# Saturation Methods for Global Model-Checking Pushdown Systems

Matthew Hague
St. John's College
University of Oxford

A thesis submitted for the degree of
*Doctor of Philosophy*

Hilary Term 2009

# Abstract

Pushdown systems equip a finite state system with an unbounded stack memory, and are thus infinite state. By recording the call history on the stack, these systems provide a natural model for recursive procedure calls. Model-checking for pushdown systems has been well-studied. Tools implementing pushdown model-checking (e.g. Moped [123, 66]) are an essential back-end component of high-profile software model checkers such as SLAM [126], Blast [125] and Terminator [26].

Higher-order pushdown systems define a more complex memory structure: a higher-order stack is a stack of lower-order stacks. These systems form a robust hierarchy closely related to the Caucal hierarchy and higher-order recursion schemes. This latter connection demonstrates their importance as models for programs with higher-order functions.

We study the global model-checking problem for (higher-order) pushdown systems. In particular, we present a new algorithm for computing the winning regions of a parity game played over an order-1 pushdown system. We then show how to compute the winning regions of two-player reachability games over order-$n$ pushdown systems. These algorithms extend the *saturation* methods of Bouajjani, Esparza and Maler for order-1 pushdown systems, and Bouajjani and Meyer for higher-order pushdown systems with a single control state. These techniques begin with an automaton recognising (higher-order) stacks, and iteratively add new transitions until the automaton becomes saturated.

The reachability result, presented at FoSSaCS 2007 [77] and in the LMCS journal [78], is the main contribution of the thesis. We break the saturation paradigm by adding new states to the automaton during the iteration. We identify the fixed points required for termination by tracking the updates that are applied, rather than by observing the transition structure. We give a number of applications of this result to LTL model-checking, branching-time model-checking, non-emptiness of higher-order pushdown automata and Büchi games.

Our second major contribution is the first application of the saturation technique to parity games. We begin with a $\mu$-calculus characterisation of the winning region. This formula alternates greatest and least fixed point operators over a kind of reachability formula. Hence, we can use a version of our reachability algorithm, and modifications of the Büchi techniques, to compute the required result. The main advantages of this approach compared to existing techniques due to Cachat [127], Serre [92] and Vardi *et al.* [90, 86] are that it is direct and that it is not immediately exponential in the number of control states, although the worst-case complexity remains the same.

# Acknowledgements

I would like to thank my supervisor Luke Ong for providing the guidance required to produce this work and make the jump from student to researcher.

Completing this thesis has been a difficult affair for which I have relied greatly on my family, friends and colleagues. In particular, I'd like to thank my parents and several close friends for their extraordinary level support over the final year. I would also like to thank Joel Ouaknine whose advice has been essential.

I am grateful to the many researchers who have responded to questions about their work. Olivier Serre has provided many invaluable comments throughout my studies. I am also enormously indebted to Arnaud Carayol whose input has been instrumental in finishing this work.

I would also like to thank my examiners Ben Worrell and Javier Esparza for their patience throughout a somewhat lengthy submission process.

Finally, this thesis would not have been possible without the financial support of the *Engineering and Physical Sciences Research Council* and the *Centre for Metacomputation* at *The Oxford University Computing Laboratory*.

# Contents


**1 Introduction** **1**

1.1 Infinite State Verification . . . 1

1.2 Higher-Order Pushdown Systems . . . 2

1.3 Global Model-Checking of Pushdown Systems . . . 2

1.4 Contributions and Document Structure . . . 3

1.5 Summary . . . 5

**2 Verifying Models of Computation** **7**

2.1 Program Models . . . 7

2.2 Specification Languages . . . 10

2.2.1 Linear Time . . . 10

2.2.2 Branching Time Logics . . . 12

2.2.3 Monadic Second-Order Logic . . . 13

2.3 Automata and Games . . . 14

2.3.1 Büchi Automata . . . 14

2.3.2 Alternating Automata . . . 15

2.3.3 Games . . . 16

2.3.4 Parity Games and the $\mu$-Calculus . . . 18

2.4 Infinite State Systems . . . 19

2.4.1 Sequential Systems . . . 19

2.4.2 Concurrent Systems . . . 22

2.5 Alternation . . . 24

2.6 Order-1 Pushdown Systems and Automata . . . 24

2.6.1 Definition . . . 24

2.6.2 Examples . . . 26

2.6.3 Local Model-Checking of Order-1 Pushdown Systems . . . 26

2.6.4 Properties of High Borel Complexity . . . 30

2.7 Order-1 Extensions of Pushdown Systems/Automata . . . 33

2.7.1 Regular Stack Properties . . . 33

2.7.2 Recursive State Machines . . . 35

2.7.3 Weighted Pushdown Systems . . . 37

2.7.4 Visibly Pushdown Games . . . 40

2.8 Higher-Order Pushdown Systems and Automata . . . 42

2.8.1 Improving the CD Player . . . 43

2.8.2 Higher-Order Model-Checking . . . 44

2.8.3 Definition . . . 45

2.8.4 The Caucal Hierarchy . . . 48
2.8.5 Local Model-Checking of Higher-Order Pushdown Systems . . . 52
2.8.6 Recursion Schemes . . . 54
2.8.7 Collapsible Pushdown Systems . . . 58
2.9 Global Model-Checking . . . 60
2.10 Summary . . . 61

**3 Saturation Methods for Global Model-Checking 63**
3.1 Order-1 Reachability Analysis . . . 63
3.1.1 Multi-Automata . . . 63
3.1.2 Backwards Reachability . . . 64
3.1.3 Extension to Alternating Pushdown Systems . . . 67
3.2 Winning Regions of Order-1 Büchi Games . . . 67
3.2.1 The Naïve Algorithm . . . 68
3.2.2 Termination . . . 70
3.2.3 The Algorithm . . . 70
3.3 Context-Free Higher-Order Pushdown Systems . . . 72
3.3.1 Nested Store Automata . . . 72
3.3.2 Backwards Reachability . . . 72
3.4 Carayol Regularity . . . 75
3.4.1 Normal Form . . . 75
3.4.2 MSO-Definability . . . 76
3.5 Summary . . . 76

**4 Order-1 Pushdown Parity Games 79**
4.1 Preliminary Definitions . . . 79
4.2 An Example . . . 81
4.3 The Algorithm . . . 84
4.3.1 Format of the Automata. . . . 84
4.3.2 Definition of the Algorithm. . . . 86
4.4 Termination and Correctness . . . 86
4.4.1 Termination . . . 86
4.4.2 Correctness . . . 87
4.5 Optimisation . . . 93
4.6 Existing Approaches . . . 95
4.6.1 Extending Walukiewicz's Algorithm . . . 95
4.6.2 Games with $\omega$-Regular Winning Conditions . . . 95
4.6.3 Two-Way Alternating Tree Automata . . . 97
4.6.4 Comparing the Approaches . . . 98
4.7 Abstract Pushdown Games and Higher-Order Pushdown Systems . . . 99
4.8 Summary . . . 101

**5 Global Reachability Analysis of Higher-Order Pushdown Systems 103**
5.1 $n$-Store Multi-Automata . . . 104
5.2 Regularity . . . 105
5.3 The Order-Two Case . . . 107
5.3.1 Example . . . 107

5.3.2 Preliminaries . . . . . . . . . . 113
5.3.3 Constructing the Sequence $A_0, A_1, \ldots$ . . . . . . . . . . 114
5.3.4 Constructing the Automaton $A_*$ . . . . . . . . . . 116
5.4 The General Case . . . . . . . . . . 118
5.4.1 Preliminaries . . . . . . . . . . 118
5.4.2 Further Examples . . . . . . . . . . 120
5.4.3 Constructing $A_0, A_1, \ldots$ . . . . . . . . . . 122
5.4.4 Soundness and Completeness for $A_0, A_1, \ldots$ . . . . . . . . . . 123
5.4.5 Constructing $A_*$ . . . . . . . . . . 132
5.4.6 Proofs for $A_*$ . . . . . . . . . . 134
5.4.7 Complexity . . . . . . . . . . 144
5.5 An Alternative Order-2 Construction . . . . . . . . . . 145
5.6 Algorithms over $n$-Store (Multi-)Automata . . . . . . . . . . 146
5.6.1 Enumerating Runs . . . . . . . . . . 146
5.6.2 Membership . . . . . . . . . . 148
5.6.3 Boolean Operations . . . . . . . . . . 149
5.7 Summary . . . . . . . . . . 153

**6 Applications** **155**
6.1 Model-Checking Linear-Time Temporal Logics . . . . . . . . . . 155
6.2 Reachability Games . . . . . . . . . . 159
6.3 Model-Checking Branching-Time Temporal Logics . . . . . . . . . . 161
6.4 Non-emptiness of Higher-Order Pushdown Automata . . . . . . . . . . 164
6.5 Regular Goal Sets of Higher-Order Büchi Games . . . . . . . . . . 168
6.6 Summary . . . . . . . . . . 173

**7 Conclusion** **175**
7.1 Summary of Contributions . . . . . . . . . . 175
7.2 Further Research . . . . . . . . . . 176
7.3 Summary . . . . . . . . . . 178

# Chapter 1

# Introduction

Computer systems have transformed and continue to transform modern society. More and more aspects of our lives are enhanced by the intelligence of software, and are therefore exposed to the mistakes of software developers. When computer programs are used in safety critical applications, such as car braking mechanisms, errors become much more than a tolerable frustration.

Traditionally, systems are tested for errors before they are deployed. The behaviour of the system is checked against a range of possible scenarios to make sure it is correct. However, testing every possible scenario is, in most cases, impractical. The designer of the tests, whether a person or a specially designed program, chooses a selection of important cases. For example, an old database might store years in a two-digit format. The database will be unlikely to behave badly when information is dated 1998, but may make mistakes when its year is 2000. The effectiveness of this approach relies on the perspicacity of the tester and their knowledge of both the system and its desired behaviour. Inevitably, not all errors are caught.

## 1.1 Infinite State Verification

Testing, therefore, cannot guarantee that a program is correct. In formal verification the requirements of the system are specified precisely, and exhaustively checked against a formal model of the system. If an error is found, it is reported to the programmer, who can use the information to correct the program.

Although we know, due to the undecidability of the halting problem, that we cannot verify all programs, there are some types of program that can be verified. The simplest such class of programs are *finite state* programs. All hardware is finite state because it can only store a finite amount of information. Although research was small-scale in the 1980s, advances in the early 1990s — such as symbolic methods [68] — greatly increased the manageable size of verification problems, resulting in an industrial acceptance of the discipline by the end of the decade [116].

However, finite state verification is not always adequate for analysing software. Fundamental constructs, such as recursion, are inherently infinite state. One of the simplest kinds of infinite state system that we can verify are *pushdown systems*. In these systems, memory is arranged as a *stack* of data. The program can only read the top of the stack and is not allowed to know how tall it is. It can also add items to the top of the stack, or remove items

from the top of the stack. Although this restriction appears quite severe, it is able to model recursive procedure calls. In fact, tools that verify these kinds of systems are used in many state-of-the-art verification suites.

## 1.2 Higher-Order Pushdown Systems

Identifying larger classes of verifiable programs is difficult. An obvious extension of pushdown systems allows two stacks instead of one. However, allowing two stacks leads to undecidability[1]. The topic of this thesis is an extension that does permit verification: *higher-order* pushdown systems. These are pushdown systems where the items on the stack are stacks themselves. Higher-order stacks allow us to model higher-order features of computer programs. In Section 2.8.2 we discuss the applications of higher-order model-checking to NASA remote agents, system design, natural language processing and security policies.

Higher-order pushdown systems are indexed by the nesting depth of their stacks. That is, an order-1 pushdown system has a stack of characters. An order-2 system has a stack of order-1 stacks and so on. They were introduced by Maslov as generators of a strict hierarchy of word languages [20].

The hierarchy of pushdown systems is robust: as generators of trees, they are equivalent to safe higher-order recursion schemes [132], and the $\varepsilon$-closure of the graphs generated by higher-order pushdown systems are equivalent to the Caucal hierarchy of graphs [13, 36]. They are also known to have a decidable MSO theories [40, 13, 132].

The connection with higher-order recursion schemes shows that higher-order pushdown systems are suitable for modelling higher-order programs written in languages such as OCaml, F♯ and C♯. Higher-order languages also provide a succinct formalism for system design [57], and are important in natural language processing [24, 49].

## 1.3 Global Model-Checking of Pushdown Systems

We distinguish two different kinds of model-checking: *local* and *global*. Whilst local model-checking ensures a specification is met from a designated initial state, global model-checking discovers the complete set of states satisfying a property. Global model-checking is the subject of this thesis.

In the order-1 case a number of global model-checking results have been presented in the literature. The first such result is due Bouajjani, Esparza and Maler [3, 8] and independently to Finkel, Willems and Wolper [15]. It is an application of an algorithm for string rewriting systems by Book and Otto [118]. They provide a technique for global model-checking with reachability specifications. That is, they compute the set of pushdown states from which a specified set of states can be reached. This is an important verification problem in its own right: many properties required in industry are *safety* properties — that is, an undesirable program state (such as deadlock) is never reached.

This result was extended by Cachat to the problem of computing the winning region in games with Büchi winning conditions [127]. Büchi winning conditions allow the expression of *liveness* properties. Whereas a safety property ensures that nothing bad happens, a liveness property ensures that something good happens. For example, we may require that, whenever

[1] Two stacks can be used to model the tape of a Turing machine: the contents of the first stack are the contents of the tape to the left of the write-head, whilst the second stack contains the balance.

a message is sent, at some point in the future a message is received. This property can be translated to a Büchi condition.

Büchi conditions express linear-time properties. These properties observe runs of a program. Hence, each position has a single successor. Alternatively, we can consider branching-time properties. For example, we may want to assert that a default state can optionally be reached from all program states, without insisting that it is always returned to. The $\mu$-calculus is an expressive logic that captures linear- and branching-time properties. This logic can be model-checked using *parity* conditions. Independently, Cachat [127] and Serre [92] provided global model-checking algorithms for parity conditions. Later, Vardi *et al.* [90, 86] provided a further solution to the problem that provides a unified solution to a number of similar model-checking problems.

In the higher-order case, Bouajjani and Meyer have provided an algorithm for global reachability analysis of higher-order pushdown systems with a single control state [2]. That is, systems that do not store any information outside of the stack. In the order-1 case, this restriction implies that procedures do not have return values. Recently, Serre *et al.* [12] provided a solution to the global model-checking problem for higher-order pushdown parity games.

## 1.4 Contributions and Document Structure

In this thesis we present two global model-checking algorithms for pushdown systems with parity (order-1) and reachability (order-$n$) conditions respectively. We then present a number of applications of the reachability result. The document is structured as follows.

### Verifying Models of Computation

In Chapter 2 we give an overview of infinite state model-checking. This chapter gives a general overview of the area. The technical details relevant to the main results of the thesis are described separately in Chapter 3 We begin with a discussion of the model-checking framework, including several specification languages, automata and games (Section 2.1 to Section 2.3). Then, in Section 2.4 we describe a number of infinite state program models before focussing on pushdown systems in Section 2.6.

After defining order-1 pushdown systems and giving examples of their use and behaviour, we describe a seminal local model-checking algorithm for parity conditions introduced by Walukiewicz [53].

In Section 2.7 we discuss some extensions to pushdown systems and related models. The first section deals with regular stack properties. These properties allow a specification to analyse the stack contents and have applications in security and dataflow analysis. We then give an account of recursive state machines. These are a natural extension of finite state models allowing recursion. They are known to coincide with pushdown systems. That is, every pushdown system can be simulated by a recursive state machine, and vice versa. We also discuss weighted pushdown systems that augment pushdown transitions with weights. These weights can be used, for example, in security to differentiate certification options by the nature of the information revealed. Finally we describe visibly pushdown automata. By tying the stack behaviour to the input received, visibly pushdown automata allow expressive specification languages to be model-checked.

Higher-order pushdown systems are detailed in Section 2.8. After motivating and defining these systems we discuss their connections with the Caucal hierarchy and higher-order recursion schemes satisfying a constraint called *safety*. We also describe a local model-checking algorithm based on techniques due to Serre which can be considered a natural extension of Walukiewicz's method for the order-1 case. Finally we discuss collapsible pushdown systems. These systems allow a *collapse* operation that returns the stack to a specified previous state. This extension coincides with recursion schemes without the *safety* constraint.

### Saturation Methods for Global Reachability Analysis

Chapter 3 gives a detailed account of two global reachability checking algorithms and a global model-checking algorithm for order-1 Büchi games. These algorithms form the basis of our contributed algorithms.

The first algorithm, discussed in Section 3.1, is Bouajjani, Esparza and Maler's original backwards reachability algorithm [3, 8]. This algorithm introduced the *saturation* method underpinning this thesis. A variation of automata over finite words are used to represent sets of pushdown states. Transitions are added to the automaton to reflect the pushdown commands available. Eventually the automaton becomes saturated. That is, no more transitions can be added.

The second algorithm is Cachat's extension of this algorithm to Büchi games. This algorithm uses a characterisation of Éloïse's winning region as the greatest fixed point of a number of applications of the reachability algorithm. A "speed-up" technique is required to ensure that this computation terminates.

In Section 3.3 we describe Bouajjani and Meyer's extension of the order-1 result to higher-order pushdown systems with a single control state [2]. Bouajjani and Meyer represent sets of higher-order pushdown states using a kind of nested automata, reflecting the nested structure of higher-order stacks. This gives us the notion of regularity used throughout the thesis.

A different notion of regularity, due to Carayol [11], is considered in Section 3.4. Rather than representing the stack explicitly, Carayol represents a stack by the sequence of commands required to construct it. This notion of regularity coincides with MSO definability and, contrary to the previous notion, sets of reachable pushdown states are also regular.

### Order-1 Pushdown Parity Games

Chapter 4 contains the first contribution of the thesis: an algorithm for computing Éloïse's winning region of an order-1 pushdown parity game. This technique uses a $\mu$-calculus characterisation of Éloïse's winning region which is the result of a number of alternating fixed point calculations. We show how Cachat's Büchi games algorithm can be extended to perform the computation.

We discuss techniques due independently to Cachat [127], Serre [92] and Vardi *et al.* [90, 86] for computing the winning region in the order-1 case. These techniques use Walukiewicz's algorithm as an oracle to construct an automaton accepting the winning region or use two-way alternating tree automata. These techniques do not follow the saturation paradigm. The main advantage of our approach is that it operates directly on the pushdown game (without reduction to a further game) and may, in some "fortunate" cases, avoid the full cost of the exponential blow-up. We then describe recent work by Serre extending his order-1 techniques to the order-$n$ case [12] and an alternative, unpublished, approach by Seth.

### Global Reachability Analysis of Higher-Order Pushdown Systems

The main contribution of the thesis is the subject of Chapter 5. This is an extension of the order-1 reachability result due to Bouajjani, Esparza and Maler, and the higher-order reachability result due to Bouajjani and Meyer to the general case of (alternating) higher-order pushdown systems with an arbitrary number of control states.

The main innovation of this approach is the use of sets describing how a nested automaton is to be updated, rather than the direct updates used by the previous algorithms. These sets allow us to easily identify the fixed points required for termination. Our algorithm runs in $n$-EXPTIME in the number of states of the automaton describing the set to be reached. This automaton is at least as large as the set of control-states of the system.

In Section 5.5 we discuss an alternative approach, due to Seth [23], for the order-2 case.

### Applications

Finally we discuss a number of applications of our reachability result. In particular our algorithm can be used to determine the set of positions from which Éloïse can win a reachability game played over a pushdown system, perform LTL model-checking, and model-checking of the alternation-free $\mu$-calculus. We also show that our algorithm can be used to test the emptiness of a higher-order pushdown automaton. This proves that our reachability algorithm is optimal.

We then consider Büchi games played over pushdown systems. We show that a reduction to *simple goal sets* is also possible in the higher-order case. The reduction we use is quite different from the order-1 technique described by Cachat.

## 1.5 Summary

We have introduced the discipline of formal verification and discussed the importance of infinite state systems for model-checking software. We have described pushdown systems, which can be used as a model for program recursion, and their extension to higher-order pushdown systems, which can be used to model higher-order constructs. We have given an overview of the global model-checking paradigm and previous results for pushdown systems and a sub-class of higher-order pushdown systems.

We then gave a summary of the structure of the thesis and it's two main contributions: global model-checking algorithms for order-1 parity games and higher-order reachability games.

# Chapter 2

# Verifying Models of Computation

An important property of any program is its correctness. In computing, errors are difficult to prevent in both large and small systems and are too often viewed as an inevitability. There are a number of high-profile cases, such as the infamous *Pentium floating point error* — which cost a reported $500 million [67] — that illustrate the need for careful checking of both hardware and software.

One basic method for checking correctness is straightforward testing: a user tries to break the system by inputting as many test cases as is feasible. However, to test every set of input conditions is almost impossible — especially in the concurrent case, where a degree of nondeterminism is present. Testing, therefore, cannot guarantee correctness. Furthermore, it can sometimes be difficult to determine, via testing, precisely where a program goes wrong.

It would be ideal, then, if we were able to prove that a system is error free, or identify errors before they identify themselves. This is the goal of the model-checking problem.

In general, this problem is undecidable. That is, for any program and any specification, we cannot automatically determine whether the specification is met. The halting problem is a classic example of this undecidability. However, we can restrict the range of expressible properties via the method we use to represent the system and its specification. Once restricted, the problem often becomes decidable, although the complexities are high.

Despite the high complexity, efficient algorithms and implementations have been developed and model-checking enjoys a number of successes in industry [116]. Recently, the 2007 Turing Award was given to Clark, Emerson and Sifakis for their work in transforming model-checking *"from a theoretical technique to a highly effective verification technology"*.

## 2.1 Program Models

### Kripke Structures

To be able to reason about programs we require a formal representation of the system. There are a number of formalisms in the literature, ranging from the theoretical (lambda calculus, pi-calculus, CSP, &c.) to the more practically motivated (Promela, SMV). In this section we introduce the model most commonly associated with model-checking: Kripke Structures. Although many model-checkers take a more practical language as input, the internal workings of the system, and the accompanying literature, often use Kripke structures. Therefore, they are the right formalism for discussing model-checking algorithms.

**Definition 2.1.1.** A **Kripke Structure** $S$ over a finite set of atomic propositions $AP$ is a tuple $(\mathcal{Q}, \Delta, l, \mathcal{I})$, where $\mathcal{Q}$ is a (possibly infinite) set of states, $\Delta \subseteq \mathcal{Q} \times \mathcal{Q}$ is a transition relation, and $l : \mathcal{Q} \to 2^{AP}$ labels each state of $Q$ with the set of atomic propositions that are true at that state. Finally $I \subseteq Q$ is a set of initial states (usually a singleton) of $S$.

Together $\mathcal{Q}$ and $\Delta$ form a directed graph. The program starts in an initial state. At each program step a transition is taken to a next state. This gives us the notion of a path and a run.

**Definition 2.1.2.** Given a Kripke Structure $S = (\mathcal{Q}, \Delta, l, \mathcal{I})$ a sequence $q_0, q_1, q_2, \ldots$ of states $q_i \in \mathcal{Q}$ such that $(q_i, q_{i+1}) \in \Delta$ is a **path** through $S$. If the sequence is maximal, it is a **fullpath**. A fullpath is a **run** when $q_0 \in \mathcal{I}$.

An equivalent, definition of Kripke Structures uses a finite alphabet $\Sigma$ rather than a set of atomic propositions. In this alternative definition it is the transition relation that is labelled rather than the states. Conceptually the program will perform an action $a \in \Sigma$ to move from one state to another.

To translate from $AP$ to $\Sigma$ we set $2^{AP}$ as our alphabet and move the labelling of a state to the transition relation. In the opposite direction, we encode $\Sigma$ using a number of atomic propositions and move the labelling to the states. If a state is reachable by both an $a \in \Sigma$ and a $b \in \Sigma$ action ($a \neq b$), then we divide it into two states — one if it is reached by an $a$, the other if it is reached by a $b$. Consequently a sequence of states (and therefore a path or a run) can equivalently be represented as a word over the alphabet $\Sigma$.

### Finite State Systems

Traditional model-checking uses finite state program models. That is, programs are described using Kripke structure with a finite set of states. Consider the following CD player:

```
program CDPlayer
  CD = null
  while input do
    case load1: CD = cd1
    case load2: CD = cd2
    case play:  playCD(CD)
  end
end program
```

Figure 2.1 shows an edge-labelled finite state Kripke structure modelling the above program. Observe that runs of this program, except in the case of an error, are infinite. This is a reactive system. Other examples of these systems are web-servers and control mechanisms (such as an auto-pilot). Since a finite program run can be modelled by an infinite run with an infinitely occurring sink state, we consider the general case of infinite runs.

### Trees

The program models discussed above are graphs. Trees are a subclass of the class of graphs and are often used to represent runs of a system. A tree has a designated root node, and all other nodes have at most one predecessor. Thus, the nodes form an acyclic structure with

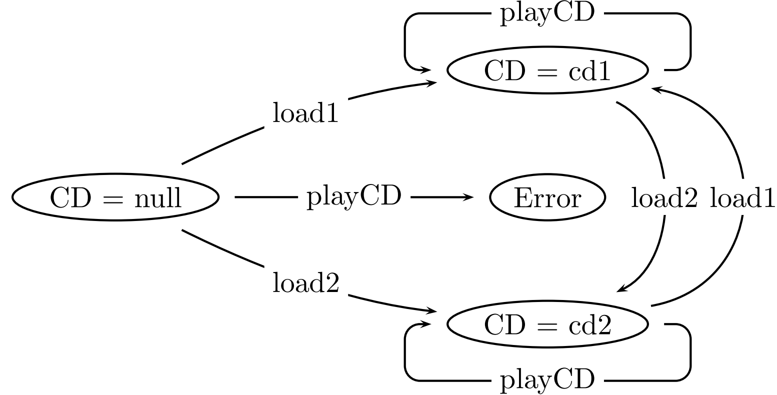



Figure 2.1: Modelling a CD player

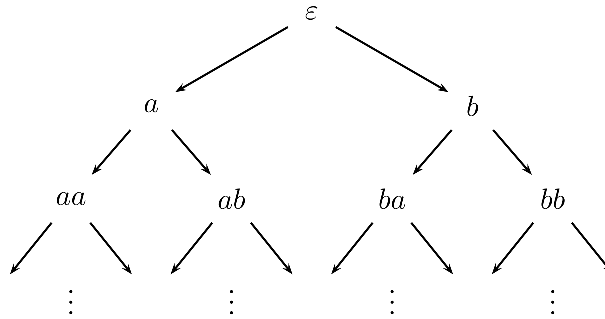



Figure 2.2: The binary tree $T = \{a, b\}^*$

independent branches. These branches may represent alternative executions of a program, or parallel executions that do not communicate with each other.

**Definition 2.1.3.** Given alphabets $\Sigma$ and $\Gamma$, a **tree** is a tuple $(T, \tau)$ where $T \subseteq \Sigma^*$ is such that $\varepsilon \in T$ and if $wa \in T$, then $w \in T$. Furthermore, $\tau : \Sigma^* \to \Gamma$ is a node labelling.

Nodes of a tree are represented by the sequence of characters required to reach them. Hence, the empty word $\varepsilon$ is the root of the tree. A node $wa \in T$ has the parent node $w$ and child nodes of the form $wab$ for some $b \in \Sigma$. The labelling $\tau$ provides information about the node, for example, $\Gamma$ may be the set $2^{AP}$ for a set of atomic propositions $AP$. This labelling may associate each node with the set of propositions it satisfies. For example, let $\Sigma = \{a, b\}$ and $T = \Sigma^*$. This is the complete binary tree, shown in Figure 2.2.

Finally, trees may be *ranked* and/or *ordered*. In a ranked tree, each node is assigned an arity. The arity of a node specifies exactly how many children the node has. In an ordered tree, the left-to-right arrangement of a node's children is a property of the node. For example, given an ordering on $\Sigma$, a tree whose branches are arranged in ascending order is distinct from a tree which has the same nodes and branches with the exception that the branches appear in descending order.

## 2.2 Specification Languages

We can reason about programs using several different languages. Two important areas of study are **linear time** and **branching time** languages. In the linear time paradigm we assume that each time step has one possible future. In the branching time philosophy, many different futures may occur. Vardi argues that the linear time paradigm is more intuitive and more useful in practice [83]. The reachability and Büchi properties investigated in this thesis fall into the linear time paradigm. Our discussion of parity games in Chapter 4 subsumes the previous linear time results and is branching time. We begin by introducing the logics.

### 2.2.1 Linear Time

#### $\omega$-Regular Languages

A program run may be considered to be a word over the alphabet $\Sigma$, where $\Sigma$ is a set of program states. An important tool for discussing the expressivity of linear time logics are $\omega$-regular languages. We can express almost all important properties of a system using these languages.

**Definition 2.2.1.** The syntax of **$\omega$-regular expressions** is,

$$\alpha \ ::= \ \varepsilon \mid a \mid \alpha \cup \alpha \mid \alpha ; \alpha \mid \alpha^* \mid \alpha^\omega$$

where $\varepsilon$ denotes the empty word, $a$ the single character $a$, $\cup$ the union of two languages, ; the composition and $*$ and $\omega$ (respectively) finite and infinite repetition of a language.

An important subclass of $\omega$-regular languages is the **star-free languages**. This is the subclass without $*$ or $\omega$, but complementation ($\overline{\alpha}$) is allowed. The star-free languages are those that can be defined using first-order logic over strings [115].

#### Linear Temporal Logic

Linear Temporal Logic (LTL) was introduced by Pnueli in 1977 [22]. It is interpreted over a linear time structure, where proposition valuations $\pi : N \rightarrow 2^{AP}$ are parameterised by a natural number denoting the time-step. What may be true initially might not be true at a time-step greater than zero.

The logic is built from a set of atomic propositions $AP$, the boolean connectives and two binary temporal operators $\bigcirc$ (tomorrow) and $U$ (until). The meaning of $\bigcirc$ is straightforward given the semantics given below. Until is more subtle. The formula $\phi U \psi$ asserts that $\phi$ holds at all time-steps from now until $\psi$ holds, and, moreover, that $\psi$ will eventually hold. We interpret $U$ non-strictly, and allow $\psi$ to hold immediately, meaning that $\phi$ may never need to be satisfied.

**Definition 2.2.2.** For a set of atomic propositions $AP$ and an **LTL** formula $\phi$, given a valuation $\pi : N \rightarrow 2^{AP}$ and time-step $i \in N$, we interpret $\phi$ as follows ($p \in AP$):

$$\begin{array}{lcl}
\pi, i \models p & \iff & p \in \pi(i) \\
\pi, i \models \phi \wedge \psi & \iff & \pi, i \models \phi \text{ and } \pi, i \models \psi \\
\pi, i \models \neg \phi & \iff & \text{not } \pi, i \models \phi \\
\pi, i \models \bigcirc \phi & \iff & \pi, i+1 \models \phi \\
\pi, i \models \phi U \psi & \iff & \text{there is } j \geq i \text{ such that } \pi, j \models \psi \text{ and for all } i \leq k < j \\
 & & \text{we have } \pi, k \models \phi
\end{array}$$

The literature uses three important abbreviations when discussing LTL: $\phi V \psi$, $F\phi$ and $G\phi$. The first of these $\phi V \psi$ is defined as the dual of $\phi U \psi$, that is $\neg(\neg\phi U \neg\psi)$. The *future* operator $F\phi$ asserts that $\phi$ holds at some point in the future, and is encoded $\top U \phi$, where $\top$ is truth. Because of the non-strict interpretation of $U$, $F\phi$ is satisfied if $\phi$ holds immediately. Finally, the *globally* operator $G\phi$ is the dual of $F$ ($\neg F \neg \phi$) and asserts that $\phi$ is true from this time-step on.

LTL can express all star-free $\omega$-regular languages, and therefore all first order properties of strings [138].

### Linear Time $\mu$-Calculus

Wolper first observed that LTL cannot express all $\omega$-regular properties [95]. It has been shown that LTL can express only the star-free properties [138]. It was also shown that LTL is not adequate for modular verification because it cannot express the required assumptions about the environment [91]. That is, we cannot break large programs into modules and verify them independently.

In response to these observations, Banieqbal and Barringer proposed the use of fixed point operators [25], yielding the Linear Time $\mu$-Calculus ($\mu$TL). This logic extends LTL with least ($\mu$) and greatest ($\nu$) fixed point operators and requires the introduction of a set of fixed point variables $\mathcal{X}$ and an environment $\mathcal{V} : \mathcal{X} \rightarrow 2^N$. The environment $\mathcal{V}$ identifies the set of time-steps where a variable — not bound by a fixed point operator ($\sigma X.\phi(X)$ where $\sigma \in \{\mu, \nu\}$) — is true. A $\mu$TL formula is closed if all fixed point variables appearing in the formula are bound by a fixed point operator.

To aid in the definition of the semantics of $\mu$TL we introduce the notation $[\![\phi]\!]^{\pi}_{\mathcal{V}}$ to denote the set of all time-steps at which $\phi$ holds, given a run $\pi$ and environment $\mathcal{V}$.

**Definition 2.2.3.** For a set of atomic propositions $AP$, a disjoint set of fixed point variables $\mathcal{X}$ and a $\mu$**TL** formula $\phi$, given a valuation $\pi : N \rightarrow 2^{AP}$ and environment $\mathcal{V} : \mathcal{X} \rightarrow 2^N$, we interpret $\phi$ as follows (in addition to the semantics of LTL)($X \in \mathcal{X}$):

$$\begin{array}{lcl} [\![X]\!]^{\pi}_{\mathcal{V}} & := & \mathcal{V}(X) \\ [\![\mu X.\phi]\!]^{\pi}_{\mathcal{V}} & := & \bigcap\{M \subseteq N | [\![\phi]\!]^{\pi}_{\mathcal{V}[X \mapsto M]} \subseteq M\} \\ [\![\nu X.\phi]\!]^{\pi}_{\mathcal{V}} & := & \bigcup\{M \subseteq N | M \subseteq [\![\phi]\!]^{\pi}_{\mathcal{V}[X \mapsto M]}\} \end{array}$$

where $\mathcal{V}[X \mapsto M]$ behaves like $\mathcal{V}$ in all cases except $\mathcal{V}(X)$ is $M$.

The theory of fixed point operators is not a simple one. Intuitively, the fixed point operators allow properties to be defined recursively. For example the formulas $\phi = \mu Z.\phi' \vee Z$ and $\phi = \nu Z.\phi' \vee Z$ assert that either $\phi'$ holds at the current state, or there exists a successor state that satisfies (recursively) $\phi$. A least fixed point operator asserts that the recursion terminates after a finite number of steps, whereas the greatest fixed point operator allows infinite recursion. These operators are allow us to encode the $\omega$-regular language operators $*$ and $\omega$ respectively. Consequently $\mu$TL can express all $\omega$-regular properties.

For example, suppose we have one atomic proposition $p$. Let 1 denote a time-step where $p$ holds and 0 denote a time-step where $p$ does not hold. If we bound the variable $X$ with $\mu$ and require that we have the sequence 110 followed by X, or that $p$ holds globally, we would be expressing the language $(110)^*(1)^\omega$. This is because we can either settle in $Gp$ now ($(1)^\omega$), or take the second route which requires (110) and then recurses (or loops) through $X$ back to

where $X$ is bound. Since $X$ is bound by $\mu$, we can only take this second route a finite number of times. Eventually, we must choose $Gp$. If instead we had bound $X$ with $\nu$ and removed $Gp$ from the disjunction, we would require an infinite repetition of 110, that is, $(110)^\omega$.

The fixed point operators can also be characterised using approximants. The denotation $[\![\mu X.\phi(X)]\!]^{\mathcal{G}}_V$ can be defined as the least fixed point of the sequence (where $i, \alpha$ and $\beta$ are ordinals):

$$\begin{array}{rcl} [\![\mu^0 X.\phi(X)]\!]^{\mathcal{G}}_V & = & \emptyset \\ [\![\mu^{i+1} X.\phi(X)]\!]^{\mathcal{G}}_V & = & [\![\phi(X)]\!]^{\mathcal{G}}_{V[X=[\![\mu^i X.\phi(X)]\!]^{\mathcal{G}}_V]} \\ [\![\mu^\alpha X.\phi(X)]\!]^{\mathcal{G}}_V & = & \bigcup_{\beta<\alpha}[\![\mu^\beta X.\phi(X)]\!]^{\mathcal{G}}_V \end{array}$$

and the set $[\![\nu Z_{m'}.\chi(Z_1,\ldots,Z_{m'})]\!]^{\mathcal{G}}_V$ can be defined as the least fixed point of the sequence:

$$\begin{array}{rcl} [\![\mu^0 X.\phi(X)]\!]^{\mathcal{G}}_V & = & 2^N \\ [\![\mu^{i+1} X.\phi(X)]\!]^{\mathcal{G}}_V & = & [\![\phi(X)]\!]^{\mathcal{G}}_{V[X=[\![\mu^i X.\phi(X)]\!]^{\mathcal{G}}_V]} \\ [\![\mu^\alpha X.\phi(X)]\!]^{\mathcal{G}}_V & = & \bigcup_{\beta<\alpha}[\![\mu^\beta X.\phi(X)]\!]^{\mathcal{G}}_V \end{array}$$

That is, the least fixed point can be calculated by repeatedly calculating $\phi$, using the empty set as the initial value of $X$. Dually, the greatest fixed point can be calculated by beginning with the set of all positions as the initial value of $X$. Note that we may need a transfinite number of iterations to compute a fixed point.

### 2.2.2 Branching Time Logics

In the linear paradigm, we assume that each moment of time has a unique successor. An alternative model of time assumes many different possible futures. To reason in this model we use branching time logics.

#### Computational Tree Logic

Computational Tree Logic was introduced in 1981 by Emerson and Clarke [46]. It is interpreted over computation trees, rather than linear sequences of time-steps. This represents the fact that any state of the program may have many possible next states.

**Definition 2.2.4.** Given a set of atomic propositions $AP$, the syntax of **CTL** is as follows ($p \in AP$),

$$\phi \ := \ p \mid \phi \wedge \psi \mid \neg\phi \mid E(\phi U \psi) \mid A(\phi U \psi) \mid E \bigcirc \phi \mid A \bigcirc \phi$$

Intuitively, the semantics of a CTL assertion is similar to LTL. The temporal connectives are augmented with existential and universal quantifiers, $E$ and $A$. The existential quantifier requires that the assertion holds on some path from the current node in the tree. Dually, the universal quantifier requires that it holds on all paths leading from the current node.

#### Computational Tree Logic*

CTL is a fragment of CTL*. CTL* is an extension of LTL with quantification over program runs.

**Definition 2.2.5.** Given a set of atomic propositions $AP$, the syntax of **CTL*** is as follows ($p \in AP$),

$$\phi \ := \ p \mid \phi \wedge \psi \mid \neg\phi \mid E\phi \mid \phi U \psi \mid \bigcirc \phi$$

Universal quantification $A\phi$ is defined as an abbreviation for $\neg E \neg \phi$.

The semantics of CTL* is defined in terms of runs of a computation tree $T$. A run $r$ of $T$ is a sequence $s_0, s_1, s_2, \ldots$ of nodes of $T$, such that $s_{i+1}$ is a child of $s_i$ for all $i$. We write $r[0, \ldots, i]$ to denote the first $i+1$ nodes in the sequence $r$.

**Definition 2.2.6.** Given a set of atomic assertions $AP$ a CTL* assertion $\phi$, a computation tree $T$, and run $r$ of $T$, and a position $i$ of $r$, we interpret $\phi$ as follows (in addition to the semantics of LTL):

$$T, r, i \models E\phi \iff T, r', i \models \phi \text{ for some } r' \text{ in } T \text{ such that } r[0, \ldots, i] = r'[0, \ldots, i]$$

**The $\mu$-Calculus**

The linear-time $\mu$-calculus is a variant of the $\mu$-calculus, which is branching time. The $\mu$-calculus is an important logic since it subsumes all of the logics discussed in this chapter. (See Arnold and Niwiński for a detailed discussion of the $\mu$-calculus [1].)

**Definition 2.2.7.** Given a set of propositions $AP$ and a disjoint set of variables $\mathcal{X}$, the syntax of the **$\mu$-calculus** is as follows (with $p \in AP$ and $X \in \mathcal{X}$):

$$\phi \ := \ p \mid \neg p \mid X \mid \neg X \mid \phi \ \wedge \ \psi \mid \phi \ \vee \ \psi \mid \Box\phi \mid \Diamond\phi \mid \mu X.\phi \mid \nu X.\phi$$

The semantics of the $\mu$-calculus are analogous to that of the linear-time $\mu$-calculus, except in the case of $\Box\phi$ and $\Diamond\phi$. In these cases we require $\phi$ to hold at every possible next state or at one or more next states respectively.

### 2.2.3 Monadic Second-Order Logic

Monadic second-order logic (**MSO**) is a highly expressive logic at the limits of decidability. In general it subsumes the $\mu$-calculus, although it is equivalent in the case of finite graphs and (possibly infinite) trees [140]. Furthermore, any bisimulation invariant MSO formula is equivalent to some formula of the $\mu$-calculus [42]. Two states are bisimilar if they satisfy the same atomic propositions, and each transition from either of the states can be matched by a transition from the other, leading to another pair of bisimilar states. That is, only a state's behaviour is taken into account, rather than its unique identity. MSO is able to identify nodes, whereas the $\mu$-calculus can only reference the behaviour, or propositions satisfied.

It is rare to find examples of structures for which the $\mu$-calculus is decidable, but MSO is not. One such example is the class of graphs definable by *collapsible pushdown systems*, discussed in Section 2.8.7.

Second-order logic generalises first-order logic by allowing quantification over sets of elements as well as single elements. It is because of this restriction to sets, rather than relations of higher-arity, that the logic is termed *monadic*. We have the syntax:

$$\phi \ := \ x \in X \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \forall x.\phi \mid \forall X.\phi$$

where $x$ is a first-order variable and $X$ is a second-order variable. Several fundamental constructs may be considered abbreviations. For example:

$$\begin{array}{rcl} x = y & := & \forall X. x \in X \iff y \in X \\ X \subseteq Y & := & \forall x. x \in X \Rightarrow x \in Y \\ X = \emptyset & := & \forall Y. X \subseteq Y \end{array}$$

Over words we have the monadic second-order theory of a single successor. In this logic, we equip MSO with a successor relation $S(x, y)$ which holds if $y$ is the successor of $x$. We can then express $x < y$, $first(x)$, and in the case of finite words, $last(x)$:

$$\begin{array}{rcl} x < y & := & \neg(x = y) \wedge \forall X. (x \in X \wedge \forall z, z' (z \in X \wedge S(z, z') \Rightarrow z' \in Z) \Rightarrow y \in X) \\ first(x) & := & \forall y. x < y \vee x = y \\ last(x) & := & \forall y. y < x \vee x = y \end{array}$$

To express $x < y$, we assert that $y$ is an element of the set of successors of $x$. Similar interpretations over trees and graphs are obtained through the addition of edge relations.

## 2.3 Automata and Games

In this section we describe two important classes of automata and discuss their relationship with LTL. (For a detailed description of the automata-theoretic approach to LTL, see Vardi's survey paper [82].)

### 2.3.1 Büchi Automata

Automata can be used to define languages over a given alphabet $\Sigma$. We are interested in infinite runs of programs, and hence infinite words. Büchi automata can be used to calculate whether a given word is in our language. That is, whether a program run meets our specification.

**Definition 2.3.1.** A **Büchi automaton** $A$ is a tuple $(\Sigma, \mathcal{Q}, \mathcal{I}, \delta, \mathcal{F})$, where $\Sigma$ is a finite, non-empty, alphabet, $\mathcal{Q}$ is a finite set of states, $\mathcal{I} \subseteq \mathcal{Q}$ is the set of initial states, and $\delta : S \times \Sigma \to 2^{\mathcal{Q}}$ is a transition relation. The set $\mathcal{F} \subseteq \mathcal{Q}$ is a set of final states.

To determine whether a Büchi automaton accepts a word $w$ we proceed as follows. The automaton starts at an initial state. It reads the characters of $w$ one by one (from left to right). At each character, we take a transition allowed by the transition relation given the current state and character. We accept the word if we meet a state in $F$ an infinite number of times.

More formally, given a word $w = a_0, a_1, a_2, \ldots$ we define a run of the automaton on $w$ as a sequence of states $q_0, q_1, q_2, \ldots$ where $q_0 \in \mathcal{I}$ and $q_{i+1} \in \delta(q_i, a_i)$ for all $i$. If $\delta(q, a)$ is a singleton for all $q$ and $a$ then the automaton is deterministic — for any given word there is only one possible run. If we have a choice of next states then the automaton is nondeterministic, and for any given word a number of runs may exist.

In order to define acceptance we define the limit of a run $r$ as $lim(r) = \{ q \mid q = q_i$ for infinitely many $i \}$. A word is accepted if there is a run $r$ of the automaton over the given word such that $lim(r) \cap F \neq \emptyset$. That is, a run exists where a final state occurs infinitely often.

The language of an automaton $A$ is written $\mathcal{L}(A)$ and is defined as the set of all words accepted by the automaton.

**Proposition 2.3.1** ([84])**.** *Given an LTL formula $\phi$, one can build a Büchi automaton $A_\phi = (\Sigma, \mathcal{Q}, \mathcal{I}, \delta, \mathcal{F})$, where $\Sigma = 2^{AP}$ and $|\mathcal{Q}|$ is $2^{O(|\phi|)}$, such that $\mathcal{L}(A_\phi)$ is exactly the set of computations satisfying the formula $\phi$.*

### 2.3.2 Alternating Automata

Alternating automata have been studied by Brzozowski and Leiss [56] and Chandra, Kozen and Stockmeyer [16]. As we noted earlier, a Büchi automaton may be non-deterministic. The automaton will accept if any of these transitions results in an accepting path. That is, there exists a transition leading to an accepting path. Alternating automata go a step further, allowing universal as well as existential quantification. Two definitions of alternating automata have been introduced in the literature [56, 16]. The class of languages definable by both types of automata are equivalent. In the first definition, the alternation appears on the states of the automata, whereas in the second, the alternation appears in the transition relation.

**Definition 2.3.2** ([16])**.** An **Alternating Büchi Automaton** (ABA) $A$ is given as a tuple $(\Sigma, \mathcal{Q}, \mathcal{I}, \delta, E, U, \mathcal{F})$, where $\Sigma$ is a finite, non-empty, alphabet, $\mathcal{Q}$ is a finite set of states, $\mathcal{I} \subseteq \mathcal{Q}$ is the set of initial states and $\delta : \mathcal{Q} \times \Sigma \rightarrow 2^{\mathcal{Q}}$ is a transition relation. The set $\mathcal{F} \subseteq \mathcal{Q}$ is a set of final states and $E$, $U$ is a partition of $\mathcal{Q}$ into existential and universal states.

Informally, a word is accepted by an alternating automaton if, from the current state, either all transitions lead to an accepting run (if we are in a universal state), or there is a transition which leads to an accepting run (if we are in an existential state).

**Definition 2.3.3** ([56])**.** An **Alternating Büchi Automaton** (ABA) $A$ is given as a tuple $(AP, \mathcal{Q}, \mathcal{I}, \delta, \mathcal{F})$, where $AP$ is a set of atomic propositions, $\mathcal{Q}$ is a finite set of states, $\mathcal{I} \subseteq \mathcal{Q}$ is the set of initial states, and $\delta : \mathcal{Q} \rightarrow \mathcal{B}(\mathcal{Q} \cup AP)$ is a transition relation, where $\mathcal{B}(\mathcal{Q} \cup AP)$ is the set of all boolean formulae over the atomic propositions in $\mathcal{Q} \cup AP$ and states in $\mathcal{Q}$ only occur positively. The set $\mathcal{F} \subseteq \mathcal{Q}$ is a set of final states.

In this model of automata we introduce alternation via the transition relation. At each point in a run of the automaton we evaluate the transition formula for the current state. The value of the atomic propositions is given by the character of input that is being read. We are looking for models of $\delta(q)$ (where $q$ is the current state). If we can find a model such that an accepting run can be found for all states in the model, then the automaton accepts. For example, if $\delta(q) = (q_1 \wedge q_2) \vee (q_3 \wedge q_4)$ we accept if accepting runs can be found from states $q_1$ and $q_2$, or if accepting runs can be found from states $q_3$ and $q_4$. When alternation is by state rather than boolean formula, it can be observed that $\delta(q) = (q_1 \wedge q_2) \vee (q_3 \wedge q_4)$ can only be encoded by introducing intermediate states. Hence, boolean formulae may allow automata with fewer states, although the transition relation is more complex.

It is known that alternating automata are no more expressive than non-deterministic automata, but exponentially more succinct. In fact, there is a linear translation from LTL to alternating automaton. However, this does not improve the complexity of LTL satisfiability because the complexity of testing for language emptiness is exponential for an alternating automaton, as opposed to linear for a non-deterministic automaton.

Two important subclasses of alternating Büchi automata are **Weak ABA** (WABA) [39] and **Linear Weak ABA** (LWAA). An ABA is *weak* if its state-set can be partitioned into sets $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$ such that if some state $q'$ appears in $\Delta(q, a)$ for some $q$ and $a$, then $q \in \mathcal{Q}_i$ and $q' \in \mathcal{Q}_j$ with $j \leq i$. That is, once a run has left a partition, it cannot return. Furthermore, each partition is either accepting, or not-accepting. That is, for all $i$, $\mathcal{Q}_i \subseteq \mathcal{F}$ or $\mathcal{Q}_i \cap \mathcal{F} = \emptyset$. A WABA is *linear* if its transition structure contains no cycles containing two states or more. That is, only self loops are permitted.

We present the following equivalences between alternating automata and linear time logics.

**Proposition 2.3.2** ([79])**.**

1. *For every $\phi \in \mu TL$ there is a WABA $A_\phi$ with $\mathcal{L}(A_\phi) = \mathcal{L}(\phi)$.*
2. *For every WABA $A$ there is a $\phi_A \in \mu TL$ with $\mathcal{L}(\phi_A) = \mathcal{L}(A)$.*

Similarly, for LTL:

**Proposition 2.3.3.**

1. *For every $\phi \in LTL$ there is an LWAA $A_\phi$ with $\mathcal{L}(A_\phi) = \mathcal{L}(\phi)$ [38].*
2. *For every LWAA $A$ there is a $\phi_A \in LTL$ with $\mathcal{L}(\phi_A) = \mathcal{L}(A)$ [31, 51].*

### 2.3.3 Games

Games are an alternative characterisation of alternating automata. A game is played over a graph, consisting of a (possibly infinite) set of nodes and directed edges. The state-set is partitioned into positions belonging to player one, **Éloïse**, and player two, **Abelard**[1]. An initial state is designated. This state belongs to either Éloïse or Abelard, and the appropriate player is able to choose which edge to follow. This edge is followed to the next state and a new round of the game begins. We consider games in which plays produce infinite paths. If a player is unable to choose a next move, they forfeit the game. In the case of an infinite play, the winner of the game is selected using an appropriate winning condition.

**Definition 2.3.4.** A **game** is a tuple $(\mathcal{Q}, \Delta, \mathcal{W})$ where $\mathcal{Q} = \mathcal{Q}_E \uplus \mathcal{Q}_A$ is a set of states belonging to Éloïse and Abelard respectively, $\Delta \subseteq \mathcal{Q} \times \mathcal{Q}$ is a transition function, and $\mathcal{W} \subseteq \mathcal{Q}^\omega$ is a winning condition.

A play from an initial state $q_0$ generates a sequence $q_0, q_1, \ldots$. Éloïse wins the game if $q_0, q_1, \ldots \in \mathcal{W}$.

#### The Borel Hierarchy

The Borel Hierarchy [17] provides a means of classifying the expressivity of logics over words. It is a hierarchy of word languages. These word languages can be used to classify winning conditions on games.

At the bottom of the hierarchy is the class of *open sets*, denoted $\Sigma_1$. These sets are of the form $W \cdot \Sigma^\omega$ for some finite alphabet $\Sigma$ and set of finite words $W \subseteq \Sigma^*$. Logics at level $\Sigma_1$ of

[1] In the literature many different names are used. For example, Player 0 and Player 1, Spoiler and Duplicator, or Adam and Eve.

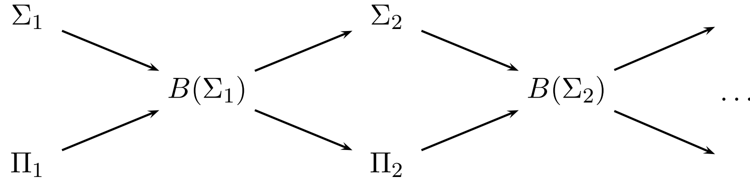



Figure 2.3: The Borel Hierarchy [17]

the Borel hierarchy are limited in their expressiveness: they are able to reason about a finite prefix of a run, but not its infinite behaviour.

Also at the bottom of the hierarchy is the class $\Pi_1$, which is the class of languages that are the complement of a language in $\Sigma_1$. The Borel hierarchy is constructed by forming boolean combinations, infinite unions and complements of classes lower in the hierarchy.

**Definition 2.3.5.** Let $\Sigma$ be a finite alphabet. The **Borel hierarchy** is defined inductively:

$$\begin{array}{rcl} \Sigma_1 & = & \{ \ W \cdot \Sigma^\omega \mid W \subseteq \Sigma^* \ \} \\ \Pi_n & = & \{ \ \overline{A} \mid A \in \Sigma_n \ \} \\ \Sigma_{n+1} & = & \{ \ \bigcup_{i>0} A_i \mid \forall i > 0 A_i \in \Pi_n \ \} \end{array}$$

Let $B(\Sigma_n)$ denote the class of boolean combinations of $\Sigma_n$ sets. The Borel hierarchy is shown in Figure 2.3, where each arrow indicates a strict inclusion. A set $A \in \Sigma_n$ with $A \notin \Pi_n$ is called a *true*-$\Sigma_n$-set.

### Winning Conditions

We restrict our attention to three important conditions: reachability, Büchi and parity:

- A **reachability** condition requires Éloïse to force play to a state in a set $\mathcal{Q}_r \subseteq \mathcal{Q}$. That is,

  $$\mathcal{W} = \mathcal{Q}^* \cdot \mathcal{Q}_r \cdot \mathcal{Q}^\omega$$

  Hence, the reachability condition is a $\Sigma_1$ winning condition.

- A **Büchi** winning condition is as defined in Section 2.3.1. Alternatively,

  $$\mathcal{W} = \bigcap_{n>0} (\mathcal{Q}^* \cdot F)^n \cdot \mathcal{Q}^\omega = \overline{\bigcup_{n>0} \overline{(\mathcal{Q}^* \cdot F)^n \cdot \mathcal{Q}^\omega}}$$

  That is, the complement of a $\Sigma_2$ condition. Hence the Büchi condition is a $\Pi_2$ condition.

- A **parity** condition assigns to each state a priority from a set $\{1, \ldots, m\}$ for some $m$. Éloïse wins the game if the lowest priority occurring infinitely often in the play is even. Let $P_i$ be the set of states of priority $i \in \{1, \ldots, m\}$. Furthermore, let $\phi(P_i)$ be the formula requiring a state in $P_i$ to occur infinitely often in the play. The formula given above to describe Büchi games can be used to define $\phi(P_i)$ by replacing occurrences of $F$ with $P_i$. Hence $\phi(P_i)$ is a $\Pi_2$ condition. The parity condition states that, for all odd

priorities $i$, $i$ does not occur infinitely often, or, some even priority less than $i$ occurs infinitely often. That is,

$$\mathcal{W} = \bigcap_{\substack{i \in \{1,\ldots,m\} \\ odd(i)}} \left( \phi(P_i) \;\Rightarrow\; \bigcup_{\substack{0 < j < i \\ even(j)}} \phi(P_j) \right)$$

Hence, the parity condition is the boolean combination of several $\Pi_2$ conditions. Thus, it is in $B(\Sigma_2)$.

### 2.3.4 Parity Games and the $\mu$-Calculus

Checking a Kripke structure against a $\mu$-calculus specification can be straightforwardly reduced to determining the winner of a parity game [32]. To prevent play terminating spuriously, we assume that every state $q$ of the Kripke structure has a next state $q'$ (that is $(q, q') \in \Delta$). This can be ensured through the addition of a looping sink state for all terminating computations.

The states of the game are formed by the product of the Kripke structure $S$ and the sub-formulae of specification $\phi$. The states and moves available to Éloïse are of the form:

1. From a state $(q, \psi_1 \vee \psi_2)$, Éloïse can move to $(q, \phi_1)$ or $(q, \phi_2)$.
2. From a state $(q, \Diamond\psi)$, Éloïse can move to $(q', \phi)$ where $(q, q') \in \Delta$.

That is, Éloïse resolves existential aspects of the computation. Conversely, Abelard resolves universal choices:

1. From a state $(q, \psi_1 \wedge \psi_2)$, Abelard can move to $(q, \phi_1)$ or $(q, \phi_2)$.
2. From a state $(q, \Box\psi)$, Éloïse can move to $(q', \phi)$ where $(q, q') \in \Delta$.

Intuitively, existential aspects of the computation permit a choice where only one successful run is required. In these cases, Éloïse can choose (fortuitously). Universal aspects of the computation require all possible choices to be successful. Hence, we allow Abelard to attempt to defeat Éloïse by making the most difficult choice. Finally, several states of the game have only a single successor and ownership is not important:

1. From a state $(q, \sigma X.\psi)$ play moves to $(q, X)$, where $\sigma \in \{\mu, \nu\}$.
2. From a state $(q, X)$ play moves to $(q, \psi)$ where $\sigma X.\psi$ is the bounding formula of the variable $X$ (with $\sigma \in \{\mu, \nu\}$).

Finally, there are four kinds state from which no moves are possible:

$$(q, p) \qquad (q, \neg p) \qquad (q, X) \qquad (q, \neg X)$$

where $p$ is an atomic proposition and $X$ is a free variable in $\phi$. In these cases, Éloïse wins the game if the state $q$ satisfies $p$, $\neg p$, $X$ or $\neg X$ respectively.

Any infinite play of the game must cycle through a bound variable $X$. These variables may be ordered by a sub-formula relation. That is, the priority of $X$ is greater than the priority of $Y$ if $\sigma_X X.\psi_X$ appears as a sub-formula of $\sigma_Y Y.\psi_Y$ for $\sigma_Y, \sigma_X \in \{\mu, \nu\}$. Furthermore, recall that $\mu$ corresponds to finite looping, and $\nu$ corresponds to infinite looping. That is, a $\nu$ variable may be returned to an infinite number of times — *without meeting a variable of lower priority* — whilst a $\mu$ variable cannot. This is a parity condition, where $\nu$ variables are assigned an even priority and $\mu$ variables are assigned an odd priority.

Once priorities matching the above requirements have been assigned to variables and the appropriate sub-formulae (to give a complete priority assignment), we have a game with a slight modification of the parity game algorithm: either the parity condition is satisfied, or play terminates in a winning state for Éloïse. We may reduce this to a true parity condition by adding sink states of appropriate priorities.

**Theorem 2.3.1.** *The model-checking problem for a Kripke structure $S$ and a $\mu$-calculus formula $\phi$ can be reduced to determining whether Éloïse wins a parity game $\mathcal{G}_{S,\mu}$.*

## 2.4 Infinite State Systems

Finite state systems can be used to model hardware and a limited range of software. To model programs with (for example) recursion, variables ranging over infinite sets — such as the integers — or unbounded data structures, infinite state systems are required. Figure 2.4 shows a number of infinite systems that have been studied, and which we discuss briefly in this section. Edges in the graph show the approximate generalisation relationship between the various models. Because a number of these systems have been introduced in recent work, their relationship has not been fully explored. Additionally, these systems may be interpreted as generators of word languages, trees or labelled/unlabelled graphs. Figure 2.4 does not explicate these differences.

We are primarily concerned with sequential infinite state systems. These models will be discussed in detail in the remainder of this chapter. Generalising sequential models by adding concurrency leads to the undecidability of reachability very quickly. In fact, any system that is both **context-sensitive** (procedures always return to their caller) and **synchronisation-sensitive** (threads can communicate) has an undecidable reachability problem [50]. Consequently, a number of attempts have been made to find restrictions of infinite state systems with concurrency that have good model-checking properties. Table 2.1 and Table 2.2 give the complexities of a selection of model-checking problems for these systems.

### 2.4.1 Sequential Systems

The **Basic Process Algebra** (BPA) [55] is a simple algebra comprising actions, sequential composition, non-deterministic choice and recursion. A process is defined by a set of definitions $\{ X_i \rightarrow e_i \mid 1 \leq i \leq k \}$ where $e_i$ is an expression of the form:

$$e \ ::= \ a \mid X \mid e_1 + e_2 \mid e_1 \cdot e_2$$

with $a \in \Sigma$ for some finite alphabet $\Sigma$. For convenience, the sequential composition operator $\cdot$ is often omitted. Figure 2.5 shows the evolution of an example process $\{ X \rightarrow a + bYc, Y \rightarrow dX \}$.

Reachability Undecidable

ADPN

DCPS

Sync. PA

CPDS PreRec.

CDPN

PRS

Sequential

HOPDA

MPDA

DPN

PAD

PAN

Concurrent

PDA/RSM

PA

Petri-Nets

BPA

BPP

Finite State

Reachability Decidable

Figure 2.4: Connections between various infinite state systems

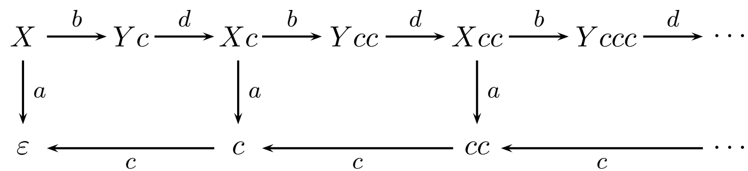



Figure 2.5: Transition graph for the BPA process $\{\ X \to a + bYc, Y \to dX\ \}$

| | Reach. | LTL |
|---|---|---|
| Finite State | Linear [107] | PSPACE-c [21] |
| BPA | P | EXPTIME-c [112] |
| PDS | P [3] | EXPTIME-c [3] |
| PreRec. | EXPTIME | EXPTIME-c [90] |
| HOPDS | (n-1)-EXPTIME-c [59] | n-EXPTIME |
| CPDS | n-EXPTIME | n-EXPTIME |
| BPP | NP-c [62] | PSPACE-c [108] |
| PA | NP-c [111] | undec. [9] |
| PAD | NP-c [112] | undec. |
| Petri-Nets | decidable [48] | decidable [60] |
| PAN/PRS | decidable [112] | undec. |
| DPN | P [7] | — |
| CDPN | decidable [7] | undec. |

Table 2.1: Model-checking complexities for various infinite state systems

| | EF | CTL | $\mu$-Calculus |
|---|---|---|---|
| Finite State | Linear | Linear [47] | NP $\cap$ co-NP [44] |
| BPA | PSPACE-c [113] | EXPTIME | EXPTIME-c [113] |
| PDS | PSPACE-c [54] | EXPTIME [53] | EXPTIME-c [53] |
| PreRec. | EXPTIME-c [120] | EXPTIME-c | EXPTIME-c [87] |
| HOPDS | n-EXPTIME | n-EXPTIME | n-EXPTIME-c [127, 129] |
| CPDS | n-EXPTIME | n-EXPTIME | n-EXPTIME-c |
| BPP | PSPACE-c [112] | undec. [63] | undec. |
| PA | decidable [110] | undec. | undec. |
| PAD | decidable [112] | undec. | undec. |
| Petri-Nets | undec. [112] | undec. | undec. |
| PAN/PRS | undec. | undec. | undec. |
| DPN | — | — | — |
| CDPN | — | — | undec. |

Table 2.2: Branching-time model-checking complexities for various infinite state systems

**Pushdown Automata** are defined formally in Section 2.6. Informally, they are rewrite systems with rules of the form $p\gamma \xrightarrow{a} qw$ where $p, q$ are control states, $\gamma$ is a letter of a stack alphabet, and $a$ is an action. The state of a PDA is a configuration $pw'$ with control state $p$ and stack $w'$. A rule of the above form is applied when the control states match and $\gamma$ is the first character of the word $w'$. The rule generates an $a$-labelled transition to the configuration $qww'$. If $w$ is the empty word, then the top character is *pop*-ed from the stack. Otherwise characters are *push*-ed onto the stack. PDAs generate infinite transition graphs and every **context-free** graph can be generated by a PDA [40]. A BPA process may be described by a Pushdown Automaton with a single control state [88, 37]. **Pushdown Systems** (PDSs) are pushdown automata without actions and hence produce unlabelled graphs. This interpretation is the subject of this thesis.

A generalisation of pushdown systems are **Prefix-Recognisable Systems** (PreRec.) [35, 36]. Rules of a prefix-recognisable system are of the form $(p, \alpha, \beta, \gamma, p')$. A configuration $pw$ has a transition to $p'w'$ iff $w = w_1w_2$ and $w' = w'_1w_2$ with $w_1 \in \alpha$, $w_2 \in \beta$ and $w'_1 \in \gamma$. These systems form part of a hierarchy of graphs discussed in Section 2.8.4.

**Higher-Order Pushdown Automata** (HOPDA) allow a more complicated stack structure than PDA. A second-order PDA has a stack of stacks. Similarly, a third-order PDA has a stack of stacks of stacks, and so on. An order-$n$ PDA has push and pop commands for every $1 \leq l \leq n$. When $l > 1$ a pop command removes the topmost order-$l$ stack. Conversely, the push command duplicates the topmost order-$l$ stack. Maslov introduced HOPDA as generators of word languages, and he shows that the hierarchy is strict [20]. That is, an order-$n$ PDA is strictly more expressive than an order-$(n-1)$ PDA. **Higher-order pushdown systems** (PDSs) are higher-order PDA viewed as generators of infinite trees or graphs. We discuss HOPDSs formally in Section 2.8.

Knapik *et al.* [132] have shown that the trees generated by deterministic order-$n$ PDSs are exactly those that are generated by order-$n$ **recursion schemes** satisfying a constraint called **safety**. MSO decidability for trees generated by arbitrary (i.e. not necessarily safe) Higher-Order Recursion Schemes (HORS) has been shown by Ong [29]. A variant kind of higher-order PDSs called *Collapsible Pushdown Systems* (extending *panic automata* [133] or *pushdown automata with links* [69] to all finite orders) have recently been shown to be equi-expressive with HORSs for generating ranked trees [76]. These automata allow **collapse** operations as well as the usual push and pop commands. When a character is pushed onto the top stack, a link is created to a lower stack of a specified order. This link always points to the same stack position even if it is copied to another part of the stack (during a higher-order push command). A collapse command reduces the stack to the position indicated by the link decorating the top character.

### 2.4.2 Concurrent Systems

**Basic Parallel Processes** are the concurrent analogue of the Basic Process Algebra, however, the two are incomparable [119]. Like, BPA, a process is defined by a set of definitions $\{ X_i \rightarrow e_i \mid 1 \leq i \leq k \}$. However, $e_i$ is an expression of the form:

$$e \ ::= \ \varepsilon \mid ae_1 \mid X \mid e_1 + e_2 \mid e_1 \parallel e_2$$

with $a \in \Sigma$ for some finite alphabet $\Sigma$. Note that although sequential composition is not allowed, expressions of the form $a_1 \ldots a_m X$ are permitted. Parallel composition $(e_1 \parallel e_2)$

is also enabled, but there is no communication between processes. BPP are equivalent to **communication-free** Petri-Nets [61].

The **Process Algebra** [55] is BPP with sequential composition, and hence is a superset of BPA and BPP. **PAD** Systems were introduced by Mayr as a generalisation of PA and PDA (PA+PD) [114, 112]. Like PA, PAD do not allow synchronisation, but, like PDA, return values can be taken into account. Similarly, **PAN** — also introduced by Mayr — combine PA processes and Petri-Nets [109, 112].

PAD are a restriction of **Process Rewrite Systems**. PRSs are defined over terms with the syntax,

$$t \ ::= \ \varepsilon \mid X \mid t_1 \cdot t_2 \mid t_1 \parallel t_2$$

Parallel composition ($\parallel$) is commutative and associative and sequential composition ($\cdot$) is associative. Each PRS has a finite set of rules of the form $t_1 \xrightarrow{a} t_2$ which can be applied modulo commutativity and associativity and the obvious inference rules:

$$\frac{t_1 \xrightarrow{a} t_1'}{t_1 \parallel t_2 \xrightarrow{a} t_1' \parallel t_2} \qquad \frac{t_2 \xrightarrow{a} t_2'}{t_1 \parallel t_2 \xrightarrow{a} t_1 \parallel t_2'} \qquad \frac{t_1 \xrightarrow{a} t_1'}{t_1 \cdot t_2 \xrightarrow{a} t_1' \cdot t_2}$$

Together, finite state systems, BPA, BPP, PDA, PA, Petri-Nets, PAD, PAN and PRS form the PRS-Hierarchy [114], which is the subject of Mayr's doctoral thesis [112].

In 2005, Bouajjani, Müller-Olm and Touili introduced **Dynamic Pushdown Networks** (DPN) and **Constrained Dynamic Pushdown Networks** (CDPN) [7]. DPN are an extension of PDA. In addition to the usual pushdown rules $p\gamma \xrightarrow{a} qw$, DPN allow processes to spawn child processes via commands of the form $p\gamma \xrightarrow{a} p_1 w_1 \rhd p_2 w_2$. These commands perform the usual pushdown update, but also create a child process with control state $p_2$ and initial stack contents $w_2$.

DPN do not permit inter-process communication. A limited form of communication is introduced by Constrained DPN. CDPN allow processes to monitor the control states of their immediate children. To do this, the rules of the system are decorated with guards. That is, rules take the form $\phi : p\gamma \xrightarrow{a} p_1 w_1 (\rhd p_2 w_2)$ where $\phi$ is a regular constraint over sequences of control states. A guarded rule can only be applied to a process if the control states of its children, when arranged in order of age, satisfy the constraint $\phi$. To ensure regularity of reachability sets, constraints must be *stable*. Intuitively, this means that if the current states of the child processes satisfy $\phi$, then the constraint will remain satisfied as the system proceeds.

**Multi-set Pushdown Systems** (MPDS) were introduced by Sen and Viswanathan in 2006 [73]. MPDSs extend PDSs with a multi-set of asynchronous procedure calls. They are designed to model concurrent systems with asynchronous procedure calls. That is, when a procedure is called, control is returned to the caller immediately while the callee is executed in parallel. Communication does not occur between processes. During normal pushdown execution, stack symbols may be added to a multi-set which is a component of MPDS configurations. Whenever the stack is empty, a stack symbol can be retrieved from the multi-set and placed onto the stack. In this way, MPDS model a serialised execution of the concurrent systems described above.

Since context-sensitive, synchronisation-sensitive infinite state systems have an undecidable reachability problem, a number of undecidable models have been proposed with abstraction-based model-checking algorithms. Examples of such systems are **Synchronised**

**PA** [6], **Dynamic Concurrent Pushdown Systems** (DCPSs) [122] and **Asynchronous Dynamic Pushdown Networks** (ADPNs) [4].

There are several kinds of abstraction in the literature. Two abstraction paradigms are **finite-chain abstraction** and **commutative abstraction** (E.g. [5]). Finite-chain abstractions produce finite sequences from infinite runs. For example **first occurrence abstraction** records the first appearance of a character only. The result is a string in which each character appears at most once, and in the order that they were first seen. A sequence *abbaccaabac* would abstract to *abc*.

Commutative abstractions are insensitive to the order in which characters occur. A simple example of this kind of abstraction are **label bit-vectors** which simply record whether a character has appeared.

An alternative approach, introduced by Qadeer and Rehof, is **context-bounded model-checking** [122]. Qadeer and Rehof obtain decidability by bounding the number of context-switches. If parallel threads are serialised to run on a single processor, a context-switch occurs when control of the process passes from one thread to another.

## 2.5 Alternation

In the sequel we will introduce several kinds of alternating automata. For convenience, we will use a non-standard definition of alternating automata that is equivalent to the standard definitions of Brzozowski and Leiss [56] and Chandra, Kozen and Stockmeyer [16]. Similar definitions have been used for the analysis of pushdown systems by Bouajjani *et al.* [3] and Cachat [127]. The alternating transition relation $\Delta \subseteq \mathcal{Q} \times \Gamma \times 2^{\mathcal{Q}}$ — where $\Gamma$ is an alphabet and $\mathcal{Q}$ is a state-set — is given in disjunctive normal form. That is, the image $\Delta(q, \gamma)$ of $q \in \mathcal{Q}$ and $\gamma \in \Gamma$ is a set $\{Q_1, \ldots, Q_m\}$ with $Q_i \in 2^{\mathcal{Q}}$ for $i \in \{1, \ldots, m\}$. When the automaton is viewed as a game, Éloïse— the existential player — chooses a set $Q \in \Delta(q, \gamma)$; Abelard— the universal player — then chooses a state $q \in Q$. The existential component of the automaton is reflected in Éloïse's selection of an element $(q, \gamma, Q)$ from $\Delta$ for a given $q$ and $\gamma$. Abelard's choice of a state $q$ from $Q$ represents the universal aspect of the automaton.

## 2.6 Order-1 Pushdown Systems and Automata

### 2.6.1 Definition

(Order-1) Pushdown systems are a simple kind of infinite state system. A state, or **configuration**, of the system consists of a control state $p \in \mathcal{P}$ — where $\mathcal{P}$ is a finite set — and a store $\gamma$. The store behaves like stack: only the uppermost symbol is visible and the store can be modified using push and pop operations.

#### Order-1 Stores

We begin by defining pushdown stores and their operations.

**Definition 2.6.1** (1-stores)**.** The set $C_1^{\Sigma}$ of **1-stores** over an alphabet $\Sigma$ is the set of words of the form $[a_1 \ldots a_m]$ with $m \geq 0$ and $a_i \in \Sigma$ for all $i \in \{1, \ldots, m\}$, $[ \notin \Sigma$ and $] \notin \Sigma$.

There are three operations defined over 1-stores: $push_w$ (for all $w \in \Sigma^*$), $pop_1$ and $top_1$:

$$\begin{array}{rcl} push_w[a_1 \dots a_m] & = & [wa_2 \dots a_m] \\ pop_1[a_1 \dots a_m] & = & [a_2 \dots a_m] \\ top_1[a_1 \dots a_m] & = & a_1 \end{array}$$

Since $pop_1 = push_\varepsilon$, we may consider the $pop_1$ command to be an abbreviation. Let $\mathcal{O}_1 = \{ push_w \mid w \in \Sigma^* \}$.

### Order-1 Pushdown Systems

We are now ready to define an order-1 pushdown system.

**Definition 2.6.2.** An **order-1 pushdown system** is a tuple $(\mathcal{P}, \mathcal{D}, \Sigma)$ where $\mathcal{P}$ is a finite set of control states, $\mathcal{D} \subseteq \mathcal{P} \times \Sigma \times \mathcal{O}_1 \times \mathcal{P}$ is a finite set of commands and $\Sigma$ is a finite alphabet.

A configuration is a pair $\langle p, \gamma \rangle$ with $p \in \mathcal{P}$ and $\gamma \in C_1^\Sigma$. We have a transition $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ iff we have $(p, a, push_w, p') \in \mathcal{D}$, $top_1(\gamma) = a$ and $push_w(\gamma) = \gamma'$.

### Alternating Order-1 Pushdown Systems

We may also define alternating pushdown systems:

**Definition 2.6.3.** An **alternating order-1 pushdown system** is a tuple $(\mathcal{P}, \mathcal{D}, \Sigma)$ where $\mathcal{P}$ is a finite set of control states, $\mathcal{D} \subseteq \mathcal{P} \times \Sigma \times 2^{\mathcal{O}_1 \times \mathcal{P}}$ is a finite set of commands and $\Sigma$ is a finite alphabet.

A configuration is a pair $\langle p, \gamma \rangle$ with $p \in \mathcal{P}$ and $\gamma \in C_1^\Sigma$. We have a transition $\langle p, \gamma \rangle \hookrightarrow C$ iff we have $(p, a, \{(push_{w_1}, p_1), \dots, (push_{w_m}, p_m)\}) \in \mathcal{D}$, $top_1(\gamma) = a$ and,

$$C = \{ \langle p_1, push_{w_1}(\gamma) \rangle, \dots, \langle p_m, push_{w_m}(\gamma) \rangle \}$$

The transition relation generalises to sets of configurations via the following rule:

$$\frac{\langle p, \gamma \rangle \hookrightarrow C}{C' \cup \langle p, \gamma \rangle \hookrightarrow C' \cup C} \quad \langle p, \gamma \rangle \notin C'$$

Finally, we define $\stackrel{*}{\hookrightarrow}$ to be the transitive closure of $\hookrightarrow$.

### Order-1 Pushdown Automata

We can augment pushdown systems with an input alphabet $\Gamma$ to define Pushdown Automata (PDA). These automata can be used to accept words or generate edge-labelled trees and graphs.

**Definition 2.6.4.** An **order-1 pushdown automaton** is a tuple $(\mathcal{P}, \mathcal{D}, \Sigma, \Gamma, c_0, \mathcal{W})$ where $\mathcal{P}$ is a finite set of control states, $\mathcal{D} \subseteq \mathcal{P} \times \Gamma \times \Sigma \times \mathcal{O}_1 \times \mathcal{P}$ is a finite set of commands and $\Sigma$, $\Gamma$ are finite alphabets. The initial configuration is $c_0$ and $\mathcal{W} \subseteq (C_1^\Sigma)^*$ is an acceptance condition (if the automaton is defined as an acceptor of word languages).

A configuration is a pair $\langle p, \gamma \rangle$ with $p \in \mathcal{P}$ and $\gamma \in C_1^\Sigma$. We have a transition $\langle p, \gamma \rangle \stackrel{\alpha}{\hookrightarrow} \langle p', \gamma' \rangle$ iff we have $(p, \alpha, a, push_w, p') \in \mathcal{D}$, $top_1(\gamma) = a$ and $push_w(\gamma) = \gamma'$. When defined as an acceptor of word languages, a PDA accepts a word $\alpha_0, \alpha_1, \dots$ iff there exists a run $c_0 \stackrel{\alpha_0}{\hookrightarrow} c_1 \stackrel{\alpha_1}{\hookrightarrow} \dots$ such that $c_0 c_1 \dots \in \mathcal{W}$.

### 2.6.2 Examples

We give two examples of order-1 pushdown systems. The first revisits the CD player of Section 2.1 whilst the second illustrates the use of pushdown systems to model procedural programs.

#### An Order-1 Pushdown CD Player

We return to the CD player example of Section 2.1 and, with the use of the store, add playlist functionality. The CD player has two control states. The CD player waits for input in the idle state $i$, and plays the programmed CDs in the second control state $p$. The commands of the system, together with some informal abbreviations are given below. Let $\Sigma = CDs \cup \{\bot\}$ for a finite set of CDs.

$$\begin{array}{rcll} \text{loadCD}(c) & = & (\text{i}, a, push_{ca}, \text{i}) & \text{for all } c \in \text{CDs and } a \in \Sigma \\ \text{playAll} & = & (\text{i}, a, push_a, \text{p}) & \text{for all } a \in \Sigma \\ \text{playCD}(c) & = & (\text{p}, c, pop_1, \text{p}) & \text{for all } c \in \text{CDs} \\ \text{done} & = & (\text{p}, \bot, push_\bot, \text{i}) & \end{array}$$

The behaviour of the system is illustrated in Figure 2.6.

#### Modelling Procedure Calls

A procedural program has a finite number of local and global variables. We restrict these variables to a finite range of values. Hence, each variable can be encoded by a number of bits. Therefore, we can assume each variable is an atomic proposition, taking the value $true$ or $false$. The values of the variables $v_1, \ldots, v_m$ can be encoded as a tuple in the set $\{true, false\}^m$.

Given a program with global variables $g_1, \ldots, g_m$ and local variable $l_1, \ldots, l_{m'}$ we construct the equivalent pushdown system as follows. The values of the global variables are stored in the control state. That is, $\mathcal{P} = \{true, false\}^m$. The values of the local variables are kept in the store alongside the current control point of the program. Thus, $\Sigma = N \times \{true, false\}^{m'}$ where $N$ is the set of program control points.

A procedure call is modelled by pushing the initial values of the local variables and the control point representing the start of the procedure onto the stack. Program statements within the procedure simply update the control state and local variables ($push_b$ where $b$ is the new assignment of values to variables). Finally, a return statement can be modelled by a pop command.

### 2.6.3 Local Model-Checking of Order-1 Pushdown Systems

In 1996, Walukiewicz presented an algorithm for determining the winner of a parity game played over order-1 pushdown systems [53]. A parity game over an order-1 pushdown system is defined by partitioning the control states of a pushdown system into two disjoint sets — those belonging to Éloïse and those belonging to Abelard. Each control state is also assigned a priority from the set $\{1, \ldots, m\}$.

Local model-checking asks whether a property holds from a given configuration $\langle p, [w] \rangle$. Play begins from this initial position and proceeds as follows: the player who owns the control state of the current configuration chooses an available command. This command is applied,

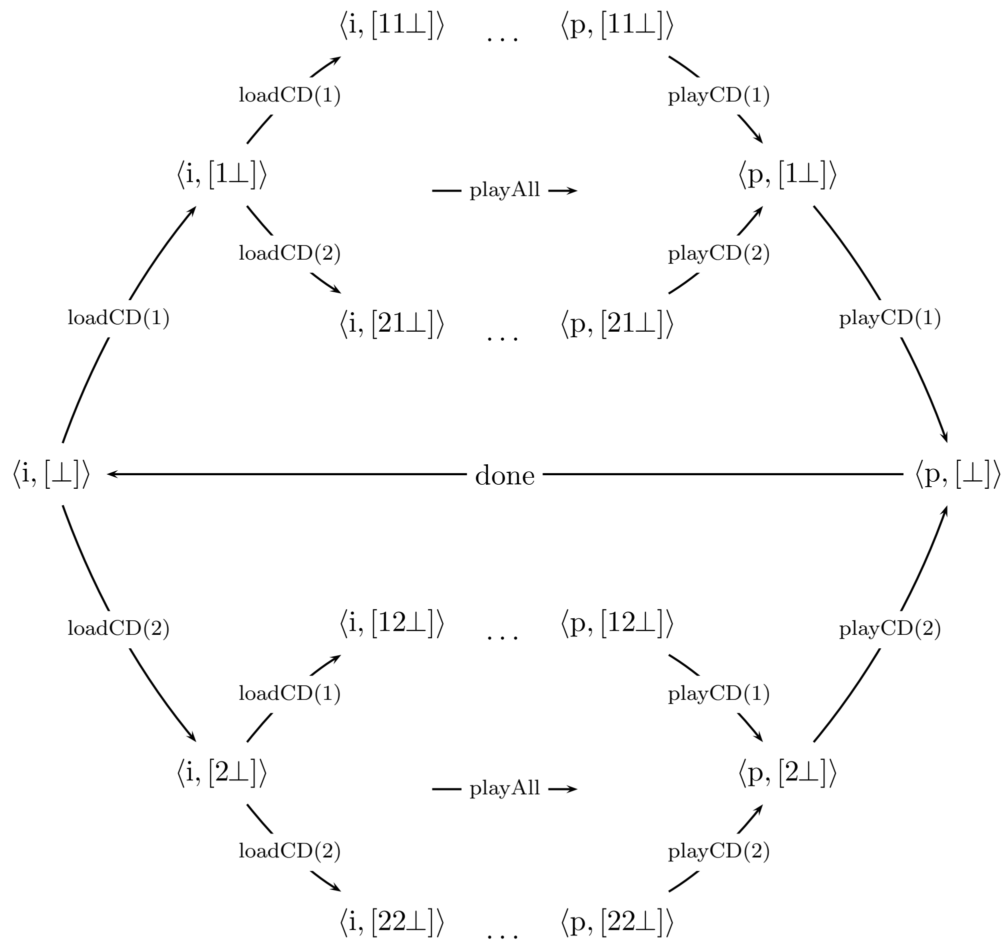



Figure 2.6: A CD player with CDs = $\{1, 2\}$

moving play to a new configuration $\langle p', [w'] \rangle$. Play continues in this manner until a player is unable to make a move — in which case they lose the game — or an infinite sequence of configurations is generated. In the latter case, we can form an infinite sequence of priorities from the control states of the configurations. Éloïse wins if the lowest priority occurring infinitely often is even. Otherwise Abelard is the winner.

