2007. doi:10.1109/LICS.2007.9.
- [158] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS '86*, pages 332–344. IEEE Computer Society, 1986.
- [159] Igor Walukiewicz. Pushdown processes: Games and model-checking. *Inf. Comput.*, 164(2):234–263, 2001. doi:10.1006/inco.2000.2894.

# Index of Acronyms

<b>ACP</b>	Algebra of Communicating Processes
<b>AP</b>	Atomic Proposition
<b>BDD</b>	Boolean Decision Diagram
<b>BPA</b>	Basic Process Algebra
<b>CEGAR</b>	Counterexample-Guided Abstraction Refinement
<b>CFG</b>	Context-Free Grammar
<b>CFL</b>	Context-Free Language
<b>CPU</b>	Central Processing Unit
<b>CTL</b>	Computation Tree Logic
<b>DCFL</b>	Deterministic Context-Free Language
<b>DFS</b>	Depth-First Search
<b>DHP</b>	Downward Hierarchical Path
<b>DSP</b>	Downward Summary Path
<b>DS</b>	Downward Summary
<b>EF</b>	Ehrenfeucht-Fraïssé
<b>FOL</b>	First-Order Logic
<b>FO</b>	First-Order
<b>FSA</b>	Finite-State Automaton
<b>JDK</b>	Java Development Kit
<b>lhs</b>	left-hand side
<b>LIFO</b>	Last In First Out
<b>LR</b>	Left-Recursive
<b>LTL</b>	Linear Temporal Logic
<b>MC</b>	Model Checking

<b>MSOL</b>	Monadic Second-Order Logic
<b>MSO</b>	Monadic Second-Order
<b>NBA</b>	Nondeterministic Büchi Automaton
<b>NWA</b>	Nested Words Automaton
<b>NWTL</b>	Nested Words Temporal Logic
<b>OPA</b>	Operator Precedence Automaton
<b>OPG</b>	Operator Precedence Grammar
<b>OPL</b>	Operator Precedence Language
<b>OPM</b>	Operator Precedence Matrix
<b>OPTL</b>	Operator Precedence Temporal Logic
<b>OP</b>	Operator Precedence
<b>PDS</b>	Pushdown System
<b>POMC</b>	Precedence Oriented Model Checker
<b>POTL</b>	Precedence Oriented Temporal Logic
<b>PR</b>	Precedence Relation
<b>RAM</b>	Random-Access Memory
<b>rhs</b>	right-hand side
<b>RR</b>	Right-Recursive
<b>RSM</b>	Recursive State Machine
<b>SAT</b>	Satisfiability
<b>SCC</b>	Strongly Connected Component
<b>SMT</b>	Satisfiability Modulo Theories
<b>ST</b>	Syntax Tree
<b>UHP</b>	Upward Hierarchical Path
<b>UML</b>	Unified Modeling Language
<b>UOT</b>	Unranked Ordered Tree
<b>USP</b>	Upward Summary Path
<b>US</b>	Upward Summary
<b>VLTL</b>	Visibly Linear Temporal Logic
<b>VPA</b>	Visibly Pushdown Automaton
<b>VPL</b>	Visibly Pushdown Language
$\omega$ <b>OPL</b>	Operator Precedence $\omega$ -Language
$\omega$ <b>OPBA</b>	Operator Precedence Büchi Automaton

# List of Figures

1.1	Time models of LTL, Nested Words and OP Words. . . . .	12
2.1	A program and the transition system modeling it. . . . .	23
2.2	Execution traces of runs $\rho_{fin}$ and $\rho_{inf}$ from Example 2.4 as words. . .	25
2.3	Generalized NBA built for formula $\diamond \text{end}$ . . . . .	32
4.1	Syntax tree of word $w_{ex}$ according to grammar $G_{\text{call}}$ . . . . .	48
4.2	The OPM $M_{\text{call}}$ . . . . .	50
4.3	The sequence of bottom-up reductions during the parsing of $w_{ex}$ . . .	50
4.4	The ST corresponding to word $w_{ex}$ . . . . .	51
4.5	Example procedural program and the derived OPA. . . . .	59
5.1	A play of game $G_3(\mathcal{A}, \mathcal{B})$ won by $\exists$ . . . . .	64
5.2	A play of game $G_3(\mathcal{A}, \mathcal{B})$ won by $\forall$ . . . . .	65
6.1	The example word $w_{ex}$ from Chapter 4 as an OP word. . . . .	74
6.2	The ST of word $w_{ex}$ , with position numbers. . . . .	75
7.1	The OPM $M^{NW}$ . . . . .	84
7.2	A nested word and its translation into an OP word. . . . .	84
7.3	Example OP word on OPM $M_{\text{call}}$ , represented as a syntax tree. . . .	88
7.4	Structure of a word in $L_{\text{call}}$ . . . . .	89
8.1	The example word $w_{ex}$ from Chapter 4 as an OP word. . . . .	93
8.2	The ST of word $w_{ex}$ , with position numbers. . . . .	94
9.1	The UOT corresponding to the word of Figure 8.1. . . . .	106
9.2	ST and UOT of a RR OP $\omega$ -word. . . . .	113
9.3	ST and UOT of a LR OP $\omega$ -word. . . . .	113
9.4	Parts in which we divide a RR UOT for Lemmas 9.16 and 9.17. . . . .	115
9.5	Parts in which we divide a LR UOT for Lemma 9.19. . . . .	120
10.1	Example accepting run of the automaton for $\chi_F^d \text{ret}$ . . . . .	131
10.2	The two possible STs of a generic OP word, and its flat representation. .	132
10.3	The structure of $u_k$ in the word of Figure 10.2. . . . .	133
10.4	The two possible STs of a generic OP word, and its flat representation. .	139
11.1	MiniProc syntax and a MiniProc program. . . . .	150
11.2	Extended OPA and OPA generated from the code of Figure 11.1. . . .	151

11.3 “Generic larger” MiniProc program. . . . . 153

# List of Tables

11.1	Results of the evaluation of hand-made OPAs. . . . .	152
11.2	Results of the evaluation of MiniProc programs, automatically transformed into OPAs. . . . .	152
11.3	Results of verification of the QuickSort benchmark. . . . .	155
11.4	Results of the additional experiments on a MiniProc program equivalent to OPA “generic larger”. . . . .	157
11.5	Results of the additional experiments on the same MiniProc program of Table 11.4, interpreted as a continuously running program. . . . .	158