We can assume without loss of generality that the initial configuration is of the form $\langle p, [\perp] \rangle$, where $\perp$ is a bottom of the stack symbol (which is neither pushed nor removed). We can modify any game so that the only available initial moves force play to any desired configuration $\langle p', [w] \rangle$, from which play proceeds as normal. We also permit a slightly different, but equivalent, set of pushdown commands: $push_a$, $pop_1$ and $skip$. These commands deal with single characters only: a $push_a$ command adds the character $a$ to the stack, $pop_1$ behaves as before, and $skip$ leaves the stack unchanged. We can simulate a $push_w$ by adding extra control states, from which the characters of $w$ are added in turn. In the other direction, we can perform a skip action by commands $(_, a, push_a, _)$, which replace the top character with itself.

**Definition 2.6.5.** An **order-1 pushdown parity game** is a tuple $(\mathcal{P}, \mathcal{D}, \Sigma, \Omega)$ where $\mathcal{P} = \mathcal{P}_A \uplus \mathcal{P}_E$ is a set of control states partitioned into states belonging to Abelard and states belonging to Éloïse, $\Sigma$ is a finite alphabet, $\mathcal{D} \subseteq \mathcal{P} \times \Sigma \times \mathcal{O}_1 \times \mathcal{P}$ is a set of pushdown commands and $\Omega : \mathcal{P} \to \{1, \ldots, m\}$ is a function assigning priorities to control states.

Solutions to finite state parity games are well known [45]. Walukiewicz shows that an (infinite state) parity game played over a pushdown system can be reduced to a finite state parity game. The intuition behind this reduction is the observation that the moves available from any given configuration depend on the control state and the symbol on the top of the stack. The potentially unbounded amount of information stored below the top of the stack is not visible.

Whenever a symbol is pushed onto the stack, there are two possibilities for the following play: either the symbol will be popped at a later stage, or it will remain on the stack for the rest of the (infinite) play. While-ever the symbol remains on the stack, the stack beneath it can be ignored. It is only when the symbol is removed from the stack that the rest of the stack is needed. In particular, only the new top symbol (which was pushed onto the stack earlier in the game) can influence the next move.

Using these observations, we can reduce a pushdown game to a finite state game. Play proceeds in the same manner as the pushdown game except, whenever a symbol is pushed onto the stack, Éloïse is required to predict the possible situations the game will be in when the symbol is finally popped. Abelard can choose to accept this prediction — in which case play moves to a predicted state — or challenge it. In the latter case the push operation occurs as normal, and play moves to a sub-game. In this sub-game, either another push operation is challenged — moving play to another sub-game — or the current top stack symbol is popped. In this case Éloïse's prediction is checked. If the situation matches the prediction, Éloïse wins the game. Otherwise Abelard is the winner. In the case of an infinite play, a modified version of the parity condition determines the winner.

When making a prediction, Éloïse is required to give a set of control states $A_c$ for each priority $c \in \{0, \ldots, m\}$. In making this prediction, Éloïse is asserting that she can ensure, when the pushed symbol has been popped, the control state will belong to some $A_c$ and that the smallest priority encountered between the push and the pop will be $c$. Whenever Abelard accepts a prediction, he chooses which control state to move to. This transition has a priority

corresponding to the minimal priority that would have been seen if the full play had taken place. This priority is taken into account when deciding whether the parity condition has been satisfied.

From a pushdown parity game $\mathcal{G}$ we define a finite game $\mathcal{M}$. The states of $\mathcal{M}$ have four core components: $\vec{A}$, $z$, $\theta$ and $p$:

- $\vec{A}$ — a tuple $(A_0, \ldots, A_m) \subseteq \mathcal{P}^m$ storing Éloïse's current prediction.
- $z$ — the symbol at the top of the current stack.
- $\theta$ — the smallest priority seen since $z$ was pushed onto the stack.
- $p$ — the current control state.

**Definition 2.6.6.** Given a pushdown parity game $\mathcal{G}$, the equivalent finite game $\mathcal{M}$ has the following states $\mathcal{Q} = \mathcal{Q}_A \uplus \mathcal{Q}_E$, for every $\vec{A}, \vec{A}_1, z, z_1, \theta, p, p_1$, all $c \in \{1, \ldots, m\}$ and a special symbol ?,

$$\begin{array}{lll} Check(\vec{A}, z, \theta, c, p) & Push(\vec{A}, z, \theta, p) & Pop(p) \\ Move((\vec{A}, z, \theta, p), (?, z_1, p_1)) & Move((\vec{A}, z, \theta, p), (\vec{A}_1, z_1, p_1)) & Err(p) \end{array}$$

The moves in $\mathcal{M}$ are:

$$\begin{array}{rcll} Check(\vec{A}, z, \theta, c, p) & \rightarrow & Check(\vec{A}, z, min(\Omega(p), \theta), c, p') & \text{if } (p, z, skip, p') \in \mathcal{D} \\ Check(\vec{A}, z, \theta, c, p) & \rightarrow & Pop(p') & \text{if } (p, z, pop_1, p') \in \mathcal{D}, p' \in A_{min(\Omega(p),\theta)} \\ Check(\vec{A}, z, \theta, c, p) & \rightarrow & Err(p') & \text{if } (p, z, pop_1, p') \in \mathcal{D}, p' \notin A_{min(\Omega(p),\theta)} \\ Check(\vec{A}, z, \theta, c, p) & \rightarrow & Move((\vec{A}, z, \theta, p), (?, z_1, p_1)) & \text{if } (p, z, push_{z_1}, p_1) \in \mathcal{D} \end{array}$$

$$\begin{array}{rcll} Push(\vec{A}, z, \theta, c, p) & \rightarrow & Check(\vec{A}, z, min(\Omega(p), \theta), c, p') & \text{if } (p, z, skip, p') \in \mathcal{D} \\ Push(\vec{A}, z, \theta, c, p) & \rightarrow & Pop(p') & \text{if } (p, z, pop_1, p') \in \mathcal{D}, p' \in A_{min(\Omega(p),\theta)} \\ Push(\vec{A}, z, \theta, c, p) & \rightarrow & Err(p') & \text{if } (p, z, pop_1, p') \in \mathcal{D}, p' \notin A_{min(\Omega(p),\theta)} \\ Push(\vec{A}, z, \theta, c, p) & \rightarrow & Move((\vec{A}, z, \theta, p), (?, z_1, p_1)) & \text{if } (p, z, push_{z_1}, p_1) \in \mathcal{D} \end{array}$$

and finally,

$$\begin{array}{rcll} Move((\vec{A}, z, \theta, p), (?, z_1, p_1)) & \rightarrow & Move((\vec{A}, z, \theta, p), (\vec{A}_1, z_1, p_1)) & \\ Move((\vec{A}, z, \theta, p), (\vec{A}_1, z_1, p_1)) & \rightarrow & Push(\vec{A}_1, z_1, m, p_1) & \\ Move((\vec{A}, z, \theta, p), (\vec{A}_1, z_1, p_1)) & \rightarrow & Check(\vec{A}, z, min(\theta, c), c, p_2) & \text{if } c \leq \Omega(p) \text{ and } p_2 \in A_c \end{array}$$

Except in the case of *Move* states, the owner of each control state is determined by the component $p$. That is,

$$\begin{array}{rcl} \mathcal{Q}_E & = & \{\ Move(\vec{A}, z, \theta, p_1), (?, z_1, p_2)), Push(\vec{A}, z, \theta, p), Check(\vec{A}, z, \theta, c, p) \mid p \in \mathcal{P}_E\ \} \\ \mathcal{Q}_A & = & \{\ Move(\vec{A}, z, \theta, p_1), (\vec{A}_1, z_1, p_2)), Push(\vec{A}, z, \theta, p), Check(\vec{A}, z, \theta, c, p) \mid p \in \mathcal{P}_A\ \} \end{array}$$

We assign priorities to states of $\mathcal{M}$ as follows: the priority of $Check(\vec{A}, z, \theta, c, p)$ is $c$, the priority of $Push(\vec{A}, z, \theta, p)$ is $\Omega(p)$ and all other states have priority $m + 1$.

Éloïse wins the game if Abelard cannot make a move, play reaches $Pop(p)$ for some $p$ or an infinite path is generated such that the lowest priority occurring infinitely often is even. Conversely, Abelard wins if Éloïse cannot make a move, play reaches $Err(p)$ for some $p$ or an infinite path with an odd smallest infinitely occuring priority is generated. The initial state of the game is $Check((\emptyset, \ldots, \emptyset), \bot, m, m, p_0)$ where $p_0$ is the initial control state of $\mathcal{G}$.

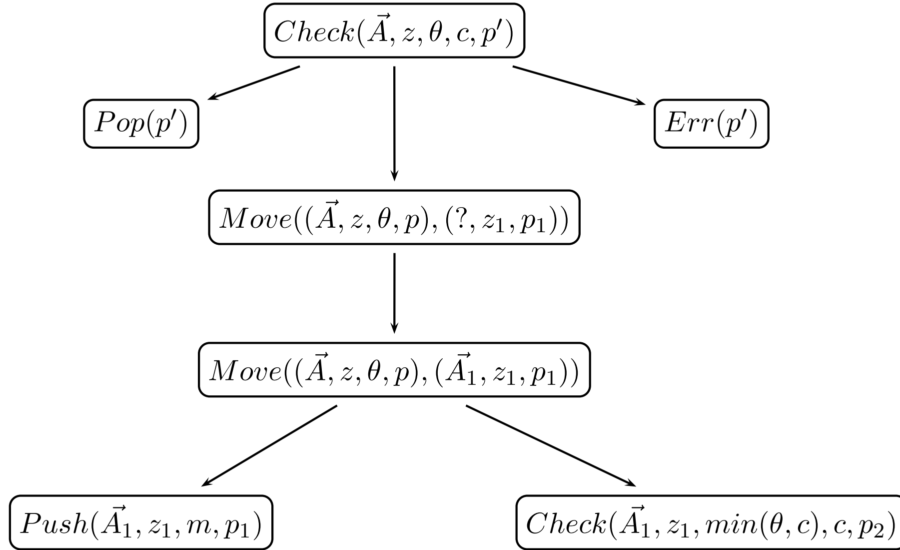



Figure 2.7: Reducing a pushdown parity game to a finite state game.

In the above definition, *Check* moves correspond most directly to the states in $\mathcal{G}$. After a push move, play moves to a *Move* state belonging to Éloïse. From this state Éloïse replaces the ? symbol with her prediction for the future play. Abelard can accept this prediction and choose where the top symbol is popped by moving to a *Check* state, or challenge the prediction by moving to a *Push* state. The moves are illustrated in Figure 2.7.

**Theorem 2.6.1** ([53]). *Éloïse has a winning strategy in the game $\mathcal{G}$ iff she has a winning strategy in the game $\mathcal{M}$ from the node $Check((\emptyset, \ldots, \emptyset), \bot, m, m, p_0)$.*

The size of $\mathcal{M}$ is exponential in the size of $\mathcal{G}$, since each state contains $m$ subsets of the control states of $\mathcal{G}$. Combined with the complexity of calculating the winner of a finite state parity game, the algorithm runs in EXPTIME. By simulating alternating linear space bounded Turing machines, Walukeiwicz shows that this complexity is optimal [53].

### 2.6.4 Properties of High Borel Complexity

Following the model-checking results discussed above, a natural next step is to consider games with more expressive winning conditions. We describe a number of these extensions.

#### A $\Sigma_3$-Complete Winning Condition

The first result of this kind over pushdown systems was published in 2002 by Cachat, Duparc and Thomas [130]. They present an algorithm that computes the winning regions of a pushdown game with a $\Sigma_3$-complete winning condition. This winning condition requires a configuration to be seen infinitely often. That is, for a play $\rho$,

> "there is a vertex $v$ such that for all time instances $t$ there is $t' > t$ such that $v$ is visited at $t'$ in the play $\rho$ under consideration."

Observe that this winning condition is vacuous for infinite plays over finite state games. To see that this winning condition is $\Sigma_3$, we state the winning condition more formally. Let $\rho = c_0, c_1, \ldots,$

$$\exists p \in \mathcal{P} \exists w \in C_1^{\Sigma} \forall n \exists m > n : c_m = \langle p, [w] \rangle$$

The condition $c_m = \langle p, [w] \rangle$ is prefix-recognisable, and hence $\Sigma_1 \cap \Pi_1$. The quantification alternates exists, for-all, exists. $\exists m : c_m = \langle p, [w] \rangle$ is $\Sigma_1$, $\forall n \exists m \ldots$ is $\Pi_2$ and finally, the full condition — being the infinite union of $\Pi_2$ conditions — is $\Sigma_3$. Cachat *et al.* prove that this winning condition is $\Sigma_3$-complete, and hence a representation lower in the Borel hierarchy cannot be found.

The solution is an extension of Cachat's algorithm for computing the winning regions of a Büchi game played over pushdown systems [127]. The $\Sigma_3$ winning condition can be restated: there exists some $h \geq 0$ such that a stack of length less than $h$ is seen infinitely often. For any given $h$ we can define a Büchi game where $\mathcal{F}$ is the set of all stacks with height less than $h$. To complete the algorithm Cachat *et al.* give a limit $N$ on the maximum $h$ for which the property may be satisfied. This limit is a function of the size of $\Sigma$, $\mathcal{Q}$ and the longest word $w$ appearing in some $(p, a, push_w, p') \in \Delta$.

### Combinations of Unboundedness and Büchi

Cachat *et al.* note that, although $\Sigma_3$-complete, the winning condition above needs to be combined with a Büchi condition to give a true-$\Sigma_3$ condition. That is, a condition in $\Sigma_3$ but not $\Pi_3$. This observation motivated Bouquet, Serre and Walukiewicz to provide a solution to games with binary combinations of a **stack unboundedness** condition and a Büchi condition [10]. Strict stack unboundedness is the complementation of the winning condition of Cachat *et al.*. Unboundedness is a relaxation of this condition, but the two are equivalent when combined with a Büchi condition. The solution is an extension of Walukiewicz' parity games algorithm, and is consequently only suitable for local model-checking.

To ease notation, Bouquet *et al.* present different EXPTIME solutions for each of the boolean combinations of unboundedness and Büchi. The reduction for the union of unboundedness and Büchi is illustrated in Figure 2.8. This game is equipped with a Büchi property (in this case, a 0 priority is seen infinitely often). For clarity, the priorities are given on the transitions and $A_0, A_1$ take the place of $\vec{A}$ in Walukiewicz' construction. Additionally, we do not keep track of the minimum priority encountered. If a 0 priority is seen, we replace $A_1$ with $A_0$ which forces play to return to $A_0$ when a pop occurs.

To win the game in Figure 2.8 Éloïse can either satisfy the unboundedness property or the Büchi property. The Büchi property is checked by the transitions from $Move$ to $Check$. Unboundedness is checked by the transition from $Move$ to $Push$. If this transition is taken an infinite number of times, then the stack will grow infinitely large. Furthermore, since the priority of the transition is 0, such a play will satisfy the Büchi condition.

### Exploration Properties

In 2004, Gimbert investigated another condition called **exploration** [52], which requires an infinite number of different states to be visited during a play. In the case of pushdown systems, this condition is equivalent to stack unboundedness, although Gimbert considers infinite state games in general. In particular, he proves the existence of positional winning strategies for infinite state games whose winning conditions are the intersection of the exploration condition

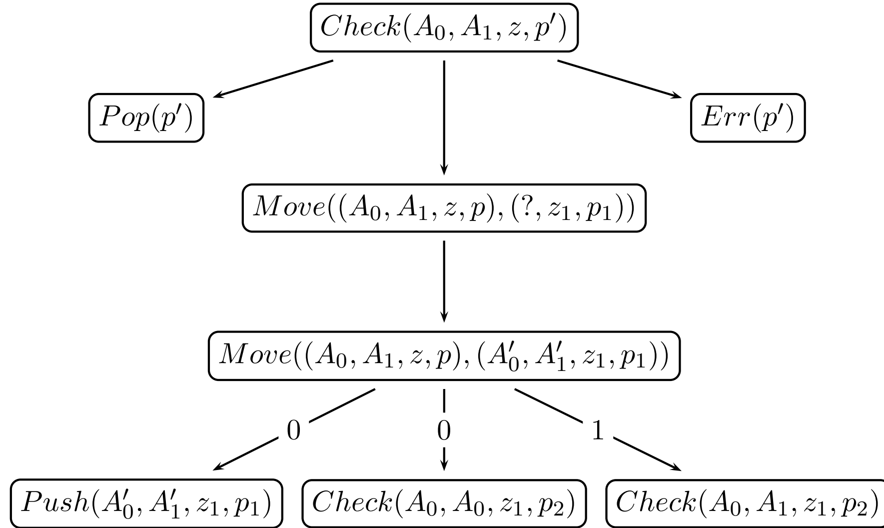



Figure 2.8: Reducing a pushdown game with an unboundedness winning condition to a finite state game.

and a parity condition, and games whose winning conditions are the union of the exploration condition and a modified parity condition allowing infinite priorities. Furthermore, it is shown how to compute the winning regions of such games played over pushdown systems in exponential time, although parity games with only a finite number of priorities are considered.

Gimbert's approach is inspired by the global model-checking techniques of Piterman and Vardi [86] discussed in Section 4.6.3. Strategies are represented as trees over the alphabet $\Sigma$. These trees are labelled by sets of pushdown commands. For example, suppose the node $w$ is labelled with $\{(p, a, push_{ab}, p'), (p', a, push_{abc}, p'')\}$. From a configuration $\langle p, [w]\rangle$ the move corresponding to the given strategy is $(p, a, push_{ab}, p')$. Similarly, $(p', a, push_{abc}, p'')$ is played from the configuration $\langle p', [w]\rangle$.

Winning regions are recognised by alternating Büchi tree automata. These automata recognise winning strategies (and thus, winning regions) and are built from the combination of a number of simpler automata that check certain properties pertaining to winning strategies. For example, the following property checks a requirement for a tree to represent a valid strategy:

If $t(aw)$ contains a *pop*-transition to a state $p'$ then $t(w)$ contains a $p'$-transition.

That is, if, from a configuration $\langle p, [aw]\rangle$, the strategy moves play to $\langle p', [w]\rangle$, a move from $\langle p', [w]\rangle$ must also be defined.

An alternating Büchi tree automaton that checks the complement of this property can be defined. The automaton uses non-determinism to guess a node $w$, a letter $a \in \Sigma$ and a move in $t(aw)$ which violates the property. Then we complement the automaton to obtain an automaton checking the required property.

### Properties of Arbitrary Borel Complexity

Finally, in 2004 Serre introduced a family of winning conditions of arbitrary Borel complexity [93]. These winning conditions are of the form $\Omega_{A_1 \triangleright \ldots \triangleright A_n \triangleright A_{n+1}}$ where $A_1, \ldots, A_{n+1}$ are

deterministic pushdown automata. An internal winning condition requires the stack produced during a play to be unbounded. The unbounded stack is an infinite word which is passed as input to the PDA $A_1$. It is required that the run of $A_1$ produces a second unbounded stack which is passed to $A_2$ and so on. Finally, the PDA $A_{n+1}$ receives an unbounded stack from $A_n$ as input. The PDA $A_{n+1}$ is equipped with a parity condition and Éloïse wins the pushdown game if this parity condition is satisfied. If the transitions of the pushdown game are labelled with characters from an alphabet $\Gamma$, an external winning condition uses the infinite word generated from $\Gamma$ as the input to $A_1$. External winning conditions are $\mathcal{B}(\Sigma_{n+2})$-complete and internal conditions are $\mathcal{B}(\Sigma_{n+3})$-complete.

The local model-checking algorithm for pushdown games equipped with these winning conditions is a further extension of the finite state game reduction of Walukiewicz. Input to the automaton $A_1$ consists of a sequence of stack symbols which, once pushed, are never popped. Thus, we can reduce the pushdown game to a finite game and form the product with $A_1$. The $A_1$ component of the product state-space is advanced when the finite state game moves from $Move$ to $Push$. This will only occur infinitely often if the stack in the original game is unbounded.

The product game will be a pushdown game with a winning condition defined by the automata $A_2, \ldots, A_n, A_{n+1}$. We can reduce the game to a finite state game and form another product with $A_2$. This procedure repeats until we have a finite state game with a parity winning condition. This solution is $(n+2)$-EXPTIME, which is shown to be optimal.

## 2.7 Order-1 Extensions of Pushdown Systems/Automata

In this section we describe several extensions of pushdown systems and automata. These extensions allow a greater range of model-checking applications. We begin with a discussion of regular stack properties. In the preceding sections, properties of a configuration were generally dependent on the control state and the $top_1$ stack symbol. By allowing these properties to depend on the entire contents of the stack, we find applications in security, data-flow analysis and CTL* model-checking.

We then describe recursive state machines, which are equivalent to pushdown automata. However, their structure suggests several natural restrictions, including modular program strategies. That is, when synthesising a program, the behaviour of a procedure should not depend on its calling context. These restrictions permit more efficient model-checking solutions.

Weighted pushdown systems give a cost to each pushdown command. This allows a finer analysis of pushdown problems. We discuss a security example, and show, that by using weights, we can differentiate two certification options, which reveal varying amounts of information.

Finally, we give an account of visibly pushdown automata. When pushdown automata are used to model programs with recursion, it is often the case that a specific inputs correspond to a procedure calls (and returns). Visibly pushdown systems formalise this observation, allowing the use of new, expressive logics.

### 2.7.1 Regular Stack Properties

When model-checking pushdown systems, each configuration is associated with the set of atomic propositions that it satisfies. That is, we have a mapping $\Lambda : C_1^{\Sigma} \rightarrow AP$, where $AP$

is a finite set of atomic propositions. In general, arbitrary mappings of configurations to propositions lead to undecidability.

We can restrict $\Lambda$ by insisting that configurations with matching control states and top stack symbols satisfy the same propositions. That is, $\Lambda : \mathcal{P} \times \Sigma \rightarrow AP$. This is the restriction used in early research into pushdown systems (For example, [3]).

Esparza, Kučera and Schwoon generalise this restriction [64]. They introduce regular valuations that assign atomic propositions based on the current control state and the complete contents of the stack. To each proposition and control state we attach a finite word automaton. If the current stack is accepted by this automaton then the atomic proposition is true at the current configuration.

Esparza *et al.* describe three applications of these extended valuations.

- **Inter-procedural data-flow analysis**. This application follows a similar encoding to that described in Section 2.6.2. The increased expressiveness of regular valuations allows the specification of properties that rely on the stack contents. For example, let $top_n$ hold when the current control point (stored on the top of the stack) is $n$, and $used_Y$ and $def_Y$ hold if the variable $Y$ is used or defined respectively at the current control point,

  $$G(top_n \Rightarrow ((\neg used_Y U def_Y) \ \vee \ (G \neg used_Y)))$$

  asserts that, for control point $n$, the variable $Y$ is not used until it is redefined. That is, $Y$ is dead at control point $n$.

  In languages such as LISP which use dynamic scoping, the scope of $Y$, and hence the values of $used_Y$ and $def_Y$, can only be determined with reference to the stack of procedure calls.

- **Pushdown Systems with Checkpoints** associate regular automata with pairs in $\mathcal{P} \times \Sigma$. Pushdown commands are labelled positive, negative or independent. A positive rule may only be applied if the current configuration has a stack accepted by the automata associated with its control state and top stack symbol. A negative rule requires the check to fail and an independent rule performs no checks.

  These systems are a generalisation of a formalism introduced by Jensen, Le Métayer and Thorn for analysing security properties [134]. One such security property, implemented in Java and .Net [74, 34], allows the programmer to decorate code with permission checks. These checks require that all callers on the stack have sufficient privileges to proceed. For example, a file access command needs to be able to access the file system. When called from a trusted context, the command should be able to proceed. However, if an untrusted caller attempts to run the command, it should be blocked. Recent research addressing the limitations of this approach to security still maintain stack-based properties. For example, Pistoia, Banerjee and Naumann propose **Information-Based Access Control**, which combines stack inspection with data-flow analysis to ensure that unauthorised code which may no longer appear on the stack does not influence the execution of secure code [80].

- *Model-checking CTL$^*$ properties.* CTL$^*$ extends LTL with a path quantifier $E\phi$ which requires that some path from the current state satisfies $\phi$. We can use the algorithm for model-checking LTL with regular valuations recursively to model-check CTL$^*$ [43]. In the base case there are no path quantifiers. Thus, $\phi$ is an LTL formula and we compute

the set of configurations satisfying $\neg\phi$ — thus, the complement is the set of configurations from which a path satisfying $\phi$ exists. Hence, we have an automaton accepting all configurations at which $E\phi$ holds. This automaton can be used to represent a new proposition that holds when $E\phi$ does. By repeating this process, we can recursively eliminate the path quantifiers from the formula and calculate the set of configurations satisfying the original CTL* formula.

Esparza *et al.* present two methods for reducing pushdown systems with regular valuations to pushdown systems with simple valuations. Both techniques give an EXPTIME algorithm (that is optimal), but only one is complete. However, it is not clear which will perform better in practice.

Both reductions assume a *deterministic* automaton for each control state/atomic proposition pair. This means that a product may be formed between the pushdown system and the valuation automata. Each time a character is pushed onto the stack, the valuation automata are advanced appropriately. We can check the truth of a proposition by observing whether an accepting state has been reached in the corresponding valuation automaton.

The two methods of encoding this product differ in where the valuation automaton states are stored. This affects the handling of pop transitions. The complete technique alters the alphabet to $\Sigma \times \mathcal{Q}_1 \times \ldots \times \mathcal{Q}_m$ where $\mathcal{Q}_1, \ldots, \mathcal{Q}_m$ are the state-sets of the valuation automata. Thus, the progress of these automata is stored on the stack. Before the pop occurs the stack is of the form $(a, q_1, \ldots, q_m)(b, q'_1, \ldots, q'_m)w$. Let $\pi(w)$ be the projection of $w$ to $\Sigma$. The states $q_1, \ldots, q_m$ are the states reached by the valuation automata after reading the word $ab\pi(w)$ in reverse (corresponding to the order the symbols were added to the stack). Similarly, the states $q'_1, \ldots, q'_m$ are the states reached after reading the word $b\pi(w)$ in reverse. Hence, when a pop command occurs, the progress is automatically reversed.

The second technique changes the set of control states to $\mathcal{P} \times \mathcal{Q}_1 \times \ldots \times \mathcal{Q}_m$. This means that the progress information is stored in the control state and reversing the information when a pop occurs must be done explicitly. To do this, we require the valuation automata to be *backwards deterministic*. Informally, this restriction requires the automata to remain deterministic when the transitions are reversed. Using this encoding, we can handle a pop command simply by following the automata backwards. Backwards determinism is a genuine restriction on the expressiveness of the automata, and the method is therefore incomplete.

### 2.7.2 Recursive State Machines

Recursive State Machines (RSMs) are equivalent to pushdown systems. They were introduced independently by Alur, Etessami and Madhusudan [96] and Benedikt, Godefroid and Reps [75]. Both of these works generalise a restricted form of RSMs introduced by Alur and Yannakakis which disallow recursion [99] and are based on highly visual hierarchical program descriptions that have been popular for some time (e.g. StateCharts [41]). An updated study of RSMs was published in 2005 [98].

We will forgo a formal description of RSMs in favour of their intuitive diagrammatic representation. Figure 2.9 shows an example RSM taken from Alur *et al.* [98]. We have omitted some of the labelling from the original diagrams since it is relevant only to the formal definition.

An RSM contains a number of modules (A1, A2 and A3 in Figure 2.9). Each of these modules contains entry, internal and exit nodes and transitions. In this respect each module is

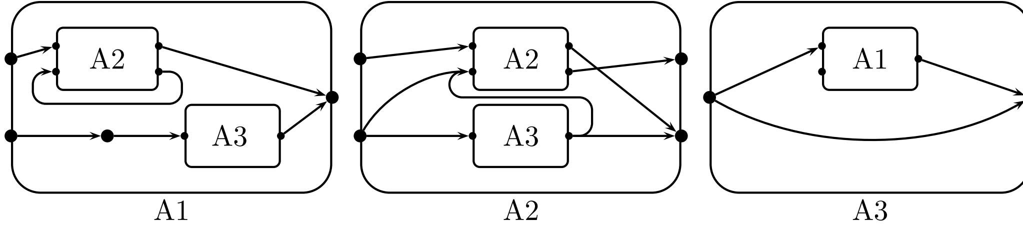


Figure 2.9: A recursive state machine [98] (simplified)

similar to a finite state Kripke structure. However, in addition to transitions between nodes, we have transitions to the entrances and from the exits of other modules (for example, in Figure 2.9 the entry node of A3 has a transition to the uppermost entry node of A1). Thus, a run of the machine can switch between modules in a call/return style

If we disallow mutual recursion (for example, A2 contains a call to itself), RSMs are no more expressive than finite word automata. As generators of regular languages they are exponentially more succinct [103].

In general, RSMs are equivalent to pushdown systems. However, we can identify subclasses of RSMs by the maximum number of module entry or exits nodes. Improved complexity results can be obtained for these subclasses. For an arbitrary number of entrances and exits, the complexity results for various model-checking problems match the results for pushdown systems.

**Theorem 2.7.1** ([98])**.** *Every PDS is bisimilar to an RSM, and vice versa. Moreover, every context-free system is bisimilar to a single-exit RSM, and vice versa.*

In fact, there are linear-time, log-space reductions between the LTL, CTL and CTL$^*$ model-checking problems over PDSs and RSMs.

**Theorem 2.7.2** ([98])**.**

- *The LTL model-checking problem for RSMs and for PDSs are inter-reducible in linear-time and logarithmic-space, and similarly for CTL and CTL$^*$.*
- *The LTL model-checking problem for single-exit RSMs and for context-free systems are inter-reducible in linear-time and logarithmic-space, and similarly for CTL and CTL$^*$.*

Theorem 2.7.2 gives us immediate complexity bounds for a variety of model-checking problems. Alur *et al.* show that, in some cases, these complexity bounds can be improved. In particular, they give a linear-time algorithm for CTL$^*$ model-checking over single-exit RSMs, which improves the quadratic bound implied by the connection to context-free processes. Additionally, when single-entry RSMs are considered, LTL model-checking can be performed in linear-time, rather than the cubic complexity of PDSs. These complexities are summarised in Table 2.3. Complexity results improved by Alur *et al.* are italicised.

#### Modular Strategies

In 2004, Etessami provided algorithms for determining winning strategies for RSM reachability and Büchi games [71]. These algorithms generalise the algorithms of Alur *et al.* [96]

| Class of RSM | Reach. | Cycle Detection | LTL | CTL | CTL* |
|---|---|---|---|---|---|
| Single-exit | Linear | Linear | Linear | Linear | *Linear* |
| Single-entry Multiple-exit | *Linear* | *Linear* | *Linear* | EXPTIME | EXPTIME |
| Multiple-entry Multiple-exit | Cubic | Cubic | Cubic | EXPTIME | EXPTIME |

Table 2.3: Improved complexity bounds for RSM model-checking [98]

and provide an alternative approach to pushdown/RSM model-checking that may prove beneficial in practice. A different type of game over RSMs is considered by Alur, La Torre and Madhusudan in their 2003 work into modular strategies [105, 104].

Modular strategies have only a local memory. That is, the strategy is independent of the play occurring before the current module was entered. These strategies have a natural interpretation in terms of the synthesis problem where the implementation of a module should not depend on the context of its use.

Alur *et al.* present the following results concerning the calculation of winning strategies:

**Theorem 2.7.3** ([105]). *Determining whether Éloïse has a modular winning strategy in an RSM reachability or safety game is NP-complete.*

**Theorem 2.7.4** ([104]). *Determining whether Éloïse has a modular winning strategy in an RSM game with a deterministic/universal Büchi or co-Büchi automaton specification is EXPTIME-complete.*

The NP-completeness of reachability and safety games contrasts with the EXPTIME-completeness of reachability games with full strategies. In fact, a strategy with full memory may exist while a modular strategy does not. Also observe that reachability and safety are not dual winning conditions when restricted to modular strategies.

### 2.7.3 Weighted Pushdown Systems

Weighted Pushdown Systems (WPDS) extend pushdown automata by adding weights to the transitions. They were introduced by Schwoon, Jha, Reps and Stubblebine [124]. The weights have applications in security [124] and dataflow analysis [135]. We will begin with a formal, abstract definition of WPDA. We will then discuss their application to the analysis of security certificates.

WPDA form the basis of a recent tool by Lal and Reps [18]. This tool uses a fast graph algorithm and a connection to abstract grammars to obtain run-times comparable to Moped. Lal, Reps and Balakrishnan also consider **Extended WPDSs** [19]. These systems associate merging functions as well as weights to $push_w$ transitions. For dataflow analysis, where the stack is used to track procedure calls, these merging functions allow the calling context to be considered when calculating the effects of a procedure.

#### Definition

A weighted pushdown system is a pushdown system whose transitions are augmented with weights from a set $W$. These weights must be well behaved. More precisely, they should form part of a *bounded idempotent semi-ring*. This structure allows weights to be combined to produce cumulative weights over a path.

**Definition 2.7.1.** A **weighted pushdown system** is a triple $(A, \mathcal{S}, f)$ where $A = (\mathcal{P}, \mathcal{D}, \Sigma)$ is a pushdown system, $\mathcal{S} = (W, \oplus, \otimes, 0, 1)$ is a bounded idempotent semi-ring, and $f : \mathcal{D} \rightarrow W$ is a function assigning weights to the transitions in $\mathcal{D}$.

**Definition 2.7.2.** A **bounded idempotent semi-ring** is a tuple $(W, \oplus, \otimes, 0, 1)$ where $W$ is a set with $0, 1 \in W$ and $\oplus$ and $\otimes$ are combine and extend operations respectively. These are binary operators such that,

- $(W, \oplus)$ is a commutative monoid with 0 as its neutral element and $\oplus$ is idempotent $(a \oplus a = a)$.
- $(W, \otimes)$ is a monoid with neutral element 1.
- $\otimes$ distributes over $\oplus$:

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \qquad \text{and} \qquad (a \oplus b) \otimes c = (a \otimes c) \oplus (a \otimes b)$$

- 0 is an annihilator with respect to $\otimes$:

$$a \otimes 0 = 0 = 0 \otimes a$$

- In the partial order $a \sqsubseteq b$ iff $a \oplus b = a$, there are no infinite descending chains.

We can extend $f$ to paths. For a sequence of pushdown commands $d_1, \ldots, d_m$;

$$f(d_1, \ldots, d_m) = f(d_1) \otimes \cdots \otimes f(d_m)$$

**Security Example**

We illustrate WPDSs using an example from security. This example motivated the introduction of WPDSs [124] and is based on the application of PDSs to a related problem [121].

The **SPKI/SDSI framework** enforces security checking using certifications,

- A name certificate is of the form $K\ A \longrightarrow S$. The symbol $K$ indicates that $A$ is a name. We consider $K$ to be the issuer of the certificate. The rule specifies that $K\ A$ can be rewritten to the value $S$, which is a sequence of identifiers, providing meaning to the certificate.
- An authorisation certificate takes the form $K_T \square \longrightarrow S \square$ or $K_T \square \longrightarrow S \blacksquare$ where $T$ specifies the permission being granted, and $\square$ and $\blacksquare$ indicate that the permission can or can not be delegated.

Rewrite rules are assigned weights in a similar manner to the transitions of WPDSs.

Suppose an insurance company $X$ offers favourable rates to patients of a particular hospital $H$. We represent this privilege $K_X \square$. The rule,

$$K_X \square \longrightarrow K_H\ \mathit{patient}\ \blacksquare \tag{2.1}$$

specifies that to be eligible for favourable rates, one must be a patient of hospital $H$, and no delegation can take place.

If a person, Alice, were to apply for cheaper insurance, she would have to prove she is a patient at $H$. She can do this by providing a certificate from one of the clinics treating her. Let $H_{AIDS}$ denote the hospital's AIDS clinic, and $H_{IM}$ denote the internal-medicine centre. If Alice were a patient at both of these clinics she would have the following certification options:

$$K_H \ patient \quad \longrightarrow \quad K_{H_{AIDS}} \ patient \tag{2.2}$$
$$K_H \ patient \quad \longrightarrow \quad K_{H_{IM}} \ patient \tag{2.3}$$

where each clinic will prove that she is a patient there:

$$K_{H_{AIDS}} \quad \longrightarrow \quad K_{Alice} \tag{2.4}$$
$$K_{H_{IM}} \quad \longrightarrow \quad K_{Alice} \tag{2.5}$$

Hence, Alice has two possible derivations $K_X \ \square \ \longrightarrow \ K_{Alice} \ \blacksquare$. However, these two derivations are not equal: Alice must reveal to the company that she is an AIDS patient or a patient at the internal medicine clinic. Since an AIDS patient is likely to receive less favourable rates than an internal medicine patient, she would obviously prefer the second choice. We can reflect this preference using the weights $I$ and $S$, denoting insensitive and sensitive information. These weights form a bounded idempotent semi-ring as follows. Let $0 = S$ and $1 = I$ and,

$$\begin{array}{ccc} I \oplus x = x \oplus I = I & \text{and} & S \oplus x = x \oplus S = S \\ S \otimes x = x \otimes S = S & \text{and} & I \otimes x = x \otimes I = x \end{array}$$

We allocate the weight $I$ to all rules except rule 2.4, which has the weight $S$. Thus, the chain (2.1) $\otimes$ (2.2) $\otimes$ (2.4) has weight $S$, whilst (2.1) $\otimes$ (2.3) $\otimes$ (2.5) has weight $I$.

**Encoding Weighted SPKI/SDSI Using Weighted Pushdown Systems**

The SPKI/SDSI framework has a rather straightforward interpretation using PDSs [121]. Similarly, when the framework is enhanced with weights, WPDSs provide a natural interpretation. We briefly describe this encoding.

Intuitively, the issuers $K_C$ are stored in the control state, whilst the list of identifiers, such as *patient* in the above example, are stored in the stack. Let $\mathcal{K}$ be the set of keys of the form $K_C$ and $\mathcal{I}$ be the set of identifiers appearing in some $C$. With an SPKI/SDSI system, we associate the pushdown system $(\mathcal{K}, \mathcal{D}, \mathcal{I} \cup \{\square, \blacksquare\})$, where,

- For every name certificate $K \ A \ \longrightarrow \ K' \ \sigma$, where $\sigma$ is a sequence of identifiers, $\mathcal{D}$ contains the rule $(K, A, push_\sigma, K')$. The weight of the command is the weight associated with the certificate.

- For every authorisation certificate $K \ \square \ \longrightarrow \ K' \ \sigma \ b$ with $b = \square$ or $b = \blacksquare$, $\mathcal{D}$ contains the command $(K, \square, push_{\sigma b}, K')$. The weight associated with the command is the weight associated with the certificate.

Hence, in the above example, Alice's preferred proof of eligibility corresponds to the run,

$$\langle K_H, \square \rangle \hookrightarrow \langle K_{H_{IM}}, patient \ \blacksquare \rangle \hookrightarrow \langle K_{Alice}, \blacksquare \rangle$$

This run can be obtained by computing the weight-minimising reachability problem from $\langle K_H, \square \rangle$ to $\langle K_{Alice}, \blacksquare \rangle$. Schwoon *et al.* provide a solution to this problem via a connection with the meet over all paths problem for abstract grammars [124].

### 2.7.4 Visibly Pushdown Games

Visibly Pushdown Automata (VPA) are a subclass of pushdown automata introduced by Alur and Madhusudan [100]. VPA provide an automata-theoretic generalisation of CaRet — a logic of calls and returns discussed below — and other formalisms which analyse the stack. The input alphabet is divided into calls, returns and internal characters. When reading a call, a VPA pushes a single character onto the stack. Analogously, a pop action is performed when reading a return. When the VPA reads an internal character, only the control state is changed. Hence, the stack actions are visible from the input alphabet.

**Definition 2.7.3.** A **visibly pushdown automaton** over an alphabet $\Gamma = \Gamma_{calls} \uplus \Gamma_{int} \uplus \Gamma_{ret}$ is a tuple $(\mathcal{P}, p_0, \mathcal{D}, \Sigma, \Omega)$ where $\mathcal{P}$ is a set of control states, $p_0$ is the initial control state,

$$\mathcal{D} \subseteq (\mathcal{P} \times \Gamma_{calls} \times \Sigma \setminus \{\bot\} \times \mathcal{P}) \cup (\mathcal{P} \times \Gamma_{int} \times \mathcal{P}) \cup (\mathcal{P} \times \Gamma_{ret} \times \Sigma \times \mathcal{P})$$

is a set of commands, $\Sigma$ is a finite stack alphabet, and $\Omega$ is an acceptance condition.

A transition $\langle p, [w] \rangle \overset{\alpha}{\hookrightarrow} \langle p', [w'] \rangle$ exists iff,

- When $\alpha \in \Gamma_{calls}$, we have $(p, \alpha, a, p') \in \mathcal{D}$ and $w' = aw$.
- When $\alpha \in \Gamma_{int}$, we have $(p, \alpha, p') \in \mathcal{D}$ and $w = w'$.
- When $\alpha \in \Gamma_{ret}$, we have $(p, \alpha, a, p') \in \mathcal{D}$ and $w = aw'$.

Visibly pushdown automata are defined as acceptors of both finite and infinite words when equipped with reachability and Büchi acceptance conditions respectively. The languages definable are closed under the boolean operations and the Kleene star. In the case of finite words, VPAs are determinisable. This does not hold over infinite words, although it is still possible to complement an automaton. VPAs also permit an MSO characterisation — through the addition of an MSO *matching* operator which holds for $x$ and $y$ when $x$ is a call and $y$ the corresponding return — and a characterisation in terms of regular trees. Furthermore, Alur and Madhusudan define visibly pushdown grammars, providing another exogenous characterisation.

#### Defining $\mathcal{B}(\Sigma_3)$ Languages

Pushdown automata can be defined as a game. During a play, the labels on the transitions form a word. This word can be used as input to another pushdown automaton that defines the winning condition. However, pushdown games with a pushdown winning condition are undecidable.

The decidability of visibly pushdown games equipped with visibly pushdown winning conditions was shown by Löding, Madhusudan and Serre [30]. The first step in the solution is to internalise the winning condition. When the winning condition is given by a deterministic automaton, we can simply form the product of the game automaton and the winning condition, forming a standard pushdown game. However, over infinite plays, VPAs cannot be determinised. Hence, Löding *et al.* introduce **stair automata**. These automata are equipped with a parity condition that is checked only when a character is pushed onto the stack and

will not be removed for the remainder of the run. More formally, we define the stack height for any $\alpha_0 \dots \alpha_m \in \Gamma^*$, where $sh(\varepsilon) = 0$,

$$sh(\gamma_0 \dots \gamma_m) = \begin{array}{ll} sh(\gamma_1 \dots \gamma_m) + 1 & \text{if } \gamma_0 \in \Gamma_{calls} \\ sh(\gamma_1 \dots \gamma_m) & \text{if } \gamma_0 \in \Gamma_{int} \\ max\{sh(\gamma_1 \dots \gamma_m) - 1, 0\} & \text{if } \gamma_0 \in \Gamma_{ret} \end{array}$$

For any word $w = \alpha_0\alpha_1 \dots \in \Gamma^\omega$, $Steps_w = \{ n \in \mathcal{N} \mid \forall m \geq n.sh(\alpha_0 \dots \alpha_m) \geq sh(\alpha_0 \dots \alpha_n) \}$. Let $n_0 < n_1 < \dots$ be an ascending enumeration of $Steps_w$. A stair automaton accepts $w$ if $\alpha_{n_0}\alpha_{n_1} \dots$ satisfies the parity condition.

An important property of stair automata is that, for every nondeterministic Büchi VPA, we can construct an equivalent deterministic stair automaton.

**Theorem 2.7.5** ([30])**.** *For each non-deterministic Büchi VPA $\mathcal{M}$ over $\Gamma$ there exists a deterministic parity stair automaton $D$ such that $\mathcal{L}(\mathcal{M}) = \mathcal{L}(D)$. Moreover, we can construct $D$ such that it has $2^{\mathcal{O}(|\mathcal{Q}|^2)}$ states, where $\mathcal{Q}$ is the state-space of $\mathcal{M}$.*

In fact, it can be shown that stair automata are equivalent to Büchi VPA.

Once we have constructed a deterministic stair automaton, we can form the product of the game PDA and the stair automaton, resulting in a pushdown game with a parity winning condition. Determining the winner of a VPA game with a Büchi VPA winning condition is 2-EXPTIME-complete. This matches the complexity of a pushdown game whose winning condition is given by a non-deterministic Büchi PDA, since a deterministic Muller PDA must be constructed before the product can be built.

Finally, we note the topological complexity of VPAs.

**Theorem 2.7.6.** *The class of $\omega$-visibly pushdown languages is contained in $\mathcal{B}(\Sigma_3)$. Furthermore, there exist VPAs accepting true $\mathcal{B}(\Sigma_3)$ sets, $\Sigma_3$-complete sets and $\Pi_3$-complete sets.*

**Connection to Logics**

**CaRet**: Visibly pushdown languages were introduced as a generalisation of existing stack based verification techniques. One particular logic that motivated this research is CaRet, introduced by Alur, Etessami and Madhusudan in 2004 [97].

CaRet is defined over Recursive State machines. It extends the LTL syntax as follows,

$$\varphi = p \mid \varphi \vee \varphi \mid \neg\phi \mid \bigcirc^g \varphi \mid \varphi U^g \varphi \mid \bigcirc^a \varphi \mid \varphi U^a \varphi \mid \bigcirc^- \varphi \mid \varphi U^- \varphi$$

The $\bigcirc$ and $U$ operators are augmented with a superscript $g$, $a$ or $(-)$. Operators superscripted by a $g$ are global operators which behave as in standard LTL. The abstract operators — marked with an $a$ — behave in a local fashion. Intuitively, they abstract over the procedure calls, moving directly from a call to its corresponding return. In a pushdown system this would move from a configuration $\langle p, [w] \rangle$ to the next configuration of the form $\langle p', [w] \rangle$, provided $w$ is a prefix of all stacks in between. If the stack is popped to a prefix of $w$, the abstract successor is $\perp$ and the formula is not satisfied. The $(-)$ superscript operates backwards over the stack of calls.

The $a$ operators can be used to specify local properties and perform Hoare-style reasoning using pre- and post-conditions:

$$G^g \left( (call \wedge \varphi_{pre}) \Rightarrow \bigcirc^a \varphi_{post} \right)$$

The $(-)$ operators may be used to define stack inspection properties. We can also use the $\bigcirc^{-}$ operator with the $\bigcirc^{a}$ operator to assert a that a property $\varphi$ holds when the current procedure returns:

$$\bigcirc^{-} \bigcirc^{a} \varphi$$

Finally, although unboundedness is not expressible in CaRet, we are able to specify that the stack repeatedly returns to a bounded stack height:

$$F^g G^g (call \Rightarrow \bigcirc^{a} return)$$

**VP-$\mu$**: Work on visibly pushdown systems motivated the logic VP-$\mu$ [102]. This is an extension of $\mu$-calculus to represent procedure calls and is shown to be more expressive than CaRet and MSO.

VP-$\mu$ is interpreted over structured trees. These are trees whose edges are tagged as calls, returns or local transitions with the proviso that along any path, all return edges have a matching call. It is easy to see that the unfolding of a pushdown configuration graph has a natural conversion to structured trees: calls correspond to edges where the stack grows, the stack height does not change during local transitions, and pop transitions are labelled as returns. The syntax follows:

$$\begin{array}{rcl} \varphi & = & p \mid \neg p \mid X \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mu X.\varphi \mid \nu X.\varphi \\ & & \langle call \rangle \varphi \{\psi_1, \ldots, \psi_k\} \mid [call]\varphi\{\psi_1, \ldots, \psi_k\} \\ & & \langle loc \rangle \varphi \mid [loc]\varphi \mid \langle ret \rangle R_i \mid [ret] R_i \end{array}$$

The *loc* operators are local next operators and behave as expected. The call and return constructs behave as follows: $[call]\varphi\{\psi_1, \ldots, \psi_k\}$ asserts that $\varphi$ holds in the subsequent call and that after returning one of $\psi_1, \ldots, \psi_k$ hold. The formula required to hold is determined by the return operator. In the case of $[ret]R_i$ for $i \in \{1, \ldots, k\}$, it must be the case that $\psi_i$ holds.

Alur *et al.* provide an EXPTIME model-checking algorithm for VP-$\mu$ over recursive state machines. They claim VP-$\mu$ is the most expressive program logic that has a decidable model-checking problem. It is also shown that the satisfiability problem for VP-$\mu$ is undecidable, and that for closed formulae, the expressiveness of the hierarchy parameterised by $k$ — the maximum number of $\psi$ formulae in a sub-formula $[call]\varphi\{\psi_1, \ldots, \psi_k\}$ — is strict. In a separate, apparently unpublished paper, it is shown that VP-$\mu$ is as expressive as *alternating visibly pushdown tree automata* [101].

## 2.8 Higher-Order Pushdown Systems and Automata

Higher-order pushdown automata generalise pushdown automata through the use of higher-order stacks. Whereas a stack in the sense of a pushdown automaton is an order-one stack — that is, a stack of characters — an order-two stack is a stack of order-one stacks. Similarly, an order-three stack is a stack of order-two stacks, and so on. An order-$n$ PDA has push and pop commands for every $1 \leq l \leq n$. When $l > 1$ a pop command removes the topmost order-$l$ stack. Conversely, the push command duplicates the topmost order-$l$ stack.

Several notable advances in recent years have sparked off a resurgence of interest in higher-order PDA/PDSs in the verification community. For example, Knapik *et al.* [132] have

shown that the ranked trees generated by deterministic order-$n$ PDSs are exactly those that are generated by order-$n$ *recursion schemes* satisfying the *safety* constraint; Carayol and Wöhrle [13] have shown that the $\varepsilon$-closure of the configuration graphs of higher-order PDSs exactly constitute Caucal's graph hierarchy [36]. Remarkably these infinite trees and graphs have decidable monadic second-order theories [40, 13, 132].

Through their connections to higher-order recursions schemes, higher-order pushdown systems provide a natural infinite state model for higher-order programs with recursive function calls and are therefore useful in software verification. This connection is discussed in Section 2.8.6.

MSO decidability for trees generated by arbitrary (i.e. not necessarily safe) HORSs has been shown by Ong [29]. A variant kind of higher-order PDSs called *Collapsible Pushdown Systems* (extending *panic automata* [133] or *pushdown automata with links* [69] to all finite orders) has recently been shown to be equi-expressive with HORSs for generating ranked trees [76]. These automata allow *collapse* operations as well as the usual push and pop commands. When a character is pushed onto the top stack, a link is created to a lower stack of a specified order. This link always points to the same stack position even if it is copied to another part of the stack (during a higher-order push command). A collapse command reduces the stack to the position indicated by the link decorating the top character. In section 2.8.7 we discuss collapsible pushdown systems in more detail.

The main result of Chapter 5 is presented over *alternating* higher-order pushdown systems. This is because, although we apply our results to higher-order PDSs, the power of alternation is required to provide solutions to reachability games and alternation-free $\mu$-calculus model-checking over higher-order PDSs.

### 2.8.1 Improving the CD Player

Using higher-order PDSs, we are able to fix two problems with the order-1 CD player described in Section 2.6.2. The primary drawback of the order-1 CD player is that a playlist cannot be replayed. This is because the list can only be read by popping (and therefore losing) the CD at the head of the list. Using a $push_2$ command we are able to create a copy of the list. The copy is destroyed when playing the CDs. Once all CDs have been played, we simply discard the exhausted copy, leaving the store in the state it was in before play was pressed. The updated player is defined below. Figure 2.10 shows the execution of the order-2 PDS.

$$\begin{array}{rcll} \text{loadCD}(c) & = & (\text{i}, a, push_{ca}, \text{i}) & \text{for all } c \in \text{CDs and } a \in \Sigma \\ \text{delCD} & = & (\text{i}, c, pop_1, \text{i}) & \text{for all } c \in \text{CDs} \\ \text{playAll} & = & (\text{i}, a, push_2, \text{p}) & \text{for all } a \in \Sigma \\ \text{playCD}(c) & = & (\text{p}, c, pop_1, \text{p}) & \text{for all } c \in \text{CDs} \\ \text{done} & = & (\text{p}, \perp, pop_1, \text{i}) & \end{array}$$

Another problem with the order-1 example is that the playlist must be programmed in reverse. Using order-2 pushdown systems we can read the stack from bottom to top as well as vice versa. This allows the playlist to be read in order.

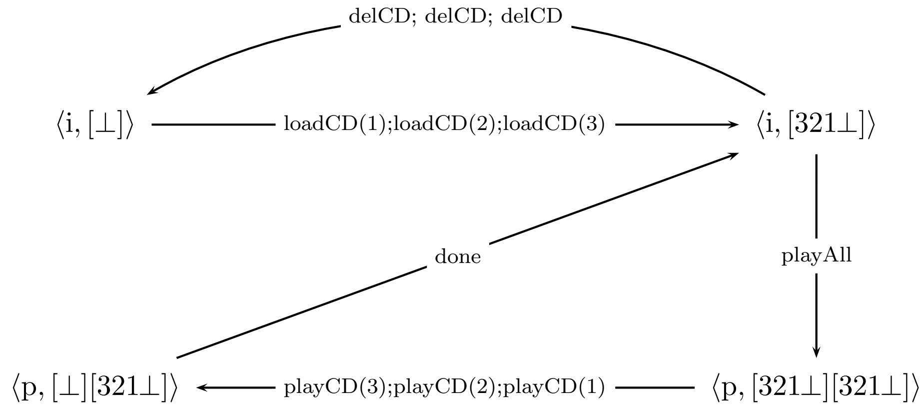



Figure 2.10: A CD player with a replayable playlist (excerpt).

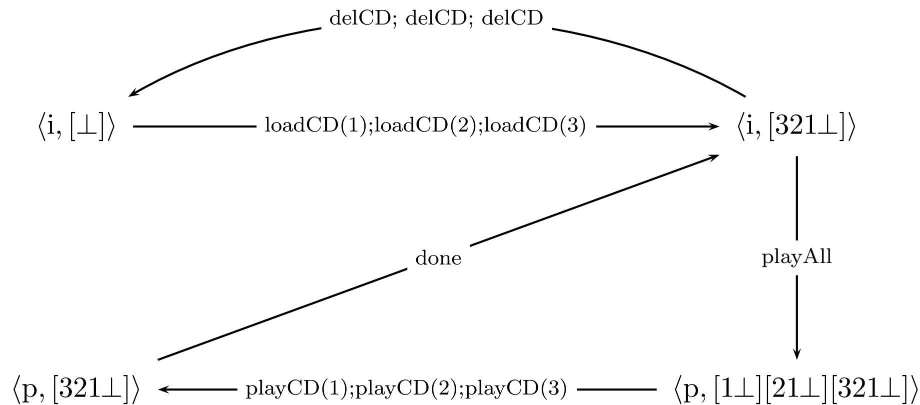



Figure 2.11: An improved CD player with a replayable playlist (excerpt).

To read the stack in reverse we create a stair structure by repeatedly applying $push_2$ and $pop_1$ commands and marking the top of the original stack:

$$[[abc\bot]] \longrightarrow \begin{bmatrix} [bc\bot] \\ [\overline{a}bc\bot] \end{bmatrix} \longrightarrow \begin{bmatrix} [c\bot] \\ [bc\bot] \\ [\overline{a}bc\bot] \end{bmatrix} \longrightarrow \begin{bmatrix} [\bot] \\ [c\bot] \\ [bc\bot] \\ [\overline{a}bc\bot] \end{bmatrix}$$

We are able to read the list in reverse by inspecting the $top_1$ character and then performing a $pop_2$ to access the next item. The procedure stops when the $top_1$ character is marked. Figure 2.11 shows the updated execution of the CD player.

### 2.8.2 Higher-Order Model-Checking

We observed above that, through their connection to recursion schemes, higher-order PDSs provide a model for higher-order programs. Such programs may be written in functional languages such as OCaml, F♯ or LISP, or in languages, such as C♯, which have higher-order

features. In this section we briefly discuss several applications of higher-order model-checking.

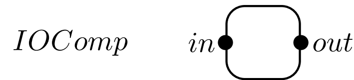
- An example of model-checking being applied to a safety-critical higher-order system is the use of SPIN to verify the Remote Agent deep-space flight software developed at NASA [72]. This software was written in LISP, and, after a lengthy conversion to SPIN's input language, a deadlock error was identified. This error manifested itself *in-flight* on a sibling system.

- Higher-order languages have applications in system design [57]. In particular, Edward Lee at Berkeley argues that the right way to approach concurrency is to separate concerns using (actor-oriented) **coordination languages**, and that higher-order languages are ideal for this purpose.

- Order-2 pushdown automata generate the **indexed languages** [24]. These languages have applications in natural language processing [49].

- Finally, higher-order verification is a current topic of interest at Microsoft (E.g. checking higher-order features of C♯ [94] or security policies specified in F♯ [70]).

We give an example of the use of higher-order languages in systems design. This example is adapted from Cataldo *et al.* [14]. We begin by assuming some input/output component *IOComp*,




Next we define *Seq* $x$ $y$ which connects components $x$ and $y$. For example, we may have,

Notice that the above system has an input and an output, and hence, can be composed using *Seq*. That is, $S4 = Seq\ S2\ S2$, which is a composition of four *IOComp* components. Generalising this procedure, we can construct a sequence of $2^n$ components with $n$ order-1 definitions[2]. In fact, Cataldo shows that there exists a countably infinite set of terms that must be a tower of exponentials in size with only order-0 constructs, but that may be defined by a higher-order term linear in size [57].

### 2.8.3 Definition

We now introduce higher-order PDSs formally. We begin by describing higher-order stores and their operations. We will then define higher-order PDSs in full.

[2] *Seq* is order-1 since it takes order-0 arguments. *IOComp* is order-0 since it takes no arguments

### Higher-order Stores

**Definition 2.8.1** ($n$-Stores)**.** For $n > 1$, the set of $n$**-stores** $C_n^\Sigma$ is the set of all $[\gamma_1 \ldots \gamma_m]$ with $m \geq 1$ and $\gamma_i \in C_{n-1}^\Sigma$ for all $i \in \{1, \ldots, m\}$.

There are three types of operations applicable to $n$-stores: *push*, *pop* and *top*. These are defined inductively. In addition to the operations over a 1-store, we have, when $n > 1$,

$$\begin{array}{rcll} push_w[\gamma_1 \ldots \gamma_m] & = & [push_w(\gamma_1)\gamma_2 \ldots \gamma_m] & \\ push_l[\gamma_1 \ldots \gamma_m] & = & [push_l(\gamma_1)\gamma_2 \ldots \gamma_m] & \text{if } 2 \leq l < n \\ push_n[\gamma_1 \ldots \gamma_m] & = & [\gamma_1\gamma_1\gamma_2 \ldots \gamma_m] & \\ pop_l[\gamma_1 \ldots \gamma_m] & = & [pop_l(\gamma_1)\gamma_2 \ldots \gamma_m] & \text{if } 1 \leq l < n \\ pop_n[\gamma_1 \ldots \gamma_m] & = & [\gamma_2 \ldots \gamma_m] & \text{if } m > 1 \\ top_l[\gamma_1 \ldots \gamma_m] & = & top_l(\gamma_1) & \text{if } 1 \leq l < n \\ top_n[\gamma_1 \ldots \gamma_m] & = & \gamma_1 & \end{array}$$

Note that we assume without loss of generality that $\Sigma \cap \mathcal{N} = \emptyset$, where $\mathcal{N}$ is the set of natural numbers. Furthermore, observe that when $m = 1$, $pop_n$ is undefined. We define $\mathcal{O}_n = \{ push_w \mid w \in \Sigma^* \} \cup \{ push_l, pop_l \mid 1 < l \leq n \}$. Further more, define $\ell(push_w) = 1$, $\ell(pop_l) = l$ and $\ell(push_l) = l$ for all $1 < l \leq n$.

### Higher-Order Pushdown Systems

**Definition 2.8.2.** An **order-$n$ PDS** is a tuple $(\mathcal{P}, \mathcal{D}, \Sigma)$ where $\mathcal{P}$ is a finite set of control states $p$, $\mathcal{D} \subseteq \mathcal{P} \times \Sigma \times \mathcal{O}_n \times \mathcal{P}$ is a finite set of commands $d$, and $\Sigma$ is a finite alphabet.

A configuration of a higher-order PDS is a pair $\langle p, \gamma \rangle$ where $p \in \mathcal{P}$ and $\gamma$ is an $n$-store. We have a transition $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ iff we have $(p, a, o, p') \in \mathcal{D}$, $top_1(\gamma) = a$ and $\gamma' = o(\gamma)$.

We define $\stackrel{*}{\hookrightarrow}$ to be the transitive closure of $\hookrightarrow$. For a set of configurations $C_{Init}$ we define $Pre^*(C_{Init})$ as the set of configurations $\langle p, \gamma \rangle$ such that, for some configuration $\langle p', \gamma' \rangle \in C_{Init}$, we have $\langle p, \gamma \rangle \stackrel{*}{\hookrightarrow} \langle p', \gamma' \rangle$.

### Alternating Higher-Order Pushdown Systems

We may generalise the previous definition to the case of Alternating higher-order PDSs.

**Definition 2.8.3.** An **order-$n$ APDS** is a tuple $(\mathcal{P}, \mathcal{D}, \Sigma)$ where $\mathcal{P}$ is a finite set of control states $p$, $\mathcal{D} \subseteq \mathcal{P} \times \Sigma \times 2^{\mathcal{O}_n \times \mathcal{P}}$ is a finite set of commands $d$, and $\Sigma$ is a finite alphabet.

A configuration of a higher-order APDS is a pair $\langle p, \gamma \rangle$ where $p \in \mathcal{P}$ and $\gamma$ is an $n$-store. We have a transition $\langle p, \gamma \rangle \hookrightarrow C$ iff we have $(p, a, OP) \in \mathcal{D}$, $top_1(\gamma) = a$, and

$$C = \{ \langle p', \gamma' \rangle \mid (o, p') \in OP \wedge \gamma' = o(\gamma) \} \cup \{ \langle p, \triangledown \rangle \mid \text{if } (o, p') \in OP \text{ and } o(\gamma) \text{ is not defined} \}$$

The transition relation generalises to sets of configurations via the following rule:

$$\frac{\langle p, \gamma \rangle \ \hookrightarrow \ C}{C' \cup \langle p, \gamma \rangle \ \hookrightarrow \ C' \cup C} \quad \langle p, \gamma \rangle \notin C'$$

We define $\stackrel{*}{\hookrightarrow}$ to be the transitive closure of $\hookrightarrow$. For a set of configurations $C_{Init}$ we define $Pre^*(C_{Init})$ as the set of configurations $\langle p, \gamma \rangle$ such that we have $\langle p, \gamma \rangle \stackrel{*}{\hookrightarrow} C$ and $C \subseteq C_{Init}$.

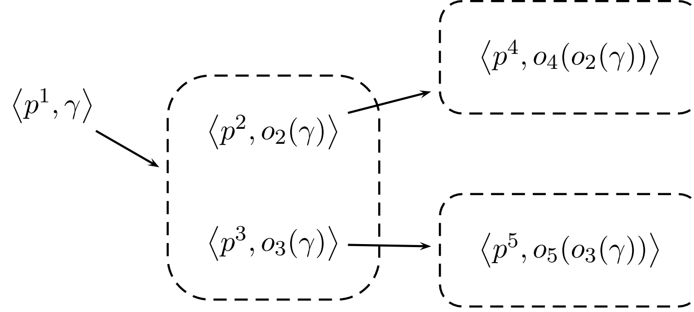



Figure 2.12: The configuration graph (excerpt) of an example higher-order APDS.

**Example 2.8.1.** We present an example to illustrate the definition of $Pre^*(C_{Init})$ for higher-order APDSs. Figure 2.12 shows an excerpt of the configuration graph of a higher-order APDS with the commands,

$$\begin{array}{c}(p^1, _, \{(o_2, p^2), (o_3, p^3)\}) \\ (p^2, _, \{(o_4, p^4)\}) \\ (p^3, _, \{(o_5, p^5)\})\end{array}$$

We consider a number of different values of $C_{Init}$.

1. Let $C_{Init} = \{\langle p^2, o_2(\gamma)\rangle\}$. In this case $Pre^*(C_{Init}) = C_{Init}$. The configuration $\langle p^1, \gamma\rangle$ is not in $Pre^*(C_{Init})$ since the configuration $\langle p^3, o_3(\gamma)\rangle$ cannot be in $Pre^*(C_{Init})$.

2. Let $C_{Init} = \{\langle p^2, o_2(\gamma)\rangle, \langle p^3, o_3(\gamma)\rangle\}$. In this case $Pre^*(C_{Init}) = C_{Init} \cup \{\langle p^1, \gamma\rangle\}$. This is because the transition from $\langle p^1, \gamma\rangle$ reaches a set that is a subset of $C_{Init}$.

3. Let $C_{Init} = \{\langle p^4, o_4(o_2(\gamma))\rangle\}$. Hence $Pre^*(C_{Init}) = C_{Init} \cup \{\langle p^2, o_2(\gamma)\}$. The configuration $\langle p^2, o_2(\gamma)\rangle$ is in the set because its transition moves to a set which is a subset of $C_{Init}$. The pair $\langle p^1, \gamma\rangle$ is not in the set because, although $\langle p^2, o_2(\gamma)\rangle$ is in $Pre^*(C_{Init})$, the configuration $\langle p^3, o_3(\gamma)\rangle$ is not.

4. Let $C_{Init} = \{\langle p^4, o_4(o_2(\gamma))\rangle, \langle p^3, o_3(\gamma)\rangle\}$. We therefore have $Pre^*(C_{Init}) = C_{Init} \cup \{\langle p^2, o_2(\gamma)\rangle, \langle p^1, \gamma\rangle\}$. We have $\langle p^2, o_2(\gamma)\rangle \in Pre^*(C_{Init})$ as before. Furthermore, we have the following run from $\langle p^1, \gamma\rangle$,

$$\langle p^1, \gamma\rangle \hookrightarrow \{\langle p^2, o_2(\gamma)\rangle, \langle p^3, o_3(\gamma)\rangle\} \hookrightarrow \{\langle p^4, o_4(o_2(\gamma))\rangle, \langle p^3, o_3(\gamma)\rangle\}$$

   Hence, $\langle p^1, \gamma\rangle \in Pre^*(C_{Init})$.

Finally, suppose the higher-order APDS also has a command of the form,

$$(p^5, _, \{(pop_l, p^4)\})$$

And it is the case that (only) $pop_l(o_5(o_3(\gamma)))$ is undefined. If $C_{Init} = \{\langle p^5, \triangledown\rangle\}$, then $Pre^*(C_{Init}) = C_{Init} \cup \{\langle p^5, o_5(o_3(\gamma))\rangle, \langle p^3, o_3(\gamma)\rangle\}$.

Observe that since no transitions are possible from an "undefined" configuration $\langle p, \triangledown\rangle$ we can reduce the reachability problem for higher-order PDSs to the reachability problem over higher-order APDSs in a straightforward manner.

### Higher-Order Pushdown Automata

The definition of higher-order pushdown automata is analogous to the order-1 case. That is, a higher-order pushdown automaton is a higher-order pushdown system which reads an input word.

**Definition 2.8.4.** An **order-$n$ PDA** is a tuple $(\mathcal{P}, \mathcal{D}, \Sigma, \Gamma, c_0, \mathcal{W})$ where $\mathcal{P}$ is a finite set of control states $p$, $\mathcal{D} \subseteq \mathcal{P} \times \Gamma \times \Sigma \times \mathcal{O}_n \times \mathcal{P}$ is a finite set of commands $d$, $\Sigma$ is a finite stack alphabet, $\Gamma$ is a finite input alphabet, $c_0$ is an initial configuration and $\mathcal{W} \subseteq C_n^\Sigma$ is an acceptance condition.

A configuration of a higher-order PDA is a pair $\langle p, \gamma \rangle$ where $p \in \mathcal{P}$ and $\gamma$ is an $n$-store. We have a transition $\langle p, \gamma \rangle \stackrel{\alpha}{\hookrightarrow} \langle p', \gamma' \rangle$ iff we have $(p, \alpha, a, o, p') \in \mathcal{D}$, $top_1(\gamma) = a$ and $\gamma' = o(\gamma)$.

A word $\alpha_1, \alpha_2, \ldots$ is accepted by the automaton iff $c_0 c_1 c_2 \ldots \in \mathcal{W}$ and,

$$c_0 \stackrel{\alpha_1}{\hookrightarrow} c_1 \stackrel{\alpha_2}{\hookrightarrow} c_3 \stackrel{\alpha_3}{\hookrightarrow} \cdots$$

## 2.8.4 The Caucal Hierarchy

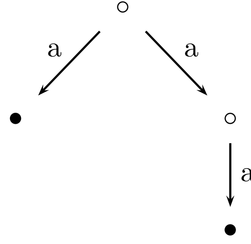
The hierarchy of pushdown automata/systems is closely related to several other theoretical constructs, and hence, can be considered robust. In particular, as acceptors of word languages, order-$n$ pushdown automata are equivalent to level-$n$ *safe* OI word grammars [137] and as generators of trees, deterministic order-$n$ pushdown automata are equivalent to *safe* order-$n$ recursion schemes (discussed in Section 2.8.6) [132]. Furthermore the $\varepsilon$-closure of the configuration graphs of higher-order PDAs match the Caucal hierarchy [36].

In this section we provide an account of the Caucal hierarchy. After the definition, to give some intuition behind the hierarchy's connection with higher-order pushdown automata, we discuss the simulation, due to Cachat, of order-2 pushdown games by games played over Caucal graphs [128].

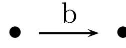
### Definition

The Caucal hierarchy contains a hierarchy of trees and a hierarchy of graphs. At level 0 of the hierarchy are the family of finite graphs and the family of finite trees over a finite alphabet $\Sigma$. Two MSO-preserving operations are alternately applied to trees and graphs to construct the entire hierarchy. An MSO-preserving operation is one which preserves MSO decidability. Let $\overline{\Sigma}$ be the set $\{ \bar{a} \mid a \in \Sigma \}$ disjoint from $\Sigma$. Intuitively, the letter $\bar{a}$ denotes an $a$-labelled directed edge being traversed backwards. We define the following MSO-preserving operations.

- A rational mapping $h \subseteq \Sigma \times (\Sigma \cup \overline{\Sigma})^*$ maps a character $a \in \Sigma$ to a rational (regular) language. Given a rational mapping $h$, an **inverse rational mapping** $h^{-1}(T)$ of a tree $T$ is the graph,
$$\{ s \xrightarrow{a} t \mid \exists w \in h(a).s \xrightarrow{w} t \text{ in } T \}$$
where the nodes of the graph are the nodes appearing in an edge of the set. For example, if $h(b) = \bar{a}aa$, and $T$ is the tree,




then $h^{-1}(T)$ is the graph,



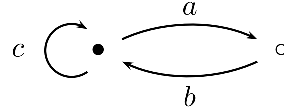

The inverse rational mapping operation was shown to be MSO-preserving by Caucal in 1996 [35]. Beginning with the regular trees, the graphs obtainable by an inverse rational mapping are the prefix recognisable graphs (PreRec.) [35, 36].

- The **unfolding** $Unf(G, r)$ of a graph $G$ from a node $r$ in $G$ is the tree,

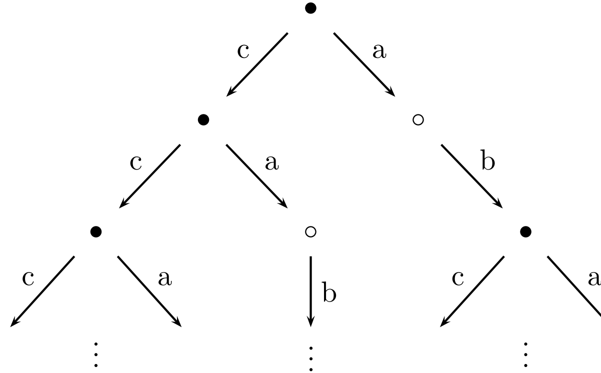
$$\{ \ ws \xrightarrow{a} wsat \mid wsat \in Path(G, r) \text{ and } s \xrightarrow{a} t \text{ in } G \ \}$$

  where the nodes of the tree are all nodes appearing in an edge of the set. The root of the tree is the node $r$. Furthermore, we have $r \in Path(G, r)$ and $wsat \in Path(G, r)$ iff $ws \in Path(G, r)$ and $s \xrightarrow{a} t$ is an edge of $G$.

  Let $G$ be the graph,




  The unfolding $Unf(G, \bullet)$ of $G$ from $\bullet$ is the infinite tree,




  Courcelle and Walukiewicz have proven the unfolding operation is MSO-preserving [28].

With these operations, we define the **Caucal hierarchy**:

$$Graph_n := Rat^{-1}(Tree_n) \quad \text{and} \quad Tree_{n+1} := Unf(Graph_n)$$

where $Rat^{-1}(Tree_n)$ is the class of graphs obtainable by an inverse rational mapping from a tree in $Tree_n$ and $Unf(Graph_n)$ is the class of trees obtainable by unfolding a graph $G$ in $Graph_n$ from a node in $G$.

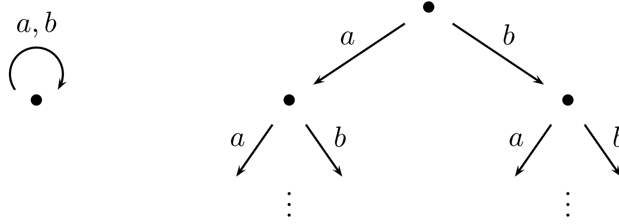



Figure 2.13: The tree $T_1$ for the stack alphabet $\{a, b\}$ obtained by unfolding the graph on the left

### Connections to Higher-Order Pushdown Automata/Systems

The Caucal hierarchy is closely related to pushdown automata/systems. In fact, if $HOPDG(n)$ denotes the graphs obtainable from the $\varepsilon$-closure of a higher-order pushdown automaton, we have the following result due to Carayol and Wörhle:

**Theorem 2.8.1** ([13])**.** *For every $n \geq 0$, $G \in HOPDG(n) \iff G \in Graph_n$.*

This result was first shown, in the order-1 case, by Courcelle in 1995 [27].

A similar result was obtained by Cachat [128] connecting games played over higher-order pushdown systems and a kind of graph automata operating over the Caucal hierarchy using a notion of *game simulation*. In particular:

**Theorem 2.8.2** ([128])**.** *Given a game $\mathcal{G}$ played over an order-$n$ pushdown system one can construct a graph automaton $A$ and a tree $T \in Tree_n$ such that $\mathcal{G}$ is game-simulated by $(T, A)$.*

**Theorem 2.8.3** ([128])**.** *Given a graph $G \in Graph_n$ and a graph automaton $A$, one can construct a game structure $\mathcal{G}$ on an order-$n$ pushdown system such that $(G, A)$ is game-simulated by $\mathcal{G}$.*

### The Order-2 Case

We give a flavour of the connections above by describing the game-simulation of an order-2 pushdown game by a graph automaton $A$ and a tree $T_2 \in Tree_2$.

The tree $T_2 \in Tree_2$ is used to represent the stack contents of the game. The graph automaton navigates this tree. Thus, information about the current control state of the pushdown game is stored in the graph automaton, whilst the stack contents are represented by the position of the graph automaton in the tree $T_2$. The moves that the graph automaton can make simulate the moves available in the pushdown game, and their effect on the pushdown store.

The tree $T_2$ is derived from a tree $T_1 \in Tree_1$, which is used in the order-1 case to simulate an order-1 stack. The tree $T_1$ is itself the unfolding of a finite graph (in $Graph_0$) and is shown in Figure 2.13 (for the stack alphabet $\{a, b\}$). In the tree $T_1$, the stack identified by a node is the sequence of transitions required to reach it. For example, the node reachable via the sequence $aab$ represents the stack $baa$.

To generate the tree $T_2$, representing order-2 stacks, we first construct a graph $G_1$ by an inverse rational mapping from $T_1$. This graph is unfolded to form $T_2$. The inverse rational

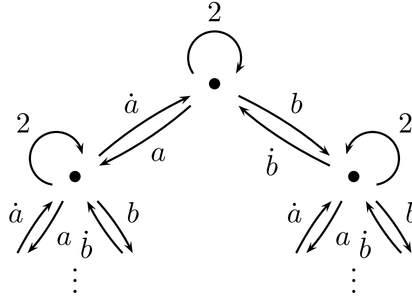


Figure 2.14: The graph $G_1 = h^{-1}(T_1)$ for the stack alphabet $\{a, b\}$

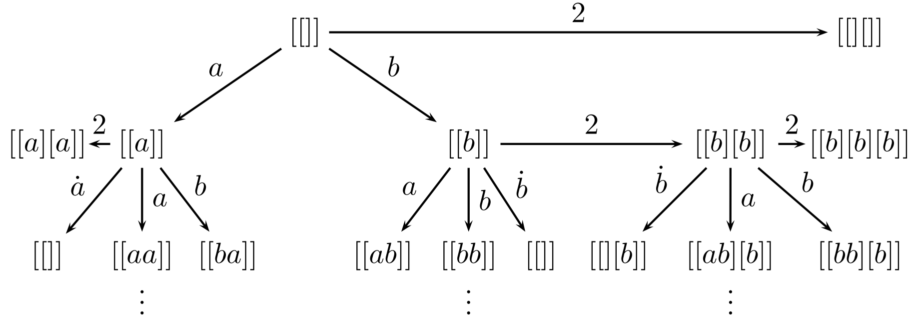


Figure 2.15: The tree $T_2$, unfolded from $G_1$

mapping is given below. Let $\dot{a}$ be a fresh character for every stack character $a \in \Sigma$, similarly, let 2 be a fresh character.

$$\begin{array}{rcl} h(a) & = & a \text{ for all } a \in \Sigma \\ h(\dot{a}) & = & \bar{a} \text{ for all } a \in \Sigma \\ h(2) & = & \varepsilon \end{array}$$

A character $\dot{a}$ represents the removal, from the top of the stack, of the character $a$. The character 2 represents a $push_2$ action. The graph $G_1 = h^{-1}(T_1)$ is shown in Figure 2.14. This graph unfolds to the tree, $T_2$, shown in Figure 2.15. The nodes are labelled with the stack contents they represent.

The graph automaton reading $T_2$ can simulate moves of the pushdown game by updating its control state and navigating $T_2$ to account for changes in the stack. In particular,

$$\begin{array}{rll} (p, a, push_w, p') & \text{corresponds to the path} & \dot{a}w \\ (p, a, pop_1, p') & \text{corresponds to the path} & \dot{a} \\ (p, a, push_2, p') & \text{corresponds to the path} & \dot{a}a2 \\ (p, a, pop_2, p') & \text{corresponds to the path} & \dot{a}\overline{\Sigma}^*\bar{2} \end{array}$$

Each path begins with an $\dot{a}$-transition to ensure that the current $top_1$ symbol of the stack matches the pushdown command. In the case of a $push_w$ operation, we then take the path $w$. This has the effect of removing the $a$ from the top of the stack (the $\dot{a}$-transition) and replacing it with the word $w$. A $pop_1$ command is identical to the case $w = \varepsilon$. The $push_2$ command replaces the $a$ removed by the $\dot{a}$-transition before taking a 2-transition, which corresponds to duplications of the top stack. Finally, the $pop_2$ operation moves up the tree via $\overline{\Sigma}$ characters

until it has reached the node representing the stack after the last $push_2$ operation. Then the 2-transition is taken in reverse, undoing the $push_2$.

### 2.8.5 Local Model-Checking of Higher-Order Pushdown Systems

Cachat used his game simulation result described in the previous section to provide a solution to the local model-checking problem for parity games [128]. This result reduces an order-$n$ pushdown game to an equivalent game played over a Caucal tree in $Tree_n$. He then shows that a game tree at level $n$ of the hierarchy can be simulated by a game graph at level $n-1$ and that this graph game can be simulated by a tree game at level $n-1$. By repeating these reductions we are left with a parity game played over a finite graph, for which solutions are well known.

**Theorem 2.8.4** ([128]). *Parity games on higher-order pushdown systems are solvable: one can determine the winner and compute a winning strategy.*

A more direct solution was presented by Serre in a more general setting [76]. The algorithm is a generalisation of the order-1 algorithm by Walukiewicz described in Section 2.6.3. Whereas Walukiewicz reduced an order-1 pushdown game to a game played over a finite graph, Serre reduces an order-$n$ pushdown game to a game played over an order-$(n-1)$ pushdown system. An order-$n$ pushdown game can be simulated by an order-$(n-1)$ pushdown game as follows:

- Moves of the form $(p, a, o, p')$ where $\ell(o) < n$ are simulated directly.
- If a move of the form $(p, a, push_n, p')$ is played, Éloïse is required to make guarantees about state of play when the new $top_n$ stack is removed. Her announcement consists of a tuple $(A_1, \ldots, A_m) \in \mathcal{Q}^m$ where $m$ is the number of priorities in the game. The meaning of this statement is that when the new stack is removed, if the smallest priority encountered since the $push_n$ operation is $i$ and the current control state is $p$, then $p \in \mathcal{P}_i$.

  Abelard then decides whether to accept this guarantee, by choosing the next control state $p' \in \mathcal{P}_i$ to move to, or challenge this statement. When the guarantee is accepted, the move has a priority $i$. This allows the parity condition to be checked correctly. In the case of a challenge, play moves to control state $p'$, but the stack is left unchanged. This is because we only store the $top_n$ stack, which, in this case, is a duplicate of the previous $top_n$ stack. Similarly, in the order-1 case, only the top character was remembered.

- If a move of the form $(p, a, pop_n, p')$ is played, the state of the game is checked against Éloïse's announcement. If she was correct, play moves to a state $Pop(p')$ and Éloïse is declared the winner. Otherwise, play moves to a state $Err(p')$ and Abelard wins the game.

Since the reduction eliminates all order-$n$ commands, we are left with an order-$(n-1)$ pushdown game. We can repeat this reduction until we have an order-1 game, to which the solution is known.

More formally, from an order-$n$ pushdown parity game $\mathcal{G}_n$ we define an order-$(n-1)$ game $\mathcal{G}_{n-1}$. The states of $\mathcal{G}_{n-1}$ have four core components: $\vec{A}$, $\gamma$, $\theta$ and $q$:

- $\vec{A}$ — a tuple $(A_0, \ldots, A_m) \subseteq \mathcal{P}^m$ storing Éloïse's current claim.

- $\gamma$ — $top_n$ of the current stack.
- $\theta$ — the smallest priority seen since the last $push_n$ simulation.
- $p$ — the current control state.

**Definition 2.8.5.** Given a pushdown parity game $\mathcal{G}_n = (\mathcal{P} = \mathcal{P}_A \uplus \mathcal{P}_E, \mathcal{D}, \Sigma, \Omega)$, the equivalent order-$(n-1)$ game $\mathcal{G}_{n-1}$ has control states $\mathcal{P}' = \mathcal{P}'_A \uplus \mathcal{P}'_E$, for every $\vec{A}, \vec{A_1}, \theta, p, p_1$, all $c \in \{1, \ldots, m\}$ and a special symbol ?,

$$\begin{array}{lll} Check(\vec{A}, \theta, c, p) & Push(\vec{A}, \theta, p) & Pop(p) \\ Move((\vec{A}, \theta, p), (?, p_1)) & Move((\vec{A}, \theta, p), (\vec{A_1}, p_1)) & Err(p) \end{array}$$

To ease notation, we write,

$$\begin{array}{lll} Check(\vec{A}, \gamma, \theta, c, p) & Push(\vec{A}, \gamma, \theta, p) & Pop(p, \gamma) \\ Move((\vec{A}, \gamma, \theta, p), (?, p_1)) & Move((\vec{A}, \gamma, \theta, p), (\vec{A_1}, \gamma, p_1)) & Err(p, \gamma) \end{array}$$

rather than using standard configuration syntax (for example $\langle Check(\vec{A}, \theta, c, p), \gamma \rangle$). Let $z = top_1(\gamma)$ and $m_{\Omega,\theta} = min(\Omega(p), \theta)$, the moves in $\mathcal{G}_{n-1}$ are:

$$\begin{array}{rcll} Check(\vec{A}, \gamma, \theta, c, p) & \to & Check(\vec{A}, o(\gamma), min(\Omega(p), \theta), c, p') & \text{if } (p, z, o, p') \in \mathcal{D}, o \in \mathcal{O}_{n-1} \\ Check(\vec{A}, \gamma, \theta, c, p) & \to & Pop(p', \gamma) & \text{if } (p, z, pop_n, p') \in \mathcal{D}, p' \in A_{m_{\Omega,\theta}} \\ Check(\vec{A}, \gamma, \theta, c, p) & \to & Err(p', \gamma) & \text{if } (p, z, pop_n, p') \in \mathcal{D}, p' \notin A_{m_{\Omega,\theta}} \\ Check(\vec{A}, \gamma, \theta, c, p) & \to & Move((\vec{A}, \gamma, \theta, p), (?, \gamma, p_1)) & \text{if } (p, z, push_n, p_1) \in \mathcal{D} \end{array}$$

$$\begin{array}{rcll} Push(\vec{A}, \gamma, \theta, c, p) & \to & Check(\vec{A}, o(\gamma), min(\Omega(p), \theta), c, p') & \text{if } (p, z, o, p') \in \mathcal{D}, o \in \mathcal{O}_{n-1} \\ Push(\vec{A}, \gamma, \theta, c, p) & \to & Pop(p', \gamma) & \text{if } (p, z, pop_n, p') \in \mathcal{D}, p' \in A_{m_{\Omega,\theta}} \\ Push(\vec{A}, \gamma, \theta, c, p) & \to & Err(p', \gamma) & \text{if } (p, z, pop_n, p') \in \mathcal{D}, p' \notin A_{m_{\Omega,\theta}} \\ Push(\vec{A}, \gamma, \theta, c, p) & \to & Move((\vec{A}, \gamma, \theta, p), (?, \gamma, p_1)) & \text{if } (p, z, push_n, p_1) \in \mathcal{D} \end{array}$$

and finally,

$$\begin{array}{rcll} Move((\vec{A}, \gamma, \theta, p), (?, \gamma, p_1)) & \to & Move((\vec{A}, \gamma, \theta, p), (\vec{A_1}, \gamma, p_1)) & \\ Move((\vec{A}, \gamma, \theta, p), (\vec{A_1}, \gamma, p_1)) & \to & Push(\vec{A_1}, \gamma, m, p_1) & \\ Move((\vec{A}, \gamma, \theta, p), (\vec{A_1}, \gamma, p_1)) & \to & Check(\vec{A}, \gamma, min(\theta, c), c, p_2) & \text{if } c \leq \Omega(p) \text{ and } p_2 \in A_c \end{array}$$

The owner of each control state is determined by the component $p$ except in the case of $Move$ states. That is,

$$\begin{array}{rcl} \mathcal{Q}_E & = & \{ \ Move(\vec{A}, \theta, p_1), (?, p_2)), Push(\vec{A}, \theta, p), Check(\vec{A}, \theta, c, p) \mid p \in \mathcal{P}_E \ \} \\ \mathcal{Q}_A & = & \{ \ Move(\vec{A}, \theta, p_1), (\vec{A_1}, p_2)), Push(\vec{A}, \theta, p), Check(\vec{A}, \theta, c, p) \mid p \in \mathcal{P}_A \ \} \end{array}$$

We assign the following priorities to the states of $\mathcal{G}_{n-1}$: $Check(\vec{A}, \theta, c, p)$ has priority $c$, $Push(\vec{A}, \theta, p)$ has priority $\Omega(p)$ and all other states have priority $m + 1$.

Éloïse wins the game if Abelard cannot make a move, play reaches $Pop(p)$ for some $p$ or an infinite path is generated such that the lowest priority occurring infinitely often is even. The initial state of the game is $Check((\emptyset, \ldots, \emptyset), \gamma_0, m, m, p_0)$ where $p_0$ is the initial control state of $\mathcal{G}_n$ and $\gamma_0$ is the initial stack.

**Theorem 2.8.5** ([53])**.** *Éloïse has a winning strategy in the game $\mathcal{G}_n$ iff she has a winning strategy in the game $\mathcal{G}_{n-1}$ from the node $Check((\emptyset, \ldots, \emptyset), \gamma_0, m, m, p_0)$.*

The size of $\mathcal{G}_{n-1}$ is exponential in the size of $\mathcal{G}_n$, since each state contains $m$ subsets of the control states of $\mathcal{G}_n$. Combined with the EXPTIME complexity of calculating the winner of an order-1 parity game, we can determine the winner of an order-$n$ pushdown game in $n$-EXPTIME. This is optimal [129] (also see Section 6.4).

### 2.8.6 Recursion Schemes

In addition to their relationship with the Caucal hierarchy, higher-order pushdown automata are closely related to higher-order recursion schemes/grammars. In this setting, both higher-order pushdown automata and higher-order recursion schemes are defined as generators of trees. It was shown by Courcelle that, in the order-1 case, trees generated by the two formalisms coincide [27]. This result was extended by Knapik, Niwiński and Urzyczyn, first to the order-2 case [131] and then to the general case [132]. A key restriction in these extensions is that the recursion scheme must satisfy a constraint called *safety*. Hence we have the following theorems:

**Theorem 2.8.6** ([132])**.** *If $t$ is a tree accepted by an order-$n$ pushdown automaton, then it is generated by a safe, homogeneous order-$n$ recursion scheme.*

**Theorem 2.8.7** ([132])**.** *A tree generated by a safe, homogeneous order-$n$ recursion scheme is accepted by an order-$n$ pushdown automaton.*

These results imply that MSO is decidable in the case of homogeneous, safe, higher-order recursion schemes. More recently, it has been shown that MSO is decidable in the more general case, which permits non-homogeneous, unsafe recursion schemes [29].

#### Definition

A **higher-order recursion scheme** is a tuple $\mathcal{R} = (\Sigma, V, S, E)$ where $\Sigma$ is a finite ranked alphabet, $V$ is a finite set of *non-terminals*, $S$ is a start symbol and $E$ is a finite set of productions. A production is of the form:

$$F z_1 \ldots z_m \Rightarrow w$$

where $F \in V$ is a non-terminal of arity $m$ and $z_1, \ldots, z_m$ are variables of the required *type*. Finally $w$ is an *applicative term*.

The set of possible types of $F$ are constructed from a unique basic type $o$. Then, if $\tau_1$ and $\tau_2$ are types, $(\tau_1 \to \tau_2)$ is also a type. We define $O(\tau_1)$ as the order of a type. For $o$ we have $O(o) = 0$, and a type $(\tau_1 \to \tau_2)$ has $O(\tau_1 \to \tau_2) = max(1 + O(\tau_1), O(\tau_2))$. A type $(\tau_1 \to \ldots \to \tau_m)$ is **homogeneous** if $O(\tau_1) \geq \ldots \geq O(\tau_m)$. We write $t : \tau$ to indicate that a term $t$ is of type $\tau$.

In the above, we require $F$ to be of homogeneous type $\tau_1 \to \ldots \to \tau_m$ and each $z_i$ for all $1 \leq i \leq m$ to be of type $\tau_i$. The order of a recursion scheme is the largest order of its non-terminals. It is assumed that there is only one production per non-terminal $F$ and $\tau_m$ is type $o$. Furthermore, $\Sigma$ is a typed alphabet of order 1.

The right-hand side of a production is an applicative term from the set $T(\Sigma \cup V \cup \{z_1, \ldots, z_m\})$, which is defined inductively:

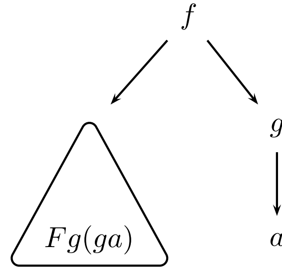


Figure 2.16: The tree $f(Fg(ga))(ga)$

1. $\Sigma \cup V \cup \{z_1, \ldots, z_m\} \subseteq T(\Sigma \cup V \cup \{z_1, \ldots, z_m\})$.
2. If $t \in T(\Sigma \cup V \cup \{z_1, \ldots, z_m\})$ is of type $(\tau_1 \to \tau_2)$ and $s \in T(\Sigma \cup V \cup \{z_1, \ldots, z_m\})$ has type $\tau_1$, then $ts \in T(\Sigma \cup V \cup \{z_1, \ldots, z_m\})$ and has type $\tau_2$.

We consider recursion schemes as generators of ranked trees. This process is illustrated with the following example:

$$\begin{array}{rcl} S & = & Fga \\ F\phi x & = & f(F\phi(\phi x))(\phi x) \end{array}$$

where $S$ is of type $o$, $F : (o \to o) \to o$, $f : o \to o \to o$, $g : o \to o$ and $a : o$, with $f, g, a \in \Sigma$. A node labelled with $f$ has arity 2. Similarly, nodes labelled with $g$ or $a$ have arities 1 and 0 respectively.

Beginning from $S$, the tree is generated from the step-wise expansion of $S$:

$$S \gg Fga \gg f(Fg(ga))(ga) \gg \ldots$$

resulting in the tree shown in Figure 2.16. A term of the form $fXY$ gives a node labelled $f$ with two children: the trees defined by $X$ and $Y$. The trees constructed are not ordered. Instead, the branches are (implicitly in Figure 2.16 and the sequel) labelled with a natural number corresponding to the order of the arguments to $f$. Using,

$$Fg(ga) \gg f(Fg(g(ga))(g(ga))$$

the tree is expanded further in Figure 2.17. In this way, an infinite tree is formed.

### Higher-order Pushdown Systems as Tree Generators

Order-$n$ pushdown systems have pushdown commands of the form for $2 \leq l \leq n$ and $w \in \Sigma^*$:

$$(p, a, push_w, p') \qquad (p, a, push_l, p') \qquad (p, a, pop_l, p')$$

To extend higher-order pushdown systems to generators of trees, we allow commands of the following form, where $r$ is the arity of $f \in \Sigma$.

$$(p, a, f, p_1, \ldots, p_r)$$

A run of a tree-generating higher-order pushdown system proceeds as expected until a command of the above form is applied. At this point a node $f$ is generated with $r$ children.

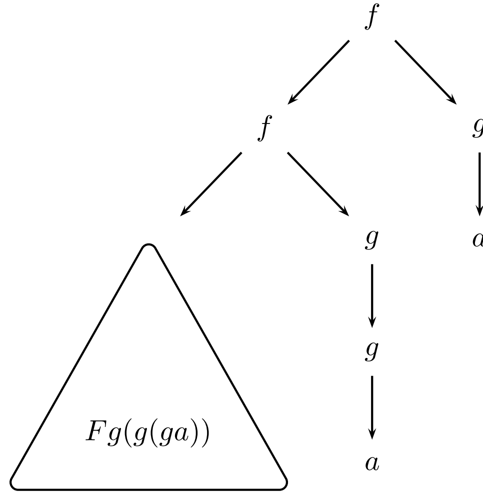



Figure 2.17: The tree $f(f(Fg(g(ga))(g(ga)))(ga)$

To generate the children of $f$, we run $r$ copies of the pushdown system from the control states $p_1, \ldots, p_r$ respectively. For example, the pushdown system defined by the commands,

$$\begin{array}{lll} (F_1, _, push_a, F_2) & (G_2, \bot, a, end) & (G_1, a, g, G_2) \\ (F_2, a, f, F_1, G_1) & (G_2, a, pop_1, G_1) & \end{array}$$

from the initial configuration $\langle F_1, [\bot] \rangle$ runs as shown in Figure 2.18, generating the tree in Figure 2.17.

### Safety

The class of trees generated by higher-order pushdown systems are equivalent to the class of trees generated by higher-order recursion schemes satisfying a constraint called *safety*. Since higher-order pushdown systems have a decidable MSO-theory, it follows that the MSO-theory of safe higher-order recursion schemes is also decidable. In fact, a result due to Ong states that the MSO-theory of non-homogeneous, unsafe higher-order recursion schemes is decidable [29].

**Definition 2.8.6** ([132])**.** A term of order $k > 0$ is **unsafe** if it contains an occurrence of a parameter of order strictly less that $k$, otherwise the term is *safe*. An occurrence of an unsafe term $t$ as a subexpression of a term $t'$ is **safe** if it is in the context $\ldots(ts)\ldots$, otherwise, the occurrence is *unsafe*. A grammar is *safe* if no unsafe term has an unsafe occurrence at a right-hand side of any production.

**Example 2.8.2** ([132])**.** Let $f, g, h, a, b$ be signature symbols of arity $2, 1, 1, 0, 0$, respectively. Furthermore, let $F$ and $\mathcal{S}$ be non-terminals of type $(o \to o) \to o \to o \to o$ and $o$ respectively. The following grammar is unsafe:

$$\begin{array}{rcl} \mathcal{S} & = & Fgab \\ F\phi xy & = & f(F(\underline{F\phi x})y(hy))(f(\phi x)y) \end{array}$$

In particular, the sub-term $t = F\phi x : (o \to o)$ is a term of order 1 containing an occurrence of an order 0 parameter. Since it does not occur as part of a term $\ldots(ts)\ldots$, it is unsafe. Therefore, the grammar is not safe.

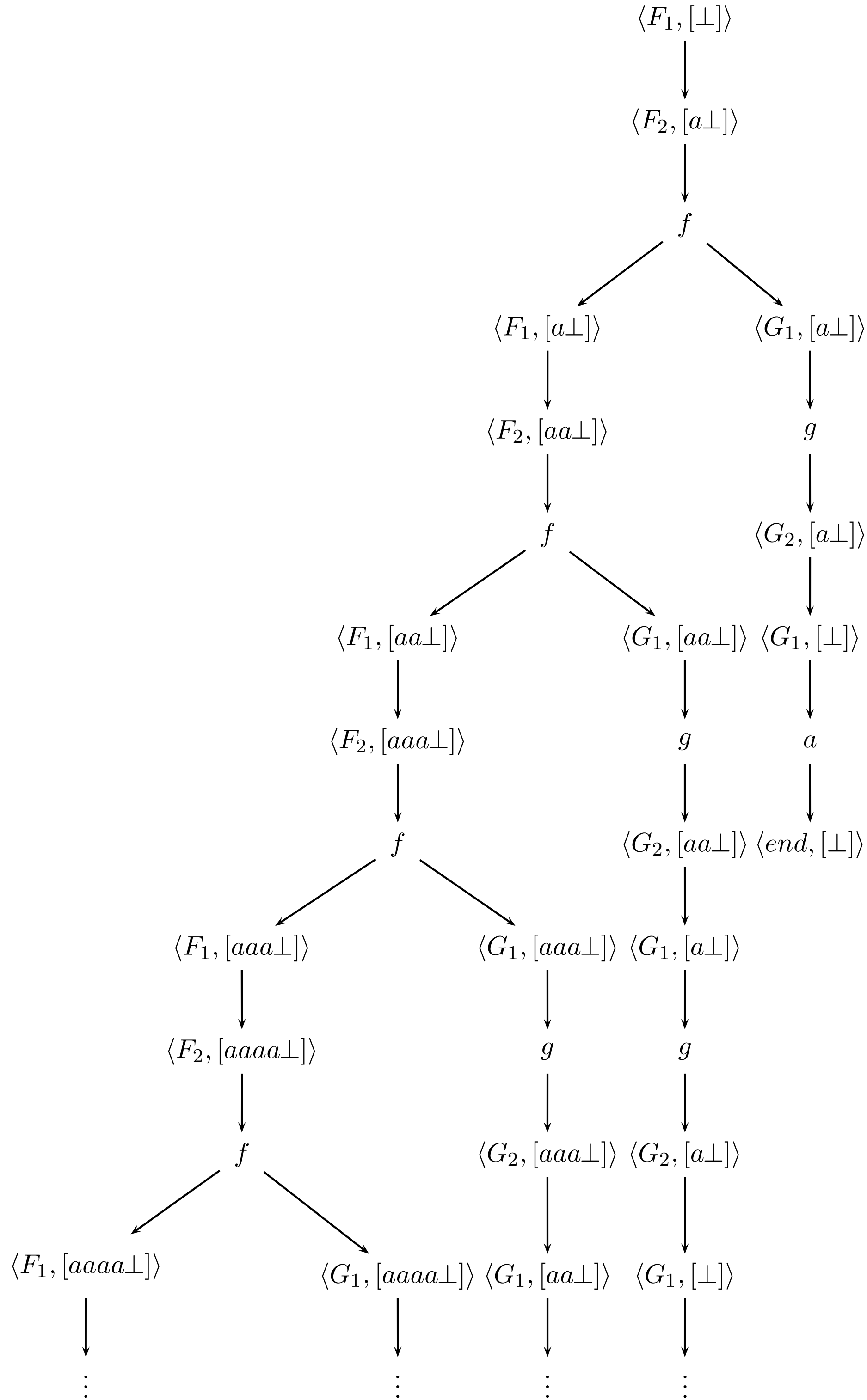



Figure 2.18: The run of a PDS generating the tree in Figure 2.17

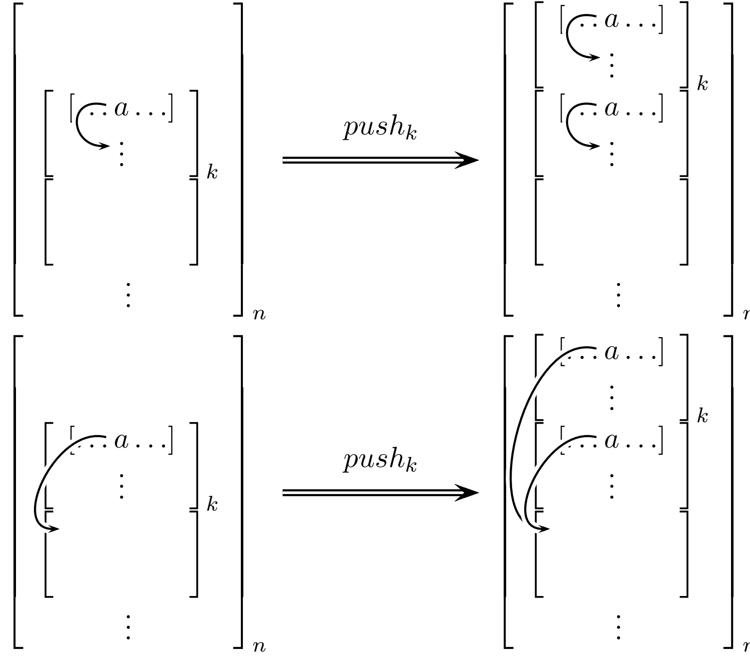



Figure 2.19: The behaviour of a collapse link for a $push_k$ operation

### 2.8.7 Collapsible Pushdown Systems

Since higher-order recursion schemes have a decidable MSO-theory, even in the case of non-homogeneous, unsafe schemes, it follows that higher-order pushdown systems do not capture the full range of structures with decidable MSO theories. **Collapsible Pushdown Systems** (CPDS) are an extension of higher-order pushdown systems which fully capture the trees and graphs definable by higher-order recursion schemes [76]. They are an order-$n$ extension of the order-2 **panic automata** introduced by Knapik *et al.* [133] or **pushdown automata with links** introduced by Aehlig *et al.* [69] to all finite orders.

A collapsible pushdown store is a higher-order store equipped with links. These links are added when a character is added to the top order-1 stack. An operation $push_a^{k'}$ adds the character $a$ to the $top_1$ stack with a link pointing to the order-$k'$ stack below, where $0 \leq k' < n$. Figure 2.19 shows the state of the links after a $push_k$ operation when $k' < k$ and $k' \geq k$ respectively.

Collapsible pushdown systems allow *collapse* operations in addition to the usual push and pop operations. When a *collapse* occurs, the portion of the stack above the destination of the current top link is removed: the stack is *collapsed* to a previous state. This is illustrated in Figure 2.20. A $push_a^0$ operation creates a link to the next order-0 stack, that is, the previous $top_1$ element. In this case, *collapse* behaves like $pop_1$. We define the set of CPDS operations,

$$\begin{array}{rcl} \mathcal{O}_1^C & = & \{skip, pop_1, collapse\} \cup \{\ push_a^0 \mid a \in \Sigma\ \} \\ \mathcal{O}_n^C & = & \{push_n, pop_n\} \cup \{\ push_a^{n-1} \mid a \in \Sigma\ \} \cup \mathcal{O}_{n-1}^C \end{array}$$

**Definition 2.8.7.** An **order-$n$ CPDS** is a tuple $(\mathcal{P}, \mathcal{D}, \Sigma)$ where $\mathcal{P}$ is a finite set of control states $p$, $\mathcal{D} \subseteq \mathcal{P} \times \Sigma \times \mathcal{O}_n^C \times \mathcal{P}$ is a finite set of commands $d$ and $\Sigma$ is a finite alphabet.

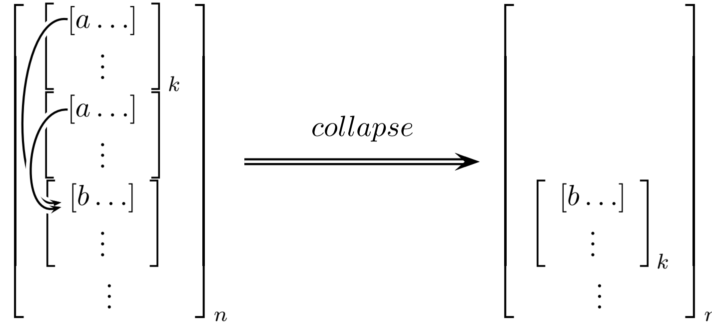



Figure 2.20: The behaviour of a *collapse* operation

A configuration of a CPDS is a pair $\langle p, \gamma \rangle$ where $p \in \mathcal{P}$ and $\gamma$ is an $n$-store with links. We have a transition $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ iff we have $(p, a, o, p') \in \mathcal{D}$, $top_1(\gamma) = a$ and $\gamma' = o(\gamma)$.

### Generating Trees

CPDSs may be defined as generators of trees using an extension similar to the higher-order PDS case. From a CPDS $A$, we can construct a recursion scheme $R_A$ which generates the same trees. Intuitively, a configuration $\langle q, \gamma \rangle$ where $top_1(\gamma) = a$ has a link to an order-$e$ stack is represented by a term of the form,

$$F_q^{a,e} \overrightarrow{\phi_c} \overrightarrow{\phi_n} \ldots \overrightarrow{\phi_1}$$

The vector of terms $\overrightarrow{\phi_c}$ contains information about the state of the stack after a *collapse* is performed, and the vectors $\overrightarrow{\phi_n}, \ldots, \overrightarrow{\phi_1}$ correspond to $pop_k$ operations for $n \leq k \leq 1$.

**Theorem 2.8.8** ([76])**.** *Let $A$ be a tree-generating CPDS, and let $R_A$ be the recursion scheme determined by $A$. Then the CPDS and the recursion scheme generate the same trees.*

In the other direction, we can use a collapsible pushdown system $A_R$ to evaluate a recursion scheme $R$. The CPDS computes traversals over the computation tree of $R$ based on a game semantics interpretation.

**Theorem 2.8.9** ([76])**.** *For every order-$n$ recursion scheme $R$, $A_R$ computes all paths in the tree generated by $R$.*

### MSO Decidability

The above connections with recursion schemes imply CPDSs have a decidable MSO-theory when viewed as generators of trees. However, the same is not true for graphs. Consider the following CPDS $A_{grid}$:

$$\begin{array}{lll} (q_0, _, push_2, q_1) & (q_1, _, push_a^1, q_0) & (q_2, \backslash\bot, collapse, q_0) \\ (q_1, _, push_b^1, q_2) & (q_2, \backslash\bot, pop_1, q_1) & \end{array}$$

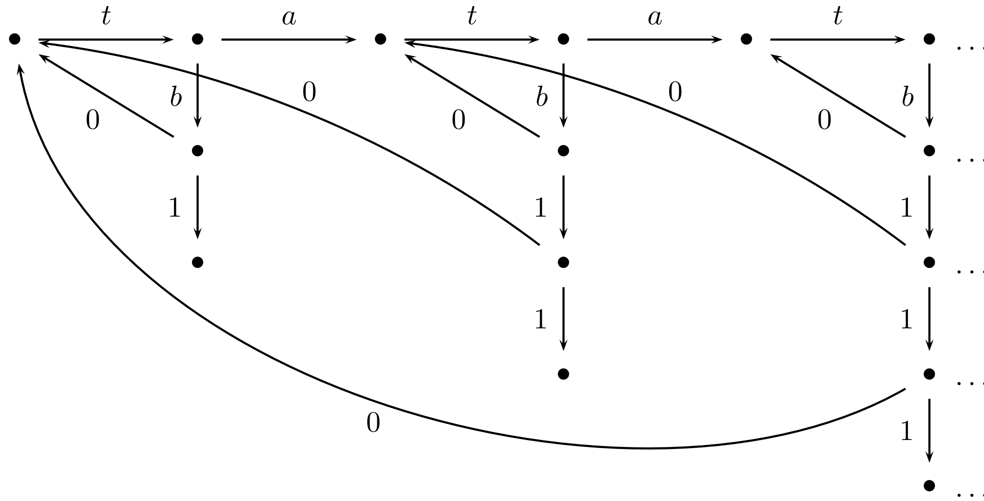



Figure 2.21: The configuration graph of $A_{grid}$

where $\backslash\bot$ indicates any non-$\bot$ character. From the configuration $\langle q_0, [[\bot]]\rangle$ the CPDS generates the graph in Figure 2.21. We use the abbreviations $t, a, b, 0, 1$ for the operations $push_2$, $push^1_a$, $push^1_b$, *collapse* and $pop_1$ respectively. We take the following inverse rational mapping of the graph generated by $A_{grid}$:

$$
\begin{array}{rcl}
h(A) & = & \bar{1}^*\bar{b}atb1^* \;\cap\; (0ta\bar{0} \;\cup\; \bar{1}0ta\bar{0}1) \\
h(B) & = & 1
\end{array}
$$

The interpretations obtains the half-grid shown in Figure 2.22. Since this graph is known to have an undecidable MSO-theory, it follows that the graph generated by the CPDS $A_{grid}$ also has an undecidable MSO-theory. Hence, CPDSs provide a rare example of a structure with a decidable MSO-theory over trees, but an undecidable MSO-theory over graphs.

## 2.9 Global Model-Checking

Whilst a local model-checking algorithm determines whether a property is met from a designated initial state, global model-checking constructs the complete set of states satisfying a property. Global model-checking is useful when results are to be combined, and is the main focus of our research. In the remaining chapters we discuss *saturation*-based global model-checking. Saturation techniques represent a set of pushdown configurations using automata. Beginning with an initial automaton, transitions are added until a fixed point is reached.

In Chapter 3 we describe previous global model-checking results for reachability and Büchi specifications. Chapter 4 presents a new algorithm for the global model-checking problem over pushdown systems with parity conditions. This is the first contribution of the thesis. This chapter also describes several existing algorithms, and provides a brief comparison of the techniques.

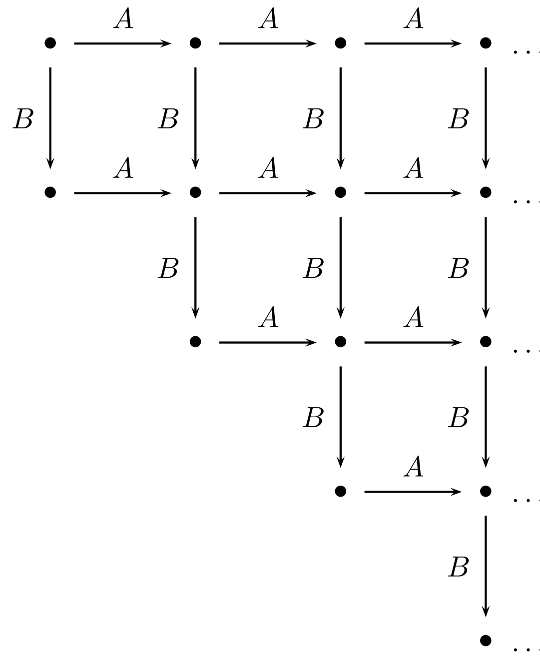



Figure 2.22: The infinite half-grid

In chapter 5 we present the main contribution of the thesis. This is a saturation-based global model-checking algorithm for reachability analysis of higher-order pushdown systems with an arbitrary number of control states. This result was presented in FoSSaCS 2007 [77]. We present a number of applications of this result in Chapter 6.

## 2.10 Summary

In this chapter we have given an overview of current research into formal methods for software verification. We have described the basic finite state framework and extensions to infinite state systems using the pushdown paradigm. We described order-1 pushdown systems and their related formalisms, as well as several kinds of model-checking technique. We then discussed the general case of higher-order pushdown systems and their connections to the Caucal hierarchy and recursion schemes. Finally we gave an account of collapsible pushdown systems: a further generalisation of pushdown systems.

# Chapter 3

# Saturation Methods for Global Model-Checking

That the set of configurations reachable by a pushdown system can be recognised by a finite automaton was first shown by Büchi [106]. In 1996/7, Bouajjani, Esparza and Maler introduced the saturation method for global model-checking pushdown systems with reachability constraints [3, 8]. This technique was also discovered independently by Finkel, Willems and Wolper [15] in 1997 and is based on a string-rewriting algorithm due to Book and Otto [118]. In 2000, Esparza *et al.* improved the efficiency of their algorithm [65]. These results formed the basis of Bouajjani and Meyer's algorithm for higher-order pushdown automata [2]. We give an account of the saturation results in Section 3.1 and Section 3.3 respectively. In Chapter 5 we extend these techniques to the case of higher-order pushdown systems with a finite number of control states. In Section 3.2 we describe Cachat's extension of the order-1 algorithm to Büchi games [127]. This algorithm is extended to the case of order-1 parity games in Chapter 4.

In Section 3.4 we describe an alternative notion of regularity introduced by Carayol [11].

## 3.1 Order-1 Reachability Analysis

The saturation approach to pushdown model-checking was introduced by Bouajjani, Esparza and Maler [3, 8] and independently by Finkel, Willems and Wolper [15]. This is a backwards-reachability algorithm that computes, given a set of configurations $C_{Init}$, the set of configurations $Pre^*(C_{Init})$ that can reach a configuration in $C_{Init}$ in a finite number of steps.

### 3.1.1 Multi-Automata

Possibly infinite sets of pushdown configurations can be represented using automata. Transitions are then added to the automaton representing reverse applications of the available pushdown commands. The automaton eventually becomes *saturated* and the algorithm terminates. The automata accept finite words, which represent order-1 stacks, over the finite alphabet $\Sigma$. We have an initial state for each control state of the pushdown automaton. A configuration $\langle p, [w] \rangle$ is accepted if, from the appropriate initial state, the word $w$ is accepted. Since the state-set and alphabet are fixed and finite, there is a bound on the number of transitions $q \xrightarrow{a} q'$. Hence, there is a limit on the number of transitions that the algorithm can

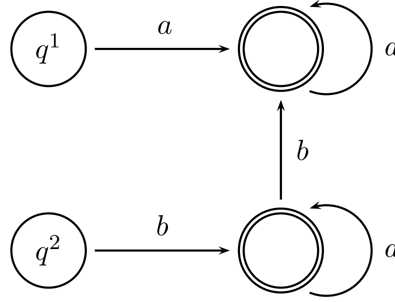



Figure 3.1: A multi-automaton $A_{eg}$ accepting $\langle p^1, [aa^*]\rangle, \langle p^2, [ba^*]\rangle$ and $\langle p^2, [ba^*ba^*]\rangle$

add.

**Definition 3.1.1.** Given an order-1 pushdown system $(\mathcal{P}, \mathcal{D}, \Sigma)$ with $\mathcal{P} = \{p^1, \ldots, p^z\}$, a **multi-automaton** $A$ is a tuple $(\mathcal{Q}, \Sigma, \Delta, I, \mathcal{F})$ where $\mathcal{Q}$ is a finite set of states, $\Delta \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ is a set of transitions, $I = \{q^1, \ldots, q^z\} \subseteq \mathcal{Q}$ is a set of initial states and $\mathcal{F} \subseteq \mathcal{Q}$ is a set of final states.

Let $q \xrightarrow{a} q'$ if $(q, a, q') \in \Delta$. Furthermore, for $a_0 \ldots a_h \in \Sigma^*$, let $q \xrightarrow{a_0 \ldots a_h} q'$ if there is a path,

$$q \xrightarrow{a_0} q_1 \xrightarrow{a_1} \cdots \xrightarrow{a_h} q'$$

A configuration $\langle p^j, [w]\rangle$ is accepted by $A$ iff we have a run $q^j \xrightarrow{w} q_f$ with $q_f \in \mathcal{F}$.

Figure 3.1 shows a multi-automaton $A_{eg}$ accepting configurations of the form $\langle p^1, [aa^*]\rangle$, $\langle p^2, [ba^*]\rangle$ and $\langle p^2, [ba^*ba^*]\rangle$.

### 3.1.2 Backwards Reachability

Given an order-1 pushdown system $(\mathcal{P}, \mathcal{D}, \Sigma)$ and a multi-automaton $A$, we can construct a multi-automaton $A_{Pre}$ such that $\mathcal{L}(\mathcal{A}_{Pre}) = Pre^*(\mathcal{L}(A))$. The algorithm proceeds by updating the automaton $A$ to obtain the set of configurations that can reach $\mathcal{A}$ after an application of a command $d \in \mathcal{D}$. We begin by presenting a naïve approach that will not terminate. This approach illustrates the principles guiding the addition of new transitions, although we do not fix the state-set.

Suppose $d = (p^1, a, pop_1, p^2)$. In this case, any configuration of the form $\langle p^1, [aw]\rangle$ where $w$ is accepted from $q^2$ in $A$ should be accepted by the updated automaton. This is because, by taking a $d$-transition, we will reach a configuration $\langle p^2, [w]\rangle \in \mathcal{L}(A)$. The automaton $A_{eg}$ updated in this way is presented in Figure 3.2.

If $d = (p^2, a, push_{ba}, p^2)$ then any configuration of the form $\langle p^2, [aw]\rangle$ where $baw$ is accepted from $q^2$ in $A$ can reach $A$ in one step. The push command replaces the top character $a$ with the word $ba$. In reverse, the word $ba$ is replaced by the character $a$. Therefore, the update adds a transition from $q^2$ which jumps over any run over $ba$ from $q^2$. This update is illustrated in Figure 3.3.

If we were to form the union of $A$ and the automata obtained by the two updates, we would have the set of configurations that can reach $\mathcal{L}(A)$ in one step or fewer. We could iterate this procedure, updating the new automaton, to produce the set of configurations that can reach $\mathcal{L}(A)$ in two steps or fewer, and so on to infinity.

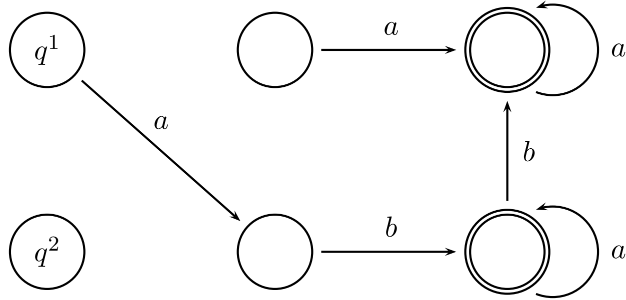



Figure 3.2: The automaton $A_{eg}$ updated by the command $(p^1, a, pop_1, p^2)$

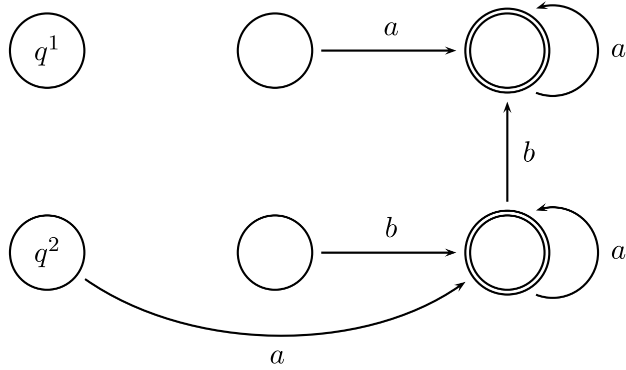



Figure 3.3: The automaton $A_{eg}$ updated by the command $(p^2, a, push_{ba}, p^2)$

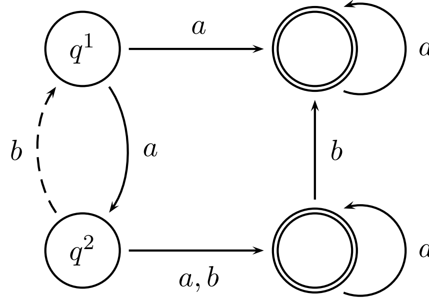



Figure 3.4: The updated automaton $A_{eg}$ without new states

### Termination

This procedure will not terminate: there is no bound on the number of steps required to meet $\mathcal{L}(A)$. To obtain a terminating algorithm we observe that we could have obtained the set of configurations that can reach $\mathcal{L}(A)$ in one step by refraining from adding new states to $A$. In the case of $A_{eg}$ the result is given by the transitions in Figure 3.4 that are not dashed.

Suppose we also had a command $(p^2, b, pop_1, p^1)$. This would result in the addition of the dashed arrow in Figure 3.4. The automaton no longer accepts configurations that reach $\mathcal{L}(A_{eg})$ in one step or fewer: a cycle has been introduced. The automaton now accepts configurations whose stacks begin with an alternating sequence of $a$ and $b$ characters.

These stacks, however, are not spurious. For example, given a configuration of the form $\langle p^1, [(ab)^* aa*]\rangle$, the cycle corresponds to repetition of the commands:

$$\langle p^1, [(ab)^* aa^*]\rangle \xrightarrow{(p^1,a,pop_1,p^2)} \langle p^2, [b(ab)^* aa^*]\rangle \xrightarrow{(p^2,b,pop_1,p^1)} \langle p^1, [(ab)^* aa^*]\rangle$$

Hence, we can apply the following sequence of commands to reach a configuration $\langle p^1, [aa^*]\rangle \in \mathcal{L}(A)$:

$$\langle p^1, [(ab)^* aa^*]\rangle \xrightarrow{(p^1,a,pop_1,p^2)} \cdots \xrightarrow{(p^2,b,pop_1,p^1)} \langle p^1, [aa^*]\rangle$$

Termination of the algorithm follows directly because we do not add new states, which leads to a finite bound on the number of transitions that can be added. A more illuminating explanation is that the introduction of cycles captures unbounded sequences of commands. Previously, an unbounded number of iterations of the algorithm would have been required.

### The Algorithm

We define the algorithm more formally. Firstly we define $T_d(A)$ for a command $d \in \mathcal{D}$ and a multi-automaton $A$. This operation performs a single update for a given command $d$. Recall $pop_1 = push_\varepsilon$.

**Definition 3.1.2.** Given a pushdown command $d = (q^j, a, push_w, q^k)$ and a multi-automaton $A = (\mathcal{Q}, \Sigma, \Delta, I, \mathcal{F})$, we define $T_d(A) = (\mathcal{Q}, \Sigma, \Delta', I, \mathcal{F})$ where,

$$\Delta' = \Delta \cup \{ (q^j, a, q) \mid q^k \xrightarrow{w} q \text{ in } A \}$$

Given a pushdown system $(\mathcal{P}, \mathcal{D}, \Sigma)$ and a multi-automaton $A$, let $\mathcal{A}_0 = \mathcal{A}$. For $\mathcal{D} = \{d_1, \ldots, d_h\}$, we define $A_{i+1} = T_{d_1}(\ldots T_{d_h}(A_i))$. There is a fixed point $A_f = A_{f+1}$ when the automaton becomes saturated. Define $A_{Pre} = A_f$.

**Theorem 3.1.1** ([3]). *Given a PDS $(\mathcal{P}, \mathcal{D}, \Sigma)$ and a regular set of configurations recognized by a multi-automaton $A$, we can construct an automaton $A_{Pre}$ recognizing $Pre^*(\mathcal{L}(\mathcal{A}))$.*

### 3.1.3 Extension to Alternating Pushdown Systems

Bouajjani, Esparza and Maler extend their algorithm to the case of alternating pushdown systems. This algorithm uses a generalisation of multi-automata: alternating multi-automata.

**Definition 3.1.3.** Given an order-1 pushdown system $(\mathcal{P}, \mathcal{D}, \Sigma)$ with $\mathcal{P} = \{p^1, \ldots, p^z\}$, an **alternating multi-automaton** $A$ is a tuple $(\mathcal{Q}, \Sigma, \Delta, I, \mathcal{F})$ where $\mathcal{Q}$ is a finite set of states, $\Delta \subseteq \mathcal{Q} \times \Sigma \times 2^{\mathcal{Q}}$ is a set of transitions, $I = \{q^1, \ldots, q^z\} \subseteq \mathcal{Q}$ is a set of initial states and $\mathcal{F} \subseteq \mathcal{Q}$ is a set of final states.

We define $q \xrightarrow{a} Q$ if $(q, a, Q) \in \Delta$. Furthermore, $q \xrightarrow{\varepsilon} \{q\}$ and $q \xrightarrow{aw} Q_1 \cup \cdots \cup Q_h$ if $q \xrightarrow{a} \{q_1, \ldots, q_h\}$ and $q_k \xrightarrow{w} Q_k$ for all $1 \leq k \leq h$. A configuration $\langle p^j, [w] \rangle$ is accepted by the automaton iff $q^j \xrightarrow{w} Q_f$ with $Q_f \subseteq \mathcal{F}$.

We are now ready to generalise the algorithm given above. We begin by extending the definition of $T_d(A)$ for a command $d \in \mathcal{D}$ and an alternating multi-automaton $A$. In the non-alternating case, a command $(p^j, a, push_w, p^k)$ added an $a$-transition from $p^j$ which jumped over a run on $w$ from $p^k$. This represents the backwards application of the $push_w$ to the stack: the top word $w$ is replaced by the character $a$. The same principle applies in the alternating case, except the alternation of the pushdown automaton is mimicked by the alternation of the multi-automaton.

**Definition 3.1.4.** Given a pushdown command $d = (q^j, a, OP)$ with,

$$OP = \{(push_{w_1}, p^{k_1}), \ldots, (push_{w_h}, p^{k_h})\}$$

and a multi-automaton $A = (\mathcal{Q}, \Sigma, \Delta, I, \mathcal{F})$, we define $T_d(A) = (\mathcal{Q}, \Sigma, \Delta', I, \mathcal{F})$ where,

$$\Delta' \ = \ \Delta \ \cup \ \{ \ (q^j, a, Q) \mid (1) \ \}$$

and (1) requires, for all $1 \leq i \leq h$, $q^{k_i} \xrightarrow{w_i} Q_i$ and $Q = Q_1 \cup \cdots \cup Q_h$.

Let $A_0 = \mathcal{A}$. For $\mathcal{D} = \{d_1, \ldots, d_h\}$, we define $A_{i+1} = T_{d_1}(\ldots T_{d_h}(\mathcal{A}_i))$. Let $A_{Pre}$ be the fixed point of the sequence generated.

**Theorem 3.1.2** ([3]). *Given an APDS $(\mathcal{P}, \mathcal{D}, \Sigma)$ and a regular set of configurations recognized by an alternating multi-automaton $A$, we can construct an automaton $A_{Pre}$ recognizing $Pre^*(\mathcal{L}(\mathcal{A}))$.*

## 3.2 Winning Regions of Order-1 Büchi Games

A Büchi game over a PDS is defined as follows:

**Definition 3.2.1.** Given an order-1 PDS $(\mathcal{P}, \mathcal{D}, \Sigma)$, an **order-1 Pushdown Büchi Game** (PBG) $(\mathcal{P}, \mathcal{D}, \Sigma, \mathcal{R})$ is given by a partition $\mathcal{P} = \mathcal{P}_A \uplus \mathcal{P}_E$ and a set $\mathcal{R}$ of fair configurations.

Play proceeds as in the case of pushdown reachability games. A play $c_0, c_1, c_2, \ldots$ is a win for Éloïse iff there exists $i \geq 0$ such that $c_i \in \mathcal{C}_A$ and Abelard is unable to move, or for all $i \geq 0$, there exists $j \geq i$ such that $c_j \in \mathcal{R}$ — that is, the set of fair configurations $\mathcal{R}$ is visited infinitely often.

Éloïse's winning region admits a fixed point characterisation. We first define $Attr^+_E(\mathcal{R})$. This is the set of configurations from which Éloïse can force play to reach $\mathcal{R}$ in at least one move.

$$\begin{array}{rcl} X_0(T) & = & \emptyset \\ X_{i+1}(T) & = & X_i(T) \cup \{ c \in \mathcal{C}_E \mid \exists c'.c \hookrightarrow c' \ \wedge \ c' \in T \cup X_i(T) \} \\ & & \cup \{ c \in \mathcal{C}_A \mid \forall c'.c \hookrightarrow c' \Rightarrow c' \in T \cup X_i(T) \} \\ Attr^+_E(T) & = & \bigcup_{i \geq 0} X_i(T) \end{array}$$

Éloïse's winning region $Büchi_E(\mathcal{R})$ can be defined as follows:

$$\begin{array}{rcll} Büchi^0(\mathcal{R}) & = & C^{\Sigma}_n & \\ Büchi^{\alpha+1}_E(\mathcal{R}) & = & Attr^+_E(Büchi^{\alpha}_E(\mathcal{R}) \cap \mathcal{R}) & \text{for any ordinal } \alpha \\ Büchi^{\lambda}_E(\mathcal{R}) & = & \bigcap_{\alpha < \lambda} Büchi^{\alpha}_E(\mathcal{R}) & \text{for a limit ordinal } \lambda \end{array}$$

There exists an ordinal $\alpha$ such that $Büchi^{\alpha}_E(\mathcal{R}) = Büchi^{\alpha+1}_E(\mathcal{R})$. We write $Büchi_E(\mathcal{R})$ to denote this fixed point. $Büchi_E(\mathcal{R})$ is the winning region for Éloïse.

The current definition of pushdown Büchi games permits $\mathcal{R}$ to be any regular set of configurations. In fact, we can restrict our attention to the case of *simple* sets of the form $\mathcal{F} \times C^{\Sigma}_1$ where $\mathcal{F} \subseteq \mathcal{P}$. The reduction is similar to the reductions used to test regular stack properties in Section 2.7.1.

**Proposition 3.2.1** ([127]). *Given a PBG $G = (\mathcal{P}, \mathcal{D}, \Sigma, \mathcal{R})$ with a regular set of configurations $\mathcal{R}$, we can reduce to the case of simple goal sets by proceeding to a new PBG $G \times A_{\mathcal{R}}$ where $A_{\mathcal{R}}$ is a finite automaton recognising $\mathcal{R}$.*

This result is obtained by taking $A_{\mathcal{R}}$ to be a finite deterministic automaton that accepts stacks from bottom to top. In the product game, this automaton can keep track of the current stack contents (using the stack to store backtracking information to aid in the handling of $pop_1$ operations). The simple set $\mathcal{R}'$ is the set of configurations containing an accepting state of $A_{\mathcal{R}}$.

In the sequel, we assume that all sets $\mathcal{R}$ are simple.

### 3.2.1 The Naïve Algorithm

To compute Éloïse's winning region, we need to calculate the greatest fixed point of,

$$X \mapsto Attr^+_E(X \cap \mathcal{R})$$

Hence, we begin with an automaton representing the set of all configurations. This automaton is shown in Figure 3.5. The greatest fixed point is the result of repeated intersections with the set $\mathcal{R}$ and applications of a reachability algorithm. This process is shown in Figures 3.6 to 3.8. The transitions added during the reachability step (Figure 3.7) are for illustrative purposes, and do not correspond to a particular Büchi game.

The algorithm begins by intersecting the current automaton with $\mathcal{R}$. This is done by adding a new set of initial states which have $\varepsilon$-transitions to the corresponding previous

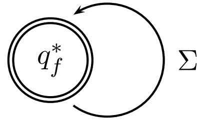



Figure 3.5: The automaton accepting $C_1^{\Sigma}$

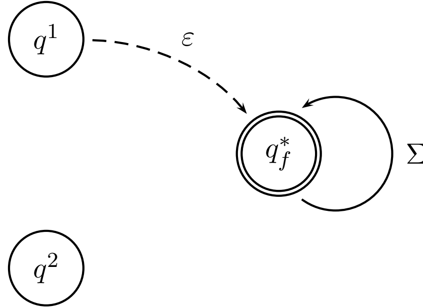



Figure 3.6: Intersecting with $\mathcal{R}$ using temporary $\varepsilon$-transitions

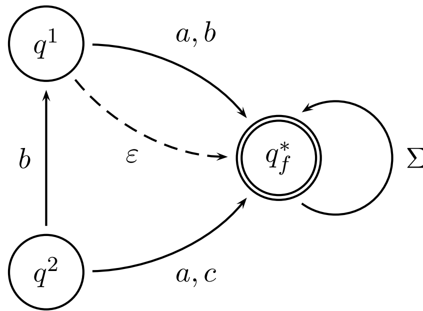



Figure 3.7: Computing the attractor using the reachability algorithm

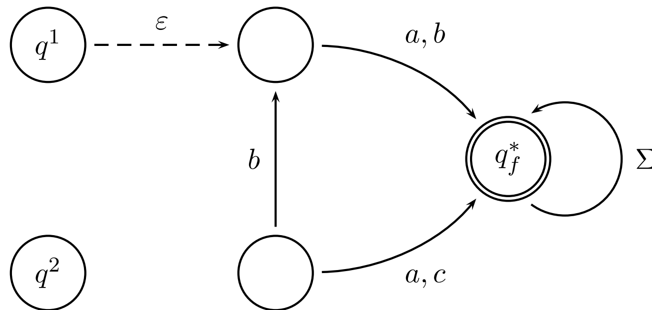



Figure 3.8: Erasing the temporary transitions and repeating the iteration

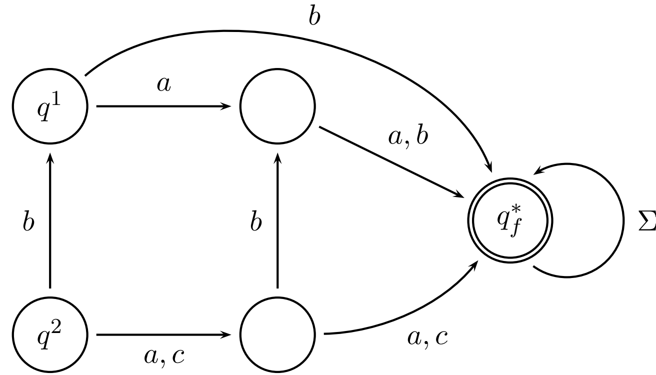



Figure 3.9: An automaton before pulling back the new transitions

initial state. These transitions are only added if the control state belongs to the set $\mathcal{R}$. This is shown in Figure 3.6. We then perform reachability analysis on the automaton obtained, leading to Figure 3.7.

So far, we have constructed $Attr^*_E(X \cap \mathcal{R})$. By removing the $\varepsilon$-transitions added initially we ensure that all paths to $q_f^*$ contain at least one transition added during the reachability analysis. Since these transitions are derived from some command $d$, the set represented is the set of configurations that can reach $X \cap \mathcal{R}$ in one step or more. That is $Attr^+_E(X \cap \mathcal{R})$. The algorithm is then iterated, as shown in Figure 3.8.

### 3.2.2 Termination

The algorithm in the previous section computes the greatest fixed point characterising Éloïse's winning region in a PBG. However, in general, we require a transfinite number of iterations to reach this fixed point. (An example is given in Section 3.4.4 of Cachat's thesis [127]). Intuitively, this follows from the unbounded state-set, which does not allow the transitions to become repetitive. By introducing a simple, although unintuitive, speed-up technique, Cachat is able to provide a terminating algorithm.

After each iteration of the algorithm, let $I'$ be the new set of initial states and $I$ be the previous set. After the first iteration, we *pull back* any transitions from $I'$ to $I$, replacing them with transitions from $I'$ to $I'$. Figure 3.9 and Figure 3.10 illustrate this process for an imagined continuation of the example in the previous section (in this example, only a pull back after the second iteration is shown. In the full algorithm, a pull back is also performed after the first iteration). Pulling back the new transitions has the effect of introducing cycles and making $I$ unreachable. Therefore, $I$ can be removed, and thus, the state-set is, in some sense, constant. By proving that the number of transitions between successive generations of initial states is decreasing, it follows that a fixed point will be reached in a finite number of steps.

### 3.2.3 The Algorithm

We present Cachat's algorithm formally. The algorithm proceeds via a number of iterations. At the beginning of each iteration, a new generation of initial states is introduced. Hence, we use a subscript $i$ to indicate the generation a state belongs to. That is, a control-state $p^j$

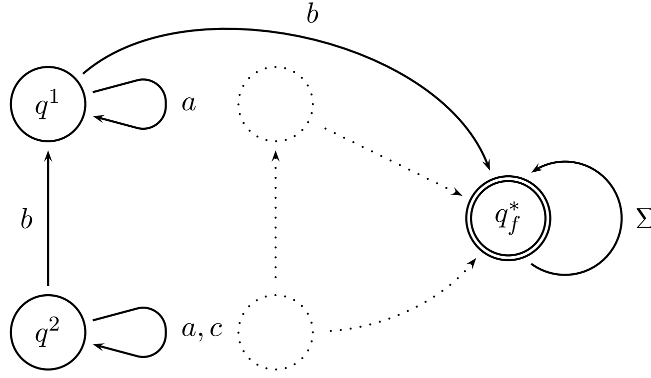



Figure 3.10: An automaton after pulling back the new transitions

initially corresponds to an initial state $q^j_0$. At the $i$th iteration, the control state corresponds to the state $q^j_i$. Our initial automaton contains a single state $q^*_f$ which is accepting. We define $q^j_0 = q^*_f$ for all $j$. We need two operations on sets of states:

$$\phi(Q) = \{ q^j_i \mid q^j_{i+1} \in Q \} \cup \{ q^*_f \mid q^*_f \in Q \}$$

$$\pi_i(Q) = \{ q^j_i \mid q^j_k \in Q \text{ for } 0 > k \leq i \} \cup \{ q^*_f \mid q^*_f \in Q \}$$

Given a PBG $G = (\mathcal{P}, \mathcal{D}, \Sigma, \mathcal{R})$ with $\mathcal{R} = \mathcal{F} \times C_1^\Sigma$ for some $\mathcal{F} \subseteq \mathcal{P}$, we construct an automaton $A_B$ accepting $B\ddot{u}chi_E(\mathcal{R})$. This automaton is a fixed point of a sequence $(A^i_B)_{i\geq 0}$ with $A^0_B$ defined as follows. The state-space of $A^0_B$ is a subset of $\{ q^1_i, \ldots, q^m_i \mid i > 0 \} \cup \{q^*_f\}$ where $\mathcal{P} = \{p^1, \ldots, p^m\}$. The transition relation is $\{ (q^*_f, a, q^*_f) \mid a \in \Sigma \}$. We define $A^{i+1}_B = T^B_\mathcal{D}(A^i_B)$ where $T^B_\mathcal{D}$ is defined:

**Definition 3.2.2.** Given a multi-automaton $A^i_B$ we construct $A^{i+1}_B = T^B_\mathcal{D}(A^i_B)$ in several stages:

- Add an $\varepsilon$-transition from $q^j_{i+1}$ to $q^j_i$ for all $p^j \in \mathcal{F}$.
- Perform reachability analysis as in Section 3.1.3.
- Remove the $\varepsilon$-transitions.
- Replace each transition $q^j_{i+1} \xrightarrow{a} Q$ with $q^j_{i+1} \xrightarrow{a} \pi_{i+1}(Q)$.

The algorithm terminates when,

$$q^j_{i+1} \xrightarrow{a} Q \iff q^j_i \xrightarrow{a} \phi(Q)$$

At this point, we set $A_B = A^{i+1}_B$.

Notice that the above algorithm doesn't, on first appearances, take into account whether a control state belongs to Éloïse or Abelard. This is because the manipulations performed in the above definition are insensitive to the ownership of the control state. However, the algorithm requires a call to the reachability algorithm of Section 3.1.3 for alternating pushdown systems and it is this algorithm that is takes into account the owner of the control states.

**Theorem 3.2.1** ([127]). *The automaton $A_B$ recognises $B\ddot{u}chi_E(\mathcal{R})$ — Éloïse's winning region of the PBG $G = (\mathcal{P}, \mathcal{D}, \Sigma, \mathcal{R})$.*

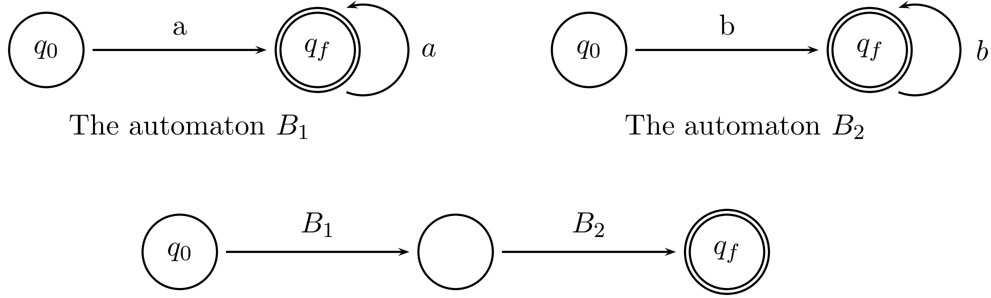



Figure 3.11: A level 2 nested store automaton $A_{ab}$ accepting $\langle q_0, [[a^+][b^+]]\rangle$

## 3.3 Context-Free Higher-Order Pushdown Systems

Global reachability analysis of higher-order pushdown systems with a single control state $p$ was considered by Bouajjani and Meyer in 2004 [2]. Their approach applies similar techniques to those described above. In the sequel, we assume $n > 1$.

### 3.3.1 Nested Store Automata

The key difference between the two methods is the representation of sets of configurations. Bouajjani and Meyer introduce *nested* store automata to recognise higher-order stores. Transitions of a level $n$ automaton are labelled with level $(n-1)$ store automata. Level 1 automata are finite word automata. The nesting reflects the nested structure of a higher-order store.

**Definition 3.3.1** ([2])**.** A **level** 1 **nested store automaton** is a finite automaton whose transitions have labels in $\Sigma$. A *nested store automaton* of level $n \geq 2$ is a finite automaton whose transitions are labelled by level $(n-1)$ nested automata over $\Sigma$.

A transition labelled by an automaton $B$ is denoted $q \xrightarrow{B} q'$. The language accepted by a level $n$ nested store automaton $A = (\mathcal{Q}, \Sigma, \Delta, q_0, q_f)$ where $q_f$ is the designated final state, is defined recursively,

$$\mathcal{L}(A) = \{\ [\mathcal{L}(A_1) \ldots \mathcal{L}(\mathcal{A}_l)] \mid q_0 \xrightarrow{A_1} \ldots \xrightarrow{A_l} q_f\ \}$$

**Proposition 3.3.1** ([2])**.** *The store languages accepted by nested store automata are the* **regular store languages***.*

Figure 3.11 shows an order-2 nested store automaton $A_{ab}$ accepting configurations of the form $\langle q_0, [[a^+][b^+]]\rangle$.

### 3.3.2 Backwards Reachability

Given an order-$n$ pushdown system with a single control state, and a level $n$ nested store automaton $A$, we can compute the set $Pre^*(A)$ using the following algorithm. Since all commands are of the form $(p, a, o, p)$ we omit the control state and write $(a, o)$ where $o$ is $push_w$, $push_l$ or $pop_l$ for $2 \leq l \leq n$ and $w \in \Sigma^*$.

The algorithm follows similar principles to the order-1 case: transitions are added to reflect a reverse application of a command $d$. When $A$ is a level $n$ nested automaton and $o$

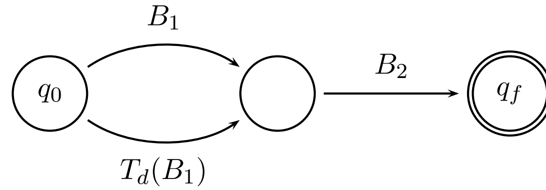



Figure 3.12: $T_d(A_{ab})$ for $d = (a, o)$ where $o$ is not $push_n$ or $pop_n$

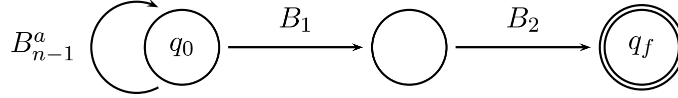



Figure 3.13: $T_d(A_{ab})$ for $d = (a, pop_1)$

is $push_w$, $push_l$ or $pop_l$ for $l < n$, then, for each initial transition $q_0 \xrightarrow{B} q$ of $A$, we introduce a new transition $q_0 \xrightarrow{B'} q$ where $B'$ is the automaton $B$ updated to reflect the operation $o$. Just as $o(\gamma w)$ for an order-$n$ store $\gamma w$ is $o(\gamma)w$, the update is passed to the automata recognising the $top_n$ stack of the store. A $push_w$ command is consequently passed to a level 1 store automaton, which is updated in the same way a $push_w$ command was handled in the order-1 case. This update is illustrated in Figure 3.12.

The processing of a $pop_n$ command is analogous to the treatment of a $pop_1$ command. A command $(a, pop_n)$ removes the $top_n$ stack provided its $top_1$ character is $a$. Hence, when applied in reverse, the command will add a new order-$(n-1)$ stack whose $top_1$ character is $a$. Let $B^a_{n-1}$ be an $(n-1)$-store automaton accepting all order-$(n-1)$ stacks with $top_1$ element $a$. A $(a, pop_n)$ update adds the transition $q_0 \xrightarrow{B^a_{n-1}} q_0$ — corresponding to an arbitrary number of applications of the command. Figure 3.13 shows the automaton $A_{ab}$ after updating for a command $(a, pop_n)$.

Finally, an $(a, push_n)$ command duplicates the $top_n$ stack, provided the $top_1$ character is $a$. When applied backwards, the two $top_n$ stacks must be identical. These two stacks are replaced by a single stack which is also identical. Hence, for every run $q_0 \xrightarrow{B_1} q' \xrightarrow{B_2} q$ we add a transition $q_0 \xrightarrow{B} q$ where $B = B^a_{n-1} \times B_1 \times B_2$. Any stack accepted by this transition must also be able to appear as the two $top_n$ stacks, and have an $a$ as its $top_1$ character. This is shown in Figure 3.14.

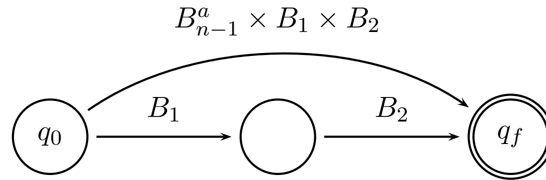



Figure 3.14: $T_d(A_{ab})$ for $d = (a, push_n)$

### The Algorithm

As in the previous algorithms, we define $T_d$, which updates an automaton to reflect a reverse application of a command $d$. We iterate this update until a fixed point is reached.

**Definition 3.3.2.** Given an order-$n$ pushdown command $d = (a, o)$ and a level $n$ nested store automaton $A = (\mathcal{Q}, \Sigma, \Delta, q_0, q_f)$, we define $T_d(A) = (\mathcal{Q}, \Sigma, \Delta', q_0, q_f)$ where,

$$\Delta' \;=\; \Delta \;\cup\; \{\; (q_0, a, q) \mid q_0 \xrightarrow{w} q \text{ in } A \;\}$$

when $n = 1$ and $o = push_w$. Otherwise $n > 1$ and we have,

$$\Delta' \;=\; \Delta \;\cup\; \{\; (q_0, B, q) \mid (1) \;\}$$

where (1) requires,

- If $o = pop_n$, then $q = q_0$ and $B = B^a_{n-1}$.
- If $o = push_n$, then
$$q_0 \xrightarrow{B_1} q' \xrightarrow{B_2} q$$
is a path in $A$ and $B = B^a_{n-1} \times B_1 \times B_2$.
- Otherwise, $q_0 \xrightarrow{B_1} q$ is a path in $A$ and $B = T_d(B_1)$.

Given a higher-order pushdown system $(\{p\}, \mathcal{D}, \Sigma)$ and a multi-automaton $A$, let $\mathcal{A}_0 = \mathcal{A}$. For $\mathcal{D} = \{d_1, \ldots, d_h\}$, we define $A_{i+1} = T_{d_1}(\ldots T_{d_h}(A_i))$. Let $A_{Pre}$ be the fixed point of this sequence.

**Theorem 3.3.1** ([2])**.** *Given an order-$n$ higher-order pushdown system with a single control state, and a regular set of $n$-stores $S$, the set $Pre^*(S)$ is regular and effectively computable.*

Termination in this case is more subtle. Since the number of level $(n-1)$ store automata is unbounded, we need to argue that the nature of the algorithm limits the automata that may label an edge of $A_i$ for some $i \geq 0$.

Each update may entail the addition of a new automaton. This automaton will be the product of several existing automata. Hence, the state-set of all level $k$ automata can be considered a sub-set of the product $\mathcal{Q}_1 \times \ldots \times \mathcal{Q}_l$ where $A_1, \ldots, \mathcal{A}_l$ are the level $k$ automata appearing in $A$ (or of the form $B^a_k$) and $\mathcal{Q}_1, \ldots, \mathcal{Q}_l$ are their respective state-sets. Hence, the state-set is bounded.

Since $\Sigma$ is finite, we immediately obtain a bound on the level 1 nested store automata that may occur. This entails a bound on the level 2 nested store automata. Hence, by an inductive argument, the number of level $(n-1)$ automata is bounded and the algorithm must terminate.

**Lemma 3.3.1** ([2])**.** *For all level $n$ nested store automata $A$ and higher-order pushdown systems $(\{p\}, \mathcal{D}, \Sigma)$, the sequence $(A_i)_{i\geq 1}$ defined with respect to $A$ eventually stabilizes: $\exists k \geq 0. \forall k' \geq k. A_k = \mathcal{A}_{k'}$, which implies $\mathcal{L}(A_k) = \bigcup_{i\geq 0} \mathcal{L}(\mathcal{A}_i)$.*

## 3.4 Carayol Regularity

The algorithms in the previous sections demonstrate that the set $Pre^*(S)$ for a regular set of configurations $S$ is itself regular. It is well know that the set $Post^*(S)$ is also regular in the order-1 case [65]. However, this property does not hold for higher-order stacks.

**Proposition 3.4.1** ([2])**.** *Given an order-n pushdown automaton and a regular set of n-stores $S$, the set $Post^*(S)$ is in general not regular. This set is a context-sensitive language.*

For example, consider the following context-free order-2 PDS:

$$(\bot, push_a) \quad (a, push_a) \quad (a, push_b) \quad (b, push_2)$$

From an initial configuration $\langle p, [[\bot]] \rangle$, the automaton can push an arbitrary number of $a$ characters onto the stack. It can then push a $b$ character. At this point, the only operation available is $push_2$. Hence, the set of reachable configurations is the set containing $\langle p, [[a^*\bot]] \rangle$ and $\langle p, [[ba^m\bot]^+] \rangle$ where $m \geq 1$. An automaton recognising the set would need to be able to count an arbitrary number of $a$ characters to ensure $m$ characters occur in each order-1 stack. The set is not regular.

Motivated by this problem, Carayol introduced an alternative notion of regularity [11]. Rather than representing the characters and structure of the stack directly, a store is represented by the sequence of stack operations required to construct it. In the example above, the set of configurations can be represented in a straightforward manner:

$$(push_\bot; (push_a)^*) \;\cup\; \left(push_\bot; (push_a)^+; push_b; (push_2)^*\right)$$

Carayol permits a slightly different set of pushdown operations.

$$\begin{array}{rcl} push_a[a_1 \dots a_m]_1 & = & [aa_1 \dots a_m]_1 \\ pop_a[aa_1 \dots a_m]_1 & = & [a_1 \dots a_m]_1 \\ copy_k[\gamma_1 \dots \gamma_m]_{k+1} & = & [\gamma_1\gamma_1 \dots \gamma_m]_{k+1} \\ \overline{copy}_k[\gamma_1\gamma_1 \dots \gamma_m]_{k+1} & = & [\gamma_1 \dots \gamma_m]_{k+1} \end{array}$$

and the test $E_k$, which asserts that the topmost $k$ stack is empty. This test replaces the $\bot$ character. These operations are more symmetrical than the standard *push* and *pop* operations, but they do not affect expressive power. Let $\mathcal{O}_1^C = \{\ push_a, pop_a \mid a \in \Sigma\ \}$ and $\mathcal{O}_{n+1}^C = \mathcal{O}_n^C \cup \{E_n, copy_n, \overline{copy}_n\}$.

**Definition 3.4.1.** A set of order-$n$ stacks is **C-regular** iff it can be represented by a regular language $L \subseteq (\mathcal{O}_n^C)^*$ applied to the empty stack $[\varepsilon]_n$. We write $Reg(S)$ to denote the regular subsets of $S^*$ for a set $S$.

### 3.4.1 Normal Form

C-regular expressions have a normal form. In the order-1 case, the normal form can be considered *loop-free.* For example $(push_a; push_b; pop_b; push_c)$ contains a loop since the $pop_b$ negates the $push_b$ operation, resulting in the stack obtained after the $push_a$ . The sequence in normal form is $(push_a; push_c)$. For order-1, a result due to Büchi states that all minimal sequences of a C-regular language are also regular [106]. Hence, all order-1 C-regular expressions have a normal form.

In the higher-order case, sequences can not be loop-free. This is because we may use a $copy_k$ operation to perform a test on the contents of the top most $k$ stack. If this test is passed, a $\overline{copy}_k$ is performed and evaluation continues from the stack constructed before the test began. For example, a test may take the form $copy_k; L; \overline{copy}_k$ where $L$ is a regular subset of $\mathcal{O}^C_{k-1}$. For a C-regular language $L$, a test may be interpreted as an order-$k$ $id$ operation, restricted to the set $\mathcal{L}(L)$, denoted $id_k|_L$.

**Definition 3.4.2.** The set $Norm_{k+1}$ of C-regular expressions in normal form is the set,

$$Norm_{k+1} = Norm_k; Reg((copy_k; Rewrite_k)^*)$$

where $Rewrite_k$ is the set of finite unions of $U; T_L; V$ where $T_L$ is a test with $L \in Norm_k$, interpreted as $id_k|_L$, $U \in Pop_k$ and $V \in Push_k$ with,

$$\begin{array}{rcl} Last(U) \cap \overline{First(V)} & = & \emptyset \\ \overline{Last(L)} \cap \left(\overline{Last(U)} \cup First(V)\right) & = & \emptyset \end{array}$$

Finally,

$$\begin{array}{rcl} Pop_{k+1} & = & Rewrite_k; Reg((\overline{copy}_k; Rewrite_k)^*) \\ Push_{k+1} & = & Rewrite_k; Reg((copy_k; Rewrite_k)^*) \end{array}$$

An order-$(k+1)$ expression in normal form begins by constructing an order-$k$ stack. The full stack is then constructed by repeatedly performing a $copy_k$ operation to add a new $k$ stack, and then rewriting the stack to the desired contents. Rewriting a $k$ stack is done in three stages: elements of the stack are removed, a test may be performed on the stack contents, and then elements are added to the stack to complete the rewriting. The conditions on $U, V$ and $L$ ensure that the three stages can be combined.

Carayol introduces $k$-automata, which are automata labelled by $\mathcal{O}^C_k$ subject to a number of constraints. These automata provide an alternative characterisation of normal form.

**Theorem 3.4.1** ([11])**.** *Every $k$-regular set can be accepted by a $k$-automaton. Hence, we have that $\mathcal{L}(Reg(\mathcal{O}^C_k)) = \mathcal{L}(Reg(Norm_k))$ is an effective boolean algebra.*

### 3.4.2 MSO-Definability

Carayol's notion of regularity captures MSO-definability over higher-order stacks. The tree $\Delta^k_2$ is the canonical structure associated with order-$k$ stacks. Figure 3.15 shows the order-2 case.

**Proposition 3.4.2** ([11])**.** *The set of order $k$ stacks MSO-definable in $\Delta^k_2$ are the C-regular sets, and the sets of relations MSO-definable in $\Delta^k_2$ are the relations in $Rewrite_k$.*

In contrast, we will see in Chapter 4 that the notion of regularity introduced by Bouajjani and Meyer (in Section 3.3) capture the $\mu$-calculus-definable sets.

## 3.5 Summary

In this chapter we described saturation methods for reachability analysis of subclasses of higher-order pushdown automata. These techniques were introduced by Bouajjani, Esparza

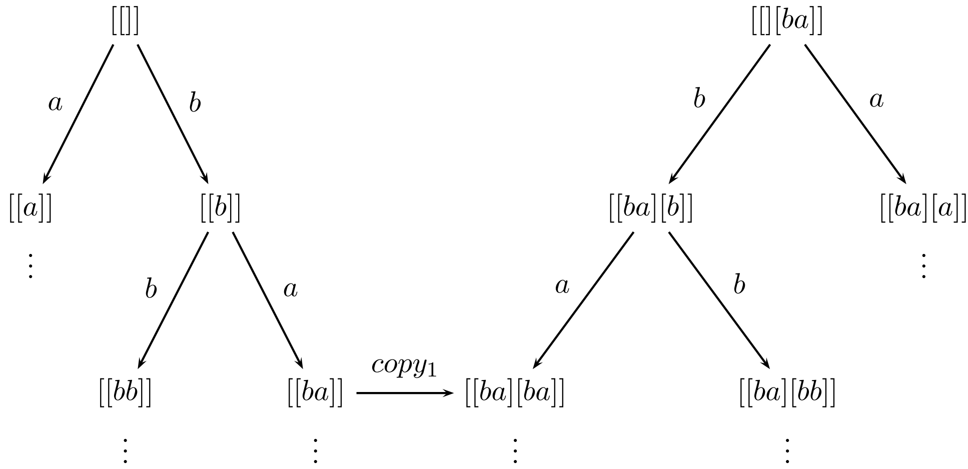



Figure 3.15: The canonical structure $\Delta_2^2$ associated with order-2 stacks

and Maler for pushdown automata [3, 8] and by Finkel, Willems and Wolper [15]. The algorithm begins with an automaton representing a set of configuration and adds new transitions corresponding the pushdown commands until saturation.

These techniques were adapted to the case of higher-order pushdown systems with a single control state by Bouajjani and Meyer [2]. The main innovation of this approach is a form of nested store automata for representing sets of configurations. The transitions of a nested store automata are automata themselves. This nesting reflects the structure of higher-order stores. Similar saturation techniques can be applied to the store automata to compute the set of configurations that can reach the set defined by the initial automaton.

Finally we discussed the limitations of the notion of regularity used by Bouajjani and Meyer. In particular, the set $Post(S)$ of a regular set of configurations $S$ is not regular. Carayol's notion of regularity [11] addresses this problem. Sets of configurations are denoted by the stack operations required to construct them. Whilst Bouajjani and Meyer's notion of regularity captures $\mu$-calculus definability, C-regularity captures MSO-definability.

# Chapter 4

# Order-1 Pushdown Parity Games

In this chapter we propose a new algorithm for computing the winning region (i.e. Éloïse's) of a pushdown parity game. Our technique is a generalisation of Cachat's saturation technique — described in Section 3.2 — for solving Büchi games [127], which is itself a generalisation of the original saturation method for reachability analysis due to Bouajjani *et al.* However, we believe our proofs to be cleaner and more complete than Cachat's Büchi games proof.

Using a modal $\mu$-calculus formula (that characterises the winning region) as a guide, we iteratively expand or contract an initial alternating multi-automaton until we have constructed an automaton that recognises precisely the winning region.

As illustrated in Section 4.2, we think our iterative algorithm is relatively simple to describe; it follows, in outline, the standard pen-and-paper approach to evaluating modal $\mu$-calculus formulae (though an important difference here is that the evaluation is over an *infinite* state-transition graph). The main conceptual contribution of our work lies in the identification and use of the *valuation soundness* and *valuation completeness* conditions, which play a pivotal role in the correctness proofs of the algorithm.

In the remainder of the chapter we describe several existing approaches — due, respectively, to Vardi *et al.* [90, 86, 85], Cachat [127] and Serre [92] — to computing the winning regions of an order-1 pushdown parity game. In Section 4.6.4 we compare the approaches with our own. Finally, in Section 4.7 we describe Serre's generalisation of his order-1 technique to pushdown systems of arbitrary order [12]. The main advantages of our approach are that it is simple, direct, and permits some simple optimisations that prevent the algorithm from being immediately exponential.

The algorithm presented computes the winning regions of a parity game by evaluating a modal $\mu$-calculus formula. We believe it is possible to extend this approach the calculate the set of states satisfying an arbitrary modal $\mu$-calculus formula. However, such an extension will require a much more careful management of the introduction of new states and projections. Hence, it is left for future work. Bouajjani *et al.* give an algorithm for computing the set of configurations satisfying an alternation-free modal $\mu$-calculus formula [3]. An order-$n$ generalisation of this algorithm is described in Section 6.3.

## 4.1 Preliminary Definitions

We begin by recalling the definition of order-1 pushdown parity games. A pushdown parity game is played between two opponents, Abelard and Éloïse. The control states of a push-

down system are divided between the two players. A play begins from some configuration $\langle p, [aw] \rangle$. The player controlling $p$ chooses a command $(p, a, push_{w'}, p') \in \mathcal{D}$ and play moves to $\langle p', [w'w] \rangle$. Then, the player controlling $p'$ is required to make a move, and so on.

As is standard, we assume a bottom-of-stack symbol $\bot \in \Sigma$ that is neither pushed onto, nor popped from, the stack. Similarly, we can assume that there exists an available move for all $p \in \mathcal{P}$ and $a \in \Sigma$. Consequently, a move is always possible from any given configuration and plays are infinite sequences of configurations.

To determine the winner of the game, each control state is given a priority from a set $\{1, \ldots, m\}$. The priority of a configuration is the priority of its control state. A priority occurs infinitely often in a play if there are an infinite number of configurations with that priority. Éloïse wins the game if the smallest priority occurring infinitely often is even. Otherwise, Abelard is the winner.

**Definition 4.1.1.** An ***order*-1 *pushdown parity game*** is a tuple $(\mathcal{P}, \mathcal{D}, \Sigma, \Omega)$ where $\mathcal{P} = \mathcal{P}_A \uplus \mathcal{P}_E$ is a set of control states partitioned into states belonging to Abelard and states belonging to Éloïse, $\Sigma$ is a finite alphabet, $\mathcal{D}$ is a set of pushdown commands and $\Omega : \mathcal{P} \rightarrow \{1, \ldots, m\}$ is a function assigning priorities to control states.

The winning region of a pushdown parity game for a given player is the set of all configurations from which that player can always win the game, regardless of the strategy employed by their opponent.

We can encode Éloïse's winning region of a game $\mathcal{G}$ using the modal $\mu$-calculus. The following formulation is presented by Walukiewicz [53]:

$$\mathcal{W}_E = [\![\mu Z_1.\nu Z_2.\ldots\mu Z_{m-1}.\nu Z_m.\varphi_E(Z_1, \ldots, Z_m)]\!]^{\mathcal{G}}_V$$

where $m$ is the maximum parity (assumed even), $V$ is a valuation, and

$$\varphi_E(Z_1, \ldots, Z_m) = \left( E \Rightarrow \bigwedge_{c \in \{1,\ldots,m\}} (c \Rightarrow \Diamond Z_c) \right) \wedge \left( \neg E \Rightarrow \bigwedge_{c \in \{1,\ldots,m\}} (c \Rightarrow \Box Z_c) \right)$$

where $E$ is an atomic proposition asserting that the current configuration belongs to Éloïse and, for each $1 \leq c \leq m$, $c$ is an atomic proposition asserting the priority of the current control state is $c$.

Each variable $Z_c$ corresponds to a priority $c$. The odd priorities are bounded by $\mu$ operators which can be intuitively understood as "finite looping". Dually, even priorities are bounded by $\nu$ operators, which can be understood as "infinite looping". The formula $\phi_E$ intuitively asserts that a variable $Z_c$ is visited whenever a configuration of priority $c$ is encountered. Thus the full formula asserts that the minimal priority occurring infinitely often must be even — otherwise a variable bound by the $\mu$ operator would be passed through infinitely often. It can be shown by a standard signature lemma that Éloïse has a winning strategy from a configuration satisfying the formula [53]. Furthermore, since the formula's inverse can be shown to be a similar formula with $\mu$ and $\nu$, and $\Box$ and $\Diamond$ reversed, it is easy to show that Abelard has a winning strategy from any configuration not in $\mathcal{W}_E$.

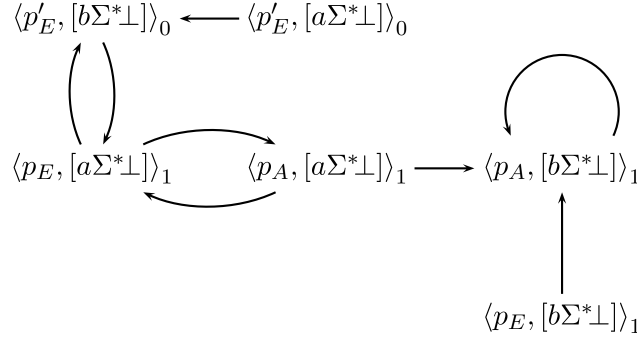



Figure 4.1: An example pushdown parity game.

## 4.2 An Example

We begin with an intuitive explanation of the algorithm by means of an example. Consider the pushdown game shown in Figure 4.1. The subscripts indicate the priority of a configuration[1]. Let $p_E, p'_E \in \mathcal{P}_E$ and $p_A \in \mathcal{P}_A$.

Éloïse can win from configurations of the form $\langle p'_E, [a\Sigma^*\bot]\rangle_0$, $\langle p_E, [a\Sigma^*\bot]\rangle_1$ or $\langle p'_E, [b\Sigma^*\bot]\rangle_0$. Éloïse can loop between the last two of these configurations, generating a run with lowest infinitely-occurring priority 0. From any other configuration, Abelard can force play to a configuration of the form $\langle p_A, [b\Sigma^*\bot]\rangle_1$ and generate a run with lowest infinitely-occurring priority 1. Computing Éloïse's winning region is equivalent to computing the set of configurations satisfying $\nu Z_0.\mu Z_1.\varphi_E(Z_0, Z_1)$. We illustrate how this is done in the following.

To compute a greatest fixed point, we begin by setting $Z_0$ to be the set of all configurations. We then calculate the (automaton that recognises the) configuration-set denotation of $\mu Z_1.\varphi_E(Z_0, Z_1)$ with this value of $Z_0$. The result is the value of $Z_0$ for the next iteration. After each iteration the value of $Z_0$ will be a subset of the previous value of $Z_0$. This computation reaches a limit when the value of $Z_0$ stabilises i.e. it remains the same from one iteration to the next. This fixed point is the denotation of the formula.

Computing the least fixed point proceeds in a similar manner, except that the initial value of $Z_1$ is set to $\emptyset$. We then compute the (automaton that recognises the) denotation of $\varphi_E(Z_0, Z_1)$, which gives us the next value of $Z_1$. Dual to the case of greatest fixed points, the value of $Z_1$ increases with each iteration.

### Constructing the Automaton

(*In the following, we shall often confuse the denotation of a formula with the automaton that recognises it, leaving it to the context to indicate which is intended.*) We begin by setting $Z_0$ to the set of all configurations. The alternating multi-automaton recognising all configurations is shown in Figure 4.2[2] (assuming, for clarity, the bottom-of-stack symbol $\bot$ is guaranteed to appear only at the bottom of the stack). Given this value of $Z_0$, we now compute the

[1] In this game our priorities begin at 0. This is for convenience and does not change the algorithm significantly.

[2] This is a simplification of the automaton used in the full algorithm. In particular, there is no alternation present. This example was deliberately designed to avoid the use of alternation due to the difficulties of cleanly drawing such automata.

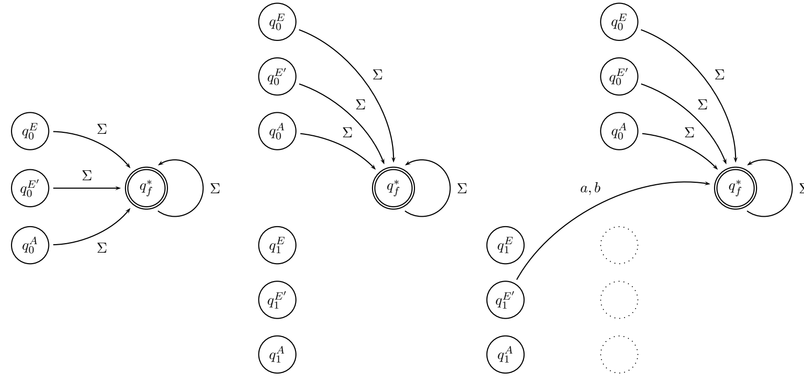



Figure 4.2: From left to right, the automaton accepting the initial value of $Z_0$; the automaton accepting the initial values of $Z_0$ and $Z_1$; and the automaton after the first round of reachability analysis.

denotation of $\mu Z_1.\varphi_E(Z_0, Z_1)$. The first step is to set the initial value of $Z_1$, which is the empty set. The corresponding automaton is also shown in Figure 4.2. Observe that we have a separate set of initial states for $Z_0$ and $Z_1$.

We are now ready to compute $\varphi_E(Z_0, Z_1)$ which will be the next value of $Z_1$. This is a reachability property. In particular, a configuration $\langle p^j, [aw]\rangle$ with priority $c$ should be accepted if Éloïse can play - or Abelard must play - a move to some $\langle p^k, [w'w]\rangle \in V(Z_c)$. The result is shown in Figure 4.2.

The new $Z_1$ accepts configurations of the form $\langle p'_E, [a\Sigma^*\bot]\rangle_0$ or $\langle p'_E, [b\Sigma^*\bot]\rangle_0$. We can see this intuitively: there are no configurations of priority 1 since the set $Z_1$ is empty, and all configurations have an outgoing transition. The set $Z_0$ contains all configurations, hence all priority 0 configurations of the form $\langle p'_E, [a\Sigma^*\bot]\rangle_0$ or $\langle p'_E, [b\Sigma^*\bot]\rangle_0$ can perform $(p'_E, a, push_b, p'_E)$ or $(p'_E, b, push_a, p_E)$ respectively to reach $Z_0$.

Observe that the computation of the new automaton has only added new transitions. It can be shown that, when computing a least fixed point, each generation of initial states has more transitions than the previous generation. In this example the number of possible transitions is finite since all transitions go to $q_f^*$. Therefore, the automaton must eventually become saturated, and the computation will terminate. In the full algorithm, transitions are *projected*[3] to ensure that the previous generation of initial states is not reachable. Hence, the number of reachable states is finite. When computing a greatest fixed point, termination can be proved by a dual argument: we begin with an automaton containing all transitions. At each stage, some transitions are removed. The algorithm terminates when no more transitions can be removed.

We now compute the next iterate of $Z_1$. We add a new set of initial states as before, and perform another round of reachability analysis. Because the $Z_1$ input to this procedure is no longer the empty set, configurations of priority 1 that can reach a configuration $\langle p'_E, [a\Sigma^*\bot]\rangle_0$ or $\langle p'_E, [b\Sigma^*\bot]\rangle_0$ will be in the new set. This is shown in Figure 4.3.

If we were to perform another round of the reachability analysis, we would see that a fixed point has been reached[4]. That is, the transitions from the new initial states corresponding to

[3]as described in Figure 4.6

[4]On first appearances, it may seem that $\langle p_A, [a\Sigma^*\bot]\rangle_1$ should be added since the configuration $\langle p_E, [a\Sigma^*\bot]\rangle_1$

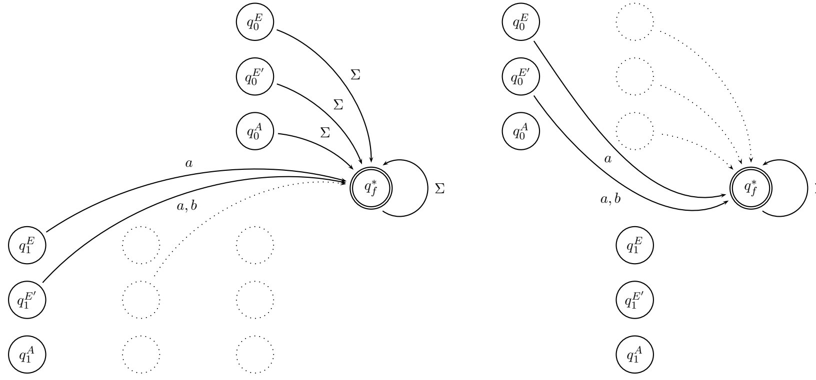



Figure 4.3: The automaton after the second round of reachability analysis; and the automaton with the new value of $Z_0$ and $Z_1$ set to the empty set.

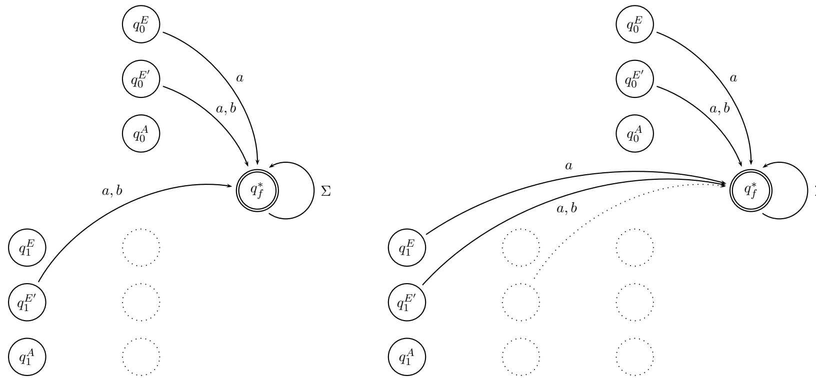



Figure 4.4: The automaton after the first round of reachability analysis with the new $Z_0$; and the automaton after the second round of reachability analysis with the new $Z_0$.

$Z_1$ have the same outgoing transitions as the old initial states. This fixed point is the next value of the set $Z_0$. Therefore, we set the current initial states of $Z_1$ to be the new initial states of $Z_0$. We then begin evaluating $\mu Z_1.\varphi_E(Z_0, Z_1)$ with our new value of $Z_0$. The initial value of $Z_1$ is the empty set, so we introduce new initial states corresponding to $Z_1$ with no outgoing transitions. Figure 4.3 shows the automaton after these steps.

We now compute the next iterate of $Z_1$ as before. The automaton at each stage is shown in Figure 4.4. The second automaton in Figure 4.4 is the fixed point of $Z_1$, and hence the new iterate of $Z_0$. Since the new $Z_0$ is identical to the previous $Z_0$, we have reached a final fixed point. By setting the initial states corresponding to $Z_1$ to be the initial states corresponding to $Z_0$, and deleting any unreachable states, we obtain the automaton shown in Figure 4.5, which accepts the winning region of Éloïse.

can be reached in one step. However, since $\langle p_A, [a\Sigma^*\bot]\rangle_1$ belongs to Abelard, it must be the case that *all* transitions reach a configuration in $Z_1$. This is not the case here.

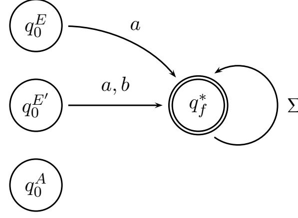



Figure 4.5: The automaton accepting the winning region of Éloïse.

## 4.3 The Algorithm

Fix a pushdown parity game $\mathcal{G} = (\mathcal{P}, \mathcal{D}, \Sigma, \Omega)$ that has $m$ priorities. The algorithm has two key components. The first — $Phi(A)$ — computes an automaton recognising $[\![\varphi_E(Z_1, \ldots, Z_m)]\!]_V^{\mathcal{G}}$, given an automaton $A$ recognising the configuration-sets $V(Z_1), \ldots, V(Z_m)$. The second — $Sig(l, A)$ — computes, for each $1 \le l \le m$, an automaton recognising $[\![\sigma Z_l.\chi_{l+1}(Z_1, \ldots, Z_l)]\!]_V^{\mathcal{G}}$ where $\sigma$ is either $\mu$ or $\nu$, given an automaton $A$ recognising the configuration-sets $V(Z_1), \ldots, V(Z_{l-1})$, where $\chi_{l+1}(Z_1, \ldots, Z_l) := \sigma Z_{l+1} \ldots \sigma Z_m.\varphi_E(Z_1, \ldots, Z_m)$.

### 4.3.1 Format of the Automata.

We describe the format of the automata constructed during the algorithm. Let $\mathcal{Q}_{all} := \{q^*, q_f^\varepsilon\}$, and $\mathcal{Q}_c := \left\{ q_c^j \mid 1 \le j \le |\mathcal{P}| \right\}$ for each $1 \le c \le m + 1$. These states are used to give the valuations of the variables $Z_1, \ldots, Z_m$, and the semantics of $\varphi_E(Z_1, \ldots, Z_m)$ when $c = m + 1$.

Let $0 \le l \le m + 1$. An automaton $A$ is said to be ***type-$l$*** just if:

1. The state-set $\mathcal{Q}_A := \mathcal{Q}_1 \cup \cdots \cup \mathcal{Q}_l \cup \mathcal{Q}_{all}$.
2. Every transition of the form $q_c^j \xrightarrow{a} Q$ has the property that $Q \neq \emptyset$, and for all $j'$ and $c' > c$, $q_{c'}^{j'} \notin Q$ (i.e. there are no transitions to states with a higher priority).
3. The only final state is $q_f^\varepsilon$ which can only be reached by a $\perp$-transition. I.e. for each $q \xrightarrow{a} Q$, we have $q_f^\varepsilon \in Q$ iff $Q = \{q_f^\varepsilon\}$ iff $a = \perp$.
4. We also have $q^* \xrightarrow{\Sigma} \{q^*\}$ and $q^* \xrightarrow{\perp} \{q_f^\varepsilon\}$ and no other transitions from $q^*$. Finally, $q_f^\varepsilon$ has no outgoing transitions.

It follows that there is a unique automaton of type-0.

In the following, let $A$ be a type-$l$ automaton, where $1 \le c \le l \le m + 1$. We define $\mathcal{L}_c(A) \subseteq \mathcal{P}\,\Sigma^* \perp$ by: for $1 \le j \le |\mathcal{P}|$, $\langle p^j, [w] \rangle \in \mathcal{L}_c(A)$ just if $w$ is accepted by $A$ from the initial state $q_c^j$. Thus $\mathcal{L}_c(A)$ is intended to represent the current valuation of the variable $Z_c$; in case $l = m + 1$, $\mathcal{L}_{m+1}(A)$ is intended to represent $[\![\varphi_E(Z_1, \ldots, Z_m)]\!]_V^{\mathcal{G}}$ where the valuation $V$ maps $Z_c$ to $\mathcal{L}_c(A)$. If we omit the subscript and write $\mathcal{L}(A)$, we mean $\mathcal{L}_l(A)$. By abuse of notation, we define $\mathcal{L}_q(A) \subseteq \Sigma^* \perp \cup \{ \epsilon \}$ to be the set of words accepted by $A$ from the state $q$ (note that $\mathcal{L}_{q^*}(A) = \Sigma^* \perp$ and $\mathcal{L}_{q_f^\varepsilon}(A) = \{ \varepsilon \}$).

procedure $Phi(A)$

*Input*: A type-$m$ automaton $A$ as valuation of $\overline{Z} = Z_1, \cdots, Z_m$.
*Output*: A type-$(m+1)$ automaton denoting $\varphi_E(\overline{Z})$, relative to $A$.

1. (*1-Step Reachability*) Construct the automaton $A'$ by adding new states $\{q^1_{m+1}, \ldots, q^{|\mathcal{P}|}_{m+1}\}$ and the following transitions to $A$. For each $1 \leq j \leq |\mathcal{P}|$, set $c := \Omega(p^j)$, and
   - if $p^j \in \mathcal{P}_E$ then $q^j_{m+1} \xrightarrow{a} Q$ if $q^k_c \xrightarrow{w} Q$ and $(p^k, w) \in Next(p^j, a)$
   - if $p^j \in \mathcal{P}_A$ then $q^j_{m+1} \xrightarrow{a} Q_1 \cup \cdots \cup Q_n$ if $q^{k_1}_c \xrightarrow{w_1} Q_1, \ \ldots, \ q^{k_n}_c \xrightarrow{w_n} Q_n$, and $Next(p^j, a) = \{(p^{k_1}, w_1), \ldots, (p^{k_n}, w_n)\}$

   where $Next(p^j, a) := \{ \ (p^k, w) \mid (p^j, a, push_w, p^k) \ \in \ \mathcal{D} \ \}$.
2. return $A'$.

procedure $Proj(l, A)$

*Input*: $1 \leq l \leq m$; a type-$(l+1)$ automaton $A$.
*Output*: A type-$l$ automaton.

1. For each $j$, replace each transition $q^j_{l+1} \xrightarrow{a} Q$ with $q^j_{l+1} \xrightarrow{a} \pi^l(Q)$ where $\pi^l(Q) := \{q^{j'}_{l+1} \mid q^{j'}_l \in Q\} \cup (Q - \mathcal{Q}_l)$.
2. For each $j$, remove the state $q^j_l$.
3. For each $j$, rename the state $q^j_{l+1}$ to $q^j_l$.

procedure $Sig(l, A)$

*Input*: $1 \leq l \leq m+1$;
a type-$(l-1)$ automaton $A$ as valuation of $Z_1, \cdots, Z_{l-1}$.
*Output*: A type-$l$ automaton denoting $\sigma Z_l \cdots \sigma Z_m \, . \, \varphi_E(\overline{Z})$, relative to $A$.

1. if $l = m+1$ then return $Phi(A)$
2. $A^0 := \begin{cases} A \text{ with new states } \mathcal{Q}_l\text{, but no new transitions} & \text{if } \sigma Z_l = \mu Z_l \\ A \text{ with new states } \mathcal{Q}_l\text{, and all outgoing} & \text{if } \sigma Z_l = \nu Z_l \\ \text{transitions obeying the format of the automata.} & \end{cases}$
3. for $i = 0$ to $\infty$ do
4. $\quad B^i := Sig(l+1, A^i)$
5. $\quad A^{i+1} := Proj(l, B^i)$
6. $\quad$ if $A^i = A^{i+1}$ then return $A^i$

*Input*: A pushdown parity game $\mathcal{G} = (\mathcal{P}, \mathcal{D}, \Sigma, \Omega)$ with $m$ priorities.
*Output*: A type-1 automaton recognising $[\![\chi_1]\!]^{\mathcal{G}}$, the winning region of $\mathcal{G}$.

begin
  return $Sig(1, A_0)$ % $A_0$ is the unique type-0 automaton.
end

Figure 4.6: Algorithm for computing winning region of a pushdown parity game.

### 4.3.2 Definition of the Algorithm.

Given a pushdown parity game $\mathcal{G}$, the algorithm presented in Figure 4.6 computes $\mathcal{W}_E$, the winning region of $\mathcal{G}$:

$$\mathcal{W}_E = [\![\mu Z_1.\nu Z_2.\dots\sigma Z_{m-1}.\sigma Z_m.\varphi_E(Z_1,\dots,Z_m)]\!]^{\mathcal{G}}_{\emptyset}.$$

In computing $[\![\varphi_E(Z_1,\dots,Z_m)]\!]^{\mathcal{G}}_{V}$ we may add an exponential number of transitions. To compute $[\![\sigma Z_l.\cdots.\sigma Z_m\,.\,\varphi_E(Z_1,\cdots,Z_m)]\!]^{\mathcal{G}}_{V}$ we may require an exponential number of iterations. Hence, in the worst case, the algorithm is exponential in the number of control states and the maximum priority $m$.

**Theorem 4.3.1.** *Given a pushdown parity game $\mathcal{G} = (\mathcal{P},\mathcal{D},\Sigma,\Omega)$, we can construct an automaton $A_{\mathcal{G}}$ recognising the winning region of Éloïse in EXPTIME in $|\mathcal{P}|\cdot m$ where $m$ is the number of priorities.*

## 4.4 Termination and Correctness

### 4.4.1 Termination

First an auxiliary notion of monotonicity for automaton constructions. Let $1 \le l,\ l' \le m+1$, and $A$ and $A'$ be type-$l$ automata. We write $A \preceq A'$ to mean: for all $q, a$ and $Q$, if $q \xrightarrow{a} Q$ is an $A$-transition then it is an $A'$-transition. We consider automaton constructions $\mathcal{T}$ (such as *Sig*, *Phi* and *Proj*) that transform type-$l$ automata to type-$l'$ automata. We say that $\mathcal{T}$ is *monotone* just if $\mathcal{T}(A) \preceq \mathcal{T}(A')$ whenever $A \preceq A'$.

To show that our winning-region construction procedure terminates, it suffices to prove the following.

**Theorem 4.4.1** (Termination)**.** *For every $1 \le l \le m+1$ and every type-$(l-1)$ automaton $A$, the procedure Sig$(l, A)$ terminates.*

We prove the Theorem by induction on $l$. It is straightforward to establish the base case of $l = m+1$: *Phi*$(A)$ (where $A$ is type-$m$) terminates. For the inductive case of *Sig*$(l,-)$ where $1 \le l \le m$, since *Sig*$(l+1,-)$ terminates by the induction hypothesis, and *Proj*$(l,-)$ clearly terminates, it remains to check that in the computation of *Sig*$(l, A)$ where $A$ is type-$(l-1)$, there exists an $i \ge 0$ such that $A^i = A^{i+1}$. Since all automata of the same type have the same finite state-set (and $A^0, A^1, \dots$ are all type-$l$) , it suffices to show (1) of the following Lemma.

**Lemma 4.4.1** (Monotonicity)**.** *We have the following properties.*

1. *Let $1 \le l \le m$ and $A$ be a type-$(l-1)$ automaton. In Sig$(l, A)$:*
   (a) *if $\sigma Z_l = \mu Z_l$ then $A^i \preceq A^{i+1}$ for all $i \ge 0$*
   (b) *if $\sigma Z_l = \nu Z_l$ then $A^{i+1} \preceq A^i$ for all $i \ge 0$.*
2. *For every $1 \le l \le m+1$, the construction Sig$(l,-)$ is monotone.*
3. *For every $1 \le l \le m$, the construction Proj$(l,-)$ is monotone.*

*Proof.* We consider each of the cases.

1. We prove the case of $\sigma Z_l = \mu Z_l$ by induction on $i$ For the base case of $i = 0$, $A^0 \preceq A^1$ trivially since there are no transitions from $q_1^j$ in $A^0$. The inductive case follows from the monotonicity of the constructions $\mathit{Sig}(l+1, -)$ and $\mathit{Proj}(l, -)$, which are the inductive hypotheses of (2) and (3) respectively.

   In the case of $\sigma Z_l = \nu Z_l$ the base case of $i = 0$, $A^1 \preceq A^0$ trivially since there are all transitions from $q_1^j$ in $A^0$. The inductive case follows as in the $\mu$ case.

2. We first establish the base case of $l = m + 1$ i.e. $\mathit{Phi}(-)$ is monotone. Let $A \preceq A'$ be type-$m$ automata. We aim to show $\mathit{Phi}(A) \preceq \mathit{Phi}(A')$ i.e. for all $1 \leq j \leq |\mathcal{P}|$, if $q^j \xrightarrow{a} Q$ in $\mathit{Phi}(A)$ then $q^j \xrightarrow{a} Q$ in $\mathit{Phi}(A')$. Since the transitions from all other states do not change, this is enough. Let $\Omega(p^j) = c$. Take $q^j \xrightarrow{a} Q$ in $\mathit{Phi}(A)$. If $p \in \mathcal{P}_E$ we have some rule $(p^j, a, push_{w_1}, p^{k_1})$ with the run $q_c^{k_1} \xrightarrow{w_1} Q$ in $A$. Otherwise, $p^j \in \mathcal{P}_A$, $Next(p, a)$ is the set $\{(p^{k_1}, w_1), \ldots, (p^{k_n}, w_n)\}$ and $Q = Q^1 \cup \cdots \cup Q^n$ with the following transitions $q_c^{k_1} \xrightarrow{w_1} Q^1$, $\ldots$, $q_c^{k_n} \xrightarrow{w_n} Q^n$ in $A$. Since the former case can easily be encoded as an instance of the latter, we argue the second case only. For all $1 \leq t \in n$, we have $q_c^{k_t} \xrightarrow{w_t} Q^t$ in $A$ and since $A \preceq A'$ we know that $q_c^{k_t} \xrightarrow{w_t} Q^t$ in $A'$. Therefore, we have $Q = Q^1 \cup \cdots \cup Q^n$ and, by the definition of the procedure $\mathit{Phi}(-)$, $q^j \xrightarrow{a} Q$ in $A'$ as required.

   For the inductive case, we consider the case of $\sigma Z_l = \mu Z_l$ (the case of $\sigma Z_l = \nu Z_l$ is omitted as the proof is dual). Let $A_1 \preceq A_2$ be type-$(l-1)$ automata. For each $i \in \{1, 2\}$, let $A_i^0, A_i^1, A_i^2, \ldots$ be the intermediate automata that are constructed in the computation of $\mathit{Sig}(l, A_i)$. By the induction hypothesis of (1), we have $A_i^0 \preceq A_i^1 \preceq A_i^2 \preceq \ldots$. Since $\mathit{Sig}(l+1, -)$ and $\mathit{Proj}(l, -)$ are monotone by the induction hypothesis of (2) and (3) respectively, we have $A_1^i \preceq A_2^i$ for each $i \geq 0$. It follows that $\mathit{Sig}(l, A_1) \preceq \mathit{Sig}(l, A_2)$ as required.

3. It can easily be seen that $\mathit{Proj}(l, -)$ is monotone.

We conclude the proof. □

### 4.4.2 Correctness

To prove correctness, we introduce the notions of *valuation soundness* and *completeness*.

#### Valuation Soundness

When computing the least fixed point of a formula of the form $\mu Z.\chi(Z)$, we begin from an empty automaton — giving an initial valuation of $Z$ — and construct an automaton accepting $\chi(Z)$. After each iteration of the algorithm for $\chi(Z)$, we perform some projections. That is, whenever we have a transition of the form $q \xrightarrow{a} q'$ where $q$ is intended to give a valuation of $\chi(Z)$ and $q'$ gives a valuation of $Z$, we replace the transition with $q \xrightarrow{a} q$. The valuation for $\chi(Z)$ after these projections becomes the new (partial) valuation for $Z$, which we want to eventually reach the valuation of $\mu Z.\chi(Z)$. By monotonicity, we know that the automaton will eventually become saturated. However, this does not justify the correctness of performing these projections.

In the case of a least fixed point, the *valuation soundness* condition intuitively asserts that if we have a transition $q \xrightarrow{a} q'$, then, whenever a word $w$ should be accepted from $q'$ in the least fixed point, then the word $aw$ should also be accepted from $q$. Since, when $Z = [\![\mu Z.\chi(Z)]\!]$,

it is the case that $[\![\mu Z.\chi(Z)]\!] = [\![\chi(\mu Z.\chi(Z))]\!]$, the projections can be seen to be correct: we are replacing a valuation of $[\![\mu Z.\chi(Z)]\!]$ with a valuation of $[\![\chi(\mu Z.\chi(Z))]\!]$.

However, the full algorithm contains both greatest and least fixed point computations. In particular, the greatest fixed point computation begins with all possible transitions. Clearly this automaton cannot be valuation sound. The solution to this problem is to use approximants. Initially, a greatest fixed point computation is only sound with respect to $\nu^0 Z.\chi(Z)$, which evaluates to the set of all configurations. It can be shown that the approximant is able to grow transfinitely until we are sound with respect to $\nu Z.\chi(Z)$.

Fix a pushdown parity game $\mathcal{G} = (\mathcal{P}, \mathcal{D}, \Sigma, \Omega)$. A *valuation profile* is a vector $\overline{S} = (S_1, \ldots, S_l)$ of configuration-sets (i.e. vertex-sets of the underlying configuration graph). We define the induced valuation $V_{\overline{S}} : Z_c \mapsto S_c$, which we extend to a map $V_{\overline{S}} : \mathcal{Q}_A \longrightarrow 2^{\Sigma^* \perp}$ on the states of a type-$l$ automaton as follows:

$$V_{\overline{S}} := \begin{cases} q_c^j & \mapsto & \{ \ w \mid \langle p^j, [w] \rangle \in S_c \ \} \quad 1 \leq j \leq |\mathcal{P}|, \ 1 \leq c \leq l \\ q^* & \mapsto & \Sigma^* \perp \\ q_f^\varepsilon & \mapsto & \{ \ \varepsilon \ \} \end{cases}$$

**Definition 4.4.1.** Given a valuation profile $\overline{S}$ of length $l$, a type-$l$ automaton $A$ is $\overline{S}$***-sound*** just if, for all $q$, $a$ and $w$, if $A$ has a transition $q \xrightarrow{a} Q$ such that $w \in V_{\overline{S}}(q')$ for all $q' \in Q$, then $a\,w \in V_{\overline{S}}(q)$.

By induction on the length of the word, valuation soundness extends to runs. We then obtain that all accepting runs are sound.

**Lemma 4.4.2.** *Let $A$ be a $\overline{S}$-sound automaton.*

1. *For all $q$, $w$ and $w'$, if $A$ has a run $q \xrightarrow{w} Q$ such that $w' \in V_{\overline{S}}(q')$ for all $q' \in Q$, then $w\,w' \in V_{\overline{S}}(q)$.*
2. *For all $q \in \mathcal{Q}_A$, $\mathcal{L}_q(A) \subseteq V_{\overline{S}}(q)$.*

*Proof.* We prove each of the properties individually.

1. We prove by induction on the length of the word $w$. When $w = a$, the property is just $\overline{S}$-soundness. Take $w = au$ and some run $q \xrightarrow{a} Q \xrightarrow{u} Q'$ such that for all $q' \in Q'$, we have $w \in V_{\overline{S}}(q')$. By the induction hypothesis, we have the property for the run $Q \xrightarrow{u} Q'$. Hence, we have for all $q' \in Q$ that, $uw' \in V_{\overline{S}}(q')$. Thus, from $\overline{S}$-soundness, we have $auw' \in V_{\overline{S}}(q)$.
2. Take an accepting run $q \xrightarrow{w} Q_f$ of $A$. We have for all $q' \in Q_f = \{q_f^\varepsilon\}$, $\varepsilon \in V_{\overline{S}}(q')$. Thanks to (1), we have $w \in V_{\overline{S}}(q)$.

Hence, the properties hold as required. □

### Valuation Completeness

Dual to valuation soundness, correctness of the algorithm also relies on a property we call valuation completeness. In the case of the greatest fixed point, this property asserts that for every word $aw$ that should be accepted from $q$ in the greatest fixed point, we have a transition $q \xrightarrow{a} q'$ such that $w$ should be accepted from $q'$ in the greatest fixed point. Whereas

the least fixed point computation begins with no transitions and only adds transitions that are justified, the greatest fixed point computation begins with all transitions, and removes those that are not.

Since a least fixed point computation begins with no transitions from the new initial states, it cannot be valuation complete with respect to the least fixed point being computed: in general, the least fixed point will contain some configuration and hence some word must be accepted from some initial state. To satisfy valuation completeness, then, some transitions may be required to exist from the new initial states. However, the zeroth approximant $[\![\mu^0 Z.\chi(Z)]\!]$ is the empty set. Since there are no configurations in this set, no transitions are required to satisfy valuation completeness. Thus, we are initially valuation complete with respect to $[\![\mu^0 Z.\chi(Z)]\!]$. We show that the construction tightens the approximation until the least fixed point is reached.

**Definition 4.4.2.** Given a valuation profile $\overline{S}$ of length $l$, a type-$l$ automaton $A$ is $\overline{S}$-***complete*** just if, for all $q$, $a$ and $w$, if $a\,w \in V_{\overline{S}}(q)$ then $A$ has a transition $q \xrightarrow{a} Q$ such that $w \in V_{\overline{S}}(q')$ for all $q' \in Q$.

By induction on the length of the word, valuation completeness extends to runs. Furthermore, an accepting run always exists when required.

**Lemma 4.4.3.** *Let $A$ be an $\overline{S}$-complete automaton.*

1. *For all $q$, $w$ and $w'$, if $w\,w' \in V_{\overline{S}}(q)$ then $A$ has a run $q \xrightarrow{w} Q$ such that $w' \in V_{\overline{S}}(q')$ for all $q' \in Q$.*
2. *For all $q \in \mathcal{Q}_A$, $V_{\overline{S}}(q) \subseteq \mathcal{L}_q(A)$.*

*Proof.* We prove each property.

1. The proof is by induction on the length of the word $w$. When $w = a$, the property is simply $\overline{S}$-completeness. Take $w = au$ and some $q$ with $auw' \in V_{\overline{S}}(q)$. From $\overline{S}$-completeness, we have a transition $q \xrightarrow{a} Q$ such that for all $q' \in Q$, we have $uw \in V_{\overline{S}}(q')$. By induction on the length of the word, we have a run $Q \xrightarrow{u} Q'$ satisfying the property. Hence, we have $q \xrightarrow{a} Q \xrightarrow{u} Q'$ as required.
2. Take $w \in V_{\overline{S}}(q)$. Instantiating (i) with $w' = \varepsilon$, we know $A$ has a run $q \xrightarrow{w} Q$. Every state in $Q$ must be accepting because $\varepsilon$ is only accepted from accepting states and there can be no $\langle p^j, [\varepsilon]\rangle$ satisfying any $S_i$ because $\varepsilon$ is not a valid stack.

Hence, the properties hold as required. □

### Correctness of the Algorithm

Using valuation soundness and valuation completeness we can show that the algorithm is correct. We begin by introducing some notation. Let $1 \leq l \leq m+1$. We write

$$\chi_l(Z_1, \ldots, Z_{l-1}) \;:=\; \sigma Z_l \cdots Z_m.\varphi_E(Z_1, \ldots, Z_m).$$

Thus $\chi_1 = \mu Z_1 \ldots \sigma Z_m.\varphi_E(\overline{Z})$ and $\chi_{m+1}(Z_1, \ldots, Z_m) = \varphi_E(\overline{Z})$. Let $\overline{S} = (S_1, \ldots, S_{l-1})$; we write $(\overline{S}, T)$ to mean $(S_1, \ldots, S_{l-1}, T)$. Thus we write (say) $\chi_l(\overline{S})$ to mean $\chi_l(S_1, \cdots, S_{l-1})$, and $\chi_{l+1}(\overline{S}, Z_l)$ to mean $\chi_{l+1}(S_1, \ldots, S_{l-1}, Z_l)$.

**Proposition 4.4.1** (Main). *Let $1 \leq l \leq m+1$, $A$ be a type-$(l-1)$ automaton, and $\overline{S}$ be a valuation profile of length $l-1$.*

1. *(**Soundness Preservation**) If $A$ is $\overline{S}$-sound, then $Sig(l, A)$ is a type-$l$ automaton which is $(\overline{S}, [\![\chi_l(\overline{S})]\!])$-sound.*[5]

2. *(**Completeness Preservation**) If $A$ is $\overline{S}$-complete, then $Sig(l, A)$ is a type-$l$ automaton which is $(\overline{S}, [\![\chi_l(\overline{S})]\!])$-complete.*

Since the type-0 automaton $A_0$ is trivially sound and complete with respect to the empty valuation profile, we obtain following as an immediate corollary.

**Theorem 4.4.2** (Correctness). *The procedure call $Sig(1, A_0)$ terminates and returns a type-1 automaton which is $([\![\chi_1]\!])$-sound and $([\![\chi_1]\!])$-complete. Hence, thanks to Lemmas 4.4.2 and 4.4.3, for each $1 \leq j \leq |\mathcal{P}|$, $V_{[\![\chi_1]\!]}(q_1^j) = \mathcal{L}_{q_1^j}(Sig(1, A_0))$ i.e. the automaton $Sig(1, A_0)$ recognises the configuration set $[\![\chi_1]\!]$, which is the winning region of the pushdown parity game $\mathcal{G}$.*

### Proof of the Main Proposition

We prove Proposition 4.4.1 by induction on $l$. First the base case: $l = m+1$.

**Lemma 4.4.4.** *Let $\overline{S}$ be a valuation profile of length $m$, and $A$ a type-$m$ automaton.*

1. *$Phi(A)$ is a type-$(m+1)$ automaton.*

2. *If $A$ is $\overline{S}$-sound then $Phi(A)$ is $(\overline{S}, [\![\varphi_E(\overline{S})]\!])$-sound.*

3. *If $A$ is $\overline{S}$-complete then $Phi(A)$ is $(\overline{S}, [\![\varphi_E(\overline{S})]\!])$-complete.*

*Proof.* We prove each case individually.

1. Let $A$ be a type-$m$ automaton. $Phi(A)$ is a type-$(m+1)$ automaton. I.e. all transitions $q \xrightarrow{a} Q$ satisfy: $q_f^\varepsilon \in Q$ iff $a = \perp$ iff $Q = \{ q^\varepsilon \}$. Suppose there is some transition $q^j \xrightarrow{\perp} Q$ with $q_f^\varepsilon \notin Q$ or $Q \neq \{q_f^\varepsilon\}$. Then the transition was added from some appropriate $Next(p^j, \perp)$. Then it must be the case that for some $(p^k, w) \in Next(p^j, \perp)$ the last character in $w$ is not $\perp$ (else $q_f^\varepsilon \in Q$). This means $\perp$ is removed from the stack, which is explicitly disallowed.

   Conversely, suppose there is some transition $q^j \xrightarrow{a} Q$ where $a \neq \perp$ and $q_f^\varepsilon \in Q$. Then the transition was added from some appropriate $Next(p^j, a)$. It must be the case that for some $(p^k, w) \in Next(p^j, a)$ the last character in $w$ is $\perp$ (else $q_f^\varepsilon \notin Q$). This means $\perp$ is pushed on to the stack, which is explicitly disallowed.

2. Set $\overline{S'} = (\overline{S}, [\![\varphi_E(\overline{S})]\!])$ and let $\Omega(p^j) = c$. Take any transition $q_{m+1}^j \xrightarrow{a} Q$ in $Phi(A)$ such that for all $q_{c'}^{j'} \in Q$, $\langle p^{j'}, [w] \rangle \in V_{\overline{S'}}(Z_{c'})$. Abusing notation, we take an appropriate assignment to $Next(p^j, a)$ — the complete value of $Next(p^j, a)$ for an Abelard position, and a single command for an Éloïse position — that led to the introduction of the transition. Since $A$ is $\overline{S}$-sound and for all $(p^k, w_k) \in Next(p^j, a)$ we have $q_c^k \xrightarrow{w_k} Q_k \subseteq$

[5]By $[\![\chi_l(S_1, \cdots, S_{l-1})]\!]$ we mean $[\![\chi_l(Z_1, \cdots, Z_{l-1})]\!]_V$ w.r.t. a valuation $V$ that maps $Z_c$ to $S_c$.

$Q$, we know that $\langle p^k, [w_k w]\rangle \in V_{\overline{S'}}(Z_c)$. Hence all $\langle p^k, [w_k w]\rangle$ are in $V_{\overline{S'}}(Z_c)$, and $\langle p^j, [aw]\rangle \in V_{\overline{S'}}(Z_{m+1}) = [\![\varphi_E(\overline{Z})]\!]^{\mathcal{G}}_{V_{\overline{S'}}}$, since all moves, in the case of Abelard, and a move in the case of Éloïse, reach configurations in $Z_c$.

3. Take any configuration $\langle p^j, [aw]\rangle \in V_{\overline{S'}}(Z_{m+1}) = [\![\varphi_E(\overline{Z})]\!]^{\mathcal{G}}_{V_{\overline{S'}}}$. Let $\Omega(p^j) = c$. There exists an appropriate assignment $\{(p^{k_1}, w_1), \ldots, (p^{k_n}, w_n)\}$ to $Next(p^j, a)$ (as before) such that $\langle p^{k_h}, [w_h w]\rangle \in V_{\overline{S'}}(Z_c)$ for all $h \in \{1, \ldots, n\}$. Since $A$ is assumed to be $\overline{S}$-complete, it follows that all $\langle p^{k_h}, [w_h w]\rangle$ have a complete run. In particular, we have a complete run $q_c^{k_h} \xrightarrow{w_h} Q_h$ for all $h$. Hence, by the definition of $Phi(A)$, there exists a transition $p^j \xrightarrow{a} Q$ that is complete.

Hence, $Phi(-)$ is correct. □

For the inductive case of $1 \leq l \leq m$, there are two sub-cases: $\sigma Z_l = \mu Z_l$, and $\sigma Z_l = \nu Z_l$. Recall that $\chi_l(Z_1, \ldots, Z_{l-1}) := \sigma Z_l.\chi_{l+1}(Z_1, \cdots, Z_l)$. We begin with the proof when $\sigma Z_l = \mu Z_l$.

**Lemma 4.4.5.** *Suppose $\sigma Z_l = \mu Z_l$. Let $\overline{S}$ be a valuation profile of length $l-1$, and $A$ be a type-$(l-1)$ automaton; set $\theta = [\![\mu Z_l.\chi_{l+1}(\overline{S}, Z_l)]\!]$.*

1. *$Sig(l, A)$ is a type-$l$ automaton.*
2. *If $A$ is $\overline{S}$-sound, then $Sig(l, A)$ is $(\overline{S}, \theta)$-sound.*
3. *If $A$ is $\overline{S}$-complete, then $Sig(l, A)$ is $(\overline{S}, \theta)$-complete.*

*Proof.* We consider each case separately.

1. The result of the recursive call to $Sig(l+1, A)$ combined with the call to $Proj$ ensures the property.

2. Let $\overline{S'} := (\overline{S}, \theta)$. It is straightforward to see that $A^0$ is $\overline{S'}$-sound, since it did not add any transitions to $A$, which is assumed to be $\overline{S}$-sound. Hence, we assume by induction $A^i$ is $\overline{S'}$-sound and argue the case for $A^{i+1}$.

   Take a transition $q_l^j \xrightarrow{a} Q$ in $A^{i+1}$ such that for all $q_{l'}^k \in Q$ we have $\langle p^k, [w]\rangle \in V_{\overline{S'}}(Z_{l'})$. Take the corresponding transition $q_{l+1}^j \xrightarrow{a} Q'$ in $Sig(l+1, A^i)$ before the projection. In particular, for every $q_l^k \in Q$ we have $q_l^k$ or $q_{l+1}^k$ in $Q'$. By the induction hypothesis, we know $Sig(l+1, A^i)$ is $(\overline{S'}, [\![\chi_{l+1}(\overline{S'})]\!])$-sound. Furthermore, $V_{\overline{S'}}(Z_l) = \theta = [\![\chi_{l+1}(\overline{S}, \theta)]\!] = V_{\overline{S'}}(Z_{l+1})$. Since $Sig(l+1, A^i)$ is $(\overline{S'}, [\![\chi_{l+1}(\overline{S'})]\!])$-sound, we have $\langle p^j, [aw]\rangle \in V_{\overline{S'}}(Z_{l+1}) = V_{\overline{S'}}(Z_l)$ as required.

3. Let $A$ be a type-$(l-1)$ automaton which is $\overline{S}$-complete. We use the shorthand $\theta^\alpha = [\![\mu^\alpha Z_l.\chi_{l+1}(\overline{S}, Z_l)]\!]$. We first show that if the type-$l$ $A^i$ is $(\overline{S}, \theta^\alpha)$-complete for some $\alpha$ then $A^{i+1}$ is $(\overline{S}, \theta^{\alpha+1})$-complete. By the induction hypothesis, $B^i := Sig(l+1, A^i)$ is $(\overline{S}, \theta^\alpha, \theta^{\alpha+1})$-complete, since $\theta^{\alpha+1} = [\![\chi_{l+1}(\overline{S}, \theta^\alpha)]\!]$. We need to show that, after the projection, $A^{i+1} := Proj(l, B^i)$ is $\overline{S'}$-complete, where $\overline{S'} := (\overline{S}, \theta^{\alpha+1})$. Take some $\langle p^j, [aw]\rangle \in V_{\overline{S'}}(Z_l)$. We know $B^i$ has a transition $q_{l+1}^j \xrightarrow{a} Q$ satisfying completeness. If $Q$ contains no states of the form $q_l^k$, then the transition $q_l^j \xrightarrow{a} Q$ satisfies completeness in $A^{i+1}$. If $Q$ contains states $q_l^k$, then $\langle p^k, [w]\rangle \in \theta^\alpha \subseteq \theta^{\alpha+1} = V_{\overline{S'}}(Z_l)$. Hence, we have

a required complete transition after the projection, and so, $A^{i+1}$ is $\overline{S'}$-complete. We require that $Sig(l, A)$ be $(\overline{S}, [\![\mu Z_l.\chi_{l+1}(\overline{S}, Z_l)]\!])$-complete. Take $i$ such that $A^i = A^{i+1} = Sig(l, A)$. Trivially $Sig(l, A)$ is $(\overline{S}, \theta^0)$-complete. We proceed by transfinite induction. For a successor ordinal we know by induction that $A^i$ is $(\overline{S}, \theta^\alpha)$-complete and from the above that $A^{i+1}$ is $(\overline{S}, \theta^{\alpha+1})$-complete. Since $Sig(l, A) = A^i = A^{i+1}$ we are done. For a limit ordinal $\lambda$, we have that $Sig(l, A)$ is $(\overline{S}, \theta^\alpha)$-complete for all $\alpha < \lambda$. Since $\theta^\lambda = \bigcup_{\alpha<\lambda} \theta^\alpha$, the result follows because each configuration in the limit appears in some smaller approximant, and the transition witnessing completeness for the approximant witnesses completeness for the limit.

This conclude the proof. ☐

Finally, we consider the greatest fixed point computations.

**Lemma 4.4.6.** *Suppose* $\sigma Z_l = \nu Z_l$. *Set* $\theta = [\![\nu Z_l.\chi_{l+1}(\overline{S}, Z_l)]\!]$.

1. $Sig(l, A)$ *is a type-*$l$ *automaton.*
2. *If* $A$ *is* $\overline{S}$*-sound, then* $Sig(l, A)$ *is* $(\overline{S}, \theta)$*-sound.*
3. *If* $A$ *is* $\overline{S}$*-complete, then* $Sig(l, A)$ *is* $(\overline{S}, \theta)$*-complete.*

*Proof.* We analyse each of the cases.

1. The result of the recursive call to $Sig(l + 1, A)$ combined with the call to *Proj* ensures the property.

2. Let $A$ be a type-$(l - 1)$ automaton which is $\overline{S}$-sound. We use the shorthand $\theta^\alpha = [\![\nu^\alpha Z_l.\chi_{l+1}(\overline{S}, Z_l)]\!]$. We first show that if $A^i$ is $(\overline{S}, \theta^\alpha)$-sound for some $\alpha$, then $A^{i+1}$ is $(\overline{S}, \theta^{\alpha+1})$-sound. By the induction hypothesis, $B^i := Sig(l + 1, A^i)$ is $(\overline{S}, \theta^\alpha, \theta^{\alpha+1})$-sound, since $\theta^{\alpha+1} = [\![\chi_{l+1}(\overline{S}, \theta^\alpha)]\!]$. We need to show that, after the projections, $A^{i+1} := Proj(l, B^i)$ is $\overline{S'}$-sound where $\overline{S'} := (\overline{S}, \theta^{\alpha+1})$. Take some transition $q^j_l \xrightarrow{a} Q$ in $A^{i+1}$ such that for all $q^k_{l'} \in Q$ we have $\langle p^k, [w]\rangle \in V_{\overline{S'}}(Z_{l'})$. We know $B^{i+1}$ had a sound, unprojected transition $q^j_{l+1} \xrightarrow{a} Q'$ such that for all $q^k_l \in Q$ we have either $q^k_l \in Q'$ or $q^k_{l+1} \in Q'$. In the former case, by assumption we know $\langle p^k, [w]\rangle \in \theta^{\alpha+1} \subseteq \theta^\alpha$. In the latter $\langle p^k, [w]\rangle \in \theta^{\alpha+1}$, also by assumption. Since $B^i$ is $(\overline{S}, \theta^\alpha, \theta^{\alpha+1})$-sound we know $\langle p^j, [aw]\rangle \in \theta^{\alpha+1}$ as required.

   We require that $Sig(l, A)$ be $(\overline{S}, [\![\nu Z_l.\chi_{l+1}(\overline{S}, Z_l)]\!])$-sound. Observe that $Sig(l, A)$ is $(\overline{S}, \theta^0)$-sound (trivially, since the zeroth approximant contains all configurations). We proceed by transfinite induction. Take $i$ such that $Sig(l, A) = A^i = A^{i+1}$. By induction we have that $A^i$ is $(\overline{S}, \theta^\alpha)$-sound, then from the above we know $A^{i+1}$ is $(\overline{S}, \theta^{\alpha+1})$-sound. Since $A^i = A^{i+1} = Sig(l, A)$ we are done. For a limit ordinal $\lambda$, we have that $Sig(l, A)$ is $(\overline{S}, \theta^\alpha)$-sound for all $\alpha < \lambda$. Since Since $\theta^\lambda = \bigcap_{\alpha<\lambda} \theta^\alpha$, the result follows because each configuration in the limit appears in all smaller approximants, and $Sig(l, A)$ is sound for all smaller approximants (and trivially for the zeroth approximant).

3. Let $\overline{S'} := (\overline{S}, \theta)$. It can be easily seen that $A^0$ is $\overline{S'}$-complete (always move to $q^*$ or $q^\varepsilon_f$ during the first transition). Hence, we assume $A^i$ is $\overline{S'}$-complete. We argue the case for $i + 1$.

Take some $\langle p^j, [aw]\rangle$ such that $\langle p^j, [aw]\rangle \in V_{\overline{S}}(Z_l)$. By the induction hypothesis, we know $Sig(l+1, A^i)$ is $(\overline{S'}, [\![\chi_{l+1}(\overline{S'})]\!])$-complete. Furthermore, we have that $V_{\overline{S'}}(Z_l) = \theta = [\![\chi_{l+1}(\overline{S}, \theta)]\!] = V_{\overline{S'}}(Z_{l+1})$. Since we have an $(\overline{S'}, [\![\chi_{l+1}(\overline{S'})]\!])$-complete transition $q^j \xrightarrow{a} Q$ in $A^{i+1}$ before the projections, it follows that, for all $q_{c'}^{j'} \in \pi^l(Q)$ we know, $\langle p^j, [w]\rangle \in V_{\overline{S}}(Z_{c'})$ as required.

□

## 4.5 Optimisation

Because the initial automaton in the greatest fixed point constructions contain all allowable transitions, the algorithm is immediately exponential. However, it is easy to see that if we have two transitions $q \xrightarrow{a} Q$ and $q \xrightarrow{a} Q'$ with $Q \subseteq Q'$, then an accepting run from $Q'$ implies an accepting run from $Q$. Hence, the transition to $Q'$ is redundant. Furthermore, one can observe that an accepting run from any state $q_c^j$ implies an accepting run from $q^*$. Using these observations, we can hope to reduce the number of transitions in our automaton. In the following definition $Q \ll Q'$ can be read to mean that an accepting run from $Q'$ implies an accepting run from $Q$.

**Definition 4.5.1.** For all non-empty sets of states $Q$ and $Q'$, we define

$$Q \ll Q' \quad \iff \big((q^* \in Q \Rightarrow \exists q \neq q_f^\varepsilon \in Q') \ \wedge \ (q \neq q^* \in Q \Rightarrow q \in Q')\big)$$

Let

$$\textsc{Expand}(A) = \{\ q \xrightarrow{a} Q' \mid q \xrightarrow{a} Q \text{ in } A \text{ and } Q \ll Q'\ \}$$

We can specify the properties of our construction with respect to $\textsc{Expand}(A)$ rather than $A$. For example, after each step of a least fixed point computation we will have $\textsc{Expand}(A^i) \preceq \textsc{Expand}(A^{i+1})$ rather than $A^i \preceq A^{i+1}$. If we do this, the initial automaton in the greatest fixed point computation only needs transitions to $q^*$ and $q_f^\varepsilon$. From these transitions, the expansion will have all possible transitions (since $q^*$ can be replaced any number of states of the form $q_j^q l$). The number of initial transitions is then reduced from exponential to linear.

Furthermore, if an automaton $A$ has transitions $q \xrightarrow{a} Q$ and $q \xrightarrow{a} Q'$ with $Q \ll Q'$, then we can remove the second transition. Since a transition to $\{q^*\}$ is very powerful with respect to $\ll$, applying this optimisation after each stage of the algorithm will hopefully keep the automaton small. However, this will have to be confirmed experimentally.

To test termination of the fixed point constructions we will need to determine when $\textsc{Expand}(A^{i+1}) = \textsc{Expand}(A^i)$. Performing the full expansion would obviously cause an exponential blow up. Fortunately we can test the condition more directly using the property below. This property allows us to check $\textsc{Expand}(A^{i+1}) \preceq \textsc{Expand}(A^i)$ by comparing the existing transitions, rather than by performing the expansion and then checking which transitions occur.

**Property 4.5.1.** *Given $A$ and $A'$, we have $\textsc{Expand}(A) \preceq \textsc{Expand}(A')$ iff,*

$$q \xrightarrow[A]{a} Q \quad \Rightarrow \quad q \xrightarrow[A']{a} Q'$$

*with $Q' \ll Q$.*

*Proof.* First we assume $\text{Expand}(A) \preceq \text{Expand}(A')$ and show,

$$q \xrightarrow[A]{a} Q \quad \Rightarrow \quad q \xrightarrow[A']{a} Q'$$

with $Q' \ll Q$. Take $q \xrightarrow{a} Q$ in $A$. Then $q \xrightarrow{a} Q \in \text{Expand}(A)$. We have $q \xrightarrow{a} Q \in \text{Expand}(A')$, and therefore $q \xrightarrow{a} Q'$ is a transition of $A'$ with $Q' \ll Q$.

In the other direction, we assume

$$q \xrightarrow[A]{a} Q \quad \Rightarrow \quad q \xrightarrow[A']{a} Q'$$

Take $q \xrightarrow{a} Q \in \text{Expand}(A^1)$. We need $q \xrightarrow{a} Q \in \text{Expand}(A')$. We have some $q \xrightarrow{a} Q'$ in $A$ with $Q' \ll Q$. Hence, we have $q \xrightarrow{a} Q''$ in $A'$ with $Q'' \ll Q$. Hence, $q \xrightarrow{a} Q \in \text{Expand}(A')$ as required. □

We can extend the definition to runs as follows. This gives us that $\text{Expand}(A) \preceq \text{Expand}(A')$ implies $\mathcal{L}(A) \subseteq \mathcal{L}(A')$. Additionally, it is easy to check using this lemma that the constructions preserve the new notion of monotonicity.

**Lemma 4.5.1.** *Given $A$ and $A'$, with $\text{Expand}(A) \preceq \text{Expand}(A')$ we have,*

$$q \xrightarrow[A]{w} Q \quad \Rightarrow \quad q \xrightarrow[A']{w} Q'$$

*with $Q' \ll Q$.*

*Proof.* The proof is by induction over the length of $w$. In the base case $w = a$ and the proof follows directly from $\preceq$. When $w = aw'$ with $w' \neq \varepsilon$ we have,

$$q \xrightarrow{a} Q_1 \xrightarrow{w'} Q_2$$

where $q_f^\varepsilon \notin Q_1$ (since $w' \neq \varepsilon$). By $\preceq$ we have $q \xrightarrow{a} Q'_1$ with $Q'_1 \ll Q_1$. By induction and that $q^* \xrightarrow{a} \{q^*\}$ for all $a \neq \perp$ and $q^* \xrightarrow{\perp} \{q_f^\varepsilon\}$ we also have $Q'_1 \xrightarrow{w'} Q'_2$ with $Q'_2 \ll Q_2$, and hence $q \xrightarrow{w} Q_2$ as required. □

Finally, we check that the optimisations do not contradict the important properties of the construction.

**Property 4.5.2.** *The optimisation preserves monotonicity and both valuation soundness and completeness.*

*Proof.* Let $A'$ be $A$ with a removed transition. That $\text{Expand}(A') \preceq \text{Expand}(A)$ is immediate, since we have only removed a transition from $A$ to obtain $A'$. To show $\text{Expand}(A) \preceq \text{Expand}(A')$ we only need to consider the removed transition (since all other transitions can be matched with their counterpart). Since $q \xrightarrow{a} Q'$ can be matched with $q \xrightarrow{a} Q$, which has $Q \ll Q'$, we are done.

Preservation of valuation soundness is straightforward since we have only removed a transition. Finally, suppose a valuation complete transition $q \xrightarrow{a} Q$ was removed by the optimisation. This implies that there exists a transition $q \xrightarrow{a} Q'$ with $Q' \ll Q$. Suppose this transition is not $V$-complete. Then there is some incomplete state $q \in Q'$. Since this state is not $q^*$, it must also appear in $Q$. This is a contradiction, since $q \xrightarrow{a} Q$ is valuation complete. □

## 4.6 Existing Approaches

There are several other algorithms for computing the winning regions of an order-1 pushdown parity games. These are due to Vardi *et al.* [90, 86, 85], Cachat [127] and Serre [92]. The approach of Vardi *et al.* uses a tree representation of order-1 stores, which is navigated by a two-way alternating tree automaton. They show that this approach can be applied to a number of infinite-state model-checking problems. The algorithms of Cachat and Serre interpret the output of Walukiewicz's local model-checking algorithm (see Section 2.6.3) to construct alternating automata recognising the winning regions. An informal account of these results is given below. In Section 4.6.4 we compare these methods to our own.

### 4.6.1 Extending Walukiewicz's Algorithm

Cachat's algorithm [127] is an extension of Walukiewicz's local model-checking algorithm described in Section 2.6.3. The local model-checking algorithm determines whether an initial configuration $\langle p, [\bot] \rangle$ is winning for Éloïse, by reduction to a finite state game. The principal components of this game are states of the form $Check(\vec{A}, z, \theta, c, p)$, where $\vec{A}$ contains conditions on any simulated $pop_1$ actions, $z$ is the $top_1$ character of the stack being simulated, $\theta$ is the smallest priority seen since $z$ was pushed onto the stack, $c$ is the priority of the state and $p$ is the control state being simulated.

Cachat's algorithm uses the winning region $W_E^{fin}$ of the finite state game to construct the winning region of the pushdown game. Let $[D]^m = (D, \ldots, D) \subseteq (2^{\mathcal{P}})^m$. The following routine determines whether a configuration $\langle p, [a_0 \ldots a_h] \rangle$ is winning for Éloïse in the pushdown parity game. It reads the stack from bottom to top, using the winning region of the finite state game to determine the minimal constraints Éloïse must satisfy to win with the current stack.

$D_{h+1} := \emptyset$
**for** $i = h$ **downto** $0$ **do**

$$D_i := \{ \ p' \in \mathcal{P} \mid Check([D_{i+1}]^m, a_i, m, \Omega(p'), p') \in W_E^{fin} \ \}$$

**return** $p \in D_0$

The above algorithm returns true if $\langle p, [a_0 \ldots a_h] \rangle$ is in Éloïse's winning region of the pushdown parity game. The iteration can be transformed into an alternating automaton over the stack contents. Hence, the winning region is regular.

**Theorem 4.6.1** ([127]). *Given a pushdown game with a parity winning condition, one can compute uniformly the winning region of Éloïse, which is regular, and a winning pushdown strategy.*

### 4.6.2 Games with $\omega$-Regular Winning Conditions

A second, independent, method for computing the winning regions of a pushdown parity game was provided by Serre [92]. This technique is similar to Cachat's, although the proof is more explicit, the complexity is analysed, and the result is generalised to any $\omega$-regular winning condition. That is, a winning condition specified by a finite, deterministic, parity automaton. Let $W_E$ be the winning region of Éloïse in the pushdown game.

### Conditional Games

Serre's construction uses a kind of oracle $\mathcal{R}(p, a)$ which defines the smallest sets of control states $R$ that Éloïse can force play to reach when the top character $a$ of the current configuration $\langle p, [aw] \rangle$ for some $w$ is popped. The definition of $\mathcal{R}(p, a)$ requires **conditional games**.

Given a pushdown parity game $G$, let $G(R)$ be a conditional game. Plays of $G(R)$ are the same as $G$, with the same winning conditions, except when the stack is emptied. In this case, Éloïse wins if the current control state is in $R$, and loses otherwise. We can construct $G(R)$ from $G$ as follows:

- Add sink states $p_A$ and $p_E$ with transitions $(p_A, \bot, push_\bot, p_A)$ and $(p_E, \bot, push_\bot, p_E)$ and priorities 1 and 2 respectively.
- Remove all commands of the form $(p, \bot, push_w, p')$.
- Add rules $(p, \bot, push_\bot, p_A)$ for all $p \notin R$ and $(p, \bot, push_\bot, p_E)$ for all $p \in R$.

That is, if the stack is emptied, play moves to a sink state from which Éloïse wins if the stack was emptied at a control state in $R$, and Abelard wins otherwise. Let $W_E(R)$ be Éloïse's winning region in $G(R)$.

For each $p \in \mathcal{P}$ and $a \in \Sigma$, we define,

$$\mathcal{R}(p, a) = \{\ R \subset \mathcal{P} \mid \langle p, [a\bot] \rangle \in W_E(R) \text{ and } \langle p, [a\bot] \rangle \notin W_E(R') \text{ for any } R' \subsetneq R\ \}$$

Intuitively, $\mathcal{R}(p, a)$ gives the minimal constraints Éloïse must satisfy in order to win a play of $G$. That is,

**Proposition 4.6.1** ([92])**.** *Let $w \in \Sigma^*$, $p \in \mathcal{P}$ and $a \in \Sigma$, then we have $\langle p, [aw\bot] \rangle \in W_E$ if and only if there exists some $R \in \mathcal{R}(p, a)$ such that $\langle p', [w\bot] \rangle \in W_E$ for all $p' \in R$.*

We can use Walukiewicz's algorithm to construct $\mathcal{R}(p, a)$ for all $p \in \mathcal{P}$ and $a \in \Sigma$.

### Constructing the Winning Regions

We can use $\mathcal{R}(p, a)$ to define an automaton accepting Éloïse's winning region.

**Definition 4.6.1** ([92])**.** Given a pushdown parity game $G = (\mathcal{P}, \mathcal{D}, \Sigma, \Omega)$, we define an alternating automaton $A_G = (\mathcal{P}, \Sigma, \delta, \mathcal{F})$ where, for every $p \in \mathcal{P}$ and $a \in \Sigma$,

$$\delta(p, a) = \bigvee_{R \in \mathcal{R}(p,a)} \bigwedge_{p' \in R} p'$$

and $\delta(p, \bot) = p$. Furthermore $\mathcal{F} = \{\ p \in \mathcal{P} \mid \langle p, [\bot] \rangle \in W_E\ \}$.

In the above definition the first transition of $A_G$ given a configuration $\langle p, [aw\,\bot] \rangle$ uses $\mathcal{R}(p, a)$ to find the range of conditions, according to Proposition 4.6.1, that must hold when the $a$ character is popped. The disjunction in the transition relation requires that one possible condition is met. The conjunction ensures that a particular condition $R$ is met, by checking that all configurations $\langle p', [w\bot] \rangle$ with $p' \in R$ are in the winning region of Éloïse.

**Theorem 4.6.2** ([92])**.** *For any two-player game on a pushdown graph with a parity winning condition, one can construct an alternating automaton $A_G$ that recognises the set $W_E$ of winning positions for Éloïse. Moreover, $A_G$ gives an exponential-time procedure to decide whether a configuration is winning for Éloïse.*

Finally, Serre shows that this theorem can be generalised to any $\omega$-regular winning condition. This is achieved via a reduction to parity conditions.

### 4.6.3 Two-Way Alternating Tree Automata

An alternative automata-theoretic approach to model-checking infinite state systems was introduced by Kupferman and Vardi in 2000 [89]. This approach was originally applied to context-free graphs and prefix-recognisable graphs. This approach was later generalised by Kupferman, Piterman and Vardi to pushdown systems and prefix-recognisable systems respectively [90, 86].

#### Two-Way Automata

A two-way alternating tree automaton operates over labelled trees. In addition to the expected constructs for tree navigation, a two-way automaton is able to move both up and down branches. Over a tree $(\Sigma^*, \tau)$ recall $\tau$ labels each node of the tree with a member of some alphabet $\Gamma$. In the following a node $a_0 \ldots a_m$ corresponds to a stack $a_m \ldots a_0$ and the function $\tau$ simply associates each stack with its top element. A two-way tree automaton can mimic a pushdown command $(_, a_m, push_w, _)$ as follows: the automaton moves one branch up the tree, to the node $a_0 \ldots a_{m-1}$, and then down the tree to the node $a_0 \ldots a_{m-1} w^{-1}$, where $w^{-1}$ is the string $w$ reversed. This sequence of actions simulates the application of a pushdown command. When using these automata to describe pushdown systems, we can store the current control state of the pushdown system in the state of the tree automaton. In this way, a run of the tree-automaton corresponds to a run of the pushdown system and tree automata can be used for both local and global model-checking.

**Definition 4.6.2** (Two-Way Alternating Parity Tree Automata)**.** A **two-way alternating parity tree automaton** (2APT) is a tuple $\mathcal{A} = (\Sigma, \mathcal{Q}, q_0, \Delta, \Omega)$ where $\Sigma$ is an input alphabet, $\mathcal{Q}$ is a finite state-set, with $q_0 \in \mathcal{Q}$ being the initial state, and $\Omega : \mathcal{Q} \to \{1, \ldots, m\}$ is a priority function. Finally, $\Delta : \mathcal{Q} \times \Gamma \to \mathcal{B}^+(ext(\Sigma) \times \mathcal{Q})$ is the transition function, where $ext(\Sigma) = \Sigma \cup \{\varepsilon, \uparrow\}$.

A run of a 2APT $\mathcal{A}$ over a tree $(\Sigma^*, \tau)$ is a labelled tree $(T_r, r)$ where $r$ associates each node of $T_r$ with a pair $(w, q)$ where $w \in \Sigma^*$ and $q \in \mathcal{Q}$ and,

1. $\varepsilon \in T_r$ and $r(\varepsilon) = (\perp, q_0)$.
2. For all $y \in T_r$ with $r(y) = (w, q)$ and $\Delta(q, \tau(w)) = \theta$, there exists a set $S \subseteq ext(\Sigma) \times \mathcal{Q}$ satisfying $\theta$ such that, for all $(a, q') \in S$, there exists $x \in \Sigma$ with $xy \in T_r$ and,
   - If $a \in \Sigma$, then $r(xy) = (aw, q')$.
   - If $a = \varepsilon$, then $r(xy) = (w, q')$.
   - If $a = \uparrow$, then $w = bw'$ for some $b \in \Sigma$ and $w' \in \Sigma^*$ and $r(xy) = (w', q')$.

The run is accepting if all infinite paths, when projected to the state-set $\mathcal{Q}$, satisfy the parity condition. 2APT form the basis of the model-checking algorithm. In particular, we can construct the set of all nodes of a tree from which $\mathcal{A}$ is satisfied.

**Theorem 4.6.3** ([86])**.** *Given a 2APT $\mathcal{A}$ and a regular tree $T = (\Sigma^*, \tau)$, we can construct a non-deterministic finite word automaton $N$ which accepts the word $w$ from a state $q$ iff $\mathcal{A}$ accepts $T$ from $(q, w)$.*

The above result is reached as follows: we add an idle self-loop to the beginning of $\mathcal{A}$ — allowing $\mathcal{A}$ to skip to any marked node in the tree. We combine the resulting automaton with one that accepts the regular tree $T$ with a single marked node to form an automaton $\mathcal{A}'$. Thus $\mathcal{A}'$ accepts any tree whose marked node satisfies $\mathcal{A}$. An analysis of when $\mathcal{A}'$ is non-empty is used to construct the required automaton $N$.

An intermediate step in this algorithm is the reduction of a two-way automaton to a one way automaton. This reduction involves the construction of an automaton $B$ whose input alphabet is constructed from subsets of the states of $\mathcal{A}$ and the priorities of $\mathcal{A}$, and is thus exponential. This automaton is then complemented and determinised, leading to an exponential blow up in the number of states.

#### Model-Checking Pushdown Systems

The global model-checking problem for pushdown automata can be reduced to the global membership problem for 2APT. We begin by identifying the specification language.

An alternating **graph automaton** runs over graphs and is similar to a 2APT. The transition relation is of type $\Delta : \mathcal{Q} \times \Gamma \rightarrow \mathcal{B}^+(\{\varepsilon, \Box, \Diamond\} \times \mathcal{Q})$. Like a 2APT, runs of the automaton are labelled trees. A node in the tree is a pair $(s, q)$ where $s$ is a node in the graph and $q$ is the current state of the automaton. A proposition $(\Box, q')$ requires that the run node has a child $(s', q')$ for every successor $s'$ of $s$. Dually, $(\Diamond, q')$ requires only a single child $(s', q')$ for some successor of $s$. Note that these automata are not two-way.

**Theorem 4.6.4** ([136])**.** *Given a $\mu$-calculus formula $\varphi$, of length $n$ and alternation depth $k$, we can construct a graph parity automaton $\mathcal{S}_\varphi$ such that $\mathcal{L}(\mathcal{S}_\varphi)$ is exactly the set of graphs satisfying $\varphi$. The automaton $\mathcal{S}_\varphi$ has $n$ states and index $k$.*

Finally, we can combine a 2APT simulating a pushdown automaton with a graph automaton specification to reduce the global model-checking problem to global membership of a 2APT.

**Theorem 4.6.5** ([86])**.** *Given a pushdown system $R = (\mathcal{P}, \mathcal{D}, \Sigma)$ with a labelling function $L : \mathcal{P} \times C_1^\Sigma \rightarrow \Gamma$ and a graph automaton $\mathcal{S} = (\Gamma, \mathcal{Q}, q_0, \Delta, \Omega)$, we can construct a 2APT $\mathcal{A}$ on $\Sigma$-trees and a function $f$ that associates states of $\mathcal{A}$ with states of $R$ such that $\mathcal{A}$ accepts $(\Sigma^*, \tau)$ from $(q, w)$ iff the configuration $\langle f(q), [w] \rangle$ satisfies $\mathcal{S}$.*

Similar results are also obtained for LTL specifications. In both cases, the algorithm runs in exponential time and space.

### 4.6.4 Comparing the Approaches

The main advantage of our approach is that it is direct and simple to describe. Conversely, the tree-automata approach of Vardi *et al.* requires several intermediate steps: a reduction

from two-way alternating tree-automata to alternating tree-automata, and then the construction of a strategy automaton which involves the complementation of Büchi automata. Serre's algorithm requires the construction of a number of sub-games, each of which requires Walukiewicz's reduction to finite state games (and the invocation of a finite state algorithm). The solutions to these sub-problems then need to be interpreted appropriately. Although it is more direct than the other existing approaches, Cachat's technique still requires a use of Walukiewicz's reduction to a finite state algorithm.

Cachat gives the shortest algorithm for computing the winning regions of an order-1 pushdown parity game, given the correctness of Walukiewicz's reduction. However, the proof is only a sketch. Serre's approach gives a more thorough treatment, and an extension to any $\omega$-regular winning condition. This approach also permits an elegant generalisation to order-$n$ parity games, which is described in Section 4.7. Finally, Vardi *et al.* give a pleasing interpretation of pushdown games using tree-automata, although the solution to the tree-automata problem is difficult. Their approach also unifies the global model-checking problems for context-free graphs, prefix-recognisable graphs, pushdown systems and prefix-recognisable systems.

The existing algorithms are all immediately exponential. That is, there are no "fortunate" cases where the full exponential cost may be avoided. However, by using the notion of $\text{EXPAND}(A)$, we were able to begin the algorithm with a polynomially sized automaton. Hence, we do not immediately pay the exponential blow-up. Hence, it is possible that, in some "fortunate" cases, we may avoid the full cost of the problem.

The main drawback of our approach is that, unlike the alternative approaches, we do not know how to construct winning strategies for the players of the game. The construction of strategies is useful in providing counter-examples and performing program synthesis. Therefore, this remains a pressing area of future work.

## 4.7 Abstract Pushdown Games and Higher-Order Pushdown Systems

Recently, Serre has generalised his algorithm to *abstract pushdown games* [12]. These games generalise pushdown automata by allowing an infinite stack alphabet. Since the alphabet is infinite, it can be used to encode the stacks of a higher-order PDS. Consequently, abstract pushdown games are a generalisation of higher-order pushdown systems.

**Definition 4.7.1.** An **abstract pushdown process** is a tuple $(\mathcal{P}, \mathcal{D}, \Sigma)$ where $\mathcal{P}$ is a finite set of control states, $\Sigma$ is a (possibly infinite) *abstract pushdown alphabet*, and,

$$\mathcal{D} : \mathcal{P} \times \Sigma \rightarrow 2^{(\{rew(q,\Sigma),\ pop(q),\ push(q,\gamma)\ \mid\ q \in \mathcal{Q}, \gamma \in \Gamma\})}$$

Configurations of an abstract pushdown process are of the form $\langle p, [w] \rangle \in \mathcal{P} \times \Sigma^*$, where $p$ is a control state and $w$ is a stack. The commands $rew(q, \gamma)$, $pop(q)$ and $push(q, \gamma)$ change the control state to $q$ and rewrite the top of the stack symbol to $\gamma$, remove the top stack symbol, and add $\gamma$ to the top of the stack respectively. Furthermore, we assume $\Sigma$ contains a bottom-of-the-stack symbol $\perp$ which is neither pushed onto, nor popped from, the stack.

A parity game over an abstract pushdown process is analogous to a parity game played over a higher-order PDS: the set $\mathcal{P}$ is partitioned $\mathcal{P}_E \uplus \mathcal{P}_A$. If the current control state is in $\mathcal{P}_E$, Éloïse chooses the next move, otherwise Abelard is to play. To calculate the winning regions of a parity game over an abstract pushdown process, we use automata with oracles.

**Definition 4.7.2.** An **automaton with oracles** $\mathcal{A}$ is a tuple $(\mathcal{Q}, \mathcal{P}, \Sigma, \delta, q_0, \mathcal{O}_1, \ldots, \mathcal{O}_n, Acc)$ where $\mathcal{Q}$ is a finite set of states, $\mathcal{P}$ is a set of control states, $\Sigma$ is a (possibly infinite) input alphabet, $q_0 \in \mathcal{Q}$ is the initial state, $\mathcal{O}_i$ are subsets of $\Sigma$ (called *oracles*) and $\delta : \mathcal{Q} \times \{0,1\}^n \rightarrow \mathcal{Q}$ is the transition function. Finally $Acc$ is a function from $\mathcal{Q}$ to $2^{\mathcal{P}}$.

A configuration $\langle p, [\gamma_m \ldots \gamma_0]\rangle$ is accepted by an automaton $\mathcal{A}$ iff there is a run $q_0, \ldots, q_m$ with $q_{i+1} \in \delta(q_i, \mathcal{O}_1(\gamma_i), \ldots, \mathcal{O}_n(\gamma_i))$ for all $i \in \{0, \ldots, m-1\}$ and $p \in Acc(q_m)$.

In the previous section we defined an alternating parity automaton accepting Éloïse's winning region with the transition relation,

$$\delta(p, a) = \bigvee_{R \in \mathcal{R}(p,a)} \bigwedge_{p' \in R} p'$$

and $\delta(p, \perp) = p$. In a similar manner, we define an automaton with oracles accepting Éloïse's winning region of an abstract pushdown game. We define an oracle $\mathcal{O}_{p,R}$ for every $p \in \mathcal{P}$ and $R \subseteq \mathcal{P}$ with $\mathcal{O}_{p,R}(\gamma) = 1$ iff $R \in \mathcal{R}(p, \gamma)$. These oracles take the place of the disjunction in the transition relation given above. The conjunction is represented via a subset construction.

**Theorem 4.7.1.** *Let $G$ be an* abstract pushdown parity game *over an abstract pushdown process $(\mathcal{P}, \mathcal{D}, \Sigma)$. The winning region of Éloïse is accepted by an automaton with oracles $\mathcal{A} = (2^{\mathcal{P}}, \mathcal{P}, \Sigma, \delta, \emptyset, \mathcal{O}_1, \ldots, \mathcal{O}_n, Acc)$ such that,*

- *For every $p \in \mathcal{P}$ and $R \subseteq \mathcal{P}$ there is an oracle $\mathcal{O}_{p,R}$ with $\mathcal{O}_{p,R}(\gamma)$ iff $R \in \mathcal{R}(p, \gamma)$ and $\gamma \neq \perp$. Furthermore there is an oracle $\mathcal{O}_\perp$ with $\mathcal{O}_\perp(\gamma) = 1$ iff $\gamma = \perp$.*
- *Using the oracles, $\delta$ behaves as follows,*
  - *From state $\emptyset$ on input $\perp$, the next state is $\{ p \mid \langle p, [\perp]\rangle$ is winning for Éloïse in $G$ $\}$,*
  - *From state $q \in 2^{\mathcal{P}}$ on input $\gamma$, the next state is $\{ p \mid q \in \mathcal{R}(p, \gamma) \}$.*
- *$Acc(p) = p$.*

If a language is accepted by an automaton whose oracles are regular languages, the language is itself regular. Furthermore, if the sets $\{ \gamma \mid R \in \mathcal{R}(p, \gamma) \}$ are regular, the oracles $\mathcal{O}_{p,R}(\gamma) = 1$ iff $R \in \mathcal{R}(p, \gamma)$ are regular, and, hence, the winning regions of the parity game are regular.

The sets $\mathcal{R}(p, \gamma)$ can be calculated, as in the previous section, using conditional games $G(R)$. Using the construction in Section 2.8.5, an order-$n$ conditional game can be reduced to an order-$(n-1)$ game $G'(R)$.

**Theorem 4.7.2.** *For every $p \in \mathcal{P}$, $\gamma \in \Sigma$ and $R \subseteq P$, we have $R \in \mathcal{R}(p, \gamma)$ iff the configuration $Check((R, \ldots, R), \gamma, m, m, p)$ is winning for Éloïse in $G'(R)$, where $c$ is the priority of $p$.*

Therefore, if the winning regions of the order-$(n-1)$ game are regular, the sets $\{ \gamma \mid R \in \mathcal{R}(p, \gamma) \}$ are regular, and hence, the winning regions of the original game are regular. In the previous sections we saw that the winning regions of an order-1 pushdown parity game are regular. By an inductive argument, and since higher-order pushdown games are a special case of abstract pushdown games, we have that the winning regions of a higher-order pushdown parity game are regular.

**Theorem 4.7.3.** *The winning regions of an order-$n$ pushdown parity game are regular and can be effectively computed in $n$-EXPTIME.*

## 4.8 Summary

We have proposed a new, direct, and easy to describe algorithm for computing the winning regions of an order-1 pushdown parity game. This algorithm extends the saturation techniques for reachability, due to Bouajjani *et al.* [3] and Finkel *et al.* [15], and for Büchi games, due to Cachat [127]. This algorithm uses a modal $\mu$-calculus formula (given by Walukiewicz [53]) that characterises Éloïse's winning region, and uses it to guide the construction of an alternating automaton accepting Éloïse's winning region.

We then described three existing approaches to the same problem, which are based on applications of Walukiewicz's original reduction from order-1 pushdown parity games to finite state parity games (Serre [92] and Cachat [127]) or an interpretation of pushdown systems using trees and two-way alternating tree-automata (Vardi *et al.* [90, 86, 85]). In Section 4.6.4 we gave a brief comparison of the available techniques.

Finally in Section 4.7 we described Serre's extension of his technique to order-$n$ pushdown games.

# Chapter 5

# Global Reachability Analysis of Higher-Order Pushdown Systems

In this chapter, we discuss the non-trivial problem[1] of extending the backwards reachability result of Bouajjani and Meyer to the general case of higher-order PDSs (by taking into account a set of control states). In fact, we consider (and solve) the backwards reachability problem for the more general case of higher-order *alternating* pushdown systems (APDSs). Following the work of Cachat [127], we show that the winning region of a reachability game played over a higher-order PDS can be computed by a reduction to the backwards reachability problem of an appropriate APDS. Furthermore, by reducing the non-emptiness problem for an order-$(n+1)$ higher-order PDA to a reachability problem over order-$n$ APDS, we show that our algorithm is optimal. This work was first published in FoSSaCS 2007 [77] and in the FoSSaCS 2007 special issue of LMCS [78]. An alternative construction, presented by Seth [23], for the order-2 case is discussed in Section 5.5.

The algorithm uses a form of nested automata to represent configurations and uses a similar routine of adding transitions determined by the transition relation of the higher-order APDS. However, naïve combinations of the multi-automaton and nested-store automaton techniques do not lead to satisfactory solutions. During our own efforts with simple combined techniques, it was unclear how to form the product of two automata and maintain a distinction between the different control states as required. To perform such an operation safely it seemed that additional states were required on top of those added by the basic product operation, invalidating the termination arguments. We overcome this problem by using alternating automata and by modifying the termination argument. Additionally, we reduce the complexity of Bouajjani and Meyer from a tower of exponentials twice the size of $n$, to a tower of exponentials as large as $n$. Hence, the problem is $n$-EXPTIME-complete.

The nesting of the automata reflects the nesting of a higher-order stack: an automaton accepting order-$n$ stacks has $n$ layers of nesting. We refer to these layers from the outer layer to the inner layer as order-$n$ to order-one of the automaton. Termination of the reachability algorithm is reached through a cascading of fixed points. Given a (nested) store-automaton, we fix the order-$n$ state-set, but allow the state-sets at orders less than $n$ to change. During a number of iterations, we add a finitely bounded number of new transitions to order-$n$ of the automaton. If a transition has already been added between a state and a set of states, we

[1] "This does not seem to be technically trivial, and naïve extensions of our construction lead to procedures which are not guaranteed to terminate." [2, p. 145]

update the automata labelling the transition, rather than add a new transition. Eventually we reach a stage where no new transitions are being added at order-$n$, although changes will continue to occur at lower orders. At this point the updates become repetitive and we are able to freeze the state-set at the second highest order. This is done by adding possibly cyclical transitions between the existing states, instead of a chain of transitions between an unbounded number of new states (see Figure 5.6). Because the order-$(n-1)$ state-set has been fixed, we reach another fixed point where no new order-$(n-1)$ transitions are added. This is similar to first fixed point at order-$n$. In this way the fixed points cascade to order-one, where the finite alphabet ensures that the automaton eventually becomes saturated. We are left with an automaton representing the set $Pre^*(C_{Init})$.

In the sequel, to ease the presentation, we assume $n > 1$. The case $n = 1$ was investigated by Bouajjani *et al.* [3].

## 5.1 $n$-Store Multi-Automata

To represent sets of configurations symbolically we will use *$n$-store multi-automata*. These are alternating automata whose transitions are labelled by *$(n-1)$-store automata*, which are also alternating. A set of configurations is *regular* iff it can be represented using an $n$-store multi-automaton. This notion of regularity coincides with the definition of Bouajjani and Meyer (see Section 5.2). In Section 5.6 we give algorithms for enumerating runs of $n$-store automata, testing membership and performing boolean operations on the automata.

**Definition 5.1.1.**

1. A **1-store automaton** is a tuple $(\mathcal{Q}, \Sigma, \Delta, q_0, \mathcal{Q}_f)$ where $\mathcal{Q}$ is a finite set of states, $\Sigma$ is a finite alphabet, $q_0$ is the initial state and $\mathcal{Q}_f \subseteq \mathcal{Q}$ is a set of final states and $\Delta \subseteq \mathcal{Q} \times \Sigma \times 2^{\mathcal{Q}}$ is a finite transition relation.

2. Let $\mathfrak{B}^{\Sigma}_{n-1}$ be the (infinite) set of all $(n-1)$-store automata over the alphabet $\Sigma$. An $n$**-store automaton** over the alphabet $\Sigma$ is a tuple $(\mathcal{Q}, \Sigma, \Delta, q_0, \mathcal{Q}_f)$ where $\mathcal{Q}$ is a finite set of states, $q_0 \notin \mathcal{Q}_f$ is the initial state, $\mathcal{Q}_f \subseteq \mathcal{Q}$ is a set of final states, and $\Delta \subseteq \mathcal{Q} \times \mathfrak{B}^{\Sigma}_{n-1} \times 2^{\mathcal{Q}}$ is a *finite* transition relation. Furthermore, let $\mathfrak{B}^{\Sigma}_0 = \Sigma$.

3. An $n$**-store multi-automaton** over the alphabet $\Sigma$ is a tuple

$$(\mathcal{Q}, \Sigma, \Delta, \{q^1, \ldots, q^z\}, \mathcal{Q}_f)$$

where $\mathcal{Q}$ is a finite set of states, $\Sigma$ is a finite alphabet, $q^i$ for $i \in \{1, \ldots, z\}$ are pairwise distinct initial states with $q^i \notin \mathcal{Q}_f$ and $q^i \in \mathcal{Q}$; $\mathcal{Q}_f \subseteq \mathcal{Q}$ is a set of final states, and,

$$\Delta \subseteq (\mathcal{Q} \times \mathfrak{B}^{\Sigma}_{n-1} \times 2^{\mathcal{Q}}) \cup (\{q^1, \ldots, q^z\} \times \{\nabla\} \times \{q^{\varepsilon}_f\})$$

is a *finite* transition relation where $q^{\varepsilon}_f \in \mathcal{Q}_f$ has no outgoing transitions.

To indicate a transition $(q, B, \{q_1, \ldots, q_m\}) \in \Delta$ we write,

$$q \xrightarrow{B} \{q_1, \ldots, q_m\}$$

A transition of the form $q^j \xrightarrow{\triangledown} \{q_f^\varepsilon\}$ indicates that the undefined configuration $\langle p^j, \triangledown \rangle$ is accepted. Runs of the automata from a state $q$ take the form,

$$q \xrightarrow{\widetilde{B}_0} \{q_1^1, \ldots, q_{m_1}^1\} \xrightarrow{\widetilde{B}_1} \ldots \xrightarrow{\widetilde{B}_m} \{q_1^{m+1}, \ldots, q_{m_l}^{m+1}\}$$

where transitions between configurations $\{q_1^x, \ldots, q_{m_x}^x\} \xrightarrow{\widetilde{B}_x} \{q_1^{x+1}, \ldots, q_{m_{x+1}}^{x+1}\}$ are such that we have $q_y^x \xrightarrow{B_y} Q_y$ for all $y \in \{1, \ldots, m_x\}$ and $\bigcup_{y \in \{1,\ldots,m_x\}} Q_y = \{q_1^{x+1}, \ldots, q_{m_{x+1}}^{x+1}\}$ and additionally $\bigcup_{y \in \{1,\ldots,m_x\}} \{B_y\} = \widetilde{B}_x$. Observe that $\widetilde{B}_0$ is necessarily a singleton set. A run over a word $\gamma_1 \ldots \gamma_m$, denoted $q \xrightarrow{\gamma_1 \cdots \gamma_m} Q$, exists whenever,

$$q \xrightarrow{\widetilde{B}_0} \ldots \xrightarrow{\widetilde{B}_m} Q$$

and for all $0 \leq i \leq m$, $\gamma_i \in \mathcal{L}(\widetilde{B}_i)$, where $\gamma \in \mathcal{L}(\widetilde{B})$ iff $\gamma \in \mathcal{L}(B)$ (defined below) for all $B \in \widetilde{B}$. If a run occurs in an automaton forming part of a sequence of automata $A_0, A_1, \ldots$, we may write $\longrightarrow_i$ to indicate which automaton $A_i$ the run belongs to.

We define $\mathcal{L}(a) = a$ for all $a \in \Sigma = \mathfrak{B}_0^\Sigma$. An $n$-store $[\gamma_1 \ldots \gamma_m]$ is accepted by an $n$-store automaton $A$ (that is $[\gamma_1 \ldots \gamma_m] \in \mathcal{L}(A)$) iff we have a run $q_0 \xrightarrow{\gamma_1 \cdots \gamma_m} Q$ in $A$ with $Q \subseteq \mathcal{Q}_f$. For a given $n$-store multi-automaton $A = (Q, \Sigma, \Delta, \{q^1, \ldots, q^z\}, \mathcal{Q}_f)$ we define,

$$\begin{array}{rcl} \mathcal{L}(A^{q^j}) & = & \{ \ [\gamma_1 \ldots \gamma_m] \mid q^j \xrightarrow{\gamma_1 \cdots \gamma_m} Q \ \wedge \ Q \subseteq \mathcal{Q}_f \ \} \\ & & \cup \ \{ \ \triangledown \mid q^j \xrightarrow{\triangledown} \{q_f^\varepsilon\} \ \} \end{array}$$

and

$$\mathcal{L}(A) = \{ \ \langle p^j, \gamma \rangle \mid j \in \{1, \ldots, z\} \wedge \gamma \in \mathcal{L}(A^{q^j}) \ \}$$

Finally, we define the automata $B_l^a$ and $X_l^a$ for all $1 \leq l \leq n$ and $a \in \Sigma$ and the notation $q^\theta$. The $l$-store automaton $B_l^a$ accepts any $l$-store $\gamma$ such that $top_1(\gamma) = a$. The $(n-1)$-store automaton $X_l^a$ accepts all $(n-1)$-stacks such that $top_1(\gamma) = a$ and $top_{l+1}(\gamma) = [[w']]$ for some $w'$. That is, $pop_l(\gamma)$ is undefined. If $\theta$ represents a store automaton, the state $q^\theta$ refers to the initial state of the automaton represented by $\theta$.

## 5.2 Regularity

We show that our notion of a regular set of $n$-stores coincides with the definition of Bouajjani and Meyer [2]. Because we are considering $n$-stores rather than configurations, we assume that there is only one control state, and hence, an $n$-store multi-automaton has only a single initial state. We also disregard the undefined store $\triangledown$, since it is not strictly a store. Observe that we are left with $n$-store automata.

In the absence of alternation, the set of $n$-store automata is definitionally equivalent to the set of level $n$ nested store automata in the sense of Bouajjani and Meyer. Hence, it is the case that every level $n$ nested store automaton is also an $n$-store automaton.

We need to prove that every $n$-store automaton has an equivalent level $n$ nested store automata. We present the following definition:

**Definition 5.2.1.** Given an $n$-store automaton $A = (\mathcal{Q}, \Sigma, \Delta, q_0, \mathcal{Q}_f)$ we define a level $n$ nested store automaton $\hat{A} = (2^{\mathcal{Q}}, \Sigma, \hat{\Delta}, \{q_0\}, 2^{\mathcal{Q}_f})$, where, if $n = 1$,

$$\hat{\Delta} = \{ \ (\{q_1, \ldots, q_m\}, a, Q') \mid \forall i \in \{1, \ldots, m\}. (\exists (q_i, a, Q_i) \in \Delta) \wedge Q' = Q_1 \cup \cdots \cup Q_m \ \}$$

and if $n > 1$,

$$\hat{\Delta} = \left\{ (\{q_1, \ldots, q_m\}, \hat{B}, Q') \mid \begin{array}{l} \forall i \in \{1, \ldots, m\}. (\exists (q_i, B_i, Q_i) \in \Delta) \wedge \\ Q' = Q_1 \cup \cdots \cup Q_m \wedge B = B_1 \cap \cdots \cap B_m \end{array} \right\}$$

where $\hat{B}$ is defined recursively and the construction of $B_1 \cap \cdots \cap B_m$ is given in section 5.6.3.

**Property 5.2.1.** *For any $w$, the run $\{q_1, \ldots, q_m\} \xrightarrow{w} Q'$ exists in the $n$-store automaton $A$ iff the run $\{q_1, \ldots, q_m\} \xrightarrow{w} Q'$ exists in $\hat{A}$.*

*Proof.* The proof is by induction over $n$ and then by a further induction over the length of $w$.

Suppose $n = 1$. When $w = \varepsilon$ the proof is immediate. When $w = aw'$ we have in one direction,

$$\{q_1, \ldots, q_m\} \xrightarrow{a} Q_1 \xrightarrow{w'} Q'$$

in $A$, and by induction over the length of the run, $Q_1 \xrightarrow{w'} Q'$ in $\hat{A}$. By definition of the runs of $A$ we have $q_i \xrightarrow{a} Q_1^i$ for each $i \in \{1, \ldots, m\}$ with $Q_1 = Q_1^1 \cup \cdots \cup Q_1^m$. Hence, by definition of $\hat{A}$ we have the transition $\{q_1, \ldots, q_m\} \xrightarrow{a} Q_1^1 \cup \cdots \cup Q_1^m = Q_1$. Hence we have the run $\{q_1, \ldots, q_m\} \xrightarrow{w} Q'$ in $\hat{A}$ as required.

In the other direction we have a run of the form

$$\{q_1, \ldots, q_m\} \xrightarrow{a} Q_1 \xrightarrow{w'} Q'$$

in $\hat{A}$, and by induction over the length of the run, $Q_1 \xrightarrow{w'} Q'$ in $A$. By definition of the transition relation of $\hat{A}$ we have $q_i \xrightarrow{a} Q_1^i$ in $A$ for each $i \in \{1, \ldots, m\}$ with $Q_1 = Q_1^1 \cup \cdots \cup Q_1^m$. Hence, we have the transition $\{q_1, \ldots, q_m\} \xrightarrow{a} Q_1^1 \cup \cdots \cup Q_1^m = Q_1$ in $A$. Thus, we have the run $\{q_1, \ldots, q_m\} \xrightarrow{w} Q'$ in $A$ as required.

When $n > 1$, when $w = \varepsilon$ the proof is immediate. When $w = \gamma w'$ we have in one direction,

$$\{q_1, \ldots, q_m\} \xrightarrow{\gamma} Q_1 \xrightarrow{w'} Q'$$

in $A$, and by induction over the length of the run, $Q_1 \xrightarrow{w'} Q'$ in $\hat{A}$. By definition of the runs of $A$ we have $q_i \xrightarrow{B_i} Q_1^i$ with $\gamma \in \mathcal{L}(B_i)$ for each $i \in \{1, \ldots, m\}$ with $Q_1 = Q_1^1 \cup \cdots \cup Q_1^m$. Consequently, we have $\gamma \in \mathcal{L}(B)$ where $B = B_1 \cap \cdots \cap B_m$. By induction over $n$ we have $\gamma \in \mathcal{L}(\hat{B})$. Hence, by definition of $\hat{A}$ we have the transition $\{q_1, \ldots, q_m\} \xrightarrow{\gamma} Q_1^1 \cup \cdots \cup Q_1^m = Q_1$. Hence we have the run $\{q_1, \ldots, q_m\} \xrightarrow{w} Q'$ in $\hat{A}$ as required.

In the other direction we have a run of the form

$$\{q_1, \ldots, q_m\} \xrightarrow{\gamma} Q_1 \xrightarrow{w'} Q'$$

in $\hat{A}$. In particular, we have $\{q_1, \ldots, q_m\} \xrightarrow{\hat{B}} Q_1$ in $\hat{A}$ with $\gamma \in \mathcal{L}(\hat{B})$. By induction over the length of the run, $Q_1 \xrightarrow{w'} Q'$ in $A$. By definition of the transition relation of $\hat{A}$ we have $q_i \xrightarrow{B_i} Q_1^i$ for each $i \in \{1, \ldots, m\}$ with $B = B_1 \cap \cdots \cap B_m$ and $Q_1 = Q_1^1 \cup \cdots \cup Q_1^m$. By induction over $n$ we have $\gamma \in \mathcal{L}(B)$ and hence $\gamma \in \mathcal{L}(B_i)$ for all $i \in \{1, \ldots, m\}$. Hence we have $q_i \xrightarrow{\gamma} Q_1^i$ in $A$ for all $i \in \{1, \ldots, m\}$. Thus, we have the transition $\{q_1, \ldots, q_m\} \xrightarrow{\gamma} Q_1^1 \cup \cdots \cup Q_1^m = Q_1$ in $A$ and the run $\{q_1, \ldots, q_m\} \xrightarrow{w} Q'$ as required. □

**Corollary 5.2.1.** *A set of $n$-stores is definable by an $n$-store automaton iff it is definable by a level $n$ nested store automaton.*

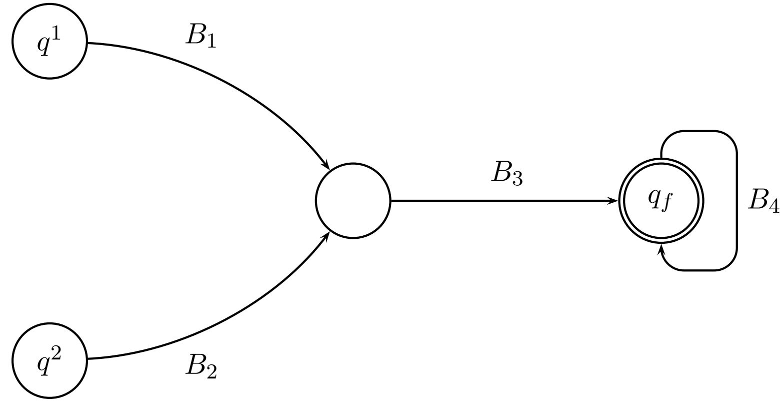



Figure 5.1: The initial 2-store multi-automaton

## 5.3 The Order-Two Case

Before introducing the full algorithm, we give an example and a description of the algorithm for the order-2 case. This will provide an introduction to the important features of the solution whilst reducing the notational complexity.

**Theorem 5.3.1.** *Given a 2-store multi-automaton $A_0$ accepting the set of configurations $C_{Init}$ of an order-2 APDS, we can construct in 2-EXPTIME (in the size of $A_0$) a 2-store multi-automaton $A_*$ accepting the set $Pre^*(C_{Init})$. Thus, $Pre^*(C_{Init})$ is regular.*

Fix an order-2 APDS. We begin by showing how to generate an infinite sequence of automata $A_0, A_1, \ldots$, where $A_0$ is such that $\mathcal{L}(A_0) = C_{Init}$. This sequence is increasing in the sense that $\mathcal{L}(A_i) \subseteq \mathcal{L}(A_{i+1})$ for all $i$, and sound and complete with respect to $Pre^*(C_{Init})$; that is $\bigcup_{i \geq 0} \mathcal{L}(A_i) = Pre^*(C_{Init})$. To conclude the algorithm, we construct a single automaton $A_*$ such that $\mathcal{L}(A_*) = \bigcup_{i \geq 0} \mathcal{L}(A_i)$.

We assume without loss of generality that all initial states in $A_0$ have no incoming transitions and there exists in $A_0$ a state $q^*_f$ from which all valid 2-stores are accepted and a state $q^\varepsilon_f \in \mathcal{Q}_f$ which has no outgoing transitions.

### 5.3.1 Example

We give an intuitive explanation of the algorithm by means of an example. Fix the following two-state order-two PDS:

$$\begin{aligned} d_1 &= (p^1, a, push_2, p^1) \\ d_2 &= (p^1, a, push_\varepsilon, p^1) \\ d_3 &= (p^2, a, push_w, p^1) \\ d_4 &= (p^2, a, pop_2, p^1) \end{aligned}$$

And a 2-store multi-automaton $A_0$ shown in Figure 5.1 with some $B_1, B_2, B_3$ and $B_4$. We proceed via a number of iterations, generating the automata $A_0, A_1, \ldots$. We construct $A_{i+1}$ from $A_i$ to reflect an additional inverse application of the commands $d_1, \ldots, d_4$.

During the construction of $A_1$, rather than manipulating the order-1 store automata labelling the edges of $A_0$ directly, we introduce new transitions (at most one between each pair of states $q_1$ and $q_2$) and label these edges with the set $\widetilde{G}^1_{(q_1,q_2)}$. This set is a recipe for the

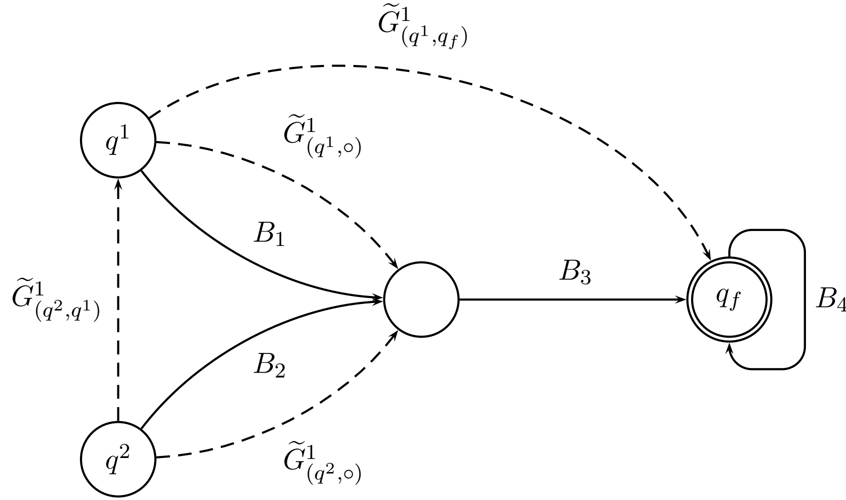



Figure 5.2: The automaton $A_1$

| | $d_1$ | $d_2$ | $d_3$ | $d_4$ |
|---|---|---|---|---|
| $\widetilde{G}^1_{(q^1,\circ)}$ | | $\{(a, push_\varepsilon, B_1)\}$ | | |
| $\widetilde{G}^1_{(q^1,q_f)}$ | $\{B_1^a, B_1, B_3\}$ | | | |
| $\widetilde{G}^1_{(q^2,\circ)}$ | | | $\{(a, push_w, B_1)\}$ | |
| $\widetilde{G}^1_{(q^2,q^1)}$ | | | | $\{B_1^a\}$ |

Table 5.1: The contents of the sets in $\widetilde{\mathcal{G}}^1$.

construction of an order-1 store automaton that will ultimately label the edge. The set $\widetilde{\mathcal{G}}^1$ is the set of all sets $\widetilde{G}^1_{(q_1,q_2)}$ introduced. After the first stage of the algorithm, the resulting $A_1$ is given in Figure 5.2 where the contents of

$$\widetilde{\mathcal{G}}^1 = \left\{\widetilde{G}^1_{(q^1,\circ)}, \widetilde{G}^1_{(q^1,q_f)}, \widetilde{G}^1_{(q^2,\circ)}, \widetilde{G}^1_{(q^2,q^1)}\right\}$$

are given in Table 5.1. The columns indicate which command introduced each element to the set.

To process the command $d_1$ we need to add to the set of configurations accepted by $A_1$ all configurations of the form $\langle p_1, [\gamma_1 \ldots \gamma_m]\rangle$ with $top_1(\gamma_1) = a$ for each configuration $\langle p_1, [\gamma_1\gamma_1 \ldots \gamma_m]\rangle$ accepted by $A_0$. This is because $push_2[\gamma_1 \ldots \gamma_m] = [\gamma_1\gamma_1 \ldots \gamma_m]$. Hence we add the transition from $q^1$ to $q_f$. The contents of $\widetilde{G}^1_{(q^1,q_f)}$ indicate that this edge must accept the product of $B_1^a$, $B_1$ and $B_3$.

The commands $d_2$ and $d_3$ update the $top_2$ stack of any configuration accepted from $q^1$ or $q^2$ respectively. In both cases this updated stack must be accepted from $q^1$ in $A_0$. Hence, the

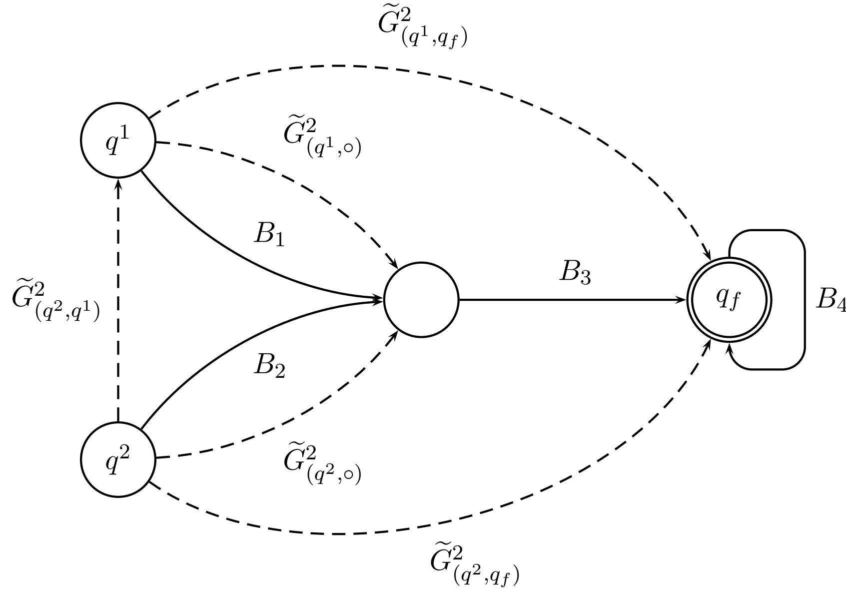



Figure 5.3: The automaton $A_2$.

contents of $\widetilde{G}^1_{(q^1,\circ)}$ and $\widetilde{G}^1_{(q^2,\circ)}$ specify that the automaton $B_1$ must be manipulated to produce the automaton that will label these new transitions. Finally, since $pop_2[\gamma_1 \ldots \gamma_m] = [\gamma_2 \ldots \gamma_m]$, $d_4$ requires an additional $top_2$ stack with $a$ as its $top_1$ element to be added to any stack accepted from $q^1$. Thus, we introduce the transition from $q^2$ to $q^1$.

To construct $A_2$ from $A_1$ we repeat the above procedure, taking into account the additional transitions in $A_1$. Observe that we do not add additional transitions between pairs of states that already have a transition labelled by a set. Instead, each labelling set may contain several element sets. The resulting $A_2$ is given in Figure 5.3 where the contents of

$$\widetilde{\mathcal{G}}^2 = \left\{ \widetilde{G}^2_{(q^1,\circ)}, \widetilde{G}^2_{(q^1,q_f)}, \widetilde{G}^2_{(q^2,\circ)}, \widetilde{G}^2_{(q^2,q^1)}, \widetilde{G}^2_{(q^2,q_f)} \right\}$$

are given in Table 5.2.

If we were to repeat this procedure to construct $A_3$ we would notice that a kind of fixed point has been reached. In particular, the transition structure of $A_3$ will match that of $A_2$ and each $\widetilde{G}^3_{(q,q')}$ will match $\widetilde{G}^2_{(q,q')}$ in everything but the indices of the labels $\widetilde{G}^1_{(_,_)}$ appearing in the element sets. We may write $\widetilde{G}^3_{(q,q')} = \widetilde{G}^2_{(q,q')}[2/1]$ where the notation $[2/1]$ indicates a substitution of the element indices.

So far we have just constructed sets to label the transitions of $A_1$ and $A_2$. To complete the construction of $A_1$ we need to construct the automata $G^1_{(q,q')}$ represented by the labels $\widetilde{G}^1_{(q,q')}$ for the appropriate $q, q'$. Because each of these new automata will be constructed from $B_1, \ldots, B_4, B^a_1$, we build them simultaneously, constructing a single (1-store multi-)automaton $\mathcal{G}^1$ with an initial state $g^1_{(q,q')}$ for each $\widetilde{G}^1_{(q,q')}$. The automaton $\mathcal{G}^1$ is constructed through the addition of states and transitions to the disjoint union of $B_1, \ldots, B_4, B^a_1$. Creating the automaton $A_2$ is analogous and $\mathcal{G}^2$ is built through the addition of states and transitions to $\mathcal{G}^1$.

The automaton $\mathcal{G}^1$ is given in Figure 5.4. We do not display this automaton in full since the number of alternating transitions entails a diagram too complicated to be illuminating.

| | $d_1$ | $d_2$ | $d_3$ | $d_4$ |
|---|---|---|---|---|
| $\widetilde{G}^2_{(q^1,\circ)}$ | | $\{(a, push_\varepsilon, B_1)\}$<br>$\{(a, push_\varepsilon, \widetilde{G}^1_{(q^1,\circ)})\}$ | | |
| $\widetilde{G}^2_{(q^1,q^f)}$ | $\{B_1^a, B_1, B_3\}$<br>$\{B_1^a, \widetilde{G}^1_{(q^1,q_f)}, B_4\}$<br>$\{B_1^a, \widetilde{G}^1_{(q^1,\circ)}, B_3\}$ | $\{(a, push_\varepsilon, \widetilde{G}^1_{(q^1,q_f)})\}$ | | |
| $\widetilde{G}^2_{(q^2,\circ)}$ | | | $\{(a, push_w, B_1)\}$<br>$\{(a, push_w, \widetilde{G}^1_{(q^1,\circ)})\}$ | |
| $\widetilde{G}^2_{(q^2,q^1)}$ | | | | $\{B_1^a\}$ |
| $\widetilde{G}^2_{(q^2,q_f)}$ | | | $\{(a, push_w, \widetilde{G}^1_{(q^1,q_f)})\}$ | |

Table 5.2: The contents of the sets in $\widetilde{\mathcal{G}}^2$.

Instead we will give the basic structure of the automaton with many transitions omitted. In particular we show a transition derived from $\{B_1^a, B_1, B_3\}$ (from state $g^1_{(q^1,q_f)}$), a transition derived from $\{(a, push_\varepsilon, B_1)\}$ (from state $g^1_{(q^1,\circ)}$) and a transition derived from $\{B_1^a\}$ (from state $g^1_{(q^2,q^1)}$).

Notably, we have omitted any transitions derived from the $push_w$ command. This is simply for convenience since we do not wish to further explicate $B_1, B_2, B_3$ or $B_4$. From this automaton we derive $G^1_{(q^1,\circ)}, G^1_{(q^1,q_f)}, G^1_{(q^2,\circ)}$ and $G^1_{(q^2,q^1)}$ by setting the initial state to $g^1_{(q^1,\circ)}, g^1_{(q^1,q_f)}, g^1_{(q^2,\circ)}$ and $g^1_{(q^2,q^1)}$ respectively.

The automaton $\mathcal{G}^2$ is shown in Figure 5.5. Again, due to the illegibility of a complete diagram, we omit many of the transitions. The new transition from $g^2_{(q^1,q_f)}$ is derived from the set $\{B_1^a, B_3, \widetilde{G}^1_{(q^1,\circ)}\}$. One of the transitions from $g^2_{(q^1,\circ)}$ and the only transition from $g^2_{(q^2,q^1)}$ are inherited from their corresponding states in the previous automaton. This inheritance ensures that we do not lose information from the previous iteration. The uppermost transition from $g^2_{(q^1,\circ)}$ derives from $\{(a, push_\varepsilon, \widetilde{G}^1_{(q^1,\circ)})\}$. From this automaton we derive $G^1_{(q^1,\circ)}, G^1_{(q^1,q_f)}, G^1_{(q^2,\circ)}$ and $G^1_{(q^2,q^1)}$.

We have now constructed the automata $A_1$ and $A_2$. We could then repeat this procedure to generate $A_3, A_4, \ldots$, resulting in an infinite sequence of automata that is sound and complete with respect to $Pre^*(\mathcal{L}(A_0))$.

To construct $A_*$ such that $\mathcal{L}(A_*) = \bigcup_{i\geq 0} \mathcal{L}(A_i)$ we observe that since a fixed point was reached at $A_2$, the update to each $\mathcal{G}^i$ to create $\mathcal{G}^{i+1}$ will use similar recipes and hence become repetitive. This will lead to an infinite chain with an unvarying pattern of edges. This chain can be collapsed as shown in Figure 5.6.

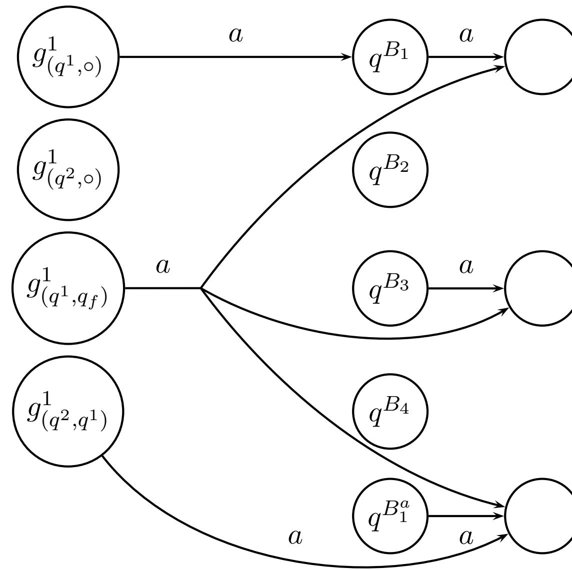



Figure 5.4: A selective view of $\mathcal{G}^1$.

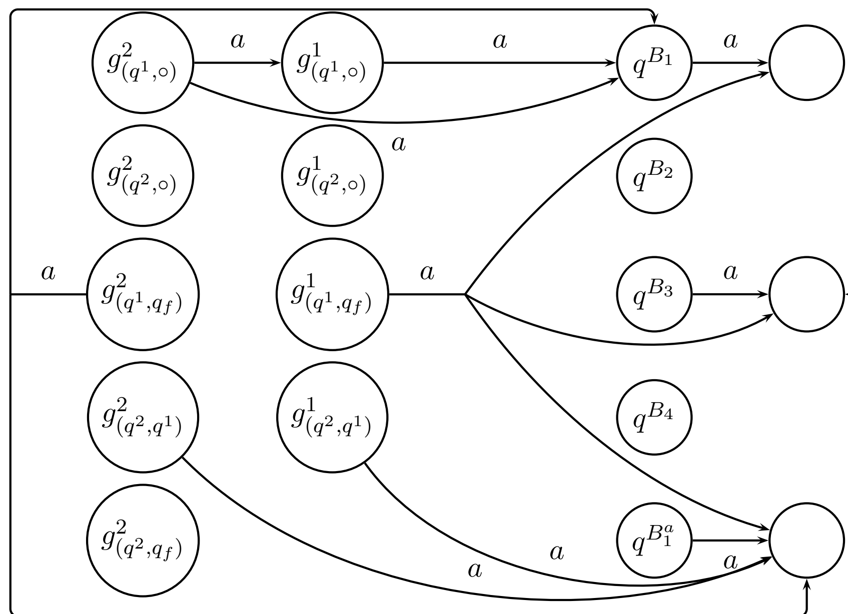



Figure 5.5: A selective view of $\mathcal{G}^2$.

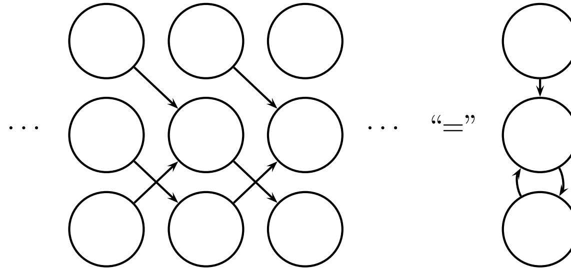



Figure 5.6: Collapsing a repetitive chain of new states.

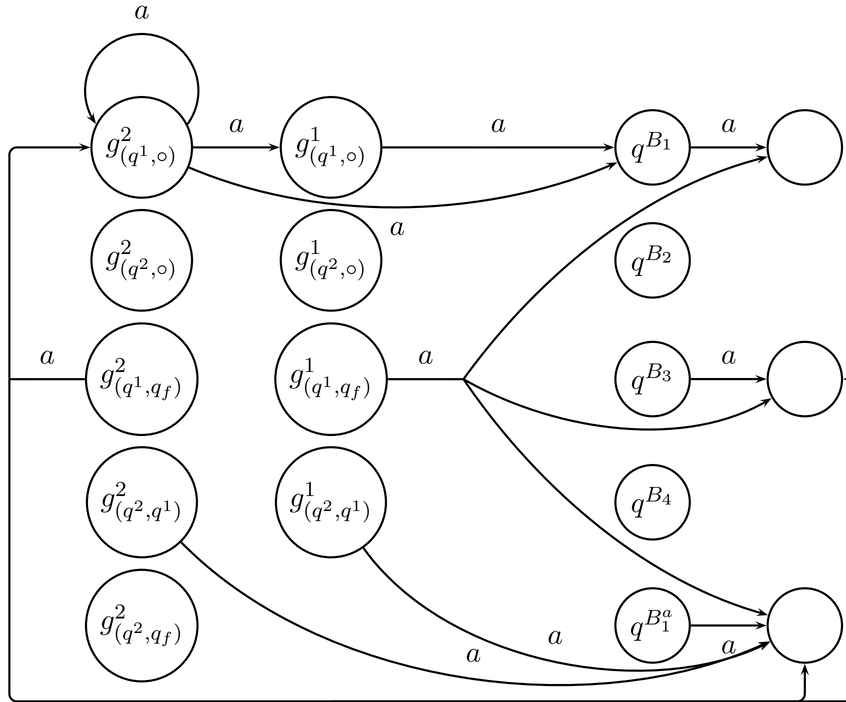



Figure 5.7: A selective view of $\hat{\mathcal{G}}^*$.

In particular, we are no longer required to add new states to $\mathcal{G}^2$ to construct $\mathcal{G}^i$ for $i > 2$. Instead, we fix the update instructions $\widetilde{G}^2_{(q,q')}[2/1]$ for all $q, q'$ and manipulate $\mathcal{G}^2$ as we manipulated the order-2 structure of $A_0$ to create $A_1$ and $A_2$. We write $\hat{\mathcal{G}}^i$ to distinguish these automata from the automata $\mathcal{G}^i$ generated without fixing the state-set.

Because $\Sigma$ and the state-set are finite (and remain unchanged), this procedure will reach another fixed point $\hat{\mathcal{G}}^*$ when the transition relation is *saturated* and $\hat{\mathcal{G}}^i = \hat{\mathcal{G}}^{i+1}$. The automaton $A_*$ has the transition structure that became fixed at $A_2$ labelled with automata derived from $\hat{\mathcal{G}}^*$. This automaton will be sound and complete with respect to $Pre^*(\mathcal{L}(A_0))$.

An abbreviated diagram of $\hat{\mathcal{G}}^*$ is given in Figure 5.7. We have hidden, for clarity, the transition derived from $\{B^a_l, B_3, \widetilde{G}^1_{(q^1,\circ)}\}$ in Figure 5.5. Instead, we show the transition introduced for the set $\{B^a_1, B_3, \widetilde{G}^1_{(q^1,\circ)}\}[2/1] = \{B^a_1, B_3, \widetilde{G}^2_{(q^1,\circ)}\}$ during the construction of $\hat{\mathcal{G}}^*$. We have also added the self-loop added by $\{(a, push_\varepsilon, G^1_{(q^1,\circ)})\}[2/1] = \{(a, push_\varepsilon, G^2_{(q^1,\circ)})\}$ that enabled the introduction of this transition.

### 5.3.2 Preliminaries

We now discuss the algorithm more formally. We begin by describing the transitions labelled by $G^i_{(q_1,Q_2)}$ before discussing the construction of the sequence $A_0, A_1, \ldots$ and the automaton $A^*$.

To aid in the construction of an automaton representing $Pre^*(C_{Init})$ we introduce a new kind of transition to the 2-store automata. These new transitions are introduced during the processing of the APDS commands. They are labelled with place-holders that will eventually be converted into 1-store automata.

Between any state $q_1$ and set of states $Q_2$ we add at most one transition. We associate this transition with an identifier $\widetilde{G}_{(q_1,Q_2)}$. To describe our algorithm we will define sequences of automata, indexed by $i$. We superscript the identifier to indicate to which automaton in the sequence it belongs. The identifier $\widetilde{G}^i_{(q_1,Q_2)}$ is associated with a set that acts as a recipe for updating the 1-store automaton described by $\widetilde{G}^{i-1}_{(q_1,Q_2)}$ or creating a new automaton if $\widetilde{G}^{i-1}_{(q_1,Q_2)}$ does not exist. Ultimately, the constructed 1-store automaton will label the new transition. In the sequel, we will confuse the notion of an identifier and its associated set. The intended usage should be clear from the context.

The sets are in a kind of disjunctive normal form. A set $\{S_1, \ldots, S_m\}$ represents an automaton that accepts the union of the languages accepted by the automata described by $S_1, \ldots, S_m$. Each set $S \in \{S_1, \ldots, S_m\}$ corresponds to a possible effect of a command $d$ at order-1 of the automaton. The automaton described by $S = \{\alpha_1, \ldots, \alpha_m\}$ accepts the intersection of languages described by its elements $\alpha_t$ ($t \in \{1, \ldots, m\}$).

An element that is an automaton $B$ refers directly to the automaton $B$. Similarly, an identifier $\widetilde{G}^i_{(q_1,Q_2)}$ refers to its corresponding automaton. Finally, an element of the form $(a, push_w, \theta)$ refers to an automaton capturing the effect of applying the inverse of the $push_w$ command to the stacks accepted by the automaton represented by $\theta$; moreover, the $top_1$ character of the stacks accepted by the new automaton will be $a$. It is a consequence of the construction that for any $S$ added during the algorithm, if $(a, push_w, \theta) \in S$ and $(a', push_{w'}, \theta') \in S$ then $a = a'$.

Formally, to each $\widetilde{G}^i_{(q_1,Q_2)}$ we attach a subset of

$$2^{\mathcal{B} \cup \widetilde{\mathcal{G}}^{i-1} \cup (\Sigma \times \mathcal{O}_1 \times (\mathcal{B} \cup \widetilde{\mathcal{G}}^{i-1}))}$$

where $\mathcal{B}$ is the set of all 1-store automata occurring in $A_0$ and all automata of the form $B^a_1$. Further, we denote the set of all identifiers $\widetilde{G}^i_{(q,Q)}$ in $A_i$ as $\widetilde{\mathcal{G}}^i$. The sets $\mathcal{B}$ and $\mathcal{O}_1$ are finite by definition. The size of the set $\widetilde{\mathcal{G}}^i$ for any $i$ is finitely bound by the (fixed) state-set of $A_i$.

We build the automata for all $\widetilde{G}^i_{(q_1,Q_2)} \in \widetilde{\mathcal{G}}^i$ simultaneously. That is, we create a single automaton $\mathcal{G}^i$ associated with the set $\widetilde{\mathcal{G}}^i$. This automaton has a state $g^i_{(q_1,Q_2)}$ for each $\widetilde{G}^i_{(q_1,Q_2)} \in \widetilde{\mathcal{G}}^i$. The automaton $G^i_{(q_1,Q_2)}$ labelling the transition $q_1 \longrightarrow_i Q_2$ is the automaton $\mathcal{G}^i$ with $g^i_{(q_1,Q_2)}$ as its initial state.

The automaton $\mathcal{G}^i$ is built inductively. We set $\mathcal{G}^0$ to be the disjoint union of all automata in $\mathcal{B}$. We define $\mathcal{G}^{i+1} = T_{\widetilde{\mathcal{G}}^{i+1}}(\mathcal{G}^i)$ where $T_{\widetilde{\mathcal{G}}^j}(\mathcal{G}^i)$ is given in Definition 5.3.1. Notice that we use $j$ rather than $(i+1)$. This is because, in Section 5.3.4 we will eventually fix a value $i_1$ and define for all $i > i_1$, $\mathcal{G}^{i+1} = T_{\widetilde{\mathcal{G}}^{i_1}[i_1/i_1-1]}(\mathcal{G}^i)$ rather than $\mathcal{G}^{i+1} = T_{\widetilde{\mathcal{G}}^{i+1}}(\mathcal{G}^i)$.

**Definition 5.3.1.** Given an automaton $\mathcal{G}^i = (\mathcal{Q}^i, \Sigma, \Delta^i, _, \mathcal{Q}_f)$ and a set of identifiers (with associated sets) $\widetilde{\mathcal{G}}^j_1$, we define,

$$\mathcal{G}^{i+1} = T_{\widetilde{\mathcal{G}}^j}(\mathcal{G}^i) = (\mathcal{Q}^{i+1}, \Sigma, \Delta^{i+1}, _, \mathcal{Q}_f)$$

where $\mathcal{Q}^{i+1} = \mathcal{Q}^i \cup \{\ g^j_{(q_1,Q_2)} \mid \widetilde{G}^j_{(q_1,Q_2)} \in \widetilde{\mathcal{G}}^j\ \}$, $\Delta^{i+1} = \Delta^{inherited} \cup \Delta^{new} \cup \Delta^i$, and,

$$\begin{array}{rcl} \Delta^{inherited} & = & \{\ g^j_{(q_1,Q_2)} \xrightarrow{a} Q \mid (g^{j-1}_{(q_1,Q_2)} \xrightarrow{a} Q) \in \Delta^i\ \} \\ \Delta^{new} & = & \left\{\ g^j_{(q_1,Q_2)} \xrightarrow{b} Q \mid\ \ \widetilde{G}^j_{(q_1,Q_2)} \in \widetilde{\mathcal{G}}^j \text{ and } b \in \Sigma \text{ and } (1)\ \right\} \end{array}$$

where (1) requires $\{\alpha_1, \ldots, \alpha_r\} \in \widetilde{G}^j_{(q_1,Q_2)}$, $Q = Q_1 \cup \cdots \cup Q_r$ and for each $t \in \{1, \ldots, r\}$ we have,

- If $\alpha_t = \theta$, then $(q^\theta \xrightarrow{b} Q_t) \in \Delta^i$.
- If $\alpha_t = (a, push_w, \theta)$, then $b = a$ and $q^\theta \xrightarrow{w} Q_t$ is a run of $\mathcal{G}^i$.

There are two key parts to Definition 5.3.1. During the first stage we add a new initial state for each automaton forming a part of $\mathcal{G}^{i+1}$. By adding new initial states, rather than using the previous set of initial states, we guarantee that no unwanted cycles are introduced, which may lead to the erroneous acceptance of certain stores. We ensure that each 1-store accepted by $\mathcal{G}^i$ is accepted by $\mathcal{G}^{i+1}$ — and the set of accepted stores is increasing — by inheriting transitions from the previous set of initial states.

During the second stage we add transitions between the set of new initial states and the state-set of $\mathcal{G}^i$ to capture the effect of a backwards application of the APDS commands to $\mathcal{L}(A_i)$.

There are two different forms for the elements $\alpha_t \in \{\alpha_1, \ldots, \alpha_r\}$. If $\alpha_t$ refers directly to an automaton, then we require that the new store is also accepted by the automaton referred to by $\alpha_t$. We simply inherit the initial transitions of that automaton in a similar manner to the first stage of $T_{\widetilde{\mathcal{G}}^j}(\mathcal{G}^i)$. If $\alpha_t$ is of the form $(a, push_w, \theta)$, then it corresponds to the effects of a command $(p, a, \{\ldots, (push_w, p'), \ldots\})$. The new store must have the character $a$ as its $top_1$ character, and the store resulting from the application of the operation $push_w$ must be accepted by the automaton represented by $\theta$. That is, the new state must accept all stores of the form $aw'$ when the store $ww'$ is accepted by $\theta$.

### 5.3.3 Constructing the Sequence $A_0, A_1, \ldots$

For a given order-2 APDS with commands $\mathcal{D}$ we define $A_{i+1} = T_\mathcal{D}(A_i)$ where the operation $T_\mathcal{D}$ follows. We assume $A_0$ has a state $q^\varepsilon_f$ with no outgoing transitions and a state $q^*_f$ from which all stores are accepted.

**Definition 5.3.2.** Given an automaton $A_i = (\mathcal{Q}, \Sigma, \Delta^i, \{q^1, \ldots, q^z\}, \mathcal{Q}_f)$ and a set of commands $\mathcal{D}$, we define,

$$A_{i+1} = T_\mathcal{D}(A_i) = (\mathcal{Q}, \Sigma, \Delta^{i+1}, \{q^1, \ldots, q^z\}, \mathcal{Q}_f)$$

where $\Delta^{i+1}$ is given below.

We begin by defining the set of labels $\widetilde{\mathcal{G}}^{i+1}$. This set contains labels on transitions present in $A_i$, and labels on transitions derived from $\mathcal{D}$. That is,

$$\widetilde{\mathcal{G}}^{i+1} \quad = \quad \left\{ \widetilde{G}^{i+1}_{(q^j,Q)} \mid (q^j \xrightarrow{\widetilde{G}^{i}_{(q^j,Q)}} Q) \in \Delta^i \text{ and } j \in \{1, \ldots, z\} \right\} \cup \left\{ \widetilde{G}^{i+1}_{(q^j,Q)} \mid (2) \right\}$$

The contents of the associated sets $\widetilde{G}^{i+1}_{(q,Q)} \in \widetilde{\mathcal{G}}^{i+1}$ are defined $\widetilde{G}^{i+1}_{(q^j,Q)} = \{ S \mid (2) \}$ where (2) requires $(p^j, a, \{(o_1, p^{k_1}), \ldots, (o_m, p^{k_m})\}) \in \mathcal{D}$, $Q = Q_1 \cup \cdots \cup Q_m$, $S = S_1 \cup \cdots \cup S_m$ and for each $t \in \{1, \ldots, m\}$ we have,

- If $o_t = push_2$, then $S_t = \{B^a_1\} \cup \widetilde{\theta}_1 \cup \widetilde{\theta}_2$ and there exists a path $q^{k_t} \xrightarrow{\widetilde{\theta}_1}_i Q' \xrightarrow{\widetilde{\theta}_2}_i Q_t$ in $A_i$.
- If $o_t = pop_2$, then $S_t = \{B^a_1\}$ and $Q_t = \{q^{k_t}\}$. Or, if $q^j \xrightarrow{\triangledown}_i \{q^\varepsilon_f\}$ exists in $A_i$, we may have $S_t = \{B^a_1\}$ and $Q_t = \{q^\varepsilon_f\}$.
- If $o_t = push_w$ then $S_t = \{(a, push_w, \theta)\}$ and there exists a transition $q^{k_t} \xrightarrow{\theta}_i Q_t$ in $A_i$.

Finally, we give the transition relation $\Delta^{i+1}$.

$$\Delta^{i+1} = \left\{ q \xrightarrow{B} Q \mid \begin{array}{l} (q \xrightarrow{B} Q) \in \Delta^i \\ \text{and } B \in \mathcal{B} \end{array} \right\} \cup \left\{ q \xrightarrow{\widetilde{G}^{i+1}_{(q,Q)}} Q \mid \widetilde{G}^{i+1}_{(q,Q)} \in \widetilde{\mathcal{G}}^{i+1} \right\}$$

We can construct an automaton whose transitions are 1-store automata by replacing each set $\widetilde{G}^{i+1}_{(q,Q)}$ with the automaton $G^{i+1}_{(q,Q)}$ which is $\mathcal{G}^{i+1}$ with initial state $g^{i+1}_{(q,Q)}$, where $\mathcal{G}^{i+1} = T_{\widetilde{\mathcal{G}}^{i+1}}(\mathcal{G}^i)$. Note that $\mathcal{G}^i$ is assumed by induction. In the base case, $\mathcal{G}^0$ is the disjoint union of all automata in $\mathcal{B}$.

The above construction is similar to Definition 5.3.1. However, because we do not change the initial states of the automaton, we do not have to perform the inheritance step. Furthermore the set of commands $\mathcal{D}$ specify how the automata should be updated, rather than a set $\widetilde{\mathcal{G}}^i$. A command $(p^j, a, \{(o_1, p^{k_1}), \ldots, (o_m, p^{k_m})\})$ takes the place of a set $\{\alpha_1, \ldots, \alpha_m\}$.

The contents of $S_t$ and $Q_t$ depend on the operation $o_t$. If $o_t$ is of a lower order than 2 (that is, a $push_w$ command) then $o_t(\gamma w) = o_t(\gamma)w$ for any store $\gamma w$. Hence we inherit the first transition from the initial state of the automaton represented $\theta$, but pass the required constraint (using $S_t = \{(a, o_t, B)\}$) to the lower orders of the automaton.

Otherwise $o_t$ is a $pop_2$ or $push_2$ operation. If is a $push_2$ command, then $push_2(\gamma w') = \gamma\gamma w'$, and hence we use $S_t$ to ensure that the top store $\gamma$ of $\gamma w'$ is accepted by the first two transitions from the initial state of the automaton represented by $\theta$ and we use $Q_t$ to ensure that the tails of the stores match. Note that, in contrast to Bouajjani and Meyer (Section 3.3), we use the power of alternation, rather than a product construction, in this case. This simplifies the handling of the state-set.

When $o_t$ is a $pop_2$ operation and the new store is simply the old store with an additional 2-store on top (that is $pop_2(\gamma w') = w'$), $Q_t$ is the initial state of the automaton represented by $\theta$ and $S_t$ contains the automaton $B^a_1$. This ensures that the $top_1$ character of the new store is $a$. We also need to consider the undefined store $\triangledown$. This affects the processing of $pop_2$ operations since their result is not always defined. Hence, when considering which new

stores may be accepted by $A_{i+1}$, we check whether the required undefined configuration is accepted by $A_i$. This is witnessed by the presence of a $\nabla$ transition from $p^j$. If the result may be undefined, we accept all stores that do not have an image under the $pop_2$ operation. That is, all stores of the form $[\gamma]$.

By repeated applications of $T_{\mathcal{D}}$ we construct the sequence $A_0, A_1, \ldots$ which is sound and complete with respect to $Pre^*(C_{Init})$.

**Property 5.3.1.** *For any configuration $\langle p^j, \gamma \rangle$ it is the case that $\gamma \in \mathcal{L}(A_i^{q^j})$ for some $i$ iff $\langle p^j, \gamma \rangle \in Pre^*(C_{Init})$.*

### 5.3.4 Constructing the Automaton $A_*$

We need to construct a finite representation of the sequence $A_0, A_1, \ldots$ in a finite amount of time. To do this we will construct an automaton $A_*$ such that $\mathcal{L}(A_*) = \bigcup_{i \geq 0} \mathcal{L}(A_i)$. We begin by introducing some notation and a notion of subset modulo $i$ for the sets $\widetilde{G}^i_{(q_1,Q_2)}$.

**Definition 5.3.3.**

1. Given $\theta \in \mathcal{B} \cup \widetilde{\mathcal{G}}^i$ for some $i$, let

$$\theta[j/i] = \begin{cases} \theta & \text{if } \theta \in \mathcal{B} \\ G^j_{(q_1,Q_2)} & \text{if } \theta = G^i_{(q_1,Q_2)} \in \widetilde{\mathcal{G}}^i \end{cases}$$

2. For a set $S$ we define $S[j/i]$ such that,
   (a) We have $\theta \in S$ iff we have $\theta[j/i] \in S[j/i]$, and
   (b) We have $(a, o, \theta) \in S$ iff we have $(a, o, \theta[j/i]) \in S[j/i]$.
3. We extend the notation $[j/i]$ to nested sets of sets structures in a point-wise fashion.

We now define $\widetilde{G}^i_{(q_1,Q_2)} \lesssim \widetilde{G}^j_{(q_1,Q_2)}$. Both sets contain elements of the form $\theta$ or $(a, o, \theta)$. In $\widetilde{G}^i_{(q_1,Q_2)}$ we have that the $\theta$ are of the form $G^{(i-1)}_{(q,Q')}$ for some $q$ and $Q$, or $\theta \in \mathcal{B}$. Similarly, in $\widetilde{G}^j_{(q_1,Q_2)}$ we have that the $\theta$ are of the form $G^{(j-1)}_{(q,Q')}$, or $\theta \in \mathcal{B}$. The $\lesssim$ relation defines a subset relation that is insensitive to the indices $(i-1)$ and $(j-1)$ appearing on the $\theta$. We have $\widetilde{\mathcal{G}}^i \simeq \widetilde{\mathcal{G}}^{i+1}$ when $\widetilde{\mathcal{G}}^i \lesssim \widetilde{\mathcal{G}}^{i+1}$ and $\widetilde{\mathcal{G}}^{i+1} \lesssim \widetilde{\mathcal{G}}^i$. When this occurs, we have reached a required fixed point as explained below.

**Definition 5.3.4.**

1. We write $\widetilde{G}^i_{(q_1,Q_2)} \lesssim \widetilde{G}^j_{(q_1,Q_2)}$ iff for each $S \in \widetilde{G}^i_{(q_1,Q_2)}$ we have $S[j-1/i-1] \in \widetilde{G}^j_{(q_1,Q_2)}$.
2. If $\widetilde{G}^i_{(q_1,Q_2)} \lesssim \widetilde{G}^j_{(q_1,Q_2)}$ and $\widetilde{G}^j_{(q_1,Q_2)} \lesssim \widetilde{G}^i_{(q_1,Q_2)}$, then we write $\widetilde{G}^i_{(q_1,Q_2)} \simeq \widetilde{G}^j_{(q_1,Q_2)}$.
3. Furthermore, we extend the notation to sets. That is, $\widetilde{\mathcal{G}}^i \lesssim \widetilde{\mathcal{G}}^j$ iff for all $\widetilde{G}^i_{(q_1,Q_2)} \in \widetilde{\mathcal{G}}^i$ we have $\widetilde{G}^j_{(q_1,Q_2)} \in \widetilde{\mathcal{G}}^j$ and $\widetilde{G}^i_{(q_1,Q_2)} \lesssim \widetilde{G}^j_{(q_1,Q_2)}$.

We now show that a fixed point is reached at order-2. That we reach a fixed point is important, since, when $\widetilde{\mathcal{G}}^i \simeq \widetilde{\mathcal{G}}^{i+1}$ there are two key consequences. Firstly, for all $q_1$ and $Q_2$, we have $\widetilde{G}^i_{(q_1,Q_2)} \in \widetilde{\mathcal{G}}^i$ iff we also have $\widetilde{G}^{i+1}_{(q_1,Q_2)} \in \widetilde{\mathcal{G}}^{i+1}$. This means that, if we ignore the automata labelling the edges of $A_i$ and $A_{i+1}$, the two automata have the same transition structure. The second consequence follows from the first: we have $\widetilde{G}^i_{(q_1,Q_2)} \simeq \widetilde{G}^{i+1}_{(q_1,Q_2)}$ for all $q_1$ and $Q_2$. That is, the automata labelling the edges of $A_i$ and $A_{i+1}$ will be updated in the same manner. It is this repetition that allows us to fix the state-set at order-1, and thus reach a final fixed point.

**Property 5.3.2.** *There exists $i_1 > 0$ such that $\widetilde{\mathcal{G}}^i \simeq \widetilde{\mathcal{G}}^{i_1}$ for all $i \geq i_1$.*

*Proof.* (Sketch) Since the order-1 state-set in $A_i$ remains constant and we add at most one transition between any state $q_1$ and set of states $Q_2$, there is some $i_1$ where no more transitions are added at order-2. That $\widetilde{\mathcal{G}}^i \simeq \widetilde{\mathcal{G}}^{i_1}$ for all $i \geq i_1$ follows since the contents of $\widetilde{G}^i_{(q_1,Q_2)}$ and $\widetilde{G}^{i_1}_{(q_1,Q_2)}$ are derived from the same transition structure. □

Once a fixed point has been reached at order-2, we can fix the state-set at order-1.

**Lemma 5.3.1.** *Suppose we have constructed a sequence of automata $\mathcal{G}^0, \mathcal{G}^1, \ldots$ with the associated sets $\widetilde{\mathcal{G}}^0, \widetilde{\mathcal{G}}^1, \ldots$. Further, suppose there exists an $i_1$ such that for all $i \geq i_1$ we have $\widetilde{\mathcal{G}}^i \simeq \widetilde{\mathcal{G}}^{i_1}$. We can define a sequence of automata $\hat{\mathcal{G}}^{i_1}, \hat{\mathcal{G}}^{i_1+1}, \ldots$ such that the state-set in $\hat{\mathcal{G}}^i$ remains constant and there exists $i_0$ such that $\hat{\mathcal{G}}^{i_0}$ characterises the sequence — that is, the following are equivalent for all $w$,*

1. *The run $g^{i_1}_{(q,Q')} \xrightarrow{w}_i Q_1$ with $Q_1 \subseteq \mathcal{Q}_f$ exists in $\hat{\mathcal{G}}^i$ for some $i$.*
2. *The run $g^{i_1}_{(q,Q')} \xrightarrow{w}_{i_0} Q_2$ with $Q_2 \subseteq \mathcal{Q}_f$ exists in $\hat{\mathcal{G}}^{i_0}$.*
3. *The run $g^{i'}_{(q,Q')} \xrightarrow{w}_{i'} Q_3$ with $Q_3 \subseteq \mathcal{Q}_f$ exists in $\mathcal{G}^{i'}$ for some $i'$.*

*Proof.* Follows from the definition of $\hat{\mathcal{G}}^{i+1} = T_{\widetilde{\mathcal{G}}^{i_1}[i_1/i_1-1]}(\hat{\mathcal{G}}^i_1)$, Lemma 5.4.11, Lemma 5.4.12 and Lemma 5.4.13. □

We use $\hat{\mathcal{G}}^{i+1} = T_{\widetilde{\mathcal{G}}^{i_1}[i_1/i_1-1]}(\hat{\mathcal{G}}^i)$ to construct the sequence $\hat{\mathcal{G}}^{i_1}, \hat{\mathcal{G}}^{i_1+1}, \ldots$ (with $\hat{\mathcal{G}}^{i_1} = \mathcal{G}^{i_1}$). Intuitively, since the transitions from the states introduced to define $\mathcal{G}^i$ for $i \geq i_1$ are derived from similar sets, we can compress the subsequent repetition into a single set of new states. The substitution $\widetilde{\mathcal{G}}^{i_1}[i_1/i_1-1]$ makes the sets in $\widetilde{\mathcal{G}}^{i_1}$ self-referential. This generates the loops shown in Figure 5.6. Since the state-set of this new sequence does not change and the alphabet $\Sigma$ is finite, the transition structure will become saturated.

We define $\hat{\mathcal{G}}^* = \hat{\mathcal{G}}^{i_0}$ letting $g^*_{(q_1,Q_2)} = g^{i_1}_{(q_1,Q_2)}$ for each $g^{i_1}_{(q_1,Q_2)}$. Finally, we show that we can construct the automaton $A_*$.

**Property 5.3.3.** *There exists an automaton $A_*$ which is sound and complete with respect to $A_0, A_1, \ldots$ and hence computes the set $Pre^*(C_{Init})$.*

*Proof.* By Property 5.3.2 there is some $i_1$ with $\widetilde{\mathcal{G}}^i \simeq \widetilde{\mathcal{G}}^{i_1}$ for all $i \geq i_1$. By Lemma 5.3.1, we have $\hat{\mathcal{G}}^* = \hat{\mathcal{G}}^{i_0}$. We then define $A_*$ from $A_{i_1}$ with each transition $q \longrightarrow_* Q'$ in $A_*$ labelled with the appropriate $B \in \mathcal{B}$ or automaton $G^*_{(q,Q')}$ from $\hat{\mathcal{G}}^* = \hat{\mathcal{G}}^{i_0}$. □

Thus, we have the following algorithm for constructing $A_*$:

1. Given $A_0$, iterate $A_{i+1} = T_{\mathcal{D}}(A_i)$ until the fixed point $A_{i_1}$ is reached.
2. Iterate $\hat{\mathcal{G}}^{i+1} = T_{\widetilde{\mathcal{G}}_l^{i_1}[i_1/i_1-1]}(\hat{\mathcal{G}}^i)$ to generate the fixed point $\hat{\mathcal{G}}^*$ from $\mathcal{G}^{i_1}$.
3. Construct $A_*$ by labelling the transitions of $A_{i_1}$ with automata derived from $\hat{\mathcal{G}}^*$.

## 5.4 The General Case

We now generalise the order-2 algorithm to APDSs of all orders.

**Theorem 5.4.1.** *Given an $n$-store multi-automaton $A$ accepting the set of configurations $C_{Init}$ of an order-$n$ APDS, we can construct in $n$-EXPTIME (in the size of $A_0$) an $n$-store multi-automaton $A_*$ accepting the set $Pre^*(C_{Init})$. Thus, $Pre^*(C_{Init})$ is regular.*

In the order-2 case, we added transitions to the 2-store multi-automaton $A_0$ depending on the commands $\mathcal{D}$ of the given APDS. The (order-1 store-)automata labelling the transitions were updated through the addition of new initial states until the transition structure at order-2 reached a fixed point. We were then able to stop adding new states at order-1, adding only new transitions until a second fixed point was reached. At this point the algorithm terminated.

In the order-$n$ case, new states are added to the automata labelling the order-$n$ edges until the order-$n$ transition structure reaches a fixed point. We are then able to stop adding new states to the order-$(n-1)$ automata, although we continue to add new states to the automata labelling the transitions at order-$(n-1)$. Since the order-$(n-1)$ state-set is fixed, the order-$(n-1)$ transition structure will reach a second fixed point. At this stage we are no longer required to add new states to order-$(n-2)$ of the automaton, and so on. In this way we have a cascade of fixed points from order-$n$ of the automaton down to order-1, at which point the algorithm terminates.

### 5.4.1 Preliminaries

We generalise the definition of $G^i_{(q_1,Q_2)} \in \widetilde{\mathcal{G}}^i$ to any order $l$ of the store-automata. That is, to each $\widetilde{G}^i_{(q_1,Q_2)}$ at order-$l$ we attach a subset of

$$2^{\mathcal{B}_{l-1} \cup \widetilde{\mathcal{G}}^{i-1}_{l-1} \cup (\Sigma \times \mathcal{O}_l \times (\mathcal{B}_{l-1} \cup \widetilde{\mathcal{G}}^{i-1}_{l-1}))}$$

where $\mathcal{B}_{l-1}$ is the union of the set of all $(l-1)$-store automata occurring in $A_0$ and in automata of the form $B^a_{l'}$ or $X^a_{l'}$ for some $l'$ (defined in Section 5.1). Further, we denote the set of all order-$(l-1)$ identifiers $\widetilde{G}^i_{(q_1,Q_2)}$ in $A_i$ as $\widetilde{\mathcal{G}}^i_{l-1}$. The sets $\mathcal{B}_{l-1}$ and $\mathcal{O}_l$ are finite by definition. If the state-set at order-$l$ is fixed, there is a finite bound on the size of the set $\widetilde{\mathcal{G}}^i_{l-1}$ for any $i$.

Similarly, we generalise the definition of the operation $T_{\widetilde{\mathcal{G}}^j_l}$. When $l = 1$, the definition is identical to Definition 5.3.1. When $l > 1$, the definition is similar to the definition of $T_{\mathcal{D}}$ in the order-2 case, except we do not have to consider the undefined store $\triangledown$ which only occurs at order-$n$ of an $n$-store multi-automaton.

**Definition 5.4.1.** Given an automaton $\mathcal{G}^i_l = (\mathcal{Q}^i, \Sigma, \Delta^i, _, \mathcal{Q}_f)$ and a set of identifiers $\widetilde{\mathcal{G}}^j_l$, we define,

$$\mathcal{G}^{i+1}_l = T_{\widetilde{\mathcal{G}}^j_l}(\mathcal{G}^i_l) = (\mathcal{Q}^{i+1}, \Sigma, \Delta^{i+1}, _, \mathcal{Q}_f)$$

where $\mathcal{Q}^{i+1} = \mathcal{Q}^i \cup \{ g^j_{(q_1,Q_2)} \mid \mathcal{G}^j_{(q_1,Q_2)} \in \widetilde{\mathcal{G}}^j_l \}$ and $\Delta^{i+1}$ depends on $l$:

*Case* $l = 1$. $\Delta^{i+1} = \Delta^{inherited} \cup \Delta^{derived} \cup \Delta^i$ and,

$$\begin{array}{rcl} \Delta^{inherited} & = & \{ g^j_{(q_1,Q_2)} \xrightarrow{a} Q \mid (g^{j-1}_{(q_1,Q_2)} \xrightarrow{a} Q) \in \Delta^i \} \\ \Delta^{derived} & = & \left\{ g^j_{(q_1,Q_2)} \xrightarrow{b} Q \mid \begin{array}{l} \{\alpha_1, \dots, \alpha_r\} \in \mathcal{G}^j_{(q_1,Q_2)} \in \widetilde{\mathcal{G}}^j_l \text{ and} \\ b \in \Sigma \text{ and } Q = Q_1 \cup \dots \cup Q_r \text{ and (1)} \end{array} \right\} \end{array}$$

where (1) requires, for each $t \in \{1, \dots, r\}$ we have,

- If $\alpha_t = \theta$, then $(q^\theta \xrightarrow{b} Q_t) \in \Delta^i$.
- If $\alpha_t = (a, push_w, \theta)$, then $b = a$ and $q^\theta \xrightarrow{w} Q_t$ is a run of $\mathcal{G}^i_1$.

*Case* $l > 1$. We begin by defining the set of labels $\widetilde{\mathcal{G}}^{i+1}_{l-1}$. This set contains labels on transitions from $\mathcal{G}^i_l$, labels on transitions from the new states $g^j_{(q_1,Q_2)}$ inherited from $g^{j-1}_{(q_1,Q_2)}$ and labels on transitions derived from $\widetilde{\mathcal{G}}^j_l$. That is,

$$\widetilde{\mathcal{G}}^{i+1}_{l-1} = \begin{array}{l} \left\{ \widetilde{G}^{i+1}_{(q,Q)} \mid (q \xrightarrow{\widetilde{G}^i_{(q,Q)}} Q) \in \Delta^i \right\} \cup \\ \left\{ \widetilde{G}^{i+1}_{(q,Q)} \mid q = g^j_{(q_1,Q_2)} \text{ and } (g^{j-1}_{(q_1,Q_2)} \xrightarrow{B} Q) \in \Delta^i \right\} \cup \\ \left\{ \widetilde{G}^{i+1}_{(q,Q)} \mid q = g^j_{(q_1,Q_2)} \text{ and } \widetilde{G}^j_{(q_1,Q_2)} \in \widetilde{\mathcal{G}}^j_l \text{ and (2)} \right\} \end{array}$$

The contents of the associated sets $\widetilde{G}^{i+1}_{(q,Q)} \in \widetilde{\mathcal{G}}^{i+1}_{l-1}$ are defined as follows,

$$\begin{array}{rcl} \widetilde{G}^{i+1}_{(q,Q)} & = & \emptyset \text{ if } q \neq g^j_{(q_1,Q_2)} \\ \widetilde{G}^{i+1}_{(g^j_{(q_1,Q_2)},Q)} & = & \{ S \mid (2) \} \cup \left\{ \{B\} \mid (g^{j-1}_{(q_1,Q_2)} \xrightarrow{B} Q) \in \Delta^i \right\} \end{array}$$

where (2) requires $\{\alpha_1, \dots, \alpha_r\} \in \widetilde{G}^j_{(q_1,Q_2)}$, $Q = Q_1 \cup \dots \cup Q_r$, $S = S_1 \cup \dots \cup S_r$ and for each $t \in \{1, \dots, r\}$ we have,

- If $\alpha_t = \theta$, then $(q^\theta \xrightarrow{B} Q_t) \in \Delta^i$ and $S_t = \{B\}$.
- If $\alpha_t = (a, push_l, \theta)$, then $q^\theta \xrightarrow{\widetilde{B}_1} Q^1 \xrightarrow{\widetilde{B}_2} Q_t$ is a path in $\mathcal{G}^i_l$ and $S_t = \{B^a_{l-1}\} \cup \widetilde{B}_1 \cup \widetilde{B}_2$.
- If $\alpha_t = (a, pop_l, \theta)$, then $Q_t = \{q^\theta\}$ and $S_t = \{B^a_{l-1}\}$.
- If $\theta = (a, o, \theta)$ when $\ell(o) < l$, then $q^\theta \xrightarrow{B} Q_t \in \Delta^i$ and $S_t = \{(a, o, B)\}$

Finally, we give the transition relation $\Delta^{i+1}$.

$$\Delta^{i+1} = \begin{array}{l} \left\{ q \xrightarrow{B} Q \mid (q \xrightarrow{B} Q) \in \Delta^i \text{ and } B \in \mathcal{B}_{l-1} \right\} \cup \\ \left\{ q \xrightarrow{\widetilde{G}^{i+1}_{(q,Q)}} Q \mid \widetilde{G}^{i+1}_{(q,Q)} \in \widetilde{\mathcal{G}}^{i+1}_{l-1} \right\} \end{array}$$

We can construct an automaton whose transitions are $(l-1)$-store automata by replacing each set $\widetilde{G}^{i+1}_{(q,Q)}$ with the automaton $G^{i+1}_{(q,Q)}$ which is $\mathcal{G}^{i+1}_{l-1}$ with initial state $g^{i+1}_{(q,Q)}$, where $\mathcal{G}^{i+1}_{l-1} = T_{\widetilde{\mathcal{G}}^{i+1}_{l-1}}(\mathcal{G}^i_{l-1})$. Note that $\mathcal{G}^{i+1}_{l-1} = T_{\widetilde{\mathcal{G}}^{i+1}_{l-1}}(\mathcal{G}^i_{l-1})$ is a recursive call and $\mathcal{G}^i_{l-1}$ is assumed by induction.

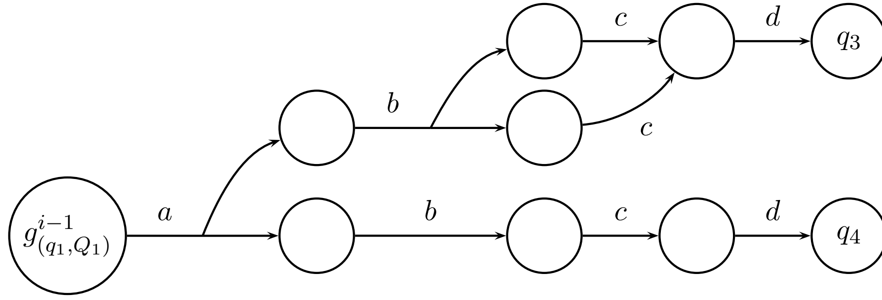



Figure 5.8: An example for $push_{abcd}$.

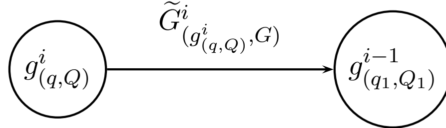



Figure 5.9: An example for $pop_l$.

### 5.4.2 Further Examples

We have given an outline of a fully worked example at order-two. However, there are many situations that this example does not illuminate. In this section we give a series of short examples showing the introduction of transitions to $\mathcal{G}^i_l$ from elements in $\widetilde{G}^i_{(q,Q')}$ not covered in the previous case.

**Case** $\{(a, push_{abcd}, \widetilde{G}^{i-1}_{(q_1,Q_1)})\} \in \widetilde{G}^i_{(q,Q)} \in \widetilde{\mathcal{G}}^i_1$

Suppose Figure 5.8 represents a sub-automaton of $\mathcal{G}^{i-1}_1$. There is one run from the state $g^{i-1}_{(q_1,Q_1)}$ over the word *abcd*. In particular we have,

$$g^{i-1}_{(q_1,Q_1)} \xrightarrow{abcd}_{i-1} \{q_3, q_4\}$$

Therefore we add the transition,

$$g^i_{(q,Q)} \xrightarrow{a}_i \{q_3, q_4\}$$

to $\mathcal{G}^i_1$.

**Case** $\{(a, pop_l, \widetilde{G}^{i-1}_{(q_1,Q_1)})\} \in \widetilde{G}^i_{(q,Q)} \in \widetilde{\mathcal{G}}^i_l$

Figure 5.9 shows the relevant subsection of $\mathcal{G}^i_l$ after the $pop_l$ has been processed. Let $G = g^{i-1}_{(q_1,Q_1)}$. We have $\{B^a_{l-1}\}$ in $\widetilde{G}^i_{(g^i_{(q,Q)},G)}$.

**Case** $\{(a, push_l, \widetilde{G}^{i-1}_{(q_1,Q_1)})\} \in \widetilde{G}^i_{(q,Q)} \in \widetilde{\mathcal{G}}^i_l$

Suppose Figure 5.10 shows a sub-automaton of $\mathcal{G}^{i-1}_l$, where $B_1, B_2, B_3 \in \mathcal{B}_{l-1} \cup \widetilde{\mathcal{G}}^{i-1}_{l-1}$. There

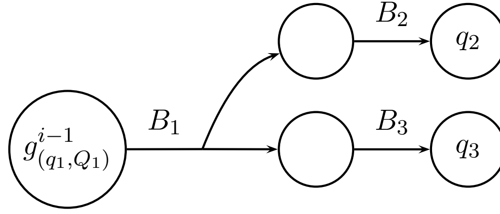



Figure 5.10: An example for $push_l$.

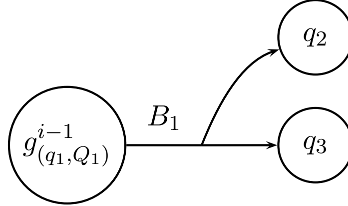



Figure 5.11: An example for $\ell(o) < l$.

is one possible value for $Q''$ when enumerating over runs of the form,

$$\{g^{i-1}_{(q_1,Q_1)}\} \xrightarrow{\widetilde{B}_1}_{i-1} Q' \xrightarrow{\widetilde{B}_2}_{i-1} Q''$$

In particular $Q'' = \{q_2, q_3\}$. We have $\widetilde{B}_1 = \{B_1\}$ and $\widetilde{B}_2 = \{B_2, B_3\}$. Consequently, we ensure that the transition,

$$g^i_{(q,Q)} \xrightarrow{\widetilde{G}^i_{(g^i_{(q,Q)},\{q_2,q_3\})}} \{q_2, q_3\}$$

exists, and that $\{B^a_{l-1}, B_1, B_2, B_3\} \in \widetilde{G}^i_{(g^i_{(q,Q)},\{q_2,q_3\})}$.

**Case** $\{(a, o, G^{i-1}_{(q_1,Q_1)})\} \in \widetilde{G}^i_{(q,Q)} \in \widetilde{\mathcal{G}}^i_l$ **and** $\ell(o) < l$

Suppose Figure 5.11 shows a sub-automaton of $\mathcal{G}^{i-1}_l$, where $B_1 \in \mathcal{B}_{l-1} \cup \widetilde{\mathcal{G}}^{i-1}_{l-1}$. There is one possible value for $Q'$ when enumerating over runs of the form,

$$\{g^{i-1}_{(q_1,Q_1)}\} \xrightarrow{\widetilde{B}_1}_{i-1} Q'$$

In particular $Q' = \{q_2, q_3\}$. Consequently we ensure that the transition,

$$g^i_{(q,Q)} \xrightarrow{\widetilde{G}^i_{(g^i_{(q,Q)},\{q_2,q_3\})}} \{q_2, q_3\}$$

exists, and that $\{(a, o, B_1)\} \in \widetilde{G}^i_{(g^i_{(q,Q)},\{q_2,q_3\})}$.

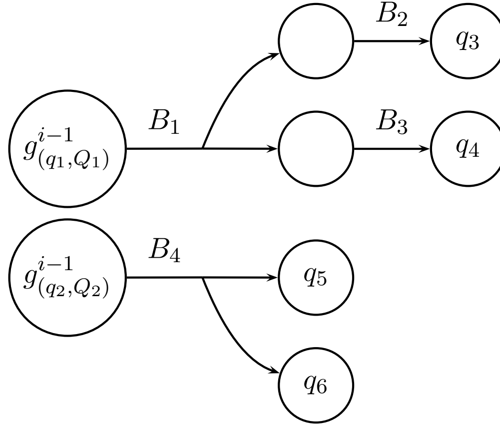



Figure 5.12: An example for a two-element $S$.

**Case** $\{(a, push_l, \widetilde{G}^{i-1}_{(q_1,Q_1)}), \widetilde{G}^{i-1}_{(q_2,Q_2)}\} \in \widetilde{G}^{i}_{(q,Q)} \in \widetilde{\mathcal{G}}^{i}_{l}$

Suppose Figure 5.12 shows a sub-automaton of $\mathcal{G}^{i-1}_{l}$, where $B_1, B_2, B_3, B_4 \in \mathcal{B}_{l-1} \cup \widetilde{\mathcal{G}}^{i-1}_{l-1}$. There is one possible value for $S = S_1 \cup S_2$ and $Q' = Q'_1 \cup Q'_2$. That is $S_1 = \{B^a_{l-1}, B_1, B_2, B_3\}$ and $Q'_1 = \{q_3, q_4\}$ (as in the previous $push_l$ example); and $S_2 = \{B_4\}$ and $Q'_2 = \{q_5, q_6\}$.

Therefore, we ensure that the transition,

$$g^{i}_{(q,Q)} \xrightarrow{\widetilde{G}^{i}_{(g^{i}_{(q,Q)}, Q')}} Q'$$

exists, and that $S \in \widetilde{G}^{i}_{(g^{i}_{(q,Q)}, Q')}$.

### 5.4.3 Constructing $A_0, A_1, \ldots$

For a given higher-order APDS with commands $\mathcal{D}$ we define $A_{i+1} = T_{\mathcal{D}}(A_i)$ where the operation $T_{\mathcal{D}}$ is as follows:

**Definition 5.4.2.** Given an automaton $A_i = (\mathcal{Q}, \Sigma, \Delta^i, \{q^1, \ldots, q^z\}, \mathcal{Q}_f)$ and a set of commands $\mathcal{D}$, we define,

$$A_{i+1} = T_{\mathcal{D}}(A_i) = (\mathcal{Q}, \Sigma, \Delta^{i+1}, \{q^1, \ldots, q^z\}, \mathcal{Q}_f)$$

where $\Delta^{i+1}$ is given below.

We begin by defining the set of labels $\widetilde{\mathcal{G}}^{i+1}_{n-1}$. This set contains labels on transitions present in $A_i$, and labels on transitions derived from $\mathcal{D}$. That is,

$$\widetilde{\mathcal{G}}^{i+1}_{n-1} \quad = \quad \left\{ \widetilde{G}^{i+1}_{(q^j,Q)} \mid (q^j \xrightarrow{\widetilde{G}^{i}_{(q^j,Q)}} Q) \in \Delta^i \text{ and } j \in \{1, \ldots, z\} \right\} \cup \left\{ \widetilde{G}^{i+1}_{(q^j,Q)} \mid (2) \right\}$$

The contents of the sets $\widetilde{G}^{i+1}_{(q,Q)} \in \widetilde{\mathcal{G}}^{i+1}_{n-1}$ are defined $\widetilde{G}^{i+1}_{(q^j,Q)} = \{\ S \mid (2)\ \}$ where (2) requires $(p^j, a, \{(o_1, p^{k_1}), \ldots, (o_m, p^{k_m})\}) \in \mathcal{D}$, $Q = Q_1 \cup \cdots \cup Q_m$, $S = S_1 \cup \cdots \cup S_m$ and for each $t \in \{1, \ldots, m\}$ we have,

- If $o_t = push_n$, then $S_t = \{B^a_{n-1}\} \cup \widetilde{\theta}_1 \cup \widetilde{\theta}_2$ and there exists a path,

$$q^{k_t} \xrightarrow{\widetilde{\theta}_1}_i Q' \xrightarrow{\widetilde{\theta}_2}_i Q_t$$

  in $A_i$.

- If $o_t = pop_n$, then $S_t = \{B^a_{n-1}\}$ and $Q_t = \{q^{k_t}\}$. Or, if $q^j \xrightarrow{\nabla}_i \{q^{\varepsilon}_f\}$ exists in $A_i$, we may have $S_t = \{B^a_{n-1}\}$ and $Q_t = \{q^{\varepsilon}_f\}$.

- If $o_t = push_w$ or $o_t = push_l$ for $l < n$, then $S_t = \{(a, o, \theta)\}$ and there exists a transition $q^{k_t} \xrightarrow{\theta}_i Q_t$ in $A_i$.

- If $o_t = pop_l$ for $l < n$, then $S_t = \{(a, o, \theta)\}$ and there exists a transition $q^{k_t} \xrightarrow{\theta}_i Q_t$ in $A_i$. Or, if $q^j \xrightarrow{\nabla}_i \{q^{\varepsilon}_f\}$ exists in $A_i$, we may have $S_t = \{X^a_l\}$ and $Q_t = \{q^*_f\}$.

Finally, we give the transition relation $\Delta^{i+1}$.

$$\Delta^{i+1} = \left\{ q \xrightarrow{B} Q \mid \begin{array}{l} (q \xrightarrow{B} Q) \in \Delta^i \\ \text{and } B \in \mathcal{B} \end{array} \right\} \cup \left\{ q \xrightarrow{\widetilde{G}^{i+1}_{(q,Q)}} Q \mid \widetilde{G}^{i+1}_{(q,Q)} \in \widetilde{\mathcal{G}}^{i+1}_{n-1} \right\}$$

We can construct an automaton whose transitions are $(n-1)$-store automata by replacing each set $\widetilde{G}^{i+1}_{(q,Q)}$ with the automaton $G^{i+1}_{(q,Q)}$ which is $\mathcal{G}^{i+1}_{n-1}$ with initial state $g^{i+1}_{(q,Q)}$, where $\mathcal{G}^{i+1}_{n-1} = T_{\widetilde{\mathcal{G}}^{i+1}_{n-1}}(\mathcal{G}^i_{n-1})$. Note that $\mathcal{G}^i_{n-1}$ is assumed by induction.

Our construction is sound and complete. The proofs for this result are given in the next section.

**Property 5.4.1.** *For any configuration $\langle p^j, \gamma\rangle$ it is the case that $\gamma \in \mathcal{L}(A^{q^j}_i)$ for some $i$ iff $\langle p^j, \gamma\rangle \in Pre^*(C_{Init})$.*

*Proof.* The property follows from Property 5.4.2 and Property 5.4.3. □

### 5.4.4 Soundness and Completeness for $A_0, A_1, \ldots$

In this section we show that the sequence $A_0, A_1, \ldots$ is sound and complete with respect to $Pre^*(C_{Init})$, where $C_{Init} = \mathcal{L}(A_0)$.

#### Preliminaries

We begin by proving some useful properties of the automaton construction. These properties assert that the automata constructed from the sets in $\widetilde{\mathcal{G}}^i_l$ are well-behaved. Once this has been established, we need only consider order-$n$ of the automata $A_0, A_1, \ldots$ to show soundness and completeness. Note that since no $g^i_{(q_1,Q_2)}$ is accepting, any store accepted by some $G^i_{(q_1,Q_2)}$ has a $top_1$ element.

In order to reason about a particular transition, we need to know its origin. Hence we introduce the notion of an **inherited** and a **derived** transition. The remaining lemmata fall into four categories:

1. Lemma 5.4.3 shows that inherited runs are sound.

2. Lemma 5.4.2 shows the completeness of inherited runs.

3. Lemma 5.4.1, Lemma 5.4.4 and Lemma 5.4.5 show that derived runs are sound.

4. Lemma 5.4.6 shows the completeness of derived runs.

**Definition 5.4.3.** A non-empty run $g^i_{(q_1,Q_2)} \xrightarrow{w}_i Q$ of $\mathcal{G}^i_l$ or $q^j \xrightarrow{w}_i Q$ of $A_i$ can be characterised by its initial transition $g^i_{(q_1,Q_2)} \xrightarrow{\gamma}_i Q'$ where $w = \gamma w'$. There are several cases,

- $l = 1$.

  Then we have $g^i_{(q_1,Q_2)} \xrightarrow{a}_i Q'$. If the transition was inherited from $g^{i-1}_{(q_1,Q_2)}$ then we say that the run is an **inherited** run. Otherwise the transition was introduced by some $S \in \widetilde{G}^i_{(q_1,Q_2)}$. We say that the run was **derived** from $S$.

- $l > 1$.

  Then we have $g^i_{(q_1,Q_2)} \xrightarrow{\widetilde{G}} Q'$ with $\gamma \in \mathcal{L}(G)$. There are three cases depending on the accepting run of $G$ and the source of the transition from $g^i_{(q_1,Q_2)}$,

  - If the accepting run of $G$ is an inherited run, then the run $g^i_{(q_1,Q_2)} \xrightarrow{w}_i Q$ is also **inherited**.
  - If the accepting run of $G$ is derived from some $S' \in \widetilde{G}$ and $S'$ was added to $\widetilde{G}$ when inheriting transitions from $g^{i-1}_{(q_1,Q_2)}$, then the run $g^i_{(q_1,Q_2)} \xrightarrow{w}_i Q$ is **inherited**.
  - If the accepting run of $G$ is derived from some $S' \in \widetilde{G}$ and $S'$ was added to $\widetilde{G}$ by some $S \in \widetilde{G}^i_{(q_1,Q_2)}$ or — if $l = n - 1$ — by $T_{\mathcal{D}}$ and the command $d$, then the run $g^i_{(q_1,Q_2)} \xrightarrow{w}_i Q$ is **derived** from $S$ or $d$ respectively.

The first lemma states that runs derived from a set $S$ which is the conjunction of several automata behaves correctly:

**Lemma 5.4.1.** *The run $g^{i+1}_{(q_1,Q_2)} \xrightarrow{w}_{i+1} Q$ derived from $S = \{\theta_1, \ldots, \theta_m\} \subseteq \mathcal{B}_l \cup \widetilde{\mathcal{G}}^i_l$ exists in $\mathcal{G}^{i+1}_l$ iff the run,*

$$\{q^{\theta_1}, \ldots, q^{\theta_m}\} \xrightarrow{w}_i Q$$

*exists in $\mathcal{G}^i_l$.*

*Proof.* We prove the lemma by induction over $l$. Observe that a derived run cannot be empty. When $l = 1$, we have $w = aw'$ for some character $a$ and word $w'$ and,

$$g^{i+1}_{(q_1,Q_2)} \xrightarrow{a}_{i+1} Q'_1 \cup \cdots \cup Q'_m \xrightarrow{w'}_i Q$$

derived from $S$ exists iff the run $q^{\theta_t} \xrightarrow{a}_i Q'_t$ exists in $\mathcal{G}^i_1$ for all $t \in \{1, \ldots, m\}$ and the run,

$$\{q^{\theta_1}, \ldots, q^{\theta_m}\} \xrightarrow{a}_i Q'_1 \cup \cdots \cup Q'_m \xrightarrow{w'}_i Q$$

exists in $\mathcal{G}^i_1$ as required.

When $l > 1$, we have $w = \gamma w'$. The transition $g^{i+1}_{(q_1,Q_2)} \xrightarrow{\widetilde{G}}_{i+1} Q'$ derived from $S$ and the run $Q' \xrightarrow{w'}_i Q$ exist in $\mathcal{G}^{i+1}_l$ where $\gamma \in \mathcal{L}(G)$ iff $Q' = Q'_1 \cup \cdots \cup Q'_m$ and the transition $q^{\theta_t} \xrightarrow{B_t}_i Q_t$

exists in $\mathcal{G}_l^i$ for all $t \in \{1, \ldots, m\}$. We have $\{B_1, \ldots, B_m\} = S' \in \widetilde{G}^{i+1}_{(g^{i+1}_{(q_1,Q_2)}, Q')} \in \widetilde{\mathcal{G}}^i_{l-1}$ and the accepting run of $G$ is derived from $S'$ iff we have $\gamma \in \mathcal{L}(B_t)$ for all $t \in \{1, \ldots, m\}$ (by induction) and the run,

$$\{q^{\theta_1}, \ldots, q^{\theta_m}\} \xrightarrow{\gamma}_{i+1} Q' \xrightarrow{w'}_i Q$$

in $\mathcal{G}_l^i$ as required. □

The language accepted by the sequence $A_0, A_1, \ldots$ or any sequence $\mathcal{G}_l^0, \mathcal{G}_l^1, \ldots$ is increasing. In particular, if $q \xrightarrow{w}_i Q$ exists in $A_i$, then $q \xrightarrow{w}_{i+1} Q$ exists in $A_{i+1}$.

**Lemma 5.4.2.**

1. *If $g^i_{(q_1,Q_2)} \xrightarrow{w}_i Q$ is a run of $G^i_{(q_1,Q_2)}$ for some $i$ and, then $g^{i+1}_{(q_1,Q_2)} \xrightarrow{w}_{i+1} Q$ is a run of $G^{i+1}_{(q_1,Q_2)}$.*
2. *For all transitions $q \xrightarrow{\gamma}_i Q'$ in $A_i$ for some $i$, we have the transition $q \xrightarrow{\gamma}_{i+1} Q'$ in $A_{i+1}$.*
3. *For all runs $q \xrightarrow{w}_i Q'$ of $A_i$ for some $i$, we have the run $q \xrightarrow{w}_{i+1} Q'$ in $A_{i+1}$.*

*Proof.* To prove (2) we observe that there are two cases. In the first case, the transition from $q$ to $Q'$ is labelled by an automaton $B \in \mathcal{B}_l$ or $\triangledown$. Because this transition will remain unchanged by $T_{\mathcal{D}}$, the lemma follows immediately. In the second case, the transition is labelled by $G^i_{(q,Q')}$ and the property follows directly from (1) and the run $g^i_{(q,Q')} \xrightarrow{w_\gamma}_i Q$ with $Q \subseteq \mathcal{Q}_f$ for $[w_\gamma] = \gamma$. Since $g^i_{(q,Q')}$ is not an accepting state, it is the case that $w_\gamma \neq \varepsilon$.

We note that (3) can be shown by repeated applications of (2).

Finally, we show (1). The automaton $G^i_{(q_1,Q_2)}$ has the run,

$$g^i_{(q_1,Q_2)} \xrightarrow{\gamma}_i Q^1 \xrightarrow{w'}_i Q$$

where $w = \gamma w'$.

In case $\widetilde{G}^i_{(q_1,Q_2)} \in \widetilde{\mathcal{G}}^i_l$ and $l = 1$, by definition the automaton $G^{i+1}_{(q_1,Q_2)}$ has the transition $g^{i+1}_{(q_1,Q_2)} \xrightarrow{a}_i Q^2$ for every transition $g^i_{(q_1,Q_2)} \xrightarrow{a}_i Q^2$. Hence, as required, we have the run,

$$g^{i+1}_{(q_1,Q_2)} \xrightarrow{a}_{i+1} Q^1 \xrightarrow{w'}_i Q$$

When $\widetilde{G}^i_{(q_1,Q_2)} \in \widetilde{\mathcal{G}}^i_l$ and $l > 1$, by definition the automaton $G^{i+1}_{(q_1,Q_2)}$ has the transition $g^{i+1}_{(q_1,Q_2)} \xrightarrow{\widetilde{G}}_i Q^2$ with $\{B\} \in \widetilde{G}$ for every transition $g^i_{(q_1,Q_2)} \xrightarrow{B}_i Q^2$. Since we have $\gamma \in \mathcal{L}(B)$ (for some $B$) and $q^B$ is of the form $g^i_{(q,Q')}$ (and hence not an accepting state) we have $\gamma \neq [\varepsilon]$ and $\gamma \in \mathcal{L}(G)$ by Lemma 5.4.1 and $\{B\} \in \widetilde{G}$. Hence we have the run,

$$g^{i+1}_{(q_1,Q_2)} \xrightarrow{\gamma}_{i+1} Q^1 \xrightarrow{w'}_i Q$$

as required. □

We now show that inheritance does not introduce spurious runs:

**Lemma 5.4.3.** *If a run* $g^{i+1}_{(q_1,Q_2)} \xrightarrow{w}_{i+1} Q$ *in* $\mathcal{G}^{i+1}_l$ *is inherited, then the run* $g^{i}_{(q_1,Q_2)} \xrightarrow{w}_{i} Q$ *exists in* $\mathcal{G}^i_l$.

*Proof.* We proceed by induction over $l$. Observe that an inherited run cannot be empty.

When $l = 1$ we have $w = aw'$ and,

$$g^{i+1}_{(q_1,Q_2)} \xrightarrow{a}_{i+1} Q' \xrightarrow{w'}_{i} Q$$

Since the run is an inherited run, we have $g^{i}_{(q_1,Q_2)} \xrightarrow{a}_{i} Q'$ in $\mathcal{G}^i_1$ and hence,

$$g^{i}_{(q_1,Q_2)} \xrightarrow{a}_{i} Q' \xrightarrow{w'}_{i} Q$$

in $\mathcal{G}^i_1$ as required.

When $l > 1$ we have $w = \gamma w'$ and $g^{i+1}_{(q_1,Q_2)} \xrightarrow{\widetilde{G}^{i+1}_{(\ldots)}}_{i+1} Q'$ with $Q' \xrightarrow{w'}_{i} Q$ in $\mathcal{G}^{i+1}_l$ and $\gamma \in \mathcal{L}(G^{i+1}_{(\ldots)})$. There are two cases depending on the accepting run of $G^{i+1}_{(\ldots)}$.

If the accepting run of $G^{i+1}_{(\ldots)}$ is inherited, then we have $\gamma \in \mathcal{L}(G^{i}_{(\ldots)})$ by induction on $l$ and hence the transition $g^{i}_{(q_1,Q_2)} \xrightarrow{\gamma}_{i} Q'$ in $\mathcal{G}^i_l$. Therefore, as required, we have the run,

$$g^{i}_{(q_1,Q_2)} \xrightarrow{\gamma}_{i} Q' \xrightarrow{w'}_{i} Q$$

If the accepting run of $G^{i+1}_{(\ldots)}$ is derived from some $S$ introduced to $\widetilde{G}^{i+1}_{(\ldots)}$ when inheriting transitions from $g^{i}_{(q_1,Q_2)}$, then $S = \{B\}$ for some $B$ and we have $g^{i}_{(q_1,Q_2)} \xrightarrow{B}_{i} Q'$ in $\mathcal{G}^i_l$. Furthermore, by Lemma 5.4.1 we have $\gamma \in \mathcal{L}(B)$ and the run,

$$g^{i}_{(q_1,Q_2)} \xrightarrow{\gamma}_{i} Q' \xrightarrow{w'}_{i} Q$$

as required. □

The next two lemmata assert that derived runs are justified.

**Lemma 5.4.4.** *Suppose the run* $g^{i+1}_{(q_1,Q_2)} \xrightarrow{w}_{i+1} Q$ *derived from* $S$ *exists in* $\mathcal{G}^{i+1}_l$ *and* $\theta_1 \in S$. *We have* $q^{\theta_1} \xrightarrow{w}_{i} Q'$ *in* $\mathcal{G}^i_l$, *where* $Q' \subseteq Q$.

*Proof.* The proof is by induction over $l$. Observe that, since the run is derived, we have $w \neq \varepsilon$.

In the base case $l = 1$. Let $w = aw'$. We have the following run in $\mathcal{G}^{i+1}_l$,

$$g^{i+1}_{(q_1,Q_2)} \xrightarrow{a}_{i+1} Q^1 \xrightarrow{w'}_{i} Q$$

and by definition, since the run is derived from $S$ and $\theta_1 \in S$, we have $q^{\theta_1} \xrightarrow{a}_{i} Q^2$ in $\mathcal{G}^i_l$ where $Q^2 \subseteq Q^1$, and hence,

$$q^{\theta_1} \xrightarrow{a}_{i} Q^2 \xrightarrow{w'}_{i} Q'$$

with $Q' \subseteq Q$ as required.

When $l > 1$, let $w = \gamma w'$. We have the run,

$$g^{i+1}_{(q_1,Q_2)} \xrightarrow{\gamma}_{i+1} Q^1 \xrightarrow{w'}_{i} Q$$

in $\mathcal{G}_l^{i+1}$. In particular, we have $g^{i+1}_{(q_1,Q_2)} \xrightarrow{\widetilde{G}}_{i+1} Q^1$ with an accepting run of $G$ over $\gamma$ that is derived from $S' \in \widetilde{G}$ which was introduced by $S$. By definition, since $\theta_1 \in S$, we have $q^{\theta_1} \xrightarrow{\theta}_i Q^2$ with $Q^2 \subseteq Q^1$ and $\theta \in S'$. By induction, we have $\gamma \in \mathcal{L}(\theta)$ and hence the run,

$$q^{\theta_1} \xrightarrow{\gamma}_i Q^2 \xrightarrow{w'}_i Q'$$

with $Q' \subseteq Q$ as required. □

**Lemma 5.4.5.** *Suppose a run $g^{i+1}_{(q_1,Q_2)} \xrightarrow{w}_{i+1} Q$ derived from $S$ exists in $\mathcal{G}_l^{i+1}$ and $(a, o, \theta_1) \in S$. Let $[w'] = o([w])$, we have $q^{\theta_1} \xrightarrow{w'}_i Q'$ in $\mathcal{G}_l^i$, where $Q' \subseteq Q$.*

*Proof.* The proof is by induction over $l$. Since the run is derived, we have $w \neq \varepsilon$.

In the base case $l = 1$. We have $w = aw''$. There is only one value of $o$,

- $o = push_{w_p}$. Then $[w'] = o([w]) = [w_p w'']$. We have the following run in $\mathcal{G}_l^{i+1}$,

  $$g^{i+1}_{(q_1,Q_2)} \xrightarrow{a}_{i+1} Q^1 \xrightarrow{w''}_i Q$$

  and by definition, since the run is derived from $S$ and $(a, o, \theta_1) \in S$, we have $q^{\theta_1} \xrightarrow{w_p}_i Q^2$ in $\mathcal{G}_l^i$ where $Q^2 \subseteq Q^1$, and hence,

  $$q^{\theta_1} \xrightarrow{w_p}_i Q^2 \xrightarrow{w''}_i Q'$$

  with $Q'_f \subseteq Q$ as required.

When $l > 1$, let $w = \gamma w''$ and we have the run,

$$g^{i+1}_{(q_1,Q_2)} \xrightarrow{\gamma}_{i+1} Q^1 \xrightarrow{w''}_i Q$$

in $\mathcal{G}_l^{i+1}$. In particular $g^{i+1}_{(q_1,Q_2)} \xrightarrow{\widetilde{G}}_{i+1} Q^1$ and there is an accepting run of $G$ over $\gamma$ derived from some $S' \in \widetilde{G}$. There are now three cases depending on $o$,

- $o = push_l$. By definition, since $(a, push_l, \theta_1) \in S$ we have in $\mathcal{G}_l^i$ the run

  $$q^{\theta_1} \xrightarrow{\widetilde{\theta}_1}_i Q^2 \xrightarrow{\widetilde{\theta}_2}_i Q^3$$

  with $Q^3 \subseteq Q^1$ and $\{B_l^a\} \cup \widetilde{\theta}_1 \cup \widetilde{\theta}_2 \subseteq S'$. Hence, by Lemma 5.4.4, we have $\gamma \in \mathcal{L}(\{B_l^a\} \cup \widetilde{\theta}_1 \cup \widetilde{\theta}_2)$, and hence $[w'] = o([w]) = [\gamma\gamma w'']$ and we have the following run of $\mathcal{G}_l^i$,

  $$q^{\theta_1} \xrightarrow{\gamma}_i Q^2 \xrightarrow{\gamma}_i Q^3 \xrightarrow{w''}_i Q'$$

  with $Q' \subseteq Q$ as required.

- $o = pop_l$. Since $(a, pop_l, \theta_1) \in S$, we have $q^{\theta_1} \in Q^1$ and $B_l^a \in S'$. Furthermore $w''$ is non-empty since $q^{\theta_1} \notin \mathcal{Q}_f$ by definition of $n$-store automata. By Lemma 5.4.4 $\gamma \in \mathcal{L}(B_l^a)$. Hence $[w'] = o([w]) = [w'']$ and we have $q^{\theta_1} \xrightarrow{w''}_i Q'$ with $Q' \subseteq Q$ as required.

- $\ell(o) < l$. By definition, $[w'] = o([w]) = [o(\gamma)w'']$. Since $(a, o, \theta_1) \in S$ we have $q^{\theta_1} \xrightarrow{\theta}_i Q^2$ with $Q^2 \subseteq Q^1$ and $(a, o, \theta) \in S'$. By induction over $l$ we have $o(\gamma) \in \mathcal{L}(\theta)$ and hence,

$$q^{\theta_1} \xrightarrow{o(\gamma)}_i Q^2 \xrightarrow{w''}_i Q'$$

  with $Q' \subseteq Q$ as required.

This completes the proof of the lemma. □

Finally, we show that the derived runs completely represent their source.

**Lemma 5.4.6.** *Let $S = \{\alpha_1, \ldots, \alpha_m\} \in \widetilde{G}^{i+1}_{(q,Q)}$. Given some $\gamma$ with $top_1(\gamma) = a$ such that for each $e \in \{1, \ldots, m\}$ we have,*

- *If $\alpha_e = \theta_e$ then $\gamma_e = \gamma$ and $\gamma_e \in \mathcal{L}(\theta_e)$*
- *If $\alpha_e = (b, o_e, \theta_e)$ then $a = b$, $o_e(\gamma) = \gamma_e$ and $\gamma_e \in \mathcal{L}(\theta_e)$*

*we have $\gamma \in \mathcal{L}(G^{i+1}_{(q,Q)})$.*

*Proof.* We have $\widetilde{G}^{i+1}_{(q,Q)} \in \widetilde{\mathcal{G}}^{i+1}_l$ for some $l$. The proof is by induction over $l$.

When $l = 1$ let $\gamma = [aw]$. We have $\alpha_e = \theta_e$ or $\alpha_e = (a, push_{w_e}, \theta_e)$. We have,

- When $\alpha_e = \theta_e$, the run,

$$q^{\theta_e} \xrightarrow{a}_i Q_e \xrightarrow{w}_i Q^e_f$$

  with $Q^e_f \subseteq \mathcal{Q}_f$ in $\mathcal{G}^i_l$. Furthermore, $\gamma_e = \gamma$.

- When $\alpha_e = (a, push_{w_e}, \theta_e)$, the run,

$$q^{\theta_e} \xrightarrow{w_e}_i Q_e \xrightarrow{w}_i Q^e_f$$

  with $Q^e_f \subseteq \mathcal{Q}_f$ in $\mathcal{G}^i_l$. Furthermore, we have $\gamma_e = [w_e w]$.

Hence, since $S \in \widetilde{G}^{i+1}_{(q,Q)}$, we have from the definition of $G^{i+1}_{(q,Q)}$ the run,

$$g^{i+1}_{(q,Q)} \xrightarrow{a}_{i+1} Q_1 \cup \cdots \cup Q_m \xrightarrow{w}_i Q^1_f \cup \cdots \cup Q^m_f$$

with $Q^1_f \cup \cdots \cup Q^m_f \subseteq \mathcal{Q}_f$. Hence $\gamma \in \mathcal{L}(G^{i+1}_{(q,Q)})$ as required.

When $l > 1$ let $\gamma = [\gamma' w]$. We have $S' = S'_1 \cup \cdots \cup S'_m$ and $Q' = Q_1 \cup \cdots \cup Q_m$ where,

- When $\alpha_e = \theta_e$, $\gamma = \gamma_e$ and we have the transition $q^{\theta_e} \xrightarrow{\theta'_e}_i Q_e$ in $\mathcal{G}^i_l$ with $\gamma' \in \mathcal{L}(\theta'_e)$ and the run $Q_e \xrightarrow{w}_i Q^e_f$ with $Q_f \subseteq \mathcal{Q}_f$. Furthermore $S'_e = \{\theta'_e\}$.
- When $\alpha_e = (a, push_l, \theta_e)$, $\gamma_e = [\gamma'\gamma' w]$. Additionally, we have the transitions,

$$q^{\theta_e} \xrightarrow{\theta^1_e}_i Q' \xrightarrow{\widetilde{\theta}^2_e}_i Q_e$$

  in $\mathcal{G}^i_l$ where $\gamma' \in \mathcal{L}(\{B^a_{l-1}, \theta^1_e\} \cup \widetilde{\theta}^2_e)$. Furthermore, we have the run $Q_e \xrightarrow{w}_i Q^e_f$ with $Q^e_f \subseteq \mathcal{Q}_f$ and $S'_e = \{B^a_{l-1}, \theta^1_e\} \cup \widetilde{\theta}^2_e$.

- When $\alpha_e = (a, pop_l, \theta_e)$, $\gamma_e = [w]$ and we have the run,

$$q^{\theta_e} \xrightarrow{w}_i Q^e_f$$

with $Q^e_f \subseteq \mathcal{Q}_f$, $S'_e = \{B^a_{l-1}\}$, $\gamma' \in \mathcal{L}(B^a_{l-1})$ and $Q_e = \{q^{\theta_e}\}$.

- $\alpha_e = (a, o_e, \theta_e)$ with $\ell(o_e) < l$, we have $\gamma_e = [o_e(\gamma')w]$, and the transition $q^{\theta_e} \xrightarrow{\theta'_e}_i Q_e$ and run $Q_e \xrightarrow{w}_i Q^e_f$ with $Q^e_f \subseteq \mathcal{Q}_f$ in $\mathcal{G}^i_l$. Additionally, $o_e(\gamma') \in \mathcal{L}(\theta'_e)$ and $S'_e = \{(a, o_e, \theta'_e)\}$.

Hence, by definition of $G^{i+1}_{(q,Q)}$, we have the transition,

$$g^{i+1}_{(q,Q)} \xrightarrow{\widetilde{G}}_{i+1} Q_1 \cup \cdots \cup Q_m$$

with $S' \in \widetilde{G}$ and by induction over $l$, $\gamma' \in \mathcal{L}(G)$. Hence we have the run,

$$g^{i+1}_{(q,Q)} \xrightarrow{\gamma'}_{i+1} Q_1 \cup \cdots \cup Q_m \xrightarrow{w}_i Q^1_f \cup \cdots \cup Q^m_f$$

with $Q^1_f \cup \cdots \cup Q^m_f \subseteq \mathcal{Q}_f$ in $G^{i+1}_{(q,Q)}$. That is, $\gamma \in \mathcal{L}(G^{i+1}_{(q,Q)})$ as required. □

**Soundness**

We show that for any configuration $\langle p^j, \gamma\rangle$ such that $\gamma \in \mathcal{L}(A^{q^j}_i)$, for some $i$, we have $\langle p^j, \gamma\rangle \overset{*}{\hookrightarrow} C$ with $C \subseteq C_{Init}$. Let $I = \{q^1, \ldots, q^z\}$. The following lemma describes the relationship between added transitions and the evolution of the order-$n$ PDS. The restrictions on $w'$ are technical requirements in the case of $pop_n$ operations. They may be justified by observing that only the empty store is accepted from the state $q^\varepsilon_f$, and that, since initial states are never accepting, the empty store cannot be accepted from an initial state.

**Lemma 5.4.7.** *For a given run $q^j \xrightarrow{w}_i Q$ of $A_i$ there exists, for any $w'$ satisfying the conditions below, some $C$ such that $\langle p^j, [ww']\rangle \overset{*}{\hookrightarrow} C$, where $C$ contains configurations of the form $\langle p^k, w''w'\rangle$ with $q^k \xrightarrow{w''}_0 Q'$ or $\langle p^j, \triangledown\rangle$ with $q^j \xrightarrow{\triangledown}_0 Q'$. Furthermore, the union of all such $Q'$ is $Q$. We require $w' \neq \triangledown$ and,*

1. *If $q^\varepsilon_f \in Q$ then $w' = \varepsilon$,*
2. *If $q^k \in Q$ for some $q^k$ then $w' \neq \varepsilon$.*

*Proof.* The proof proceeds by induction on $i$. In the base case $i = 0$ and the property holds trivially. We now consider the case for $i + 1$. Since $T_{\mathcal{D}}$ does not add any $\triangledown$-transitions, we can assume $w \neq \triangledown$.

We perform a further induction over the length of the run. In the base case we have $w = \gamma$ (the case $w = \varepsilon$ is immediate with $C = \{\langle p^j, [w']\rangle\}$) and consider the single transition $q^j \xrightarrow{\gamma}_{i+1} Q$. We assume that the transition is not inherited, else the property holds by Lemma 5.4.3 and induction over $i$. If the transition is not inherited, then the run is derived from some $d$ and we have $\gamma \in \mathcal{L}(G^{i+1}_{(q^j,Q)})$ and the accepting run of $G^{i+1}_{(q^j,Q)}$ is derived from some $S \in \widetilde{G}^{i+1}_{(q^j,Q)}$ introduced during the processing of $d$.

Let $d = (p^j, a, \{(o_1, p^{k_1}), \ldots, (o_m, p^{k_m})\})$. We have $\langle p^j, [\gamma w'] \rangle \hookrightarrow C'$ where,

$$\begin{array}{rcl} C' & = & \{ \langle p^{k_t}, \gamma' \rangle \mid t \in \{1, \ldots, m\} \wedge \gamma' = o_t([\gamma w']) \} \\ & & \cup \{ \langle p^j, \triangledown \rangle \mid \text{if } o_t([\gamma w']) \text{ with } t \in \{1, \ldots, m\} \text{ is not defined } \} \end{array}$$

We can decompose the new transition as per the definition of $T_{\mathcal{D}}$. That is $Q = Q'_1 \cup \cdots \cup Q'_m$. There are several cases:

- $o_t = push_n$.

  By definition of $T_{\mathcal{D}}$, we have the run,

  $$q^{k_t} \xrightarrow{\widetilde{\theta}_1}_i Q' \xrightarrow{\widetilde{\theta}_2}_i Q'_t$$

  with $\{B^a_{n-1}\} \cup \widetilde{\theta}_1 \cup \widetilde{\theta}_2 \subseteq S$. By Lemma 5.4.4 we have $\gamma \in \mathcal{L}(\{B^a_{n-1}\} \cup \widetilde{\theta}_1 \cup \widetilde{\theta}_2)$. Hence we have,

  $$q^{k_t} \xrightarrow{\gamma}_i Q' \xrightarrow{\gamma}_i Q'_t$$

  We have $push_n[\gamma w'] = [\gamma\gamma w']$ and $\langle p^{k_t}, [\gamma\gamma w'] \rangle \in C'$. Via induction over $i$ we have the set $C_t$ with $\langle p^{k_t}, o_t[\gamma w'] \rangle \overset{*}{\hookrightarrow} C_t$ which satisfies the lemma.

- $o_t = pop_n$.

  We have $B^a_{n-1} \in S$. We have, by Lemma 5.4.4, $\gamma \in \mathcal{L}(B^a_{n-1})$.

  If $Q_t = \{q^{k_t}\}$ we have $pop_n[\gamma w'] = [w']$ since $w'$ is non-empty and $C_t = \{\langle p^{k_t}, [w'] \rangle\}$. Note $q^{k_t} \xrightarrow{\varepsilon}_0 \{q^{k_t}\}$.

  If $Q_t = \{q^\varepsilon_f\}$ then $w' = \varepsilon$ and $pop_n[\gamma w']$ is undefined. By definition of $T_{\mathcal{D}}$ we have $q^j \xrightarrow{\triangledown}_0 \{q^\varepsilon_f\}$. Let $C_t = \{\langle p^j, \triangledown \rangle\}$.

- $\ell(o_t) < n$ and if $o_t = pop_l$ then $X^a_{\ell(o_t)} \notin S$.

  By definition, we have $q^{k_t} \xrightarrow{\theta}_i Q_t$ in $A_i$ with $(a, o_t, \theta) \in S$. Therefore, by Lemma 5.4.5, we have $o_t[\gamma] \in \mathcal{L}(\theta)$ and the run $q^{k_t} \xrightarrow{o_t[\gamma]}_i Q_t$ in $A_i$.

  Furthermore, we have $\langle p^{k_t}, o_t[\gamma w'] \rangle \in C'$ and via induction over $i$ we have a set $C_t$ with $\langle p^{k_t}, o_t[\gamma w'] \rangle \overset{*}{\hookrightarrow} C_t$ which satisfies the lemma.

- $\ell(o_t) < n$, $o_t = pop_l$ and $X^a_{\ell(o_t)} \in S$.

  Since $X^a_l \in S$ by Lemma 5.4.4 we have $\gamma \in \mathcal{L}(X^a_l)$. Hence $o_t[\gamma w']$ is undefined and we have $\langle p^j, \triangledown \rangle \in C'$. Because $X^a_l \in S$, by definition we have $q^j \xrightarrow{\triangledown}_i \{q^\varepsilon_f\}$. Since $\triangledown$ transitions are never added, it must be the case that $q^j \xrightarrow{\triangledown}_0 \{q^\varepsilon_f\}$. Let $C_t = \{\langle p^j, \triangledown \rangle\}$.

Hence, we have $\langle p^j, [ww'] \rangle \hookrightarrow C' \overset{*}{\hookrightarrow} C_1 \cup \cdots \cup C_m = C$ where $C$ satisfies the lemma.

This completes the proof of the single transition case. Let $w = \gamma_1 \ldots \gamma_m$ and (for any $Q$) let $Q = Q^I \cup Q^{\setminus I}$ where $Q^I$ contains all initial states in $Q$ and $Q^{\setminus I} = Q \setminus Q^I$. We have the run,

$$q^j \xrightarrow{\gamma_1}_{i+1} Q_1 \xrightarrow{\gamma_2}_{i+1} \ldots \xrightarrow{\gamma_m}_{i+1} Q_m$$

For each $q^k \in Q^I_1$ we have a run,

$$q^k \xrightarrow{\gamma_2}_{i+1} Q^k_2 \xrightarrow{\gamma_3}_{i+1} \ldots \xrightarrow{\gamma_m}_{i+1} Q^k_m$$

and by induction on the length of the run we have $C_k$ such that $\langle p^k, [\gamma_2 \dots \gamma_m w']\rangle \overset{*}{\hookrightarrow} C_k$ and $C_k$ satisfies the lemma. Furthermore, since we only add new transitions to initial states, we have,

$$Q_1^{\setminus I} \xrightarrow{\gamma_2}_0 \dots \xrightarrow{\gamma_m}_0 Q'_m$$

and $Q_m = Q'_m \cup \bigcup_{q^k \in Q_1^I} Q_m^k$.

From $q^j \xrightarrow{\gamma_1}_{i+1} Q_1$ we have $C_1$ with $\langle p^j, [\gamma_1 \dots \gamma_m w']\rangle \overset{*}{\hookrightarrow} C_1$ satisfying the lemma for a single transition. Let $C_1^I$ be the set of all $\langle p^k, \gamma_2 \dots \gamma_m w'\rangle \in C_1$ and $C'_1 = C_1 \setminus C_1^I$. For each $q^k \in Q_1^I$ we have $\langle p^k, [\gamma_2 \dots \gamma_m w']\rangle \in C_1$ since there are no transitions to initial states in $A_0$ (and hence we must have $q^k \xrightarrow{\varepsilon}_0 \{q^k\}$ to satisfy the conditions of the lemma for $C_1$). From $\langle p^k, [\gamma_2 \dots \gamma_m w']\rangle \overset{*}{\hookrightarrow} C_k$ and since we have $Q_1^{\setminus I} \xrightarrow{\gamma_2 \dots \gamma_m}_0 Q'_m$, it is the case that the set $C = C'_1 \cup \bigcup_{q^k \in Q_1^I} C_k$ which has $\langle p^j, [\gamma_1 \dots \gamma_m w']\rangle \overset{*}{\hookrightarrow} C_1 \overset{*}{\hookrightarrow} C$ and satisfies the lemma as required. □

**Property 5.4.2** (Soundness)**.** *For any configuration $\langle p^j, \gamma\rangle$ such that $\gamma \in \mathcal{L}(A_i^{q^j})$ for some $i$, we have $\langle p^j, \gamma\rangle \overset{*}{\hookrightarrow} C$ such that $C \subseteq C_{Init}$. That is, $\langle p^j, \gamma\rangle \in Pre^*(C_{Init})$.*

*Proof.* Let $\gamma = [w_\gamma]$. Since $\gamma \in \mathcal{L}(A_i^{q^j})$ we have a run $q^j \xrightarrow{w_\gamma}_i Q_f$ with $Q_f \subseteq \mathcal{Q}_f$. Since $\mathcal{Q}_f$ contains no initial states, we apply Lemma 5.4.7 with $w' = \varepsilon$. Therefore, we have $\langle p^j, \gamma\rangle \overset{*}{\hookrightarrow} C \subseteq \mathcal{L}(A_0^{q^k})$. Since $A_0$ is defined to represent $C_{Init}$, soundness follows. □

### Completeness

**Property 5.4.3** (Completeness)**.** *For all $\langle p^j, \gamma\rangle \in Pre^*(C_{Init})$ there is some $i$ such that $\gamma \in \mathcal{L}(A_i^{q^j})$.*

*Proof.* We take $\langle p^j, \gamma\rangle \in Pre^*(C_{Init})$ and reason by induction over the length of the shortest path $\langle p^j, \gamma\rangle \overset{*}{\hookrightarrow} C$ with $C \subseteq C_{Init}$.

In the base case the path length is zero and we have $\langle p^j, \gamma\rangle \in C_{Init}$ and hence $\gamma \in \mathcal{L}(A_0^{q^j})$. For the inductive step we have $\langle p^j, \gamma\rangle \hookrightarrow C_1 \overset{*}{\hookrightarrow} C_2$ with $C_2 \subseteq C_{Init}$ and some $i$ such that $C_1 \subseteq \mathcal{L}(A_i)$ by induction. We show $\gamma \in \mathcal{L}(A_{i+1}^{q^j})$ by analysis of the higher-order APDS command $d$ used in the transition $\langle p^j, \gamma\rangle \hookrightarrow C_1$.

Let $d = (p^j, a, \{(o_1, p^{k_1}), \dots, (o_m, p^{k_m})\})$. We have

$$\begin{array}{rcl} C_1 & = & \{\ \langle p^{k_t}, \gamma'\rangle \mid t \in \{1, \dots, m\} \wedge\ \gamma' = o_t(\gamma)\ \} \\ & & \cup\ \{\ \langle p^j, \triangledown\rangle \mid \text{if } o_t(\gamma) \text{ with } t \in \{1, \dots, m\} \text{ is not defined } \} \end{array}$$

By induction we have for each $e \in \{1, \dots, m\}$ that $q^{k_e} \xrightarrow{w_{o_e(\gamma)}}_i Q_f^e$ with $Q_f^e \subseteq \mathcal{Q}_f$ in $A_i$ if $o_e(\gamma) = [w_{o_e(\gamma)}]$ is defined. Otherwise we have $q^j \xrightarrow{\triangledown}_i \{q_f^\varepsilon\}$ in $A_i$.

Let $\gamma = [\gamma' w]$. We have $S' = S'_1 \cup \dots \cup S'_m$ and $Q' = Q_1 \cup \dots \cup Q_m$ where, for each $e \in \{1, \dots, m\}$,

- When $o_e = push_n$, $o_e(\gamma) = [\gamma'\gamma' w]$. Additionally, we have the transitions,

$$q^{k_e} \xrightarrow{\theta_e^1}_i Q' \xrightarrow{\widetilde{\theta}_e^2}_i Q_e$$

in $A_i$ where $\gamma' \in \mathcal{L}(\{B_{n-1}^a, \theta_e^1\} \cup \widetilde{\theta}_e^2)$. Furthermore, we have the run $Q_e \xrightarrow{w}_i Q_f^e$ with $Q_f^e \subseteq \mathcal{Q}_f$ and $S'_e = \{B_{n-1}^a, \theta_e^1\} \cup \widetilde{\theta}_e^2$.

- When $o_e = pop_n$. If $o_e(\gamma) = [w]$, we have the run,

$$q^{k_e} \xrightarrow{w}_i Q_f^e$$

in $A_i$ with $Q_f^e \subseteq \mathcal{Q}_f$, $S'_e = \{B_{n-1}^a\}$, $\gamma' \in \mathcal{L}(B_{n-1}^a)$ and $Q_e = \{q^{k_e}\}$.
If $o_e(\gamma)$ is undefined we have $w = \varepsilon$ and the run,

$$q^j \xrightarrow{\nabla}_i \{q_f^\varepsilon\}$$

if $A_i$. Hence we have $S'_e = \{B_{n-1}^a\}$, $\gamma' \in \mathcal{L}(B_{n-1}^a)$ and $Q_e = Q_f^e = \{q_f^\varepsilon\}$.

- When $\ell(o_e) < n$, and we have $o_e(\gamma) = [o_e(\gamma')w]$, we have the transition $q^{k_e} \xrightarrow{\theta'_e}_i Q_e$ and run $Q_e \xrightarrow{w}_i Q_f^e$ with $Q_f^e \subseteq \mathcal{Q}_f$ in $A_i$. Additionally, $o_e(\gamma') \in \mathcal{L}(\theta'_e)$ and $S'_e = \{(a, o_e, \theta'_e)\}$.
If $o_e(\gamma)$ is not defined we have $\gamma = [\gamma' w]$, $o_e = pop_l$ for some $l < n$ and the run,

$$q^j \xrightarrow{\nabla}_i \{q_f^\varepsilon\}$$

in $A_i$. Hence we have $S'_e = \{X_l^a\}$, $\gamma' \in \mathcal{L}(X_l^a)$ and $Q_e = \{q_f^*\}$ with $q_f^* \xrightarrow{w}_i Q_f^e$ and $Q_f^e \subseteq \mathcal{Q}_f$.

Hence, by definition of $A_{i+1}$, we have the transition,

$$q^j \xrightarrow{\widetilde{G}}_{i+1} Q_1 \cup \cdots \cup Q_m$$

with $S' \in \widetilde{G}$ and by Lemma 5.4.6 $\gamma' \in \mathcal{L}(G)$. Hence we have the run,

$$q^j \xrightarrow{\gamma'}_{i+1} Q_1 \cup \cdots \cup Q_m \xrightarrow{w}_i Q_f^1 \cup \cdots \cup Q_f^m$$

with $Q_f^1 \cup \cdots \cup Q_f^m \subseteq \mathcal{Q}_f$ in $A_{i+1}$. That is, $\gamma \in \mathcal{L}(A_{i+1}^{q^j})$ as required. □

### 5.4.5 Constructing $A_*$

We have shown how to construct a sequence of automata $A_0, A_1, \ldots$ which is sound and complete with respect to $Pre^*(\mathcal{L}(A_0))$. In this section we show how to compute an automaton $A_*$ that is a finite representation of this sequence. That is, $\mathcal{L}(A_*) = Pre^*(\mathcal{L}(A_0))$. We begin by defining $\lesssim$, generalising the notion of subset modulo $i$ for the sets $\widetilde{G}^i_{(q,Q')}$.

**Definition 5.4.4.**

1. Given $\theta \in \mathcal{B}_l \cup \widetilde{\mathcal{G}}^i_l$ for some $i$ and $l$, let

$$\theta[j/i] = \begin{cases} \theta & \text{if } \theta \in \mathcal{B}_l \\ G^j_{(q_1,Q_2)} & \text{if } \theta = G^i_{(q_1,Q_2)} \in \widetilde{\mathcal{G}}^i_l \end{cases}$$

2. For a set $S$ we define $S[j/i]$ such that,
   (a) We have $\theta \in S$ iff we have $\theta[j/i] \in S[j/i]$, and
   (b) We have $(a, o, \theta) \in S$ iff we have $(a, o, \theta[j/i]) \in S[j/i]$.

3. We extend the notation $[j/i]$ to nested sets of sets structures in a point-wise fashion.

**Definition 5.4.5.**

1. We write $\widetilde{G}^i_{(q,Q')} \lesssim \widetilde{G}^j_{(q,Q')}$ iff for each $S \in \widetilde{G}^i_{(q,Q')}$ we have $S[j-1/i-1] \in \widetilde{G}^j_{(q,Q')}$.
2. If $\widetilde{G}^i_{(q,Q')} \lesssim \widetilde{G}^j_{(q,Q')}$ and $\widetilde{G}^j_{(q,Q')} \lesssim \widetilde{G}^i_{(q,Q')}$, then we write $\widetilde{G}^i_{(q,Q')} \simeq \widetilde{G}^j_{(q,Q')}$.
3. Furthermore, we extend the notation to sets. That is, $\widetilde{\mathcal{G}}^i_l \lesssim \widetilde{\mathcal{G}}^j_l$ iff for all $\widetilde{G}^i_{(q_1,Q_2)} \in \widetilde{\mathcal{G}}^i_l$ we have $\widetilde{G}^j_{(q_1,Q_2)} \in \widetilde{\mathcal{G}}^j_l$ and $\widetilde{G}^i_{(q,Q')} \lesssim \widetilde{G}^j_{(q,Q')}$.

A fixed point is reached as in the order-2 case: eventually, no more transitions are added at order-$n$ of the automaton, allowing us to fix the state-set at order-$(n-1)$. This begins a cascade of fixed points which ends at order-1, resulting in termination.

**Property 5.4.4.** *There exists $i_{n-1} > 0$ such that $\widetilde{\mathcal{G}}^i_{n-1} \simeq \widetilde{\mathcal{G}}^{i_{n-1}}_{n-1}$ for all $i \geq i_{n-1}$.*

*Proof.* (Sketch) Since the order-$n$ state-set in $A_i$ remains constant and we add at most one transition between any state $q$ and set of states $Q$, there is some $i_{n-1}$ where no more transitions are added at order-$n$. That $\widetilde{\mathcal{G}}^i_{n-1} \simeq \widetilde{\mathcal{G}}^{i_{n-1}}_{n-1}$ for all $i \geq i_{n-1}$ follows since the contents of any $\widetilde{G}^i_{(q,Q')}$ and $\widetilde{G}^{i_{n-1}}_{(q,Q')}$ are derived from the same transition structure. □

Once a fixed point has been reached at order-$(l+1)$, we can fix the state-set at order-$l$.

**Lemma 5.4.8.** *Suppose we have constructed a sequence of automata $\mathcal{G}^0_l, \mathcal{G}^1_l, \ldots$ and associated sets $\widetilde{\mathcal{G}}^0_l, \widetilde{\mathcal{G}}^1_l, \ldots$. Further, suppose there exists an $i_l$ such that for all $i \geq i_l$ we have $\widetilde{\mathcal{G}}^i_l \simeq \widetilde{\mathcal{G}}^{i_l}_l$. We can define a sequence of automata $\hat{\mathcal{G}}^{i_l}_l, \hat{\mathcal{G}}^{i_l+1}_l, \ldots$ such that the state-set in $\hat{\mathcal{G}}^i_l$ remains constant (although the automata labelling the transitions may gain states when $l > 1$). The following are equivalent for all $w$,*

1. *The run $g^{i_l}_{(q,Q')} \xrightarrow{w}_i Q$ with $Q \subseteq \mathcal{Q}_f$ exists in $\hat{\mathcal{G}}^i_l$ for some $i$.*
2. *The run $g^{i'}_{(q,Q')} \xrightarrow{w}_{i'} Q''$ with $Q'' \subseteq \mathcal{Q}_f$ exists in $\mathcal{G}^{i'}_l$ for some $i'$.*

Again, we use $\hat{\mathcal{G}}^{i+1} = T_{\widetilde{\mathcal{G}}^{i_l}_l[i_l/i_1-l]}(\hat{\mathcal{G}}^i_l)$ to construct the sequence $\hat{\mathcal{G}}^{i_l}_l, \hat{\mathcal{G}}^{i_l+1}_l, \ldots$ with $\hat{\mathcal{G}}^{i_l}_l = \mathcal{G}^{i_l}_l$. Soundness and completeness are shown in Section 5.4.6. Once the state-set has been fixed at order-$l$, we will eventually reach another fixed point. In this way the fixed points cascade.

**Property 5.4.5.** *For a sequence of automata $\mathcal{G}^0_l, \mathcal{G}^1_l, \ldots$ such that the state-set at order-$l$ of $\mathcal{G}^i_l$ remains constant there exists $i_{l-1} > 0$ such that $\widetilde{\mathcal{G}}^i_{l-1} \simeq \widetilde{\mathcal{G}}^{i_{l-1}}_{l-1}$ for all $i \geq i_{l-1}$.*

*Proof.* (Sketch) Since the order-$l$ state-set in $\widetilde{\mathcal{G}}^i_l$ remains constant and we add at most one transition between any state $q$ and set of states $Q$, there is some $i_{l-1}$ where no more transitions are added at order-$l$. That $\widetilde{\mathcal{G}}^i_{l-1} \simeq \widetilde{\mathcal{G}}^{i_{l-1}}_{l-1}$ for all $i \geq i_{l-1}$ follows since the contents of any $\widetilde{G}^i_{(q,Q')}$ and $\widetilde{G}^{i_{l-1}}_{(q,Q')}$ are derived from the same transition structure. □

When the state-set has been fixed at order-1, the finiteness of $\Sigma$ dictates that only a finite number of transitions can be added before saturation is reached. At this point any updates to the automata at order-1 will not change the set of accepted 1-stores. Termination at order-1 implies termination at order-2, and so on down to order-$n$. That is, for any $l$, we are able to define an automaton $\mathcal{G}^*_l$ which finitely represents the infinite sequence $\mathcal{G}^0_l, \mathcal{G}^1_l, \ldots$ used to construct $A_0, A_1, \ldots$.

**Lemma 5.4.9.** *Suppose we have constructed a sequence of automata $\mathcal{G}_l^0, \mathcal{G}_l^1, \ldots$ and associated sets $\widetilde{\mathcal{G}}_l^0, \widetilde{\mathcal{G}}_l^1, \ldots$. Further, suppose there exists an $i_l$ such that for all $i \geq i_l$ we have $\widetilde{\mathcal{G}}_l^i \simeq \widetilde{\mathcal{G}}_l^{i_l}$. We can define an automaton $\mathcal{G}_l^*$ such that the following are equivalent for all $w$,*

1. *The run $g^*_{(q,Q')} \xrightarrow{w}_* Q$ with $Q \subseteq \mathcal{Q}_f$ exists in $\mathcal{G}_l^*$.*
2. *The run $g^i_{(q,Q')} \xrightarrow{w}_i Q''$ with $Q'' \subseteq \mathcal{Q}_f$ exists in $\mathcal{G}_l^i$ for some $i$.*

*Proof.* The complete proof is given in Section 5.4.6. We show here how to construct $\mathcal{G}_l^*$.

We proceed by induction over $l$. In the base case $l = 1$ and the result follows because the alphabet is finite: there is a finite bound on the number of new transitions that can be added. Let $\mathcal{G}^{i_0}$ denote this saturation point. We define $\mathcal{G}_1^* = \mathcal{G}_1^{i_0}$ letting $g^*_{(q,Q)} = g^{i_1}_{(q,Q)}$ for each $g^{i_1}_{(q,Q)}$.

When $l > 1$ we generate the sequence $\hat{\mathcal{G}}_l^{i_l}, \hat{\mathcal{G}}_l^{i_l+1}, \ldots$ by Lemma 5.4.8. Since the state-set remains constant, it follows from Property 5.4.5 that there is some $i_{l-1}$ with $\widetilde{G}^i_{l-1} \simeq \widetilde{G}^{i_{l-1}}_{l-1}$ for all $i \geq i_{l-1}$.

By induction, we have $\mathcal{G}^*_{l-1}$. We then define $\mathcal{G}_l^*$ from $\hat{\mathcal{G}}_l^{i_{l-1}}$ with $g^*_{(q,Q')} = g^{i_l}_{(q,Q')}$ for all $q, Q'$ and each transition $q \longrightarrow_* Q'$ in $\mathcal{G}_l^*$ labelled with the appropriate $B \in \mathcal{B}_{n-1}$ or automaton $G^*_{(q,Q')}$ from $\mathcal{G}^*_{l-1}$. □

Finally, we have the following algorithm for constructing $A_*$:

1. Given $A_0$, iterate $A_{i+1} = T_{\mathcal{D}}(A_i)$ until the fixed point $A_{i_{n-1}}$ is reached.
2. For $l = n-1$ down to $l = 1$: iterate $\mathcal{G}_l^{i+1} = T_{\widetilde{\mathcal{G}}_l^{i_l}[i_l/i_l-1]}(\mathcal{G}_l^i)$ to generate the fixed point $\mathcal{G}_l^{i_{l-1}}$ from $\mathcal{G}_l^{i_l}$.
3. For $l = 1$ to $l = n-1$: construct $G_l^*$ as in Lemma 5.4.9.
4. Construct $A_*$ as in Property 5.4.6.

**Property 5.4.6.** *There exists an automaton $A_*$ which is sound and complete with respect to $A_0, A_1, \ldots$ and hence computes the set $Pre^*(C_{Init})$.*

*Proof.* By Property 5.4.4 there is some $i_{n-1}$ with $\widetilde{\mathcal{G}}^i_{n-1} \simeq \widetilde{\mathcal{G}}^{i_{n-1}}_{n-1}$ for all $i \geq i_{n-1}$. By Lemma 5.4.9, we have $\mathcal{G}^*_{n-1}$. We then define $A_*$ from $A_{i_{n-1}}$ with each transition $q \longrightarrow_* Q'$ in $A_*$ labelled with the automaton $G^*_{(q,Q')}$ from $\mathcal{G}^*_{n-1}$. □

### 5.4.6 Proofs for $A_*$

In this section we provide proofs of Lemma 5.4.8 and Lemma 5.4.9. The proof of the first lemma is somewhat involved, hence we deal with the order-1 and the order-$l$ for $l > 1$ cases individually. The main idea in both proofs is that the loops in $\hat{\mathcal{G}}_l^i$ can simulate, correctly, the prefix of any run in $\mathcal{G}_l^{i'}$ and vice-versa. That is, a run in $\hat{\mathcal{G}}_l^i$ begins by traversing its initial loops before progressing to its accepting states. If we unroll this looping we will construct a run of $\mathcal{G}_l^{i'}$ for a sufficiently large $i'$. In the other direction, the prefix of a run in $\mathcal{G}_l^{i'}$ can be simulated by the initial looping behaviour of $\hat{\mathcal{G}}_l^i$.

We begin by proving a small lemma that will ease the remaining proofs.

**Lemma 5.4.10.** *Given $g^{i_y}_{(q_y,Q_y)} \xrightarrow{w}_{i_y} Q_y$ for all $y \in \{1, \ldots, h\}$ for some $h$, let $i_{max}$ be the maximum $i_y$. We have $\{g^{i_{max}}_{(q_1,Q_1)}, \ldots, g^{i_{max}}_{(q_h,Q_h)}\} \xrightarrow{w} \bigcup_{y\in\{1,\ldots,h\}} Q_y$.*

*Proof.* By Lemma 5.4.2 we have $g^{i_{max}}_{(q_y,Q_y)} \xrightarrow{w}_{i_{max}} Q_y$ for each $y \in \{1, \ldots, h\}$. Hence we have the run as required. □

**Proof of Lemma 5.4.8 for $l = 1$**

There are three parts to the proof. We first prove that a fixed point $i_0$ is reached. We then prove Lemma 5.4.8 in both directions.

**Lemma 5.4.11.** *There exists some $i_0$ such that $\hat{\mathcal{G}}^i_1 = \hat{\mathcal{G}}^{i_0}_1$ for all $i > i_0$. Furthermore, we have the run $g^{i_1}_{(q,Q')} \xrightarrow{w}_i Q_f$ with $Q_f \subseteq \mathcal{Q}_f$ for some $i$ iff we have $g^{i_1}_{(q,Q')} \xrightarrow{w}_{i_0} Q_f$ in $\hat{\mathcal{G}}^{i_0}_1$.*

*Proof.* This is a simple consequence of the finiteness of $\Sigma$ and that $T_{\widetilde{\mathcal{G}}^{i_1}_1[i_1/i_1-1]}$ only adds transitions and never states. The automaton will eventually become saturated and no new transitions will be added. □

**Lemma 5.4.12.** *For all $w$, if $g^i_{(q,Q')} \xrightarrow{w}_i Q_1$ with $Q_1 \subseteq \mathcal{Q}_f$ is a run in $\mathcal{G}^i_1$ for some $i$, then we have $g^{i_1}_{(q,Q')} \xrightarrow{w}_{i_0} Q_2$ with $Q_2 \subseteq \mathcal{Q}_f$ in $\hat{\mathcal{G}}^{i_0}_1$.*

*Proof.* We prove the following property. For any path $g^i_{(q,Q')} \xrightarrow{w}_i \{q_1, \ldots, q_h\}$ in $\mathcal{G}^i_1$, we have a path $g^{i_1}_{(q,Q')} \xrightarrow{w}_{i_0} \{q^!_1, \ldots, q^!_h\}$ in $\hat{\mathcal{G}}^{i_0}_1$ with,

$$q^!_y = \begin{cases} g^{i_1}_{(q',Q'')} & \text{if } q_y = g^{i'}_{(q',Q'')} \text{ and } i' \geq i_1 \\ q_y & \text{otherwise} \end{cases}$$

for all $y \in \{1, \ldots, h\}$. Since $q^!_f = q_f$ for all $q_f \in \mathcal{Q}_f$, the lemma follows. When $Q = \{q_1, \ldots, q_h\}$ we write $Q^!$ to denote the set $\{q^!_1, \ldots, q^!_h\}$.

There are two cases. When $i \leq i_1$, then using that we have only added transitions to $\mathcal{G}^{i_1}_1$ to define $\hat{\mathcal{G}}^{i_0}_1$ and that $q^!_y = q_y$ for all $y$, we have $g^{i_1}_{(q,Q')} \xrightarrow{w'}_{i_0} \{q^!_1, \ldots, q^!_h\}$ in $\hat{\mathcal{G}}^{i_0}_1$.

We now consider the case $i > i_1$. We begin by proving that for a single transition,

$$g^i_{(q,Q')} \xrightarrow{b}_i \{q_1, \ldots, q_h\}$$

in $\mathcal{G}^i_1$ with $b \in \Sigma$, we have the following transition in $G^{i_0}_{(q,Q')}$,

$$g^{i_1}_{(q,Q')} \xrightarrow{b}_{i_0} \{q^!_1, \ldots, q^!_h\}$$

We consider the source $S = \{\alpha_1, \ldots, \alpha_m\} \in \widetilde{G}^i_{(q,Q')}$ of the transition from $g^i_{(q,Q')}$. Since $\widetilde{G}^i_{(q,Q')} \simeq \widetilde{G}^{i_1}_{(q,Q')}$ we have $S[i_1/i-1] \in \widetilde{G}^{i_1}_{(q,Q')}[i_1/i_1-1]$. Furthermore, we have $\{q_1, \ldots, q_h\} = Q_1 \cup \cdots \cup Q_m$. For $e \in \{1, \ldots, m\}$ there are two cases,

- If $\alpha_e = \theta$, then let $g = q^\theta$. We have $g \xrightarrow{b}_{i-1} Q_e$ exists in $\mathcal{G}^{i-1}_1$. By induction over $i$ we have $g^! \xrightarrow{b}_{i_0} Q^!_e$ in $\hat{\mathcal{G}}^{i_0}_1$.

- $\alpha_e = (a, push_{w_p}, \theta)$. Then $b = a$. Let $g = q^\theta$. By definition of $T_{\widetilde{\mathcal{G}}^{i_1}[i_1/i_1-1]}$, we have the path $g \xrightarrow{w_p}_{i-1} Q_e$ in $\mathcal{G}^i_1$. By induction on $i$ we have the path $g^! \xrightarrow{w_p}_{i_0} Q^!_e$ in $\hat{\mathcal{G}}^{i_0}_1$.

We have $Q^!_1 \cup \dots \cup Q^!_m = \{q^!_1, \dots, q^!_h\}$. Since $\widetilde{G}^i_{(q,Q')} \simeq \widetilde{G}^{i_1}_{(q,Q')}$ and $S[i_1/i-1] \in \widetilde{G}^{i_1}_{(q,Q')}[i_1/i_1-1]$, by definition of $\hat{\mathcal{G}}^{i_0}_1$, we have,

$$g^{i_1}_{(q,Q')} \xrightarrow{b}_{i_0} \{q^!_1, \dots, q^!_h\}$$

in $\hat{\mathcal{G}}^{i_0}_1$ as required.

We now prove the result for a run of more than one step by induction over the length of the run. In the base case we have a run of a single transition. The result in this case has already been shown. In the inductive case we have a run of the form,

$$g^i_{(q,Q')} \xrightarrow{a_0}_i \{q^1_1, \dots, q^1_{h_1}\} \xrightarrow{a_1}_i \dots \xrightarrow{a_m}_i \{q^m_1, \dots, q^m_{h_m}\}$$

in $\mathcal{G}^i_1$. For each $y \in \{1, \dots, h_1\}$ we have a run $q^1_y \xrightarrow{a_1 \dots a_m}_i Q_y$ such that $\bigcup_{y \in \{1,\dots,h_1\}} Q_y = \{q^m_1, \dots, q^m_{h_m}\}$. By induction over the length of the run we have $q^{!1}_y \xrightarrow{a_1 \dots a_m}_{i_0} Q^!_y$ for each $y$. Hence, since we have $g^{i_1}_{(q,Q')} \xrightarrow{a_0}_{i_0} \{q^{!1}_1, \dots, q^{!1}_{h_1}\}$ from the above proof for one transition, we have a run of the form,

$$g^{i_1}_{(q,Q')} \xrightarrow{a_0}_{i_0} \{q^{!1}_1, \dots, q^{!1}_{h_1}\} \xrightarrow{a_1}_{i_0} \dots \xrightarrow{a_m}_{i_0} \{q^{!m}_1, \dots, q^{!m}_{h_m}\}$$

in $\hat{\mathcal{G}}^{i_0}_1$ as required. □

**Lemma 5.4.13.** *For all $w$, if we have $g^{i_1}_{(q,Q')} \xrightarrow{w}_i Q_f$ with $Q_f \subseteq \mathcal{Q}_f$ in $\hat{\mathcal{G}}^i_1$ for some $i$, then there is some $i'$ such that the run $g^{i'}_{(q,Q')} \xrightarrow{w}_{i'} Q_f$ exists in $\mathcal{G}^{i'}_1$.*

*Proof.* We take a run of $\hat{\mathcal{G}}^i_{(q,Q')}$,

$$g^{i_1}_{(q,Q')} \xrightarrow{w}_i \{q_1, \dots, q_h\}$$

We show that for all $i^1 \geq i_1$, there is some $i^2 > i^1$ such that,

$$g^{i^2}_{(q,Q')} \xrightarrow{w}_{i^2} \{q^?_1, \dots, q^?_h\}$$

in $G^{i^2}_{(q,Q')}$ where, for $y \in \{1, \dots, h\}$,

$$q^?_y = \begin{cases} g^{i^1}_{(q',Q'')} & \text{if } q_1 = g^{i_1}_{(q',Q'')} \\ q_y & \text{otherwise} \end{cases}$$

Since $q^?_f = q_f$ for all $q_f \in \mathcal{Q}_f$, the lemma follows. For a set $Q = \{q_1, \dots, q_h\}$ we write $Q^? = \{q^?_1, \dots, q^?_h\}$.

The proof proceeds by induction over $i$. In the base case $i \leq i_1$ and the property holds by Lemma 5.4.2 and since $\hat{\mathcal{G}}^{i_1}_1 = \mathcal{G}^{i_1}_1$ and there are no incoming transitions to any $g^{i_1}_{(q',Q'')}$ in $\mathcal{G}^{i_1}_1$.

In the inductive case, we begin by showing for a single transition,

$$g^{i_1}_{(q,Q')} \xrightarrow{b}_i \{q_1, \dots, q_h\}$$

in $\hat{G}^{i}_{(q,Q')}$ with $b \in \Sigma$, we have, for all $i^1 \geq i_1$, there is some $i^2 > i^1$ such that,

$$g^{i^2}_{(q,Q')} \xrightarrow{b}_{i^2} \{q^?_1, \ldots, q^?_h\}$$

in $G^{i^2}_{(q,Q')}$. We analyse the $S \in \widetilde{G}^{i_1}_{(q,Q')}[i_1/i_1 - 1]$ that spawned the transition from $g^{i_1}_{(q,Q')}$ (we assume the transition is new, else the property holds by induction). Let $S = \{\alpha_1, \ldots, \alpha_m\}$. We have $\{q_1, \ldots, q_h\} = Q_1 \cup \cdots \cup Q_m$. For each $e \in \{1, \ldots, m\}$, there are several cases,

- $\alpha_e = \theta$.

  Let $g_e = q^\theta$. By definition of $\hat{\mathcal{G}}^i_1$ we have the transition $g_e \xrightarrow{b}_{i-1} Q_e$ in $\hat{\mathcal{G}}^{i-1}_1$.

  If $\theta = \widetilde{G}^{i_1}_{(q',Q'')}$ then by induction we have $i^2_e > i^1$ such that $g^{i^2_e}_{(q',Q'')} \xrightarrow{b}_{i^2_e} Q^?_e$ in $\mathcal{G}^{i^2_e}_1$. Otherwise $g_e$ is initial in some $B \in \mathcal{B}_1$ and the transition $g_e \xrightarrow{b}_{i-1} Q_e$ also exists in $\mathcal{G}^0_1$ and is the same as $g_e \xrightarrow{b}_0 Q^?_e$. Let $w_e = b$.

- $\alpha_e = (a, push_{w_p}, \theta)$. Then $b = a$.

  Let $g_e = q^\theta$. By definition of $\hat{\mathcal{G}}^i_1$ we have the run $g_e \xrightarrow{w_p}_{i-1} Q_e$ in $\hat{\mathcal{G}}^{i-1}_1$.

  If $\theta = \widetilde{G}^{i_1}_{(q',Q'')}$ then by induction we have $i^2_e > i^1$ such that $g^{i^2_e}_{(q',Q'')} \xrightarrow{w_p}_{i^2_e} Q^?_e$ in $\mathcal{G}^{i^2_e}$. Otherwise $g_e$ is initial in some $B \in \mathcal{B}_1$ and the transition $g_e \xrightarrow{w_p}_{i-1} Q_e$ also exists in $\mathcal{G}^0_1$ and is the same as $g_e \xrightarrow{w_p}_0 Q^?_e$. Let $w_e = w_p$.

Let $i_{max}$ be the maximum $i^2_e$. If $g_e = g^{i_1}_{(q',Q'')}$, we have, by Lemma 5.4.2, $g^{i_{max}}_{(q',Q'')} \xrightarrow{w_e}_{i_{max}} Q^?_e$. Also, by Lemma 5.4.2 we have $g_e \xrightarrow{w_e}_{i_{max}} Q^?_e$ when $g_e$ is not of the form $g^{i_1}_{(q',Q'')}$. Since we have $\widetilde{G}^{i_{max}+1}_{(q,Q')} \simeq \widetilde{G}^{i_1}_{(q,Q')}$ we have $S[i_{max}/i_1 - 1] \in \widetilde{G}^{i_{max}+1}_{(q,Q')}$ and since $Q^?_1 \cup \cdots \cup Q^?_m = \{q^?_1, \ldots, q^?_h\}$ we have,

$$g^{i_{max}+1}_{(q,Q')} \xrightarrow{b}_{i_{max}+1} \{q^?_1, \ldots, q^?_h\}$$

in $G^{i_{max}+1}_{(q,Q')}$. Let $i^2 = i_{max} + 1$ and we are done in the case of a single transition.

We now expand the result to a complete run by induction over the length of the run. That is, we take a run of $\hat{\mathcal{G}}^i_{(q,Q')}$,

$$g^{i_1}_{(q,Q')} \xrightarrow{w}_i \{q_1, \ldots, q_h\}$$

and show that for all $i^1 \geq i_1$ there is some $i^2 > i^1$ such that,

$$g^{i^2}_{(q,Q')} \xrightarrow{w}_{i^2} \{q^?_1, \ldots, q^?_h\}$$

in $G^{i^2}_{(q,Q')}$.

The base case has already been shown. We now consider the run,

$$g^{i_1}_{(q,Q')} \xrightarrow{a_0}_i \{q^1_1, \ldots, q^1_{h_1}\} \xrightarrow{a_1}_i \ldots \xrightarrow{a_m}_i \{q^m_1, \ldots, q^m_{h_m}\}$$

We have $q^1_y \xrightarrow{a_1 \ldots a_m}_i Q_y$ for each $y \in \{1, \ldots, h_1\}$ and $\bigcup_{y \in \{1,\ldots,h_1\}} Q_y = \{q^m_1, \ldots, q^m_{h_m}\}$. Then for all $y \in \{1, \ldots, h_1\}$ via induction and Lemma 5.4.10 we have for all $i^1 > i_1$ an $i_{max}$ with

$$\{q^{?1}_1, \ldots, q^{?1}_{h_1}\} \xrightarrow{a_1 \ldots a_m}_{i_{max}} \{q^{?m}_1, \ldots, q^{?m}_{h_m}\}$$

We then use the result for a single transition to obtain the result for the complete run. That is, we have for $i_{max}$ an $i^2 > i_{max}$ such that,

$$g^{i^2}_{(q,Q')} \xrightarrow{a_0}_{i^2} \{q^{?1}_1, \dots, q^{?1}_{h_1}\} \xrightarrow{a_1 \dots a_m}_{i^2} \{q^{?m}_1, \dots, q^{?m}_{h_m}\}$$

exists in $\mathcal{G}^{i^2}_1$ as required. □

We are now ready to prove the desired property.

**Corollary 5.4.1.** *Suppose we have constructed a sequence of automata $\mathcal{G}^0_1, \mathcal{G}^1_1, \dots$ and associated sets $\widetilde{\mathcal{G}}^0_1, \widetilde{\mathcal{G}}^1_1, \dots$. Further, suppose there exists an $i_1$ such that for all $i \geq i_1$ we have $\widetilde{\mathcal{G}}^i_1 \simeq \widetilde{\mathcal{G}}^{i_1}_1$. We can define a sequence of automata $\hat{\mathcal{G}}^{i_1}_1, \hat{\mathcal{G}}^{i_1+1}_1, \dots$ such that the state-set in $\hat{\mathcal{G}}^i_1$ remains constant and there exists $i_0$ such that $\hat{\mathcal{G}}^{i_0}_1$ characterises the sequence — that is, the following are equivalent for all $w$,*

1. *The run $g^{i_1}_{(q,Q')} \xrightarrow{w}_i Q_1$ with $Q_1 \subseteq \mathcal{Q}_f$ exists in $\hat{\mathcal{G}}^i_1$ for some $i$.*
2. *The run $g^{i_1}_{(q,Q')} \xrightarrow{w}_{i_0} Q_2$ with $Q_2 \subseteq \mathcal{Q}_f$ exists in $\hat{\mathcal{G}}^{i_0}_1$.*
3. *The run $g^{i'}_{(q,Q')} \xrightarrow{w}_{i'} Q_3$ with $Q_3 \subseteq \mathcal{Q}_f$ exists in $\mathcal{G}^{i'}_1$ for some $i'$.*

*Proof.* Follows from the definition of $\hat{\mathcal{G}}^{i+1}_1 = T_{\widetilde{\mathcal{G}}^{i_1}_1[i_1/i_1-1]}(\hat{\mathcal{G}}^i_1)$, Lemma 5.4.11, Lemma 5.4.12 and Lemma 5.4.13. □

**Proof of Lemma 5.4.8 for $l > 1$**

In this section we prove Lemma 5.4.8 for the case when $l > 1$. The structure of the proof is similar to the previous section.

**Lemma 5.4.14.** *For all $w$, if $g^i_{(q,Q')} \xrightarrow{w}_i Q_f$ with $Q_f \subseteq \mathcal{Q}_f$ is a run in $\mathcal{G}^i_l$ for some $i$, then we have $g^{i_l}_{(q,Q')} \xrightarrow{w}_{i'} Q_f$ in $\hat{\mathcal{G}}^{i'}_l$ for some $i'$.*

*Proof.* We prove the following property. For any path $g^i_{(q,Q')} \xrightarrow{w}_{i'} \{q_1, \dots, q_h\}$ we can construct a path $g^{i_l}_{(q,Q')} \xrightarrow{w}_{i'} \{q^!_1, \dots, q^!_h\}$ with,

$$q^!_y = \begin{cases} g^{i_l}_{(q',Q'')} & \text{if } q_y = g^{i''}_{(q',Q'')} \text{ and } i'' \geq i_1 \\ q_y & \text{otherwise} \end{cases}$$

Since $q^!_f = q_f$ for all $q_f \in \mathcal{Q}_f$, the lemma follows. When $Q = \{q_1, \dots, q_h\}$ we write $Q^!$ to denote the set $\{q^!_1, \dots, q^!_h\}$.

There are two cases. When $i \leq i_l$, using Lemma 5.4.2, that $\mathcal{G}^{i_l}_l = \hat{\mathcal{G}}^{i_l}_l$ and that $q^!_y = q_y$ for all $y$, we have $g^{i_l}_{(q,Q')} \xrightarrow{w'}_{i_l} \{q^!_1, \dots, q^!_h\}$ in $\hat{\mathcal{G}}^{i_l}_l$.

We now consider the case $i > i_l$. We begin by proving that for a single transition,

$$g^i_{(q,Q')} \xrightarrow{\gamma}_i \{q_1, \dots, q_h\}$$

in $\mathcal{G}^i_l$, we have the following transition in $\hat{G}^{i'}_{(q,Q')}$ for some $i'$,

$$g^{i_l}_{(q,Q')} \xrightarrow{\gamma}_{i'} \{q^!_1, \dots, q^!_h\}$$

We consider the $S = \{\alpha_1, \ldots, \alpha_m\} \in \widetilde{G}^i_{(q,Q')}$ from which the transition from $g^i_{(q,Q')}$ was derived (if it was inherited, then the property holds by induction over $i$). We have $\{q_1, \ldots, q_h\} = Q_1 \cup \cdots \cup Q_m$. For each $e \in \{1, \ldots, m\}$ there are several cases,

- $\alpha_e = \theta$.

  Let $g_e = q^\theta$. We have $g_e \xrightarrow{\theta'}_{i-1} Q_e$ in $\mathcal{G}^{i-1}_l$ with $\theta' \in S' \in \widetilde{G}^i_{(g^i_{(q,Q')}, \{q_1,\ldots,q_h\})}$ and the accepting run of $\gamma$ derived from $S'$. By Lemma 5.4.4 we have $\gamma \in \mathcal{L}(\theta')$. Hence, we have $g_e \xrightarrow{\gamma}_{i-1} Q_e$ in $\mathcal{G}^{i-1}_l$. By induction over $i$, we have $g^!_e \xrightarrow{\gamma}_{i'_e} Q^!_e$ in $\hat{G}^{i'_e}_{(q,Q')}$ for some $i'_e$. Furthermore, let $w_e = \gamma$.

- $\alpha_e = (a, push_l, \theta)$.

  Let $g_e = q^\theta$. We have the path

  $$g_e \xrightarrow{\theta_1}_{i-1} Q \xrightarrow{\widetilde{\theta}_2}_{i-1} Q_e$$

  in $\mathcal{G}^{i-1}_l$ with $\{B^a_{l-1}, \theta_1\} \cup \widetilde{\theta}_2 \subseteq S' \in \widetilde{G}^i_{(g^i_{(q,Q')}, \{q_1,\ldots,q_h\})}$ and the accepting run of $\gamma$ is derived from $S'$. By Lemma 5.4.4 we have $\gamma \in \mathcal{L}(\{B^a_{l-1}, \theta_1\} \cup \widetilde{\theta}_2)$. Hence, we have the run,

  $$g_e \xrightarrow{\gamma}_{i-1} Q \xrightarrow{\gamma}_{i-1} Q_e$$

  By induction over $i$, we have,

  $$g^!_e \xrightarrow{\gamma\gamma}_{i'_e} Q^!_e$$

  for some $i'_e$. Furthermore, let $w_e = \gamma\gamma$.

- $\alpha_e = (a, pop_l, \theta)$.

  Let $g_e = q^\theta$. We have $Q_e = \{g_e\}$.

  Additionally, we have $B^a_{l-1} \in S' \in \widetilde{G}^i_{(g^i_{(q,Q')}, \{q_1,\ldots,q_h\})}$ and the accepting run of $\gamma$ derived from $S'$. By Lemma 5.4.4 we have $\gamma \in \mathcal{L}(B^a_{l-1})$. Furthermore, let $w_e = \varepsilon$.

- $\theta = (a, o, \theta)$ where $\ell(d) < l$.

  Let $g_e = q^{\theta_x}$. We have $g_e \xrightarrow{\theta'}_{i-1} Q_e$ in $\mathcal{G}^{i-1}_l$ with $(a, o, \theta') \in S' \in \widetilde{G}^i_{(g^i_{(q,Q')}, \{q_1,\ldots,q_h\})}$ and the accepting run of $\gamma$ derived from $S'$. By Lemma 5.4.5 we have $o(\gamma) \in \mathcal{L}(\theta')$. Hence, we have $g_e \xrightarrow{o(\gamma)}_{i-1} Q_e$ in $\mathcal{G}^{i-1}_l$. By induction over $i$, we have $g^!_e \xrightarrow{o(\gamma)}_{i'_e} Q^!_e$ in $\hat{G}^{i'_e}_{(q,Q')}$ for some $i'_e$. Furthermore, let $w_e = o(\gamma)$.

Let $i_{max}$ be the maximum $i'_e$. By Lemma 5.4.2 we have $g^!_e \xrightarrow{w_e}_{i_{max}} Q^!_e$ for all $e$. Since $S[i_{max}/i-1] \in \widetilde{G}^{i_{max}+1}_{(q,Q')}$ and $Q^!_1 \cup \cdots \cup Q^!_m = \{q^!_1, \ldots, q^!_h\}$ we have via Lemma 5.4.6,

$$g^{i_1}_{(q,Q')} \xrightarrow{\gamma}_{i_{max}+1} \{q^!_1, \ldots, q^!_h\}$$

in $\hat{G}^{i_{max}+1}_{(q,Q')}$ as required.

We now prove the result for a run of more than one step by induction over the length of the run. In the base case we have a run of a single transition. The result in this case has already been shown.

In the inductive case we have a run of the form,

$$g^{i}_{(q,Q')} \xrightarrow{\gamma_0}_{i} \{q^1_1, \ldots, q^1_{h_1}\} \xrightarrow{\gamma_1}_{i} \ldots \xrightarrow{\gamma_m}_{i} \{q^m_1, \ldots, q^m_{h_m}\}$$

For each $y \in \{1, \ldots, h_1\}$ we have $q^1_y \xrightarrow{\gamma_1 \ldots \gamma_m}_i Q_y$ such that $\bigcup_{y \in \{1,\ldots,h_1\}} Q_y = \{q^m_1, \ldots, q^m_{h_m}\}$. By induction on the length of the run we have $q^{!1}_y \xrightarrow{\gamma_1 \ldots \gamma_m}_{i'} Q^{!}_y$ for each $y$ for some $i'$. Let $i_{max}$ be the maximum $i'$. We also have $g^{i_1}_{(q,Q')} \xrightarrow{\gamma_0}_{i'} \{q^{!1}_1, \ldots, q^{!1}_{h_1}\}$ for some $i'$ from the above proof for one transition. We have, via Lemma 5.4.10, a run of the form,

$$g^{i_1}_{(q,Q')} \xrightarrow{\gamma_0}_{i_{max}} \{q^{!1}_1, \ldots, q^{!1}_{h_1}\} \xrightarrow{\gamma_1}_{i_{max}} \ldots \xrightarrow{\gamma_m}_{i_{max}} \{q^{!m}_1, \ldots, q^{!m}_{h_m}\}$$

in $\hat{\mathcal{G}}^{i_{max}}_l$ as required. ☐

**Lemma 5.4.15.** *For all $w$, if we have $g^{i_l}_{(q,Q')} \xrightarrow{w}_i Q_f$ with $Q_f \subseteq \mathcal{Q}_f$ in $\hat{\mathcal{G}}^i_l$ for some $i$, then there is some $i'$ such that the run $g^{i'}_{(q,Q')} \xrightarrow{w}_{i'} Q_f$ exists in $\mathcal{G}^{i'}_l$.*

*Proof.* We take a run of $\hat{\mathcal{G}}^i_{(q,Q')}$,

$$g^{i_l}_{(q,Q')} \xrightarrow{w}_i \{q_1, \ldots, q_h\}$$

We show that for all $i^1 \geq i_l$, there is some $i^2 > i^1$ such that,

$$g^{i^2}_{(q,Q')} \xrightarrow{w}_{i^2} \{q^?_1, \ldots, q^?_h\}$$

in $G^{i^2}_{(q,Q')}$ where, for $y \in \{1, \ldots, h\}$,

$$q^?_y = \begin{cases} g^{i^1}_{(q',Q'')} & \text{if } q_1 = g^{i_l}_{(q',Q'')} \\ q_y & \text{otherwise} \end{cases}$$

Since $q^?_f = q_f$ for all $q_f \in \mathcal{Q}_f$, the lemma follows. Given a set $Q = \{q_1, \ldots, q_m\}$, we write $Q^?$ to denote the set $\{q^?_1, \ldots, q^?_m\}$.

The proof proceeds by induction over $i$. In the base case $i \leq i_l$ and the property holds by Lemma 5.4.2 and since $\hat{\mathcal{G}}^{i_l}_1 = \mathcal{G}^{i_l}_1$ and there are no incoming transitions to any $g^{i_l}_{(q',Q'')}$ in $\mathcal{G}^{i_l}_1$.

In the inductive case, we begin by showing for a single transition,

$$g^{i_l}_{(q,Q')} \xrightarrow{\gamma}_i \{q_1, \ldots, q_h\}$$

in $\hat{G}^i_{(q,Q')}$ we have, for all $i^1 \geq i_l$, there is some $i^2 > i^1$ such that,

$$g^{i^2}_{(q,Q')} \xrightarrow{\gamma}_{i^2} \{q^?_1, \ldots, q^?_h\}$$

in $G^{i^2}_{(q,Q')}$. We analyse the $S \in \widetilde{G}^{i_l}_{(q,Q')}[i_l/i_l - 1]$ from which the transition from $g^{i_l}_{(q,Q')}$ was derived (we assume the transition is not inherited, else the property holds by induction).

Let $S = \{\alpha_1, \ldots, \alpha_m\}$. We have $\{q_1, \ldots, q_h\} = Q_1 \cup \cdots \cup Q_m$. For each $e \in \{1, \ldots, m\}$ there are several cases,

- $\alpha_e = \theta$.

  Let $g_e = q^\theta$. We have $g_e \xrightarrow{\theta'}_{i-1} Q_e$ in $\hat{\mathcal{G}}_l^{i-1}$ with $\theta' \in S' \in \widetilde{G}^i_{(g^{i_l}_{(q,Q')},\{q_1,\ldots,q_h\})}$ and the accepting run of $\gamma$ derived from $S'$. By Lemma 5.4.4 we have $\gamma \in \mathcal{L}(\theta')$. Hence, we have $g_e \xrightarrow{\gamma}_{i-1} Q_e$ in $\hat{\mathcal{G}}_l^{i-1}$.

  If $\theta = \widetilde{G}^{i_l}_{(q',Q'')}$, by induction over $i$, we have $i_e^2 > i^1$ such that $g^{i_e^2}_{(q',Q'')} \xrightarrow{\gamma}_{i_e^2} Q_e^?$ in $G^{i_e^2}_{(q,Q')}$. Otherwise $g_e$ is initial in some $B \in \mathcal{B}_l$ and the transition $g_e \xrightarrow{\gamma}_{i-1} Q_e$ also exists in $\mathcal{G}_l^0$ and is the same as $g_e \xrightarrow{\gamma}_0 Q_e^?$. Let $w_e = \gamma$.

- $\alpha_e = (a, push_l, \theta)$.

  Let $g_e = q^\theta$. We have the path

$$g_e \xrightarrow{\theta_1}_{i-1} Q \xrightarrow{\widetilde{\theta}_2}_{i-1} Q_e$$

  in $\hat{\mathcal{G}}_l^{i-1}$ with $\{B^a_{l-1}, \theta_1\} \cup \widetilde{\theta}_2 \subseteq S' \in \widetilde{G}^i_{(g^{i_l}_{(q,Q')},\{q_1,\ldots,q_h\})}$ and the accepting run of $\gamma$ is derived from $S'$. By Lemma 5.4.4 we have $\gamma \in \mathcal{L}(\{B^a_{l-1}, \theta_1\} \cup \widetilde{\theta}_2)$. Hence, we have the run,

$$g_e \xrightarrow{\gamma}_{i-1} Q \xrightarrow{\gamma}_{i-1} Q_e$$

  If $\theta = \widetilde{G}^{i_l}_{(q',Q'')}$, by induction over $i$, we have $i_e^2 > i^1$ such that,

$$g^{i_e^2}_{(q',Q'')} \xrightarrow{\gamma\gamma}_{i_e^2} Q_e^?$$

  Otherwise $g_e$ is initial in some $B \in \mathcal{B}_l$ and the transition $g_e \xrightarrow{\gamma\gamma}_{i-1} Q_e$ also exists in $\mathcal{G}_l^0$ and is the same as $g_e \xrightarrow{\gamma\gamma}_0 Q_e^?$. Let $w_e = \gamma\gamma$.

- $\alpha_e = (a, pop_l, \theta)$.

  Let $g_e = q^\theta$. We have $Q_e = \{g_e\}$. Additionally, we have $B^a_{l-1} \in S' \in \widetilde{G}^i_{(g^{i_l}_{(q,Q')},\{q_1,\ldots,q_h\})}$ and the accepting run of $\gamma$ derived from $S'$. By Lemma 5.4.4 we have $\gamma \in \mathcal{L}(B^a_{l-1})$. Furthermore, let $w_e = \varepsilon$.

- $\theta = (a, o, \theta)$ where $\ell(d) < l$.

  Let $g_e = q^\theta$. We have $g_e \xrightarrow{\theta'}_{i-1} Q_e$ in $\hat{\mathcal{G}}_l^{i-1}$ with $(a, o, \theta') \in S' \in \widetilde{G}^i_{(g^{i_l}_{(q,Q')},\{q_1,\ldots,q_h\})}$ and the accepting run of $\gamma$ derived from $S'$. By Lemma 5.4.5 we have $o(\gamma) \in \mathcal{L}(\theta')$. Hence, we have $g_e \xrightarrow{o(\gamma)}_{i-1} Q_e$ in $\hat{\mathcal{G}}_l^{i-1}$.

  When $\theta = \widetilde{G}^{i_l}_{(q',Q'')}$, we have by induction over $i$ some $i_e^2 > i^1$ such that $g^{i_e^2}_{(q',Q'')} \xrightarrow{o(\gamma)}_{i_e^2} Q_e^?$ in $\hat{G}^{i_e^2}_{(q,Q')}$. Otherwise $g_e$ is initial in some $B \in \mathcal{B}_l$ and the transition $g_e \xrightarrow{o(\gamma)}_{i-1} Q_e$ also exists in $\mathcal{G}_l^0$ and is the same as $g_e \xrightarrow{o(\gamma)}_0 Q_e^?$. Let $w_e = o(\gamma)$.

Let $i_{max}$ be the maximum $i_e^2$. If $g_e = g^{i_l}_{(q',Q'')}$, we have, by Lemma 5.4.2, $g^{i_{max}}_{(q',Q'')} \xrightarrow{w_e}_{i_{max}} Q_e^?$. Also, by Lemma 5.4.2 we have $g_e \xrightarrow{w_e}_{i_{max}} Q_e^?$ when $g_e$ is not of the form $g^{i_l}_{(q',Q'')}$. Since we

have $\widetilde{G}^{i_{max}+1}_{(q,Q')} \simeq \widetilde{G}^{i_l}_{(q,Q')}$ we have $S[i_{max}/i_l] \in \widetilde{G}^{i_{max}+1}_{(q,Q')}$ and since $Q^?_1 \cup \cdots \cup Q^?_m = \{q^?_1, \ldots, q^?_h\}$ we have via Lemma 5.4.6,

$$g^{i_{max}+1}_{(q,Q')} \xrightarrow{\gamma}_{i_{max}+1} \{q^?_1, \ldots, q^?_h\}$$

in $G^{i_{max}+1}_{(q,Q')}$. Let $i^2 = i_{max} + 1$ and we are done in the case of a single transition.

We now expand the result to a complete run by induction over the length of the run. That is, we take a run of $\hat{\mathcal{G}}^i_{(q,Q')}$,

$$g^{i_l}_{(q,Q')} \xrightarrow{w}_i \{q_1, \ldots, q_h\}$$

and show that for all $i^1 \geq i_l$ there is some $i^2 > i^1$ such that,

$$g^{i^2}_{(q,Q')} \xrightarrow{w}_{i^2} \{q^?_1, \ldots, q^?_h\}$$

in $G^{i^2}_{(q,Q')}$.

The base case has already been shown. We now consider the run,

$$g^{i_l}_{(q,Q')} \xrightarrow{\gamma_0}_i \{q^1_1, \ldots, q^1_{h_1}\} \xrightarrow{\gamma_1}_i \ldots \xrightarrow{\gamma_m}_i \{q^m_1, \ldots, q^m_{h_m}\}$$

We have $q^1_y \xrightarrow{\gamma_1 \ldots \gamma_m}_i Q_y$ for each $y \in \{1, \ldots, h_1\}$ and $\bigcup_{y \in \{1,\ldots,h_1\}} Q_y = \{q^m_1, \ldots, q^m_{h_m}\}$. Then for all $y \in \{1, \ldots, h_1\}$ we have via induction and Lemma 5.4.10 we have for all $i^1 > i_l$ an $i_{max}$ with

$$\{q^{?1}_1, \ldots, q^{?1}_{h_1}\} \xrightarrow{\gamma_1 \cdots \gamma_m}_{i_{max}} \{q^{?m}_1, \ldots, q^{?m}_{h_m}\}$$

We then use the result for a single transition to obtain the result for the complete run. That is, we have for $i_{max}$ an $i^2 > i_{max}$ such that,

$$g^{i^2}_{(q,Q')} \xrightarrow{\gamma_0}_{i^2} \{q^{?1}_1, \ldots, q^{?1}_{h_1}\} \xrightarrow{a_1 \ldots a_m}_{i^2} \{q^{?m}_1, \ldots, q^{?m}_{h_m}\}$$

as required. □

**Corollary 5.4.2.** *Suppose we have constructed a sequence of automata $\mathcal{G}^0_l, \mathcal{G}^1_l, \ldots$ and associated sets $\widetilde{\mathcal{G}}^0_l, \widetilde{\mathcal{G}}^1_l, \ldots$. Further, suppose there exists an $i_l$ such that for all $i \geq i_l$ we have $\widetilde{\mathcal{G}}^i_l \simeq \widetilde{\mathcal{G}}^{i_l}_l$. We can define a sequence of automata $\hat{\mathcal{G}}^{i_l}_l, \hat{\mathcal{G}}^{i_1+1}, \ldots$ such that the state-set in $\hat{\mathcal{G}}^i_l$ remains constant (although the automata labelling the transitions may gain states). The following are equivalent for all $w$,*

1. *The run $g^{i_l}_{(q,Q')} \xrightarrow{w}_i Q_1$ with $Q_1 \subseteq \mathcal{Q}_f$ exists in $\hat{\mathcal{G}}^i_l$ for some $i$.*
2. *The run $g^{i'}_{(q,Q')} \xrightarrow{w}_{i'} Q_2$ with $Q_2 \subseteq \mathcal{Q}_f$ exists in $\mathcal{G}^i_l$ for some $i'$.*

*Proof.* From Lemma 5.4.14 and Lemma 5.4.15. □

### Proof of Lemma 5.4.9

Finally, we show that the constructed automaton $\mathcal{G}^*_l$ is correct.

**Lemma 5.4.9.** *Suppose we have constructed a sequence of automata $\mathcal{G}^0_l, \mathcal{G}^1_l, \ldots$ and associated sets $\widetilde{\mathcal{G}}^0_l, \widetilde{\mathcal{G}}^1_l, \ldots$. Further, suppose there exists an $i_l$ such that for all $i \geq i_l$ we have $\widetilde{\mathcal{G}}^i_l \simeq \widetilde{\mathcal{G}}^{i_l}_l$. We can define an automaton $\mathcal{G}^*_l$ such that the following are equivalent for all $w$,*

1. *The run* $g^*_{(q,Q')} \xrightarrow{w}_* Q$ *with* $Q \subseteq \mathcal{Q}_f$ *exists in* $\mathcal{G}^*_l$.

2. *The run* $g^i_{(q,Q')} \xrightarrow{w}_i Q''$ *with* $Q'' \subseteq \mathcal{Q}_f$ *exists in* $\mathcal{G}^i_l$ *for some* $i$.

*Proof.* We proceed by induction over $l$. In the base case $l = 1$ and the result follows from Corollary 5.4.1. That is, $\mathcal{G}^*_1 = \mathcal{G}^{i_0}_1$ letting $g^*_{(q,Q)} = g^{i_1}_{(q,Q)}$ for each $g^{i_1}_{(q,Q)}$. The equivalence of (1) and (2) is immediate.

When $l > 1$ we generate the sequence $\hat{\mathcal{G}}^{i_l}_l, \hat{\mathcal{G}}^{i_l+1}_l, \ldots$ by Corollary 5.4.2. Since the state-set remains constant, it follows from Property 5.4.5 that there is some $i_{l-1}$ with $\widetilde{G}^i_{l-1} \simeq \widetilde{G}^{i_{l-1}}_{l-1}$ for all $i \geq i_{l-1}$.

By induction, we have $\mathcal{G}^*_{l-1}$. We then define $\mathcal{G}^*_l$ from $\hat{\mathcal{G}}^{i_{l-1}}_l$ with $g^*_{(q,Q')} = g^{i_l}_{(q,Q')}$ for all $q, Q'$ and each transition $q \longrightarrow_* Q'$ in $\mathcal{G}^*_l$ labelled with the automaton $G^*_{(q,Q')}$ from $\mathcal{G}^*_{l-1}$.

We show (1) and (2) are equivalent. By Lemma 5.4.8 (2) is equivalent to a run $g^{i_l}_{(q,Q')} \xrightarrow{w}_{i'} Q_1$ with $Q_1 \subseteq \mathcal{Q}_f$ in $\hat{\mathcal{G}}^{i'}_l$ for some $i'$.

We proceed by induction over the length of $w$. Note that $g^{i_l}_{(q,Q')} = g^*_{(q,Q')}$. We prove that if $w \neq \varepsilon$ a run $g^i_{(q,Q')} \xrightarrow{w}_* Q$ exists in $\mathcal{G}^*_l$ iff a run $g^i_{(q,Q')} \xrightarrow{w}_{i'} Q$ exists in $\hat{\mathcal{G}}^{i'}_l$ for some $i'$. It is necessarily the case that $i \leq i_l$. The Lemma follows because a state of the form $g^i_{(q',Q'')}$ is never accepting and therefore $w \neq \varepsilon$.

In the base case, let $w = \gamma$. Let $g = g^i_{(q,Q')}$. To prove (1) implies (2) we assume the transition $g \xrightarrow{G^*_{(g,Q)}}_* Q$ in $\mathcal{G}^*_l$ with $\gamma \in G^*_{(g,Q)}$. By induction over $l$ we have $\gamma \in \hat{G}^{i'}_{(g,Q)}$ for some $i'$. Hence, we have $g \xrightarrow{\gamma}_{i'} Q$ in $\hat{\mathcal{G}}^{i'}_l$ as required.

To show (2) implies (1) we assume the transition $g \xrightarrow{\hat{G}^{i'}_{(g,Q_1)}}_{i'} Q_1$ in $\hat{\mathcal{G}}^{i'}_l$ for some $i'$ with $\gamma \in \hat{G}^{i'}_{(g,Q_1)}$. By induction over $l$ we have $\gamma \in G^*_{(g,Q_1)}$. Hence, we have $g \xrightarrow{\gamma}_* Q_1$ in $\mathcal{G}^*_l$ as required. This completes the proof of the base case.

For the induction, let $w = \gamma w'$ and $w' \neq \varepsilon$. To show (2) follows from (1) assume we have a transition $g \xrightarrow{G^*_{(g,Q)}}_* Q = \{q_1, \ldots, q_m\}$ with $\gamma \in G^*_{(g,Q)}$ in $\mathcal{G}^*_l$ and the run,

$$\{q_1, \ldots, q_m\} \xrightarrow{w'}_* Q_f$$

in $\mathcal{G}^*_l$. As before, we have $g \xrightarrow{\gamma}_{i'} Q$ in $\hat{\mathcal{G}}^{i'}_l$ for some $i'$. For all $e \in \{1, \ldots, m\}$, we have a run $q_e \xrightarrow{w'}_* Q^e_f$ with $Q_f = Q^1_f \cup \cdots \cup Q^m_f$. If $q_e$ is of the form $g^j_{(q',Q'')}$ for some $j$, then by induction we have $g^j_{(q',Q'')} \xrightarrow{w'}_{i'} Q^e_f$ in $\hat{\mathcal{G}}^{i'}_l$ for some $i'$ by induction over the length of the word. Otherwise the run over $w'$ uses only states and transitions left unchanged by the algorithm. Hence we have $q_e \xrightarrow{w'}_{i_l} Q^e_f$ in $\hat{\mathcal{G}}^{i_l}_l$. As required, by Lemma 5.4.10 we have in $\hat{\mathcal{G}}^{i_{max}}_l$ for some $i_{max}$,

$$g \xrightarrow{\gamma}_{i_{max}} \{q_1, \ldots, q_m\} \xrightarrow{w'}_{i_{max}} Q^1_f \cup \cdots \cup Q^m_f$$

To show (1) follows from (2) assume we have a transition $g \xrightarrow{\hat{G}^{i'}_{(g,Q)}}_{i'} Q = \{q_1, \ldots, q_m\}$ with $\gamma \in \hat{G}^{i'}_{(g,Q)}$ in $\hat{\mathcal{G}}^{i'}_l$ and the run,

$$\{q_1, \ldots, q_m\} \xrightarrow{w'}_{i'} Q_f$$

in $\mathcal{G}_l^{i'}$ for some $i'$. As in the base case, we have $g \xrightarrow{\gamma}_* Q$ in $\mathcal{G}_l^*$. For all $e \in \{1, \ldots, m\}$, we have a run $q_e \xrightarrow{w'}_{i'} Q_f^e$ with $Q_f = Q_f^1 \cup \cdots \cup Q_f^m$. If $q_e$ is of the form $g^j_{(q',Q'')}$ for some $j$, then by induction we have $g^j_{(q',Q'')} \xrightarrow{w'}_* Q_f^e$ in $\mathcal{G}_l^*$ by induction over the length of the word. Otherwise the run over $w'$ uses only states and transitions left unchanged by the algorithm. Hence, since $\mathcal{G}_l^*$ is derived from $\hat{\mathcal{G}}_l^{i_{l-1}}$, we have $q_e \xrightarrow{w'}_* Q_f^e$ in $\mathcal{G}_l^0$. Subsequently, we have,

$$g \xrightarrow{\gamma}_* \{q_1, \ldots, q_m\} \xrightarrow{w'}_* Q_f^1 \cup \cdots \cup Q_f^m$$

in $\mathcal{G}_l^*$ as required. □

### 5.4.7 Complexity

We claim our algorithm runs in $n$-EXPTIME. We define,

$$exp_1(m) = 2^{\mathcal{O}(m)} \qquad \text{and} \qquad exp_l(m) = 2^{\mathcal{O}(exp_{l-1}(m))}$$

Furthermore, let $\mathcal{B} = \bigcup_{1 \leq l < n} \mathcal{B}_l$.

The algorithm requires the construction of an automaton $\mathcal{G}_l^*$ from $\mathcal{G}_l^{i_l}$ for $1 \leq l \leq n-1$ and fixed point $i_l$. Because the construction of $\mathcal{G}_l^*$ follows the same pattern as the construction of $\mathcal{G}_l^{i+1}$ from $\mathcal{G}_l^i$ but with at least as many states, it follows that it dominates the complexity of the algorithm. We calculate the complexity inductively, beginning at order-one with a fixed state-set. Ultimately, the induction reaches order-$n$, where the state-set is fixed from the start of the algorithm.

Let $|\mathcal{Q}|$ be the number of states in $\mathcal{G}_l^{i_l}$. When $l = 1$ we can add at most $\mathcal{O}(|\mathcal{Q}| \times exp_1(|\mathcal{Q}|) \times |\Sigma|)$ transitions to $\mathcal{G}_1^{i_1}$ to define $\mathcal{G}_1^*$, where $\mathcal{Q}$ is the state-set of $\mathcal{G}_1^{i_1}$. This means we need at most $\mathcal{O}(exp_1(|\mathcal{Q}|))$ iterations before a fixed point is reached. Each iteration requires the processing of up to $\mathcal{O}(|\mathcal{Q}|)$ sets $\widetilde{G}_{(q_1,Q_2)}$ (one for each state $g^{i_1}_{(q_1,Q_2)}$). To process each set we need to process up to $\mathcal{O}(exp_1(|\mathcal{Q}| + |\mathcal{B}|))$ sets $S$. This step in turn requires the analysis of $\mathcal{O}(|\mathcal{Q}| + |\mathcal{B}|)$ elements. The most expensive operation is to enumerate all runs over a word for a *push* command. To enumerate all runs of the form $Q_1 \xrightarrow{w} Q_2$ for some $w$ (from a $push_w$ command) we require time $\mathcal{O}(exp_1(|\mathcal{Q}|))$ (Proposition 5.6.1). Therefore, to construct $\mathcal{G}_1^*$ from $\mathcal{G}^{i_1}$ takes $\mathcal{O}(|\mathcal{Q}| \times (|\mathcal{B}| + |\mathcal{Q}|) \times exp_1(|\mathcal{B}| + 3 \times |\mathcal{Q}|))$. That is, $\mathcal{O}(exp_1(|\mathcal{Q}|) \times exp_1(|\mathcal{B}|))$.

When $l > 1$ and the state-set $\mathcal{Q}$ of $\mathcal{G}_l^{il}$ is frozen, we add at most $\mathcal{O}(|\mathcal{Q}| \times exp_1(|\mathcal{Q}|))$ transitions to $\mathcal{G}_l^{i_l}$ during the construction of $\mathcal{G}_l^*$. This means we need at most $\mathcal{O}(exp_1(|\mathcal{Q}|))$ iterations before a fixed point is reached. Each iteration requires the processing of up to $\mathcal{O}(|\mathcal{Q}|)$ sets $\widetilde{G}_{(q_1,Q_2)}$ (one for each state $g^{i_1}_{(q_1,Q_2)}$). To process each set we need to process up to $\mathcal{O}(exp_1(|\mathcal{Q}| + |\mathcal{B}|))$ sets $S$. This step in turn requires the analysis of $\mathcal{O}(|\mathcal{Q}| + |\mathcal{B}|)$ elements. The most expensive operation is to enumerate all runs of two transitions for a *push* command. To enumerate all runs of the form $Q_1 \xrightarrow{\widetilde{B_1}} Q_2 \xrightarrow{\widetilde{B_2}} Q_3$ requires time $\mathcal{O}(exp_1(|\Delta| + |\mathcal{Q}|))$ (by Proposition 5.6.2) Since $|\Delta|$ is $\mathcal{O}(exp_1(|\mathcal{Q}|))$, the run enumeration requires time $\mathcal{O}(exp_2(|\mathcal{Q}|))$.

Since the contents of the sets $\widetilde{G}^i_{(q,Q')} \in \widetilde{\mathcal{G}}^i_{l-1}$ are derived from the transition structure, we will reach a fixed point one step after we have reached a point where no more transitions are added. During this step we will add $\mathcal{O}(exp_1(|\mathcal{Q}|))$ states to $\mathcal{G}_{l-1}^{i_l}$ to construct $\mathcal{G}_{l-1}^{i_{l-1}}$ (since there are $\mathcal{O}(exp_1(|\mathcal{Q}|))$ new transitions added in $\mathcal{O}(exp_1(|\mathcal{Q}|))$ steps). By induction, the construction of $\mathcal{G}_{l-1}^{i+1}$ from $\mathcal{G}_{l-1}^i$ performed at the end of each iteration takes $\mathcal{O}(exp_{(l-1)}(exp_1(|\mathcal{Q}|)) \times exp_1(|\mathcal{B}|))$ time.

Hence, to reach a fixed point we require time $\mathcal{O}(exp_1(|\mathcal{Q}|) \times (((|\mathcal{Q}| + |\mathcal{B}|) \times exp_1(|\mathcal{Q}| + |\mathcal{B}|) \times exp_2(|\mathcal{Q}|) + (exp_l(|\mathcal{Q}|) \times exp_1(|\mathcal{B}|))))$. That is $\mathcal{O}(exp_l(|\mathcal{Q}|) \times exp_1(|\mathcal{B}|))$. By induction we can construct $\mathcal{G}^*_{l-1}$ in time $\mathcal{O}(exp_{(l-1)}(exp_1(|\mathcal{Q}|)) \times exp_1(|\mathcal{B}|))$. Hence, we can construct $\mathcal{G}^*_l$ in time $\mathcal{O}(exp_l(|\mathcal{Q}|) \times exp_1(|\mathcal{B}|))$.

Finally, at order-$n$ we can add at most $\mathcal{O}(exp_1(|\mathcal{Q}|))$ new transitions before a fixed point is reached. Hence we must perform $\mathcal{O}(exp_1(|\mathcal{Q}|))$ iterations. Each iteration requires us to process $\mathcal{O}(exp_1(|\mathcal{Q}|))$ commands (since $|\mathcal{Q}|$ is larger than the number of control states in the higher-order APDS). In processing each command we must process $\mathcal{O}(|\mathcal{Q}|)$ pairs of the form $(o, p)$. The most expensive of which requires run enumeration that takes $\mathcal{O}(exp_2(|\mathcal{Q}|))$ time. Hence we construct $\mathcal{G}^{i_{n-1}}_{n-1}$ with $\mathcal{O}(exp_1(|\mathcal{Q}|))$ states in $\mathcal{O}(exp_1(|\mathcal{Q}|) \times (exp_1(|\mathcal{Q}|) \times |\mathcal{Q}| \times exp_2(|\mathcal{Q}|) + (exp_{(n-1)}(exp_1(|\mathcal{Q}|)) \times exp_1(|\mathcal{B}|))))$ time. Thus, $\mathcal{G}^*_{n-1}$ and hence $A_*$ can be constructed in time $\mathcal{O}(exp_{(n-1)}(exp_1(|\mathcal{Q}|)) \times exp_1(|\mathcal{B}|))$, that is $\mathcal{O}(exp_n(|\mathcal{Q}|) \times exp_1(|\mathcal{B}|))$. The algorithm runs in $n$-EXPTIME.

## 5.5 An Alternative Order-2 Construction

An alternative backwards reachability algorithm for order-2 pushdown systems was presented by Seth [23]. This construction was developed independently of our work. Using this method, sets of order-2 stacks are represented with multi-automata as in the order-1 case. That is, the automata are not nested. The stack alphabet is augmented with the symbols $[_1, [_2, ]_1$ and $]_2$. The stack $[[abc][abc]]$ corresponds to a run over the word $[_2[_1abc]_1[_1abc]_1]_2$. It was shown by Bouajjani and Meyer that these automata are equally expressive as nested automata, and hence, the notions of regularity coincide.

### The Algorithm

The algorithm is a saturation algorithm based on the order-1 construction discussed in Section 3.1. The order-1 command $push_w$ is handled in a directly analogous manner. To process a $(p^j, a, pop_2, p^k)$ command a transition is made to a state that checks that the current $top_1$ character is an $a$ and then skips over the remainder of the $top_2$ stack until a $]_1$ character is reached. We then move to the state $q^k$ and check that the rest of the order-2 stack can be accepted. This is analogous to the nested construction where we add a transition $q^j \xrightarrow{B^a_1} q^k$.

The idea underlying the nested construction for a command $(p^j, a, push_2, p^k)$ is the addition of a transition $q^j \xrightarrow{B_1 \times B_2} q$ for every run $q^k \xrightarrow{B_1} q' \xrightarrow{B_2} q$. Seth's construction uses the power of alternation. From the state $q^j$ we add a branch that splits into two runs. The first run checks that the current $top_2$ stack is accepted by $B_1$. The second checks that the stack is accepted by $B_2$.

Because the automata are not nested, it is not clear where the portion of the run corresponding to $B_2$ begins. We use nondeterminism to overcome this problem: the state analogous to the state $q'$ above is guessed and the second run proceeds as if it had started from $q'$. After checking $B_2$, this thread continues to check the acceptance of the remainder of the stack. The first thread performs two duties: it ensures that the current $top_2$ stack is accepted by $B_1$, and it makes sure that the guessed $q'$ is correct. That is, the run proceeds as if it had begun at $q^k$, but, when the symbol $]_1$ is encountered — marking the end of the $top_2$ stack — we insist that the next state must be $q'$. If this is the case, the runs terminates with acceptance.

Although extra states are needed to provide the memory required for the above strategy to work, these states can be computed from the given PDS and are only polynomial in number. The saturation step operates on this fixed state-set and hence, termination is obvious. We refer the reader to the paper for the formal details of the algorithm [23].

**Discussion**

The complexity of Seth's approach matches the complexity of ours. In particular, when the order-2 pushdown system is non-alternating, the construction runs in EXPTIME. This complexity is matched by our algorithm under the assumption that the initial 2-store automaton does not use alternation. This can be seen because all commands are of the form $(p^j, a, o, p^k)$ and hence all new transitions are of the form $q \xrightarrow{\widetilde{G}} q'$. This means that a polynomial, rather than exponential, number of transitions are added at order-2, and the second exponential blow-up is avoided.

Our algorithm does not require the addition of new, worker states to the automaton. Hence, a pre-computation step is avoided. However, since our algorithm adds new states during the saturation step, it is not clear which will perform better in practice. It can also be noted that the worker states needed by Seth's construction can be computed *on-the-fly*.

An advantage of Seth's approach is the computation of winning strategies. We have avoided asking this of our algorithm for the sake of syntactical complexity. Seth shows that, from an accepting run of a configuration, a winning strategy from that configuration can be computed. Furthermore, there is a linear time algorithm for determining the move a player should make from a winning configuration to reach the target destination in the shortest time. Although the above construction is limited to order-2 PDSs, it may be generalised to the order-$n$ case.

## 5.6 Algorithms over $n$-Store (Multi-)Automata

In this section we describe several algorithms over $n$-store automata and $n$-store multi-automata. These algorithms explain how to perform operations required in the preceding sections, as well as construct boolean combinations of automata. Observe that an $n$-store automaton is a special case of an $n$-store multi-automaton.

### 5.6.1 Enumerating Runs

**Proposition 5.6.1.** *Given a* 1*-store (multi-)automaton* $A = (\mathcal{Q}, \Sigma, \Delta, _, \mathcal{Q}_f)$*, a set of states* $Q$ *and word* $w$*, the set of all* $Q'$ *reachable via a run* $Q \xrightarrow{w} Q'$ *can be calculated in time* $\mathcal{O}(exp_1(|\mathcal{Q}|))$*.*

*Proof.* We define the following procedure, which given a set of sets of states $\widetilde{Q}$ computes the set of sets $Q'$ with $Q \in \widetilde{Q}$ and $Q \xrightarrow{a} Q'$.

EXPAND$(a, \widetilde{Q})$

**let** $\widetilde{Q}_{next} = \emptyset$
**for each** $\{q_1, \ldots, q_m\} \in \widetilde{Q}$
  **let** $ok = (\exists (q_1, a, _) \in \Delta)$ and $\widetilde{Q}_\Delta = \Delta(q_1, a)$
  **for** $i = 2$ **to** $m$

$ok = ok \wedge (\exists (q_i, a, _) \in \Delta)$
$\widetilde{Q}_\Delta = \{ Q' \cup Q'' \mid Q' \in \widetilde{Q}_\Delta \wedge (q_i, a, Q'') \in \Delta \}$
**if** $ok$ **then** $\widetilde{Q}_{next} = \widetilde{Q}_{next} \cup \widetilde{Q}_\Delta$
**return** $\widetilde{Q}_{next}$

The outer loop repeats $\mathcal{O}(exp_1(|\mathcal{Q}|))$ times and the inner loop $\mathcal{O}(|\mathcal{Q}|)$. Since the number of $Q' \in \widetilde{Q}_\Delta$ is $\mathcal{O}(exp_1(|\mathcal{Q}|))$ and the number of $(q_i, a, Q'') \in \Delta$ is also $\mathcal{O}(exp_1(|\mathcal{Q}|))$, construction of $\widetilde{Q}_\Delta$ takes time $\mathcal{O}(exp_1(|\mathcal{Q}|))$. Hence the procedure takes time $\mathcal{O}(exp_1(|\mathcal{Q}|) \times |\mathcal{Q}| \times exp_1(|\mathcal{Q}|))$, that is $\mathcal{O}(exp_1(|\mathcal{Q}|))$.

Expand is correct since $Q \in \widetilde{Q}_{next}$ at the end of the procedure iff we have $\{q_1, \ldots, q_m\} \in \widetilde{Q}$ and some $(q_i, a, Q^i) \in \Delta$ for each $i \in \{1, \ldots, m\}$ with $Q = Q^1 \cup \cdots \cup Q^m$.

Over a word $w = a_1 \ldots a_m$ we define the following procedure,

ExpandWord$(a_1 \ldots a_m, Q)$

**let** $\widetilde{Q} = \{Q\}$
**for** $i = 1$ **to** $m$
$\widetilde{Q} =$ Expand$(a_i, \widetilde{Q})$
**return** $\widetilde{Q}$

This procedure requires $m$ runs of Expand and consequently runs in time $\mathcal{O}(exp_1(|\mathcal{Q}|))$.

We prove the correctness of ExpandWord by induction over the length of the word. When $w = a_1$ correctness follows from the correctness of Expand. In the inductive case $w = a_1 \ldots a_m$. We have all runs of the form $Q \xrightarrow{a_1} Q_1$ as before, and all runs over $a_2 \ldots a_m$ from all $Q_1$ by induction. We have all runs of the form $Q \xrightarrow{w} Q'$ therefrom. □

**Proposition 5.6.2.** *Given an $l$-store (multi-)automaton $A = (\mathcal{Q}, \Sigma, \Delta, _, \mathcal{Q}_f)$ with $l > 1$, and a set of states $Q$, the set of all $Q'$ reachable via a run $Q \xrightarrow{\widetilde{B}_1} Q' \xrightarrow{\widetilde{B}_2} Q''$ can be calculated in time $\mathcal{O}(exp_1(|\Delta| + |\mathcal{Q}|))$.*

*Proof.* We define the following procedure, which given a set of sets of states $\widetilde{Q}$ computes the set of sets $Q'$ and set of $(l-1)$-store automata $\widetilde{B}$ with $Q \in \widetilde{Q}$ and $Q \xrightarrow{\widetilde{B}} Q'$.

Expand$(\widetilde{Q})$

**let** $\widetilde{Q}_{next} = \emptyset$
**for each** $\{q_1, \ldots, q_m\} \in \widetilde{Q}$
**for each** set $\{(q_1, B_1, Q^1), \ldots, (q_m, B_m, Q^m)\} \subseteq \Delta$
$\widetilde{Q}_{next} = \widetilde{Q}_{next} \cup \{(\{B_1, \ldots, B_m\}, Q^1 \cup \cdots \cup Q^m)\}$
**return** $\widetilde{Q}_{next}$

The outer loop repeats at most $\mathcal{O}(exp_1(|\mathcal{Q}|))$ times. At most $\mathcal{O}(exp_1(|\Delta|))$ sets need to be enumerated during the inner loop. Hence, Expand runs in time $\mathcal{O}(exp_1(|\Delta| + |\mathcal{Q}|))$. The correctness of Expand is immediate.

To complete the algorithm, we define the following procedure,

ExpandETimes$(e, Q)$

> **let** $\widetilde{Q} = \textsc{Expand}(\{Q\})$
> **for** h = 1 **to** e
>     **for each** $(\widetilde{B}_1, \ldots, \widetilde{B}_h, Q') \in \widetilde{Q}$
>         $\widetilde{Q} = \widetilde{Q} \cup (\{(\widetilde{B}_1, \ldots, \widetilde{B}_h)\} \times \textsc{Expand}(\{Q'\}))$
> **return** $\widetilde{Q} \cap ((\mathcal{B}_l)^e \times 2^{\mathcal{Q}})$

This procedure requires $\mathcal{O}(e \times (e \times exp_1(|\Delta|)) \times exp_1(|\mathcal{Q}|))$ iterations of the loop. Each iteration requires time $\mathcal{O}(exp_1(|\Delta| + |\mathcal{Q}|))$ and consequently the procedure runs in time $\mathcal{O}(exp_1(|\Delta| + |\mathcal{Q}|))$.

By the correctness of Expand we have $(\widetilde{B}, Q') \in \widetilde{Q}$ iff we have the path $Q \xrightarrow{\widetilde{B}} Q'$ in $A$. After execution of the loop we have, by correctness of Expand, $(\widetilde{B}_1, \ldots, \widetilde{B}_e, Q') \in \widetilde{Q}$ iff we have the following path in $A$: $Q \xrightarrow{\widetilde{B}_1} \ldots \xrightarrow{\widetilde{B}_e} Q'$. □

### 5.6.2 Membership

**Proposition 5.6.3.** *Given an $n$-store (multi-)automaton $A = (\mathcal{Q}, \Sigma, \Delta, , \mathcal{Q}_f)$ and an $n$-store $w$ we can determine whether there is an accepting run over $w$ in $A$ from a given state $q \in \mathcal{Q}$ in time $\mathcal{O}(|w||\Delta||\mathcal{Q}|)$.*

*Proof.* When $w = \triangledown$ we can check membership immediately. Otherwise the algorithm is recursive. In the base case, when $n = 1$ and $w = a_1 \ldots a_m$, we present the following well-known algorithm,

> **let** $Q = \mathcal{Q}_f$
> **for** $i = m$ **downto** 1
>     $Q = \{ q' \mid (q', a_i, Q') \in \Delta \wedge Q' \subseteq Q \}$
> **return** $(q \in Q)$

This algorithm requires time $\mathcal{O}(m|\Delta||\mathcal{Q}|)$. We prove that this algorithm is correct at order-1 by induction over $m$. When $m = 1$, we have $q \in Q$ at the end of the algorithm iff there exists a transition $(q, a_1, Q') \in \Delta$ where $Q' \subseteq \mathcal{Q}_f$. When $w = a_1 a_2 \ldots a_m$ we have $q \in Q$ at the end of the algorithm iff there exists a transition $(q, a_1, Q')$ where, by induction if $q' \in Q'$ then the word $a_2 \ldots a_m$ is accepted from $q'$. Hence, we have $q \in Q$ iff there is an accepting run over $w$ from $q$.

When $n > 1$ we generalise the algorithm given above. Let $w = \gamma_1 \ldots \gamma_m$,

> **let** $Q = \mathcal{Q}_f$
> **for** $i = m$ **downto** 1
>     $Q = \{ q' \mid (q', B, Q') \in \Delta \wedge \gamma \in \mathcal{L}(B) \wedge Q' \subseteq Q \}$
> **return** $(q \in Q)$

The outer loop of the program repeats $m$ times, there are $|\Delta|$ transitions to be checked. By considering all labelling automata as a single automaton with an initial state for each (as in the backwards reachability construction), we make a single recursive call (for each $\gamma$ in $w$), obtaining all states accepting $\gamma$. Checking $\gamma \in \mathcal{L}(B)$ then requires checking whether the

appropriate initial state is in the result of the recursive call. We have $|w| = |\gamma_1| + \cdots + |\gamma_m|$, hence the algorithm requires $\mathcal{O}(|\gamma_1||\Delta_1||\mathcal{Q}| + \cdots + |\gamma_m||\Delta_1||\mathcal{Q}|) = \mathcal{O}(|w||\Delta_1||\mathcal{Q}|)$ time for the pre-computation, then $\mathcal{O}(m|\Delta_2||\mathcal{Q}|)$ time for the body of the algorithm, where $\Delta = \Delta_1 \cup \Delta_2$ is the partition of $\Delta$ into the order-$n$ and lower-order parts. Hence, we require $\mathcal{O}(|w||\Delta||\mathcal{Q}|)$ time.

We prove that this algorithm is correct at order $n > 1$ by induction over $m$. When $m = 1$, we have $q \in Q$ at the end of the algorithm iff there exists a transition $(q, B, Q') \in \Delta$ with $\gamma \in \mathcal{L}(B)$ and $Q' \subseteq \mathcal{Q}_f$. When $w = \gamma_1\gamma_2 \ldots \gamma_m$ we have $q \in Q$ at the end of the algorithm iff there exists a transition $(q, B, Q')$ where $\gamma \in \mathcal{L}(B)$ and, by induction, if $q' \in Q'$ then the word $a_2 \ldots a_m$ is accepted from $q'$. Hence, we have $q \in Q$ iff there is an accepting run over $w$ from $q$. □

### 5.6.3 Boolean Operations

We define the boolean operations on $n$-store multi-automata.

**Definition 5.6.1.** Given two $n$-store multi-automata $A^1 = (\mathcal{Q}_1, \Sigma, \Delta_1, \{q_1^1, \ldots, q_1^z\}, \mathcal{Q}_f^1)$ and $A^2 = (\mathcal{Q}_2, \Sigma, \Delta_2, \{q_2^1, \ldots, q_2^z\}, \mathcal{Q}_f^2)$, we define,

$$A^1 \cup A^2 = (\mathcal{Q}_1 \cup \mathcal{Q}_2 \cup \{q^1, \ldots, q^z\}, \Sigma, \Delta_\cup, \{q^1, \ldots, q^z\}, \mathcal{Q}_f^\cup)$$

where $q^i \notin \mathcal{Q}_1 \cup \mathcal{Q}_2$ for $i \in \{1, \ldots, z\}$ and

$$\begin{aligned} \Delta_\cup &= \Delta_1 \cup \Delta_2 \\ &\cup \ \{\ (q^i, B, Q) \mid (q_1^i, B, Q) \in \Delta_1 \ \wedge\ 1 \leq i \leq z\ \} \\ &\cup \ \{\ (q^i, B, Q) \mid (q_2^i, B, Q) \in \Delta_2 \ \wedge\ 1 \leq i \leq z\ \} \\ \\ \mathcal{Q}_f^\cup &= \mathcal{Q}_f^1 \ \cup\ \mathcal{Q}_f^2 \ \cup\ \{\ q^i \mid q_j^i \in \mathcal{Q}_f^j \ \wedge\ 1 \leq j \leq 2 \ \wedge 1 \leq i \leq z\ \} \end{aligned}$$

We define,

$$A^1 \cap A^2 = (\mathcal{Q}_1 \cup \mathcal{Q}_2 \cup \{q^1, \ldots, q^z\}, \Sigma, \Delta_\cap, \{q^1, \ldots, q^z\}, \mathcal{Q}_f^\cap)$$

where $q^i \notin \mathcal{Q}_1 \cup \mathcal{Q}_2$ for $i \in \{1, \ldots, z\}$ and at order-1 we have,

$$\begin{aligned} \Delta_\cap &= \Delta_1 \cup \Delta_2 \cup \{\ (q^i, a, Q_1 \cup Q_2) \mid (q_1^i, a, Q_1) \in \Delta_1 \wedge (q_2^i, a, Q_2) \in \Delta_2 \ \wedge\ 1 \leq i \leq z\ \} \\ \mathcal{Q}_f^\cap &= \mathcal{Q}_f^1 \ \cup\ \mathcal{Q}_f^2 \ \cup\ \{\ q^i \mid q_1^i \in \mathcal{Q}_f^1 \ \wedge\ q_2^i \in \mathcal{Q}_f^2 \ \wedge\ 1 \leq i \leq z\ \} \end{aligned}$$

Otherwise we have,

$$\begin{aligned} \Delta_\cap &= \Delta_1 \cup \Delta_2 \\ &\cup \{\ (q^i, B_1 \cap B_2, Q_1 \cup Q_2) \mid (q_1^i, B_1, Q_1) \in \Delta_1 \wedge (q_2^i, B_2, Q_2) \in \Delta_2 \wedge 1 \leq i \leq z\ \} \\ &\cup \{\ (q^i, \triangledown, \{q_f^\varepsilon\}) \mid (q_1^i, \triangledown, \{q_f^\varepsilon\}) \in \Delta_1 \wedge (q_2^i, \triangledown, \{q_f^\varepsilon\}) \in \Delta_2 \wedge 1 \leq i \leq z\ \} \\ \\ \mathcal{Q}_f^\cap &= \mathcal{Q}_f^1 \cup \mathcal{Q}_f^2 \end{aligned}$$

Where $B_1 \cap B_2$ is defined recursively.

**Property 5.6.1.** *Given two $n$-store multi-automata $A_1$ and $A_2$, we have $\mathcal{L}(A_1) \cap \mathcal{L}(A_2) = \mathcal{L}(A_1 \cap A_2)$ and $\mathcal{L}(A_1) \cup \mathcal{L}(A_2) = \mathcal{L}(A_1 \cup A_2)$.*

*Proof.* To prove the property for $A_1 \cup A_2$ in the case of an empty stack $\varepsilon$ we have for all $i \in \{1, \ldots, z\}$ that $q^i \in \mathcal{Q}_f^{\cup}$ iff we have $q_j^i \in \mathcal{Q}_f^j$ for $j = 1$ or $j = 2$. Otherwise, we observe that we have for any $i \in \{1, \ldots, z\}$ an accepting run in $A_1 \cup A_2$ of a store $[\gamma w]$,

$$q^i \xrightarrow{\gamma} Q \xrightarrow{w} Q_f$$

with $Q_f \subseteq \mathcal{Q}_f$ iff we have a run,

$$q_j^i \xrightarrow{\gamma} Q \xrightarrow{w} Q_f$$

in $A_i$ for $j = 1$ or $j = 2$.

For $A_1 \cap A_2$ we proceed by induction over $n$. When $n = 1$, in the case of the empty stack $\varepsilon$ we have for any $i \in \{1, \ldots, z\}$ that $q^i \in \mathcal{Q}_f^{\cap}$ iff $q_j^i \in \mathcal{Q}_f^j$ for $j = 1$ and $j = 2$. Otherwise, we have an accepting run of $A_1 \cap A_2$ over the word $aw$,

$$q^i \xrightarrow{a} Q_1 \cup Q_2 \xrightarrow{w} Q_f^1 \cup Q_f^2$$

with $Q_f^1 \cup Q_f^2 \subseteq \mathcal{Q}_f$ iff we have the accepting runs,

$$q_j^i \xrightarrow{a} Q_j \xrightarrow{w} Q_f^j$$

in $A_j$ for $j = 1$ and $j = 2$.

When $n > 1$, we have an accepting run of $A_1 \cap A_2$ over the word $\gamma w$ (note we may have $\gamma = \nabla$ and $w = \varepsilon$),

$$q_0 \xrightarrow{\gamma} Q_1 \cup Q_2 \xrightarrow{w} Q_f^1 \cup Q_f^2$$

with $Q_f^1 \cup Q_f^2 \subseteq \mathcal{Q}_f$ iff (via the induction hypothesis) we have the accepting runs,

$$q^i \xrightarrow{\gamma} Q_i \xrightarrow{w} Q_f^i$$

in $A_i$ for $i = 1$ and $i = 2$. ☐

We now show how to complement $n$-store multi-automata. We begin by defining an operation on sets of sets.

**Definition 5.6.2.** Given a set of sets $\{Q_1, \ldots, Q_m\}$ we define,

$$\mathit{invert}(\{Q_1, \ldots, Q_m\}) = \{\ \{q_1, \ldots, q_m\} \mid q_i \in Q_i \wedge 1 \leq i \leq m\ \}$$

**Definition 5.6.3.** Given an $n$-store multi-automaton $A = (\mathcal{Q}, \Sigma, \Delta, \{q^1, \ldots, q^z\}, \mathcal{Q}_f)$, we define $\bar{A}$ as follows.

- When $n = 1$ we assume $A$ is *total* (this is a standard assumption that can easily be satisfied by adding a sink state). We define $\bar{A} = (\mathcal{Q}, \Sigma, \Delta', \{q^1, \ldots, q^z\}, \mathcal{Q} \setminus \mathcal{Q}_f)$ where $\Delta'$ is the smallest set such that for each $q \in \mathcal{Q}$ and $a \in \Sigma$ we have,

  1. The transitions from $q$ in $\Delta$ over $a$ are $(q, a, Q_1), \ldots, (q, a, Q_m)$, and
  2. $Q_a = \mathit{invert}\left(\bigcup_{1 \in \{1,\ldots,m\}} \{Q_i\}\right)$, and
  3. $\Delta'(q, a) = Q_a$.

  Since $Q_a$ may be exponential in size, the construction runs in exponential time when $n = 1$.

- When $n > 1$ we define $\bar{A} = (\mathcal{Q} \cup \{q_f^*, q_f^\varepsilon\}, \Sigma, \Delta', \{q^1, \ldots, q^z\}, (\mathcal{Q} \cup \{q_f^*, q_f^\varepsilon\}) \setminus \mathcal{Q}_f)$ where $q_f^*, q_f^\varepsilon \notin \mathcal{Q}$, all $n$-stores are accepted from $q_f^*$ and $q_f^\varepsilon$ has no outgoing transitions.

  Furthermore $\Delta'$ is the smallest set such that for each $q \in \mathcal{Q}$ we have,

  1. The non-$\triangledown$ transitions from $q$ in $\Delta$ are $(q, B_1, Q_1), \ldots, (q, B_m, Q_m)$, and
  2. For all $\widetilde{B} \in 2^{\{B_1,\ldots,B_m\}}$ we have,

$$Q_{\widetilde{B}} = \begin{cases} \{q_f^*\} & \text{if } \widetilde{B} = \emptyset \\ invert\left(\bigcup_{B_i \in \widetilde{B}} \{Q_i\}\right) & \text{otherwise} \end{cases}$$

$$B_{\widetilde{B}} = \bigcap_{B_i \in \widetilde{B}} B_i \cap \bigcap_{B_i \notin \widetilde{B}} \bar{B}_i$$

     Note we have $\bar{B}_i$ recursively; and
  3. $\Delta'(q, B_{\widetilde{B}}) = Q_{\widetilde{B}}$, and
  4. For all $j \in \{1, \ldots, z\}$ we have $(q^j, \triangledown, \{q_f^\varepsilon\}) \in \Delta'$ iff there is no $\triangledown$-transition from $q^j$ in $A$.

  Overall, when $n > 1$ there may be an exponential blow-up in the number of transitions and the construction of each $B_{\widetilde{B}}$ may take exponential time. The construction is therefore exponential.

We now show that the above definition is correct.

**Property 5.6.2.** *Given an $n$-store multi-automaton $A$, we have $\mathcal{L}(\bar{A}^{q^j}) = \overline{\mathcal{L}(A^{q^j})}$ for all $q^j \in \{q^1, \ldots, q^z\}$.*

*Proof.* We propose the following induction hypothesis: an accepting run $q \xrightarrow{w} Q$ exists in $\bar{A}$ iff there is no accepting run $q \xrightarrow{w} Q'$ in $A$. We proceed first by induction over $n$ and then by induction over the length of the run.

When $n = 1$, and the length of the run is zero, the induction hypothesis follows since $\mathcal{Q}_f \cap (\mathcal{Q} \setminus \mathcal{Q}_f) = \emptyset$. When the length of the run is larger than zero, we begin by showing the if direction. Assume we have an accepting run,

$$q \xrightarrow{a} Q^1 \xrightarrow{w} Q$$

in $\bar{A}$ for some $a$ and $w$. Suppose for contradiction we have a run,

$$q \xrightarrow{a} Q^2 \xrightarrow{w} Q'$$

in $A$ with $Q' \subseteq \mathcal{Q}_f$. Then, by induction over the length of the run, there are no accepting runs over $w$ in $\bar{A}$ from any state in $Q^2$. In $\Delta$ we have the transition $(q, a, Q^2)$. By definition there is some $q' \in Q^2$ with $q' \in Q^1$ and consequently the accepting run $Q^1 \xrightarrow{w} Q$ cannot exist in $\bar{A}$. We have a contradiction.

In the only-if direction, assume there is no run,

$$q \xrightarrow{a} Q^1 \xrightarrow{w} Q'$$

with $Q' \subseteq \mathcal{Q}_f$ in $A$. For all transitions of the form $q \xrightarrow{a} Q^1$ (guaranteed to exist since $A$ is total) there is no accepting run $Q^1 \xrightarrow{w} Q'$. Hence, there is some $q' \in Q^1$ with no accepting

run over $w$, and by induction over the length of the run, there is an accepting run from $q'$ over $w$ in $\bar{A}$.

Let $\{(q, a, Q_1^\top), \ldots, (q, a, Q_e^\top)\}$ be the set of all transitions in $\Delta$ from $q$ over $a$. For each $i \in \{1, \ldots, e\}$, let $q_i^\top \in Q_i^\top$ be the state from which there is no accepting run over $w$ in $A$ and hence an accepting run over $w$ in $\bar{A}$. By definition of $\Delta'$ the transition $q \xrightarrow{a} \{q_1^\top, \ldots, q_e^\top\}$ exists in $\bar{A}$. Hence we have the accepting run,

$$q \xrightarrow{a} \{q_1^\top, \ldots, q_e^\top\} \xrightarrow{w} Q'$$

in $\bar{A}$ as required.

We now consider the inductive case $n > 1$. If $q = q_f^*$ or $q_f^\varepsilon$ the result is immediate. Similarly, when the length of the run is zero, then the property follows since $\mathcal{Q}_f \cap (\mathcal{Q} \cup \{q_f^\varepsilon, q_f^*\}) \setminus \mathcal{Q}_f = \emptyset$. Furthermore, since we have an (accepting) $\triangledown$-transition from $q^j$ for all $j \in \{1, \ldots, z\}$ in $A$ iff there is no (accepting) $\triangledown$-transition from $q^j$ in $\bar{A}$ the result is also straightforward in this case.

Otherwise, in the if direction, assume we have an accepting run,

$$q \xrightarrow{\gamma} Q^1 \xrightarrow{w} Q$$

in $\bar{A}$ for some $\gamma$ and $w$. Suppose for contradiction we have a run,

$$q \xrightarrow{\gamma} Q^2 \xrightarrow{w} Q'$$

in $A$ with $Q' \subseteq \mathcal{Q}_f$. Then, by induction over the length of the run, there are no accepting runs over $w$ in $\bar{A}$ from any state in $Q^2$. In $\Delta$ we have the transition $(q, B, Q^2)$ with $\gamma \in \mathcal{L}(B)$, hence $B$ must appear positively on the transition in $\Delta'$ from $q$ to $Q^1$ (else $\bar{B}$ appears, and by induction over $n$, $\gamma \notin \mathcal{L}(\bar{B})$). By definition there is some $q' \in Q^2$ with $q' \in Q^1$ and consequently the run $Q^1 \xrightarrow{w} Q$ cannot exist in $\bar{A}$. We have a contradiction.

In the only-if direction, assume there is no run,

$$q \xrightarrow{\gamma} Q^1 \xrightarrow{w} Q'$$

with $Q' \subseteq \mathcal{Q}_f$ in $A$. There are two cases.

- If there are no transitions $q \xrightarrow{\gamma} Q^1$ in $A$ then for all $q \xrightarrow{B} Q^1$ we have $\gamma \in \bar{B}$ by induction over $n$. Hence, in $\bar{A}$ we have a run,

$$q \xrightarrow{\gamma} q_f^* \xrightarrow{w} Q^*$$

  which is an accepting run as required.

- If there are transitions of the form $q \xrightarrow{\gamma} Q^1$ in $A$ then for each of these runs there is no accepting run $Q^1 \xrightarrow{w} Q'$. Hence, there is some $q' \in Q^1$ with no accepting run over $w$, and by induction over the length of the run, there is an accepting run from $q'$ over $w$ in $\bar{A}$.

  Let $\{(q, B_1^t, Q_1^t), \ldots, (q, B_e^t, Q_e^t), (q, B_1^f, Q^f), \ldots, (q, B_h^f, Q_h^f)\}$ be the set of all transitions in $\Delta$ from $q$ such that $\gamma \in B_i^t$ for all $i \in \{1, \ldots, e\}$ and $\gamma \notin B_i^f$ for all $i \in \{1, \ldots, h\}$ (and consequently $\gamma \in \bar{B}_i^f$). For each $i \in \{1, \ldots, e\}$ let $q_i^t \in Q_i^t$ be the state from which $\bar{A}$ has no accepting run over $w$ in $A$ and hence has an accepting run over $w$ in $\bar{A}$. By

definition of $\Delta'$ the transition $q \xrightarrow{B} \{q_1^t, \ldots, q_e^t\}$ with $B = B_1^t \cap \cdots \cap B_e^t \cap \bar{B}_1^f \cap \cdots \cap \bar{B}_h^f$ exists in $\bar{A}$. Hence we have the accepting run,

$$q \xrightarrow{\gamma} \{q_1^t, \ldots, q_e^t\} \xrightarrow{w} Q'$$

in $\bar{A}$ as required.

We have shown that $\bar{A}$ has an accepting run from any state iff there is no accepting run from that state in $A$ as required. □

## 5.7 Summary

In this chapter we discussed our backwards-reachability algorithm for higher-order pushdown systems. This result was first published in FoSSaCS 2007 [77]. This algorithm constructs in $n$-EXPTIME an $n$-store multi-automaton which accepts the complete set of configurations that can reach a given set of configurations $C_{Init}$. This extends work due to Bouajjani *et al.* [3] and Bouajjani and Meyer [2]. The main innovation of the approach is the careful management of transition updates that allow us to identify a series of cascading fixed points, resulting in termination. We also described an alternative proof due to Seth [23].

In the next chapter, we discuss applications of this result to reachability games and non-emptiness of higher-order pushdown automata. The application to the non-emptiness of higher-order pushdown automata shows that the problem is $n$-EXPTIME-hard, and hence, our algorithm is optimal.

# Chapter 6

# Applications

In this chapter we discuss some of the applications of our algorithm to decision problems over higher-order PDSs. In particular, we investigate LTL and branching-time model-checking over non-deterministic higher-order pushdown systems, reachability games, the non-emptiness of non-deterministic higher-order pushdown automata and goal sets in Büchi games over higher-order pushdown systems.

The non-emptiness in Section 6.4 also shows the $n$-EXPTIME-hardness of the reachability problem for higher-order pushdown systems.

## 6.1 Model-Checking Linear-Time Temporal Logics

Bouajjani *et al.* use their backwards reachability algorithm to provide a model-checking algorithm for linear-time temporal logics over the configuration graphs of pushdown systems [3]. In this section we show that this work permits a simple generalisation to higher-order PDSs.

Let $Prop$ be a finite set of atomic propositions and $(\mathcal{P}, \mathcal{D}, \Sigma)$ be a higher-order PDS with a labelling function $\Lambda : \mathcal{P} \rightarrow 2^{Prop}$ which assigns to each control state a set of propositions deemed to be *true* at that state. Given formula $\phi$ of an $\omega$-regular logic such as LTL or $\mu$TL, we calculate the set of configurations $C$ of $(\mathcal{P}, \mathcal{D}, \Sigma)$ such that every run from each $c \in C$ satisfies $\phi$.

It is well known that any formula of an $\omega$-regular logic has a Büchi automaton representation [139, 82, 81] etc.. We form the product of the higher-order PDS and the Büchi automaton corresponding to the negation of $\phi$. This gives us a higher-order Büchi PDS; that is, a higher-order PDS with a set $\mathcal{F}$ of accepting control states. Thus, model-checking reduces to the non-emptiness problem for higher-order Büchi PDSs. Specifically, we compute the set of configurations from which there is an infinite run visiting configurations with control states in $\mathcal{F}$ infinitely often. Note that $C$ is the complement of this set.

This problem can be reduced further to a number of applications of the reachability problem. We present a generalisation of the reduction of Bouajjani *et al.*. Let $[^1 a]^1$ denote the order-1 stack consisting of a single character $a$ and $[^l a]^l$ for $l > 1$ denote the stack consisting of a single order-$(l-1)$ stack $[^{(l-1)} a]^{(l-1)}$.

**Proposition 6.1.1.** *Let $c$ be a configuration of an order-$n$ Büchi PDS $BP$. It is the case that $BP$ has an accepting run from $c$ iff there exist distinct configurations $\langle p^j, [^n a]^n \rangle$ and $\langle p^j, \gamma_2 \rangle$ with $top_1(\gamma_2) = a$ and a configuration $\langle p^f, \gamma_1 \rangle$ such that $p^f \in \mathcal{F}$ and,*

1. $c \overset{*}{\hookrightarrow} \langle p^j, \gamma_3 \rangle$ *for some* $\gamma_3$ *with* $top_1(\gamma_3) = a$, *and*

2. $\langle p^j, [^n a]^n \rangle \overset{*}{\hookrightarrow} \langle p^f, \gamma_1 \rangle \overset{*}{\hookrightarrow} \langle p^j, \gamma_2 \rangle$

*Proof.* $\Rightarrow$: Every higher-order stack may be flattened into a well bracketed string, as per Definition 2.8.1. Given a suffix of an $n$-store $w$, let $comp(w)$ be a number of symbols "[" added to the beginning of $w$ to form an $n$-store proper.

Given an accepting run of BP $\rho = c_0 c_1 \ldots$, there exists a sequence of suffixes $w_1, w_2, \ldots$ such that there exists an increasing sequence of natural numbers $i_1, i_2, \ldots$ and for all $j > 0$ and $i \geq i_j$ $c_i$ has a stack with the suffix $w_j$. Additionally $c_{i_j}$ has the $n$-store $comp(w_j)$ and $w_i$ is a suffix of $w_j$ for all $i \leq j$ (it may be the case that $w_i = w_j$). Take the sequence $c_{i_1} c_{i_2} \ldots$. Due to the finiteness of $\mathcal{P}$ and $\Sigma$ there must be $p, a$ with an infinite number of $c_{i_j}$ with control state $p$ and a stack whose $top_1$ element is $a$. Furthermore, since $\rho$ is accepting, we must have distinct $c_{i_a}$ and $c_{i_b}$ with $p$ as their control states and $a$ as the $top_1$ element, with a $c_f$ whose control state is $p^f \in \mathcal{F}$,

$$c_0 \overset{*}{\hookrightarrow} c_{i_a} \overset{*}{\hookrightarrow} c_f \overset{*}{\hookrightarrow} c_{i_b}$$

We have (1) from $c_0 \overset{*}{\hookrightarrow} c_{i_a}$. By definition of $c_{i_1}, c_{i_2} \ldots$ we have $c_{i_a} = \langle p, comp(w_{i_a}) \rangle$ and all configurations between $c_{i_a}$ and $c_{i_b}$ have the suffix $w_{i_a}$. This implies,

$$\langle p, [^n a]^n \rangle \overset{*}{\hookrightarrow} \langle p^f, u \rangle \overset{*}{\hookrightarrow} \langle p, v \rangle$$

with $top_1(v) = a$. Hence, (2) holds as required.

$\Leftarrow$: From (1) we have $c \overset{*}{\hookrightarrow} \langle p, \gamma_1 \rangle$ with $top_1(\gamma_1) = a$. From (2) we can construct a path,

$$\langle p, \gamma_2 \rangle \overset{*}{\hookrightarrow} \langle p^f, \gamma_3 \rangle \overset{*}{\hookrightarrow} \langle p, \gamma_4 \rangle$$

with $p^f \in \mathcal{F}$ and $top_1(\gamma_4) = a$ for any $\gamma_2$ with $top_1(\gamma_2) = a$. Thus, through infinite applications of (2), we can construct an accepting run of $BP$. □

We reformulate these conditions as follows, where $C_n^{\Sigma}$ is the set of all order-$n$ stacks over the alphabet $\Sigma$. We remind the reader that $B_n^a$ is the $n$-store automaton accepting all $n$-stores $\gamma$ such that $top_1(\gamma) = a$.

1. $c \in Pre^*(\{p^j\} \times \mathcal{L}(B_n^a))$,

2. $\langle p^j, [^n a]^n \rangle \in Pre^*((\mathcal{F} \times C_n^{\Sigma}) \cap Pre^+(\{p^j\} \times \mathcal{L}(B_n^a)))$

We can compute the set of pairs $\langle p^j, [^n a]^n \rangle$ satisfying (2) in $(n-1)$-EXPTIME by calculating $Pre^*(\{p^j\} \times \mathcal{L}(B_n^a))$ over the higher-order PDS described below. Note that the $(n-1)$-EXPTIME complexity is lower than the $n$-EXPTIME claimed in Section 5.4.7. This is because there is no alternation in the PDS. Hence, no alternation is required at order-$n$ of the automaton describing $Pre^*(\{p^j\} \times \mathcal{L}(B_n^a))$ (since $B_n^a$ also requires no alternation). Hence, we avoid the final exponential blow up.

**Definition 6.1.1.** Given an order-$n$ Büchi PDS $BP = (\mathcal{P}, \mathcal{D}, \Sigma, \mathcal{F})$ we define $BP' = (\mathcal{P} \times \{0, 1\}, \mathcal{D}', \Sigma)$ where,

$$\begin{aligned} \mathcal{D}' = \; & \{ \; ((p,0), b, o, (p',0)) \mid p \in \mathcal{P} \cap \overline{\mathcal{F}} \; \wedge \; (p, b, o, p') \in \mathcal{D} \; \} \; \cup \\ & \{ \; ((p,0), b, o, (p',1)) \mid p \in \mathcal{F} \; \wedge \; (p, b, o, p') \in \mathcal{D} \; \} \; \cup \\ & \{ \; ((p,1), b, o, (p',1)) \mid (p, b, o, p') \in \mathcal{D} \; \} \end{aligned}$$

**Lemma 6.1.1.** *There exists a run* $\langle (p, 0), [^n a]^n \rangle \stackrel{*}{\hookrightarrow} \langle (p, 1), w' \rangle$ *with* $w' \in \mathcal{L}(B^a_n)$ *in* $BP'$ *iff* $\langle p, [^n a]^n \rangle$ *satisfies (2).*

*Proof.* We begin by showing if $\langle p, [^n a]^n \rangle$ satisfies (2), then a run $\langle (p, 0), [^n a]^n \rangle \stackrel{*}{\hookrightarrow} \langle (p, 1), \gamma \rangle$ with $\gamma \in \mathcal{L}(B^a_n)$ exists in $BP$. The run over $BP$ satisfying (2) can be split into two parts,

$$\langle p, [^n a]^n \rangle \stackrel{*}{\hookrightarrow} \langle p^f, \gamma_f \rangle \stackrel{*}{\hookrightarrow} \langle p, \gamma \rangle$$

with $\gamma \in \mathcal{L}(B^a_n)$ and $p^f$ is the first accepting state seen in the run. We consider each part separately.

- Suppose we have a run,

  $$\langle p_0, \gamma_0 \rangle \hookrightarrow \ldots \hookrightarrow \langle p_m, \gamma_m \rangle$$

  such that $p_m$ is the only accepting control state in the run. This run is derived from a sequence of commands $d_1, \ldots, d_m$. Let $d_i = (p_{i-1}, a_i, o_i, p_i)$ for all $i \in \{1, \ldots, m\}$. We show the run,

  $$\langle (p_0, 0), \gamma_0 \rangle \hookrightarrow \ldots \hookrightarrow \langle (p_m, 0), \gamma_m \rangle$$

  exists in $BP'$ by induction over $m$. In the base case $m = 0$ and the result is trivial. Suppose we have,

  $$\langle (p_1, 0), \gamma_1 \rangle \hookrightarrow \ldots \hookrightarrow \langle (p_m, 0), \gamma_m \rangle$$

  by the induction hypothesis. Since $d_1 = (p_0, a_1, o_1, p_1)$ and $p_0 \notin \mathcal{F}$, we have that $((p_0, 0), a_1, o_1, (p_1, 0))$ is in $\mathcal{D}'$. Hence we have the run,

  $$\langle (p_0, 0), \gamma_0 \rangle \hookrightarrow \ldots \hookrightarrow \langle (p_m, 0), \gamma_m \rangle$$

  as required.

- We have $\langle p^f, \gamma_f \rangle \in (\mathcal{F} \times C^\Sigma_n) \cap Pre^+(\{p\} \times \mathcal{L}(B^a_n)))$, we show there exists the run $\langle (p^f, 0), \gamma_f \rangle \stackrel{*}{\hookrightarrow} \langle (p, 1), \gamma \rangle$ in BP' with $\gamma \in \mathcal{L}(B^a_n)$.

  We have the run $\langle p^f, \gamma_f \rangle \stackrel{*}{\hookrightarrow} \langle p, \gamma \rangle$ in BP with $\gamma \in \mathcal{L}(B^a_n)$. This run is of the form,

  $$\langle p_0, \gamma_0 \rangle \hookrightarrow \langle p_1, \gamma_1 \rangle \hookrightarrow \ldots \hookrightarrow \langle p_m, \gamma_m \rangle$$

  with $m \geq 1$, $p_0 = p^f$, $\gamma_0 = \gamma_f$, $p_m = p$ and $\gamma_m = \gamma$. The run is the consequence of a sequence of commands $d_1, \ldots, d_m$. Let $d_i = (p_{i-1}, a_i, o_i, p_i)$. Since $p_0 \in \mathcal{F}$ we have $((p_0, 0), a_1, o_1, (p_1, 1))$ in $\mathcal{D}'$ by definition. Furthermore, for $i \in \{2, \ldots, m\}$ we have $((p_{i-1}, 1), a_i, o_i, (p_i, 1))$ in $\mathcal{D}'$. We have the run

  $$\langle (p_0, 0), \gamma_0 \rangle \hookrightarrow \langle (p_1, 1), \gamma_1 \rangle \hookrightarrow \ldots \hookrightarrow \langle (p_m, 1), \gamma_m \rangle$$

  in BP' therefrom.

The proof of this direction follows immediately.

We now consider the proof in the opposite direction. Suppose we have $\langle (p, 0), [^n a]^n \rangle \stackrel{*}{\hookrightarrow} \langle (p, 1), \gamma \rangle$ with $\gamma \in \mathcal{L}(B^a_n)$. From the definition of $\mathcal{D}'$ it follows that the run is of the form,

$$\langle (p, 0), [^n a]^n \rangle \hookrightarrow \ldots \hookrightarrow \langle (p^f, 0), \gamma_f \rangle \hookrightarrow \langle (p', 1), \gamma' \rangle \hookrightarrow \ldots \hookrightarrow \langle (p, 1), \gamma \rangle$$

where the second element of each control state/flag pair changes only in the position shown. Furthermore, $p^f$ is the first occurrence of an accepting control state in $BP$. This run is the result of a sequence of commands $d_1, \ldots, d_m$ where $m \geq 1$. From a simple projection on the first element of each control state/flag pair, we immediately derive a sequence commands $d'_1, \ldots, d'_m$ in $\mathcal{D}$ and the following run of $BP$,

$$\langle p, [^n a]^n \rangle \hookrightarrow \ldots \hookrightarrow \langle p^f, \gamma_f \rangle \hookrightarrow \langle p', \gamma' \rangle \hookrightarrow \ldots \hookrightarrow \langle p, \gamma \rangle$$

Since $\langle p^f, \gamma_f \rangle$ and $\langle p', \gamma' \rangle$ must be distinct, the existence of this run implies $\langle p, [^n a]^n \rangle$ satisfies (2).

Since $BP'$ is twice as large as $BP$, and no alternation is used, $Pre^*(\{p^j\} \times \mathcal{L}(B^a_n))$ can be calculated in $(n-1)$-EXPTIME. □

To construct an $n$-store automaton accepting all configurations from which there is an accepting run, we calculate the configurations $\langle p^j, [^n a]^n \rangle$ satisfying the second condition. Since there are only finitely many $p^j \in \mathcal{P}$ and $a \in \Sigma$ we can perform a simple enumeration. We then construct an $n$-store automaton $A$ corresponding to the union of the $n$-store automata accepting these configurations and compute $Pre^*(\mathcal{L}(A))$.

**Theorem 6.1.1.** *Given an order-$n$ Büchi PDS $BP = (\mathcal{P}, \mathcal{D}, \Sigma, \mathcal{F})$, we can calculate in $(n-1)$-EXPTIME the set of configurations $C$ such that from all $c \in C$ there is an accepting run of $BP$.*

*Proof.* We appeal to Lemma 6.1.1 for each $p^j$ and $a$ (of which there are polynomially many) to construct an $n$-store automaton $\mathcal{O}(exp_{n-1}(2 \times |\mathcal{P}|))$ in size which accepts $\langle p^j, [^n a]^n \rangle$ iff it satisfies (2). Membership can be checked in polynomial time (Proposition 5.6.3).

It is straightforward to construct an automaton $A$ polynomial in size which accepts $\langle p, w \rangle$ iff $\langle p, [^n top_1(w)]^n \rangle$ satisfies (2). We can construct $Pre^*(\mathcal{L}(A))$ in $(n-1)$-EXPTIME. Thus, the algorithm requires $(n-1)$-EXPTIME. □

**Corollary 6.1.1.** *Given an order-$n$ PDS $(\mathcal{P}, \mathcal{D}, \Sigma)$ with a labelling function $\Lambda : \mathcal{P} \rightarrow 2^{Prop}$ and a formula $\phi$ of an $\omega$-regular logic, we can calculate in $(n+1)$-EXPTIME the set of configurations $C$ of $(\mathcal{P}, \mathcal{D}, \Sigma)$ such that every run from each $c \in C$ satisfies $\phi$.*

*Proof.* The construction of $BP$ is exponential in size. Hence, we construct the $n$-store multi-automaton $A$ that accepts the set of configurations from which there is a run satisfying the negation of $\phi$ as described above in time $\mathcal{O}(exp_{n-1}(exp_1(|\phi|)))$. To calculate $C$ we complement $A$ as described in Appendix 5.6.3. This may include an exponential blow-up in the transition relation of $A$, hence we have $(n+1)$-EXPTIME. □

Observe that since we can test $c \in C$ by checking $c \notin \mathcal{L}(A)$ where $A$ is defined as above, we may avoid the complementation step, giving us an $n$-EXPTIME algorithm. Currently we do not have a tight lower bound for the LTL model-checking problem. From the $(n-1)$-EXPTIME completeness of the emptiness problem for HOPDA [59], we obtain an $(n-1)$-EXPTIME lower bound for LTL model-checking. The final exponential blow up in our algorithm comes from the translation from LTL to Büchi automata. We believe it is unlikely that such a blow is avoidable.

## 6.2 Reachability Games

Our algorithm may be used to compute the winning region for a player in a two-player reachability game over higher-order PDSs. This generalises a result due to Cachat [127]. We call our players Éloïse and Abelard.

**Definition 6.2.1.** Given an order-$n$ PDS $(\mathcal{P}, \mathcal{D}, \Sigma)$, an order-$n$ **Pushdown Reachability Game** (PRG) $(\mathcal{P}, \mathcal{D}, \Sigma, \mathcal{R})$ over the order-$n$ PDS is given by a partition $\mathcal{P} = \mathcal{P}_A \uplus \mathcal{P}_E$ and a set $\mathcal{R}$ of configurations considered winning for Éloïse.

We write $\langle p, \gamma \rangle \in \mathcal{C}_E$ iff $p \in \mathcal{P}_E$ and $\langle p, \gamma \rangle \in \mathcal{C}_A$ iff $p \in \mathcal{P}_A$. From a configuration $\langle p, \gamma \rangle$ play proceeds as follows:

- If $\langle p, \gamma \rangle \in \mathcal{C}_A$, Abelard chooses a move $(p, a, o, p') \in \mathcal{D}$ with $top_1(\gamma) = a$ and $o(\gamma)$ defined. Play moves to the configuration $\langle p', o(\gamma) \rangle$.
- If $\langle p, \gamma \rangle \in \mathcal{C}_E$, Éloïse chooses a move $(p, a, o, p') \in \mathcal{D}$ with $top_1(\gamma) = a$ and $o(\gamma)$ defined. Play moves to the configuration $\langle p', o(\gamma) \rangle$.

Éloïse wins the game iff play reaches a configuration $\langle p, \gamma \rangle$ where $\langle p, \gamma \rangle \in \mathcal{R}$ or $p \in \mathcal{P}_A$ and Abelard is unable to choose a move. Abelard wins otherwise.

The **winning region** for a given player is the set of all configurations from which that player can force a win. The winning region for Éloïse can be characterised using an *attractor* $Attr_E(\mathcal{R})$ defined as follows,

$$\begin{array}{rcl} Attr_E^0(\mathcal{R}) & = & \mathcal{R} \\ Attr_E^{i+1}(\mathcal{R}) & = & Attr_E^i(\mathcal{R}) \cup \{ \ c \in \mathcal{C}_E \mid \exists c'.c \hookrightarrow c' \wedge c' \in Attr_E^i(\mathcal{R}) \ \} \\ & & \cup \ \{ \ c \in \mathcal{C}_A \mid \forall c'.c \hookrightarrow c' \Rightarrow c' \in Attr_E^i(\mathcal{R}) \ \} \\ Attr_E(\mathcal{R}) & = & \bigcup_{i \geq 0} Attr_E^i(\mathcal{R}) \end{array}$$

Conversely, the winning region for Abelard is $\overline{Attr_E(\mathcal{R})}$. Intuitively, from a position in $Attr_E^i(\mathcal{R})$, Éloïse's winning strategy is to simply choose a move such that the next configuration is in $Attr_E^{i-1}(\mathcal{R})$. Abelard's strategy is to avoid Éloïse's winning region.

We can use backwards-reachability for order-$n$ APDSs to calculate $Attr_E(\mathcal{R})$, and hence the winning regions of both Abelard and Éloïse. To simplify the reduction, we make a *totality* assumption. That is, we assume a bottom-of-the-stack symbol $\perp$ that is never popped nor pushed, and for all $a \in \Sigma \cup \{\perp\}$ and control states $p \in \mathcal{P}$, there exists a command $(p, a, o, p') \in \mathcal{D}$. This can be ensured by adding sink states $p^E_{lose}$ and $p^A_{lose}$ from which Éloïse and Abelard lose the game. In particular, for every $p \in \mathcal{P}$ and $a \in \Sigma \cup \{\perp\}$ we have $(p, a, push_a, p^x_{lose})$ where $x = E$ if $p \in \mathcal{P}_E$ or $x = A$ otherwise. Furthermore, the only commands available from $p^x_{lose}$ are of the form $(p^x_{lose}, a, push_a, p^x_{lose})$ for $x \in \{A, E\}$. To ensure that $p^A_{lose}$ is losing for Abelard, we set $\langle p^A_{lose}, \gamma \rangle \in \mathcal{R}$ for all $\gamma$. Conversely, $\langle p^E_{lose}, \gamma \rangle \notin \mathcal{R}$ for all $\gamma$.

**Definition 6.2.2.** Given an order-$n$ PRG $(\mathcal{P}, \mathcal{D}, \Sigma, \mathcal{R})$ we define an order-$n$ APDS $(\mathcal{P}, \mathcal{D}', \Sigma)$ where,

$$\begin{array}{rcl} \mathcal{D}' & = & \{ \ (p, a, \{(o, p')\}) \mid (p, a, o, p') \in \mathcal{D} \wedge p \in P_E \ \} \\ & & \cup \ \{ \ (p, a, \{ \ (o, p') \mid (p, a, o, p') \in \mathcal{D} \ \}) \mid p \in \mathcal{P}_A \ \} \end{array}$$

Furthermore, let $R_{stuck}$ be the set of configurations $\langle p, \nabla \rangle$ such that $p \in \mathcal{P}_A$. The set $\mathcal{R}_{stuck}$ is regular and represents the configurations reached if Abelard performs an move with an undefined next stack.

Let $C_A^{\triangledown}$ be the set of order-$n$ configurations with an undefined stack and a control state belonging to Abelard.

**Theorem 6.2.1.** *Given an order-n PRG, where $\mathcal{R}$ is a regular set of configurations, and an order-n APDS as defined above, $Attr_E(\mathcal{R})$ is regular and equivalent to $Pre^*(\mathcal{R} \cup \mathcal{R}_{stuck}) \setminus C_A^{\triangledown}$. Hence, computing the winning regions in the order-n PRG is n-EXPTIME.*

*Proof.* Let $\mathcal{R}' = \mathcal{R} \cup \mathcal{R}_{stuck}$. Since the size of an $n$-store multi-automaton recognising $\mathcal{R}_{stuck}$ is linear, the complexity follows from the complexity of computing $Pre^*(\mathcal{R}')$.

We show $Attr_E(\mathcal{R}) = Pre^*(\mathcal{R}') \setminus \mathcal{C}_A^{\triangledown}$. We begin by proving $Attr_E(\mathcal{R}) \supseteq Pre^*(\mathcal{R}') \setminus \mathcal{C}_A^{\triangledown}$.

Take a configuration $\langle p, \gamma \rangle \in Pre^*(\mathcal{R}') \setminus \mathcal{C}_A^{\triangledown}$. We show $\langle p, \gamma \rangle \in Attr_E(\mathcal{R})$ by induction over the shortest path $\langle p, \gamma \rangle \overset{*}{\hookrightarrow} C$ of the order-$n$ APDS with $C \subseteq \mathcal{R}'$.

For the base case, we have $\langle p, \gamma \rangle \in \mathcal{R}' \setminus \mathcal{C}_A^{\triangledown}$. Hence, $\langle p, \gamma \rangle \in Attr_E(\mathcal{R})$ since $\mathcal{R} \subseteq Attr_E(\mathcal{R})$.

Now, suppose we have $\langle p, \gamma \rangle \hookrightarrow C$ via the command $d = (p, a, OP)$ in the higher-order APDS with $C \in Pre^*(\mathcal{R}) \setminus \mathcal{C}_A^{\triangledown}$ and by induction $C \subseteq Attr_E^i(\mathcal{R})$ for some $i$. There are two cases,

- If $p \in \mathcal{P}_A$ then for each $(o, p') \in OP$ and hence each move $(p, a, o, p')$ in the higher-order PDS we have a corresponding $\langle p', \gamma' \rangle \in C$. We have either $\langle p', \gamma' \rangle \in Pre^*(\mathcal{R}') \setminus \mathcal{C}_A^{\triangledown}$ or we have $\langle p', \gamma' \rangle = \langle p, \triangledown \rangle$.

  If we have $\langle p', \gamma' \rangle \in Pre^*(\mathcal{R}') \setminus \mathcal{C}_A^{\triangledown}$ then $\langle p', \gamma' \rangle \in Attr_E^i(\mathcal{R})$ for some $i$ by induction.

  If we have $\langle p', \gamma' \rangle = \langle p, \triangledown \rangle$ then $o(\gamma)$ is undefined. Hence $(p, a, o, p')$ is not a valid move for Abelard.

  Hence we have $\langle p, \gamma \rangle \in \mathcal{C}_A$ and $\forall c'.\langle p, \gamma \rangle \hookrightarrow c' \Rightarrow c' \in Attr_E^i(\mathcal{R})$ which implies $\langle p, \gamma \rangle \in Attr_E^{i+1}(\mathcal{R}) \subseteq Attr_E(\mathcal{R})$.

- If $p \in \mathcal{P}_E$ then $C = \{\langle p', o(\gamma) \rangle\}$ and $(p, a, o, p') \in \mathcal{D}$. Thus, we have $\exists c'.\langle p, \gamma \rangle \hookrightarrow c' \wedge c' \in Attr_E^i(\mathcal{R})$ and $\langle p, \gamma \rangle \in \mathcal{C}_E$. Therefore $\langle p, \gamma \rangle \in Attr_E^{i+1}(\mathcal{R}) \subseteq Attr_E(\mathcal{R})$.

Thus, we have $Attr_E(\mathcal{R}) \supseteq Pre^*(\mathcal{R}') \setminus \mathcal{C}_A^{\triangledown}$ as required.

To show $Attr_E(\mathcal{R}) \subseteq Pre^*(\mathcal{R}') \setminus \mathcal{C}_A^{\triangledown}$ we use induction over $i$ in $Attr_E(\mathcal{R}) = \bigcup_{i \leq 0} Attr_E^i(\mathcal{R})$. When $i = 0$ we have $Attr_E^0(\mathcal{R}) = \mathcal{R} \subseteq \mathcal{R}' \setminus \mathcal{C}_A^{\triangledown} \subseteq Pre^*(\mathcal{R}') \setminus \mathcal{C}_A^{\triangledown}$. For $i > 1$ there are two cases for all $c$ such that $c \notin Attr_E^{i-1}(\mathcal{R})$ and $c \in Attr_E^i(\mathcal{R})$,

- $c \in \{\ c \in \mathcal{C}_E \mid \exists c'.c \hookrightarrow c' \wedge c' \in Attr_E^{i-1}(\mathcal{R})\ \}$.

  Hence there is some command $d = (p, a, o, p')$ in the higher-order PDS and command $(p, a, \{(o, p')\})$ in the higher-order APDS. By induction $c' \in Pre^*(\mathcal{R}') \setminus \mathcal{C}_A^{\triangledown}$ and $c = \langle p, \gamma \rangle$ and $c' = \langle p', o(\gamma) \rangle$. Hence $c \in Pre^*(\mathcal{R}') \setminus \mathcal{C}_A^{\triangledown}$.

- $c \in \{\ c \in \mathcal{C}_A \mid \forall c'.c \hookrightarrow c' \Rightarrow c' \in Attr_E^{i-1}(\mathcal{R})\ \}$.

  Let $c = \langle p, \gamma \rangle$. We have $d = (p, a, OP)$ in the higher-order APDS such that for all moves $(p, a, o, p')$ we have $(o, p') \in OP$. If $o(\gamma)$ is defined, we have $\langle p, \gamma \rangle \hookrightarrow \langle p', o(\gamma) \rangle$ and $\langle p', o(\gamma) \rangle \in Pre^*(\mathcal{R}')$ by induction. If $o(\gamma)$ is undefined, then since we have $\langle p, \triangledown \rangle \in \mathcal{R}'$ we have $\langle p, \triangledown \rangle \in Pre^*(\mathcal{R}')$.

  Thus, we have $\langle p, \gamma \rangle \hookrightarrow C$ via an application of the command $d$ such that $C \subseteq Pre^*(\mathcal{R}')$. Hence $\langle p, \gamma \rangle \in Pre^*(\mathcal{R}')$ and since $\gamma \neq \triangledown$, we have $\langle p, \gamma \rangle \in Pre^*(\mathcal{R}') \setminus \mathcal{C}_A^{\triangledown}$ as required.

Thus, we have $Attr_E(\mathcal{R}) = Pre^*(\mathcal{R}') \setminus \mathcal{C}_A^{\triangledown}$. □

## 6.3 Model-Checking Branching-Time Temporal Logics

Generalising a further result of Bouajjani *et al.* [3], we show that backwards-reachability for higher-order APDSs may be used to perform model-checking for the alternation-free (propositional) $\mu$-calculus over higher-order PDSs. Common logics such as CTL are sub-logics of the alternation-free $\mu$-calculus.

### Preliminaries

Given a set of atomic propositions *Prop* and a finite set of variables $\chi$, the propositional $\mu$-calculus is defined by the following grammar,

$$\phi \ := \ \pi \in Prop \mid X \in \chi \mid \neg\phi \mid \phi_1 \cup \phi_2 \mid \ \diamond \phi \mid \mu X.\phi$$

with the condition that, for a formula $\mu X.\phi$, $X$ must occur under an even-number of negations. This ensures that the logic is monotonic. As well as the usual abbreviations for $\Rightarrow$ and $\wedge$, we may also use, $\Box\phi = \neg \diamond \neg\phi$, $\nu X.\phi(X) = \neg\mu X.\neg\phi(\neg X)$ and $\sigma$ for either $\mu$ or $\nu$. A $\sigma$-formula is of the form $\sigma X.\phi$.

A variable $X$ is bound in $\phi$ if it occurs as part of a sub-formula $\sigma X.\phi'(X)$. We call an unbound variable *(*free) and write $\phi(X)$ to indicate that $X$ is free in $\phi$. A *closed* formula has no variables occurring free, otherwise the formula is *open*.

Formulae in *positive normal form* are defined by the following syntax,

$$\phi \ := \ \pi \in Prop \mid \neg\pi \mid X \in \chi \mid \phi_1 \cup \phi_2 \mid \phi_1 \cap \phi_2 \mid \ \diamond \phi \mid \Box\phi \mid \mu X.\phi \mid \nu X.\phi$$

We can translate any formula into positive normal form by "pushing in" the negations using the abbreviations defined above.

A $\sigma$-sub-formula of $\sigma X.\phi(X)$ is *proper* iff it does not contain any occurrence of $X$. We are now ready to define the alternation-free $\mu$-calculus:

**Definition 6.3.1.** The alternation-free $\mu$-calculus is the set of formulae in positive normal form such that for every $\sigma$-sub-formula $\psi$ of $\phi$ we have,

- If $\psi$ is $\mu$-formula, then all $\nu$-sub-formulae of $\psi$ are proper, and
- If $\psi$ is a $\nu$-formula, then all $\mu$-sub-formulae of $\psi$ are proper.

The *closure* $cl(\phi)$ of a formula $\phi$ is the smallest set such that,

- If $\psi_1 \wedge \psi_2 \in cl(\phi)$ or $\psi \vee \psi \in cl(\phi)$, then $\psi_1 \in cl(\phi)$ and $\psi_2 \in cl(\phi)$, and
- If $\diamond\psi \in cl(\phi)$ or $\Box\psi \in cl(\phi)$, then $\psi \in cl(\phi)$, and
- If $\sigma X.\psi(X) \in cl(\phi)$, then $\psi(\sigma X.\psi(X)) \in cl(\phi)$.

The closure of any formula is a finite set whose size is bounded by the length of the formula.

Finally, we give the semantics of the $\mu$-calculus over higher-order PDSs. Given a formula $\phi$, an order-$n$ PDS $(\mathcal{P}, \mathcal{D}, \Sigma)$, a labelling function $\Lambda : \mathcal{P} \to 2^{Prop}$, and a valuation function

$\mathcal{V}$ assigning a set of configurations to each variable $X \in \chi$, the set of configurations $[\![\phi]\!]_{\mathcal{V}}$ satisfying $\phi$ is defined,

$$\begin{array}{rcl} [\![\pi]\!]_{\mathcal{V}} & = & \Lambda^{-1}(\pi) \times C_n^{\Sigma} \\ [\![X]\!]_{\mathcal{V}} & = & \mathcal{V}(X) \\ [\![\neg\psi]\!]_{\mathcal{V}} & = & (\mathcal{P} \times C_n^{\Sigma}) \setminus [\![\psi]\!]_{\mathcal{V}} \\ [\![\psi_1 \vee \psi_2]\!]_{\mathcal{V}} & = & [\![\psi_1]\!]_{\mathcal{V}} \cup [\![\psi_2]\!]_{\mathcal{V}} \\ [\![\diamond\psi]\!]_{\mathcal{V}} & = & Pre([\![\psi]\!]_{\mathcal{V}}) \\ [\![\mu X.\psi]\!]_{\mathcal{V}} & = & \bigcap\{\ C \subseteq \mathcal{P} \times C_n^{\Sigma} \mid [\![\psi]\!]_{\mathcal{V}[X \mapsto C]} \subseteq C\ \} \end{array}$$

where $\mathcal{V}[X \mapsto C]$ is the valuation mapping all variables $Y \neq X$ to $\mathcal{V}(Y)$ and $X$ to $C$.

### Model-Checking the Alternation-Free $\mu$-Calculus

Given an order-$n$ PDS $(\mathcal{P}, \mathcal{D}, \Sigma)$ with a labelling function $\Lambda : \mathcal{P} \rightarrow 2^{Prop}$, a formula $\phi$ of the alternation-free $\mu$-calculus, and a valuation $\mathcal{V}$ we show that we can generalise the construction of Bouajjani *et al.* to produce an $n$-store multi-automata $A_\phi$ accepting the set $[\![\phi]\!]_{\mathcal{V}}$.

Initially, we only consider formulae whose $\sigma$-sub-formulae are $\mu$-formulae. We construct a product of the higher-order PDS and the usual "game" interpretation of $\phi$ [117, 33] as follows: observing that the commands of the form $(_, a, push_a, _)$ do not alter the contents of the stack, we construct the order-$n$ PRG $A = (\mathcal{P}^{(\mathcal{P},\phi)}, \mathcal{D}^{\phi}_{\mathcal{P}}, \Sigma, \mathcal{R})$ where $\mathcal{P}^{(\mathcal{P},\phi)}_A$, $\mathcal{P}^{(\mathcal{P},\phi)}_E$ and $\mathcal{D}^{\phi}_{\mathcal{P}}$ are the smallest sets such that for every $(p, \psi) \in \mathcal{P} \times cl(\phi)$ and $a \in \Sigma$,

- If $\psi = \psi_1 \vee \psi_2$, then $(p, \psi) \in \mathcal{P}^{(\mathcal{P},\phi)}_E$ and the commands $((p, \psi), a, push_a, (p, \psi_1))$ and $((p, \psi), a, push_a, (p, \psi_2))$ are in $\mathcal{D}^{\phi}_{\mathcal{P}}$,
- If $\psi = \psi_1 \wedge \psi_2$, then $(p, \psi) \in \mathcal{P}^{(\mathcal{P},\phi)}_A$ and the commands $((p, \psi), a, push_a, (p, \psi_1))$ and $((p, \psi), a, push_a, (p, \psi_2))$ are in $\mathcal{D}^{\phi}_{\mathcal{P}}$,
- If $\psi = \mu X.\psi'(X)$, then $(p, \psi) \in \mathcal{P}^{(\mathcal{P},\phi)}_A$ and the command $((p, \psi), a, push_a, (p, \psi'(\psi)))$ is in $\mathcal{D}^{\phi}_{\mathcal{P}}$,
- If $\psi = \diamond\psi'$ and $(p, a, o, p') \in \mathcal{D}$, then $(p, \psi) \in \mathcal{P}^{(\mathcal{P},\phi)}_E$ and $((p, \psi), a, o, (p', \psi'))$ is in $\mathcal{D}^{\phi}_{\mathcal{P}}$,
- If $\psi = \Box\psi'$, then $(p, \psi) \in \mathcal{P}^{(\mathcal{P},\phi)}_A$ and for every $(p, a, o, p') \in \mathcal{D}$ we have the command $((p, \psi), a, o, (p', \psi'))$ in $\mathcal{D}^{\phi}_{\mathcal{P}}$.

Finally, we define the set of configurations $\mathcal{R}$ that indicate that the formula $\phi$ is satisfied by $(\mathcal{P}, \mathcal{D}, \Sigma)$, $\Lambda$ and $\mathcal{V}$. The set $\mathcal{R}$ contains all configurations of the form,

- $\langle (p, \pi), \gamma \rangle$ where $\pi \in \Lambda(p)$,
- $\langle (p, \neg\pi), \gamma \rangle$ where $\pi \notin \Lambda(p)$,
- $\langle (p, X), \gamma \rangle$, where $X$ is free in $\phi$ and $\langle p, w \rangle \in \mathcal{V}(X)$.

If $\mathcal{V}(X)$ is regular for all $X$ free in $\phi$, then $\mathcal{R}$ is also regular.

Commands of the form $(_, a, push_a, _)$ are designed to deconstruct sub-formulae into literals that can be evaluated immediately. These commands require that the top order-one stack is not empty — otherwise play would be unable to proceed. Correctness of the construction

requires the top order-one stack to contain at least one stack symbol. This condition may be ensured with a special "bottom of the stack" symbol $\bot \in \Sigma$. This symbol marks the bottom of all order-one stacks and is never pushed or popped, except in the case of a command $(_, \bot, push_\bot, _)$. The use of such a symbol is common throughout the literature [53, 132, 127] etc..

**Proposition 6.3.1.** *Given the order-n PRG $A = (\mathcal{P}^{(\mathcal{P},\phi)}, \mathcal{D}^{\phi}_{\mathcal{P}}, \Sigma, \mathcal{R})$ constructed from the order-n PDS $(\mathcal{P}, \mathcal{D}, \Sigma)$, a labelling function $\Lambda$, a valuation $\mathcal{V}$ and a formula of the alternation-free $\mu$-calculus $\phi$ such that all $\sigma$-sub-formulae of $\phi$ are $\mu$-sub-formulae, we have $\langle p, \gamma \rangle \in [\![\phi]\!]_{\mathcal{V}}$ iff $\langle (p, \phi), \gamma \rangle \in Attr_E(\mathcal{R})$.*

*Proof.* The result follows from the fundamental theorem of the propositional $\mu$-calculus [117, 58]. If $\langle (p, \phi), \gamma \rangle \in Attr_E(\mathcal{R})$, then there is a winning strategy for Éloïse in $A$. In the absence of $\nu$-sub-formulae, this winning strategy defines a well-founded choice function and hence a well-founded pre-model for $(\mathcal{P}, \mathcal{D}, \Sigma)$, $\Lambda$, $\mathcal{V}$ and $\phi$ with initial state $\langle p, \gamma \rangle$. Thus, by the fundamental theorem, $\langle p, \gamma \rangle$ satisfies $\phi$.

In the opposite direction, if $\langle p, \gamma \rangle$ satisfies $\phi$, then — by the fundamental theorem — there is a well-founded pre-model with choice function $f$. Since there are no $\nu X.\psi$ sub-formula in $\phi$, all paths in the pre-model are finite and all leaves are of a form accepted by $\mathcal{R}$. Hence, a winning strategy for Éloïse is defined by $f$ and we have $\langle (p, \phi), \gamma \rangle \in Attr_E(\mathcal{R})$. ☐

In the dual case — when all $\sigma$-sub-formulae of $\phi$ are $\nu$-sub-formulae — we observe that the negation $\bar{\phi}$ of $\phi$ has only $\mu$-sub-formulae. We construct $Attr_E(\mathcal{R})$ for $\bar{\phi}$ and complement the resulting $n$-store multi-automaton (see Section 5.6.3) to construct the set of configurations satisfying $\phi$.

We are now ready to give a recursive algorithm for model-checking with the alternation-free $\mu$-calculus. We write $\Phi = \{\phi_i\}_{i=1}^{m}$ to denote a set of sub-formulae such that no $\phi_i$ is a sub-formula of another. Furthermore, we write $\phi[U/\Phi]$ where $U = \{U_i\}_{i=1}^{m}$ is a set of fresh variables to denote the simultaneous substitution in $\phi$ of $\phi_i$ with $U_i$ for all $i \in \{1, \ldots, m\}$. The following proposition is taken directly from [3]:

**Proposition 6.3.2.** *Let $\phi$ be a $\mu$-formula ($\nu$-formula) of the alternation-free $\mu$-calculus, and let $\Phi = \{\phi_i\}_{i=1}^{n}$ be the family of maximal $\nu$-sub-formulae ($\mu$-sub-formulae) of $\phi$ with respect to the sub-formula relation. Then,*

$$[\![\phi]\!]_{\mathcal{V}} = [\![\phi[U/\Phi]]\!]_{\mathcal{V}'}$$

*where $U = \{U_i\}_{i=1}^{n}$ is a suitable family of fresh variables, and $\mathcal{V}'$ is the valuation which extends $\mathcal{V}$ by assigning to each $U_i$ the set $[\![\phi_i]\!]_{\mathcal{V}}$.*

Since, given a $\mu$-formula ($\nu$-formula) $\phi$, the formula $\phi[U/\Phi]$ has only $\mu$-sub-formulae ($\nu$-sub-formulae) we can calculate $[\![\phi_i]\!]_{\mathcal{V}}$ for all $\phi_i \in \Phi$, using the above propositions to calculate an automaton recognising $[\![\phi]\!]_{\mathcal{V}}$.

**Theorem 6.3.1.** *Given an order-n PDS $(\mathcal{P}, \mathcal{D}, \Sigma)$, a labelling function $\Lambda$, a valuation function $\mathcal{V}$ and a formula $\phi$ of the alternation-free $\mu$-calculus, we can construct an n-store multi-automaton $A$ such that $\mathcal{L}(A) = [\![\phi]\!]_{\mathcal{V}}$.*

**Complexity**

A formula $\phi$ can be described as a tree structure with $\phi$ at the root. Each node in the tree is a $\mu$-sub-formula or a $\nu$-sub-formula $\psi$ of $\phi$. The children of the node are all maximal $\nu$-sub-formulae or $\mu$-sub-formulae of $\psi$ respectively. There are at most $n_\phi$ nodes in the tree, where $n_\phi$ is the length of $\phi$. Let $n_\mathcal{R}$ be the number of states in the $n$-store automaton recognising $\mathcal{R}$. The size of this automata is linear in the size of the automata specifying $\mathcal{V}$ for each variable $X$.

The $n$-store multi-automaton recognising $[\![\psi]\!]_\mathcal{V}$ for a leaf node $\psi$ has $\mathcal{O}(exp_n(n_\mathcal{R}))$ states. Together with a possible complementation step (which does not increase the state-set) we require $\mathcal{O}(exp_{n+1}(n_\mathcal{P} \cdot n_\phi))$ time and $\mathcal{B}$ may be of size $\mathcal{O}(exp_{n+1}(n_\mathcal{V}))$.

Similarly, the $n$-store multi-automaton recognising $[\![\psi]\!]_{\mathcal{V}'}$ for an internal node $\psi$ with children $\phi_1, \ldots, \phi_m$ has $\mathcal{O}(exp_n(\Sigma_{i=1}^m n_i + n_\mathcal{R}) \times exp_1(b_i))$ states, where $n_i$ is the size of the automaton recognising $[\![\phi_i]\!]_{\mathcal{V}_i}$ for $i \in \{1, \ldots, m\}$ and $b_i$ is the size of $\mathcal{B}$ for that automaton. Due to the final complementation step, $|\mathcal{B}|$ may be of size $\mathcal{O}(exp_{n+1}(\Sigma_{i=1}^m n_i + n_\mathcal{R}))$, which is also the total time required.

Subsequently, the automaton $A$ recognising $[\![\phi]\!]_{\mathcal{V}'}$ has $\mathcal{O}(exp_{n_\phi \cdot n}(n_\mathcal{R}))$ states and can be constructed in $\mathcal{O}(exp_{(n_\phi \cdot n)+1}(n_\mathcal{R}))$ time. Since we may test $c \in C$ for any configuration $c$ and set of configurations $C$ by checking $c \notin \overline{C}$, we may avoid the final complementation step to give us an $\mathcal{O}(exp_{n_\phi \cdot n}(n_\mathcal{R}))$ time algorithm.

## 6.4 Non-emptiness of Higher-Order Pushdown Automata

We show that the $n$-EXPTIME complexity of the algorithm is optimal. In fact, the backwards-reachability problem for order-$n$ PDSs is $n$-EXPTIME-complete. This result is widely regarded to follow from the work of Engelfriet [59]. However, because Engelfriet considers a broad range of automata, it is not immediately clear that his work can be applied directly to our own. We provide another proof of the result which uses a clearly stated theorem of Engelfriet: the non-emptiness problem for (non-deterministic) order-$n$ pushdown automata is $(n-1)$-EXPTIME-complete [59][1].

Walukiewicz and Cachat have provided another proof of this property [129]. Initially this proof was not published due to Engelfriet's result. The following proof was constructed before their paper was made available. The proof strategy is due to Olivier Serre.

We show that the reachability problem for order-$n$ APDS is $n$-EXPTIME-hard via a polynomial reduction from the non-emptiness problem over non-alternating order-$(n+1)$ pushdown automata. Let $\bot \in \Sigma$ be a dedicated "bottom of the stack" symbol that is neither popped from nor pushed onto the stack. We define $\bot_1 = [\bot]$ and $\bot_n = [\bot_{n-1}]$. In this case, the initial configuration of the automaton is of the form $\langle p_0, \bot_n \rangle$ for some $p_0$.

We define the order-$n$ PRG $P_G$ which can be used to determine whether the order-$(n+1)$ pushdown automaton $P$ is non-empty. Note that the construction is polynomial in size. Also, observe that commands of the form $(p, a, push_a, p')$ leave the stack contents unchanged.

[1]Completeness follows from Theorem 2.6, which states that 2N(multi)-$P^k$ = $(k-1)$-EXPTIME, and Theorem 7.11, which can be instantiated to show that one-way non-deterministic order-$k$ pushdown automata are log-space complete in 2N(multi)-$P^k$.

**Definition 6.4.1.** Given an order-$(n+1)$ pushdown automaton $P = (\mathcal{P}, \mathcal{D}, \Sigma, \Gamma, p_0, \mathcal{P}_f)$ we define the order-$n$ PRG $P_G = (\mathcal{P}', \mathcal{D}', \Sigma, \mathcal{R})$ where,

$$\begin{array}{rcl} \mathcal{P}' & = & \mathcal{P}'_E \cup \mathcal{P}'_A \\ \mathcal{P}'_E & = & \mathcal{P} \times (\mathcal{P} \cup \{\otimes\}) \ \cup \ \{\texttt{ff}\} \\ \mathcal{P}'_A & = & \mathcal{P} \times (\mathcal{P} \cup \{\otimes\}) \times (\mathcal{P} \cup \{\otimes\}) \ \cup \ \{\texttt{tt}\} \end{array}$$

with $\texttt{tt}, \texttt{ff}, \otimes \notin \mathcal{P}$. Furthermore,

$$\mathcal{R} = (\mathcal{P}_f \times \{\otimes\} \times C_n^{\Sigma})$$

Finally,

$$\begin{array}{rcl} \mathcal{D}' & = & \{ \ ((p, p_r), a, o, (p', p_r)) \mid (p, _, a, o, p') \in \mathcal{D} \ \wedge \ o \in \mathcal{O}_n \ \} \ \cup \\ & & \{ \ ((p, p_r), a, push_a, \texttt{tt}) \mid (p, _, a, pop_{n+1}, p_r) \in \mathcal{D} \ \} \ \cup \\ & & \{ \ ((p, p_r), a, push_a, \texttt{ff}) \mid (p, _, a, pop_{n+1}, p') \in \mathcal{D} \ \wedge \ p \neq p_r \ \} \ \cup \\ & & \{ \ ((p, p_r), a, push_a, (p', p'_r, p_r)) \mid (p, _, a, push_{n+1}, p') \in \mathcal{D} \ \wedge \ p'_r \in (\mathcal{P} \cup \{\otimes\}) \ \} \ \cup \\ & & \{ \ ((p', p'_r, p_r), a, push_a, (p', p'_r)) \mid p' \in \mathcal{P} \ \wedge \ p'_r, p_r \in (\mathcal{P} \cup \{\otimes\}) \ \wedge \ a \in \Sigma \ \} \ \cup \\ & & \{ \ ((p', p'_r, p_r), a, push_a, (p'_r, p_r)) \mid p', p'_r \in \mathcal{P} \ \wedge \ p_r \in (\mathcal{P} \cup \{\otimes\}) \ \wedge \ a \in \Sigma \ \} \end{array}$$

Intuitively, $P_G$ is defined to directly simulate any order-$n$ moves of the pushdown automaton. When a $push_{n+1}$ move is to be played, Eloise is required to give a control state $p_r$ which she claims play will be returned to when the top store added by the $push_{n+1}$ command is removed. If the top store will never be removed, she is able to play $p_r = \otimes$. Abelard then has a choice: either he can accept this assertion and let play continue from $p_r$ with the current store contents, or he can challenge it. If he challenges Eloise's claim, play moves to the control state specified by the $push_{n+1}$ command. From this configuration, Eloise is required to move play to the state $p_r$ as the current top store is $pop_{n+1}$-ed. If she succeeds, she wins the game, otherwise the play is a victory for Abelard.

**Property 6.4.1.** *If $\mathcal{L}(P) \neq \emptyset$ then $\langle (p_0, \otimes), \perp_n \rangle \in Attr_E(\mathcal{R})$.*

*Proof.* Assume there exists $w = \alpha_1 \ldots \alpha_m \in \mathcal{L}(P)$. Since $w \in \mathcal{L}(P)$ there exists a run,

$$\langle p_0, \gamma_0 \rangle \stackrel{\alpha_1}{\hookrightarrow} \ldots \stackrel{\alpha_m}{\hookrightarrow} \langle p_m, \gamma_m \rangle$$

with $\gamma_0 = \perp_{n+1}$, $p_m \in \mathcal{P}_f$ and the corresponding sequence of commands $d_1, \ldots, d_m$ such that for each $1 \leq i \leq m$ we have $d_i = (p_{i-1}, \alpha_i, a_i, o_i, p_i)$ with $top_i(\gamma_{i-1}) = a_i$ and $\gamma_i = o_i(\gamma_{i-1})$.

We show by induction over the number of order-$(n+1)$ commands in the sequence $d_1, \ldots, d_m$ that Eloise has a winning strategy in the game $P_G$. In particular we show that if there exists a run,

$$\langle p_0, \gamma_0 \rangle \stackrel{\alpha_1}{\hookrightarrow} \ldots \stackrel{\alpha_m}{\hookrightarrow} \langle p_m, \gamma_m \rangle$$

for arbitrary $p_0$ and $\gamma_0$ — with the further condition that if $o_i = pop_{n+1}$ for some $i$, then there exists $i' < i$ with $o_{i'} = push_{n+1}$ — then from a configuration $\langle (p_0, p_r), top_{n+1}(\gamma_0) \rangle$ of $P_G$ Eloise has a strategy to reach the configuration $\langle (p_m, p_r), top_{n+1}(\gamma_m) \rangle$ or win the game. For convenience we define the abbreviation $\gamma^n = top_{n+1}(\gamma)$ for all $\gamma$.

In the base case, no order-$(n+1)$ commands appear in the sequence $d_1, \ldots, d_m$. That is $o_i \in \mathcal{O}_n$ for all $1 \leq i \leq m$. Therefore, from the configuration $\langle (p_0, p_r), \gamma_0^n \rangle$ Eloise can play the

sequence of moves $d'_1, \ldots, d'_m$ where $d'_i = ((p_{i-1}, p_r), a_i, o_i, (p_i, p_r))$ and reach the configuration $\langle (p_m, p_r), \gamma^n_m \rangle$.

In the inductive case we have $o_i = push_{n+1}$ for some $i$ such that for all $i' < i$, $o_{i'} \in \mathcal{O}_n$. By induction Eloise can force play into the configuration $\langle (p_{i-1}, p_r), \gamma^n_{i-1} \rangle$ or win the game. In the former case there are two further cases to consider. Note that $\gamma^n_{i-1} = \gamma^n_i$.

- There is no $i' \geq i$ such that $\gamma_{i-1} = \gamma_{i'}$. That is, the top stack added by the push command is never removed. In this case Eloise moves play to $\langle (p_i, \otimes, p_r), \gamma^n_i \rangle$ via the command $d'_i = ((p_{i-1}, p_r), a, push_a, (p_i, \otimes, p_r))$ where $a = top_1(\gamma_{i-1})$. From this configuration Abelard may only move play to the configuration $\langle (p_i, \otimes), \gamma^n_i \rangle$. By induction, Eloise has a strategy as required from this configuration.

- There is some $i' \geq i$ such that $\gamma_{i-1} = \gamma_{i'}$. Take the least such $i'$. We have $o_{i'} = pop_{n+1}$. Eloise moves play to the configuration $\langle (p_i, p_{i'}, p_r), \gamma^n_i \rangle$ via the command $d'_i = ((p_{i-1}, p_r), a, push_a, (p_i, p_{i'}, p_r))$ where $a = top_1(\gamma_{i-1})$. There are two further subcases:
  - Abelard moves play to the configuration $\langle (p_i, p_{i'}), \gamma^n_i \rangle$. By induction, Eloise can either win from this configuration, or force play to $\langle (p_{i'-1}, p_{i'}), \gamma^n_{i'-1} \rangle$. In the latter case, since $o_{i'} = pop_{n+1}$, Eloise moves play to $\langle \texttt{tt}, \gamma^n_{i'-1} \rangle$ and wins the game.
  - Abelard moves play to the configuration $\langle (p_{i'}, p_r), \gamma^n_i \rangle = \langle (p_{i'}, p_r), \gamma^n_{i'} \rangle$. By induction Eloise can either win from this configuration or force play to the configuration $\langle (p_m, p_r), \gamma^n_m \rangle$ as required.

Since any run from $\langle p_0, \bot_{n+1} \rangle$ must perform a $push_{n+1}$ before a $pop_{n+1}$ can occur, it is the case that from $\langle (p_0, \otimes), \bot_{n+1} \rangle$ Eloise can either win the game or force play to the configuration $\langle (p_m, \otimes), \gamma^n_m \rangle$. Since $p_m \in \mathcal{P}_f$, this is a victory for Eloise. □

**Property 6.4.2.** *If* $\langle (p_0, \otimes), \bot_n \rangle \in Attr_E(\mathcal{R})$ *then* $\mathcal{L}(P) \neq \emptyset$.

*Proof.* Assume that Eloise has a winning strategy from the configuration $\langle (p_0, \otimes), \bot_n \rangle$ in the game $P_G$. We begin by proving that there is some bound $k$ on the length of any play of $P_G$ played according to Eloise's strategy. We then show that a word $w$ can be constructed such that $w \in \mathcal{L}(P)$.

To show there is some bound $k$ on the length of any play of $P_G$ we observe that if $\langle (p_0, \otimes), \bot_n \rangle \in Attr_E(\mathcal{R})$ then $\langle (p_0, \otimes), \bot_n \rangle \in Attr^i_E(\mathcal{R})$ for some $i$. We proceed by induction over $i$. We show for any $c \in Attr^i_E(\mathcal{R})$, there is a bound on the length of any play according to Eloise's winning strategy.

When $i = 0$ then $c \in \mathcal{R}$ and $k = 0$. For $i + 1$ then in both cases $c \in \mathcal{P}'_E$ or $c \in \mathcal{P}'_A$ play moves to a configuration $c' \in Attr^i_E(\mathcal{R})$. By induction there is some bound $k$ on the length of any play according to Eloise's strategy from $c'$. Hence, we have a bound $k + 1$ on the length of any play from $c$. Thus we have a bound on the length of any play from $\langle (p_0, \otimes), \bot_n \rangle$ as required.

We now show that there exists a word $w$ such that $w \in \mathcal{L}(P)$. We proceed by induction over $k$. In particular, we prove the following result: if Eloise has a winning strategy from a configuration $\langle (p, p_r), \gamma \rangle$, then there exists some $w \in \Gamma^*$ with, for all $\gamma'$ such that $top_{n+1}(\gamma') = \gamma$ and — if $p_r \neq \otimes$ — $pop_{n+1}(\gamma')$ is defined, a run of $P$ of the form $\langle p, \gamma' \rangle \xhookrightarrow{w} \langle p', \gamma'' \rangle$ where either $p' = p_r$, $\gamma'' = pop_{n+1}(\gamma')$ or $p' \in \mathcal{P}_f$ and $p_r = \otimes$.

In the base case $k = 0$. That is $p \in \mathcal{P}_f$ and $p_r = \otimes$. The property holds trivially. For $k + 1$ there are several cases depending on Eloise's next move $d$.

- $d = ((p, p_r), a, o, (p', p_r))$ and $o \in \mathcal{O}_n$.

  By definition of $\mathcal{D}'$, there exists some command $(p, \alpha, a, o, p') \in \mathcal{D}$. Hence, the transition $\langle p, \gamma' \rangle \overset{\alpha}{\hookrightarrow} \langle p', o(\gamma') \rangle$ exists in $P$.

  After Eloise moves, play continues from $\langle (p', p_r), o(\gamma) \rangle$ and $top_{n+1}(o(\gamma')) = o(\gamma)$. By induction over $k$ there exists $w' \in \Gamma^*$ such that $P$ has the run $\langle p', o(\gamma') \rangle \overset{w'}{\hookrightarrow} \langle p'', \gamma'' \rangle$ satisfying the induction hypothesis. Hence, the run $\langle p, \gamma' \rangle \overset{\alpha w'}{\longhookrightarrow} \langle p'', \gamma'' \rangle$ exists in $P$ and satisfies the induction hypothesis.

- $d = ((p, p_r), a, push_a, \mathtt{tt})$.

  By definition of $\mathcal{D}'$, there exists some command $(p, \alpha, a, pop_{n+1}, p_r) \in \mathcal{D}$. Hence, because $p_r \in \mathcal{P}$ and thus $p_r \neq \otimes$, the transition $\langle p, \gamma' \rangle \overset{\alpha}{\hookrightarrow} \langle p_r, \gamma'' \rangle$ exists in $P$ where $top_{n+1}(\gamma') = \gamma$ and $\gamma'' = pop_{n+1}(\gamma')$.

- $d = ((p, p_r), a, push_a, \mathtt{ff})$.

  In this case, play moves to the position $\langle \mathtt{ff}, \gamma \rangle$, which is a loss for Eloise. Since Eloise's strategy is winning, this case cannot occur.

- $d = ((p, p_r), a, push_a, (p', p'_r, p_r))$.

  By definition of $\mathcal{D}'$, there exists some command $(p, \alpha, a, push_{n+1}, p') \in \mathcal{D}$. Hence, the transition $\langle p, \gamma' \rangle \overset{\alpha}{\hookrightarrow} \langle p', push_{n+1}(\gamma') \rangle$ exists in $P$. Note that it is the case that $top_{n+1}(\gamma') = top_{n+1}(push_{n+1}(\gamma')) = \gamma$.

  Play moves to the configuration $\langle (p', p'_r, p_r), \gamma \rangle$. Eloise's strategy must accommodate both of Abelard's possible replies. In the case where Abelard moves play to the configuration $\langle (p', p_r), \gamma \rangle$, since Eloise's strategy is winning we have by induction $w' \in \Gamma'$ such that $\langle p', push_{n+1}(\gamma') \rangle \overset{w'}{\hookrightarrow} \langle p'', \gamma'' \rangle$ exists and satisfies the induction hypothesis.

  If $p'' \in \mathcal{P}_f$ and $p_r = \otimes$, then we have the run $\langle p, \gamma' \rangle \overset{\alpha w'}{\longhookrightarrow} \langle p'', \gamma'' \rangle$ and the induction hypothesis is satisfied.

  Otherwise we have a run $\langle p, \gamma' \rangle \overset{\alpha w'}{\longhookrightarrow} \langle p_r, \gamma' \rangle$. In the case where Abelard moves play to the configuration $\langle (p'_r, p_r), \gamma \rangle$, we have by induction a run $\langle p_r, \gamma' \rangle \overset{w''}{\hookrightarrow} \langle p''', \gamma''' \rangle$ for some $w''$ which satisfies the induction hypothesis. We consequently have a run $\langle p, \gamma' \rangle \overset{\alpha w' w''}{\longhookrightarrow} \langle p''', \gamma''' \rangle$ as required.

Thus, since Eloise has a winning strategy from the configuration $\langle (p_0, \otimes), \bot_n \rangle$, we have a run $\langle p_0, \bot_{n+1} \rangle \overset{w}{\hookrightarrow} \langle p, \gamma \rangle$ for some $w, p$ and $\gamma$. Since $\otimes \notin \mathcal{P}$, we must have $p \in \mathcal{P}_f$. Thus, $\mathcal{L}(P) \neq \emptyset$ as required. ☐

**Corollary 6.4.1.** *$\mathcal{L}(P) \neq \emptyset$ iff $\langle (p_0, \otimes), \bot_n \rangle \in Attr_E(\mathcal{R})$.*

**Corollary 6.4.2.** *The backwards reachability problem for order-$n$ APDSs is $n$-EXPTIME-hard.*

*Proof.* The non-emptiness problem for an order-$(n+1)$ pushdown automata is $n$-EXPTIME-hard [59]. By Definition 6.4.1 and Property 6.4.1 there is a polynomial reduction of this

problem to computing the Eloise's winning region in an order-$n$ pushdown game. By Theorem 6.2.1 there is a further polynomial reduction of this problem to backwards reachability over an order-$n$ alternating pushdown system. Hence, the backwards reachability problem for order-$n$ APDSs is $n$-EXPTIME-hard. □

## 6.5 Regular Goal Sets of Higher-Order Büchi Games

In this section we generalise pushdown Büchi games to higher-order PBGs and show how nested automata can be used to reduce regular sets of goal sets to set defined by control state only. A higher-order parity games algorithm can then be used to compute the winners of the game.

**Definition 6.5.1.** Given an order-$n$ PDS $(\mathcal{P}, \mathcal{D}, \Sigma)$, an **order-$n$ Pushdown Büchi Game** (PBG) $(\mathcal{P}, \mathcal{D}, \Sigma, \mathcal{R})$ is given by a partition $\mathcal{P} = \mathcal{P}_A \uplus \mathcal{P}_E$ and a set $\mathcal{R}$ of fair configurations.

Play proceeds as in the order-1 case. A play $c_0, c_1, c_2, \ldots$ is a win for Eloise iff there exists $i \geq 0$ such that $c_i \in \mathcal{C}_A$ and Abelard is unable to move, or for all $i \geq 0$, there exists $j \geq i$ such that $c_j \in \mathcal{R}$ — that is, the set of fair configurations $\mathcal{R}$ is visited infinitely often. Similarly to Section 6.2 we make a totality assumption. In the Büchi case, we set $\langle p^A_{lose}, \gamma \rangle \in \mathcal{R}$ for all $\gamma$.

We define $Attr^+_E(T)$ and $B\ddot{u}chi_E(\mathcal{R})$:

$$\begin{array}{rcl} X_0(T) & = & \emptyset \\ X_{i+1}(T) & = & X_i(T) \ \cup \ \{ \ c \in \mathcal{C}_E \mid \exists c'.c \hookrightarrow c' \ \wedge \ c' \in T \cup X_i(T) \ \} \\ & & \cup \ \{ \ c \in \mathcal{C}_A \mid \forall c'.c \hookrightarrow c' \Rightarrow c' \in T \cup X_i(T) \ \} \\ Attr^+_E(T) & = & \bigcup_{i \geq 0} X_i(T) \end{array}$$

$$\begin{array}{rcll} B\ddot{u}chi^0(\mathcal{R}) & = & C^\Sigma_n & \\ B\ddot{u}chi^{\alpha+1}_E(\mathcal{R}) & = & Attr^+_E(B\ddot{u}chi^\alpha_E(\mathcal{R}) \cap \mathcal{R}) & \text{for any ordinal } \alpha \\ B\ddot{u}chi^\lambda_E(\mathcal{R}) & = & \bigcap_{\alpha < \lambda} B\ddot{u}chi^\alpha_E(\mathcal{R}) & \text{for a limit ordinal } \lambda \end{array}$$

Current algorithms for computing the winner of the game assume the set $\mathcal{R}$ is of the form $\mathcal{R} = \mathcal{F} \times C^\Sigma_n$ for some $\mathcal{F} \subseteq \mathcal{P}$. That is, the set of fair configurations is defined by a set of control states. In the order-1 case, Cachat shows that this assumption can be made without loss of generality. In the next section, we show, using a different proof technique, that this result holds in the higher-order case.

A **simple goal set** $\mathcal{R}$ is of the form $\mathcal{F} \times C^\Sigma_n$ for some $\mathcal{F} \subseteq \mathcal{P}$. In this section we show that for every Büchi game, there is an equivalent game with a simple goal set.

We begin by showing how to construct a reachability game $G_\mathcal{R}$ such that Eloise can win from $\langle p, \gamma \rangle$ iff $\langle p, \gamma \rangle \in \mathcal{R}$. To encode a Büchi game as a game with a simple goal set Eloise declares after each move whether the current configuration is in $\mathcal{R}$. Abelard can choose to accept or challenge this claim. Eloise wins the game if Abelard accepts an infinite number of positive claims or play reaches an accepting state of $G_\mathcal{R}$ (this can be translated to a Büchi condition by adding a self-loop).

We first construct a reachability game $G_\mathcal{R}$ which Eloise can win from a configuration $\langle p, \gamma \rangle$ iff $\langle p, \gamma \rangle \in \mathcal{R}$. We assume that the bottom of every order-one stack is marked with the symbol $\perp$ that is neither pushed onto nor popped from the stack. Since no $n$-store is empty, every $n$-store (multi-)automaton can be assumed to have a single final state $q_f$ with no outgoing transitions. Furthermore, we assume, without loss of generality, that $\mathcal{R}$ is represented by a *non-deterministic* $n$-store multi-automaton.

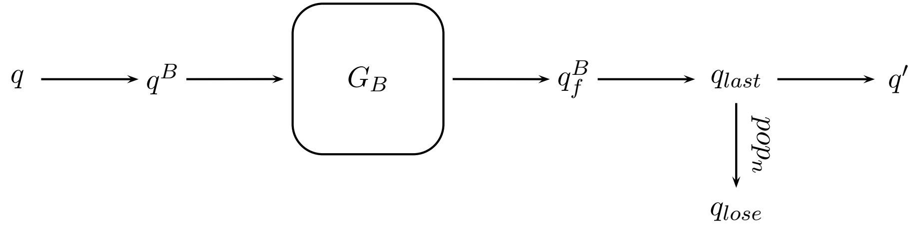



Figure 6.1: Encoding an $n$-store transition as a game

The idea behind the translation is that a transition $q \xrightarrow{B} q'$ is represented by a transition from the state $q$ into a sub-game representing the automaton $B$. Should this automaton accept the current top stack, play will return from the sub-game to the state $q'$, removing the top stack in the process.

The main subtlety of this approach is that, once a final state of $B$ has been reached, we must only exit the sub-game if the complete top stack has been checked. In the order-1 case, this can be easily checked using the $\perp$ symbol. In the higher-order case the bottom stack cannot be marked in this way. Instead, when Eloise wishes to exit the sub-game, she must pass control to Abelard. At this stage, Abelard can force play to a designated losing state (for Eloise) if a $pop_n$ action can be performed on the stack. In the case that the current stack is the bottom stack, the $pop_n$ operation is not defined, therefore Abelard must exit the sub-game as Eloise requested. Figure 6.1 illustrates the construction.

**Definition 6.5.2.** Let $B = (\mathcal{Q}, \Sigma, \Delta, \{q^1, \ldots, q^z\}, \{q_f\})$ be a non-deterministic $n$-store (multi-)automaton accepting $n$-stores in $\mathcal{R}$. We define $G_{\mathcal{R}} = G_B$. $G_B$ is defined recursively.

When $n = 1$, $G_B = (\mathcal{Q}_B, \Sigma, \mathcal{D}_B, \{q^B_f\} \times C^\Sigma_n)$ where $\mathcal{Q}_B = \mathcal{Q}$, $q^B_f = q_f$, and,

$$\begin{array}{rcl}\mathcal{D}_B & = & \{\ (q, a, pop_1, q') \mid (q, a, q') \in \Delta \ \text{ and } q' \neq q_f\ \} \cup \\ & & \{\ (q, \perp, push_\perp, q^B_f) \mid (q, \perp, q_f) \in \Delta\ \}\end{array}$$

When $n > 1$, $G_B = (\mathcal{Q}_B, \mathcal{D}_B, \Sigma, \{q^B_f\} \times C^\Sigma_n)$,

$$\mathcal{Q}_B = \mathcal{Q} \uplus \{q_{last}, q_{lose}\} \uplus \biguplus_{(q,B',q') \in \Delta} \mathcal{Q}_{B'}$$

and (where $q^{B'}$ and $q^{B'}_f$ are the initial state and finial state respectively of an automaton $B'$),

$$\begin{array}{rcl}\mathcal{D}_B & = & \{\ (q, a, push_a, q^{B'}) \mid (q, B', q') \in \Delta,\ q' \neq q_f \text{ and } a \in \Sigma\ \} \cup \\ & & \{\ (q^{B'}_f, \perp, pop_n, q') \mid (q, B', q') \in \Delta\ \} \cup \\ & & \{\ (q^{B'}_f, \perp, push_\perp, q_{last}) \mid (q, B', q_f) \in \Delta\ \} \cup \\ & & \{\ (q_{last}, a, pop_n, q_{lose}) \mid a \in \Sigma\ \} \cup \\ & & \{\ (q_{last}, a, push_a, q^B_f) \mid a \in \Sigma\ \} \cup \\ & & \bigcup_{(q,B',q') \in \Delta} D_{B'}\end{array}$$

In all cases, all control states except $q_{last}$ belong to Eloise and we assume $q_{last}$ and $q_{lose}$ are fresh states.

**Property 6.5.1.** $\langle p, \gamma \rangle \in \mathcal{L}(B) \iff \langle p, \gamma \rangle \in Attr_E(Q^B_f \times C^\Sigma_n)$.

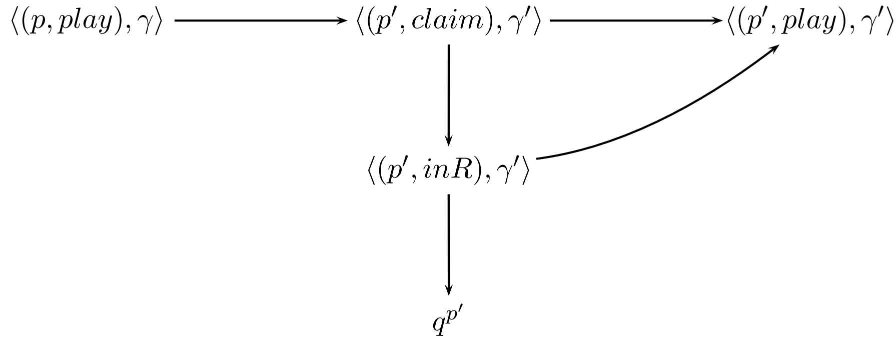



Figure 6.2: Reduction to a game with simple goal sets

*Proof.* We induct over $n$. In the base case $n = 1$.

Let $\gamma = [w_\gamma \bot]$. We show by induction over the length of $w_\gamma$ that $w_\gamma \bot$ is accepted from $q$ iff Eloise wins the reachability game from $\langle q, [w_\gamma \bot] \rangle$. In the base case $w_\gamma = \varepsilon$ and the result is trivial. Let $w_\gamma = aw'$. By induction, $w' \bot$ is accepted from $q'$ iff $\langle q', [w' \bot] \rangle$ is winning for Eloise. By induction and since $(q, a, q') \in \Delta$ iff $(q, a, pop_1, q') \in \mathcal{D}$, it follows that $q \xrightarrow{a} q' \xrightarrow{w'\bot} q_f$ iff $\langle q, [aw'\bot] \rangle \overset{*}{\hookrightarrow} \langle q', [w'\bot] \rangle$.

When $n > 1$, let $\gamma = [w_\gamma]$. We show by induction over the length of $w_\gamma$ that $q \xrightarrow{w_\gamma} q_f$ iff Eloise can win from $\langle q, [w_\gamma] \rangle$. In the base case $w_\gamma = [w]$ where $w \in C^\Sigma_{n-1}$. For every transition $q \xrightarrow{B} q_f$ we have that Eloise can force $\langle q^B, [[w]] \rangle$ to $\langle q^B_f, [[w']] \rangle$ with $top_1(w') = \bot$ iff $[w] \in \mathcal{L}(B)$. Therefore, we have $q \xrightarrow{[w]} q_f$ iff we have $\langle q, [[w]] \rangle \overset{*}{\hookrightarrow} \langle q^B, [[w]] \rangle \overset{*}{\hookrightarrow} \langle q^B_f, [[w']] \rangle \overset{*}{\hookrightarrow} \langle q_{last}, [[w']] \rangle$. Furthermore, since $pop_n$ cannot be performed, Abelard must move play to $\langle q^B_f, [[w']] \rangle$.

Let $w_\gamma = [w]w'$. By induction over the length, $q \xrightarrow{w'} q_f$ iff $\langle q, [w'] \rangle \overset{*}{\hookrightarrow} \langle q^B_f, [w''] \rangle$ with $top_1(w'') = \bot$. Since $(q, B', q') \in \Delta$ iff $(q, a, push_a, q^{B'}) \in \mathcal{D}$ it follows by induction on $n$ that $q \xrightarrow{[w]} q'$ iff Eloise can force $\langle q, [[w]w'] \rangle \overset{*}{\hookrightarrow} \langle q^{B'}, [[w]w'] \rangle \overset{*}{\hookrightarrow} \langle q^{B'}_f, [[w'']w'] \rangle$.

If Eloise plays $(q^{B'}_f, \bot, push_\bot, q_{last})$, Abelard will play $(q_{last}, a, pop_n, q_{lose})$ and Eloise will lose the game. Hence, she must play $(q^{B'}_f, \bot, pop_n, q')$ to $\langle q', [w'] \rangle$, from which she can win by induction over the length of the $w'$ iff $q' \xrightarrow{w'} q_f$. Hence $q \xrightarrow{[w]w'} q_f$ iff Eloise can force play from $\langle q, [w]w' \rangle$ to $\langle q^B_f, [w''] \rangle$ with $top_1(w'') = \bot$. □

For a Büchi game $G$, we are now ready to construct an equivalent game $G_S$ with a simple goal set. Intuitively, after each game move Eloise is asked to state whether the current configuration is in $\mathcal{R}$. If a positive claim is made, Abelard can challenge the claim by moving play to $G_\mathcal{R}$ (as in Definition 6.5.2) or let play continue in $G$. If Eloise can make an infinite number of positive claims without a successful challenge from Abelard, then it follows that the set $\mathcal{R}$ was seen an infinite number of times. The moves of the game for a move $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ are illustrated in Figure 6.2.

**Definition 6.5.3.** Given a Büchi game $G = (\mathcal{P}, \mathcal{D}, \Sigma, \mathcal{R})$ and $G_\mathcal{R} = (\mathcal{P}^\mathcal{R}, \mathcal{D}_\mathcal{R}, \Sigma, \{q_f\} \times C^\Sigma_n)$ constructed according to Definition 6.5.2, we define $G_S = (\mathcal{P}^S, \mathcal{D}_S, \Sigma, \mathcal{R}_S)$ where (letting

$q^p = q^j$ be the (initial) control state of $G_{\mathcal{R}}$ corresponding to the control state $p = p^j$ of $G$),

$$\begin{array}{rcl}
\mathcal{P}^S_A & = & (\mathcal{P}_A \times \{play\}) \ \cup \ (\mathcal{P} \times \{inR\}) \ \cup \ \mathcal{P}^{\mathcal{R}}_A \\
\mathcal{P}^S_E & = & (\mathcal{P}_E \times \{play\}) \ \cup \ (\mathcal{P} \times \{claim\}) \ \cup \ \mathcal{P}^{\mathcal{R}}_E \\
\mathcal{D}_S & = & \{ \ ((p, play), a, o, (p', claim)) \mid (p, a, o, p') \in \mathcal{D} \ \} \cup \\
& & \{ \ ((p, claim), a, push_a, (p, play)) \mid a \in \Sigma \ \} \cup \\
& & \{ \ ((p, claim), a, push_a, (p, inR)) \mid a \in \Sigma \ \} \cup \\
& & \{ \ ((p, inR), a, push_a, (p, play)) \mid a \in \Sigma \ \} \cup \\
& & \{ \ ((p, inR), a, push_a, q^p) \mid a \in \Sigma \ \} \cup \\
& & \{ \ (q_f, a, push_a, q_f) \mid a \in \Sigma\} \cup \\
& & \mathcal{D}_{\mathcal{R}} \\
\mathcal{R}_S & = & ((\mathcal{P} \times \{inR\}) \ \cup \ \{q_f\}) \times C^{\Sigma}_n
\end{array}$$

**Property 6.5.2.** *Given a Büchi game $G$ with winning region (for Eloise) $Büchi_E(\mathcal{R})$ and associated reachability game $G_{\mathcal{R}}$ with winning region $Attr_E(\{q_f\} \times C^{\Sigma}_n)$, let*

$$\begin{array}{rcl}
winS & = & Attr_E(\{q_f\} \times C^{\Sigma}_n) \cup \\
& & \{ \ \langle (p, play), \gamma \rangle \mid \langle p, \gamma \rangle \in Büchi_E(\mathcal{R}) \ \} \cup \\
& & \{ \ \langle (p, claim), \gamma \rangle \mid \langle p, \gamma \rangle \in Büchi_E(\mathcal{R}) \ \} \cup \\
& & \{ \ \langle (p, inR), \gamma \rangle \mid \langle p, \gamma \rangle \in Büchi_E(\mathcal{R}) \ \cap \ \mathcal{R} \ \}
\end{array}$$

*It is the case that $winS$ is Eloise's winning region in $G_S$.*

*Proof.* We show that $winS$ is the greatest fixed point of $X \mapsto Attr^+_E(X \cap \mathcal{R}_S)$.

*$winS$ is a fixed point:* to show $winS \subseteq Attr^+_E(winS \cap \mathcal{R}_S)$ we take $c \in winS$. There are several cases,

- Case $c = \langle p, \gamma \rangle \in Attr_E(\{q_f\} \times C^{\Sigma}_n)$: if $p = q_f$ and $top_1(\gamma) = a$ then Eloise can play $(q_f, a, push_a, q_f)$ to $c \in winS \ \cap \ \mathcal{R}_S$ and hence $c \in Attr^+_E(winS \ \cap \ \mathcal{R}_S)$. Otherwise, since $c \in Attr_E(\{q_f\} \times C^{\Sigma}_n)$ we have $c \in Attr^+_E(winS \ \cap \ \mathcal{R}_S)$ immediately.

- Case $c = \langle (p, play), \gamma \rangle$. Since $\langle p, \gamma \rangle \in Büchi_E(\mathcal{R})$ there exists (if $p \in \mathcal{P}_E$) a $(p, a, o, p') \in \mathcal{D}$ with — or (if $p \in \mathcal{P}_A$) for all $(p, a, o, p') \in \mathcal{D}$ it is the case that — $\langle p', o(\gamma) \rangle \in Attr^*_E(Büchi_E(\mathcal{R}) \cap \mathcal{R})$. We proceed by induction over the number of moves required for Eloise to reach $Büchi_E(\mathcal{R}) \cap \mathcal{R}$. In the base case, no moves are required. That is $\langle p', o(\gamma) \rangle \in Büchi_E(\mathcal{R}) \cap \mathcal{R}$. Therefore $\langle (p', inR), o(\gamma) \rangle \in winS \cap \mathcal{R}_S$. Thus, since Eloise can force play from $\langle (p', claim), o(\gamma) \rangle$ to $\langle (p', inR), o(\gamma) \rangle$, we have $(p', claim), o(\gamma) \rangle \in Attr^+_E(winS \cap \mathcal{R}_S)$.

  In the inductive case, Eloise is able to force play to $\langle p'', \gamma'' \rangle \in Attr_E(Büchi_E(\mathcal{R}) \cap \mathcal{R})$ in one move. Therefore, in $G_S$, Eloise is able to force play from $\langle (p', play), o(\gamma) \rangle$ to $\langle (p'', claim), \gamma'' \rangle$. By induction over the distance to $Büchi_E(\mathcal{R}) \cap \mathcal{R}$ we have that $\langle (p'', claim), \gamma'' \rangle \in Attr^+_E(winS \cap \mathcal{R}_S)$. Since Eloise can force play from $\langle (p', claim), o(\gamma) \rangle$ to $\langle (p', play), o(\gamma) \rangle$ and then to $\langle (p'', claim), \gamma'' \rangle$, we have that $\langle (p', claim), o(\gamma) \rangle \in Attr^+_E(winS \cap \mathcal{R}_S)$.

  Since Eloise can force play to $\langle (p', claim), o(\gamma) \rangle$, it follows that we have $\langle (p, play), \gamma \rangle, \in Attr^+_E(winS \cap \mathcal{R}_S)$.

- Case $c = \langle (p, inR), \gamma \rangle$: since $c \in winS$ it must be the case that $\langle p, \gamma \rangle \in Büchi_E(\mathcal{R}) \cap \mathcal{R}$. Abelard can move play to $\langle (p, play), \gamma \rangle$ or $\langle q^p, \gamma \rangle$. Both of these configurations have been shown in the previous cases to be in $Attr^+_E(winS \cap \mathcal{R}_S)$, hence $c$ is also.

- Case $c = \langle (p, claim), \gamma \rangle$: since $c \in winS$ we know $\langle p, \gamma \rangle \in Büchi_E(\mathcal{R})$. There are two cases. If $\langle p, \gamma \rangle \in \mathcal{R}$, then $\langle (p, inR), \gamma \rangle \in winS$, and, by the previous case $\langle (p, inR), \gamma \rangle \in Attr^+_E(winS \cap \mathcal{R}_S)$. Since Eloise can always move to this configuration, we have $\langle p, \gamma \rangle \in Attr^+_E(winS \cap \mathcal{R}_S)$.

  In the other case, $\langle p, \gamma \rangle \notin \mathcal{R}$, since $\langle p, \gamma \rangle \in Büchi_E(\mathcal{R})$, we have $\langle (p, play), \gamma \rangle \in winS$ and by the previous case $\langle (p, play), \gamma \rangle \in winS \in Attr^+_E(winS \cap \mathcal{R}_S)$. Since Eloise can force play to this position, we also have $\langle p, \gamma \rangle \in Attr^+_E(winS \cap \mathcal{R}_S)$.

We now show $Attr^+_E(winS \cap \mathcal{R}_S) \subseteq winS$. Take $c \in Attr^+_E(winS \cap \mathcal{R})$. If no moves are possible from this configuration it must belong to Abelard and be in $Attr_E(\{q_f\} \times C^\Sigma_n) \subseteq winS$ or of the form $\langle (p, play), \gamma \rangle$. In the latter case Abelard will have no available moves from $\langle p, \gamma \rangle$ in $G$ and hence $\langle p, \gamma \rangle \in Büchi_E(\mathcal{R})$ and $\langle (p, play), \gamma \rangle \in winS$.

Otherwise, we induct over the number of steps required to reach $(winS \cap \mathcal{R}_S)$. In the base case we have that Eloise can force play to a configuration $c' \in winS \cap \mathcal{R}_S$ in one move. There are two cases. If $c' = \langle q_f, \gamma \rangle$ then $c \in Attr_E(\{q_f\} \times C^\Sigma_n) \subseteq winS$. Otherwise $c' = \langle (p, inR), \gamma \rangle$, $\langle p, \gamma \rangle \in Büchi_E(\mathcal{R}) \cap \mathcal{R}$ and $c = \langle (p, claim), \gamma \rangle$. Hence $c \in winS$.

In the inductive case Eloise can force play to some $c'$ with $c' \in win$ by induction. We consider the format of $c$:

- $c = \langle q, \gamma \rangle$: In this case $c$ and hence $c'$ are in $Attr_E(\{q_f\} \times C^\Sigma_n) \subseteq winS$.

- $c = \langle (p, play), \gamma \rangle$: $c' = \langle (p', claim), \gamma' \rangle$ and since $c' \in winS$ we have that $\langle p', \gamma' \rangle \in Büchi_E(\mathcal{R})$. Since Eloise can force play to $c'$ it follows that Eloise can force play to $\langle p', \gamma' \rangle$ from $\langle p, \gamma \rangle$ in $G$, hence $\langle p, \gamma \rangle \in Büchi_E(\mathcal{R})$ and $c \in winS$.

- $c = \langle (p, claim), \gamma \rangle$: In this case $c' = \langle (p, play), \gamma \rangle$ or $\langle (p, inR), \gamma \rangle$. Both cases imply $\langle p, \gamma \rangle \in Büchi_E(\mathcal{R})$ (since $c' \in winS$) and hence $c \in winS$.

- $c = \langle (p, inR), \gamma \rangle$: Abelard can force play to both $c' = \langle (p, play), \gamma \rangle$ and $c' = \langle q^p, \gamma \rangle$. Hence $\langle p, \gamma \rangle \in Büchi_E(\mathcal{R}) \cap \mathcal{R}$. Thus, $c \in winS$.

Thus, we have that $winS$ is a fixed point of $X \mapsto Attr^+_E(X \cap \mathcal{R}_S)$. We now show that $winS$ is the greatest such fixed point.

*winS is the greatest fixed point:* assume $Y_S \supset winS$ is a fixed point of $X \mapsto Attr^+_E(X \cap \mathcal{R}_S)$. There cannot be some configuration $\langle q, \gamma \rangle \in Attr_E(\{q_f\} \times C^\Sigma_n)$ in $Y_S$ but not in $winS$ since $winS \supseteq Attr_E(\{q_f\} \times C^\Sigma_n)$. Furthermore, there cannot be some configuration $\langle (p, inR), \gamma \rangle$ in $Y_S$ with $\langle p, \gamma \rangle \notin \mathcal{R}$. If there were, Abelard could simply move play to $\langle q^p, \gamma \rangle \notin Attr^+_E(Y_S \cap \mathcal{R}_S)$. Hence, all $c$ in $Y_S$ but not $winS$ must be of the form $\langle (p, play), \gamma \rangle$ or $\langle (p, claim), \gamma \rangle$, or of the form $\langle (p, inR), \gamma \rangle$ with $\langle p, \gamma \rangle \in \mathcal{R}$. Furthermore, whenever $\langle (p, inR), \gamma \rangle$ is in $Y_S$ then $\langle (p, play), \gamma \rangle$ is since Abelard can always move to this configuration. Finally, whenever $\langle (p, claim), \gamma \rangle$ is in $Y_S$, then Eloise's available moves imply that either $\langle (p, inR), \gamma \rangle$ and hence $\langle (p, play), \gamma \rangle$ is in $Y_S$ or $\langle (p, play), \gamma \rangle$ is in $Y_S$ directly.

We define,

$$Y \quad = \quad \{ \ \langle p, \gamma \rangle \mid \langle (p, play), \gamma \rangle \in Y_S \ \}$$

From the above, we have $Y \supset Büchi_E(\mathcal{R})$. We show that if $Y_S$ is a fixed point of $X \mapsto Attr^+_E(X \cap \mathcal{R}_S)$, then $Y$ is a fixed point of $X \mapsto Attr^+_E(X \cap \mathcal{R})$ in $G$. This is a contradiction since $Büchi_E(\mathcal{R})$ is the greatest such fixed point.

We show $Y \subseteq Attr^+_E(Y \cap \mathcal{R})$. Take $\langle p, \gamma \rangle \in Y$. If no moves are available from $\langle p, \gamma \rangle$, then no moves are available from $\langle (p, play), \gamma \rangle$ in $G_S$. Since $\langle (p, play), \gamma \rangle \in Attr^+_E(Y_S \cap \mathcal{R}_S)$, it follows that $p \in \mathcal{P}_A$. Hence $\langle p, \gamma \rangle \in Attr^+_E(Y \cap \mathcal{R})$.

Otherwise we induct over the number of moves from $\langle (p, play), \gamma \rangle$ to $Y_S \cap \mathcal{R}_S$ in $G_S$. The base case is two moves $\langle (p, play), \gamma \rangle \hookrightarrow \langle (p', claim), \gamma' \rangle \hookrightarrow \langle (p', inR), \gamma' \rangle$. From the fact that $\langle (p', inR), \gamma' \rangle \in Y_S \cap \mathcal{R}_S$ we have $\langle p', \gamma' \rangle \in \mathcal{R}$ and $\langle p', \gamma' \rangle \in Y \cap \mathcal{R}$. Since Eloise can force play to $\langle (p', claim), \gamma' \rangle$ in $G_S$ it follows that Eloise can force play to $\langle p', \gamma' \rangle$ in $G$. Hence $\langle p, \gamma \rangle \in Attr^+_E(Y \cap \mathcal{R})$.

In the inductive case Eloise can force play to $\langle (p', claim), \gamma' \rangle \in Attr^+_E(Y_S \cap \mathcal{R}_S)$. If Eloise moves play directly to $\langle (p', play), \gamma' \rangle$ then by induction we have $\langle p', \gamma' \rangle$ and hence $\langle p, \gamma \rangle$ in $Attr^+_E(Y \cap \mathcal{R})$. If Eloise moves play to $\langle (p', inR), \gamma' \rangle$ then since Abelard can move play to $\langle (p', play), \gamma' \rangle$ and by induction we have $\langle p', \gamma' \rangle$ and hence $\langle p, \gamma \rangle$ in $Attr^+_E(Y \cap \mathcal{R})$.

Finally, we show $Attr^+_E(Y \cap \mathcal{R}) \subseteq Y$. Take $\langle p, \gamma \rangle \in Attr^+_E(Y \cap \mathcal{R})$. If no moves are available from $\langle p, \gamma \rangle$ it follows that $p \in \mathcal{P}_A$ and no moves are available from $\langle (p, play), \gamma \rangle$ in $G_S$. Hence $\langle (p, play), \gamma \rangle \in Y_S$. Thus, $\langle p, \gamma \rangle \in Y$.

Otherwise we induct over the number of moves required to reach $Y \cap \mathcal{R}$. In the base case Eloise can force play to $\langle p', \gamma' \rangle \in Y \cap \mathcal{R}$ in one move. By definition of $Y$ we have $\langle (p', play), \gamma' \rangle$ in $Y_S$ and hence (since $Y_S = Attr^+_E(Y_S \cap \mathcal{R}_S)$) we have $\langle (p', inR), \gamma' \rangle \in Y_S \cap \mathcal{R}_S$ and $\langle (p', claim), \gamma' \rangle \in Y_S$. Since Eloise can force play from $\langle p, \gamma \rangle$ to $\langle p', \gamma' \rangle$ in $G$, it follows that Eloise can force play from $\langle (p, play), \gamma \rangle$ to $\langle (p', claim), \gamma' \rangle$ in $G_S$. Hence $\langle (p, play), \gamma \rangle \in Y_S$ and $\langle p, \gamma \rangle \in Y$.

In the inductive case Eloise can force play to $\langle p', \gamma' \rangle \in Attr^+_E(Y \cap \mathcal{R})$ and by induction $\langle p', \gamma' \rangle \in Y$. Hence $\langle (p', play), \gamma' \rangle \in Y_S$. From $\langle (p', claim), \gamma' \rangle$ Eloise can move to $\langle (p', play), \gamma' \rangle$, hence $\langle (p', claim), \gamma' \rangle$ is in $Y_S$. Since Eloise can force play from $\langle p, \gamma \rangle$ to $\langle p', \gamma' \rangle$ in $G$, it follows that Eloise can force play from $\langle (p, play), \gamma \rangle$ to $\langle (p', claim), \gamma' \rangle$ in $G_S$. Hence $\langle (p, play), \gamma \rangle \in Y_S$ and $\langle p, \gamma \rangle \in Y$.

Therefore, if $Y_S$ is a greater fixed point in $G_S$ than $winS$, then $Y$ is a greater fixed point in $G$ than $Büchi_E(\mathcal{R})$, which is a contradiction as required. □

## 6.6 Summary

In this chapter we provided a number of applications of our backwards reachability result to LTL and branching-time model-checking, reachability games, Büchi games and the non-emptiness of higher-order pushdown automata.

# Chapter 7

# Conclusion

We are now ready to conclude. We begin with a summary of the work presented before discussing possible avenues of future research.

## 7.1 Summary of Contributions

We have provided several new model-checking algorithms for pushdown systems. In Chapter 5 we described a backwards reachability algorithm for alternating higher-order pushdown systems and showed how it can be used to calculate the winning regions of higher-order pushdown games played over such systems. In Chapter 6 we gave a number of applications of this work. Finally, in Chapter 4 we gave an novel algorithm for computing the winning regions of pushdown parity games which is more direct that previous approaches, and is not immediately exponential.

#### Parity Conditions

The first contribution of the thesis is what we believe to be the first extension of saturation techniques to parity games played over pushdown systems. Previous methods, due independently to Cachat [127] and Serre [92], computed winning regions in the order-1 case using Walukiewicz's local model-checking algorithm [53] as an oracle. These techniques have the disadvantage of having to construct a large (immediately exponential), finite parity game for the local model-checking algorithm. Another approach, due to Vardi *et al.* [90, 86] uses two-way alternating parity tree-automata, a reduction to one-way alternating parity tree automata, and then the construction of a Büchi automaton requiring a complementation step. In contrast, our saturation algorithm constructs an automaton accepting Éloïse's winning region directly by expanding a small initial automaton. Consequently, a large automaton will only be constructed if necessary.

Our technique used a characterisation, due to Walukiewicz [53], of Éloïse's winning region as a series of fixed points of a kind of reachability formula. We were able to adapt the order-1 reachability algorithm and Cachat's Büchi algorithm to compute greatest fixed points, and, with a slight variation, least fixed points. These three components were combined to give an algorithm for computing Éloïse's winning region of an order-1 pushdown parity game.

### Reachability Conditions

Bouajjani, Esparza and Maler [3, 8] and Finkel, Willems and Wolper [15] first introduced saturation methods for model-checking order-1 pushdown systems. They provided a symbolic backwards reachability algorithm. Given a regular set of pushdown configurations $C$ — represented using a finite multi-automaton — their algorithm computed the set of configurations that could reach $C$ in a finite number of moves. The computation is a fixed point calculation: during each iteration, transitions are added to the initial automaton, which has the effect of adding new configurations that can reach $C$. Eventually, no more new transitions can be added to the automaton, and we are left with the set $Pre^*(C)$.

In 2007, Bouajjani and Meyer extended this technique to the case of higher-order pushdown systems with a single control-state [2]. Their approach introduced nested automata which mimicked the nested structure of higher-order pushdown stores. Transitions of these automata were labelled by nested automata that were ultimately labelled by finite-word automata. During iteration, updates filtered through the nested structure. A slightly modified saturation argument ensured termination.

In Chapter 5 we provided an extension of these algorithms to the general case of alternating higher-order pushdown systems. To ensure termination, rather than updating the automata labelling transitions directly, we labelled new transitions with a recipe for the construction of the required automata. This allowed us to identify when updates become repetitive, and hence, we were able to identify a cascade of fixed points, resulting in termination.

### Applications

In Chapter 6 we showed that an order-$n$ reachability game can be encoded as an order-$n$ APDS, and hence, we can compute the winning regions of such a game. We also showed that the $n$-EXPTIME complexity of our reachability algorithm is optimal. The proof is a reduction from the non-emptiness problem for order-$(n+1)$ pushdown automata and is based on techniques suggested by Olivier Serre.

In this chapter, we also generalised results due to Bouajjani *et al.* [3] that show how to use the backwards reachability algorithm to perform LTL model-checking and branching-time model-checking for the alternation-free $\mu$-calculus. Finally, we presented a new reduction from regular goal sets to simple goal sets for order-$n$ pushdown Büchi games, using a game simulation, rather than the product construction used by Cachat in the order-1 case.

## 7.2 Further Research

We have presented algorithms for several model-checking problems for higher-order pushdown systems. This work suggests a number of possibilities for future research.

### Symmetric Higher-Order Pushdown Systems

An alternative definition of higher-order pushdown systems defines the higher-order pop operation as the inverse of the push operation. That is, a stack may only be popped if it matches the stack below. The results of Carayol [11] imply that the set $Pre^*(C_{Init})$ over these structures is regular, using Carayol's notion of regularity. However, the complexity of computing this set is unknown. We may attempt to adapt our algorithm to this setting and prove the required complexity bounds.

### Winning Strategies

**Winning strategies** are important in model-checking — for constructing counter-examples — and synthesis. Whilst our algorithms compute the winning regions for several varieties of higher-order pushdown game, we do not give a method for constructing the associated winning strategies. In the order-1 case, Cachat [127] extended Bouajjani, Esparza and Maler's [3] algorithm to compute the winning strategies of a reachability or Büchi game. His approach annotates new transitions with the information required to associate an accepting run of a configuration with a strategy that reaches the destination set. We believe a similar approach will work in the higher-order case.

In the case of order-1 parity games, we may attempt an approach based on *signatures*. However, initial efforts in this direction have proved unsuccessful.

### Order-$n$ Parity Games

Given a suitable extension of the $\mu$- and $\nu$-safety properties used to show the correctness of our order-1 parity games algorithm, we may be able to extend the approach to higher-order games. However, this does not seem to be technically trivial since the nested automata become in some sense detached from the game structure. However, the sets $\widetilde{\mathcal{G}}$ used in the higher-order reachability algorithm have the flavour of an order-$(n-1)$ pushdown system. Using this observation we may be able to perform the full generalisation.

### Implementation

In this thesis we have extended the theory behind successful order-1 model-checking tools such as Moped [123, 66] to the case of higher-order pushdown systems. A natural next step is to attempt an implementation of these algorithms.

Our algorithms have an $n$-EXPTIME complexity for an order-$n$ pushdown system, which is optimal. Even though we expect $n$ to be low in most cases, the complexity is still high. A full implementation will require significant optimisation through the use of (for example) CEGAR, BDDs, SAT solving and SMT solving.

There are a number of candidate input languages for our tool, each suited to different applications. For example, we may use recursion schemes — which mimic the structure of functional programs — or an extension of Moped's input language. The choice of input language will require research into the features that can be supported by higher-order pushdown systems.

### Collapsible Pushdown Systems

The ($\varepsilon$-closure) of the graphs generated by *collapsible pushdown systems* (CPDSs) strictly contain those generated by higher-order pushdown systems [76]. In order to capture the full expressibility of higher-order recursion schemes we may extend our techniques to collapsible pushdown systems. This will require the development of new algorithms if we are to maintain the global model-checking paradigm.

The first question to be addressed is the representation of sets of collapsible pushdown configurations. The primary difficulty in this task is the representation of the collapse links. Because this information will be encoded implicitly using a notion of regularity based on

that of Carayol (see Section 3.4), we expect a generalisation of Carayol's approach will be preferred.

Following this work we may search for generalisations of CPDSs which define a larger class of structures with decidable MSO properties.

#### Concurrency

A number of concurrent extensions of pushdown systems have been discussed in the literature. Some of these systems have decidable reachability problems [7, 112] whilst others do not [6, 4]. In the cases where reachability is undecidable, abstraction techniques such as *finite-chain* abstraction [5] or *context-bounded* model-checking [122] have provided verification algorithms.

As an extension of our work, we may explore the decidability of model-checking problems over higher-order pushdown systems equipped with concurrent constructs. Since the techniques used to show the decidability of reachability for dynamic networks of pushdown systems [6] are based on the reachability algorithm due to Bouajjani *et al.* [3] on which our work is based, it is likely that the algorithms will generalise — using our reachability results — to the higher-order case. However, because these techniques may be complex, it may be beneficial to attempt to find more elegant algorithms which may include the use of abstraction.

#### Winning Conditions of High Borel Complexity

In Section 2.6.4 we discussed a number of model-checking results for order-1 pushdown games with winning conditions of high Borel complexity. We may wish to investigate whether similar model-checking results can be obtained in the higher-order case. For example, Cachat's $\Sigma_3$ winning condition requires an extension of the reachability algorithm that provided the basis for our higher-order algorithm. A similar extension may be successful for order-$n$ pushdown systems.

#### Lower Bounds for Model-Checking Temporal Logics

In Section 6.1 we observed that a tight lower bound for LTL model-checking over higher-order pushdown systems is unknown. It would be interesting to investigate the complexity of model checking for temporal logics such as LTL, CTL and CTL* and their respective fragments.

## 7.3 Summary

We have presented a summary of the contributions made during this thesis. These include saturation based global model-checking algorithms for order-1 parity games and higher-order pushdown games with reachability conditions. We have also discussed several avenues of further research.

# Index


1-stores, 24
$B_l^a$, 105
$Norm_{k+1}$, 76
$T_{\widetilde{\mathcal{G}}_l^j}$, 118
$T_{\mathcal{D}}$, 122
$X_l^a$, 105
$\lesssim$, 132
$\mu$-calculus, 13
$\mu$TL, 11
$\omega$-regular expressions, 10
$\overline{S}$-complete, 89
$\overline{S}$-sound, 88
$exp_l$, 144
$n$-store automaton, 104
$n$-store multi-automaton, 104
$n$-stores, 46
Éloïse, 16
1-store automaton, 104

Abelard, 16
Abstract pushdown process, 99
Alternating Büchi automaton, 15
Alternating multi-automaton, 67
Alternating order-1 pushdown system, 25
Asynchronous dynamic pushdown networks, 23
Automaton with oracles, 100

Büchi, 17
Büchi automaton, 14
Basic parallel processes, 22
Basic process algebra, 19
BDDs, 177
Borel hierarchy, 17
Bounded idempotent semi-ring, 38
Branching time, 10

C♯, 44
C-regular, 75
CaRet, 41
Caucal hierarchy, 49
CEGAR, 177
Collapse, 22
Collapsible pushdown systems, 58
Communication-free, 22
Commutative abstraction, 24
Completeness preservation, 90
Conditional games, 96
Configuration, 24
Constrained dynamic pushdown networks, 23
Context-bounded model-checking, 24
Context-free, 22
Context-free higher-order pushdown systems, 72
Context-sensitive, 19
Coordination languages, 45
CTL, 12
CTL*, 13

Derived, 123
Dynamic concurrent pushdown systems, 23
Dynamic pushdown networks, 23

Exploration, 31
Extended WPDSs, 37

F♯, 44
Finite state systems, 8
Finite-chain abstraction, 24
First occurrence abstraction, 24
Fullpath, 8

Game, 16
Global model-checking, 2
Graph automaton, 98

Higher-order model-checking, 44
Higher-order pushdown automata, 22
Higher-order pushdown systems, 22
Higher-order recursion scheme, 54

Homogeneous, 54

Indexed languages, 45
Infinite state systems, 19
Information-based access control, 34
Inherited, 123
Inter-procedural data-flow analysis, 34
Inverse rational mapping, 48

Kripke structure, 8

Label bit-vectors, 24
Level 1 nested store automaton, 72
Linear time, 10
Linear weak ABA, 16
LISP, 44
Local model-checking, 2
LTL, 10

Microsoft, 45
Modular strategies, 36
MSO, 13
Multi-automaton, 64
Multi-set pushdown systems, 23

NASA, 45
Natural language processing, 45

OCaml, 44
Order-1 pushdown automaton, 25
Order-1 pushdown Büchi game, 67
Order-1 pushdown parity game, 28, 80
Order-1 pushdown system, 25
Order-$n$ APDS, 46
Order-$n$ CPDS, 58
Order-$n$ PDA, 48
Order-$n$ PDS, 46
Order-$n$ pushdown Büchi game, 168

PAD, 23
PAN, 23
Panic automata, 58
Parity, 17
Path, 8
Prefix-recognisable systems, 22
Process algebra, 23
Process rewrite systems, 23
Pushdown automata, 19
Pushdown automata with links, 58
Pushdown reachability game, 159
Pushdown systems, 22
Pushdown systems with checkpoints, 34

Rational mapping, 48
Reachability, 17
Recursion schemes, 22
Recursive state machines, 35
Regular stack properties, 33
Regular store languages, 72
Run, 8

Safe, 56
Safety, 22
SAT solving, 177
Saturation, 63
Simple goal set, 168
SMT solving, 177
Soundness preservation, 90
SPIN, 45
SPKI/SDSI framework, 38
Stack unboundedness, 31
Stair automata, 40
Star-free languages, 10
Synchronisation-sensitive, 19
Synchronised PA, 23

Tree, 9
Two-way alternating parity tree automaton, 97
Type-$l$, 84

Unfolding, 49
Unsafe, 56

Visibly pushdown automaton, 40
VP-$\mu$, 42

Weak ABA, 16
Weighted pushdown system, 38
Winning region, 159
Winning strategies, 177

# Bibliography


[1] A. Arnold and D. Niwiński. *Rudiments of μ-Calculus.* Elsevier, Amsterdam, The Netherlands, 2001.

[2] A. Bouajjani and A. Meyer. Symbolic Reachability Analysis of Higher-Order Context-Free Processes. In *Proc. 24rd Conf. on Found. of Software Technology and Theoretical Computer Science (FSTTCS'04)*, volume 3328 of *Lecture Notes in Computer Science*, Madras, India, December 2004. Springer Pub.

[3] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *International Conference on Concurrency Theory*, pages 135–150, 1997.

[4] A. Bouajjani, J. Esparza, S. Schwoon, and J. Strejcek. Reachability analysis of multithreaded software with asynchronous communication. In *FSTTCS*, pages 348–359, 2005.

[5] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. *SIGPLAN Not.*, 38(1):62–73, 2003.

[6] A. Bouajjani, J. Esparza, and T. Touili. Reachability analysis of synchronized PA-systems. In *Proceedings of Infinity 2004*, 2004.

[7] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. *CONCUR 2005 - Concurrency Theory*, pages 473–487, 2005.

[8] A. Bouajjani and O. Maler. Reachability analysis of pushdown automata, 1996. Technical Report MIP-9614, Faculty of Mathematics and Computer Science, University of Passau. In *INFINITY'96*.

[9] A. Bouajjani and P. Habermehl. Constrained properties, semilinear systems, and petri nets. In *CONCUR '96: Proceedings of the 7th International Conference on Concurrency Theory*, pages 481–497, London, UK, 1996. Springer-Verlag.

[10] A. Bouquet, O. Serre, and I. Walukiewicz. Pushdown games with unboundedness and regular conditions. In *FSTTCS'03*, volume 2914 of *lncs*, pages 88–99. Springer-Verlag, 2003.

[11] A. Carayol. Regular sets of higher-order pushdown stacks. In *MFCS*, pages 168–179, 2005.

[12] A. Carayol, M. Hague, A. Meyer, C.-H. L. Ong, and O. Serre. Winning regions of higher-order pushdown games. In *Logic in Computer Science*, pages 193–204. IEEE Computer Society, 2008.

[13] A. Carayol and S. Wöhrle. The caucal hierarchy of infinite graphs in terms of logic and higher-order pushdown automata. In *FSTTCS*, pages 112–123, 2003.

[14] A. Cataldo, E. Cheong, T. H. Feng, E. A. Lee, and A. Mihal. A formalism for higher-order composition languages that satisfies the church-rosser property. Technical Report 48, EECS Dept., University of California, Berkeley, December 2006.

[15] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *Proc. 2nd Int. Workshop on Verification of Infinite State Systems (INFINITY'97), Bologna, Italy, July 11–12, 1997*, volume 9 of *Electronic Notes in Theor. Comp. Sci.* Elsevier, 1997.

[16] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

[17] A. Kechris. *Classical Descriptive Set Theory (Graduate Texts in Mathematics)*. Springer, January 1995.

[18] A. Lal and T. W. Reps. Improving pushdown system model checking. In *CAV*, pages 343–357, 2006.

[19] A. Lal, T. W. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *CAV*, pages 434–448, 2005.

[20] A. N. Maslov. Multilevel stack automata. *Problems of Information Transmission*, 15:1170–1174, 1976.

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

[22] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[23] A. Seth. An alternative construction in symbolic reachability analysis of second order pushdown systems. In *Workshop on Reachability Problems*, 2007.

[24] A. V. Aho. Indexed grammars — an extension of context-free grammars. *J. ACM*, 15(4):647–671, 1968.

[25] B. Banieqbal and H. Barringer. Temporal logic with fixed points. In *Temporal Logic in Specification*, pages 62–74, 1987.

[26] B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond safety. In *CAV*, pages 415–418, 2006.

[27] B. Courcelle. The monadic second-order logic of graphs ix: Machines and their behaviours. *Theoretical Computer Science*, 151:125–162(38), 1995.

[28] B. Courcelle and I. Walukiewicz. Monadic second-order logic, graph coverings and unfoldings of transitions systems. In *Computer Science Logic*, pages 53–75, 1995.

[29] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS '06: Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science*, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[30] C. Löding, P. Madhusudan, and O. Serre. Visibly pushdown games. In *Proceedings of FSTTCS'04*, volume 3328 of *LNCS*, pages 408–420. Springer-Verlag, 2004.

[31] C. Löding and W. Thomas. Alternating automata and logics over infinite words. In *TCS '00: Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics*, pages 521–535, London, UK, 2000. Springer-Verlag.

[32] C. Stirling. Bisimulation, model checking and other games, June 1997. Notes for a Mathfit instructional meeting on games and computation, held in Edinburgh, Scotland.

[33] C. Stirling. Bisimulation, modal logic and model checking games. *Logic Journal of the IGPL*, 7(1):103–124, 1999.

[34] D. Box and T. Pattison. *Essential .NET: The Common Language Runtime*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[35] D. Caucal. On infinite transition graphs having a decidable monadic theory. In F. Meyer auf der Heide and B. Monien, editors, *Proceedings of the 23th International Colloquium on Automata, Languages and Programming (ICALP'96)*, volume 1099, pages 194–205, Berlin-Heidelberg-New York, 1996. Springer.

[36] D. Caucal. On infinite terms having a decidable monadic theory. In *Proc. MFCS'02*, pages 165–176, 2002. LNCS 2420.

[37] D. Caucal and R. Monfort. On the transition graphs of automata and grammars. In *WG '90: Proceedings of the 16rd International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 311–337, London, UK, 1991. Springer-Verlag.

[38] D. E. Muller, A. Saoudi, and P. E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In Y. Gurevich, editor, *Proceedings of the Third Annual IEEE Symp. on Logic in Computer Science, LICS 1988*, pages 422–427. IEEE Computer Society Press, July 1988.

[39] D. E. Muller, A. Saoudi, and P. E. Schupp. Alternating automata, the weak monadic theory of trees and its complexity. *Theor. Comput. Sci.*, 97(2):233–244, 1992.

[40] D. E. Muller and P. E. Schupp. The theory of ends, pushdown automata, and second-order logic. *Theor. Comput. Sci.*, 37:51–75, 1985.

[41] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[42] D. Janin and I. Walukiewicz. On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic. In *CONCUR '96: Proceedings of the 7th International Conference on Concurrency Theory*, pages 263–277, London, UK, 1996. Springer-Verlag.

[43] E. A. Emerson and C.-L. Lei. Modalities for model checking: branching time logic strikes back. *Sci. Comput. Program.*, 8(3):275–306, 1987.

[44] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model-checking for fragments of mu-calculus. In *5th International Computer-Aided Verification Conference*, 1993.

[45] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.

[46] E. M. Clark and E. A. Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic. In *Proceedings IBM Workshop on Logics of Programs*, volume 131, pages 52–71. Lecture Notes in Computer Science, Springer, 1981.

[47] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[48] E. W. Mayr. An algorithm for the general petri net reachability problem. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 238–246, New York, NY, USA, 1981. ACM Press.

[49] G. Gazdar. Applicability of indexed grammars to natural languages. *Natural Language Parsing and Linguistic Theories*, pages 69–94, 1988.

[50] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.

[51] G. S. Rohde. *Alternating automata and the temporal logic of ordinals*. PhD thesis, University of Illinois, 1997. Adviser-Paul E. Schupp.

[52] H. Gimbert. Parity and exploration games on infinite graphs. In *Proceedings of CSL'04*, volume 3210 of *LNCS*, pages 56–70. Springer-Verlag, 2004.

[53] I. Walukiewicz. Pushdown processes: Games and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 62–74, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[54] I. Walukiewicz. Model checking ctl properties of pushdown systems. In *FST TCS 2000: Proceedings of the 20th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 127–138, London, UK, 2000. Springer-Verlag.

[55] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[56] J. A. Brzozowski and E. L. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theor. Comput. Sci.*, 10:19–35, 1980.

[57] J. A. Cataldo. *The Power of Higher-Order Composition Languages in System Design*. PhD thesis, University of California, Berkeley, December 2006.

[58] J. Bradfield and C. Stirling. Modal logics and mu-calculi: an introduction, 2001.

[59] J. Engelfriet. Iterated pushdown automata and complexity classes. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 365–373, New York, NY, USA, 1983. ACM Press.

[60] J. Esparza. On the decidability of model checking for several mu-calculi and petri nets. In *CAAP: Colloquium on Trees in Algebra and Programming*. LNCS, Springer-Verlag, 1994.

[61] J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Inf.*, 34(2):85–107, 1997.

[62] J. Esparza. Petri nets, commutative context-free grammars, and basic parallel processes. *Fundam. Inf.*, 31(1):13–25, 1997.

[63] J. Esparza and A. Kiehn. On the model checking problem for branching time logics and basic parallel processes. In *Proceedings of the 7th International Conference on Computer Aided Verification*, pages 353–366, London, UK, 1995. Springer-Verlag.

[64] J. Esparza, A. Kučera, and S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. In *Proc. of TACS 2001*, number 2215 in Lecture Notes in Computer Science, pages 306–339, 2001.

[65] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification*, pages 232–247, 2000.

[66] J. Esparza, S. Kiefer, and S. Schwoon. Abstraction refinement with craig interpolation and symbolic pushdown systems. In *TACAS*, pages 489–503, 2006.

[67] J. Harrison. Formal verification at intel. In *LICS '03: Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, page 45, Washington, DC, USA, 2003. IEEE Computer Society.

[68] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: $10^{20}$ States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[69] K. Aehlig, J. G. de Miranda, and C.-H. L. Ong. Safety is not a restriction at level 2 for string languages. In *FoSSaCS*, pages 490–504, 2005.

[70] K. Bhargavan, C. Fournet, and Andrew D. Gordon. Verified reference implementations of ws-security protocols. In *WS-FM*, pages 88–106, 2006.

[71] K. Etessami. Analysis of recursive game graphs using data flow equations. In *VMCAI*, pages 282–296, 2004.

[72] K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. L. White. Formal analysis of the remote agent before and after flight. In *5th NASA Langley Formal Methods Workshop, Williamsburg, Virginia*, 2000.

[73] K. Sen and M. Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *CAV*, pages 300–314, 2006.

[74] L. Gong. *Inside Java 2 platform security architecture, API design, and implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[75] M. Benedikt, P. Godefroid, and T. W. Reps. Model checking of unrestricted hierarchical state machines. In *ICALP '01: Proceedings of the 28th International Colloquium on Automata, Languages and Programming,*, pages 652–666, London, UK, 2001. Springer-Verlag.

[76] M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *Logic in Computer Science*, pages 452–461. IEEE Computer Society, 2008.

[77] M. Hague and C.-H. L. Ong. Symbolic backwards-reachability analysis for higher-order pushdown systems. In *FoSSaCS*, pages 213–227, 2007.

[78] M. Hague and C.-H. L. Ong. Symbolic backwards-reachability analysis for higher-order pushdown systems. *Logical Methods in Computer Science*, 4(4), 2008.

[79] M. Lange. Weak automata for the linear time $\mu$-calculus. In R. Cousot, editor, *Proc. 6th Int. Conf. on Verification, Model Checking and Abstract Interpretation, VMCAI'05*, volume 3385 of *LNCS*, pages 267–281, 2005.

[80] M. Pistoia, A. Banerjee, and D. A. Naumann. Beyond stack inspection: A unified access-control and information-flow security model. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 149–163, Washington, DC, USA, 2007. IEEE Computer Society.

[81] M. Y. Vardi. A temporal fixpoint calculus. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 250–259, New York, NY, USA, 1988. ACM Press.

[82] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff Higher Order Workshop*, pages 238–266, 1995.

[83] M. Y. Vardi. Branching vs. linear time: Final showdown. In *TACAS*, pages 1–22, 2001.

[84] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 15 1994.

[85] N. Piterman. *Verification of Infinite State Systems.* PhD thesis, Weizmann Institute of Science, 2004.

[86] N. Piterman and M. Y. Vardi. Global model-checking of infinite-state systems. In *CAV*, pages 387–400, 2004.

[87] O. Burkart. Model checking rationally restricted right closures of recognizable graphs. *Electr. Notes Theor. Comput. Sci.*, 9, 1997.

[88] O. Burkart, D. Caucal, and B. Steffen. Bisimulation collapse and the process taxonomy. In *CONCUR '96: Proceedings of the 7th International Conference on Concurrency Theory*, pages 247–262, London, UK, 1996. Springer-Verlag.

[89] O. Kupferman and M. Y. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV'00)*, volume 1855. Springer, 2000.

[90] O. Kupferman, N. Piterman, and M. Vardi. Model checking linear properties of prefix-recognizable systems. In *Proc. 14th CAV*, pages 371–385, 2002.

[91] O. Lichtenstein, A. Pnueli, and L. D. Zuck. The glory of the past. In *Proceedings of the Conference on Logic of Programs*, pages 196–218, London, UK, 1985. Springer-Verlag.

[92] O. Serre. Note on winning positions on pushdown games with $\omega$-regular conditions. *Information Processing Letters*, 85:285–291, 2003.

[93] O. Serre. Games with winning conditions of high Borel complexity. In *Proceedings of ICALP'04*, volume 3142 of *LNCS*, pages 1150–1162. Springer-Verlag, 2004.

[94] P. Müller and J. N. Ruskiewicz. A modular verification methodology for C# delegates. In *Rigorous Methods for Software Construction and Analysis*, 2007.

[95] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99, 1983.

[96] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 207–220, London, UK, 2001. Springer-Verlag.

[97] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS*, 2004.

[98] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.

[99] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst.*, 23(3):273–303, 2001.

[100] R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211, New York, NY, USA, 2004. ACM Press.

[101] R. Alur, S. Chaudhuri, and P. Madhusudan. Visibly pushdown tree languages. `http://www.cis.upenn.edu/~swarat/pubs/vptl.ps`.

[102] R. Alur, S. Chaudhuri, and P. Madhusudan. A fixpoint calculus for local and global program flows. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 153–165, New York, NY, USA, 2006. ACM Press.

[103] R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. In *ICAL '99: Proceedings of the 26th International Colloquium on Automata, Languages and Programming*, pages 169–178, London, UK, 1999. Springer-Verlag.

[104] R. Alur, S. La Torre, and P. Madhusudan. Modular strategies for infinite games on recursive graphs. In *CAV*, pages 67–79, 2003.

[105] R. Alur, S. La Torre, and P. Madhusudan. Modular strategies for recursive game graphs. In *TACAS*, pages 363–378, 2003.

[106] R. Büchi. Regular canonical systems. *Archiv fur Math. Logik und Grundlagenforschung 6*, pages 91–111, 1964.

[107] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[108] R. Mayr. Weak bisimulation and model checking for basic parallel processes. In *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 88–99, London, UK, 1996. Springer-Verlag.

[109] R. Mayr. Combining petri nets and pa-processes. In *TACS '97: Proceedings of the Third International Symposium on Theoretical Aspects of Computer Software*, pages 547–561, London, UK, 1997. Springer-Verlag.

[110] R. Mayr. Model checking pa-processes. In *CONCUR '97: Proceedings of the 8th International Conference on Concurrency Theory*, pages 332–346, London, UK, 1997. Springer-Verlag.

[111] R. Mayr. Tableau methods for PA-processes. In *Analytic Tableaux and Related Methods*, pages 276–290, 1997.

[112] R. Mayr. *Dedicability and Complexity of Model Checking Problems for Infinite-State Systems*. PhD thesis, TU-München, 1998.

[113] R. Mayr. Strict lower bounds for model checking BPA. In *Electronic Notes in Theoretical Computer Science (ENTCS)*, volume 18, 1998.

[114] R. Mayr. Process rewrite systems. *Inf. Comput.*, 156(1-2):264–286, 2000.

[115] R. Mc Naughton and S. Papert. *Counter-Free Automata*. M.I.T. Press, Cambridge, Mass., 1971.

[116] R. P. Kurshan. Formal verification in a commercial setting. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 258–262, New York, NY, USA, 1997. ACMPress.

[117] R. S. Streett and E. A. Emerson. An automata theoretic decision procedure for the propositional mu-calculus. *Inf. Comput.*, 81(3):249–264, 1989.

[118] R. V. Book and F. Otto. *String-Rewriting Systems*. Springer-Verlag, 1993.

[119] S. Christensen. *Decidability and Decomposition in process Algebras*. PhD thesis, University of Edinburgh, 1993.

[120] S. Göller and M. Lohrey. Infinite state model-checking of propositional dynamic logics, 2006. Technical report 2006/04 of Universität Stuttgart, Institut für Formale Methoden der Informatik, Theoretische Informatik.

[121] S. Jha and T. Reps. Analysis of SPKI/SDSI certificates using model checking. In *CSFW '02: Proceedings of the 15th IEEE workshop on Computer Security Foundations*, page 129, Washington, DC, USA, 2002. IEEE Computer Society.

[122] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, pages 93–107, 2005.

[123] S. Schwoon. *Model-checking Pushdown Systems.* PhD thesis, Technical University of Munich, 2002.

[124] S. Schwoon, S. Jha, T. Reps, and S. Stubblebine. On generalized authorization problems. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW)*, pages 202–218. IEEE Computer Society, June 2003.

[125] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 526–538, London, UK, 2002. Springer-Verlag.

[126] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, January 16–18, 2002.

[127] T. Cachat. *Games on Pushdown Graphs and Extensions.* PhD thesis, RWTH Aachen, 2003.

[128] T. Cachat. Higher order pushdown automata, the caucal hierarchy of graphs and parity games. In *ICALP*, pages 556–569, 2003.

[129] T. Cachat and I. Walukiewicz. The complexity of games on higher order pushdown automata, 2007. `http://www.citebase.org/abstract?id=oai:arXiv.org:0705.0262`.

[130] T. Cachat, J. Duparc, and W. Thomas. Solving pushdown games with a $\Sigma_3$ winning condition. In *Proceedings of the 11th Annual Conference of the European Association for Computer Science Logic (CSL'02)*, volume 2471 of *LNCS*, pages 322–336. Springer-Verlag, 2002.

[131] T. Knapik, D. Niwinski, and P. Urzyczyn. Deciding monadic theories of hyperalgebraic trees. In *TLCA*, pages 253–267, 2001.

[132] T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In *FoSSaCS '02: Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*, pages 205–222, London, UK, 2002. Springer-Verlag.

[133] T. Knapik, D. Niwinski, P. Urzyczyn, and I. Walukiewicz. Unsafe grammars and panic automata. In *ICALP*, pages 1450–1461, 2005.

[134] T. P. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security properties. In *IEEE Symposium on Security and Privacy*, pages 89–103, 1999.

[135] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.

[136] T. Wilke. CTL$^+$ is exponentially more succinct than CTL. In *Foundations of Software Technology and Theoretical Computer Science*, pages 110–121, 1999.

[137] W. Damm and A. Goerdt. An automata-theoretical characterization of the oi-hierarchy. *Inf. Control*, 71(1-2):1–32, 1986.

[138] W. Thomas. A combinatorial approach to the theory of $\omega$-automata. In *Information and Computation*, volume 48, pages 261–283, 1981.

[139] W. Thomas. Automata on infinite objects. *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 133–191, 1990.

[140] W. Thomas. Languages, automata, and logic. *Handbook of formal languages, vol. 3: beyond words*, pages 389–455, 1997.